



Version: 1.9.1

Major changes from 1.8.1 to 1.9.1

The GAMA development team is pleased to announce the release of GAMA 1.9.1

This version, while maintaining the power, stability, expressiveness and ease of use of GAMA, brings new capabilities and openings to the platform, making it even more intuitive to use by modelers and even more versatile in terms of applications.

This major release of GAMA contains many new features and fixes, including:

- **A much more fluid and powerful IDE**, offering support for all the latest technologies, from HiDPI displays to JDK 17 and Apple Silicon processors.
- **A new server mode** of GAMA, offering a clear and extensible exchange protocol, which completely revolutionizes the way to interact with the platform from R, Python or any web client.
- **Increased model exploration possibilities** thanks to new calibration and optimization methods, also directly usable in the server mode.
- **The addition of the two new data types** `field` and `image`, which make it even easier to load, analyze, visualize and produce raster data
- **A much more powerful graph manipulation** than previous versions, but still easy to couple with agents
- **A focus on urban mobility applications**, with `skill` `advanced_driving` and `pedestrian`, which make it much easier to produce realistic large-scale models.
- **The possibility to simulate physical interactions between agents** thanks to the new `skills` `static_body`, `dynamic_body` and `physical_simulation`, which

rely on the native `bullet` library.

- **New and faster display capabilities**, offering more intuitive handling of agents and organisation of display surfaces, making it easier than ever to build interactive simulations, serious games or advanced scientific visualisations.

Comparison chart

	Gama 1.8.1	Gama 1.9.1
Java and environments	Java 11 and x86 (intel architecture)	Java 17, x86 and ARM architectures (notably Apple Silicon)
Server mode	-	Headless server / connection with Python and R
Model exploration	Exhaustive sampling and calibration	Several new sampling methods (e.g. latinhypercube), sensitivity analysis (e.g. Sobol) and calibration
Physics modeling	Limited	Extended features with native bullet library and influences/forces computations
Mobility modeling	<code>moving</code> and <code>driving</code> skills	<code>moving</code> , <code>advanced_driving</code> (non normative traffic) and <code>pedestrian</code> (Social force model) skills
Raster data integration	Limited with <code>grid</code> (bad performances above 500x500)	New <code>field</code> and <code>image</code> types allow larger sizes and better performances

	Gama 1.8.1	Gama 1.9.1
Graph integration	Programmatic with fixed layout	Import / export to 6 graph file formats (e.g. .graphml, .gml) with various spatial layouts rendering

GAMA Server mode

gama-server is a new way of running GAMA experiments. It consists of an instance of gama-headless that, once launched, waits for commands sent through websockets and executes them. These commands follow a clear and extensible protocol, enabling its use in many contexts, from the definition of experiment plans in R to the design of dashboards in JavaScript. See the corresponding [wiki page](#) to setup a server instance of Gama.

Modelling improvements

field type

A new variable type (field) to support the management (import and use) of large raster geographic data. It allows in particular to:

- import large mono/multi-band rasters
- simply access / modify values of spatial grids as simply as before, but with very high performance improvement

Try out:

- Basic syntax to create and visualize fields: [Fields.gaml](#)
- Basic syntax to access/write values in fields: [Accessing Fields.gaml](#)
- Use of field to superpose information in a traffic simulation: [Traffic and Pollution.gaml](#)
- Use of field to represent flows (water): [Waterflow Field Elevation.gaml](#) Use of field to support diffusion process: Anisotropic Diffusion & Uniform Diffusion

image type

- Easier to work with images

Try out:

- Basic syntax to create an image: [Declaring Images.gaml](#)
- Basic syntax to manipulate an image: [Image Manipulation.gaml](#)
- Save snapshot of a simulation displays: [Manual Snapshot.gaml](#)

pedestrian skill

A new plugin has been integrated in GAMA that allows to simulate pedestrian movement. This plugin uses Helbing's social force model as a basis to support pedestrian walk and offers tools to reconstruct paths from an open environment and obstacles. This two new features are identified by a skill (`pedestrian`) and an operator (`generate_pedestrian_network`) respectively. You can find examples in the models below.

Try out:

- How to build the pedestrian network that agents use to manage the origin and destination of their trip in the open environment: [Generate pedestrian](#)

[paths.gaml](#)

- A comprehensive list of the parameters that makes it possible how agent avoid obstacles, in a simple ([Simple environment - walk_to.gaml](#)) and a complex ([Complex environment - walk.gaml](#)) environment

advanced_driving skill

The driving skill has been completely redesigned in order to offer a more realistic representation of driver behavior (by explicitly using the Intelligent Driver Model and Lane-change Model MOBIL) and by allowing to take into account multi-lane vehicles - this allows for example to simulate mixed traffic composed of motorcycles and cars. Besides, the behavior of drivers can be custom to represent non normative behavior, such as dangerous take-off, disrespect of signals, signs, speed limit or road direction and lanes.

Try out:

- An abstract representation of vehicles size (bus, car, motorcycle) and free use of road lanes and direction ([Drive Random.gaml](#))
- An abstract representation of vehicles managing cross section, with collision avoidance, priority, etc. ([Simple Intersection.gaml](#))
- A very small road system with stops to simulate congestion ([Following Paths.gaml](#))

Physics extension improvement

Physics plugin has been completely rewritten and allows to use native implementations of the bullet library in a redesigned framework (where physical agents can coexist with non-physical ones).

Try out:

- Interaction between static (skill `static_body`) and dynamic (skill `dynamic_body`) 3D objects ([Eroding Vulcano.gaml](#))
 - Manage 3D objects movement based on a Digital Elevation Model ([Flow on Terrain.gaml](#))
-

Experiment

Batch methods

Batch experiments have been reworked to better distinguish simulation exploration and model calibration. On the first hand, modelers should engage in simulation exploration if they want to launch many simulations across the parameter space, better understand the contribution of stochasticity and evaluate the specific contribution of given parameters to output variability. On the other hand, modelers should use calibration methods if they want to find parameters values of the models, so the simulation outputs are as close as possible to desired ones. A detailed description is provided in this [wiki page](#).

Try out:

- A walkthrough of all provided methods to explore, method `exploration`, and analyse the sensitivity of your model, including a tool to decide method `stochanalyse` or method `sobol` ([Exploration.gaml](#))
- A walkthrough of minimal way to setup calibration, including the new `PSO` algorithm ([Calibration.gaml](#))

Headless batch

We implement a way to launch Gama `batch` experiment in headless with a simple

command line, using the gama-headless.sh bash script with `-batch` option. For more information, see the related ([wiki page](#)).

Reproducibility and random number generation

- Great effort towards tracking and limiting the use of random generators outside the ones built in GAMA
 - Addition of several new random number generators
-

Displays

OpenGL improvements

Great improvements have been done on the displays and specifically on opengl ones. Key points are:

- Lot faster (2 times) on geometries
- Rendering of large-scale images, grids, fields or matrices using the new `mesh` layer, with several colouring options
- More flexibility:
 - `camera` statement to specify the dynamic movements of the camera
 - `light` statements to specify the lighting(s) of the scene
 - `rotate` statement to specify the rotation of the full screen
- Better and more accurate rendering of texts (with 3D, etc.)
- Possibility to choose between several predefined cameras, to save cameras, etc.

mesh layer

- display large rasters

layout improvements

- Allow to easily split or compose the displays
 - Possibility to define borderless displays
-

User Interface

Support of HiDPI

- HiDPI and various "display zooms" are now supported natively. Displays, text and icons scale up and down accordingly. Only issues remaining is that the text and icons can be blurry and pixelised on some configurations (Windows 10, Windows 11 with 150% zoom, etc.)

Support of dark mode

- Light and dark modes are also now supported out of the box. Preferences allow GAMA to impose its own theme or follow the one defined in the OS. A new syntax highlighting theme for dark mode is accessible from the preferences too.
-

User Interaction

Addition of wizards and dialogs

- It is now possible to open wizards and dialogs from the GAML code thanks to the `user_confirm`, `user_input_dialog`, `wizard` and `wizard_page` operators.

Try out:

- How to define a new wizard ([Wizard.gaml](#))
- detailed use of the new user_input ([User input.gaml](#))

Addition of events

- new events can be defined as `display` layers: `#arrow_down`, `#arrow_up`,
`#arrow_left`, `#arrow_right`, `#escape`, `#tab`, `#enter`, `#page_up`, `#page_down`

Clipboard

- the clipboard can be written and read using the `copy_to_clipboard(value)` and
`copy_from_clipboard(type)` operators
-

Advanced programming usages

Additions to GAML

- `on_change`: facet can be added to attributes and parameters to trigger any behaviour in response to a change of value. Particularly useful for defining

interactive parameters.

- `abort` statement can be defined in any agent (incl. `global` and `experiment`) and executed just before the agent is disposed of.

thread skill

The new thread skill allows to run actions in a specific thread. In particular, this skill is intended to define the minimal set of behaviours required for agents that are able to run an action in a thread.

File manipulations: `copy`, `zip`, `delete`, `save` improvements

- One can now completely manipulate files directly in the gama models with dedicated `copy_file`, `delete_file`, `rename_file` (which can be used to move a file), `zip` and `unzip` operators.
- `save` accepts more file formats and provides a hook for developers to develop `ISaveDelegates`

network skill improvements

To increase the integration between Gama and other applications we improved a lot the network capabilities:

- The communication with *web-services* is now easier with the possibility to execute post/get/update/delete HTTP requests directly in gaml with extensions of the `send` action of the networking skill, as described in the `HTTP POST.gaml` and `HTTP GET.gaml` of the `Plugin models` library.
- Adding support for the websocket protocol in the `network` skill
- General work on the network skill with communication outside of Gama in mind

Graph improvements

Shortest paths

Integration of new algorithms for computing shortest paths in graphs.

- BidirectionalDijkstra: default one - ensure to find the best shortest path - compute one shortest path at a time:
<https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html>
- DeltaStepping: ensure to find the best shortest path - compute one shortest path at a time: The delta-stepping algorithm is described in the paper: U. Meyer, P. Sanders, \$\Delta\$-stepping: a parallelizable shortest path algorithm, Journal of Algorithms, Volume 49, Issue 1, 2003, Pages 114-152, ISSN 0196-6774
- CHBidirectionalDijkstra: ensure to find the best shortest path - compute one shortest path at a time. Based on precomputations (first call of the algorithm). Implementation of the hierarchical query algorithm based on the bidirectional Dijkstra search. The query algorithm is originally described the article: Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In Proceedings of the 7th international conference on Experimental algorithms (WEA'08), Catherine C. McGeoch (Ed.). Springer-Verlag, Berlin, Heidelberg, 319-333
- TransitNodeRouting: ensure to find the best shortest path - compute one shortest path at a time. Based on precomputations (first call of the algorithm). The algorithm is designed to operate on sparse graphs with low average outdegree. the algorithm is originally described the article: Arz, Julian & Luxen, Dennis & Sanders, Peter. (2013). Transit Node Routing Reconsidered. 7933. 10.1007/978-3-642-38527-8_7.

Input/output

You can now load / save your graph into dedicated file format such as .gml, .dot or .gefx to build your graph.

Try out:

- Load agents from a graph file ([Graph Agents Importation.gaml](#))
- Load the entire graph from files ([Graph Importation.gaml](#))
- Save graphs into dedicated files format ([Save Graphs.gaml](#))

Layout

Non spatial graph can be rendered using operators to locate nodes on a circle, as a grid lattice or considering connection as forces.

OS and computing environments

GAMA 1.9.1 has been tested on:

- Windows 10 and 11 on Intel processors
- MacOS Monterey, Ventura on Intel & Apple Silicon computers
- Ubuntu 20.04 and 22.04 on Intel processors

Note that this version drops the support for 32 bits architectures.

Support of JDK 17+

Gama 1.9.1 brings compatibility with JDK17+ and should remain compatible for the following JDK versions.

Support of ARM processors

A specific version of GAMA is now built for Apple Silicon processors on macOS. Even if no specific version is produced for the ARM version of Windows, reports show that it works well in emulated mode.

New installers for Windows, Mac (brew) and Linux (aur, deb)

Gama 1.9.1 comes with a dedicated installer for every platform, so it's easier for newcomers to get it working. In addition, the macOS version is now fully signed. Linux and macOS users can also benefit from CLI installers.

New versions of native libraries: SWT, JTS, GeoTools, bullet, JOGL, JGraphT

All the major libraries on which GAMA is relying have been bumped to their latest versions, except GeoTools (version 25) and JGraphT (version 1.5.1).

Changes that can impact models

🔴 Errors 🔴: concepts that cannot be used anymore

- `gama.pref_lib_r`, `gama.pref_lib_spatialite`,
`gama.pref_optimize_agent_memory`, `gama.pref_display_triangulator` have
been removed
- In experiment, the method statement `exhaustive` and `explicit` does not exist
anymore. Use `exploration` instead, see the related documentation on `batch`.

- the `material` type (and the corresponding `material:` facet in `draw:`) does not exist anymore and has not been replaced.
- the built-in `equation` types (`SIR`, etc.) do not exist anymore and have not been replaced.
- `field` cannot be used anymore as a species or variable name.
- `image` cannot be used anymore as a species or variable name.
- `to_list` cannot be used anymore as a species or variable name.

🔴 Errors 🔴: concepts that need to be written differently

- `timeStamp()` in `SQLSKILL` does not exist anymore. Use `machine_time` instead.
- `dem(...)` operators do not exist anymore. Use a combination of `field` and `mesh` layer to load and draw a digital elevation model
- `event ['k']` should be rewritten as `event 'k'`.
- `generate_complete_graph`, `generate_barabasi_albert`, `generate_watts_strogatz`, and `as_distance_graph` now take different arguments. Please refer to their documentation.
- `load_graph_from_file` has been removed and replaced by the use of the corresponding graph file types (`graphml_file`, etc.)
- `simplex_generator` has been removed and replaced by `generate_terrain`

🟡 Warnings 🟡:

- `grid + lines:` is deprecated and replaced by `border:`
- `save + type:` is deprecated and replaced by `format:`
- `display + draw_env:` is deprecated and replaced by `axes:`
- `display + synchronized:` is deprecated. `synchronized:` should now be defined on `output:`

- `display + camera_pos:` is deprecated. Should be replaced by `location:` defined on a `camera` statement inside the `display`
 - `display + camera_interaction:` is deprecated. Should be replaced by `locked:` defined on a `camera` statement inside the `display`
 - `display + camera_up_vector:` is deprecated. Not used anymore.
 - `display + camera_look_pos:` is deprecated. Should be replaced by `target:` defined on a `camera` statement inside the `display`
 - `display + focus:` is deprecated. Should be replaced by `target:` defined on a `camera` statement inside the `display`
 - `display + ambient_light:` is deprecated. Should be replaced by `intensity:` defined on a `light #ambient` statement inside the `display`
 - `light + position:` is deprecated and replaced by `location:`
 - `light + update:` is deprecated and replaced by `dynamic:`
 - `light + color:` is deprecated and replaced by `intensity:`
 - `light + name:` now takes a `string` and not an `int`
 - `light + draw_light:` is deprecated and replaced by `show:`
 - `light + type:` now takes a `string` among `#spot`, `#point` or `#direction`
 - `user_input` is deprecated and should be replaced by `user_input_dialog`
 - `draw + empty:` is deprecated and replaced by `wireframe:`
 - `image (layer) + file:` is deprecated and replaced by the direct use of the file name as the default facet
 - `event` now takes a string for its default facet (preferably the defined constants like `#mouse_move`, `#left_arrow`, etc.)
 - `event + action:` is deprecated as the definition of the action should directly follow the statement definition
 - the `with_optimizer_type` operator is deprecated and replaced by `with_shortestpath_algorithm`
-

Preferences

The description of all preferences can be found at this [page](#). A number of new preferences have been added to cover existing or new aspects of the platform. They are summarised below.

New preferences

Interface tab

- *Startup* Remember Gama windows sizes
- *Startup* Several prompts related to the use of workspaces
- *Startup* Setup a model to run at start

Editors tab

- *Edition* More options (3) for automatic typing
- *Edition* Turns experiment buttons into a drop down list
- *Syntax* Coloring according to Gama theme (light|dark)

Execution tab

- (New) *Parameters* Customize parameter view
- *Parallelism* Use all available threads in batch mode

Display tab

- *Chart preferences* Choose resolution of charts
- (Removed) *Advanced*
- *OpenGL* Limit the number of frames

- *OpenGL* Sensitivity of keyboard/mouse/trackpad
- *OpenGL* Ambiant light intensity
- *OpenGL* Default camera orientation

Data and Operator

- *Random Number Generator* Display RNG in parameter view
- (Removed) *Optimization* Many options have been removed to enforce reproducibility

(New) Experimental

This tab holds experimental preferences that should be used with care

Setting and sharing preferences

Gama 1.9.1 brings new options for setting preferences and sharing them among models.

Passing preferences to GAMA at startup

Modellers running the headless or gui versions of GAMA can now pass preferences to the executable using arguments (either in the headless script or in the `Gama.ini` file). The syntax is `-Dpref_name=value` (for instance `-Dpref_display_synchronized=true` to synchronise displays, including snapshots of headless GAMA, with the simulation).

Global or workspace scopes

The default behaviour of GAMA makes sharing preferences between workspaces and models easy, since they are global to the user account. In some instances, however, it can be necessary to restrict them to a local scope (i.e. a workspace). In that case, launching GAMA with the `-Duse_global_preference_store=false` will make it save

its preferences in the current workspace and not globally anymore.

Bug fixes

You can also check the complete list of the closed issues on the [github repository](#). Keep in mind that this list is incomplete as a lot of problems were solved without being linked to any issue on github (via the mailing list or internally for example).

Added models

The library of models has undergone some changes. Besides making sure all the models compile and run fine under the new version of GAMA, it also brings some new models, which are listed below:

Usage of the **pedestrian** skill

- [miat.gaml.extensions.pedestrian/models/Pedestrian Skill/models/Complex environment - walk.gaml](#)
 - [miat.gaml.extensions.pedestrian/models/Pedestrian Skill/models/Generate pedestrian paths.gaml](#)
 - [miat.gaml.extensions.pedestrian/models/Pedestrian Skill/models/Simple environment - walk_to.gaml](#)
-

New **graph** capabilities

- [msi.gama.models/models/Data/Data Exportation/models/Save Graphs.gaml](#)

- [msi.gama.models/models/Data/Data Importation/models/Graph Agents Importation.gaml](#)
 - [msi.gama.models/models/Data/Data Importation/models/Graph Importation.gaml](#)
 - [msi.gama.models/models/Modeling/Spatial Topology/Graphs/models/Clustering.gaml](#)
-

Utilities

- [msi.gama.models/models/Data/Utils/models/FileUtils.gaml](#)
 - [msi.gama.models/models/Data/Utils/models/TestWebAddress.gaml](#)
 - [msi.gama.models/models/Data/Utils/models/ZipUnzip.gaml](#)
-

Elements of GAML syntax

- [msi.gama.models/models/GAML Syntax/Abort statement/Abort.gaml](#)
 - [msi.gama.models/models/GAML Syntax/Data Types And Structures/Fields.gaml](#)
 - [msi.gama.models/models/GAML Syntax/Loop And Iterations/Break and Continue.gaml](#)
 - [msi.gama.models/models/GAML Syntax/System/Clipboard.gaml](#)
 - [msi.gama.models/models/GAML Syntax/System/Elements of Syntax.gaml](#)
 - [msi.gama.models/models/GAML Syntax/System/RunThread.gaml](#)
 - [msi.gama.models/models/GAML Syntax/Variables/Declaration of Parameters.gaml](#)
 - [msi.gama.models/models/GAML Syntax/Variables/Notifying Variables.gaml](#)
-

New batch capabilities

- [msi.gama.models/models/Model Exploration/Batch Simulation/Calibration.gaml](#)
 - [msi.gama.models/models/Model Exploration/Batch Simulation/Exploration.gaml](#)
-

Toy models

- [msi.gama.models/models/Toy Models/Art/Gama 1.9/models/GAMA 1.9.gaml](#)
 - [msi.gama.models/models/Toy Models/Games/Snake.gaml](#)
 - [msi.gama.models/models/Toy Models/K Nearest Neighbours/models/knn.gaml](#)
-

Declaration and usage of field

- [msi.gama.models/models/Modeling/Spatial Topology/Fields/Accessing Fields.gaml](#)
- [msi.gama.models/models/Toy Models/Waterflow/models/Waterflow Field Elevation.gaml](#)
- [msi.gama.models/models/Toy Models/Traffic/models/Traffic and Pollution.gaml](#)
- [ummisco.gaml.extensions.maths/models/Diffusion Statement/models/Anisotropic Diffusion \(Simple, Field\).gaml](#)
- [ummisco.gaml.extensions.maths/models/Diffusion Statement/models/Uniform Diffusion \(Field\).gaml](#)
- [msi.gama.models/models/Visualization and User Interaction/Visualization/Building Heatmap.gaml](#)
- [msi.gama.models/models/Visualization and User Interaction/Visualization/DEM Generator.gaml](#)
- [msi.gama.models/models/Visualization and User Interaction/Visualization/](#)

Plettes and Gradients.gaml

- [msi.gama.models/models/Visualization and User Interaction/Visualization/Worm Heatmap.gaml](#)
-

New user interaction modalities

- [msi.gama.models/models/Visualization and User Interaction/GUI Design/Parameters and Commands.gaml](#)
 - [msi.gama.models/models/Visualization and User Interaction/User Interaction/models/Confirm Dialog.gaml](#)
 - [msi.gama.models/models/Visualization and User Interaction/User Interaction/models/User input.gaml](#)
 - [msi.gama.models/models/Visualization and User Interaction/User Interaction/models/Wizard.gaml](#)
-

New camera and light definitions

- [msi.gama.models/models/Visualization and User Interaction/Visualization/3D Visualization/models/Camera Definitions.gaml](#)
 - [msi.gama.models/models/Visualization and User Interaction/Visualization/3D Visualization/models/Camera Shared Zoom.gaml](#)
 - [msi.gama.models/models/Visualization and User Interaction/Visualization/3D Visualization/models/Specular Effects.gaml](#)
-

Physics engine demonstrations

- [simtools.gaml.extensions.physics/models/Physics Engine/models/Eroding](#)

Vulcano.gaml

- simtools.gaml.extensions.physics/models/Physics Engine/models/Flow on Terrain.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Perfect Gas Chamber.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Play Pool.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Stairs.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Testing Restitution.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Testing Steps.gaml
 - simtools.gaml.extensions.physics/models/Physics Engine/models/Tricky Fountain.gaml
-

New driving skill

- simtools.gaml.extensions.traffic/models/Driving Skill/models/Advanced models/Drive Random.gaml
 - simtools.gaml.extensions.traffic/models/Driving Skill/models/Advanced models/Following Paths.gaml
 - simtools.gaml.extensions.traffic/models/Driving Skill/models/Advanced models/Simple Intersection.gaml
 - simtools.gaml.extensions.traffic/models/Driving Skill/models/Advanced models/Traffic.gaml
 - simtools.gaml.extensions.traffic/models/Driving Skill/models/Simple model/ Simple Traffic Model.gaml
-

New network capabilities

- ummisco.gama.network/models/Network/2 Available protocols/HTTP Request/

HTTP GET.gaml

- [ummisco.gama.network/models/Network/2 Available protocols/HTTP Request/HTTP POST.gaml](#)
 - [ummisco.gama.network/models/Network/2 Available protocols/TCP protocol/TCP Server And Client Example .gaml](#)
 - [ummisco.gama.network/models/Network/2 Available protocols/TCP protocol/TCP Server Example.gaml](#)
 - [ummisco.gama.network/models/Network/2 Available protocols/WebSocket protocol/WebSocket Server And Client Example .gaml](#)
 - [ummisco.gama.network/models/Network/2 Available protocols/WebSocket protocol/WebSocket Server Example.gaml](#)
-

Usage of the `image` type

- [ummisco.gaml.extensions.image/models/Images/models/Casting Images.gaml](#)
 - [ummisco.gaml.extensions.image/models/Images/models/Declaring Images.gaml](#)
 - [ummisco.gaml.extensions.image/models/Images/models/Image Manipulation.gaml](#)
 - [ummisco.gaml.extensions.image/models/Images/models/Manual Snapshot.gaml](#)
-

New mathematical tests

- [ummisco.gaml.extensions.maths/tests/ODE Tests/models/Consistency Test.gaml](#)
- [ummisco.gaml.extensions.maths/tests/ODE Tests/models/Events Test.gaml](#)



>

Platform

Version: 1.9.1

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.

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](#) and [related tools](#), 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.

Version: 1.9.1

Installation and Launching

The GAMA platform can be easily installed on your machine: Windows, macOS or Ubuntu. Depending on user needs, GAMA can then be extended by using a number of additional plugins.

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

- [Installation](#)
- [Launching GAMA](#)
- [Headless Mode](#)
- [Updating GAMA](#)
- [Installing Plugins](#)

Version: 1.9.1

Installation

We made efforts to make the last release of GAMA (1.9.0) as easy as possible to install, by providing a version with an embedded Java JDK, limiting the installation to a 3-steps procedure: download, install and launch.

Download GAMA

GAMA 1.9.0 (the lastest release) comes in a version for each 3 environments Windows, macOS and Linux packaged in easy to use installers. You simply have to go on the [downloads page](#) and pick the one for your system.

For advanced users :

- GAMA provide also some versions without embedded JDK allowing you to download a lighter archive
 - This version requires **Java 17 JDK** to be installed on your computer (at least 17.0.3+7)
- It's also possible to download these 2 kinds of releases in simple zip archive (i.e. without installers), if you do so, please refer to the [post-installation procedure](#) at the end of this page

You'll find them on the [Github Release page](#)

Install procedure

After downloading the chosen GAMA version from the [Downloads page](#) for your

computer, you only have run the installer, follow steps, and [launch GAMA](#).

Step-by-step Windows

1. [Download](#) the installer `.exe` for Windows
2. Double-click on the downloaded `.exe` file
3. Accept to run the app



4. Follow the installer with the onscreen steps
5. Done, you can start GAMA from your computer now.

NB: If you need to launch [GAMA Headless](#), GAMA is installed under `C:\Program Files\Gama` by default

Step-by-step macOS

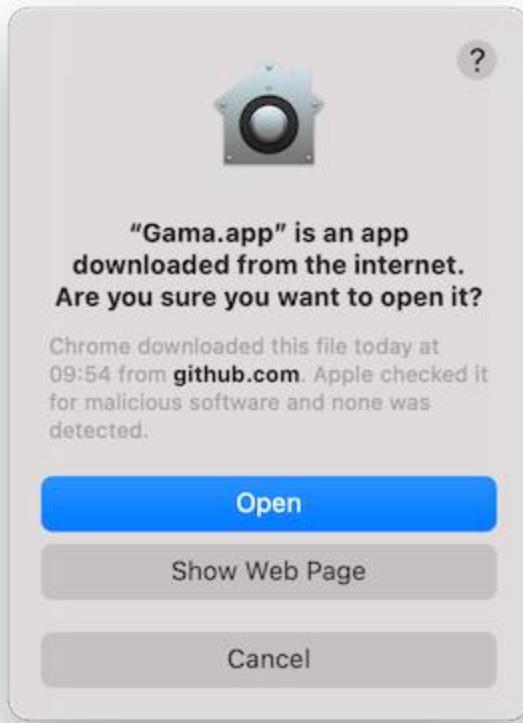
In macOS we have two ways of installing gama: either the regular and user friendly `.dmg` installer, or the command line way with the `homebrew` package manager.

DMG version

1. [Download](#) the installer `.dmg` for macOS
 - There is a built specifically for Macintosh M1 (also called *with Apple silicon*). You can check by clicking on the top-left apple, `About this Mac`: the pop-up window will give details about the processor. If you're not sure and your Macintosh is from 2021 (or earlier) you probably need this specific version
2. Double-click on the downloaded `.dmg` file
3. Drag'n'drop GAMA icon to your computer (Applications folder or Desktop for instance)



4. Done, you can start GAMA from your computer now. At the first launch of GAMA, a popup window will appear warning you that GAMA is a software downloaded from internet and asking whether you want to open it. Click on the Open button.



NB: Note that the first launch of GAMA should be made in GUI mode (clicking on the icon) and not in headless mode. NB2: If you need to launch **GAMA Headless** after, GAMA is installed where you *dragged and dropped* the Gama.app

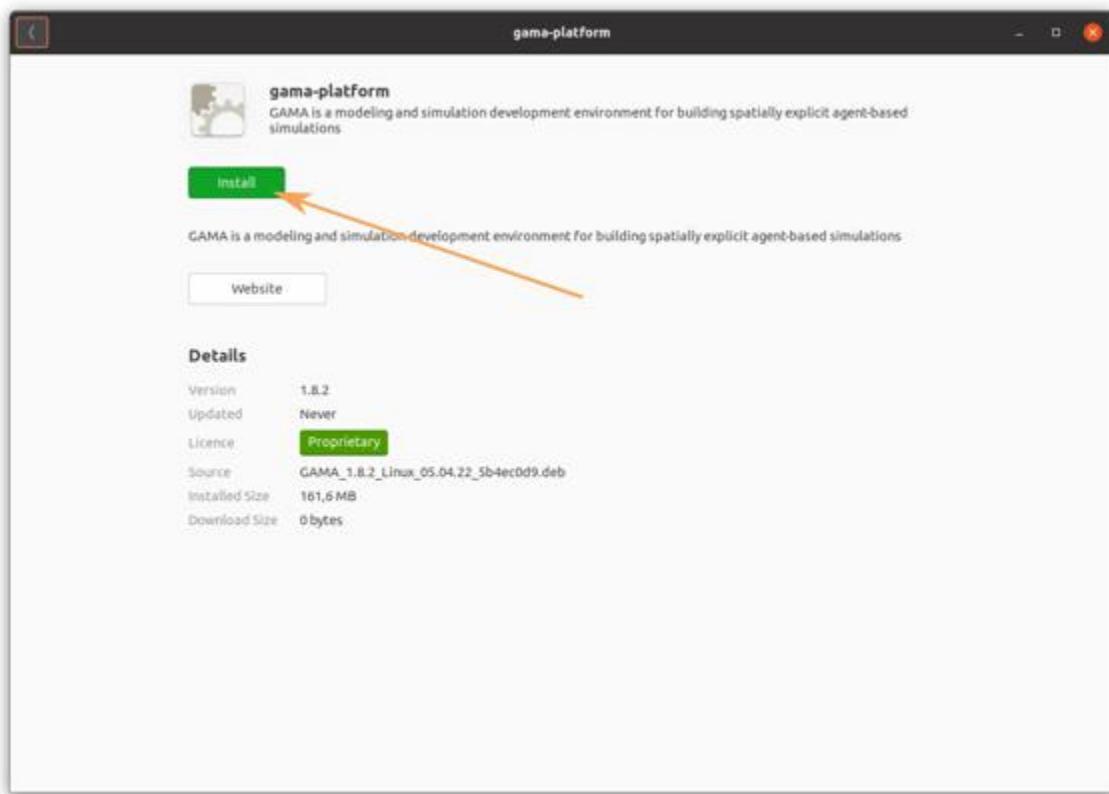
brew version

1. Install brew on your computer: just follow the instruction from the [website](#).
2. Open a terminal
3. Run the command `brew install gama` or `brew install gama-jdk` for the JDK version

Step-by-step Linux

Debian/Ubuntu based

1. Download the installer `.deb` for Ubuntu (and any Debian-based systems)
2. Double-click on the downloaded `.deb` file
3. Click on `install`



4. You could be asked for your password
5. Done, you can start GAMA from your computer now

NB: If you need to launch [GAMA Headless](#), GAMA is installed under `/opt/gama-platform`

Arch Linux based

AUR packages with latest version of GAMA exists for both version with and without embedded JDK. You can download them with a command as follows:

```
yay -S gama-platform{-jdk}
```

Step-by-step Docker

A Docker image of GAMA exists to execute [GAMA Headless](#) inside a container.

1. Install docker on your system following the [official documentation](#)
2. Pull the GAMA's docker you want to use (e.g. `docker pull gamaplatform/gama:1.9.0`)
3. Run this GAMA's docker with your headless command (e.g. `docker run gamaplatform/gama:1.9.0 -help`)

You can found all the tags and more detailed documentation on the [Docker Hub](#) or on the corresponding [Github's Repository](#)

System Requirements

GAMA 1.9.0 requires approximately 540MB of disk space and a minimum of 2GB of RAM (to increase the portion of memory usable by GAMA, please refer to [these instructions](#)).

Windows post-installation setting (only for zip install)

If you decided to install gama yourself from the `zip` file, it is important that you change the Windows HDPI compatibility settings. To do so, go to your `Gama.exe` file, right click and it and select `properties`, then go to the `Compatibility` tab and click on the `Change high DPI settings` button:

G Gama.exe Properties X

Security	Details	Previous Versions
General	Compatibility	Digital Signatures

If this program isn't working correctly on this version of Windows, try running the compatibility troubleshooter.

[Run compatibility troubleshooter](#)

[How do I choose compatibility settings manually?](#)

Compatibility mode

Run this program in compatibility mode for:

Windows 8 ▾

Settings

Reduced color mode

8-bit (256) color ▾

Run in 640 x 480 screen resolution

Disable fullscreen optimizations

Run this program as an administrator

Register this program for restart

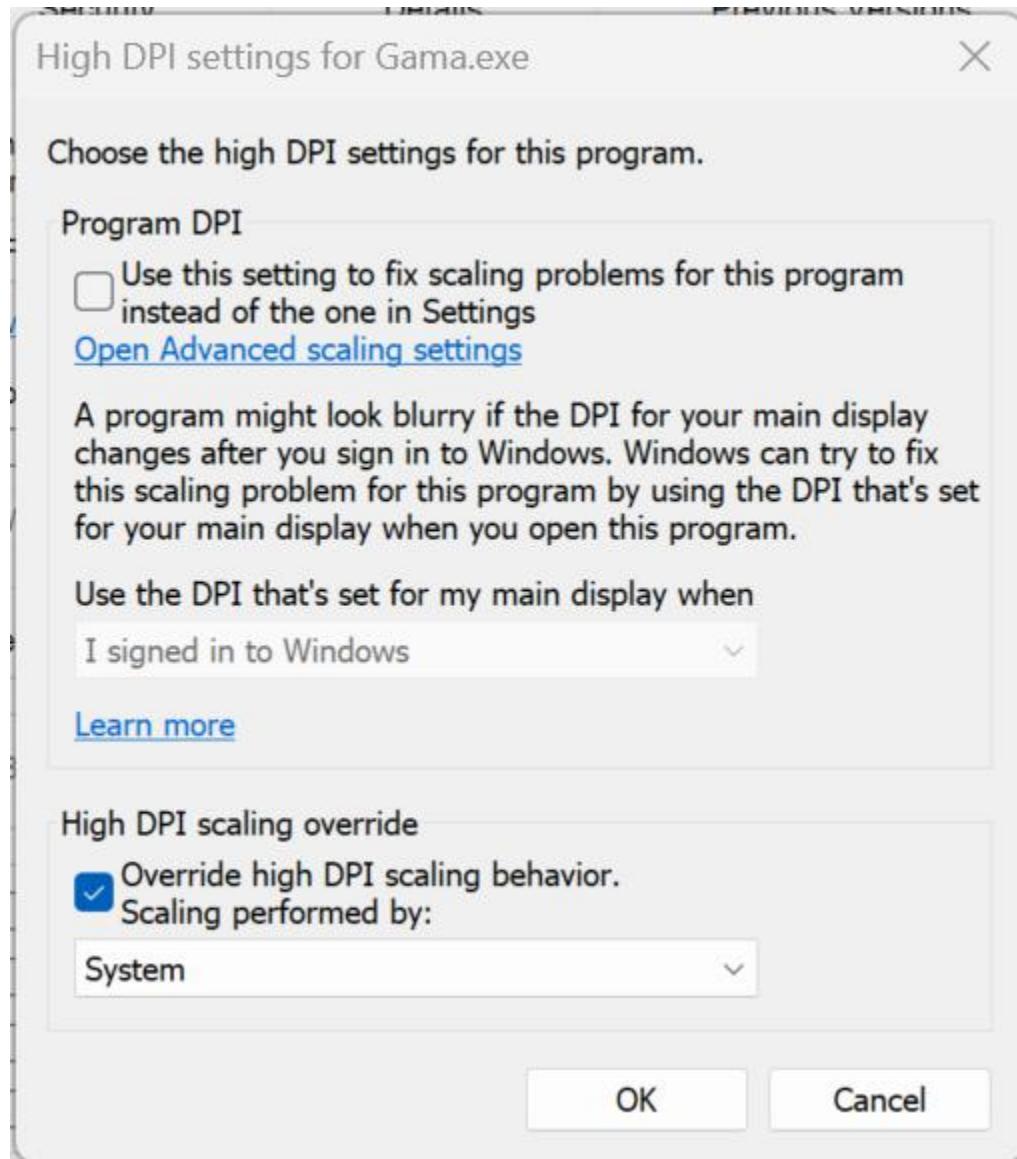
Use legacy display ICC color management

Change high DPI settings

 Change settings for all users

OK Cancel Apply

In the new window, check the `Override high DPI scaling behavior` option and select the `System` value.



These options are necessary to avoid most graphical problem using gama on Windows

Version: 1.9.1

Launching GAMA

Running GAMA for the first time requires that you launch the application (`Gama.app` on Mac OS X, `Gama.exe` on Windows, `Gama` on Linux, located in the folder called `GAMA_1.8_YOUR_OS_NAME` once you have unzipped the downloaded archive). In case you are unable to launch the application, or if error messages appear, please refer to the [installation](#) or [troubleshooting](#) instructions.

Table of contents

- [Launching GAMA](#)
 - [Launching the Application](#)
 - [Launching the Application from the command line](#)
 - [Choosing a Workspace](#)
 - [Welcome Page](#)

Launching the Application

The extraction of the downloaded archive provides:

- on Mac OS X: a single file named `Gama.app`
- on Windows and Linux: a folder named `GAMA_1.8_YOUR_OS_NAME` containing, among many other files and folders, the `Gama.exe` file (for Windows) and `Gama` (for Linux).

Running GAMA requires that you launch the application file (`Gama.app`) on Mac OS X,

`Gama.exe` on Windows, `Gama` on Linux) by double-clicking on them or from a terminal.

Launching the Application from the command line

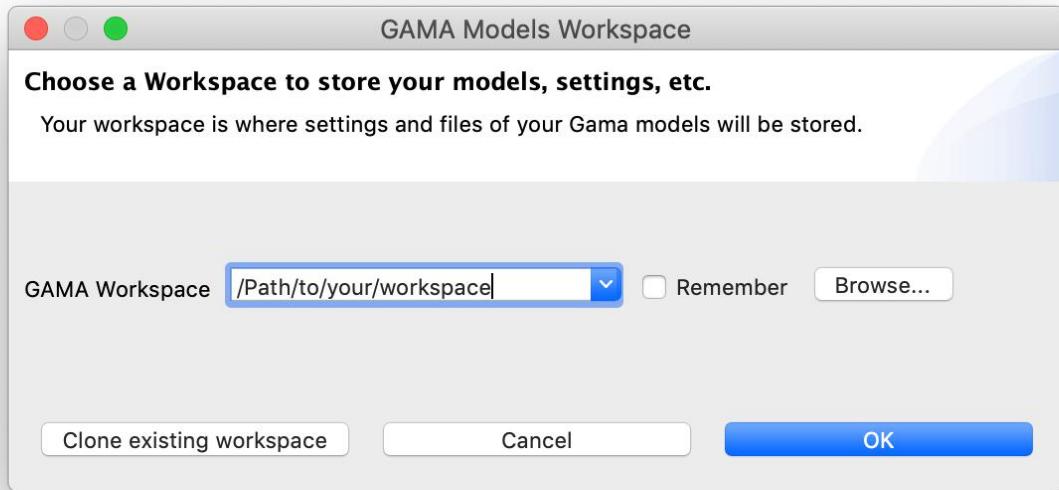
Note that GAMA can also be launched in two different other ways:

1. In a so-called *headless mode* (i.e. without a 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 Mac OS 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

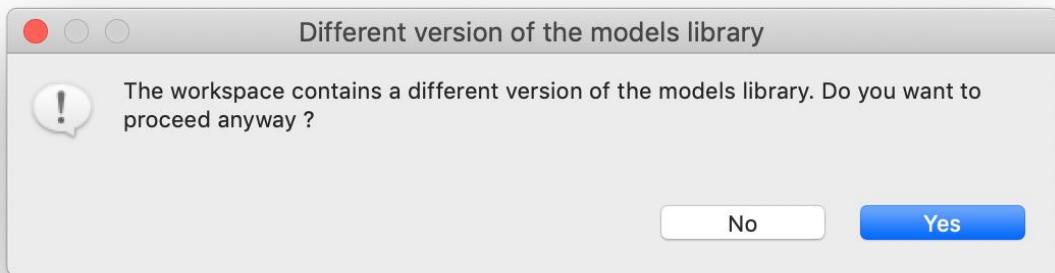
Past the splash screen, GAMA will ask you to choose a workspace in which to store your models and their associated data and settings. The workspace can be any folder in your filesystem on which you have read/write privileges. If you want GAMA to remember your choice next time you run it (it can be handy 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 a previous GAMA version (and still "remembered"). In that case, a warning will be produced to indicate that the model 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.

When you try to choose a workspace used with a previous of GAMA, the following pop-up will appear.



The following pop-up appears when the user wants to create a new workspace in a folder that does not exist. Click on OK to create the folder and set this new folder as the GAMA workspace.

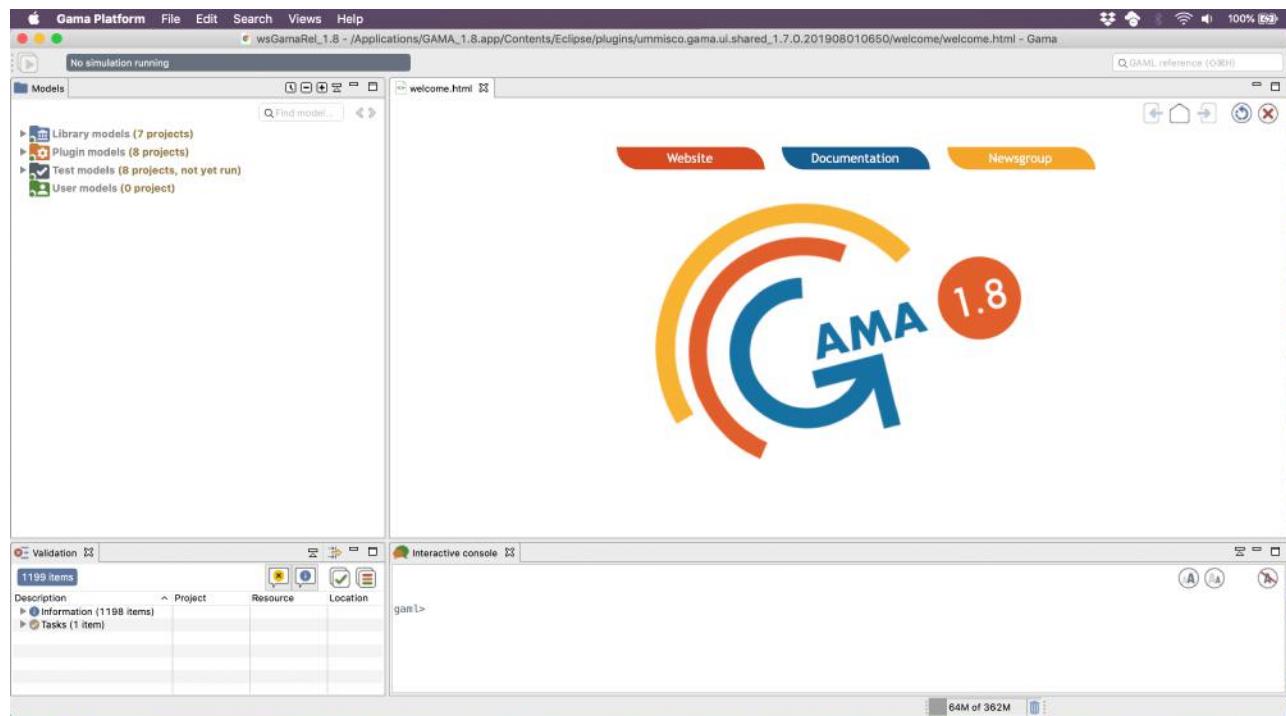


Welcome Page

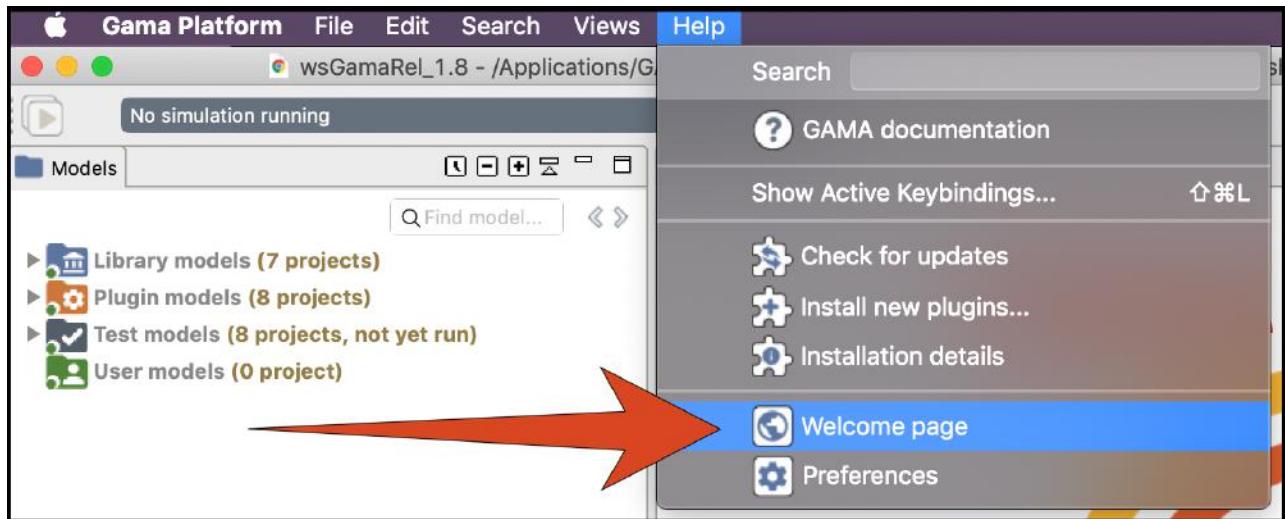
As soon as the workspace is created, GAMA will open and you will be presented with its **first window**. GAMA is based on [Eclipse](#) and reuses most of its visual metaphors for organizing the work of the modeler. The main window is then composed of several **parts**, which can be **views** or **editors**, 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), the Problems view, which indicates errors and warnings present in the models stored in the workspace and an interactive console, which allows the modeler to try some expressions and get an immediate result.



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 "Help" menu).



From this point, you are now able to [edit a new model](#), [navigate in the model library](#), or [import an existing model](#).

Version: 1.9.1

Updating GAMA

Unless you are using the version of GAMA built from the sources available in the GIT repository of the project (see [here](#)), you are normally running a specific **release** of GAMA that sports a given **version number** (e.g. GAMA 1.8.1, GAMA 1.7, GAMA 1.6.1, 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 the version 1.6.1, however, GAMA has been enhanced to support a *self_update* mechanism, which allows you to import from the GAMA update site additional plugins (offering new features) or updated versions of the plugins that constitute the core of GAMA.

The update of GAMA will be detailed on this page; to install new additional plugins (from the GAMA community or third-party developers) see the page dedicated to [the installation of new plugins](#).

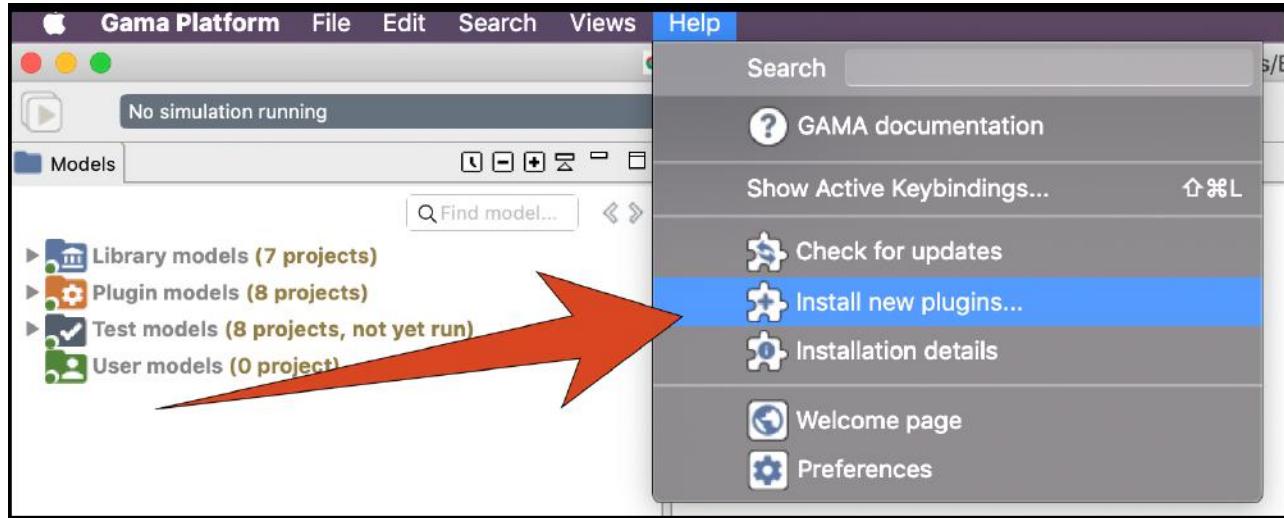
Table of contents

- [Updating GAMA](#)
 - [Manual Update](#)
 - [Automatic Update](#)

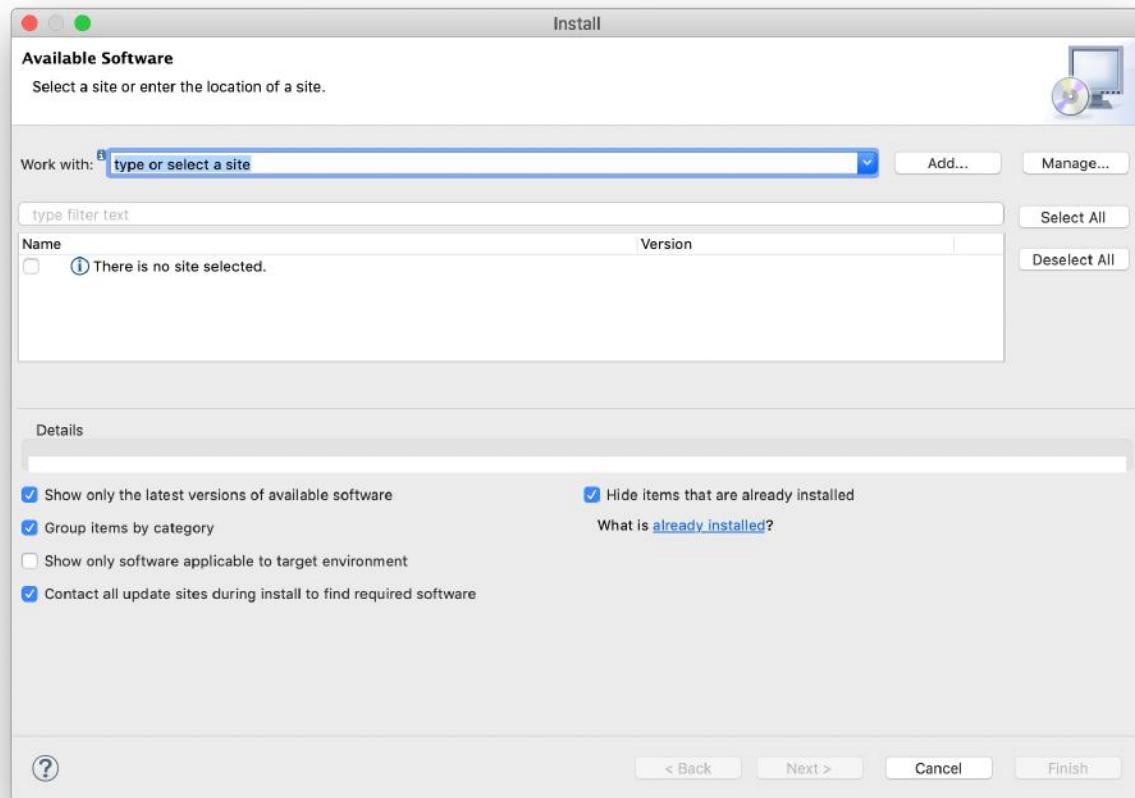
Manual Update

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 *uninstalling* some plugins) are provided. Once invoked, it makes the following dialog appear:



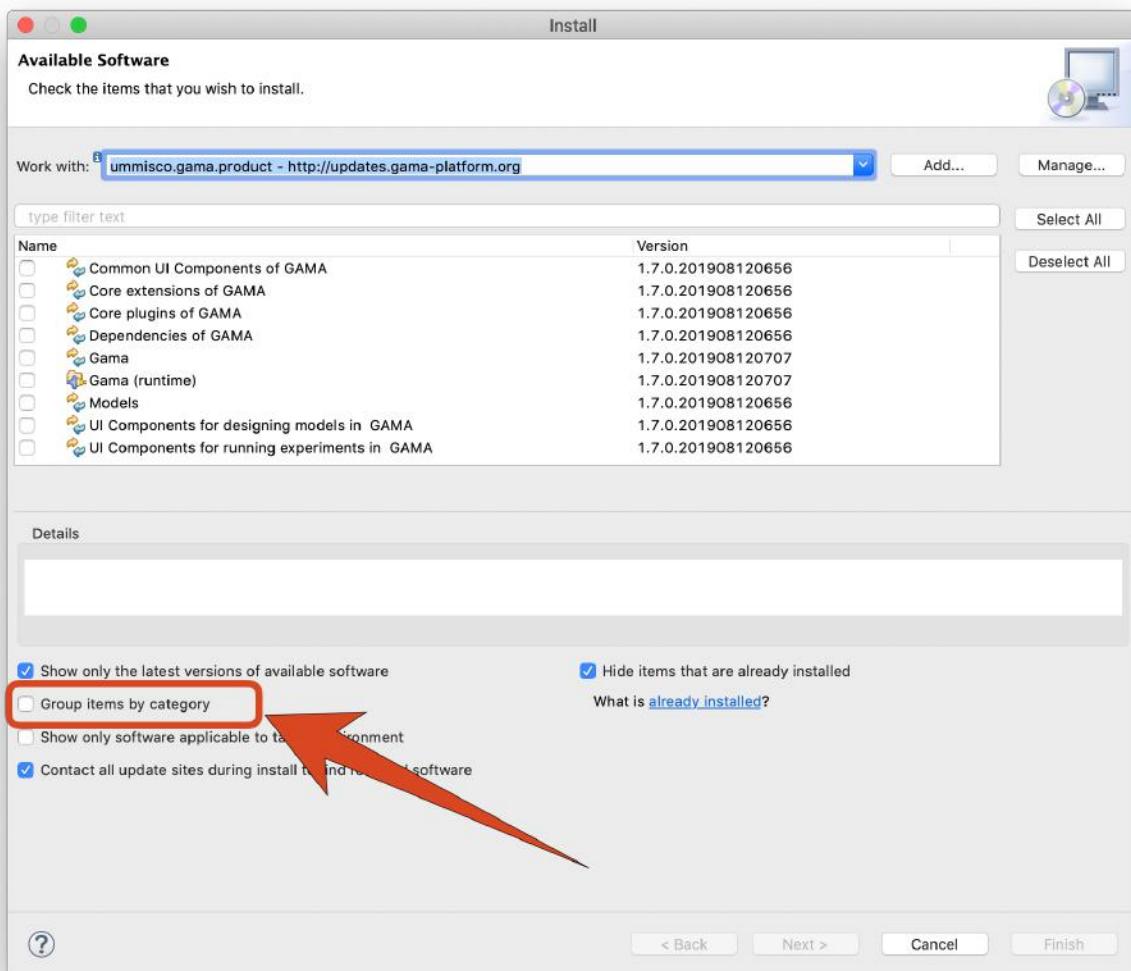
GAMA expects the user to enter a so-called *update site*. You can copy and paste the following line (or choose it from the drop-down menu as this address is built inside GAMA):

<http://updates.gama-platform.org>

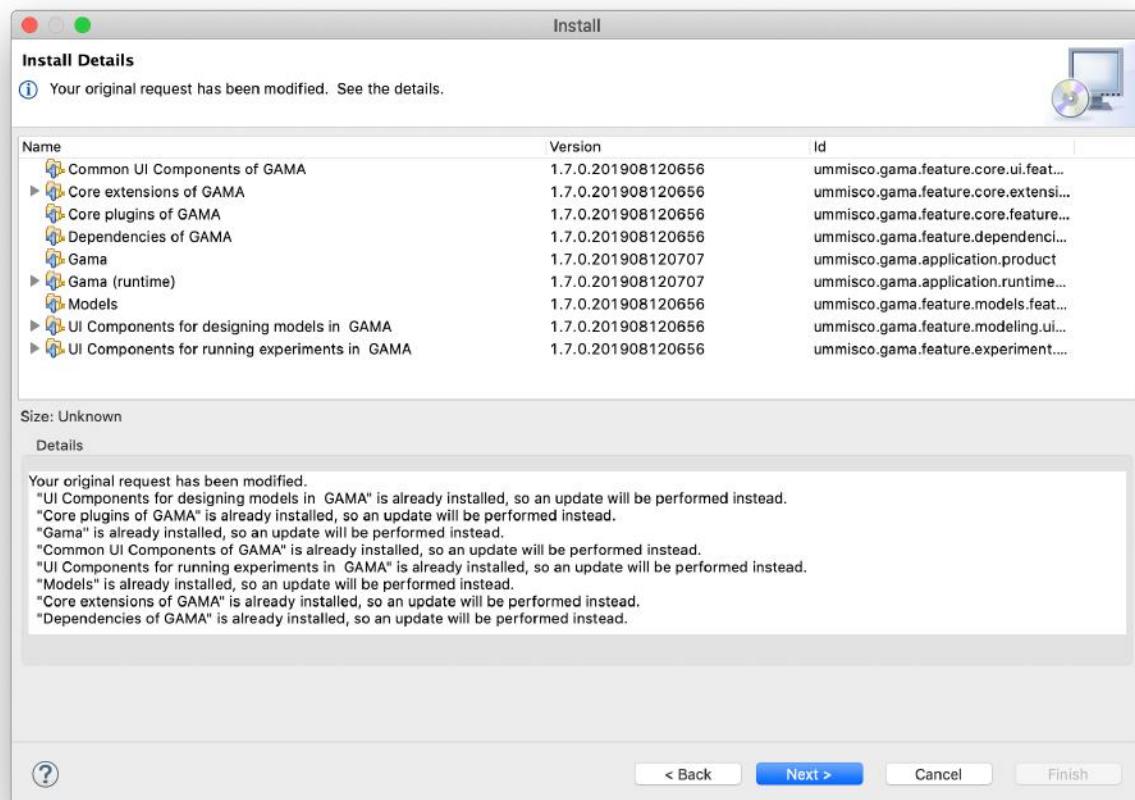
GAMA will then scan the entire update site, looking both for new plugins of the GAMA kernel and updates to existing plugins. The list available in your installation will, of course, be different from the one displayed here.

In order to make the plugins appear, you need to uncheck the option "Group items by category". Choose the ones you want to update (or install) and click "Next"

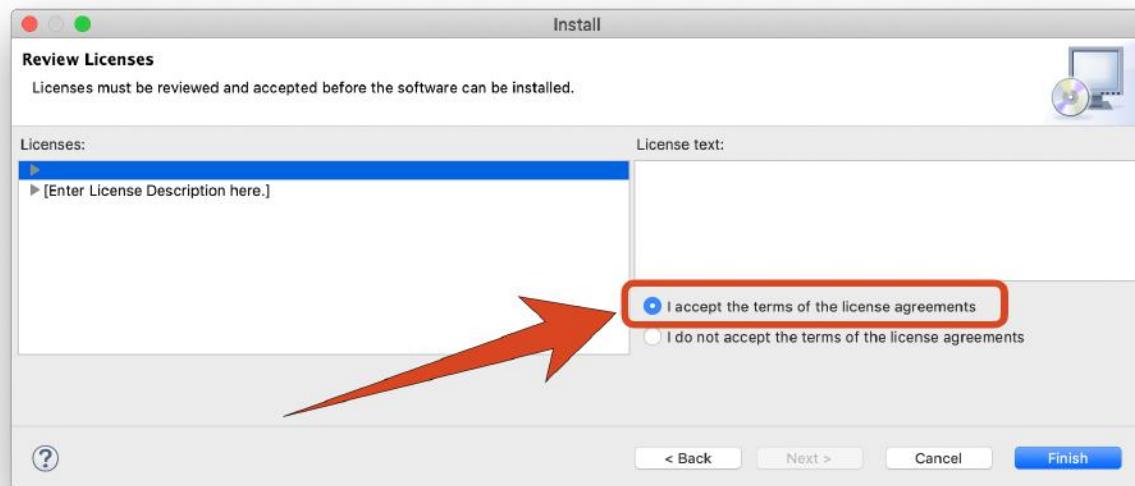
>".



A summary page will appear, indicating which plugins will actually be installed (since some plugins might require additional plugins to run properly). Click on the "Next >" button.



A license page will then appear: you have to accept all of them. Click on "Finish".



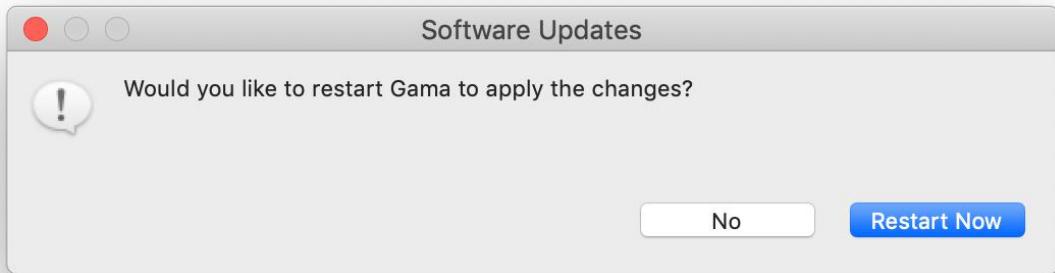
GAMA will then proceed to the installation (that can be canceled at any time) of the chosen plugins.

During the course of the installation, you might receive the following warning, that you can dismiss by clicking "OK". You can click on the "Details" button to see which plugins contain unsigned contents.



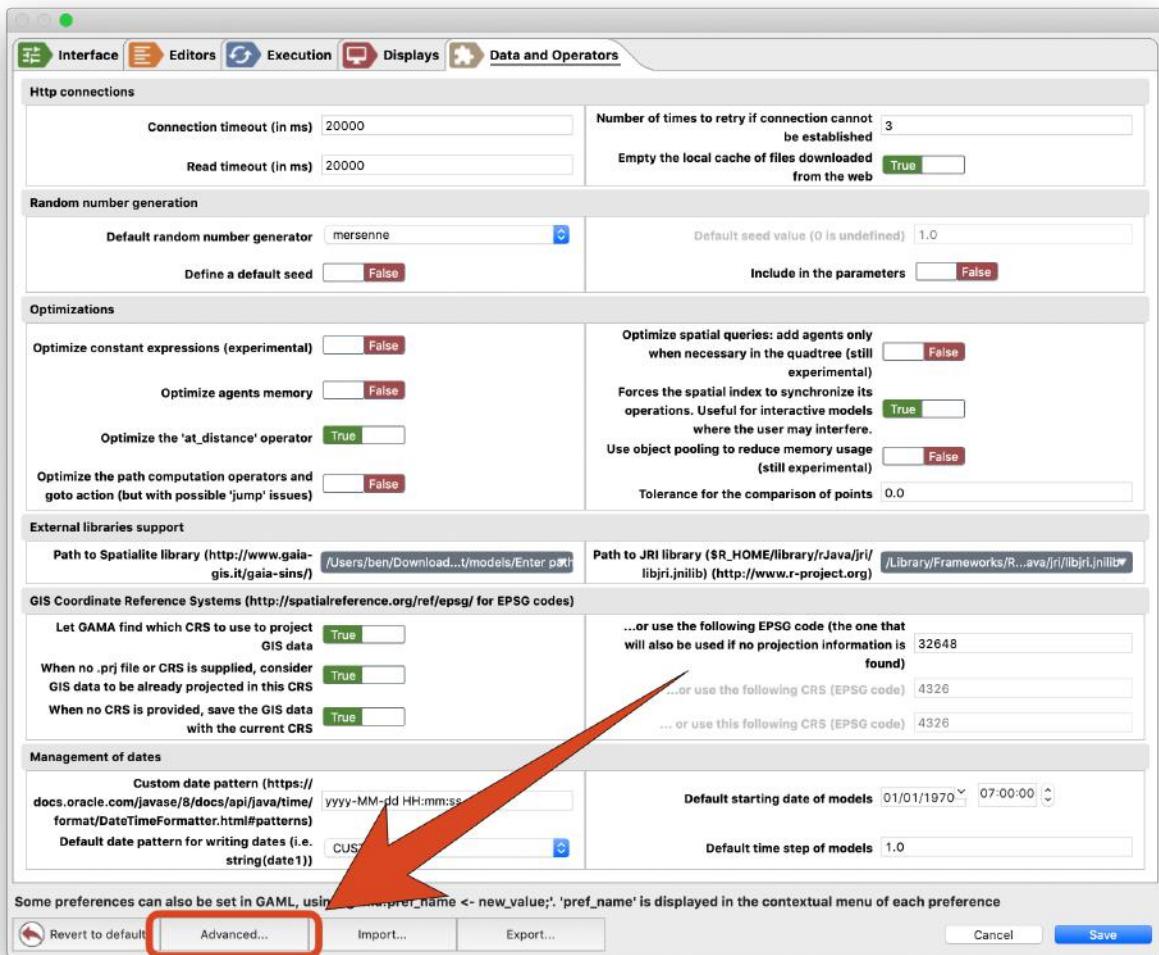
Once the plugins are installed, GAMA will ask you whether you want to restart or not.

It is always safer to do so, so select "Restart now" and let it close by itself, register the new plugins and restart.

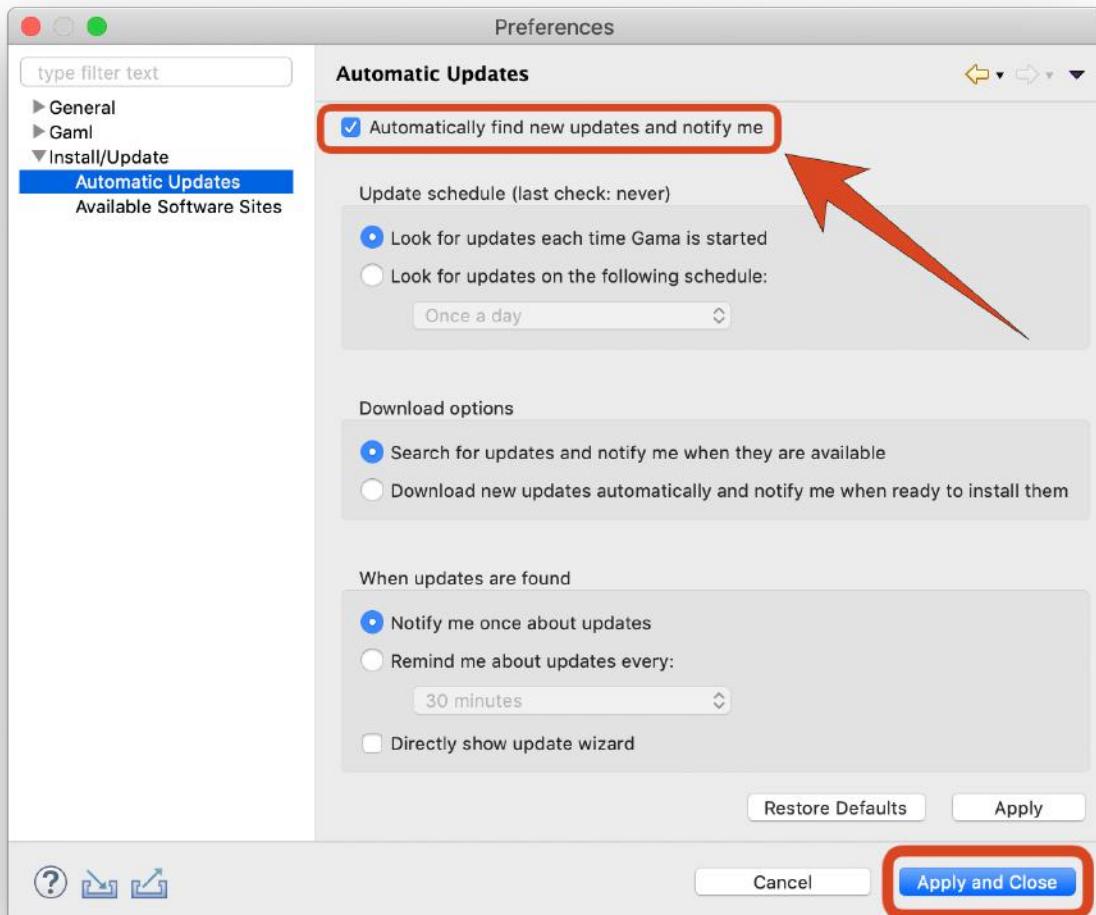


Automatic Update

GAMA offers a mechanism to monitor the availability of updates to the plugins already installed. To activate 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 in the top of the dialog and choose the best settings for your workflow. Clicking on "Apply and close" 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.

Version: 1.9.1

Installing Plugins

Besides the plugins delivered by the developers of the GAMA platform, 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 2022-12 (version number 4.26.0), which excludes de facto all the plugins targeting solely a specific different version 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 1.9.0 require [XText v. 2.29.0](#) to be installed.

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

Table of contents

- [Installing Plugins](#)
 - [Installation](#)
 - [Selected plugins provided by GAMA community](#)
 - [Toward participative simulations with Remote.Gui and Gaming plugins](#)
 - [RJava plugin](#)
 - [Weka and Matlab plugins](#)
 - [Selected Plugins](#)
 - [Git](#)
 - [CSV Edit](#)
 - [Quickimage](#)
 - [RSS/Atom Feed View](#)
 - [CKEditor](#)
 - [Startexplorer](#)
 - [Pathtools](#)

Installation

Installing new plugins is a process identical to the one described when [updating GAMA](#), with one exception: the *update site* to enter is normally provided by the vendor of the additional plugin and must be entered instead of GAMA's one in the dialog.

Let suppose that we want to install a GAMA plugin developed in order to allow GAMA to ask [R](#) to do some computations. This plugin is developed by the GAMA community, but the installation of any plugin will be similar, **only the address of the update site will change**. To install this plugin, [open the pane to install new plugins](#): "Help >

Install new plugins ...".

Choose in the "Work with..." text field:

```
msi.gama.experimental.p2updatesite -  
http://updates.gama-platform.org/experimental
```

If it is not available, you can simply type the address of the update site in the text field:

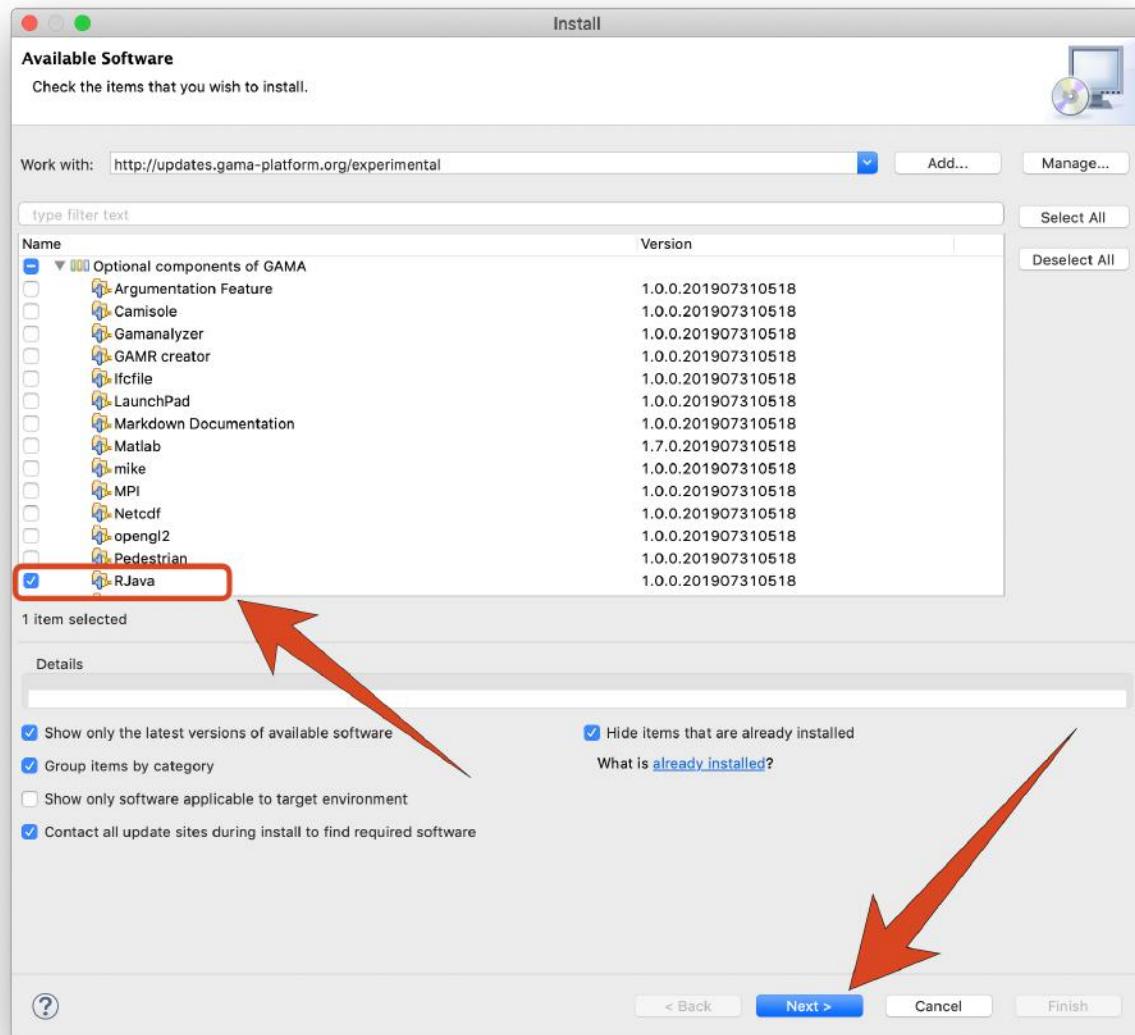
```
http://updates.gama-platform.org/experimental/<GAMA-VERSION>
```

Note: The `<GAMA-VERSION>` should be replaced by the version of GAMA you are using.

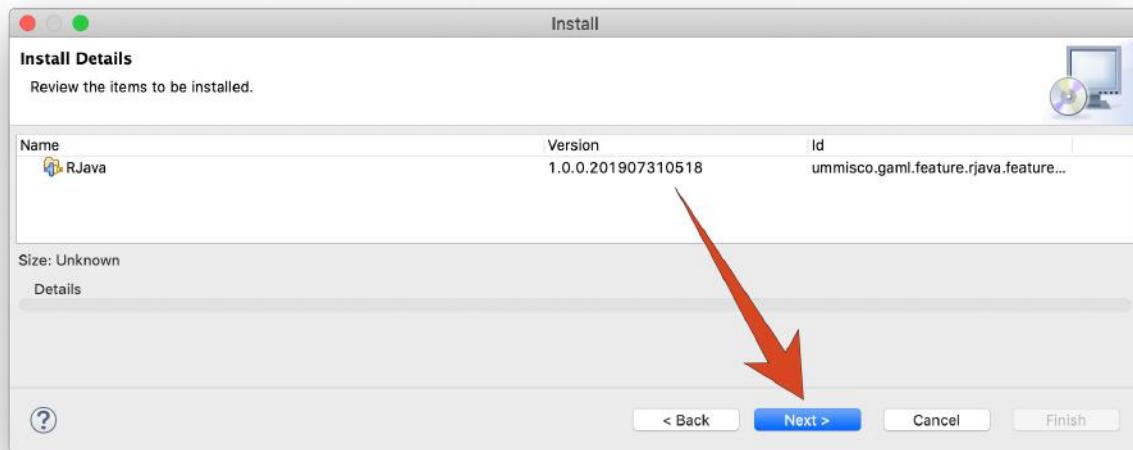
For example, current latest version is **GAMA 1.9.0**, then the address is this :

```
http://updates.gama-platform.org/experimental/1.9.0
```

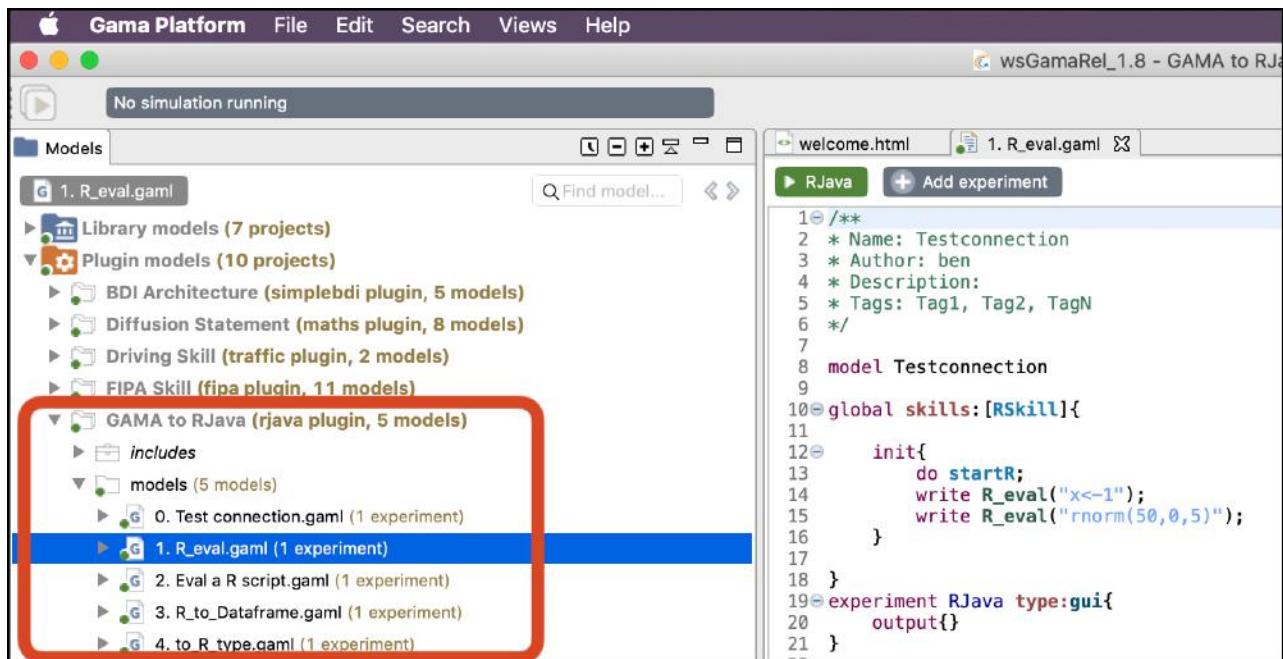
Among all the plugins, select `RJava` in the category "Optional components of GAMA" and click on "Next >" button.



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. In the case of plugins extending the features of GAMA, some example models are often provided with the new plugins to illustrate its use (and it is the case for RJava). These new models are accessible in GAMA from Plugin models in a dedicated folder (GAMA to Rjava in the case of RJava). We may need to refresh the model library to let it appear. **Notice that you need to configure GAMA to access R before running these models.**



Selected plugins provided by the GAMA community

The update site located at the address <http://updates.gama-platform.org/experimental> contains new plugins for GAMA mainly developed by the GAMA community ([its Github repository is available here](#)). As the name of the repository highlights it, these plugins are most of them still in development, before integration in the kernel of GAMA.

Toward participative simulations with `Remote.Gui` and `Gaming` plugins

There are more and more applications of GAMA for participative simulations ([LittoSim](#), [MarakAir](#), [HoanKiemAir](#)...). There was thus a need for new features to improve the possible interactions with simulations and the definition of the Graphical User Interface. The two plugins `Remote.Gui` and `Gaming` (available in the

"Participative simulation" category) attempts to fill this need.

- `Remote.Gui` allows exposing some model parameters, in order that they can be modified through [a network](#). This allows, for example, to develop a remote application (e.g. Android application) to control the parameters' values during the simulation.
- `Gaming` allows the modeler to define displays that are much more interactive. This is used to define serious games in which the users can have a wide range of possible interactions with the simulation.

RJava plugin

This plugin allows the modeler to launch some computation on the [R](#) software. To this purpose, [R](#) should be installed on your computer and [GAMA](#) should be properly [configured](#).

This possible connection to [R](#) opens thus the possibility for the modeler to use all the statistical functions and libraries developed in this tool of reference. In addition, R scripts defined by the modeler can also be used directly from his/her GAMA model.

Weka and Matlab plugins

Similarly to [RJava](#), [Matlab](#) and [Weka](#) plugins allow the modeler to run computations on the [Matlab](#) and [Weka](#) software, taking advantages of all the possibilities of these softwares and of scripts defined by him/herself.

Notice that the [Matlab](#) plugin requires that MATLAB 2019a is installed and activated on your computer.



> Platform

> Workspace, Projects and Models

Version: 1.9.1

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, the 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 4 [categories](#), which are themselves organized into [projects](#).

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

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

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

1. Navigating in the Workspace

2. Changing Workspace

3. Importing Models

Version: 1.9.1

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**.

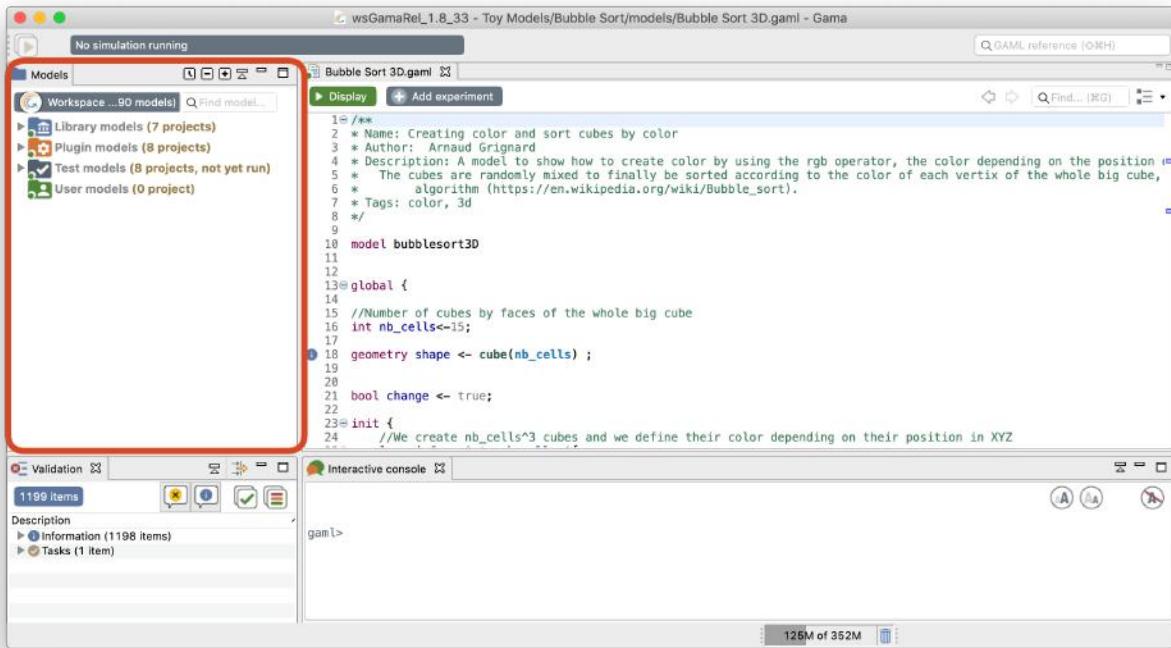
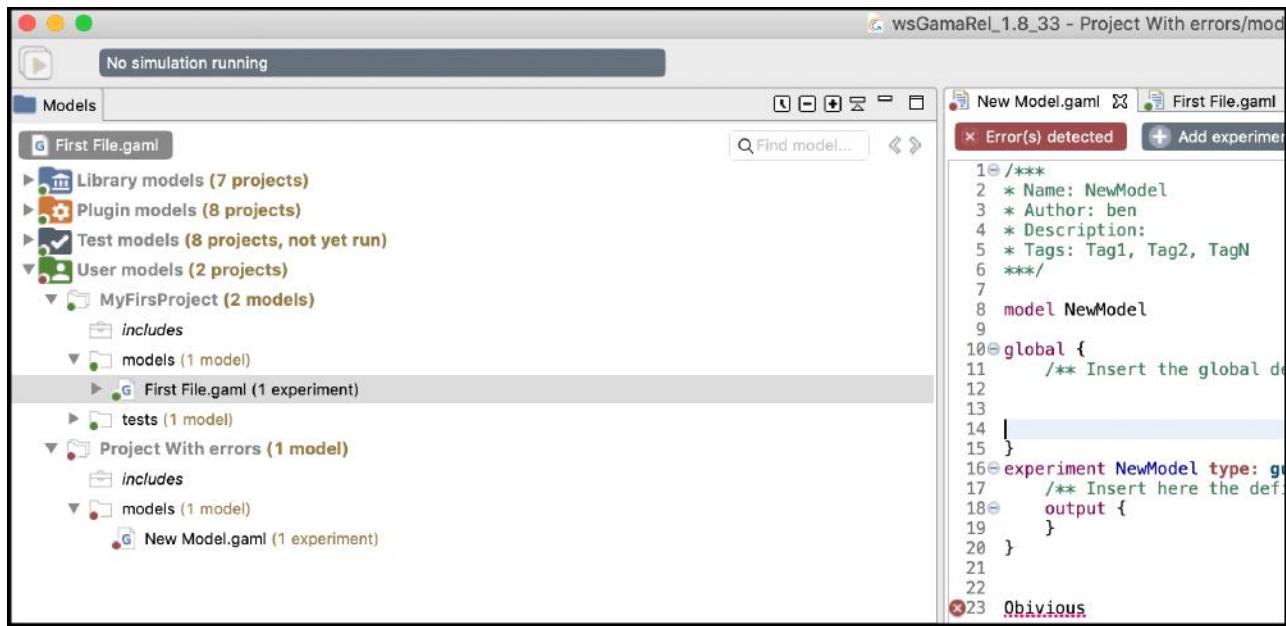


Table of contents

- Navigating in the Workspace
 - Status of projects and models
 - The Different Categories of Models
 - Library models
 - Plugin models
 - Test models
 - User models
 - Inspect Models
 - Moving Models Around
 - Closing and Deleting Projects

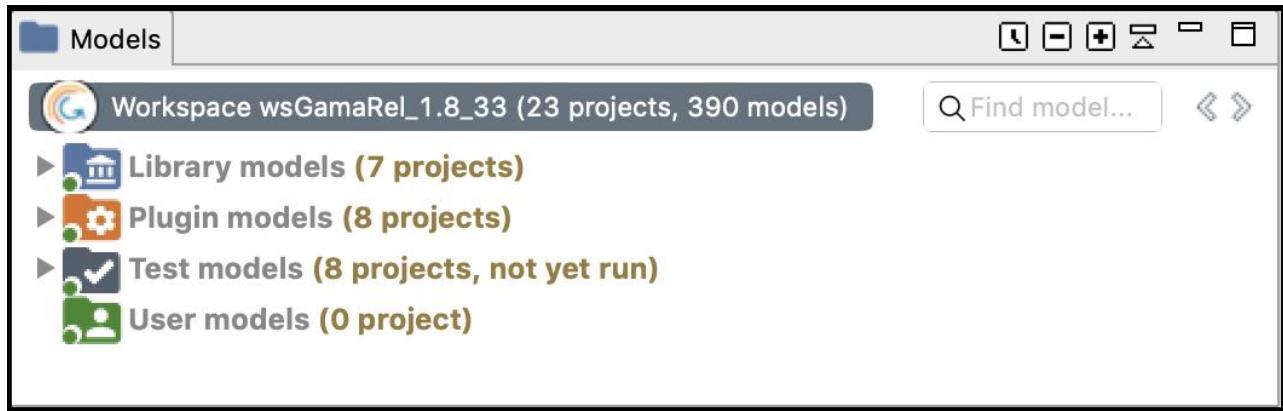
Status of projects and models

All the projects and models have an icon with a red or green circle on it. This eases to locate models containing **compilation errors** (red circle) and projects that have been successfully validated (green circle).



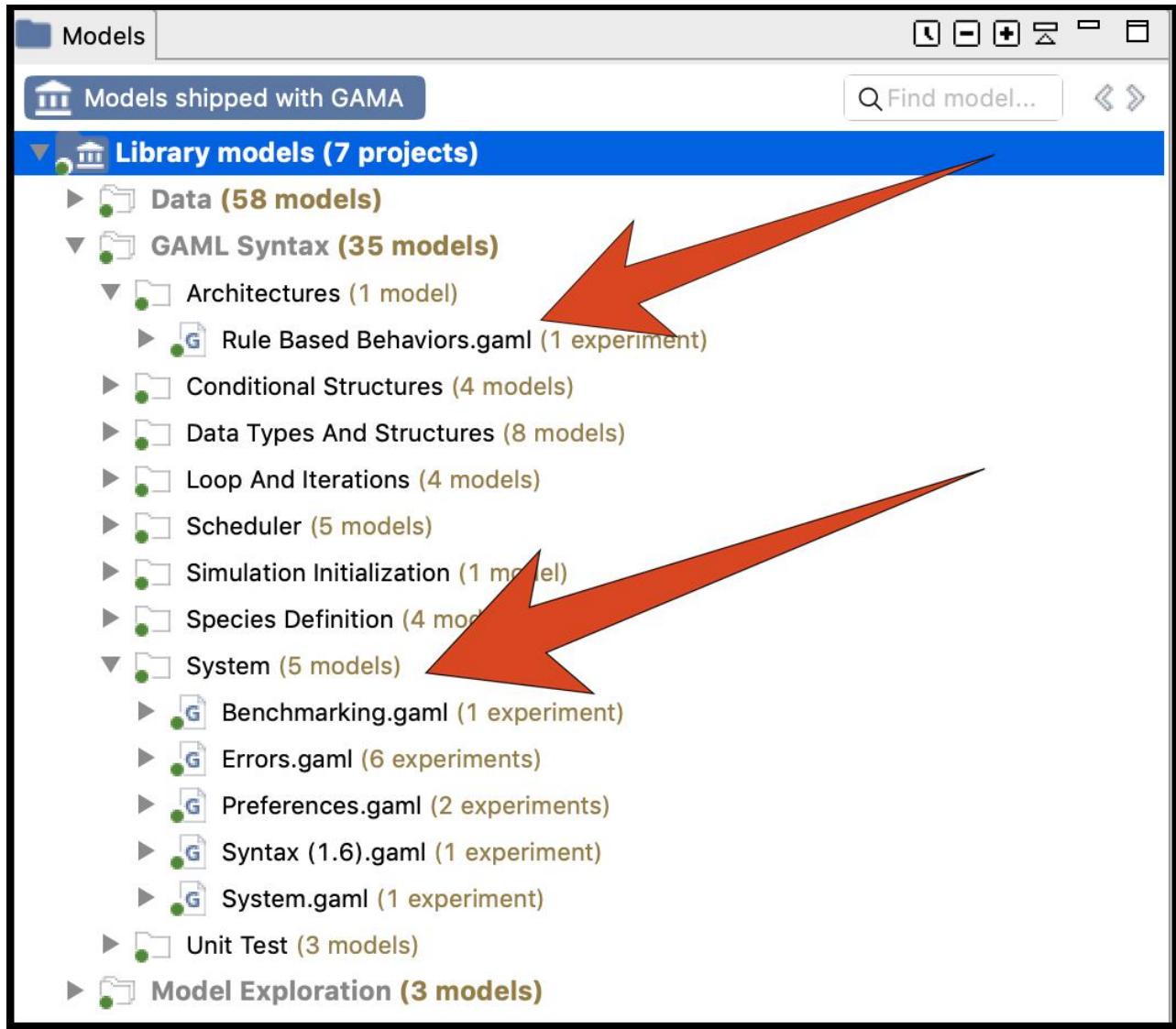
The Different Categories of Models

In the *Navigator*, models are organized in four different categories: *Models library*, *Plugin models*, *Test 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*) needed for the models to function (such as data files). 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.



Library models

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 7 main projects in GAMA 1.9.0, which are further refined in folders and sub-folders that contain model files and resources.

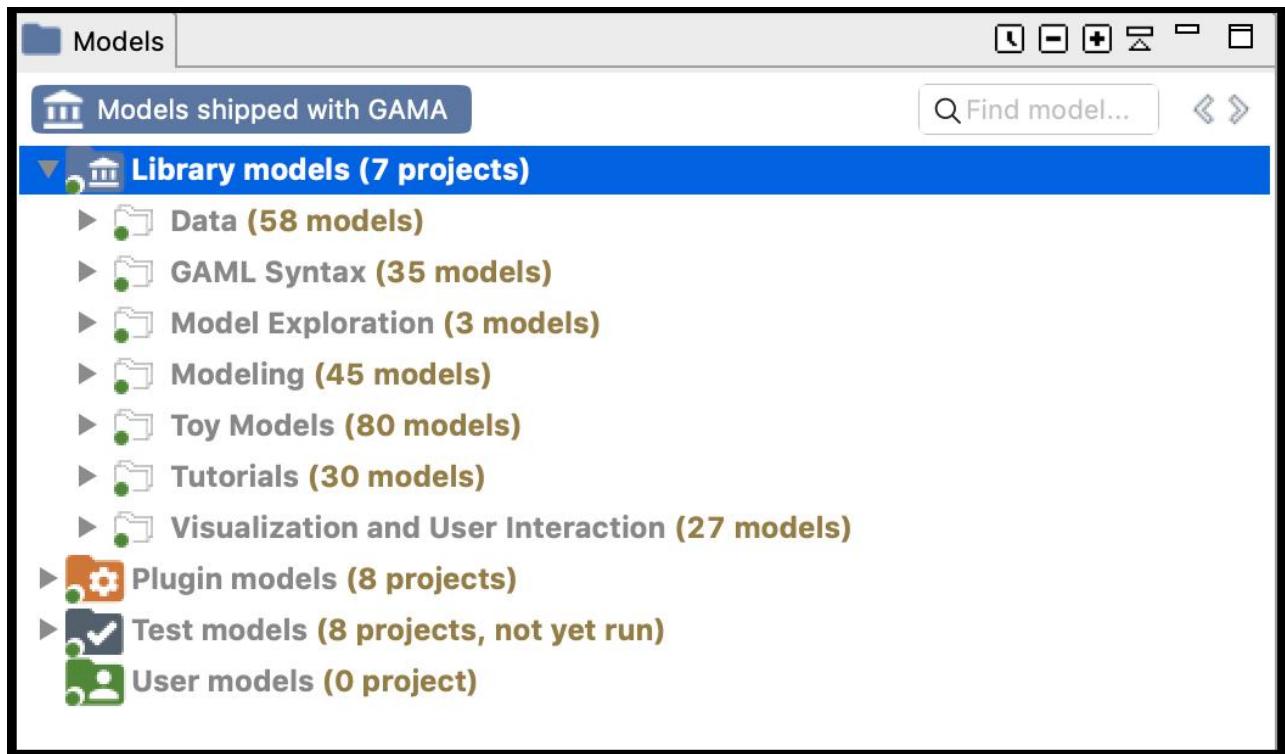


The 7 main projects on the Library models are:

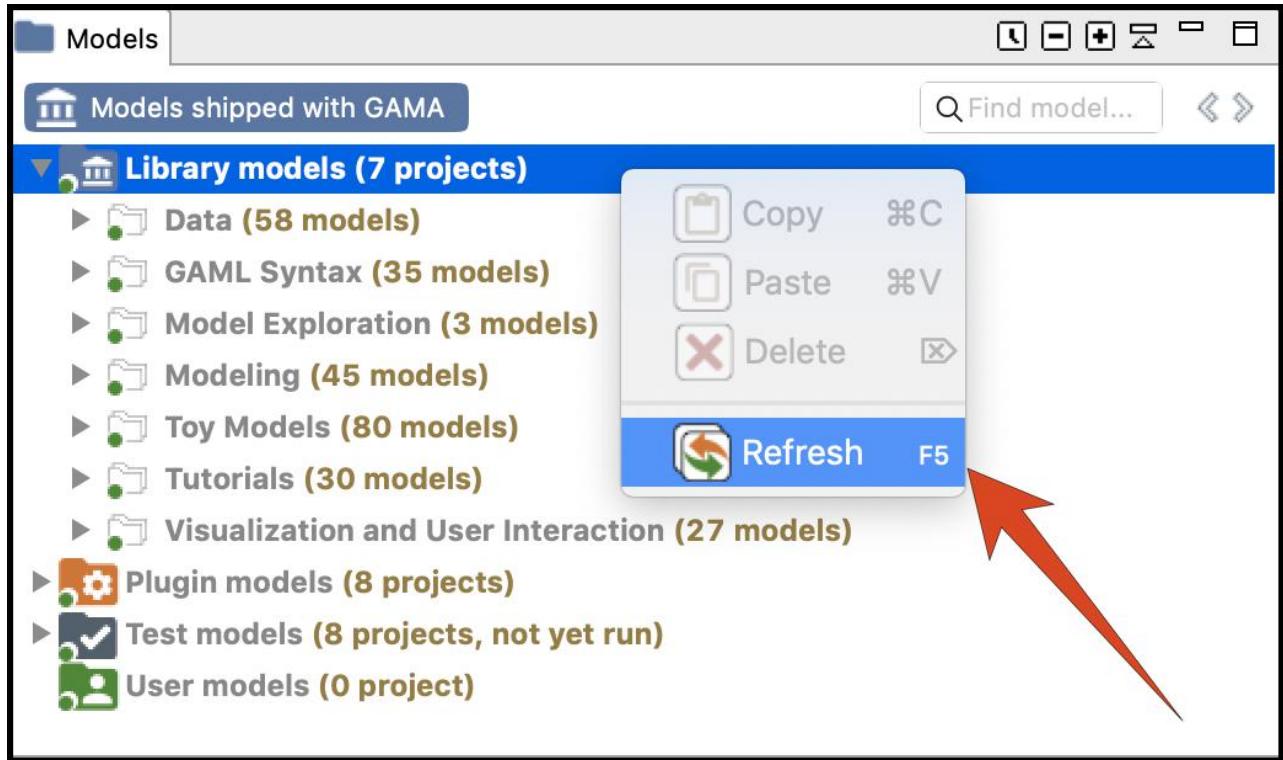
- **Data:** all these plugins illustrate how to manage data in GAML. This includes how to **import** data (in all the supported formats) into a model, **export** (i.e. save) agents or data in the simulations in files, **clean** data (e.g. clean a road network), get and save data in **databases**, and use **data analysis** operators.
- **GAML Syntax:** these models have the only goal to illustrate the syntax of the GAML language. This includes how to use the various data structures (list, map, matrix...), architectures, loop, interactions, and conditional structures, or how to

schedule agents...

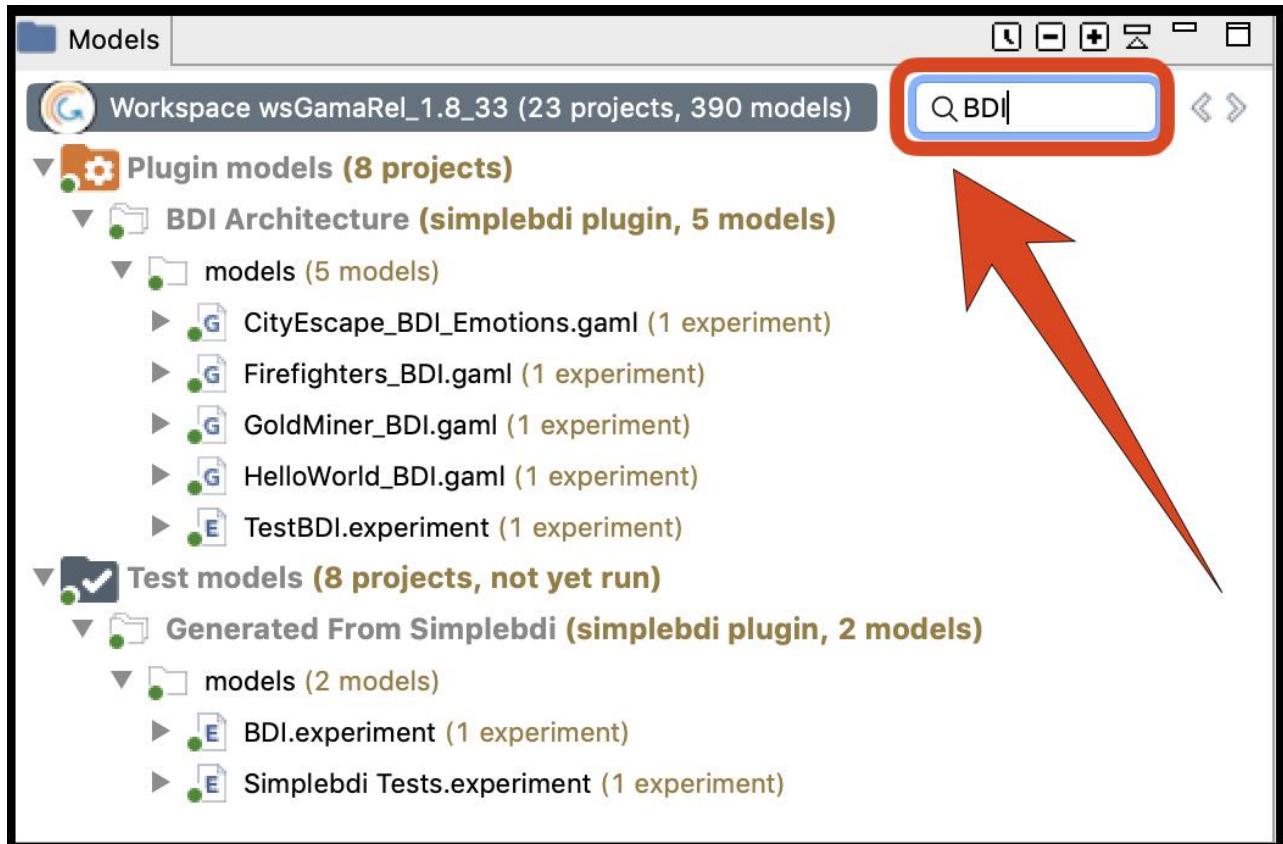
- **Model Exploration:** all these models illustrate the various ways to explore models and in particular the various possible experiment (batch, multi-simulations...).
- **Modeling:** these models provide implementations of various classical difficulties encountered by modelers: how to make agents move (on a graph, a grid...), how to implement decision-making process...
- **Toy Models:** these models are replications of classical models from the literature, including Sugarscape, Schelling, ants, boids...
- **Tutorials:** this project contains all the files of the various tutorials (available from the website).
- **Visualization and User Interaction:** these models illustrate most of the GAMA features in terms of visualization and interactions with the simulation, e.g. the 3D visualization...



It may happen, on 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.

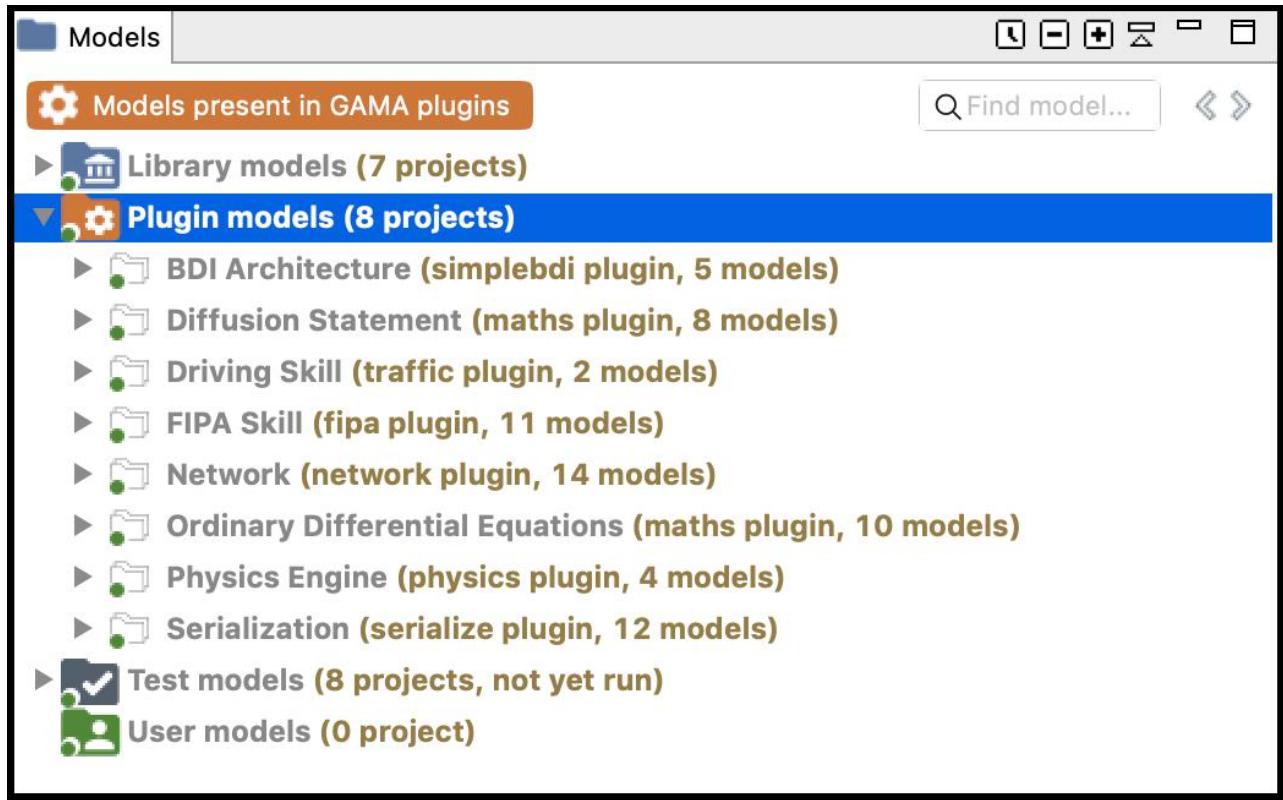


To look up for a particular model in the library, users can use the "Find model..." search bar, which allows looking for models by their title (for example, models containing "BDI" in the example below).



Plugin models

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



When you add an additional plugin extending the GAML language is added, a new project can be added to this category.

Test models

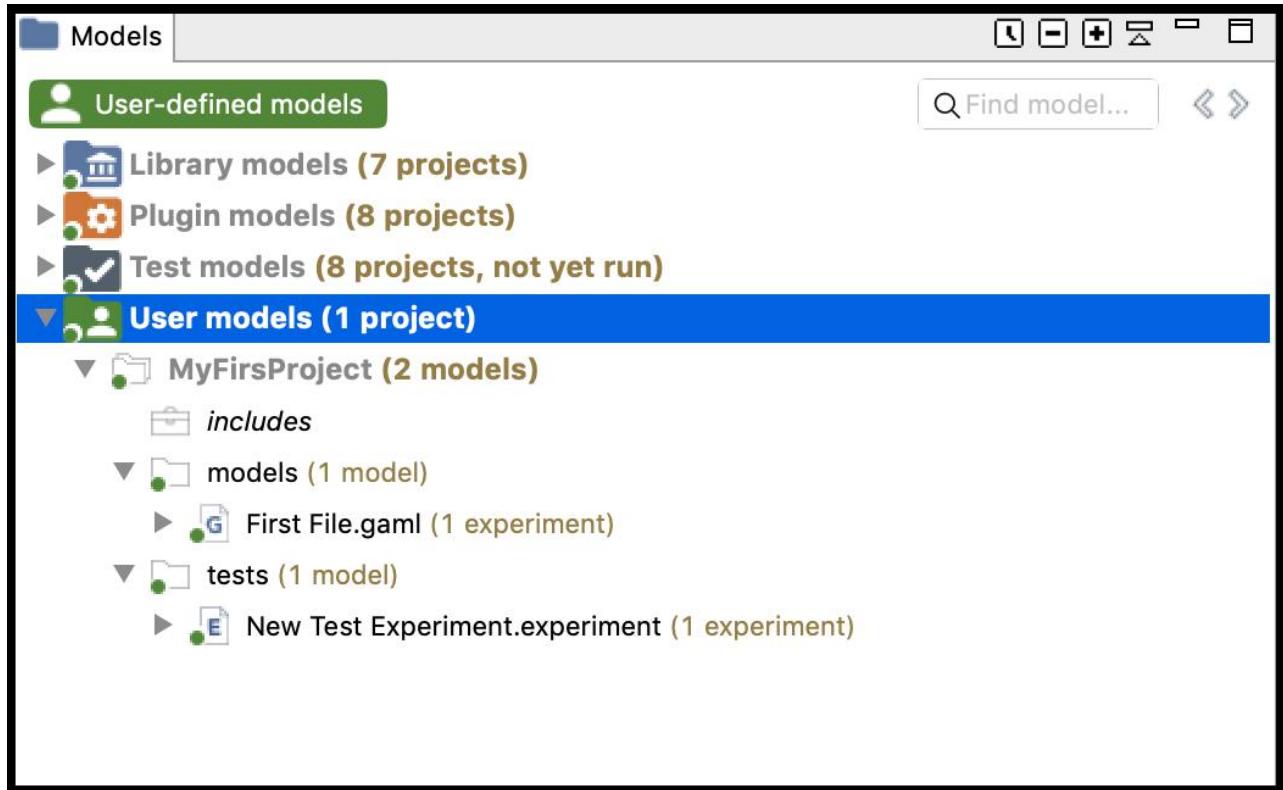
These models are unit tests for the GAML language: they aim at testing each element of the language to check whether they produce the expected result. The aim is to avoid regression after evolutions of the software. [They can be run](#) from the validation view.

User models

This category regroups all the projects that have been [created](#) or [imported](#) in the workspace by the user. Each project can be actually a folder that resides in the folder of the workspace (so they can be easily located from within the filesystem) or a link

to a folder located anywhere in the filesystem (in case of a project importation). Any modification (addition, removal of files...) made to them in the file system (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`. Similarly, all the test models are located in the `tests` folder.



Inspect Models

Each model is presented as a node in the navigation workspace, including *Experiment* buttons and/or a *Contents* node and/or a *Uses* node and/or a *Tags* node and/or an *Imports* node.

The screenshot shows the GamaRIDE interface. On the left, the workspace tree displays various model categories and files. A red box highlights the 'Imports' section under the 'Ant Foraging.gaml' node, which contains a file named 'New Model.gaml'. To the right, the code editor shows the Gaml script for the 'Ant Foraging.gaml' model. The code includes comments and declarations for global variables, utility functions, and simulation parameters.

```

1 /**
2 * Name: Ant Foraging (Complex)
3 * Author:
4 * Description: Toy Model ant using the question of ho
5 * nest once they did find food.
6 * Tags: gui, fsm, grid, diffusion
7 */
8 model ants
9
10 import "New Model.gaml"
11
12@global {
13     //Utilities
14     bool use_icons <- true ;
15     bool display_state <- false;
16     //Evaporation value per cycle
17     float evaporation_per_cycle <- 5.0 min: 0.0 max:
18     //Diffusion rate of the pheromon among the grid
19     float diffusion_rate <- 1.0 min: 0.0 max: 1.0 pa
20     //Size of the grid
21     int gridsize <- 100 min: 30 parameter: 'Width and
22     //Number of ants
23     int ants_number <- 200 min: 1 parameter: 'Number
24     //Frequency of update of the grid
25     int grid_frequency <- 1 min: 1 max: 100 parameter:
26     //Number of food places among the grid
27     int number_of_food_places <- 5 min: 1 parameter:
28     float grid_transparency <- 1.0;
29     image_file ant_shape const: true <- file('..../im
30     svg_file ant_shape_svg const: true <- svg_file('
31     obj_file ant3D_shape const: true <- obj_file('..
32
33     //The center of the grid that will be considered
34     point center const: true <- {round(gridsize / 2),
35     int food_gathered <- 1;
36     int food_placed <- 1;
37     rgb background const: true <- rgb(99, 200, 66);
38     rgb food_color const: true <- rgb(31, 22, 0);

```

- **Imports:** The node *Imports* lists all the model files that are imported in the current model.

The screenshot shows the GamaRIDE interface. On the left, the workspace tree displays 'Modeling (45 models)' and 'Toy Models (81 models)', with 'Ants (Foraging and Sorting) (3 models)' expanded. Inside this folder are 'images' and 'models'. The 'models' folder contains 'Ant Foraging.gaml (4 experiments)'. This experiment folder is highlighted with a yellow rounded rectangle. It contains 'Imports' and 'Uses'. The 'Imports' section lists 'New Model.gaml (New Model.gaml)'. The 'Uses' section lists three files: 'ant.png' (path: './images/ant.png'), 'ant.svg' (path: './images/ant.svg'), and 'soil.jpg' (path: './images/soil.jpg'). On the right, the code editor shows the GAML code for the 'Ant Foraging.gaml' model. An orange arrow points from the 'Uses' section in the workspace tree to the corresponding code in the editor.

```

1  /**
2  * Name: Ant Foraging (Complex)
3  * Author:
4  * Description: Toy Model ant using the
5  * nest once they did find food.
6  * Tags: gui, fsm, grid, diffusion
7 */
8 model ants
9
10 import "New Model.gaml"
11
12 global {
13     //Utilities
14     bool use_icons <- true;
15     bool display_state <- false;
16     //Evaporation value per cycle
17     float evaporation_per_cycle <- 5.0
18     //Diffusion rate of the pheromon am
19     float diffusion_rate <- 1.0 min: 0.

```

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

The screenshot shows the GamaRIDE interface. The workspace tree on the left shows 'Ants (Foraging and Sorting) (2 models)'. The 'Images' folder under this model contains files: 'ant.png', 'ant.svg', 'fire-ant.jpg', 'fire-ant.mtl', 'fire-ant.obj', and 'soil.jpg'. A blue arrow points from the 'ant.svg' file in the workspace tree to its declaration in the code editor. Another blue arrow points from the 'fire-ant.mtl' file in the workspace tree to its declaration in the code editor. The code editor on the right shows the GAML code for the 'Ant Foraging.gaml' model. Two specific lines of code are highlighted with blue boxes: 'image_file ant_shape const: true <- file("../images/ant.png");' and 'image_file terrain <- image_file("../images/soil.jpg");'. These lines correspond to the declarations in the workspace tree.

```

14 //Evaporation value per cycle
15 float evaporation_per_cycle <- 5.0 min: 0.0 max: 240.0 parameter: 'Evaporation of the signal (unit/cycle):' category: 'Environment and Population';
16 //Diffusion rate of the pheromon among the grid
17 float diffusion_rate <- 1.0 min: 0.0 max: 1.0 parameter: 'Rate of diffusion of the signal (%/cycle):' category: 'Environment and Population';
18 //Size of the grid
19 int gridSize <- 100 min: 30 parameter: 'Width and Height of the grid:' category: 'Environment and Population';
20 //Number of ants
21 int ants_number <- 200 min: 1 parameter: 'Number of ants:' category: 'Environment and Population';
22 //Frequency of update of the grid
23 int grid_frequency <- 1 min: 1 max: 100 parameter: 'Grid updates itself every:' category: 'Environment and Population';
24 //Number of food places among the grid
25 int number_of_food_places <- 5 min: 1 parameter: 'Number of food depots:' category: 'Environment and Population';
26
27 image_file ant_shape const: true <- file("../images/ant.png");
28 svg_file ant_shape_svg const: true <- svg_file("../images/ant.svg");
29
30 //The center of the grid that will be considered as the nest location
31 point center const: true <- {round(gridsize / 2), round(gridsize / 2)};
32 int food_gathered <- 1;
33 int food_placed <- 1;
34 int food_background const: true <- rgb(99, 200, 66);
35 int food_color const: true <- rgb(31, 22, 0);
36 int rgb nest_color const: true <- rgb(6, 0, 0);
37
38
39 image_file terrain <- image_file("../images/soil.jpg");
40
41
42 init {
43
44

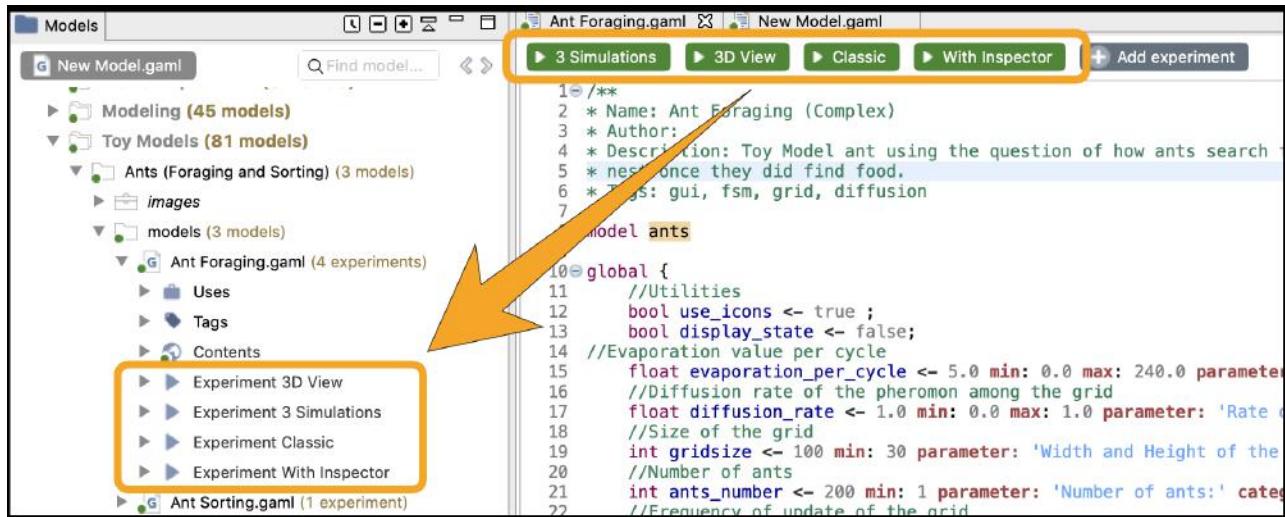
```

- **Tags node:** The node *Tags* lists all the tags that have been specified in the header of the model.

The screenshot shows the NetLogo interface. On the left, the 'Models' tab is selected in the top navigation bar. Below it, the 'Ant Foraging.gaml' model is selected in the 'Ant Foraging.gaml' section of the Model Explorer. A red box highlights the 'Tags' node under the model's experiments. To the right, the code editor displays the GAML code for the 'Ant Foraging (Complex)' model. A red arrow points from the 'Tags' node in the Model Explorer to the 'Tags' line in the code editor, which contains the values 'gui, fsm, grid, diffusion'. The code editor also includes tabs for '3 Simulations', '3D View', 'Classic', 'With Inspector', and 'Add experiment'.

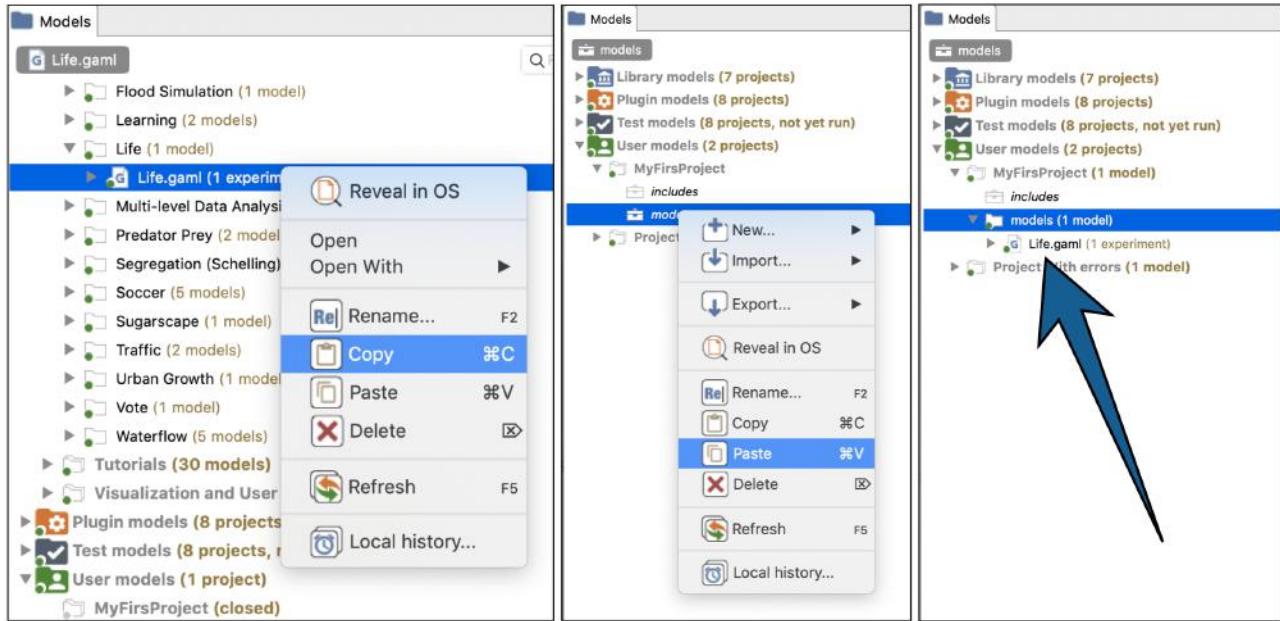
```
1 *//  
2 * Name: Ant Foraging (Complex)  
3 * Author:  
4 * Description: Toy Model ant using the question of how ants search food and use pheromons to  
5 * not once they did find food  
6 * Tags: gui, fsm, grid, diffusion  
7 */  
8 model ants  
9  
10 global {  
11   //Utilities
```

- **Contents:** The node *Contents* describes the tree of all the elements in the model. It is similar to the Overview view.
 - **Experiment button :** Experiment buttons are present if your model contains experiments (it is usually the case !). To run the corresponding experiment, just click on it. To learn more about running experiments, jump into this [section](#).



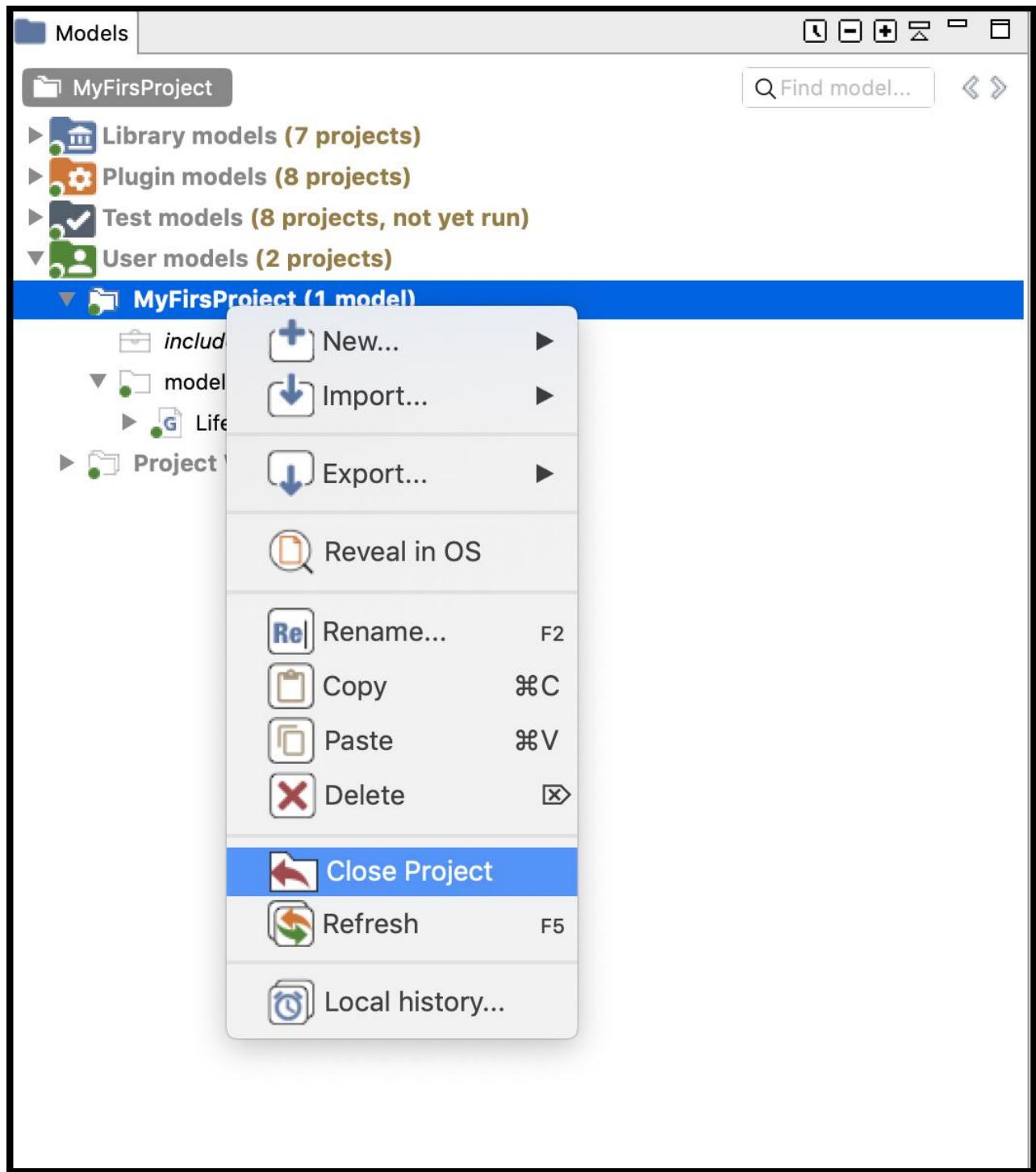
Moving Models Around

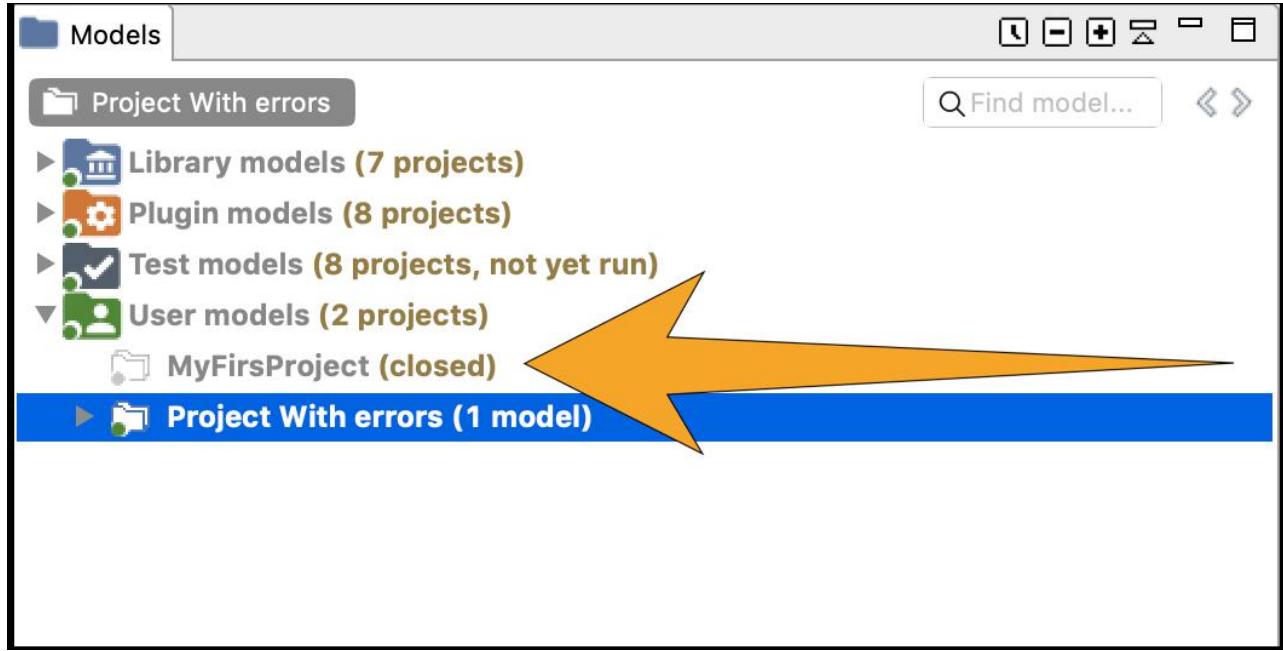
Model files, as well as resources, or even complete projects, can be moved around between the "Models Library"/"Plugin Models" and "Users Models" categories, or within them, directly in the *Navigator*. **Drag'n drop operations are supported**, as well as copy and paste. For example, the model `Life.gaml`,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



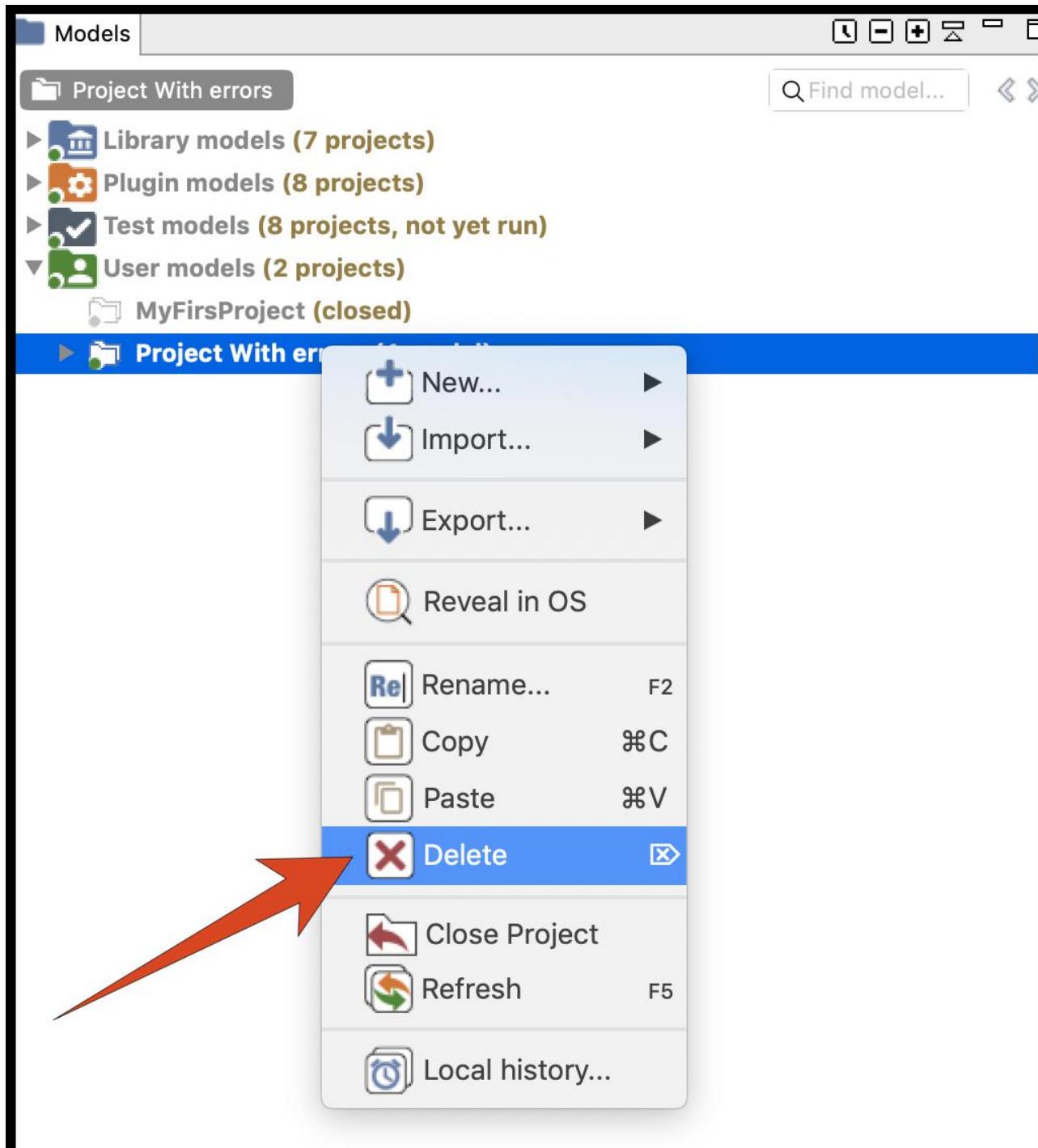
Closing and Deleting Projects

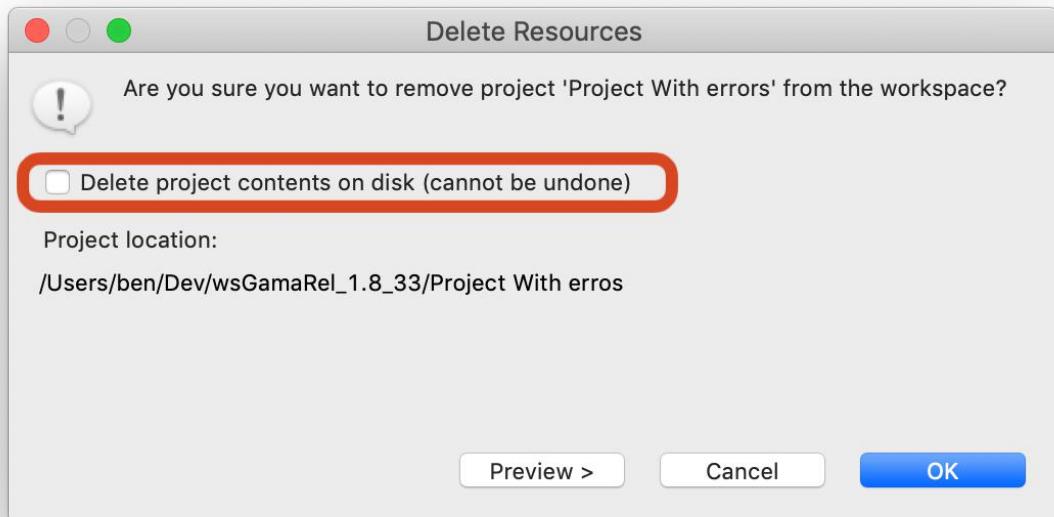
Users can choose to get rid of old projects by either **closing** or **deleting** them. Closing a project means that it will still reside in the workspace (and be still visible, although a bit differently, in the *Navigator*) but its model(s) won't participate to the build process and won't be displayable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





Version: 1.9.1

Changing Workspace

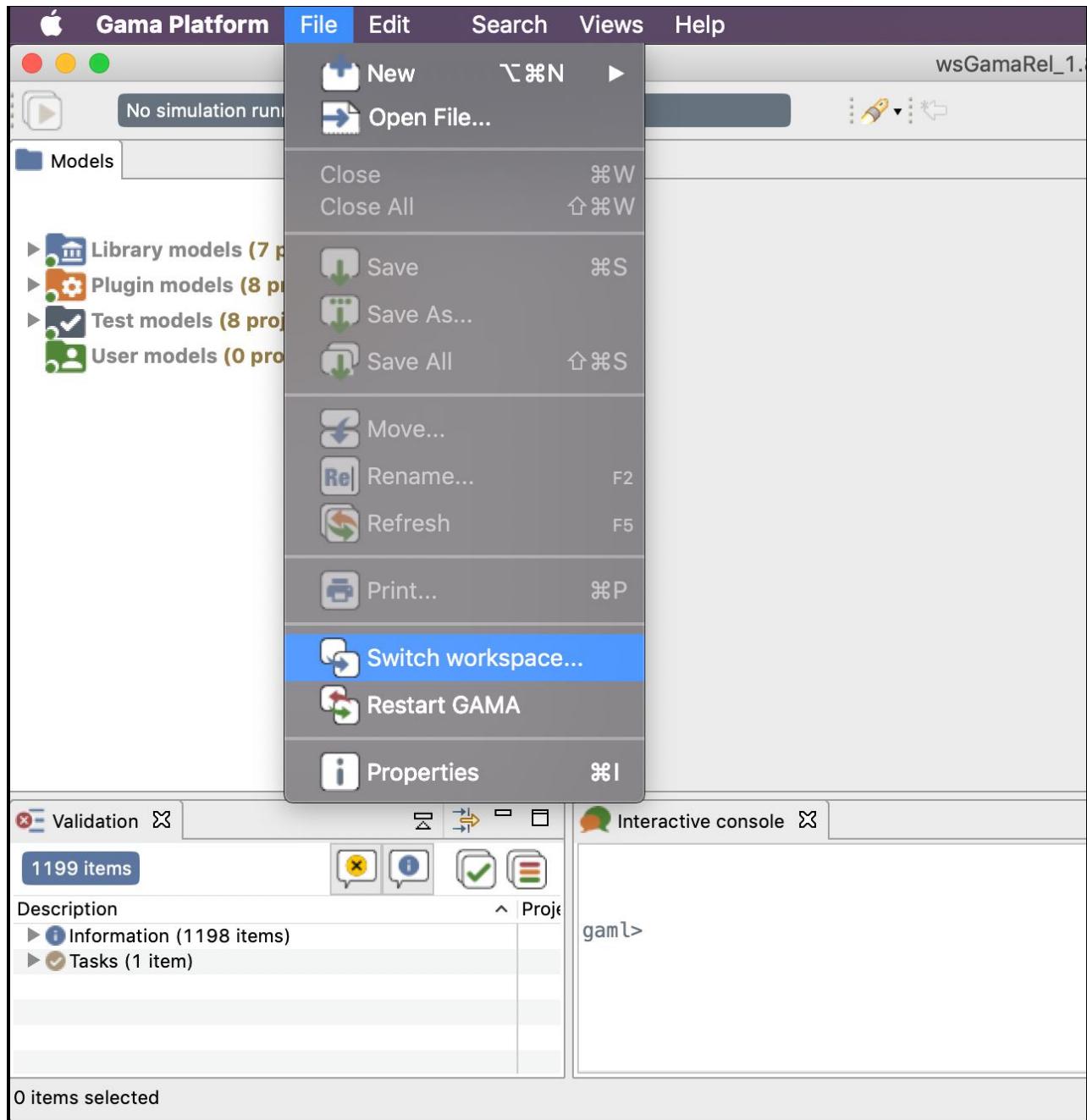
It is possible, and actually common, to store different projects/models in different workspaces and to tell GAMA to switch between these workspaces. Doing so involves being able to create one or several new workspace locations (even if GAMA has been told to [remember](#) the current one) and being able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

Table of contents

- [Changing Workspace](#)
 - [Switching to another Workspace](#)
 - [Cloning the Current Workspace](#)

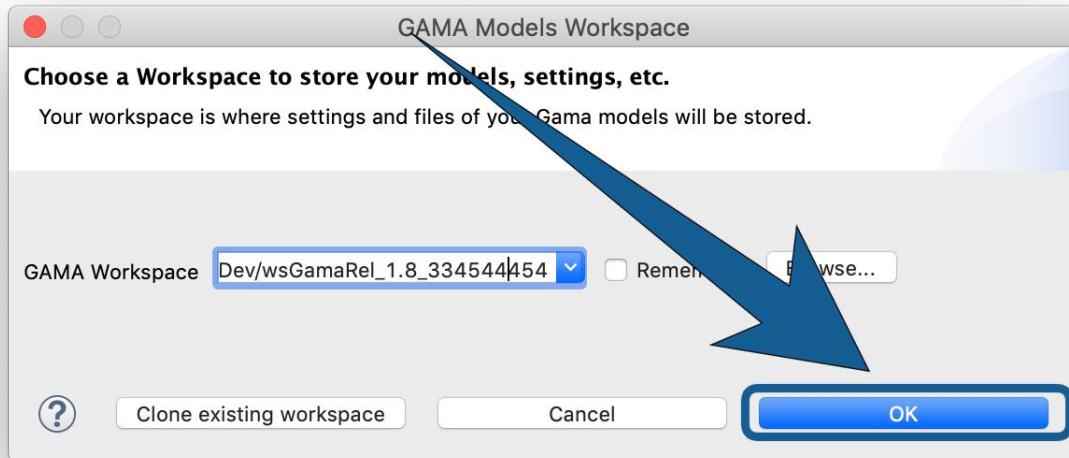
Switching to another Workspace

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).



Notice that, when GAMA restarts and that you have not ticked "Remember workspace", GAMA will ask you again the workspace (just as when you launch GAMA).

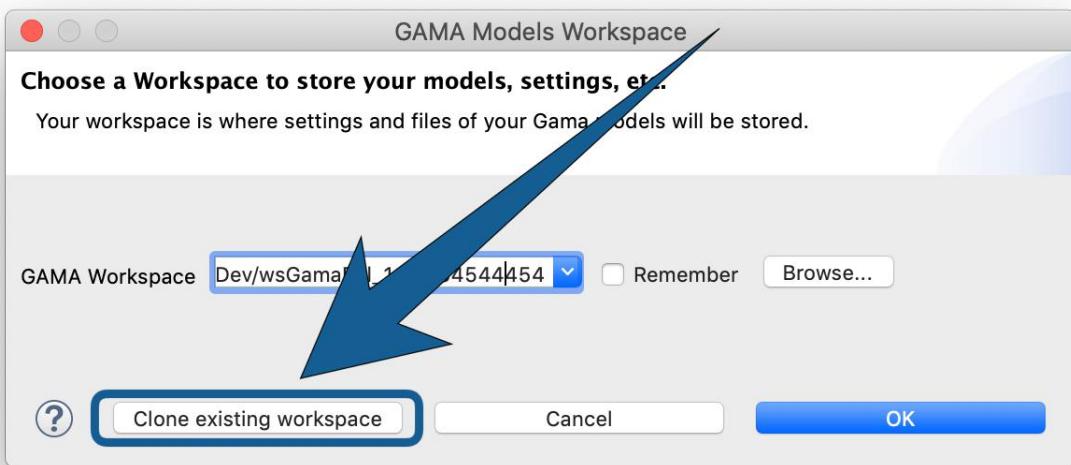
Cloning the Current Workspace

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 projects/

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 having different workspaces "containing" the same project)

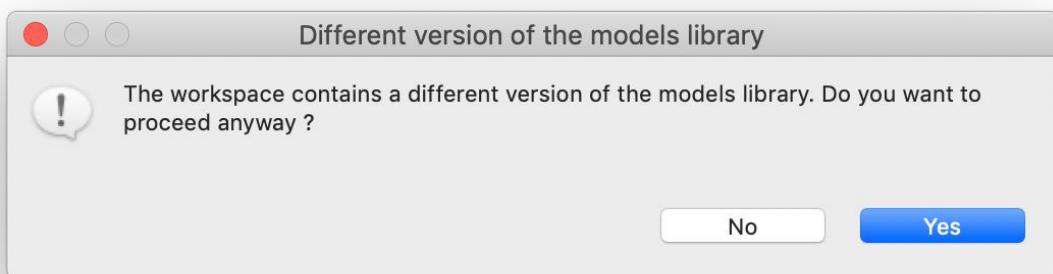
This can be done by entering the new workspace location and choosing "Clone current workspace" in the previous dialog instead of "Ok".



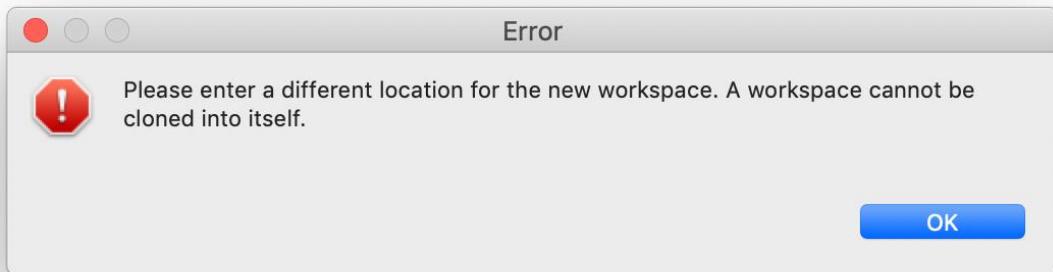
If the new location does not exist, GAMA will ask you to confirm the creation and cloning using a specific dialog box. Dismissing it will cancel the operation.



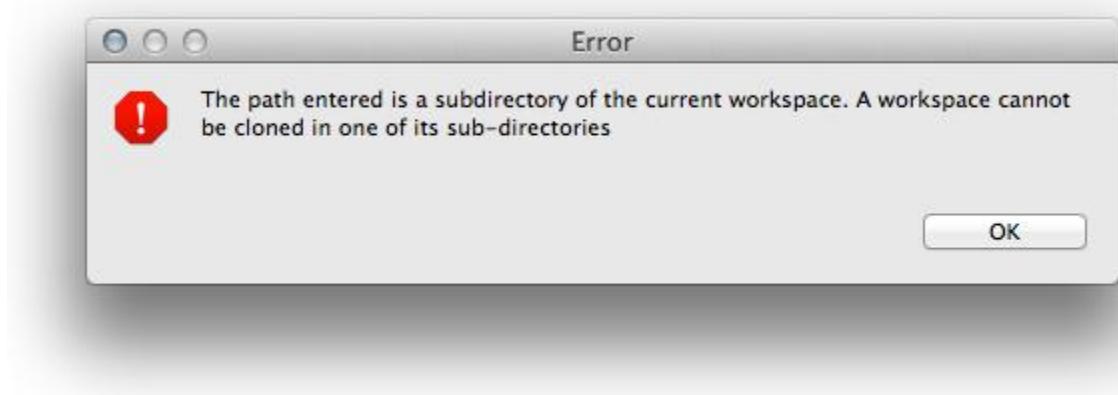
If the new location is already the location of an existing workspace, another confirmation dialog is produced. **It is important to note that all projects in the target workspace will be erased and replaced by the projects in the current workspace if you proceed.** Dismissing it will cancel the operation.



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).



Version: 1.9.1

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 a Git repository, 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 simplest, safest and most secure way to import a project into the workspace is to follow [instructions from this section](#).

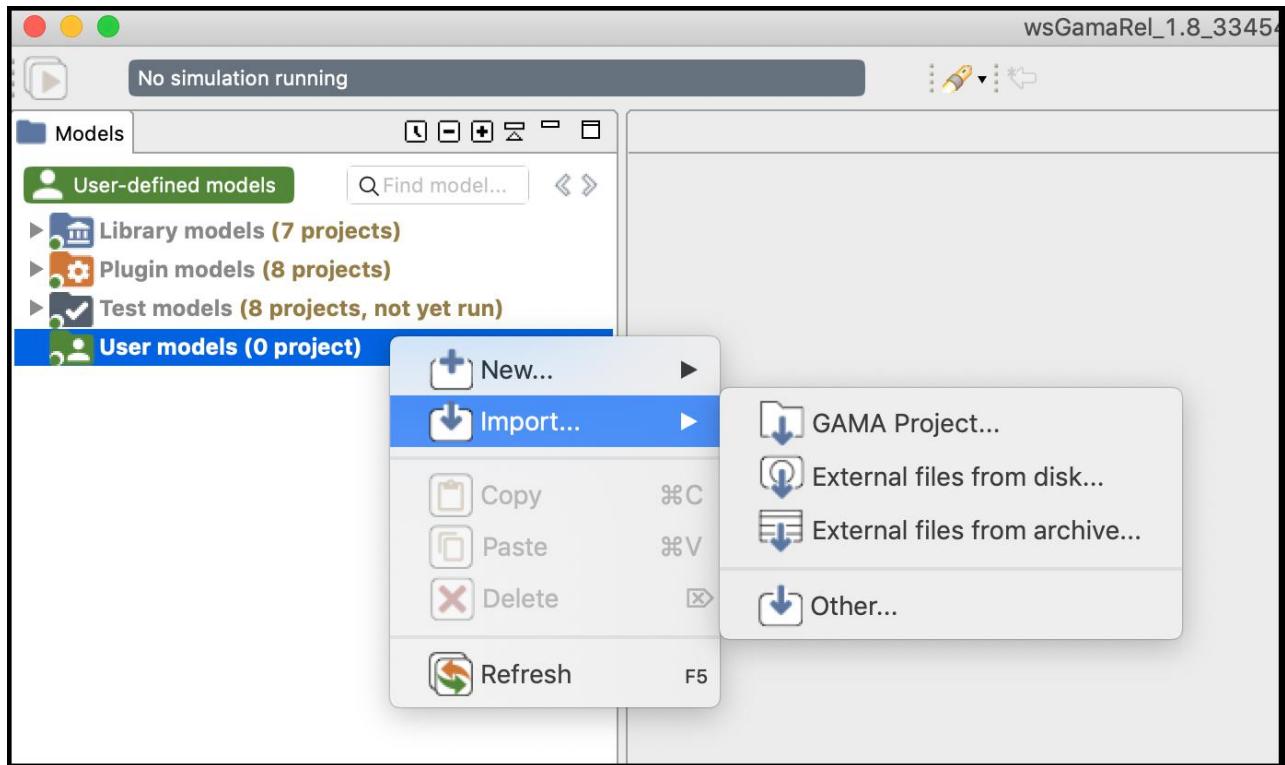
Table of contents

- Importing Models

- The "Import..." Menu Command
 - Import "GAMA Project..."
 - Import "External files from disk..." and "External files from archive..."
 - "Other" imports
- Silent import
- Drag'n Drop / Copy-Paste Limitations
- Import from GitHub repository

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 contextual menu on the *User models* (the modeler can only import projects in this category).



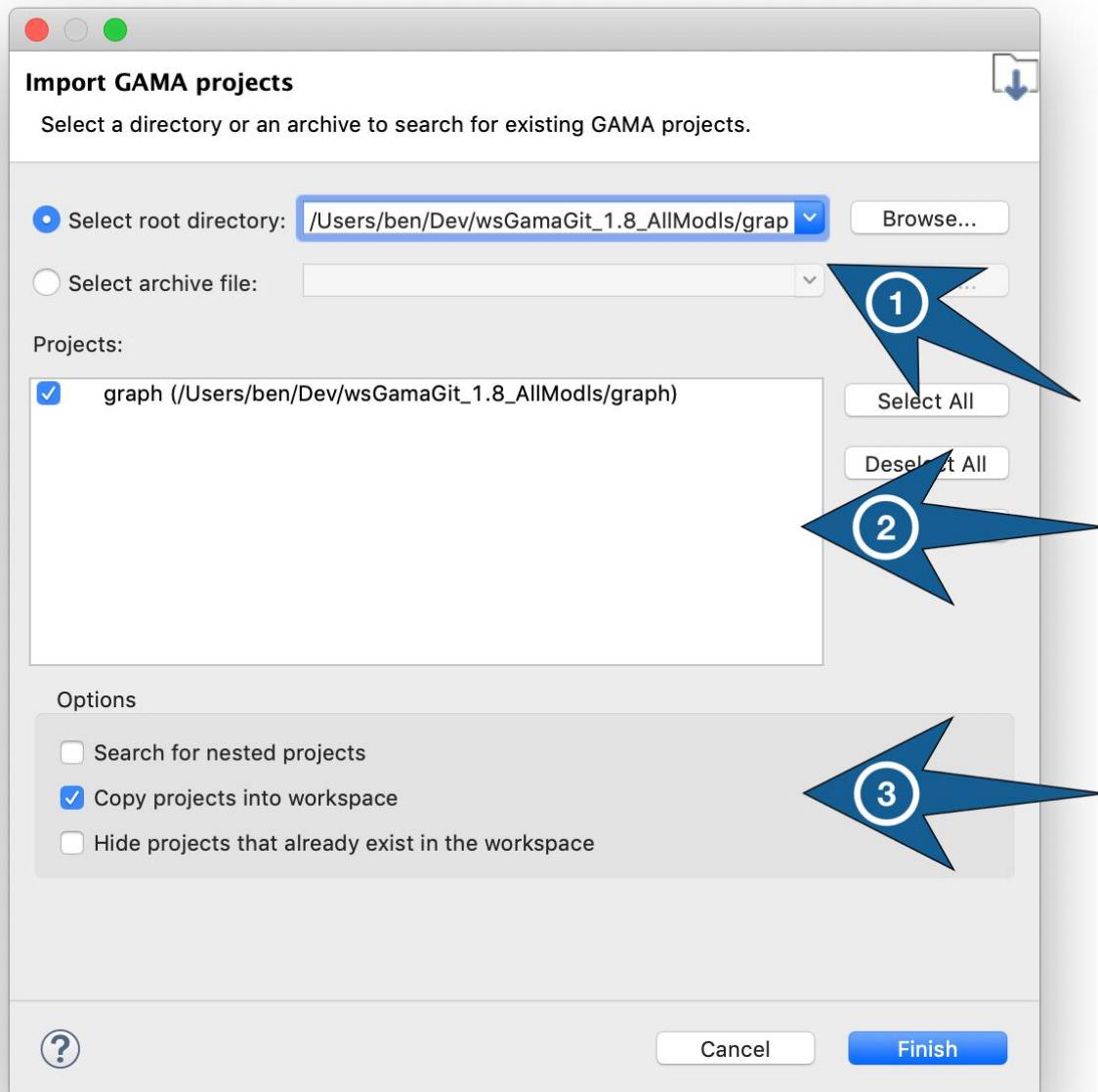
The "Import..." command allows the modeler to choose between:

- "GAMA Project...": **import a project** in the workspace (from another folder or an archive),
- "External files from disk...": **import any files** in a project of the workspace (from a folder),
- "External files from archive...": **import any files** in a project of the workspace (from an archive),
- "Other": other ways of importation.

Import "GAMA Project..."

When "GAMA project..." is chosen, a dialog box will pop-up where the user will be asked to:

1. Enter a location (or browse to a location) containing the GAMA project(s) to import. This can be the folder of a single project or a folder containing several projects. 2 possibilities are available:
 - "Select root directory": the user selects a **folder** containing the project,
 - "Select archive file": the user selects an archive file (e.g. a **.zip** file) containing the project.
2. Choose among the list of available projects (computed by GAMA) the ones to effectively import. Only projects that are not already in the workspace can be imported.
3. Indicate whether or not these projects need to be **copied to** or **linked from** the workspace (the latter is done by default). In the case of an import from an archive, the content will be automatically copied in the workspace.

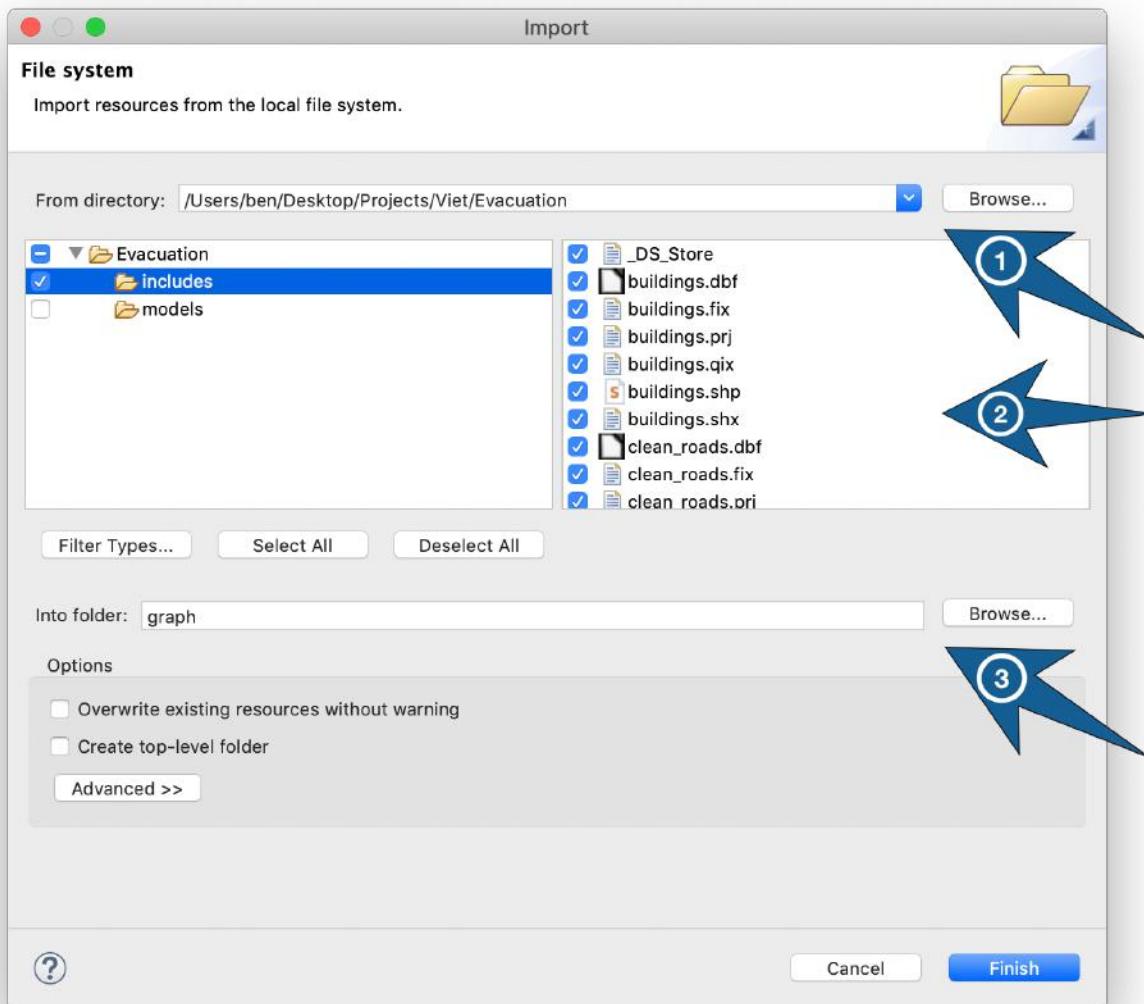


Import "External files from disk..." and "External files from archive..."

These two commands allow the user to **import some external files into an existing project of the workspace**. These two commands are very similar, only the source of

files is different: a folder or an archive. They allow to filter and select the files to import. The user will be asked to:

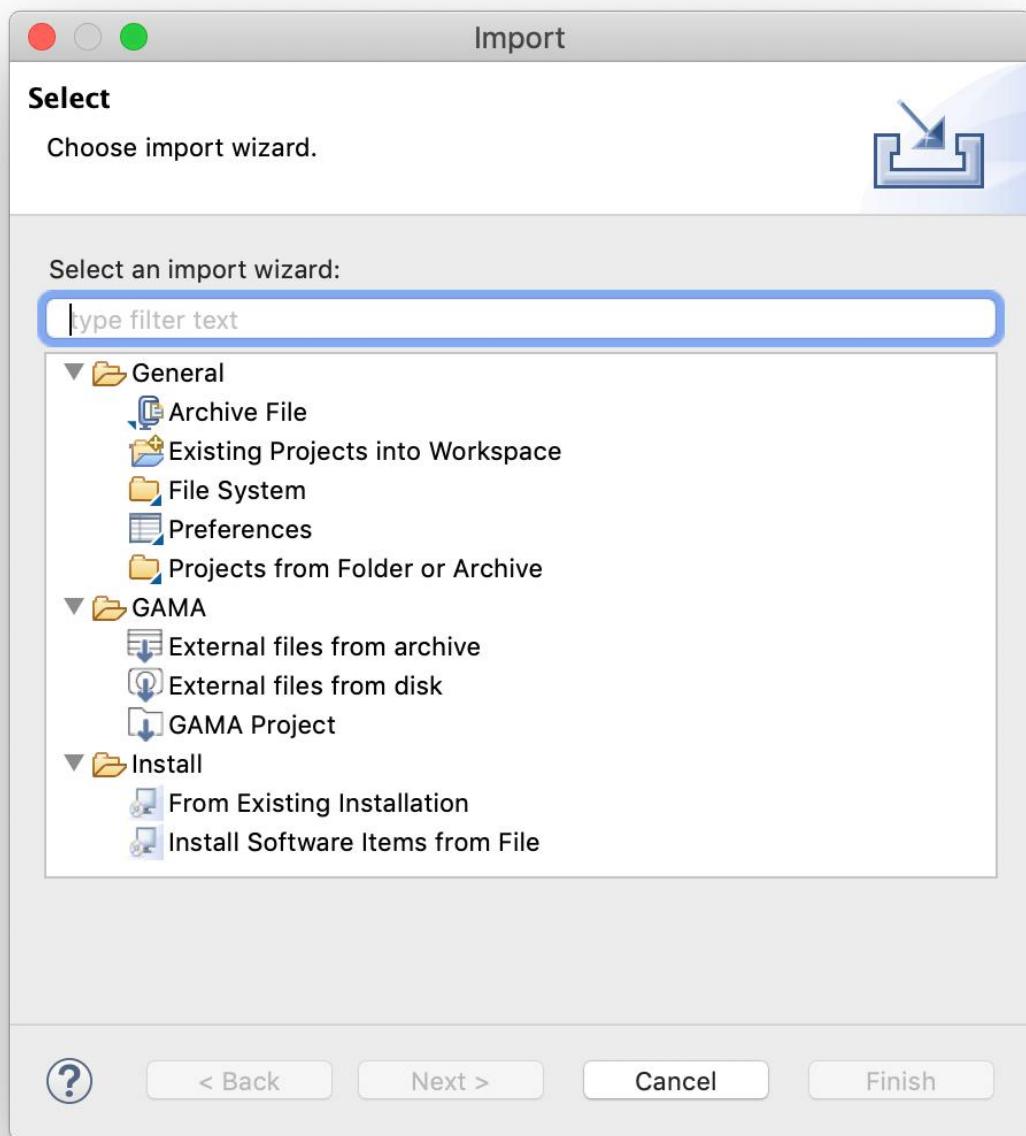
1. Enter a location (or browse to a location) containing the files to import.
2. Select the files to import.
3. Select the project in the workspace where the files will be copied.



"Other" imports

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, etc. It is safer, in any case, to choose "Existing Projects into Workspace".

If some extensions have been installed, they could add some entries in this menu (e.g. the [Git extension](#)).



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 sometime in the future, however, allowing users to use the *Navigator* as a project drop or paste target, but it is not the case yet.

Import from GitHub repository

In the case where the [Git plugin is installed in GAMA](#), projects can be imported from a Git repository, as detailed in [the recipes related to the use of Git in GAMA](#).

Version: 1.9.1

Editing models

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

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

This section covers the following sub-sections:

1. [GAML Editor Generalities](#)
2. [GAML Editor Toolbar](#)
3. [Validation of Models](#)
4. [Graphical Editor](#)

Version: 1.9.1

The GAML Editor - Generalities

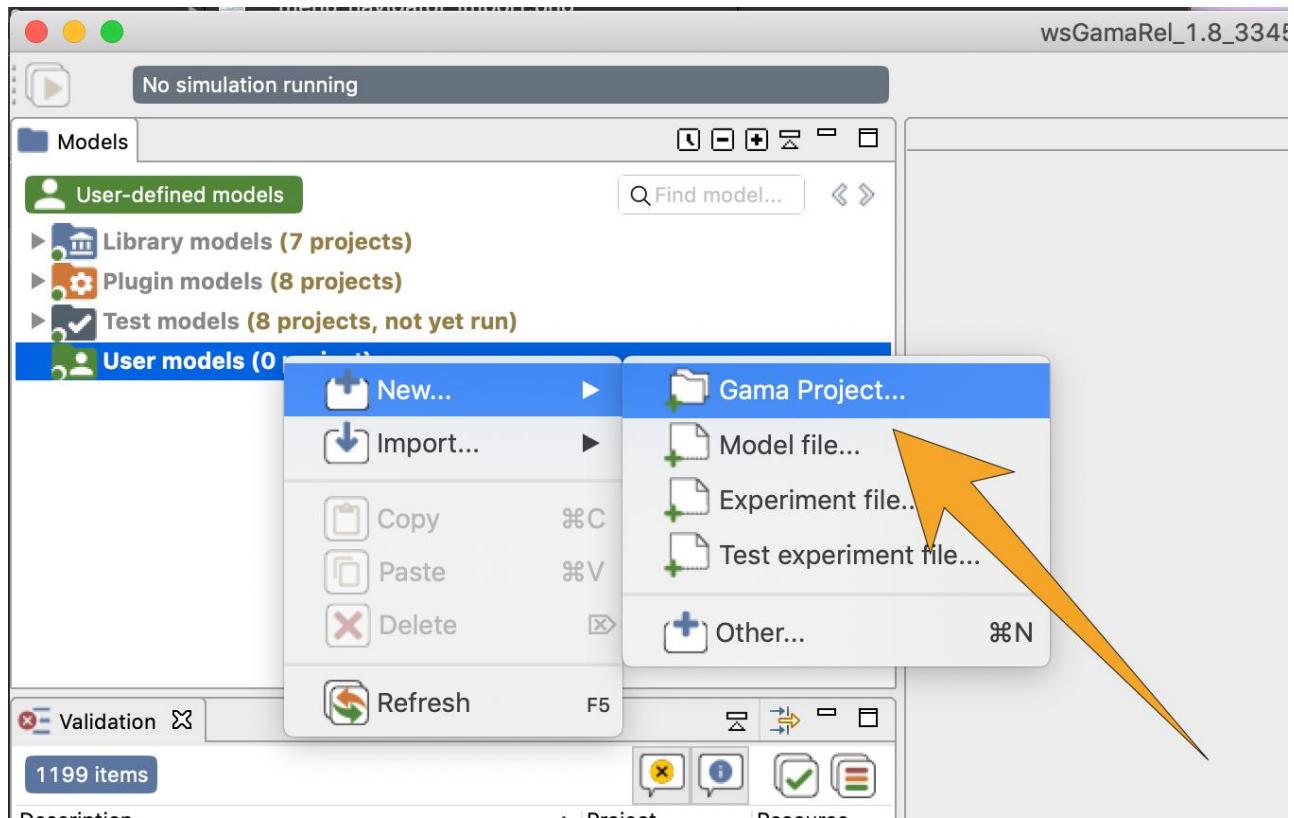
The GAML Editor is a text editor that proposes several services to support the modeler in writing correct models: an integrated live validation system, a ribbon header that gives access to [experiments](#), information, warning and error markers.

Table of contents

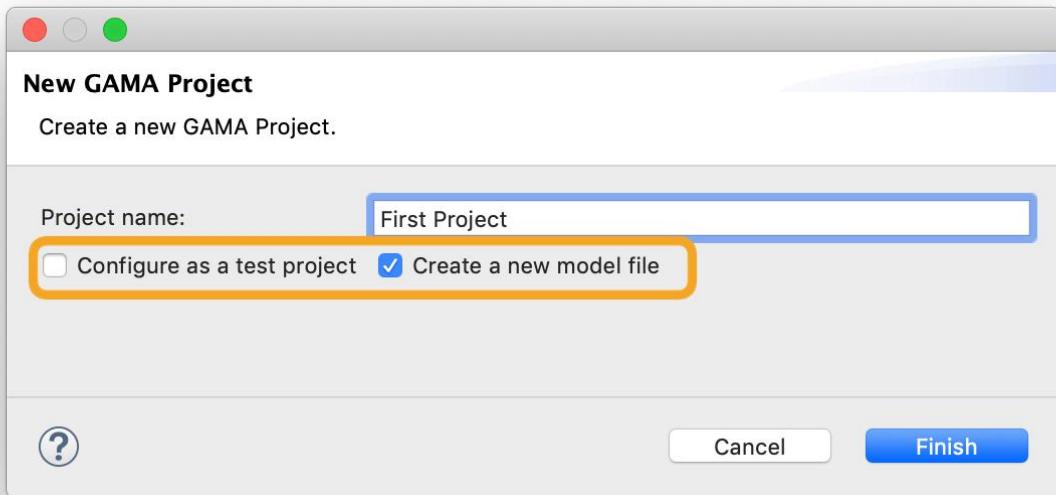
- [The GAML Editor - Generalities](#)
 - [Creating a first model](#)
 - [Create a new model or test file](#)
 - [Create a new experiment file](#)
 - [Status of models in editors](#)
 - [Editor Preferences](#)
 - [Additional information in the Editor](#)
 - [Multiple editors](#)
 - [Local history](#)

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).

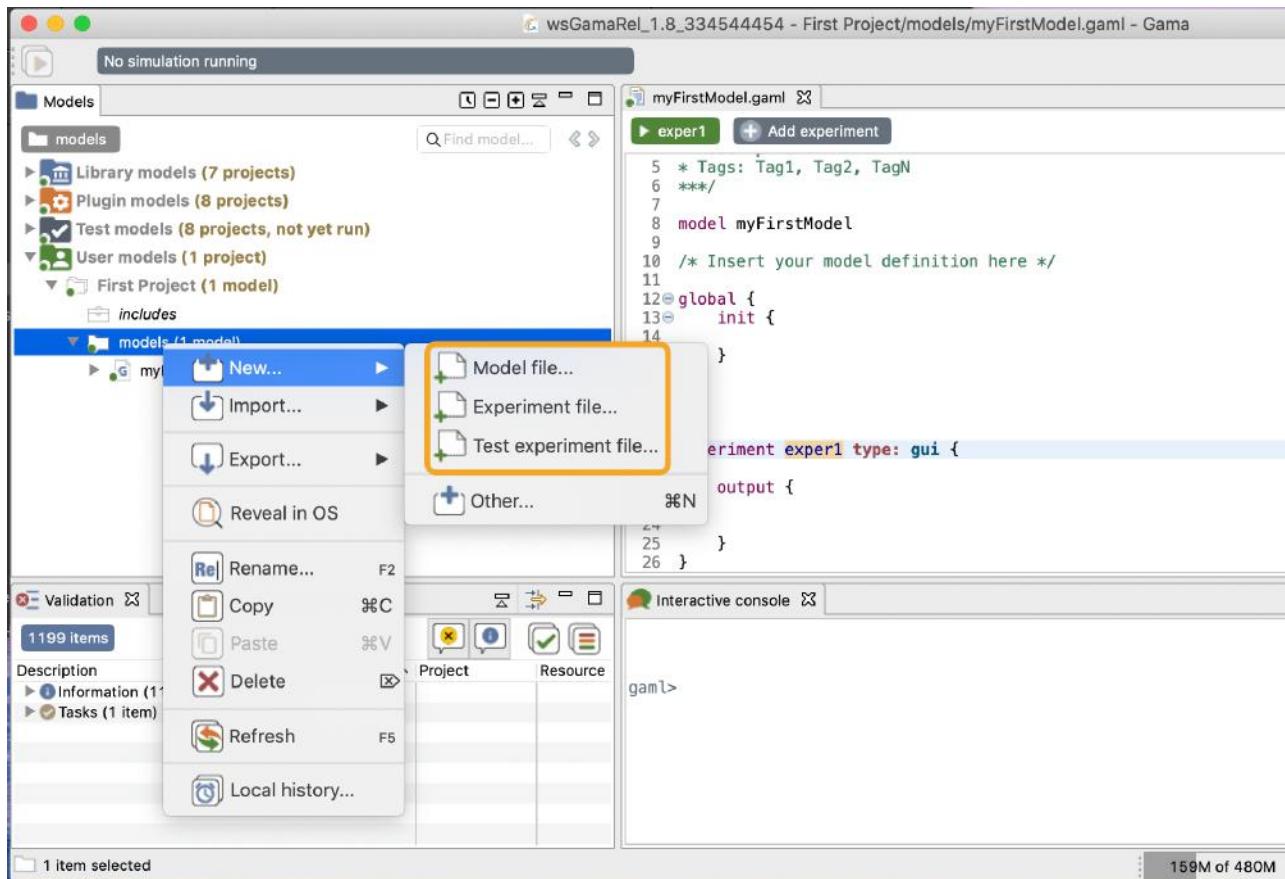


A dialog is then displayed, offering you to enter the name of the project. You can also choose whether you want to create at the same time a first model file and if you want the project contains **test models**. An error will be displayed if the project name already exists in the workspace, in which case you will have to change it. Two projects with similar names cannot coexist in the workspace (even if they belong to different categories).



If you want to create a new model file in your project, navigate to it and right-click on it and on the command "New ...>". You have a choice between three kinds of file:

- **Model file:** to create a normal `.gaml` **model file used to define your model**.
- **Experiment file:** to create a file containing only an experiment on an existing model.
- **Test experiment file:** to create [unit test experiment](#). It is typically used to define some unit test on a given model, to ensure its quality and prevent regression bugs.

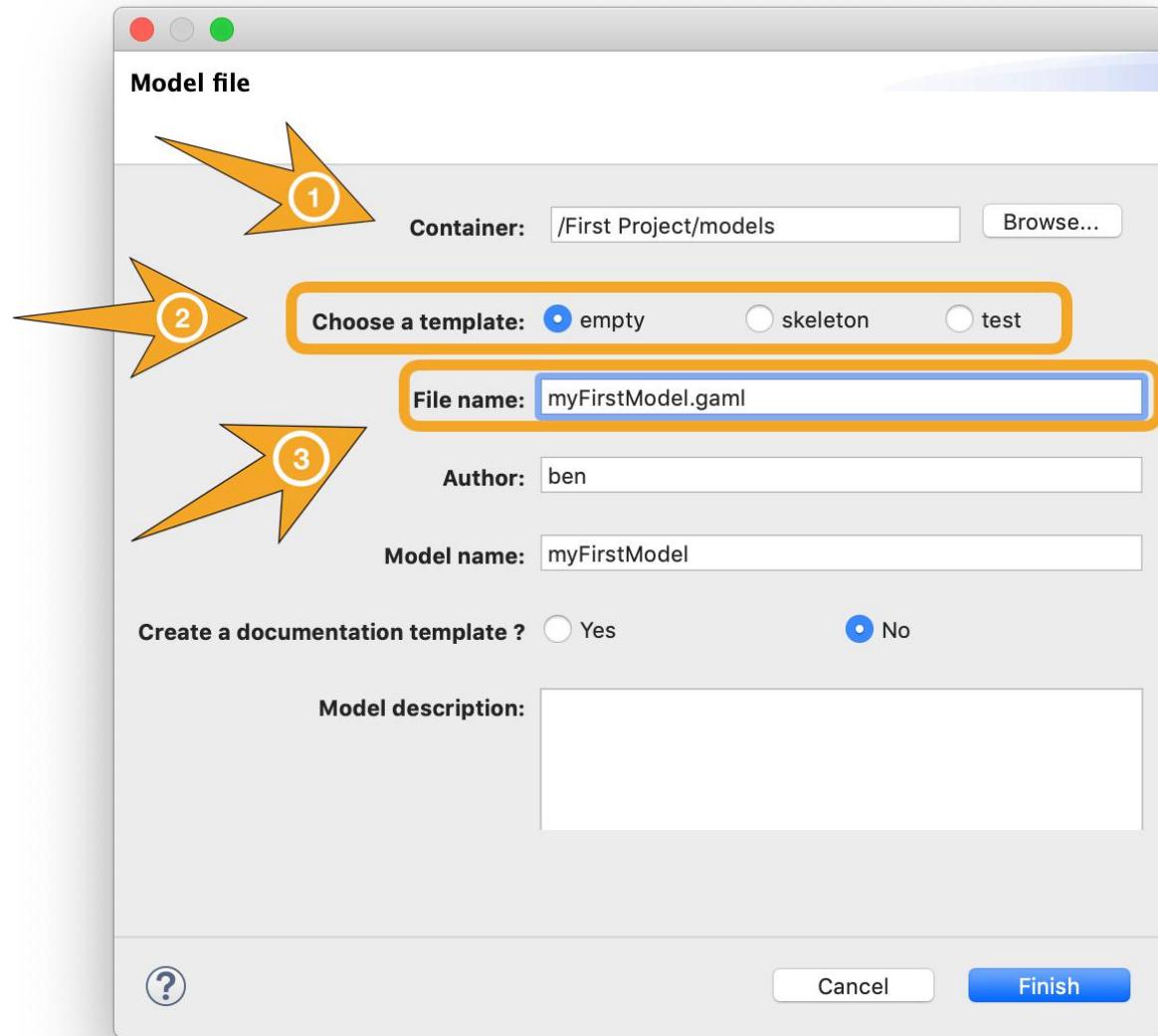


Create a new model or test file

A new dialog is displayed, which asks for several required or optional information:

1. 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.
2. You can then choose whether you want to create an *empty* file, a file with already a *skeleton* of model (with [the main needed elements of a model file](#)) or simply a test model.
3. Finally, 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

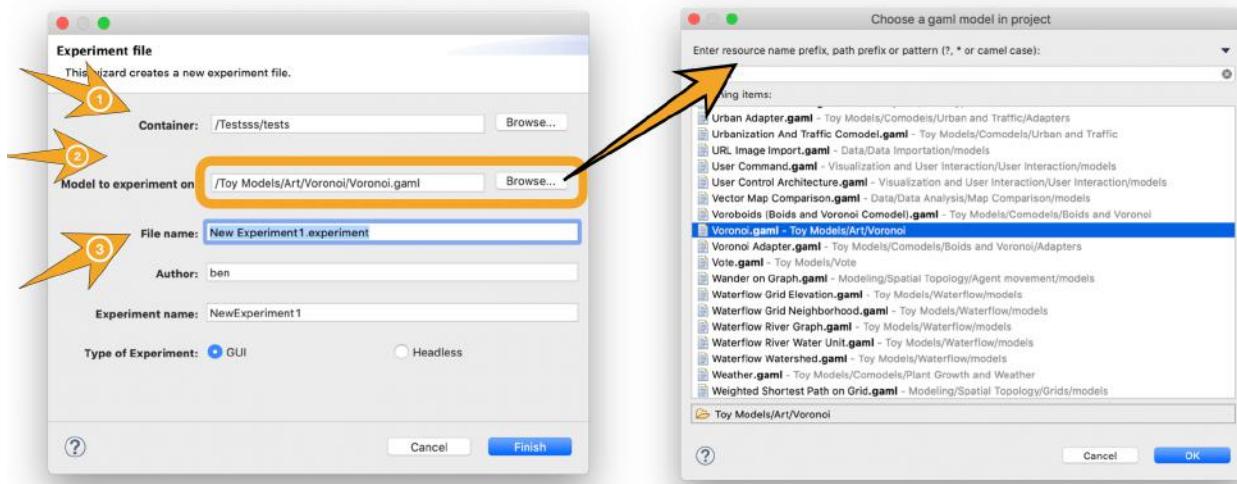
name of the author, as well as the textual description of the model and the creation of an HTML documentation, are optional.



Create a new experiment file

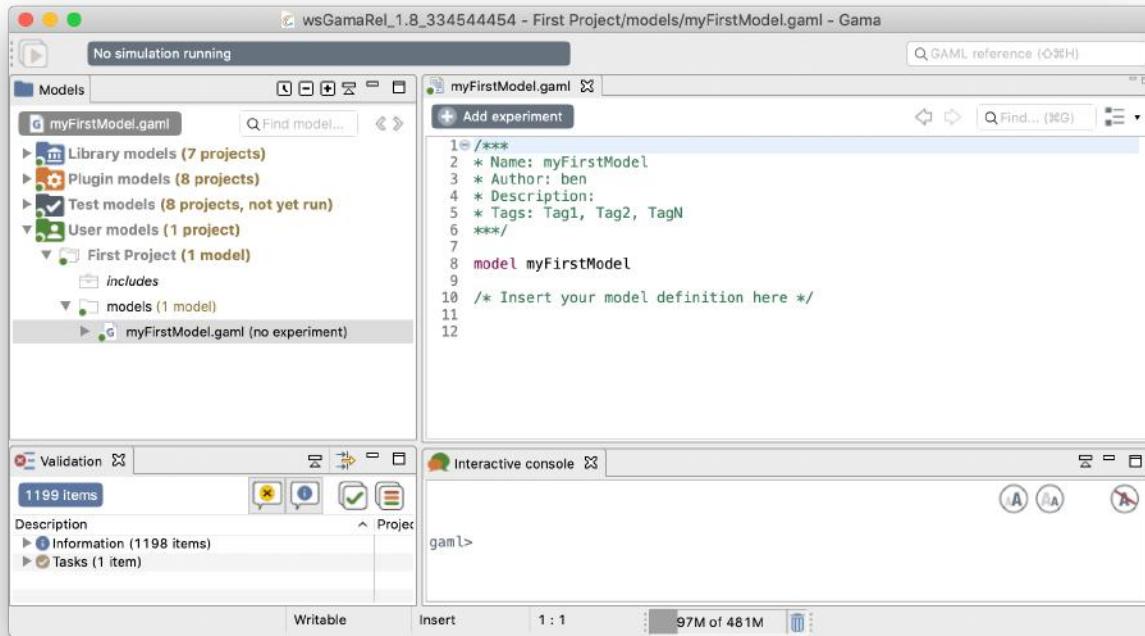
A new dialog is displayed, which asks for several required or optional information:

1. 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.
2. You can then choose the model you want to experiment on. Just type the path toward this *gaml* model, or browse and select one among all the models existing in the workspace.
3. Finally, 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.



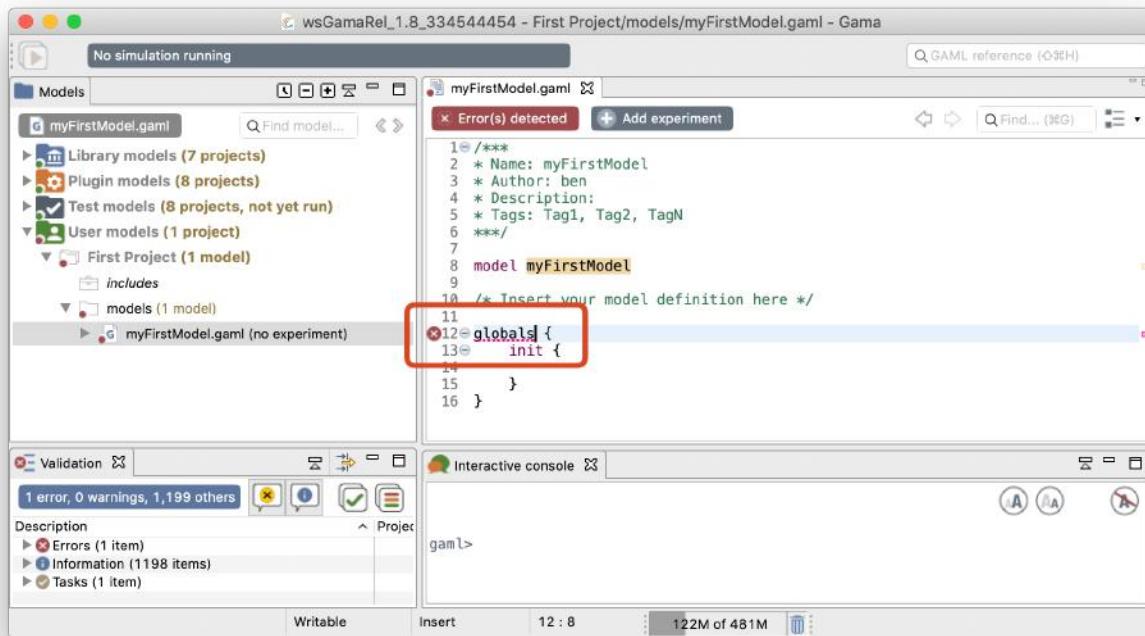
Status of models in editors

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

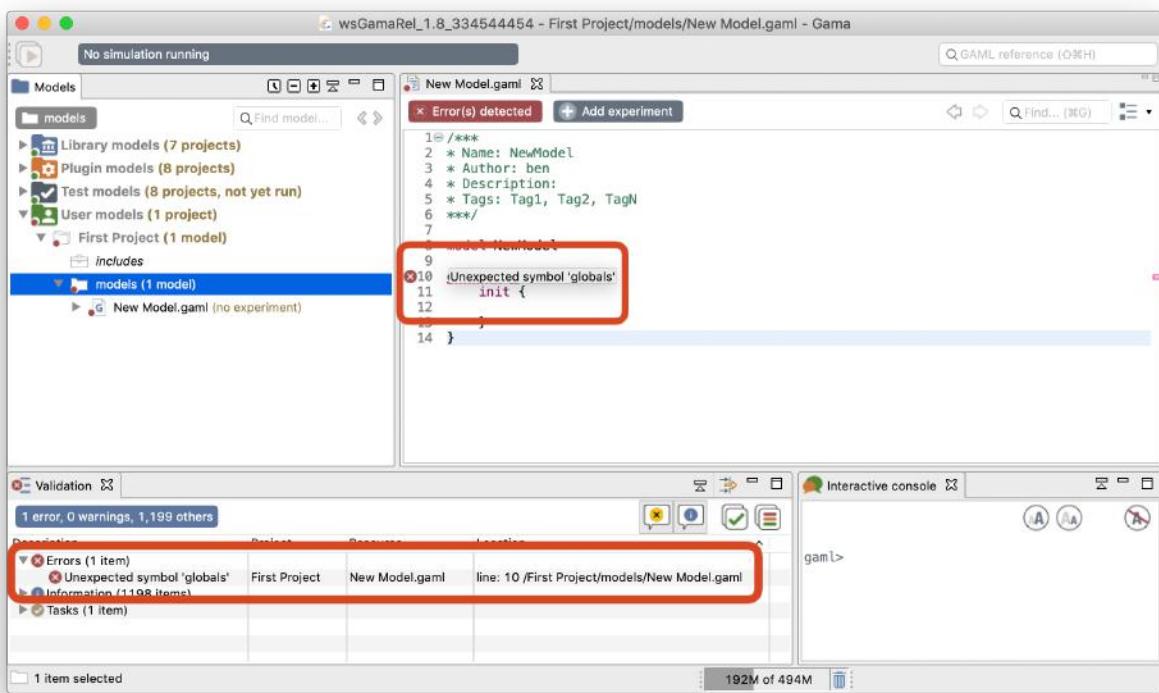


Although GAML files are just plain text files, and can, therefore, be produced or modified in any text processor, using the dedicated GAML editor offers many advantages, among which the live display of errors and model statuses. A model can actually be in four different states, which are visually accessible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

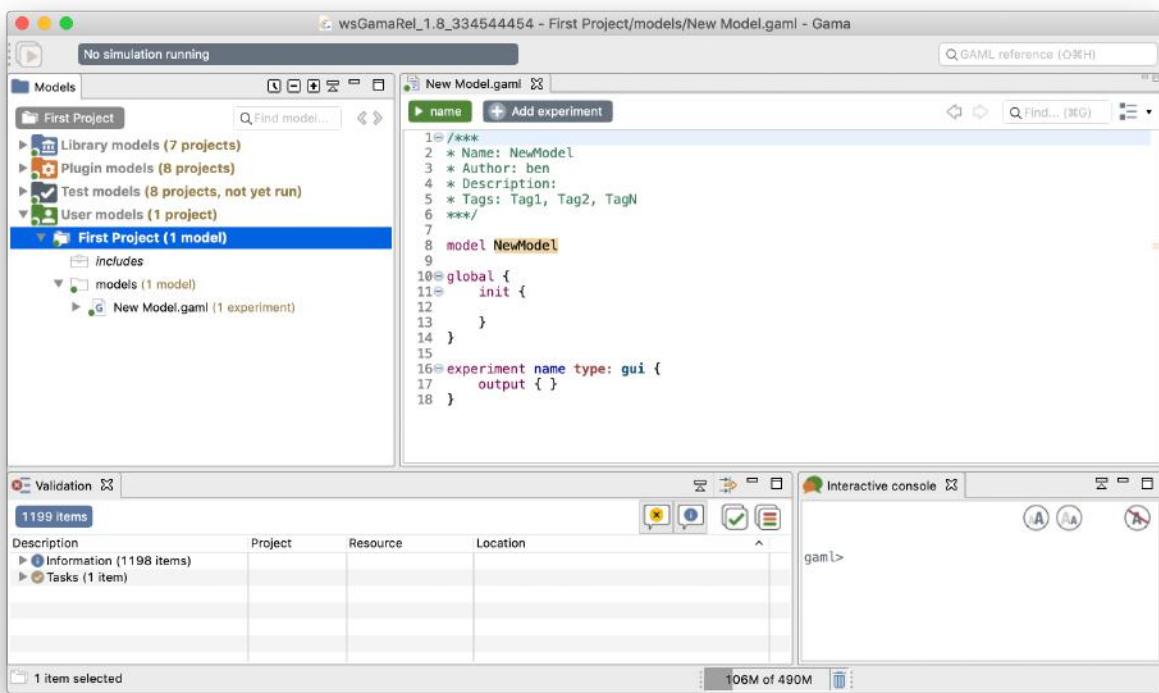
In its initial state, a model is always in the *Functional* state (when it is empty), which means it compiles without problems, but cannot be used to launch experiments. If the model is created with a skeleton, it is *Experimentable*. The *InError* state, depicted below, occurs when the file contains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



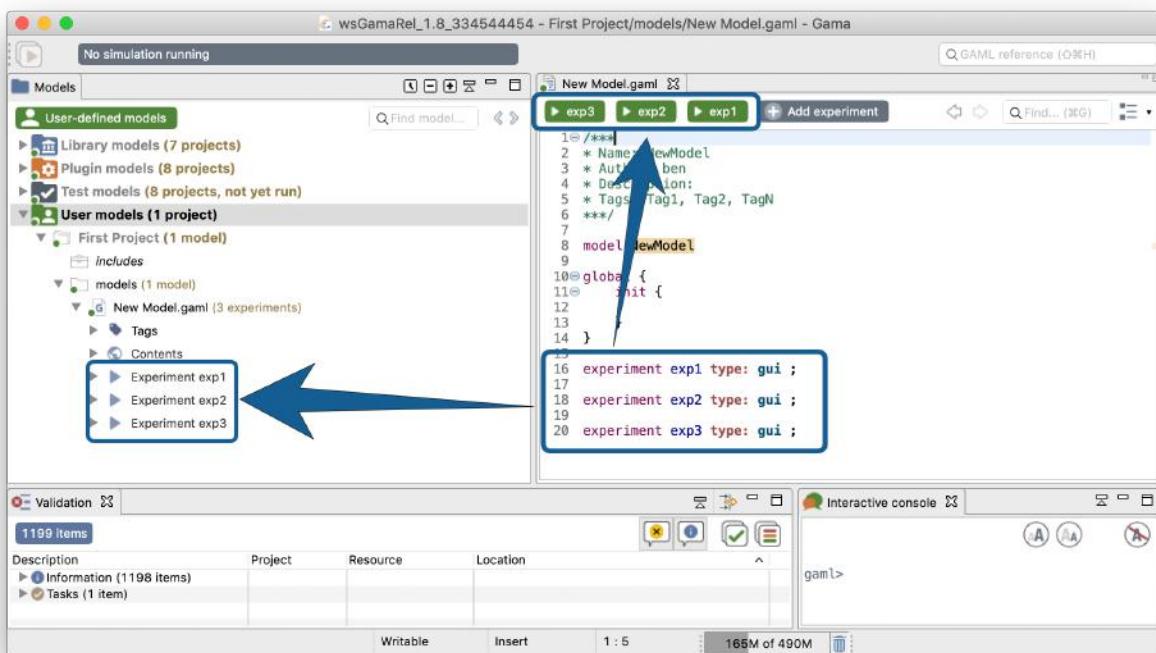
While the file is not saved, these errors remain displayed in the editor and nowhere else. If you save the file, they are now considered as "workspace errors" and get displayed in the "Syntax errors" view below the editor and explanation is available on the error icon in the GAML editor.



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 is in your code. If more than one experiment is defined, corresponding buttons will be displayed in addition to the first one.



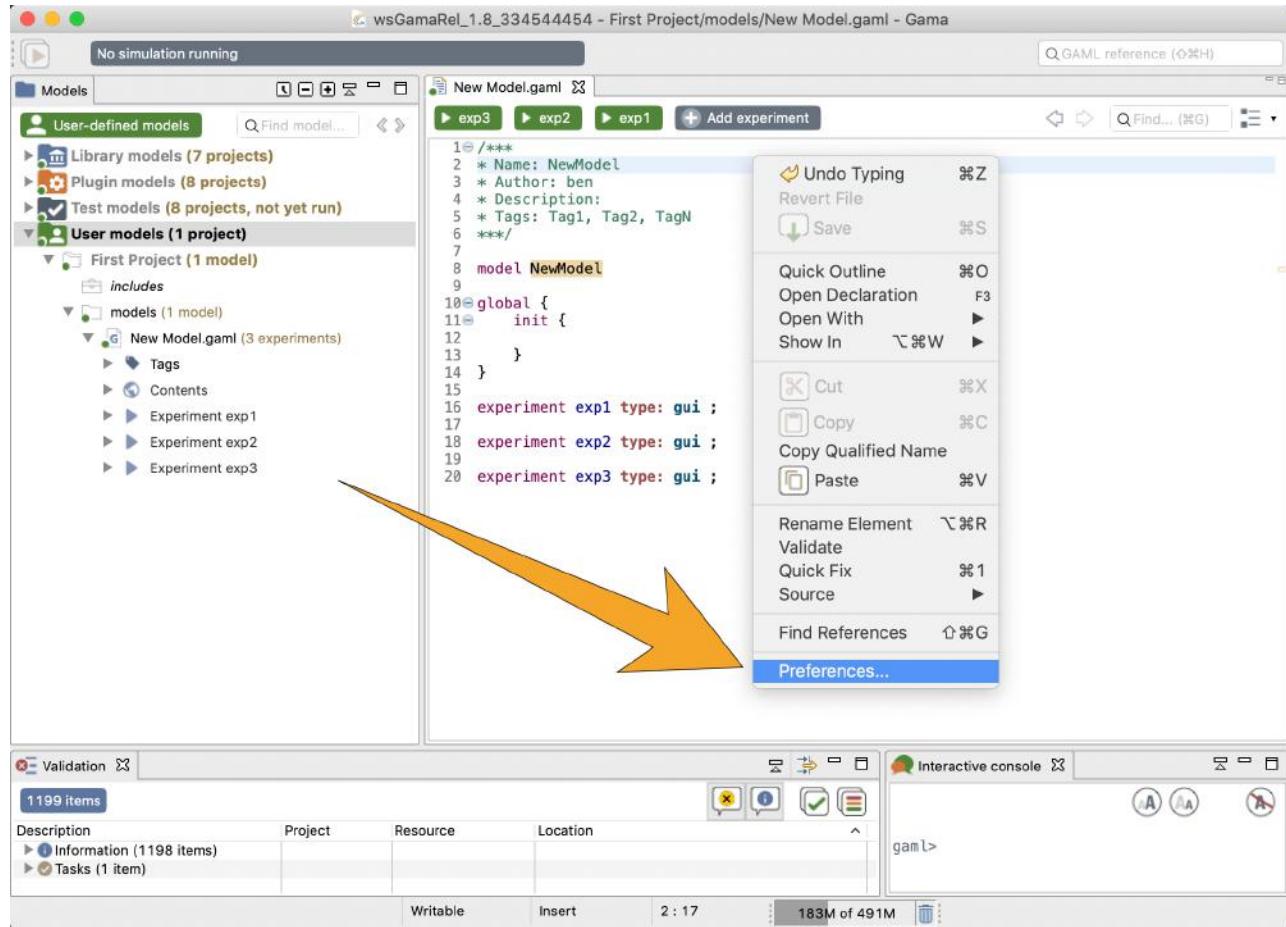
The toolbar on the top of the GAML editor displays, in addition to the green experiment buttons, a button to add an experiment in the current model.



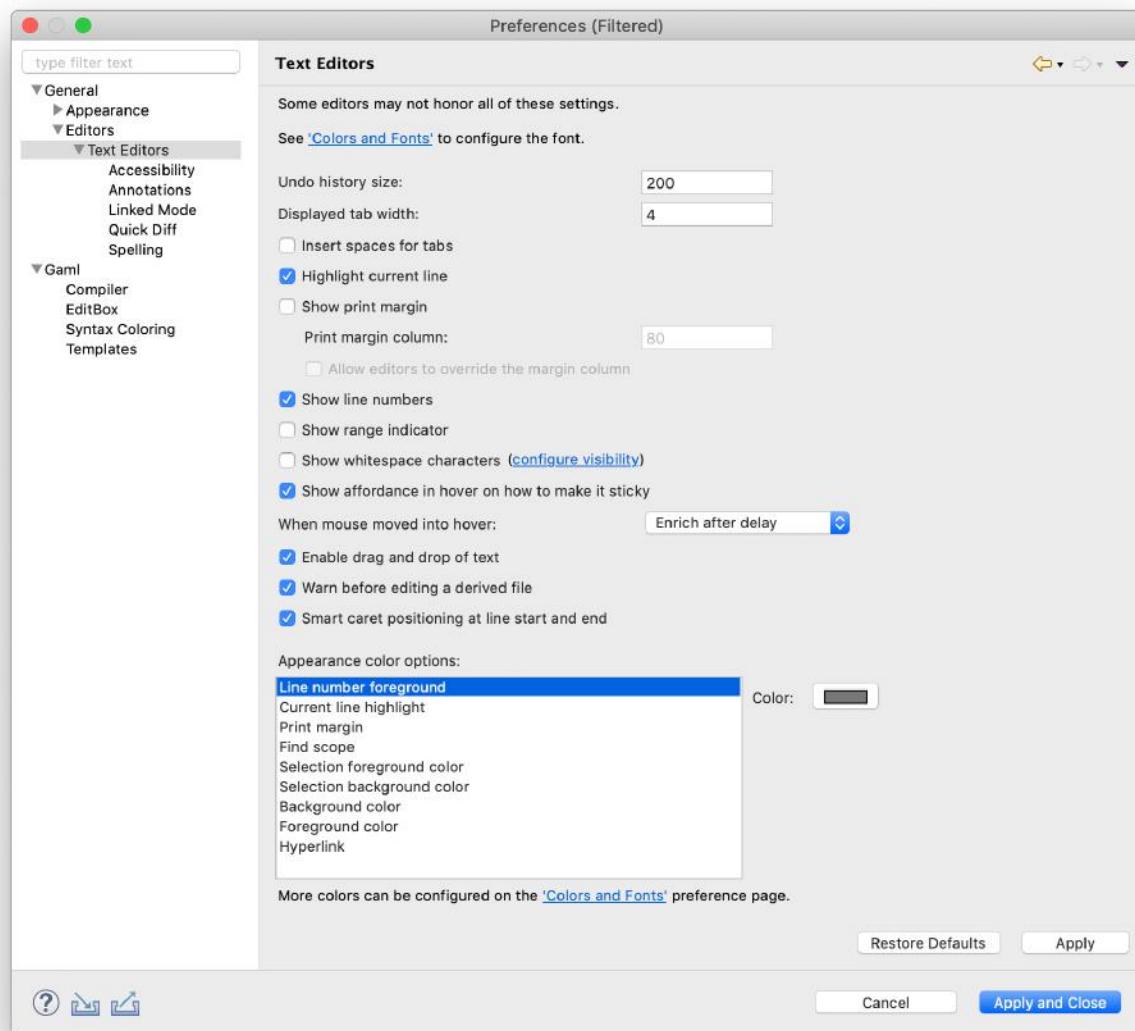
Editor Preferences

Text editing in general, and especially in Eclipse-based editors, sports several 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.

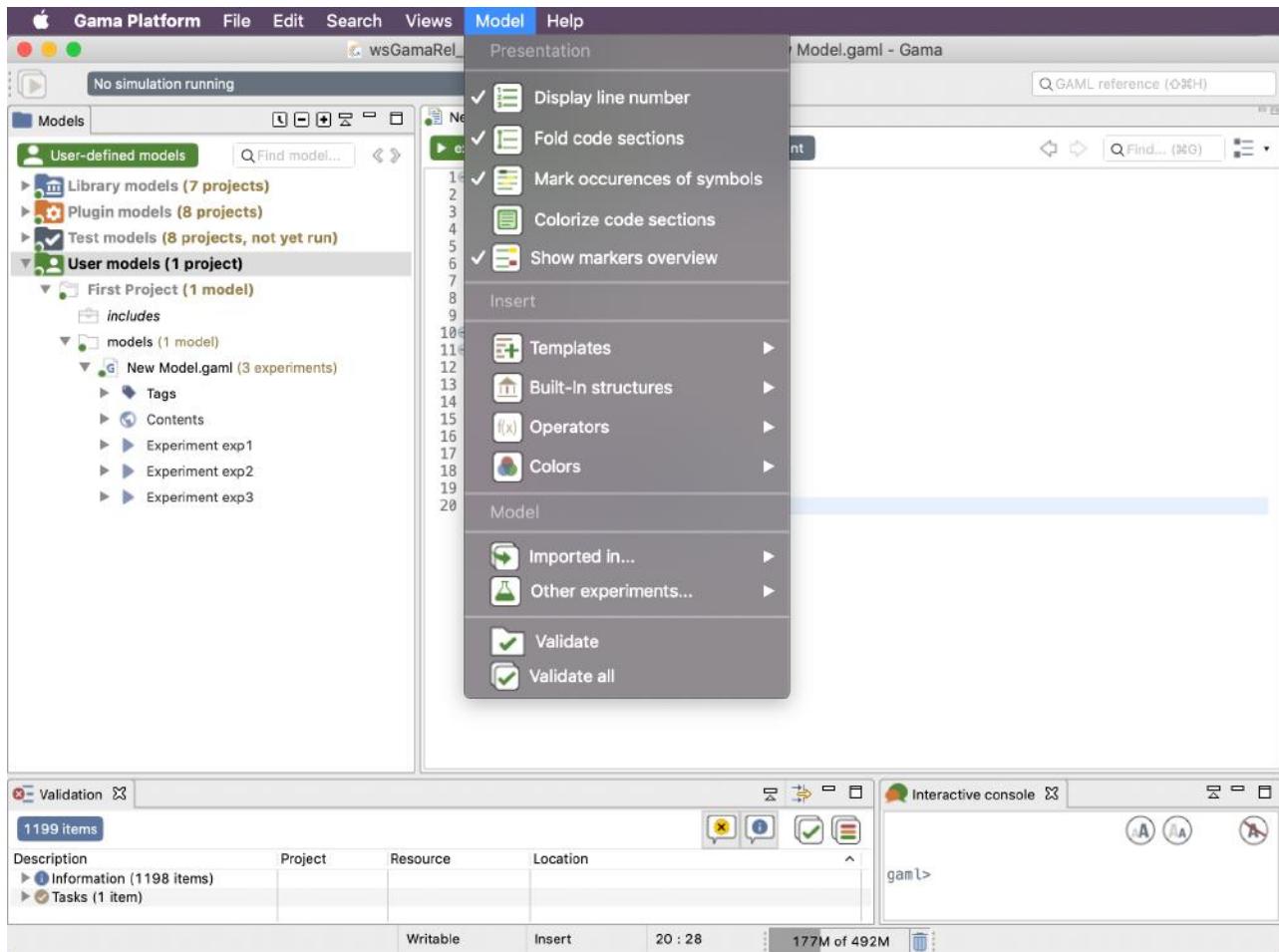


Explore the different items present there, keeping in mind that these preferences will apply to all the editors of GAMA and will be stored in your workspace.



Additional information in the Editor

You can choose to display or not some information in your Editor, from the Models menu available when the GAML editor is active.



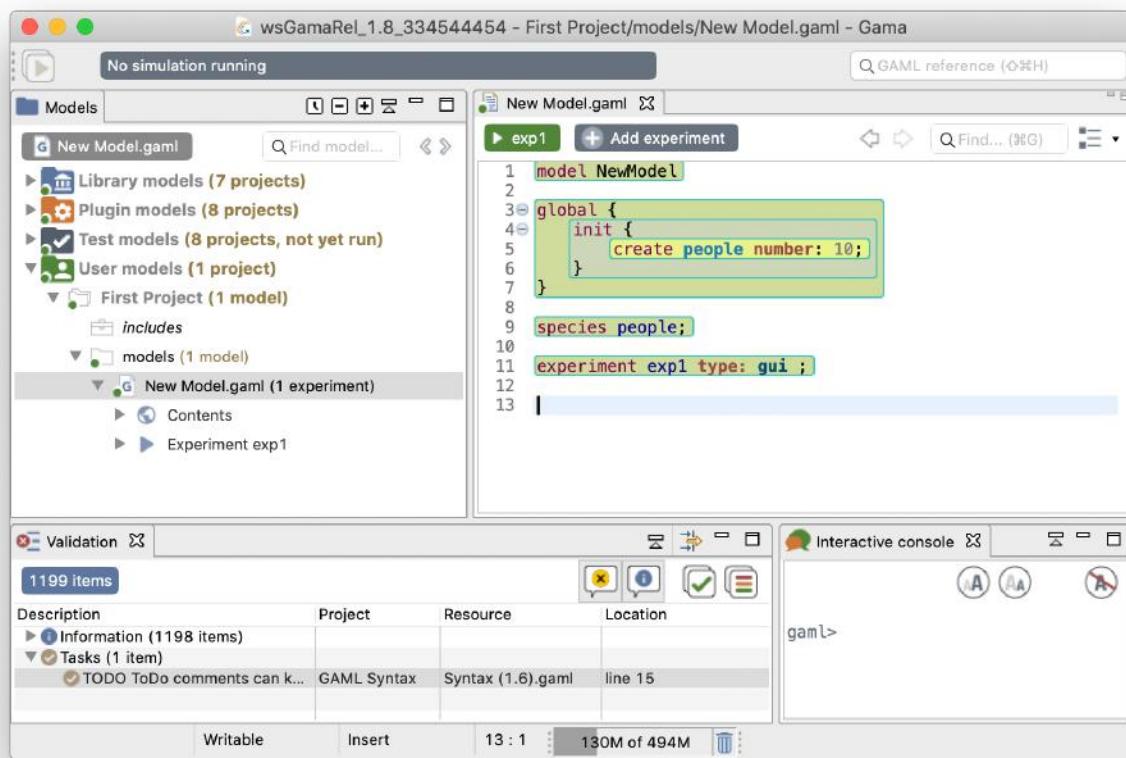
In particular, this menu allows the user to activate/deactivate the additional information that can be displayed in the editor:

- "Display line number": the display of the line number in the left margin.
- "Fold code sections": the **-** and **+** icons on the left of each code section can fold/unfold the associated code section.
- "Mark occurrence of symbols": when the name of a variable or species is selected in the code, all its other occurrences will be also marked.
- "Colorize code sections": the code section can be colorized, improving the visualization of the model organization (see below).
- "Show markers overview": a right-click on the left margin of the editor allows the user to add either bookmarks or tasks to the editor (with a mark on the right)

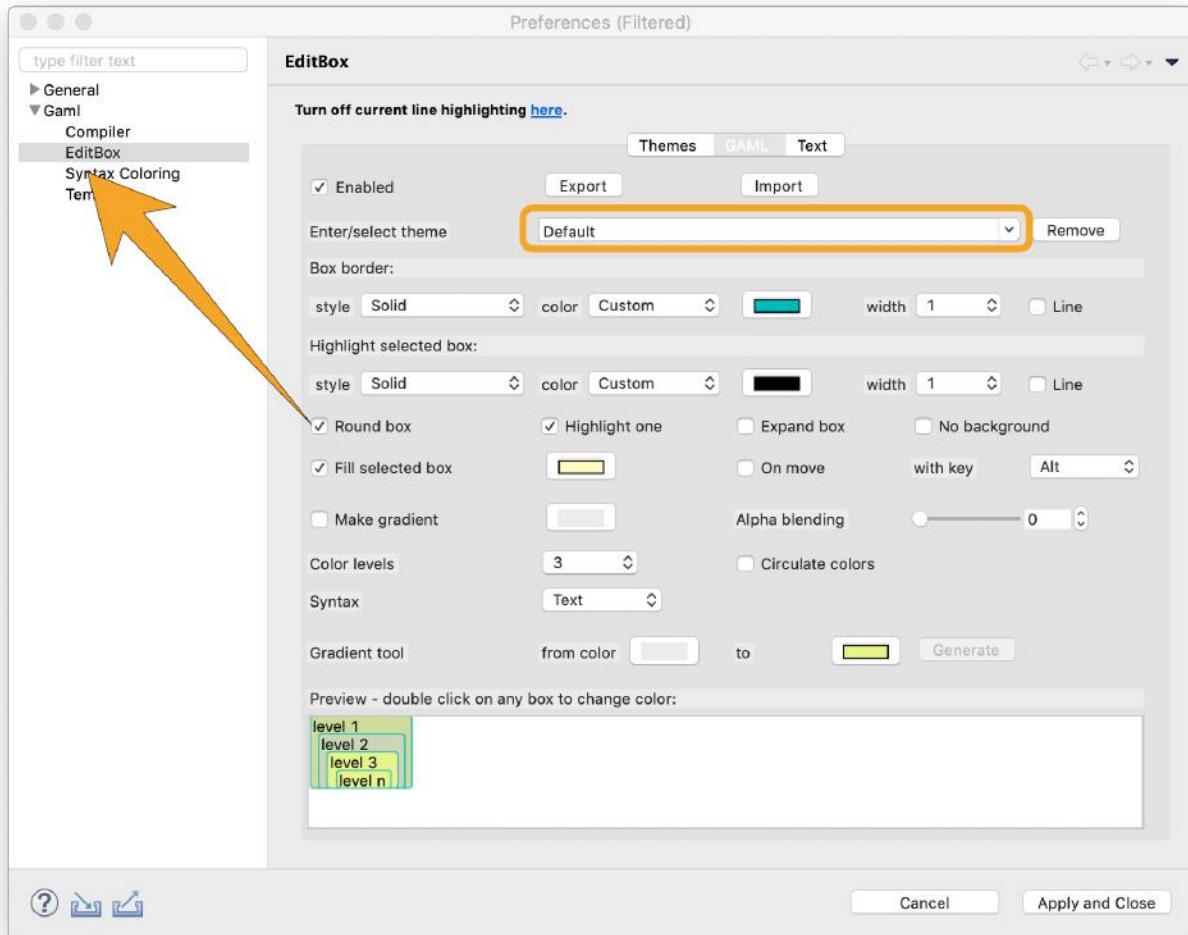
margin.



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](#).

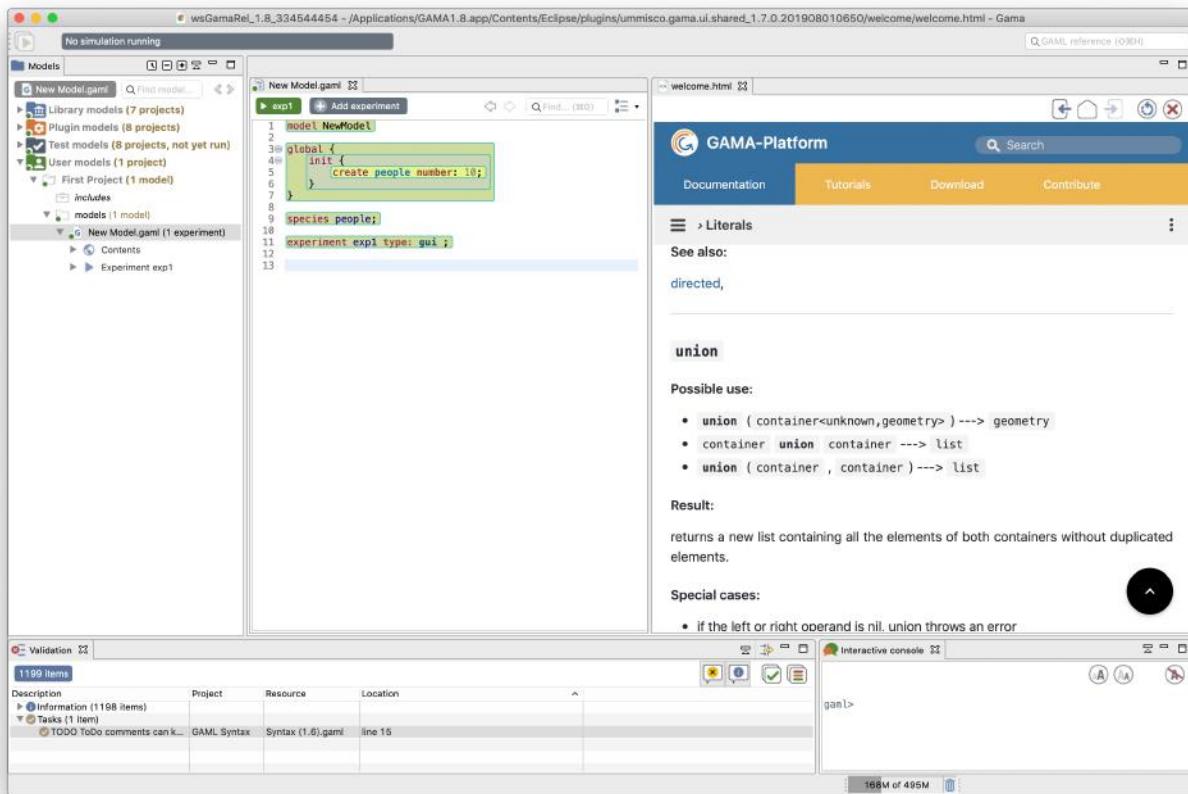


The Default theme of [EditBox](#) might not suit everyone's tastes, so the preferences allow to entirely customize how the "boxes" are displayed and how they can support the modeler in better understanding "where" it is in the code. The "themes" defined in this way are stored in the workspace, but can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



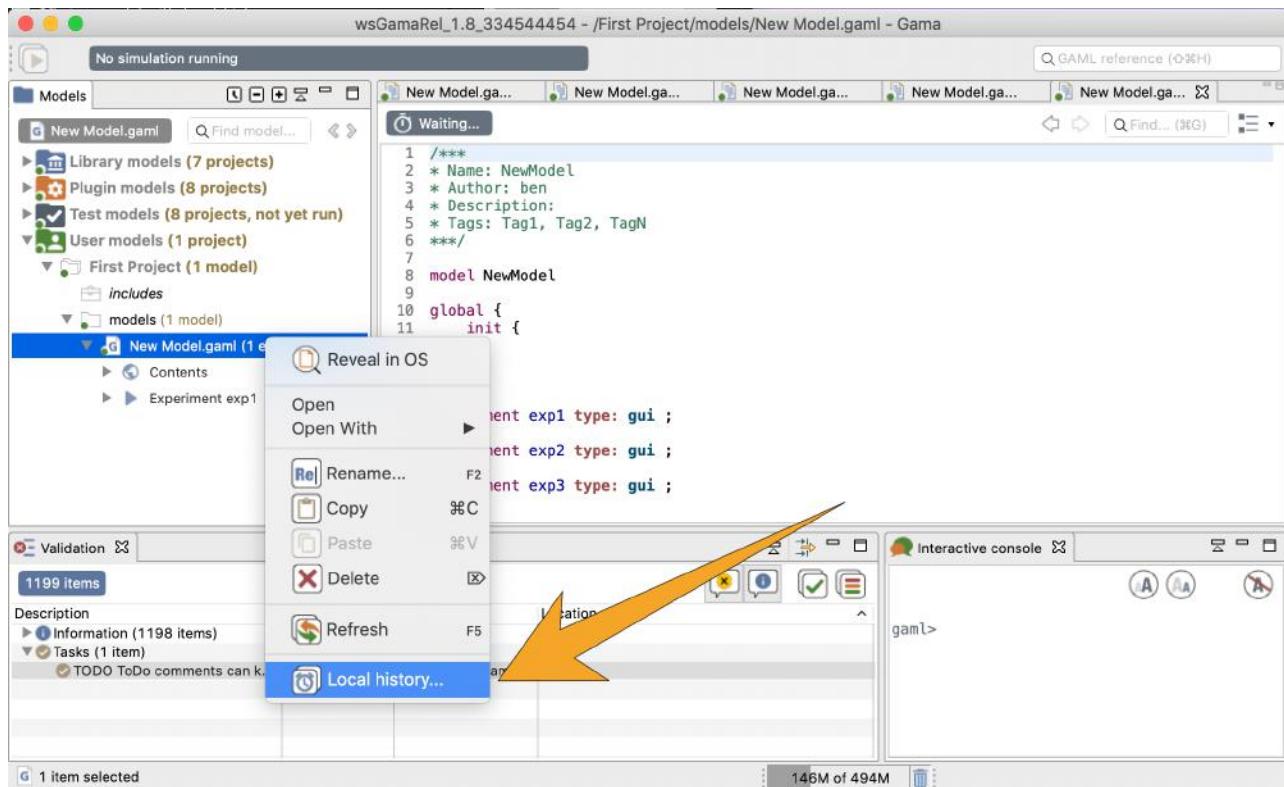
Multiple editors

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.

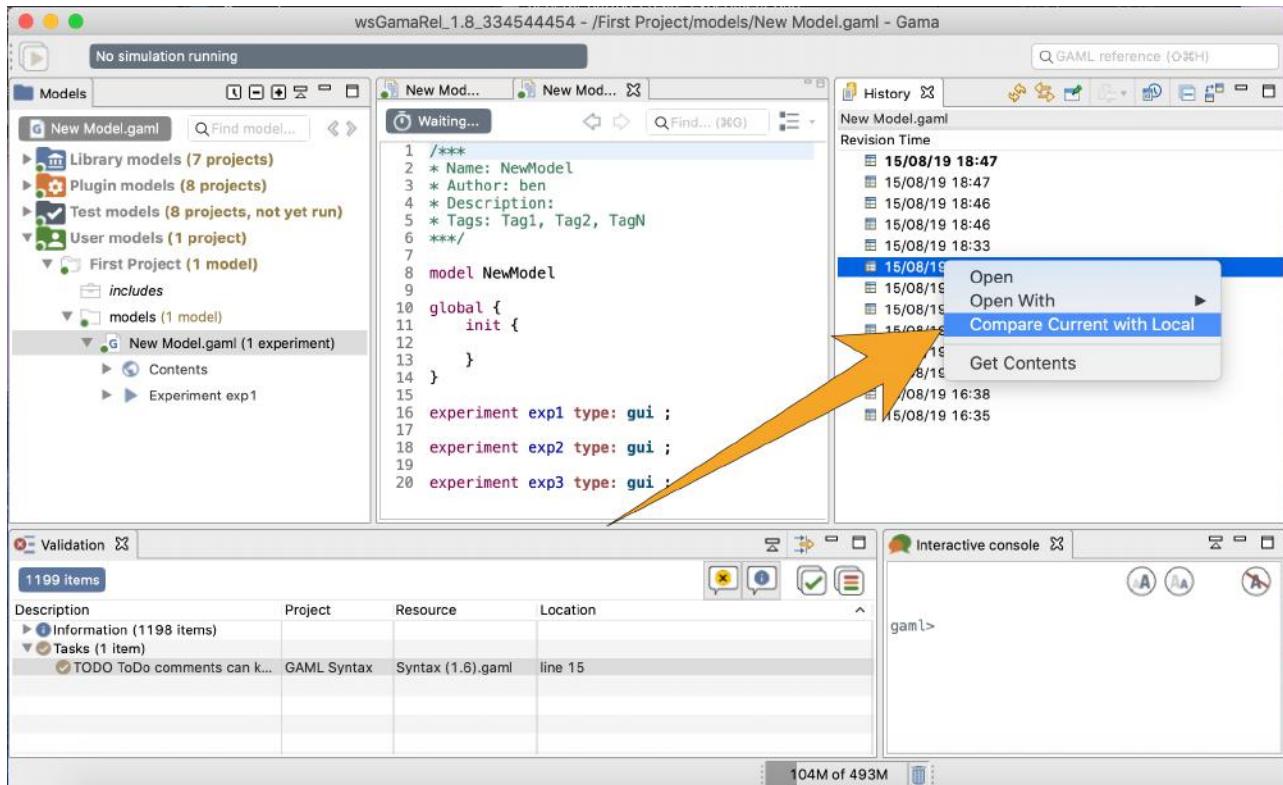


Local history

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 the contextual menu on the chosen model.



This command invokes the opening of a new view, which you can see in 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!

wsGamaRel_1.8_334544454 - Compare /First Project/models/New Model.gaml Current and Local Revision - Gama

```

Local: New Model.gaml
1 model NewModel
2
3 global {
4     init {
5         create people number: 10;
6     }
7 }
8
9 species people;
10
11 experiment exp1 type: gui ;
12
13
14
15
16 experiment exp1 type: gui ;
17
18 experiment exp2 type: gui ;
19
20 experiment exp3 type: gui ;

```

Local history: New Model.gaml 15-Aug-2019 17:55:56

```

1 /**
2 * Name: NewModel
3 * Author: ben
4 * Description:
5 * Tags: Tag1, Tag2, TagN
6 */
7
8 model NewModel
9
10 global {
11     init {
12
13
14
15
16 experiment exp1 type: gui ;
17
18 experiment exp2 type: gui ;
19
20 experiment exp3 type: gui ;

```

Revision Time

- 15/08/19 18:47
- 15/08/19 18:47
- 15/08/19 18:46
- 15/08/19 18:46
- 15/08/19 18:33
- 15/08/19 17:56
- 15/08/19 17:55
- 15/08/19 17:55
- 15/08/19 17:52
- 15/08/19 16:46
- 15/08/19 16:41
- 15/08/19 16:38
- 15/08/19 16:35

Validation

Description	Project	Resource	Location
Information (1198 items)	GAML Syntax	Syntax (1.6).gaml	line 15
Tasks (1 item)			
TODO ToDo comments can k...			

Interactive console

```

gaml>

```

Left: 5 : 24, Right: 8 : 1, incoming change #2 (Left: 5 : 5, Right: 12 : 12)

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.

Version: 1.9.1

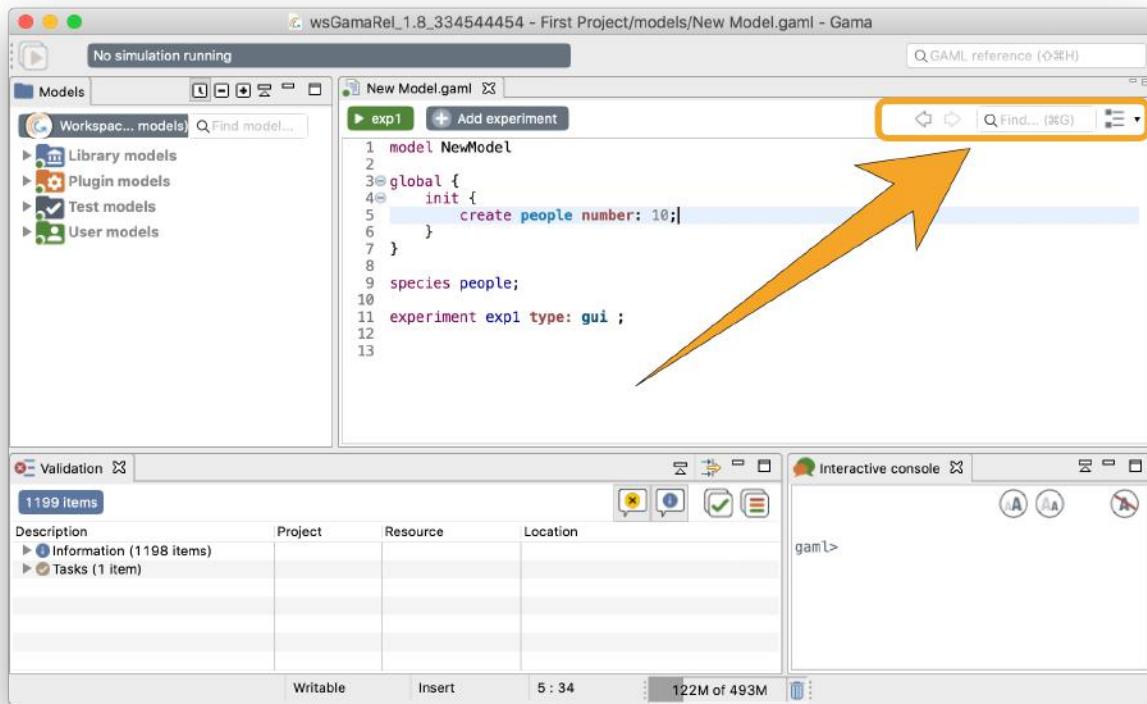
GAML Editor tools

The GAML Editor provides some tools to make the editing easier, covering a lot of functionalities, such as tools for changes of visualization, tools for navigation through your model, tools to format your code, or also tools to help you to find the correct keywords to use in a given context. Some can be accessed directly from the toolbar on top of the editor, but most of the tools are available in the menu "Model", **that is only available when the GAML editor is active (i.e. when the modeler is editing the model).**

Table of contents

- [GAML Editor tools](#)
 - [Navigation tools in the editor](#)
 - [Visualization tools in the menu](#)
 - [Vocabulary tools in the menu](#)
 - [Vocabulary tools in the toolbar](#)
 - [Formatting tools in the contextual menu](#)
 - [Mini-map](#)

Navigation tools in the editor



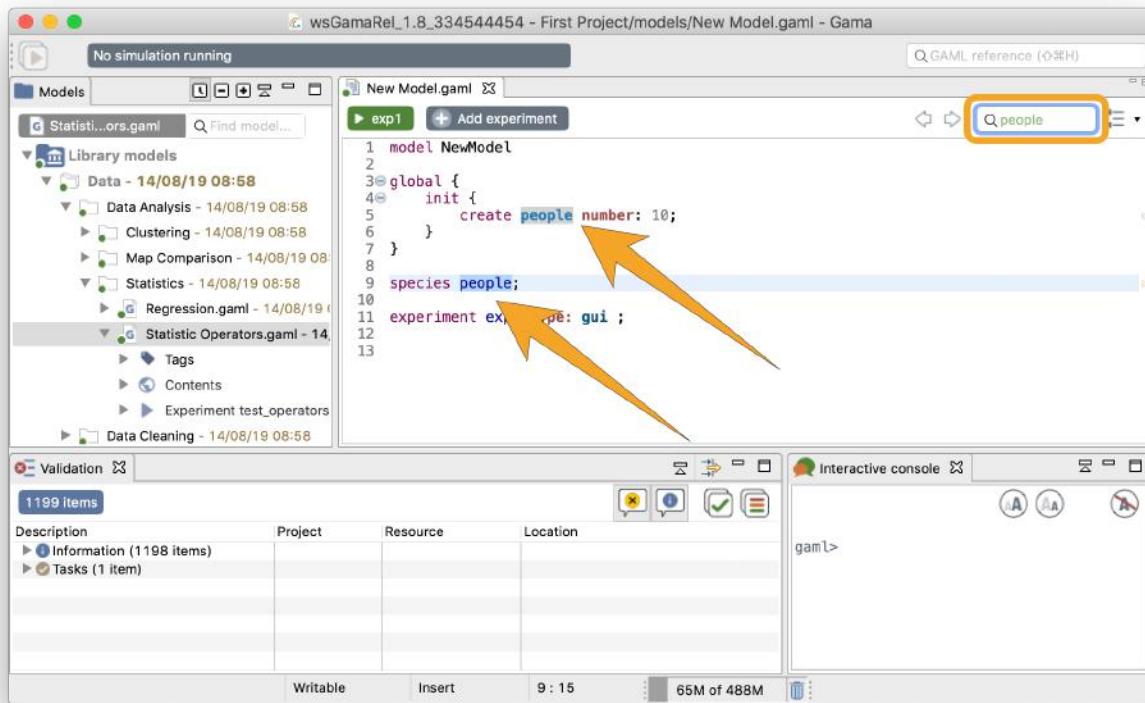
In the Editor toolbar, you have some tools for search and navigation through the code. Here are explanations for each functionality:

Previous/next edit locations

The two arrow shape buttons that are coming first are used to jump from the current location of your cursor to the last position, even if the last position was in another file (and even if this file has been closed !).

The search engine

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

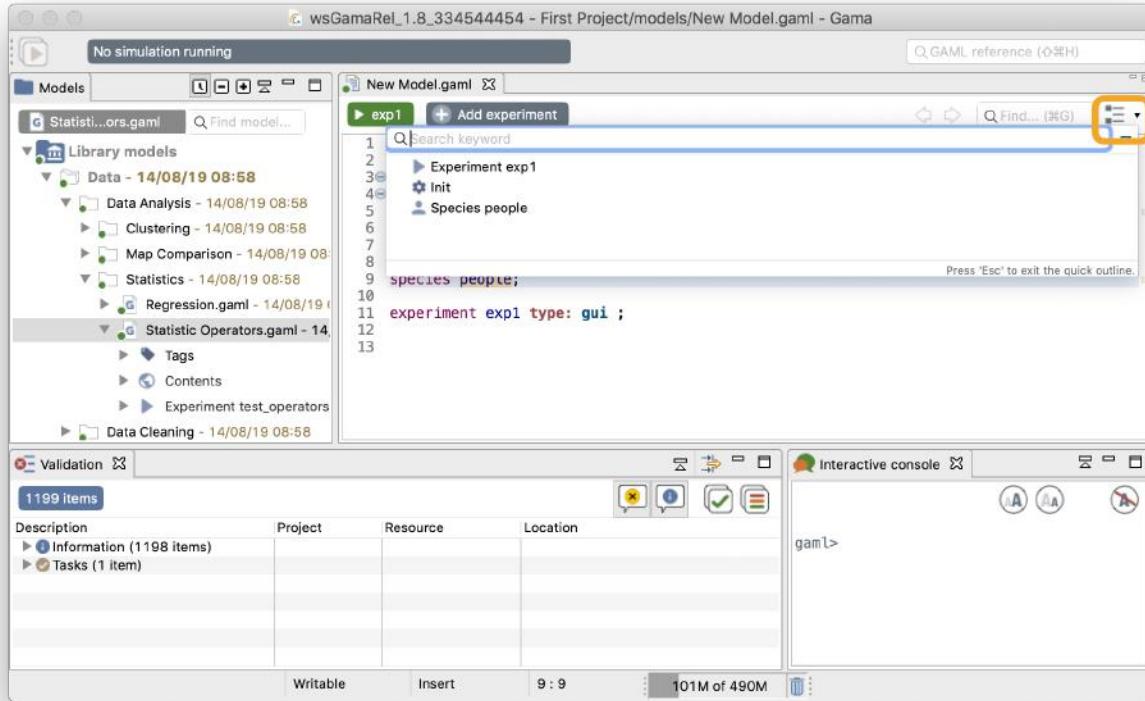


With the left/right arrows, you can highlight the previous/next occurrence of the word. If you prefer the eclipse interface for the search engine, you can also access the tool by taping **Ctrl+F**.

Show outline

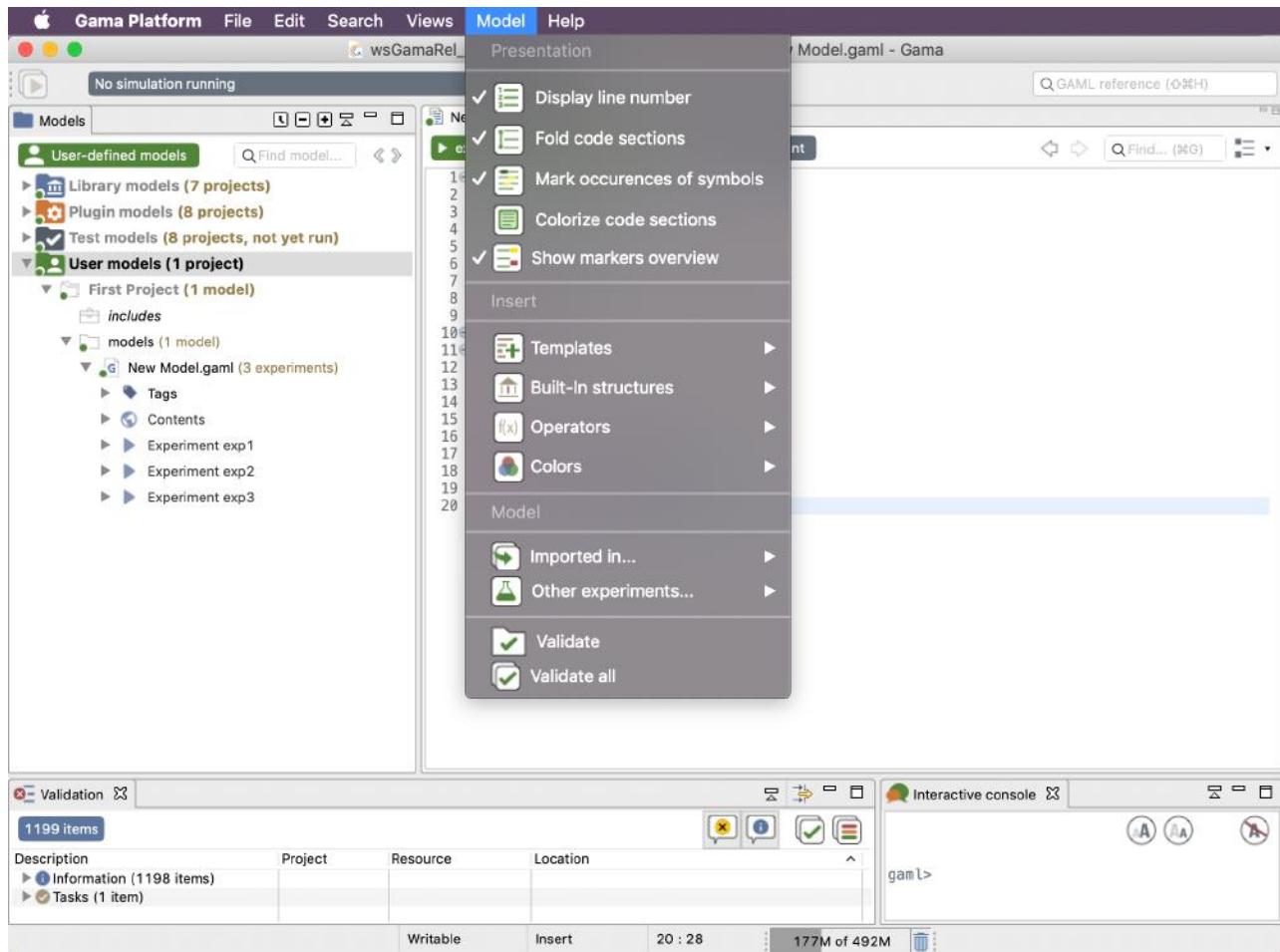
This last tool of this section is used to show the global architecture of your model, with explicit icons for each section. A search field is also available if you want to search for a specific section. By double-clicking one line of the outline, you can jump

directly to the chosen section. This feature can be useful if you have a big model to manipulate.



Visualization Tools in the menu

You can choose to display or not some information in your Editor, from the Model menu. Here are the different features available.



Display line number

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

Fold code sections

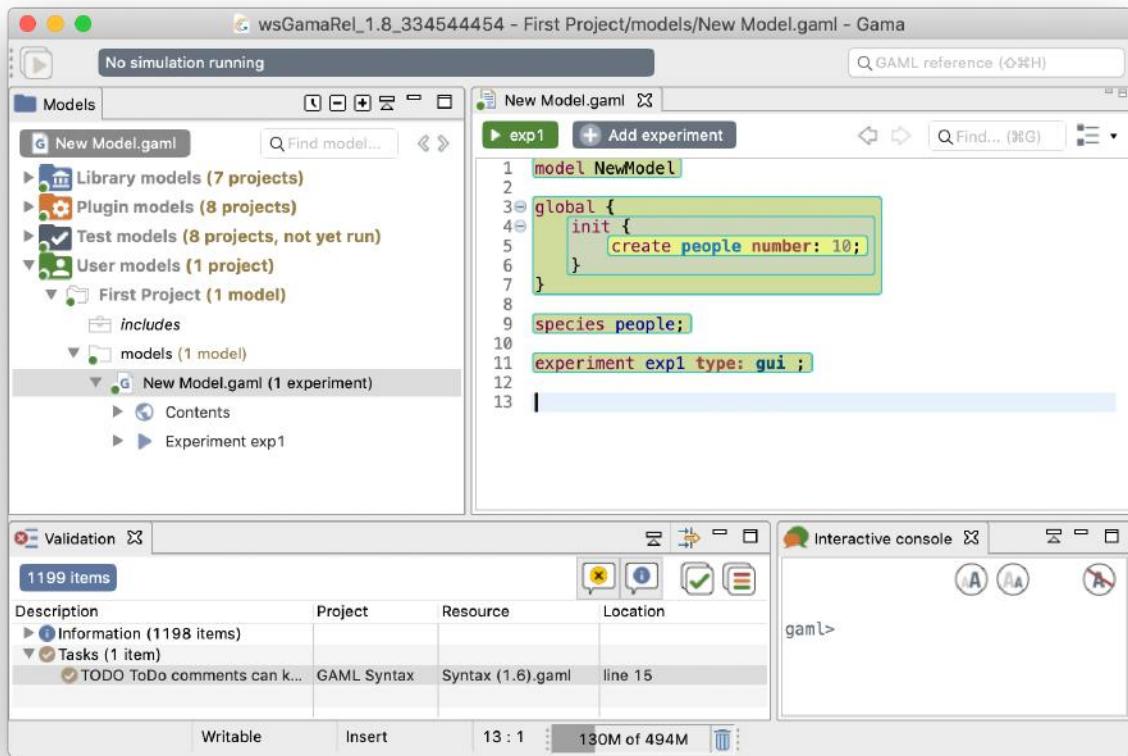
The second toggle provides you the possibility to expand or collapse lines in your model depending on the indentation. This feature can be very useful for big models, to collapse the part you have already finished.

Mark occurrences of symbols

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

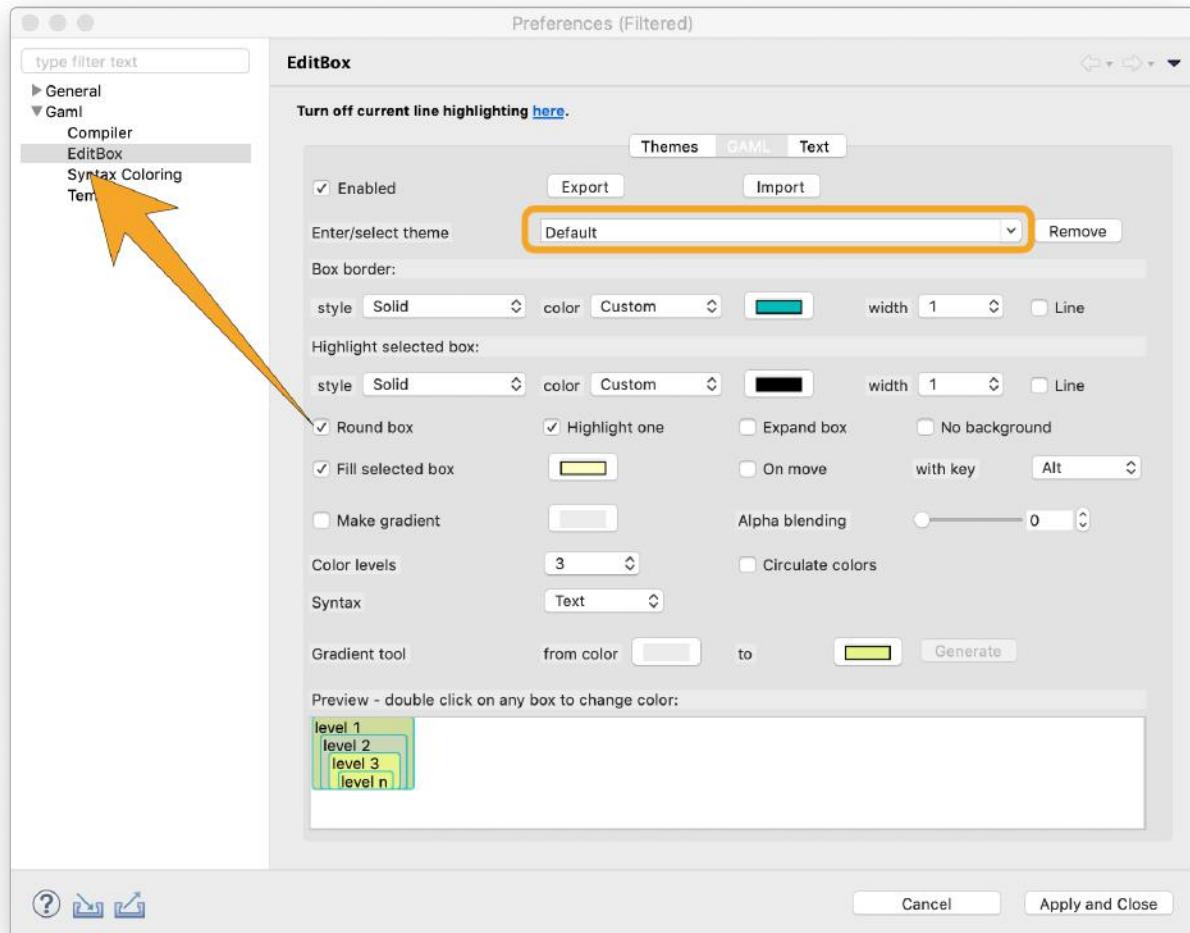
Colorize code sections

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

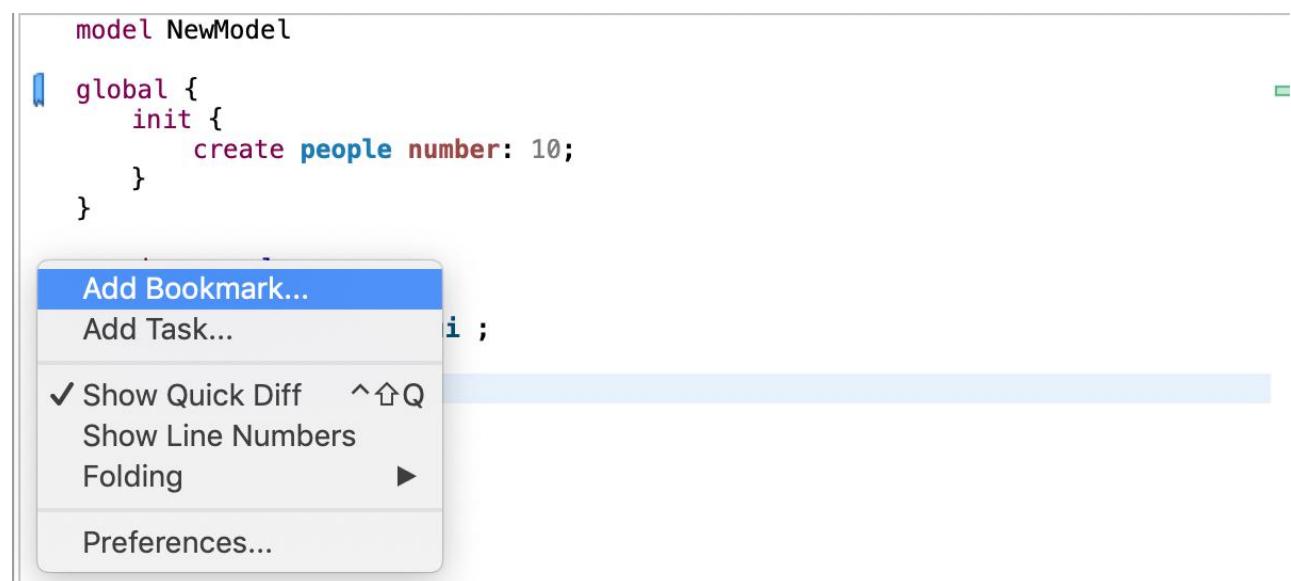
allow to entirely customize how the "boxes" are displayed and how they can support the modeler in better understanding "where" it is in the code. The "themes" defined in this way are stored in the workspace, but can also be exported for reuse in other workspaces, or sharing them with other modelers.



Show markers overview

It is possible to add two kinds of marker on the code: *Bookmarks* (a simple bookmark on a line of code that helps to go back to some lines of interest) and *Tasks* (in addition to a marker on a line, a Task expresses that something should be done, with a given priority, on the code line). The markers are also visible in the right margin of the

editor. An additional view (named Tasks, that can be opened from the Views menu) gathers all the tasks, helping modelers to organize their work.



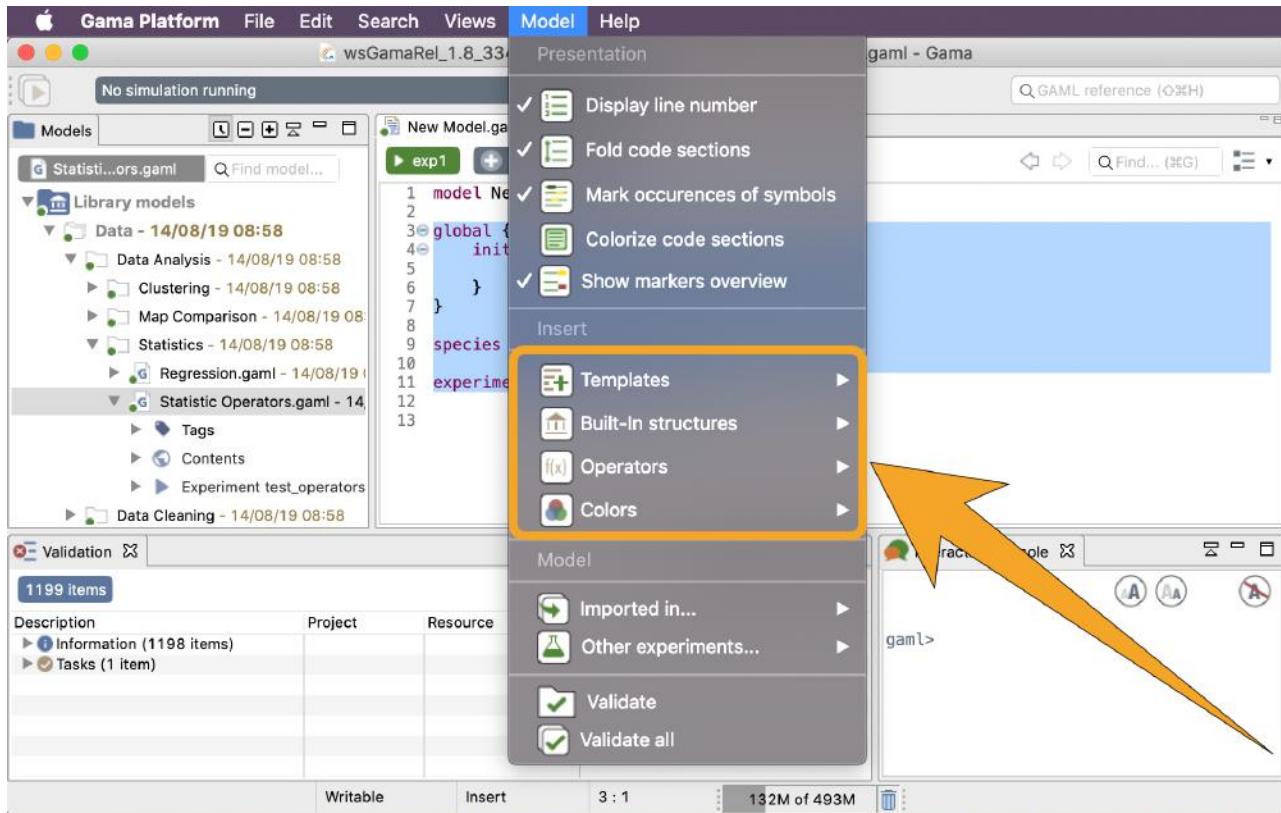
A screenshot of a code editor interface. The editor window contains the following code:

```
model NewModel
  global {
    init {
      create people number: 10;
    }
  }
```

A context menu is open over the code, listing the following options:

- Add Bookmark...
- Add Task...
- ✓ Show Quick Diff ^⇧Q
- Show Line Numbers
- Folding ▶
- Preferences...

Vocabulary tools in the menu



The second group of commands in the Model menu are used to search the correct way to write a certain keyword.

Templates

The templates button is used to insert directly a code snippet in the current position of the cursor. Some snippets are already available, ordered by scope. You can custom the list of templates as much as you want, new templates can be added from the **Preferences** dialog.

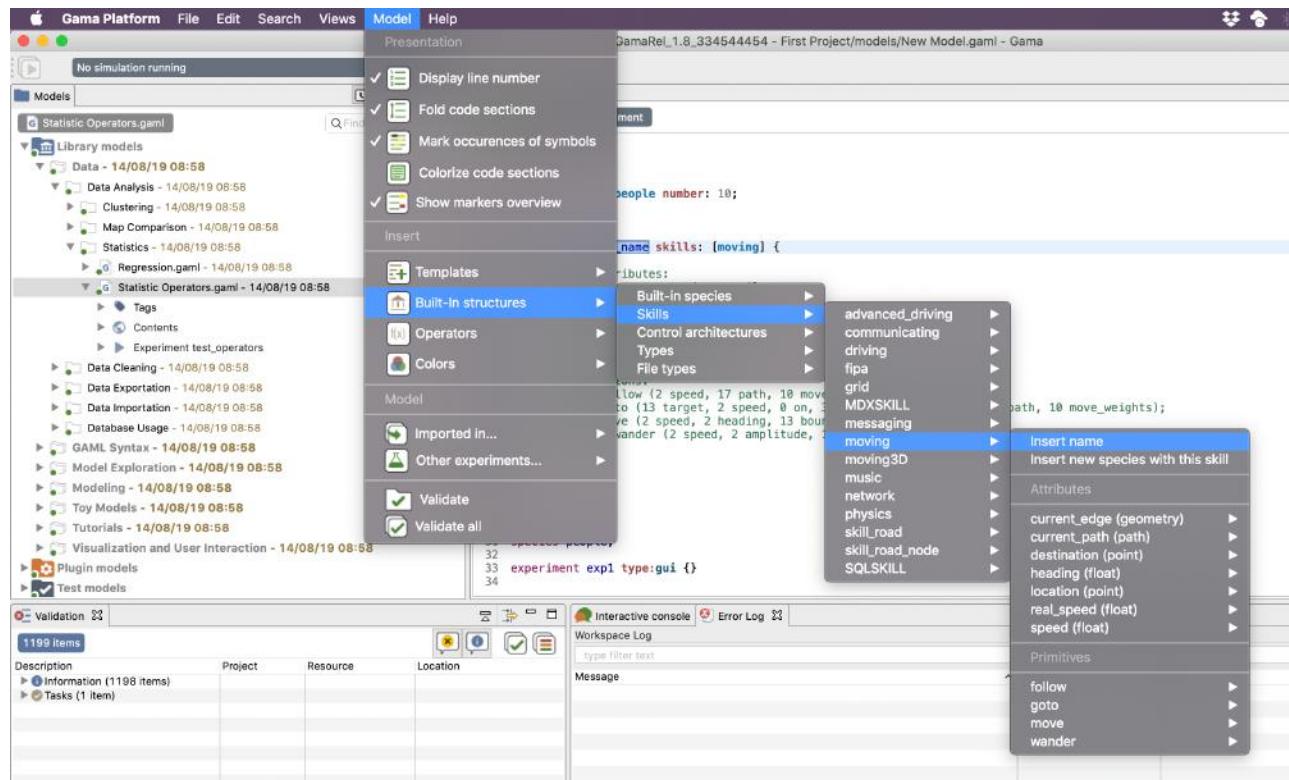
For example, if the modeler clicks on **Template** > **Species** > **grid** > **insert**, the

following code is generated:

```
grid name width:grid_w height:grid_h {  
}
```

Built-in structures

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



... will generate the following code:

```
9 species species_name skills: [moving] {
10 /**
11 * Inherited attributes:
12 *     geometry current_edge <- nil ;
13 *     path current_path <- nil ;
14 *     point destination ;
15 *     float heading <- rnd(360.0) ;
16 *     point location ;
17 *     float real_speed <- 0.0 ;
18 *     float speed <- 1.0 ;
19 * Inherited actions:
20 *     path follow (2 speed, 17 path, 10 move_weights, 3 return_path);
21 *     path goto (13 target, 2 speed, 0 on, 3 recompute_path, 3 return_path, 10 move_weights);
22 *     path move (2 speed, 2 heading, 13 bounds);
23 *     action wander (2 speed, 2 amplitude, 13 bounds, 15 on, 10 proba_edges);
24 */
25
26 }
27
```

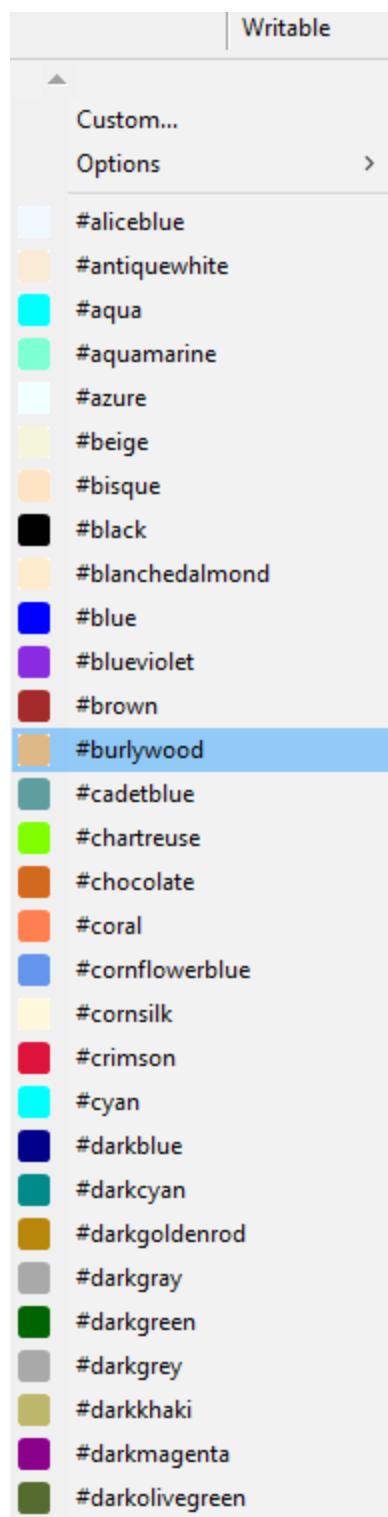
All the comments are generated automatically from the current documentation.

Operators

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

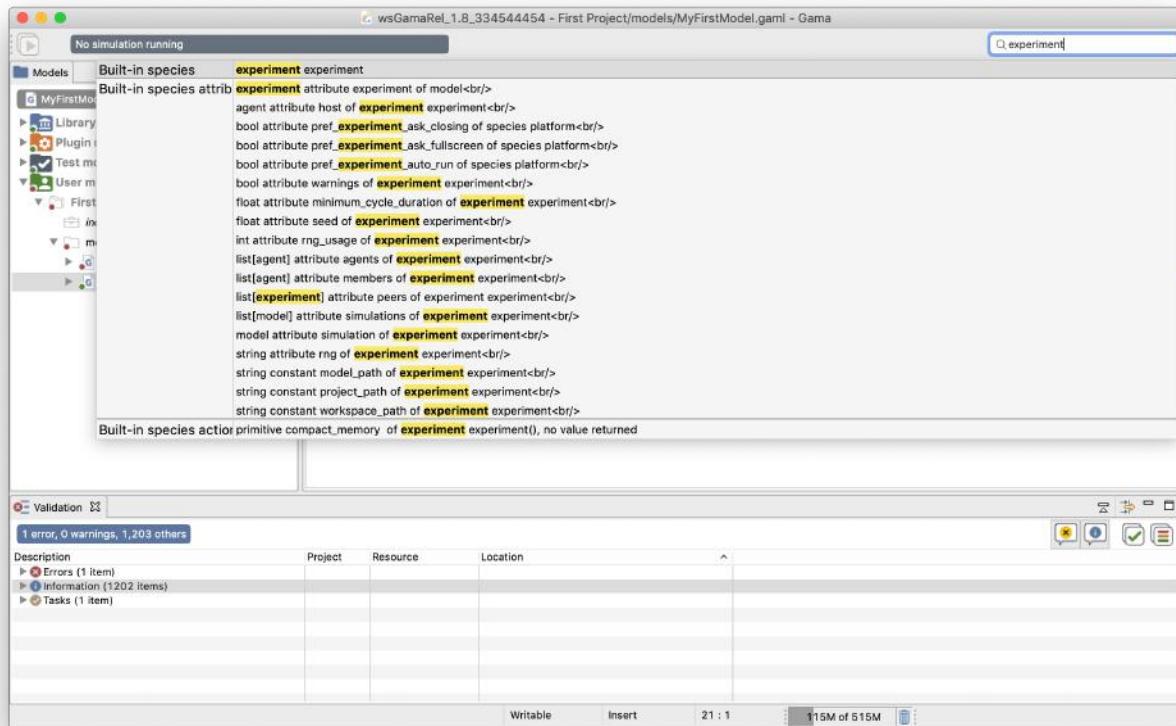
Colors

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

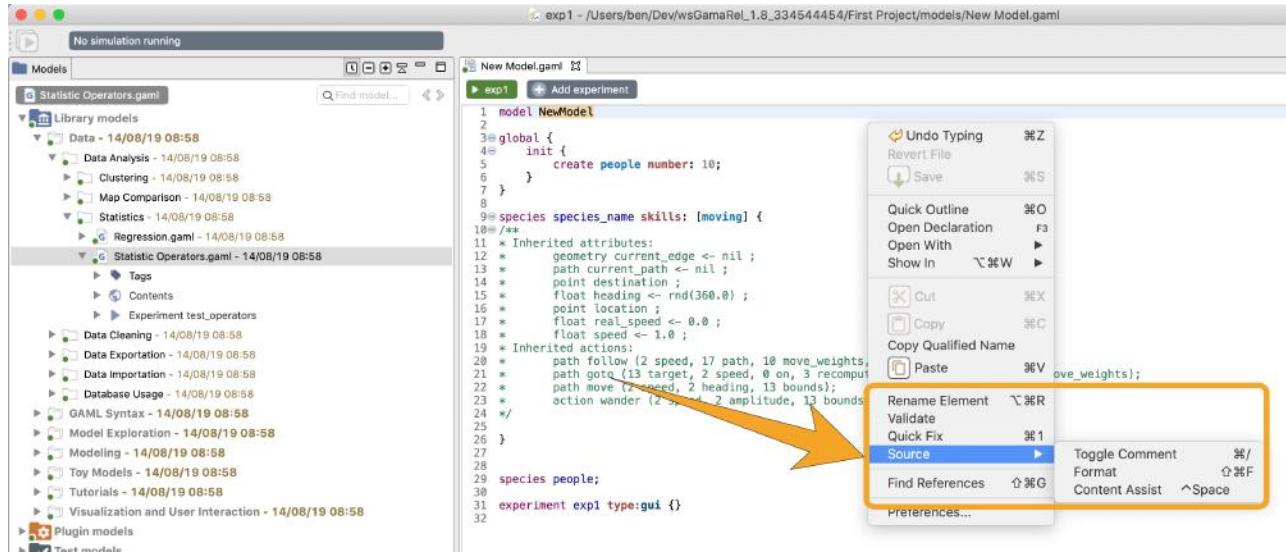


Vocabulary tools in the toolbar

All the information that is available in the "Model" menu can also be accessed, in another way, from the research engine located in the GAMA interface toolbar, named "GAML reference". Any word typed in this search engine will be searched in all the keyword of the GAML language: for example, if the word `experiment` is searched, the search engine retrieves its occurrence as a built-in species, a statement, an attribute or a type... This is definitely the easiest way to get information about any GAML keyword.



Formatting tools in the contextual menu



Some other tools are available in the contextual menu to help for the formatting and refactoring of the model:

Rename element

Once an element selected, this command allows the modeler to rename it. All the occurrences of this element name will be altered. This is particularly useful in a model when we want to refactor the model: e.g. rename an attribute and that this modification to be taken into account in all the model code.

Source > Comment

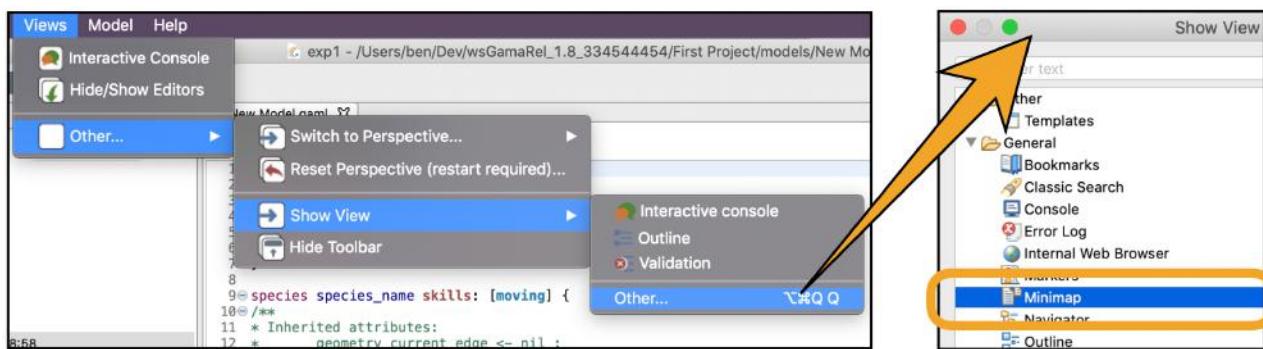
This command is used to comment a line (or a group of lines).

Source > Format

This useful feature re-indent automatically all your model.

Mini-map

The mini-map provides a view of the whole model in a very tiny font. It helps to have an overview of the model. The mini-map is a *View* that needs to be shown, from the *View* menu.



The mini-map view can be moved close to the editor. The modeler can navigate in the mini-map to move quickly between the various parts of the model.

No simulation running

Models Lists.gaml Add experiment

Lists.gaml

```

83 species test_species()
84
85 species accessing_list_elements {
86     list-int l1 <- [1,2,3,4,5,6,7,8,9,10];
87     list-string l2 <- [ 'one', 'is', 'a', 'list', 'of', 'strings'];
88 }
89 init {
90     write "";
91     write "----- ACCESSING LIST ELEMENTS -----";
92     write "";
93     write sample(l1);
94     write sample(first(l1));
95     write sample(last(l1));
96     write sample(length(l1));
97     write sample(contains_all(l1));
98     write sample(contains_any(l1));
99     write sample(min(l1));
100    write sample(max(l1));
101    write sample(index(l1));
102    write sample(index(l1, 2));
103    write sample(index(l1, 1));
104    write sample(index(l1, 2));
105    write sample(index(l1, 1));
106    write sample(index(l1, 1));
107    write sample(l1 contains_all [1..6, 14]);
108    write sample(l1 contains_any [1..23]);
109    write sample(l1 collect (each + 1));
110    write sample(l1 collect (norm(each, each, each)));
111    write sample(l1 filter (each > 5));
112    write sample(l1 filter (each > 5));
113    write sample(l1 group_by (even(each)));
114    write sample(l2 index_by (each + "_index"));
115    write sample(l2 last_index_of 'is');
116    write sample(l2 sort_by each);
117    write sample(l2 sort_by each);
118    write sample(l2 first_with (first(each) = 'o'));
119    write sample(l2 where (length(each) = 2));
120    write sample(l2 with_min_index (length(each)));
121    write sample(l2 with_max_index (length(each)));
122    write sample(l2 min_of (length(each)));
123    write sample(l2 max_of (length(each)));
124    write sample(l2 copy_between(l2, 1, length(l2) - 1));
125    write sample(l2 as_map (length(l2)::new+each));
126    write sample(l2 as_map (length(l2)::new+each));
127    write sample(l2 as_map (length(l2)::new+each));
128    write sample(l2 as_map (length(l2)::new+each));
129 }
130 }
```

Minimap

Validation

Interactive console

gaml>

1198 items

Description	Project	Resource	Location
Information (1198 items)			
Tasks (1 item)			

108M of 724M

Version: 1.9.1

Validation of Models

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

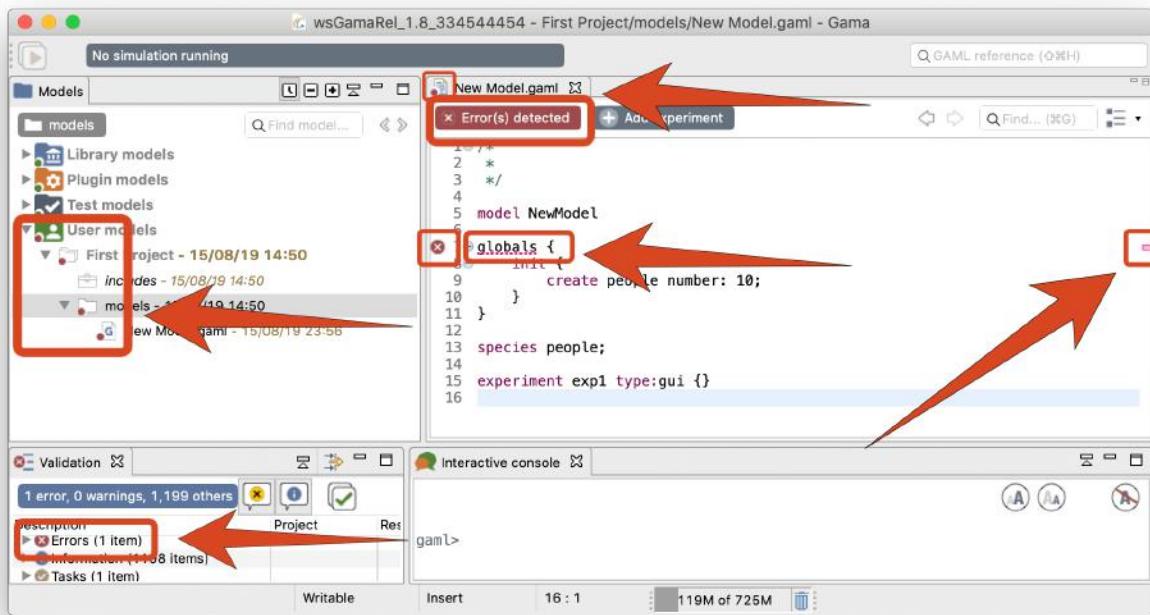
Table of contents

- Validation of Models
 - Syntactic errors
 - Semantic errors
 - Semantic warnings
 - Semantic information
 - Semantic documentation
 - Changing the visual indicators
 - Errors in imported files
 - Cleaning models

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. Finally, the validation view gathers all the errors of the workspace.



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

```
4
5 model NewModel
6
7 global {
8     init {
9         create people number: 10;
10    }
11
12
13 species people;
14
15 ✘Multiple markers at this line
16 -
```

- Symbol 'experiment' seems to be incomplete

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.

12
✖ 13 species people parent: people;
14

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 the 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 executable. 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 the same error, see below).

```
10
✖ 11     geometry g <- square(10,10);
12
```

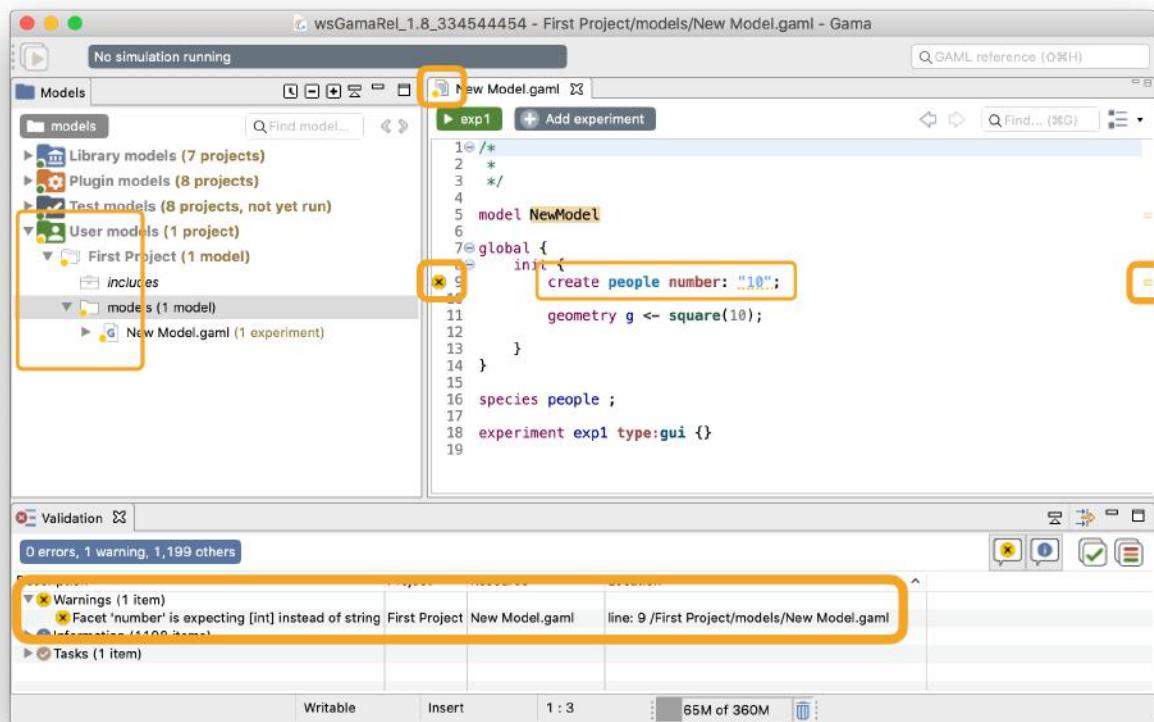
10
✖ 11 No operator found for applying 'square' to types [int, int] (operators available for [type float])
12

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.

```
create people number: "10";
```

GAMA will emit a warning in such a case, indicating that `number:` expects an integer and that the string passed will be cast 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.



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

```
7 ⊕ global {
8 ⊕   init {
9 Facet 'number' is expecting [int] instead of string
10
11   geometry g <- square(10);
12
13 }
14 }
```

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).

The screenshot shows the Gama IDE interface. The main window displays a GAML file named 'New Model.gaml' with the following code:

```

1 /* 
2 *
3 */
4 model NewModel
5 global {
6     geometry shape <- circle(100);
7 }
8 init {
9     create people number: 10;
10 }
11 species people ;
12 experiment exp1 type:gui {}
13
14
15
16
17
18
19

```

A blue arrow points from the left towards the line 'geometry shape <- circle(100);'. A blue box highlights the information marker icon (a small circle with a dot) located next to the line number 6. Another blue box highlights the 'Validation' tab at the bottom of the interface.

Validation

1201 items

Description	Project	Resource	Location
Information (1200 items)			
Tasks (1 item)			

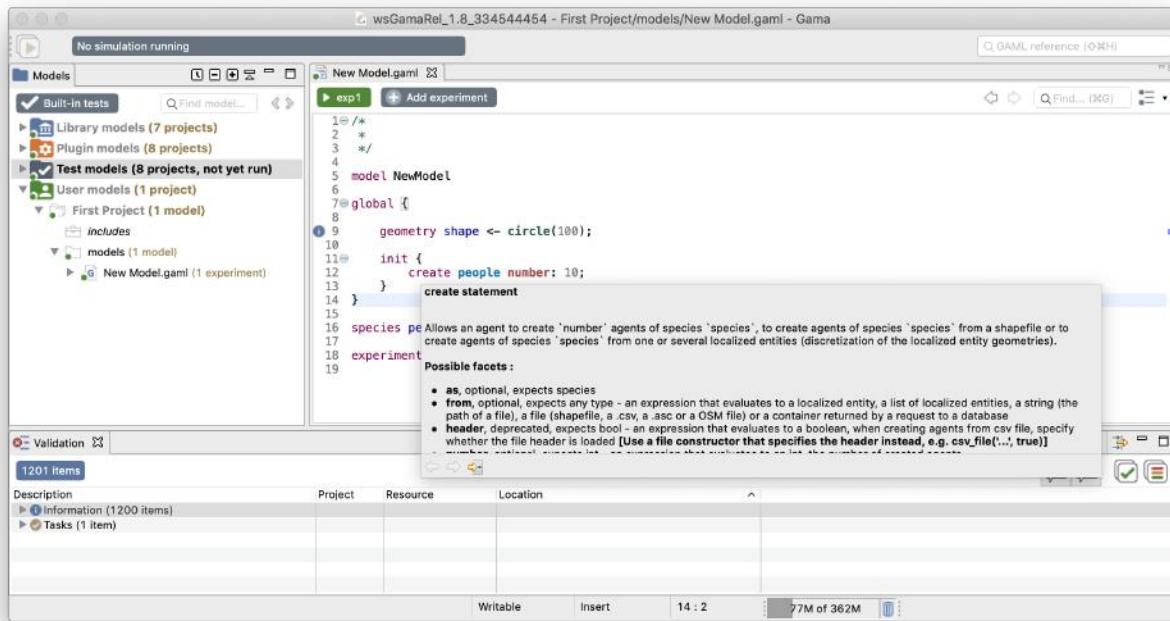
1 item selected

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

The screenshot shows the Gama IDE interface. The main window displays the same 'New Model.gaml' file as before. A blue arrow points from the left towards the line 'geometry shape <- circle(100);'. A large blue box highlights the information marker icon (a small circle with a dot) located next to the line number 6. A tooltip box appears over the marker, containing the message: 'This definition of shape supersedes the one in built-in species agent'. The validation panel at the bottom is identical to the one in the previous screenshot.

Semantic documentation

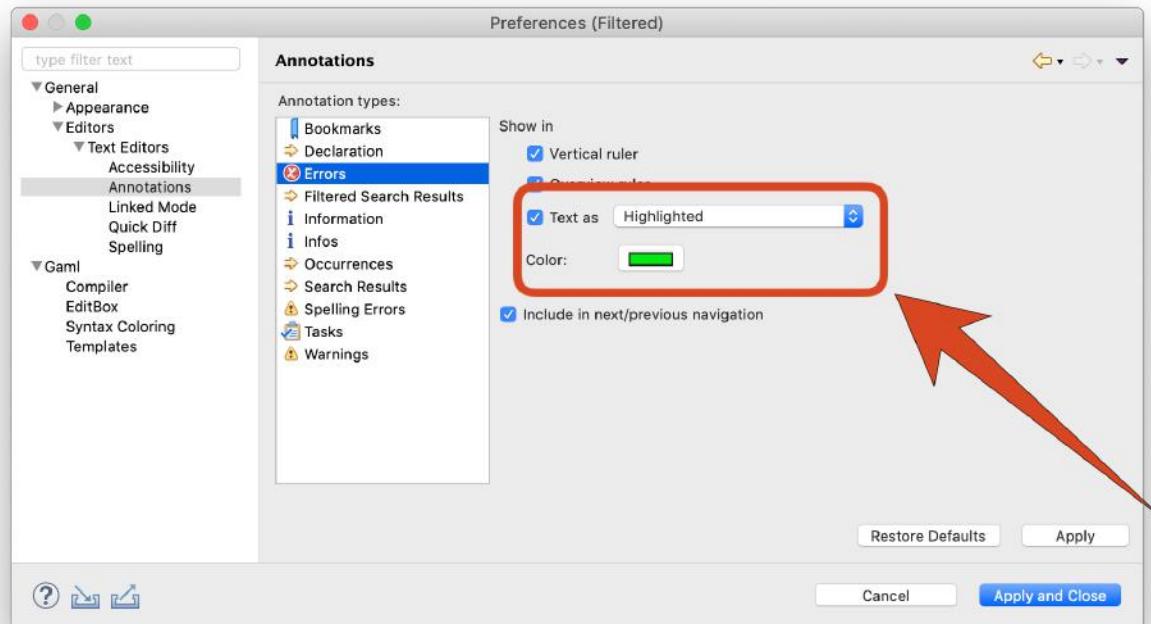
The last type of output of the semantic validation phase consists of a complete documentation of the various elements present in the model, which the user can retrieve by hovering over the different symbols. Note that although the best effort is being made in producing a complete and consistent documentation, 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:

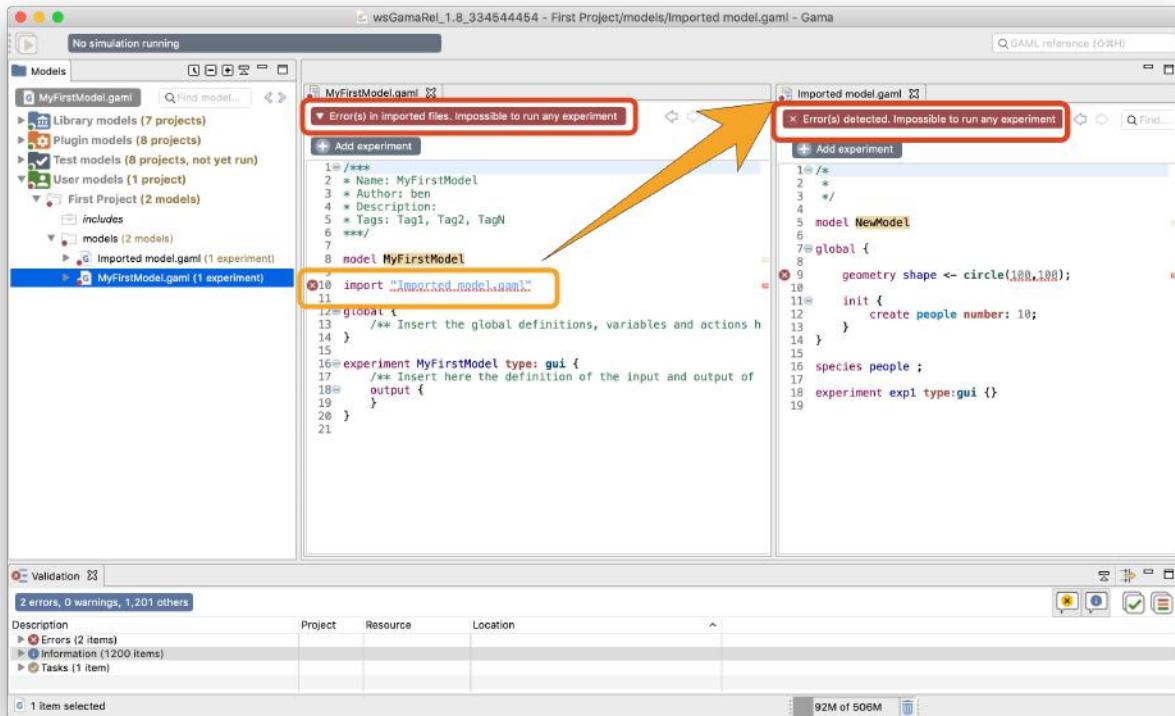
The screenshot shows the Gama Integrated Development Environment (IDE). The main window title is "wsGamaRel_1.8_334544454 - First Project/models/New Model.gaml - Gama". The left sidebar lists "Models" with sections for "Built-in tests", "Library models (7 projects)", "Plugin models (8 projects)", "Test models (8 projects, not yet run)", and "User models (1 project)" which contains "First Project (1 model)" and "models (1 model)" which contains "New Model.gaml (1 experiment)". The right pane displays the GAML code for "New Model.gaml":

```
1/*  
2*  
3*/  
4  
5 model NewModel  
6  
7@global {  
8  
9     geometry shape <- circle(50, 50);  
10    init {  
11        create people number: 10;  
12    }  
13}  
14  
15  
16 species people ;  
17  
18 experiment exp1 type:gui {}  
19
```

A red arrow points to the line "geometry shape <- circle(50, 50);". A red box highlights this line, and a red error icon is displayed above the line in the code editor. Below the code editor, the validation status bar shows "1 error, 0 warnings, 1,201 others".

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, `MyFirstModel.gaml` imports `Imported Model.gaml`, which sports an error.



In such a case, the importing model refuses to compile (although it is itself valid), showing an error in the `import` statement of the model with errors. There are cases, however, where the same importation can work. Consider the previous example, where `Imported Model.gaml` sports a **semantic error** in the definition of the global 'shape' attribute.

The screenshot shows the Gama IDE interface with two code editors open. The left editor contains `MyFirstModel.gaml` and the right editor contains `Imported model.gaml`. Both files have red error bars under the `shape` attribute in their respective `global` sections. The `MyFirstModel.gaml` file also has an error bar under the `import` statement. The validation panel at the bottom shows 1 error, 0 warnings, and 1,203 others.

```

wsGamaRel_1.8_334544464 - First Project/models/Imported model.gaml - Gama
No simulation running
Models
MyFirstModel.gaml Find model...
Library models (7 projects)
Plugin models (8 projects)
Test models (8 projects, not yet run)
User models (1 project)
First Project (2 models)
includes
models (2 models)
Imported model.gaml (1 experiment)
MyFirstModel.gaml (1 experiment)

*MyFirstModel.gaml*
Error(s) in imported files. Impossible to run any experiment.
Add experiment
1/* 
2 * Name: MyFirstModel
3 * Author: ben
4 * Description:
5 * Tags: Tag1, Tag2, TagN
6 */
7
8 model MyFirstModel
9
10 import "Imported.model.gaml"
11
12@global {
13
14 }
15
16@experiment MyFirstModel type: gui {
17     /* Insert here the definition of the input and output of
18     output {
19     }
20 }
21

Imported model.gaml
Error(s) detected. Impossible to run any experiment.
Add experiment
1/*
2 *
3 */
4
5 model NewModel
6
7@global {
8
9     geometry shape <- circle(100,100);
10
11@init {
12     create people number: 10;
13 }
14
15
16 species people ;
17
18 experiment exp1 type:gui {}
19

```

Validation: 1 error, 0 warnings, 1,203 others

Description	Project	Resource	Location
Errors (1 item)			
Information (1202 items)			
Tasks (1 item)			

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

The screenshot shows the Gama IDE interface with two code editors open:

- MyFirstModel.gaml** (Left Editor):


```

1 /* */
2 * Name: MyFirstModel
3 * Author: ben
4 * Description:
5 * Tags: Tag1, Tag2, TagN
6 */
7
8 model MyFirstModel
9
10 import "Imported model.gaml"
11
12 @global {
13     geometry shape <- circle{100,100};
14 }
15
16 @experiment MyFirstModel type:gui {
17     /* Insert here the definition of the input and output of
18     * output {
19     }
20 }
21
      
```
- Imported model.gaml** (Right Editor):


```

1 /* */
2 *
3 */
4
5 model NewModel
6
7 @global {
8     geometry shape <- circle{100,100};
9
10     INIT {
11         create people number: 10;
12     }
13 }
14
15
16 species people ;
17
18 experiment exp1 type:gui {}
19
      
```

A blue arrow points from the line `geometry shape <- circle{100,100};` in the **MyFirstModel.gaml** editor to the same line in the **Imported model.gaml** editor, indicating a redefinition.

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

```

wsGamaRel_1.8_334544454 - First Project/models/MyFirstModel.gaml - Gama
No simulation running

Models
MyFirstModel.gaml Find model...
Library models (7 projects)
Plugin models (8 projects)
Test models (8 projects, not yet run)
User models (1 project)
First Project (2 models)
includes
models (2 models)
Imported model.gaml (1 experiment)
MyFirstModel.gaml (1 experiment)

MyFirstModel.gaml
Add experiment
1/*-
2 * Name: MyFirstModel
3 * Author: ben
4 * Description:
5 * Tags: Tag1, Tag2, TagN
6 */
7
8 model MyFirstModel
9
10 import "Imported_model.gaml"
11
12@global {
13    Markers at this line
14    This definition of shape supersedes the one in imported file
15    Imported%0model.gaml
16    This definition of shape supersedes the one in built-in species
17    agent
18    output {
19}
20
21

Imported model.gaml
Add experiment
Error(s) detected. Impossible to run any experiment.
1/*
2 *
3 */
4
5 model NewModel
6
7@global {
8
9    geometry shape <- circle{100,100};
10
11@ init {
12    create people number: 10;
13 }
14
15
16 species people ;
17
18 experiment exp1 type:gui {}

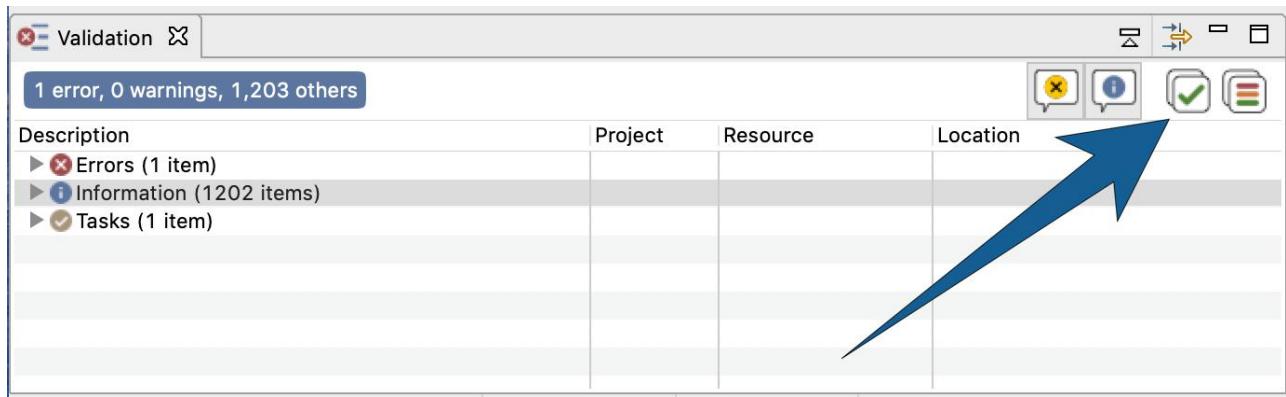
Validation
1 error, 0 warnings, 1,203 others
Description
Project Resource Location
Writable Insert 13 : 35 82M of 507M
Errors (1 item)
Information (1202 items)
Tasks (1 item)

```

Cleaning models

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

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



Version: 1.9.1

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.

Generic knowledge to start GAMA Headless

There are two ways to run a GAMA experiment in headless mode: using a dedicated bash wrapper (recommended) or directly from the command line.

Bash Wrapper

The file can be found in the `headless` directory located inside the [GAMA's installed folder](#). It is named `gama-headless.sh` on macOS and Linux, or `gama-headless.bat` on Windows.

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

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

- For example (using the provided sample), navigate in your terminal to the `headless` folder inside your GAMA root folder and type:

```
bash gama-headless.sh 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 gama-headless command must be `$GAMA_PATH/headless`

Java Command

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

- with:
 - `$GAMA_CLASSPATH` GAMA classpath: contains the 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.

Version: 1.9.1

Launching Experiments from the User Interface

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

Table of contents

- [Launching Experiments from the User Interface](#)
 - [From an Editor](#)
 - [From the Navigator](#)
 - [Running Experiments Automatically](#)
 - [Running Several Simulations](#)

From an Editor

As already mentioned in [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.

```

286 parameter 'Rate of diffusion of the signal' var: diffusion_rate category: 'Model';
287 parameter 'Use icons for the agents' var: icons category: 'Display';
288 parameter 'Display state of agents' var: state category: 'Display';
289 output {
290   display Ants type: opengl synchronized: true;
291   image terrain;
292   agents "Grid" transparency: 0.4 value: ant_grid where (each.food > 0) or (each.read > 0) or (each.is_nest);
293   species ant aspect: info;
294 }
295 }
296 //Complete experiment that will inspect all ants in a table
297 experiment "3D View" type: gui {
298   parameter 'Number' var: ants_number init: 30 unit: 'ants' category: 'Environment and Population';
299   parameter 'Grid dimension' var: gridsize init: 100 unit: '(number of rows and columns)' category: 'Environment and Population';
300   parameter 'Number of food depots' var: number_of_food_places init: 5 min: 1 category: 'Environment and Population';
301
302   output {
303     display Ants3D type: opengl show_fps: true;
304     grid_ant_grid elevation: grid_values triangulation: true texture: terrain refresh: false;
305     agents "Trail" transparency: 0.7 position: { 0.05, 0.05, 0.02 } size: { 0.9, 0.9 } value: (ant_grid as list) where ((each.
306     species ant position: { 0.05, 0.05, 0.025 } size: { 0.9, 0.9 } aspect: threeD;
307     light 1 type:point color:#yellow position:(world.shape.width+0.5 - world.shape.width*1.5,world.shape.width*0.5,world.shape.
308   }
309 }
310
311 experiment "3D with mem" type: memorize parent: "3D View";
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349

```

For each of those launching buttons, you can see different pictograms, showing the type of experiment. [The various kinds of experiment are described in this page.](#)

```

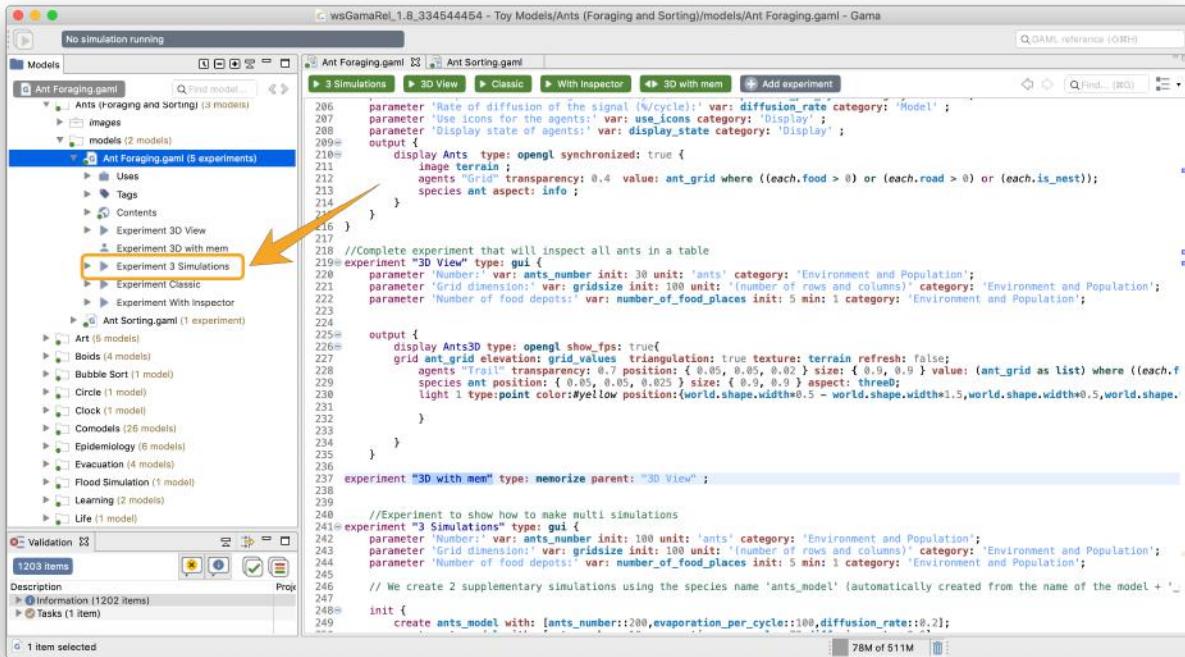
156
157   if(display_state) {
158     draw state color: #yellow font: font("Helvetica" 18, #bold) at: my_location
159   }
160 }
161
162 aspect threeD {
163   draw ant3D_shape size: {7,5} at: (location + {0,0,1}) rotate: heading ;
164 }
165
166 aspect icon {

```

From the Navigator

You can also launch your experiments from the navigator, by expanding a model and double-clicking on one of the experiments available (the number of experiments for

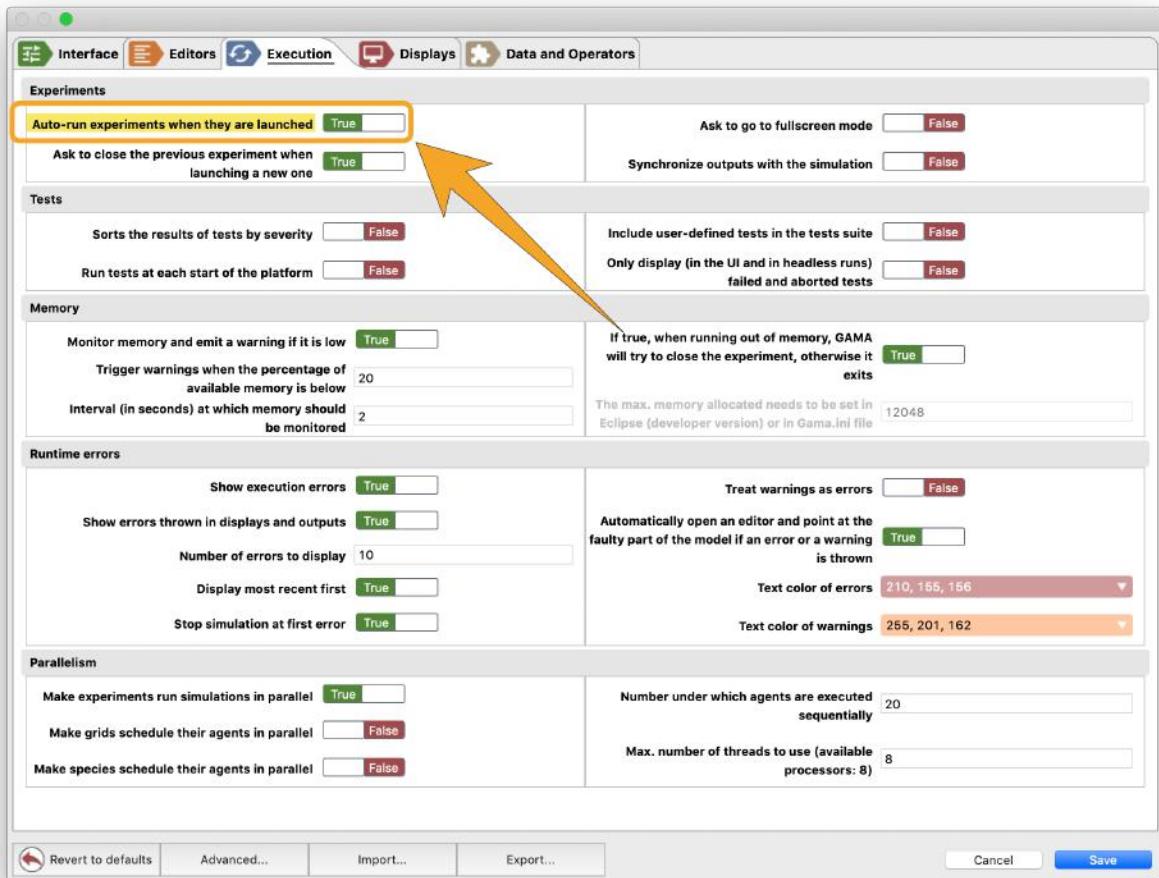
each model is visible also in the navigator). As for the editor, the various types of experimentations are differentiated by a pictogram.



Running Experiments Automatically

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

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

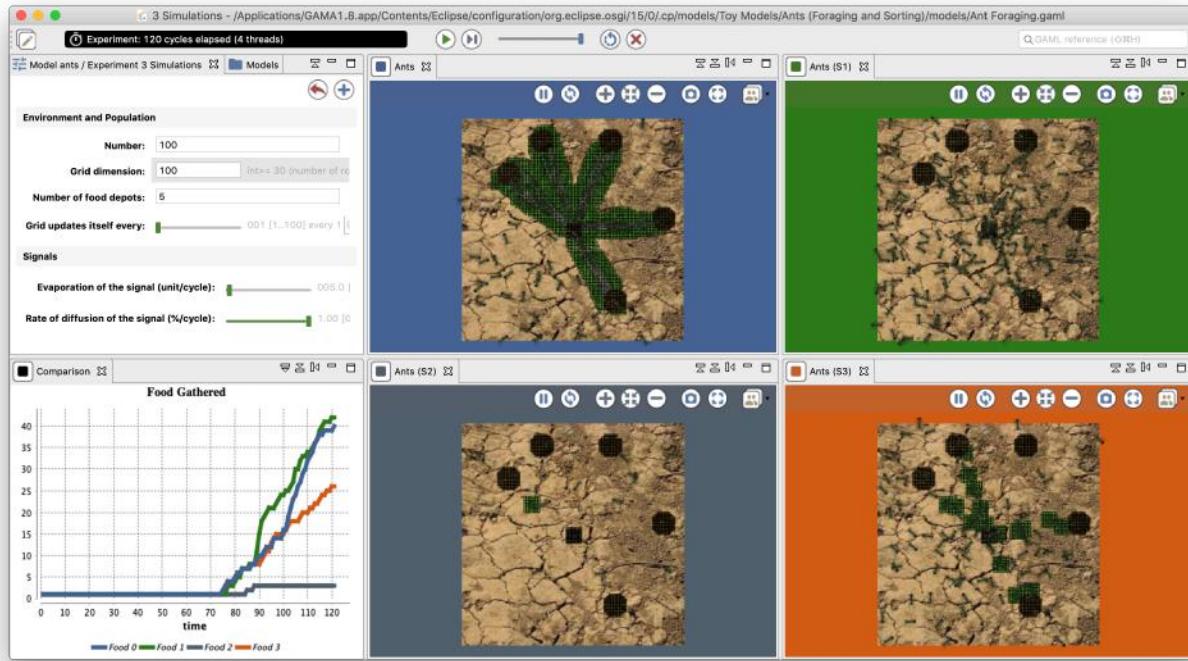


When the autorun is set in the Preferences, all the experiments in the workspace will be in autorun mode. If you want to activate this option only for a single experiment, it can be done programmatically by adding the `autorun:` to the `experiment` statement as detailed in this page.

Running Several Simulations

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

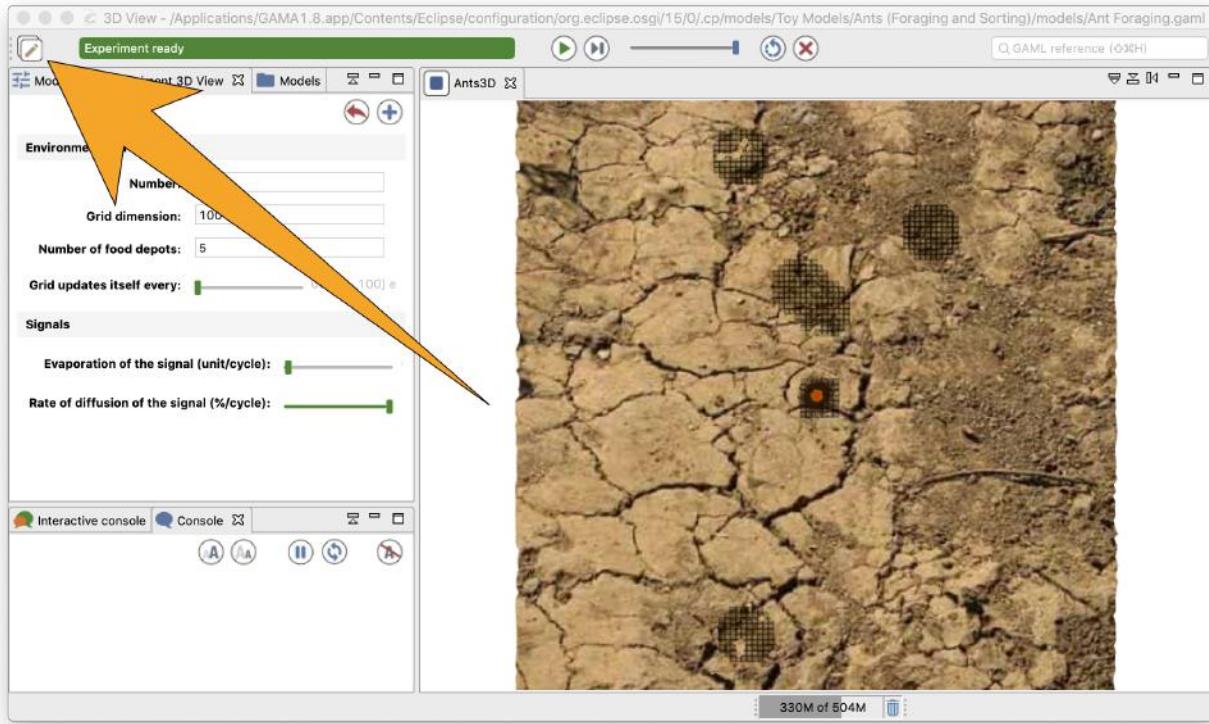
To run several simulations, you have to write it directly in your model.



Version: 1.9.1

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 available in this perspective (below the parameter view), though, and it is still possible to [edit models](#) in it, but it is considered as good practice to use each perspective for what it has been designed for: editing models in the **Modeling perspective** and running simulations in the **Simulation perspective**. Switching perspectives is easy. The small button in the top-left corner of the window allows to switch back and forth the two perspectives.



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

Version: 1.9.1

Controls of experiments

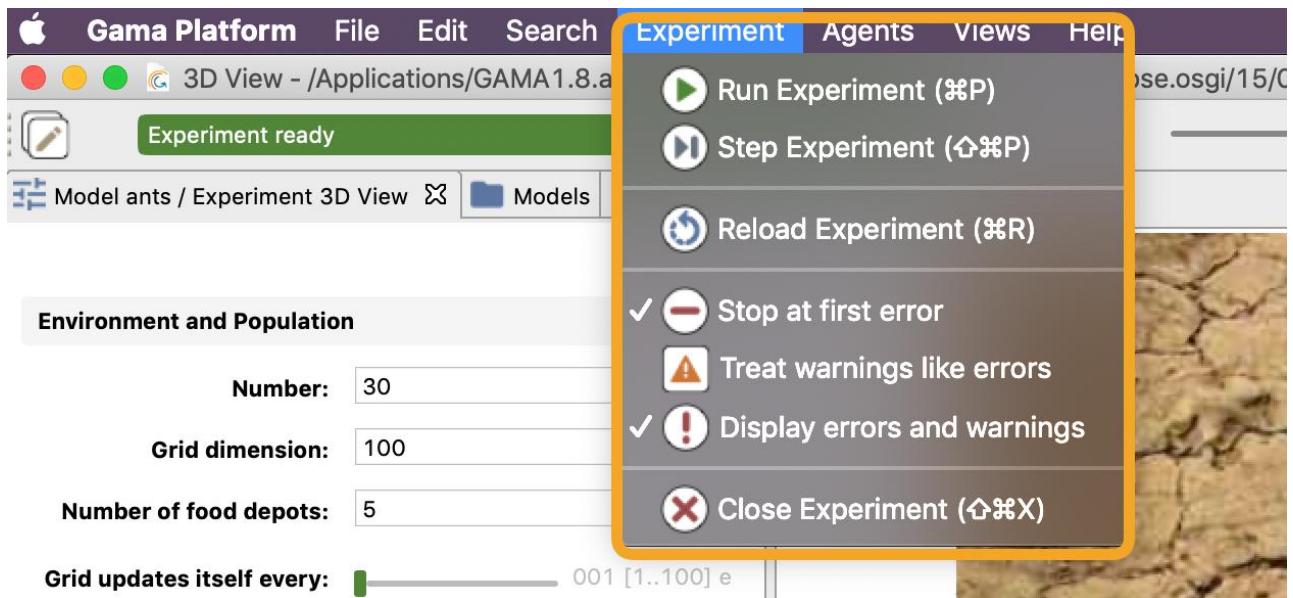
The simulation perspective adds on the user interface a number of new menus and commands (i.e. buttons) that are specific to experiment-related tasks.

Table of contents

- [Controls of experiments](#)
 - [Experiment Menu](#)
 - [Agents Menu](#)
 - [General Toolbar](#)

Experiment Menu

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

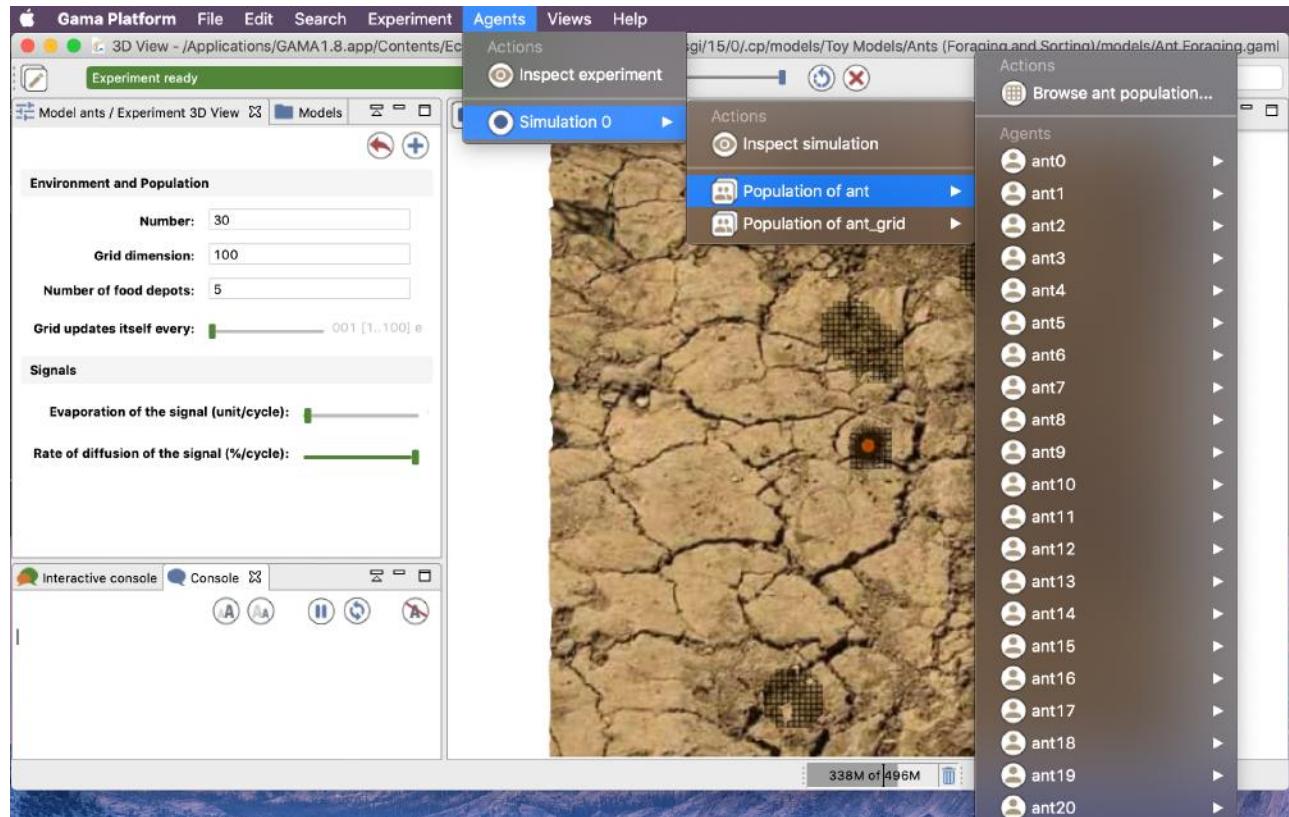


- **Run/Pause Experiment:** allows to run or pause the experiment depending on its current state.
- **Step Experiment:** runs the experiment for one cycle and pauses it after.
- **Reload Experiment:** stops the current experiment, deletes its contents and reloads it, **taking into account the parameters 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 like 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 errors and warning:** 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](#).
- **Close Experiment:** 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

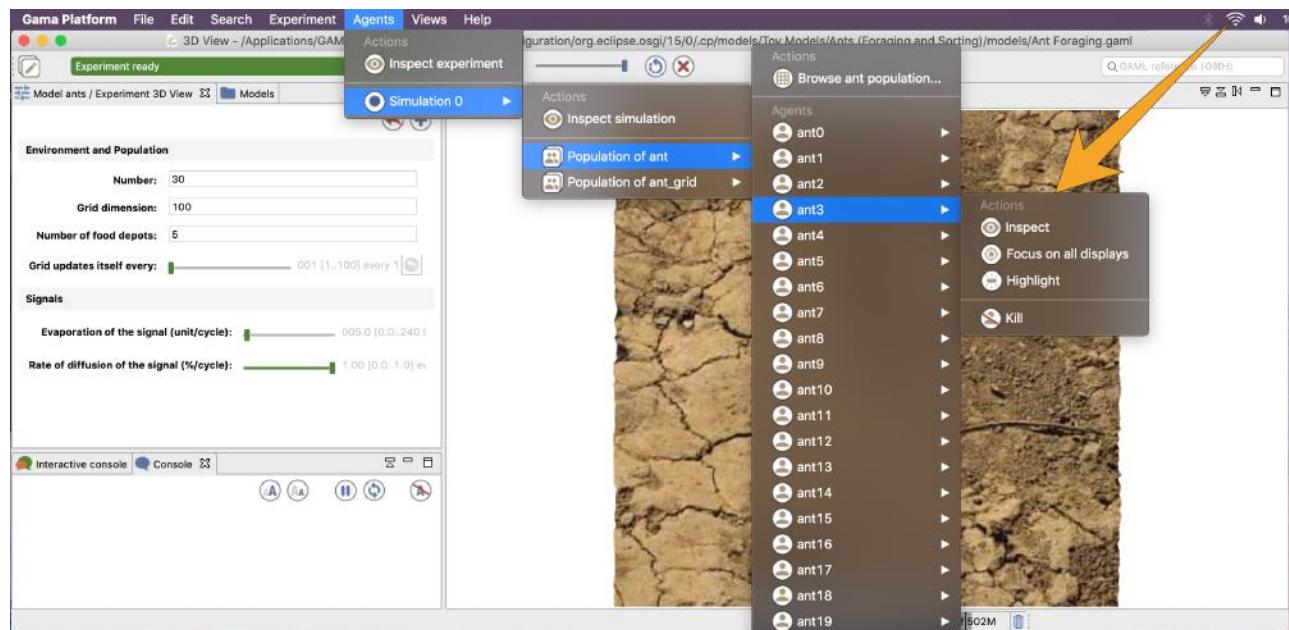
A second menu is added in the simulation perspective: "**Agents**". This menu allows for easy access to the different agents that populate an experiment.



This hierarchical menu is always organized in the same way, whatever the experiment being run. A first level is dedicated to the current top-level **experiment** agent: it allows the modeler to inspect the agent itself and to [browse](#) its population(s) (i.e. the **simulation** agents). A second level lists the "micro-populations" present in each simulation agent and allows to inspect the agent itself. And the third level will give access to an overview of all the agents of the population in a table ("Browse ant population...") and to each individual agent in these

populations. This organization is, of course, recursive: if these agents are themselves, hosts of micro-populations, they will be displayed in their individual menu.

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

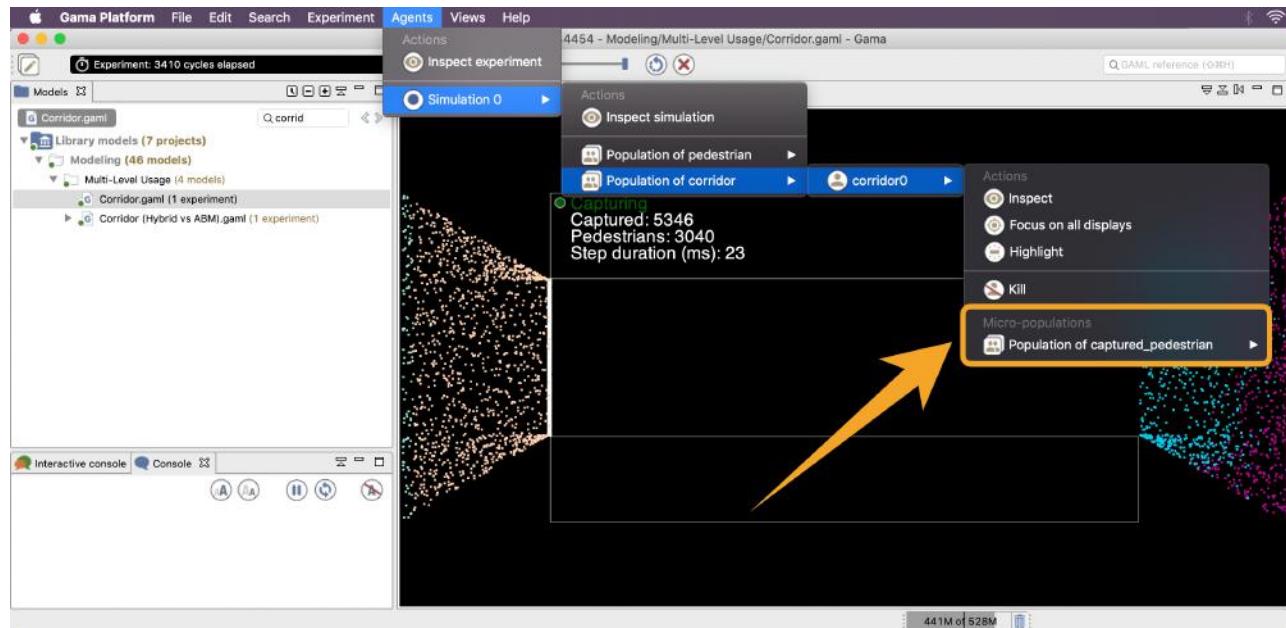


These are the 4 actions that will be there most of the time:

- **Inspect:** open an [inspector](#) on this agent.
- **Focus on all displays:** this option is not accessible if no displays are defined. Makes all the displays zoom on the selected agent (if it is displayed) so that it occupies the whole view.
- **Highlight:** makes this agent the current "highlighted" agent, forcing it to appear "highlighted" in all the displays that might have been defined.
- **Kill:** destroys the selected agent and disposes of it. **Use this command with caution**, as it can have undesirable effects if the agent is currently executing its

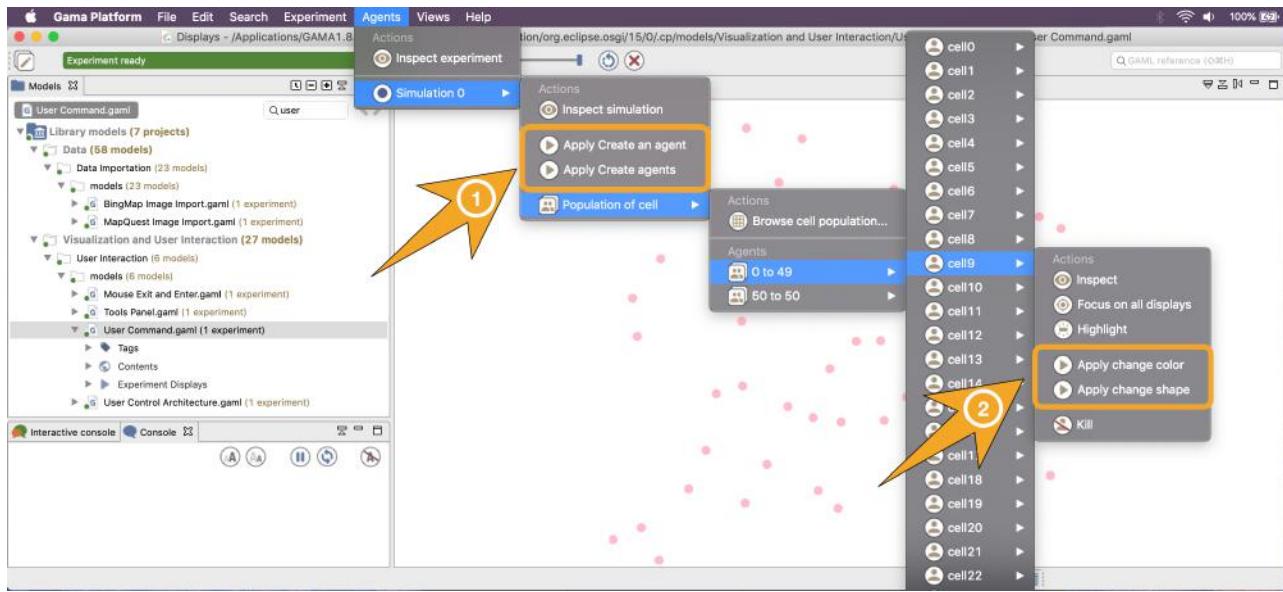
behavior.

If an agent hosts other agents (it is the case in [multi-level architecture](#)), you can access to the micro-population quite easily (e.g. in the model [Library models/Modeling/Multi-Level Usage/Corridor.gaml](#)):



If [user commands](#) are defined for a species (for example in the existing model [Library models/Visualization and User Interaction/User Interaction/User Command.gaml](#)), their individuals' menu will look like the following. Notice that in this model two species have user command:

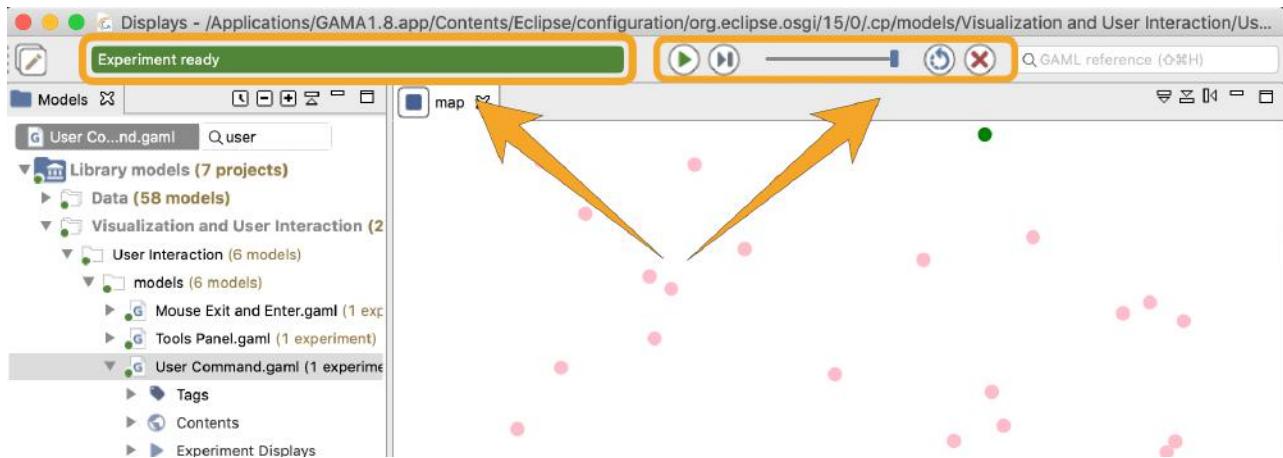
1. the simulation agent (2 user commands are defined in the `global` section of the model),
2. the ants agents (2 user commands defined in the `species` definition section).



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 four buttons already present in the **experiment menu** (namely, "Play/Pause Experiment", "Step Experiment", "Reload Experiment" and "Close Experiment"), which do not need to be detailed here, and two new controls ("Experiment status" and "Cycle Delay"), which are explained below.



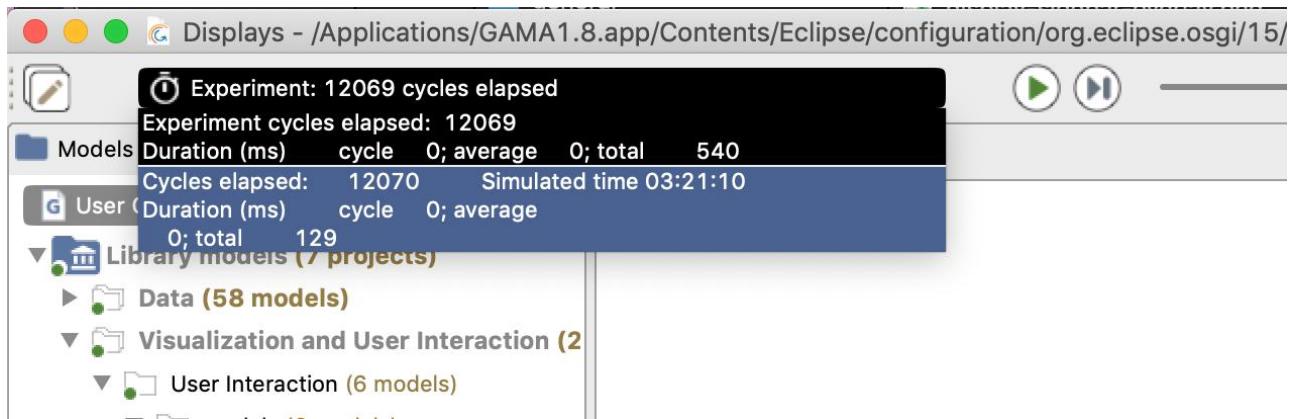
While opening an experiment, the status will display some information about what's going on. For instance, GAMA is busy instantiating the agents or opening the displays.



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.



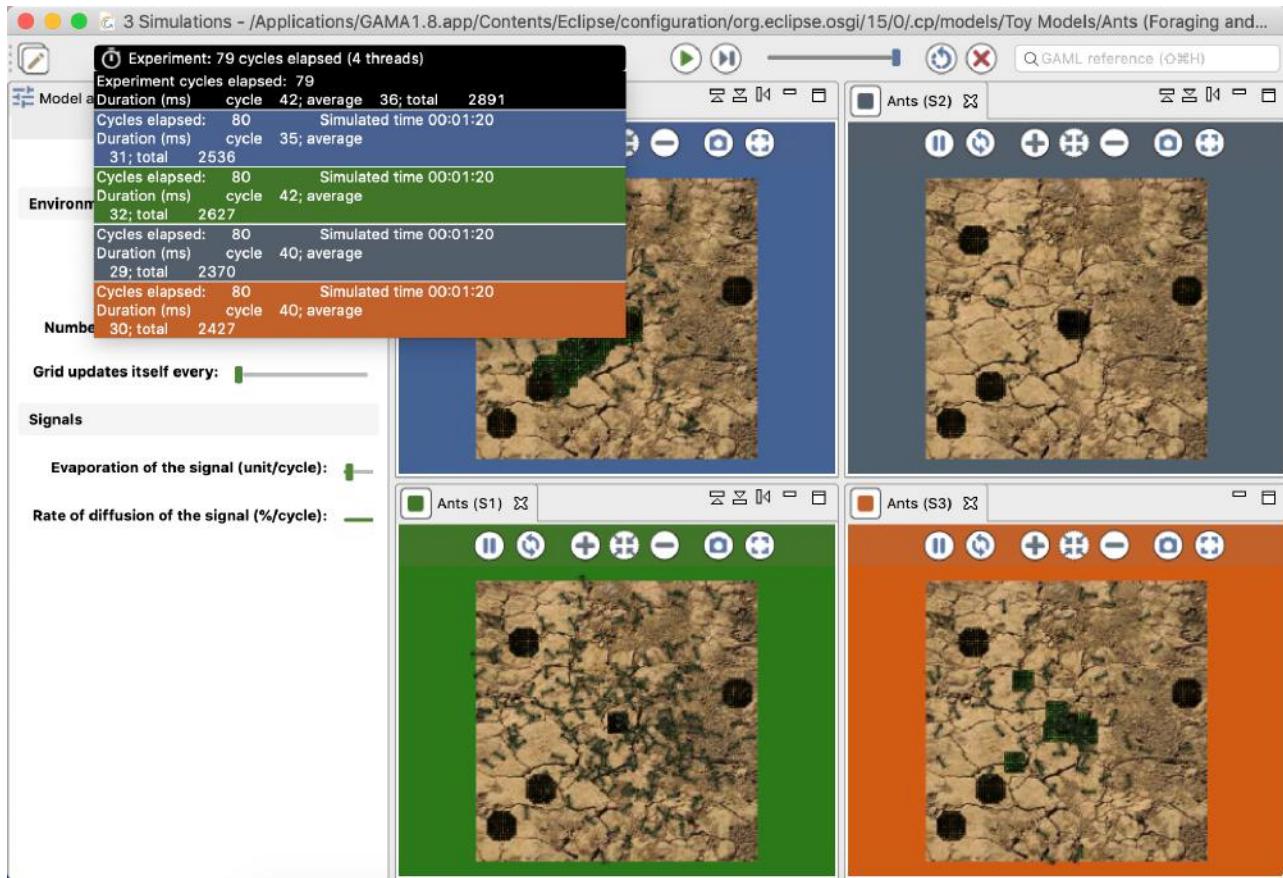
Hovering over the status produces more accurate information about the internal clock of the simulation.



When we launch an experiment, an experiment agent is created with its own internal clock. It will then create 1 (or more) simulation agent(s). The toolbar provides thus information about both the experiment agent and the simulation(s), from top to bottom:

- the number of cycles elapsed,
- the simulated time already elapsed (in the example above, one cycle lasts one second of *simulated time*) for the simulation agents only,
- the duration of cycle in milliseconds,
- the average duration of one cycle (computed over the number of cycles elapsed),
- the total duration, so far, of the simulation (still in milliseconds).

In the case of a multi-simulation (i.e. an experiment launching several simulations), one block per simulation is displayed.



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 0s (its initial position) to 1s. 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](#).

Version: 1.9.1

Parameters View

In the case of an [experiment](#), the modeler can [define the parameters](#) s/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.

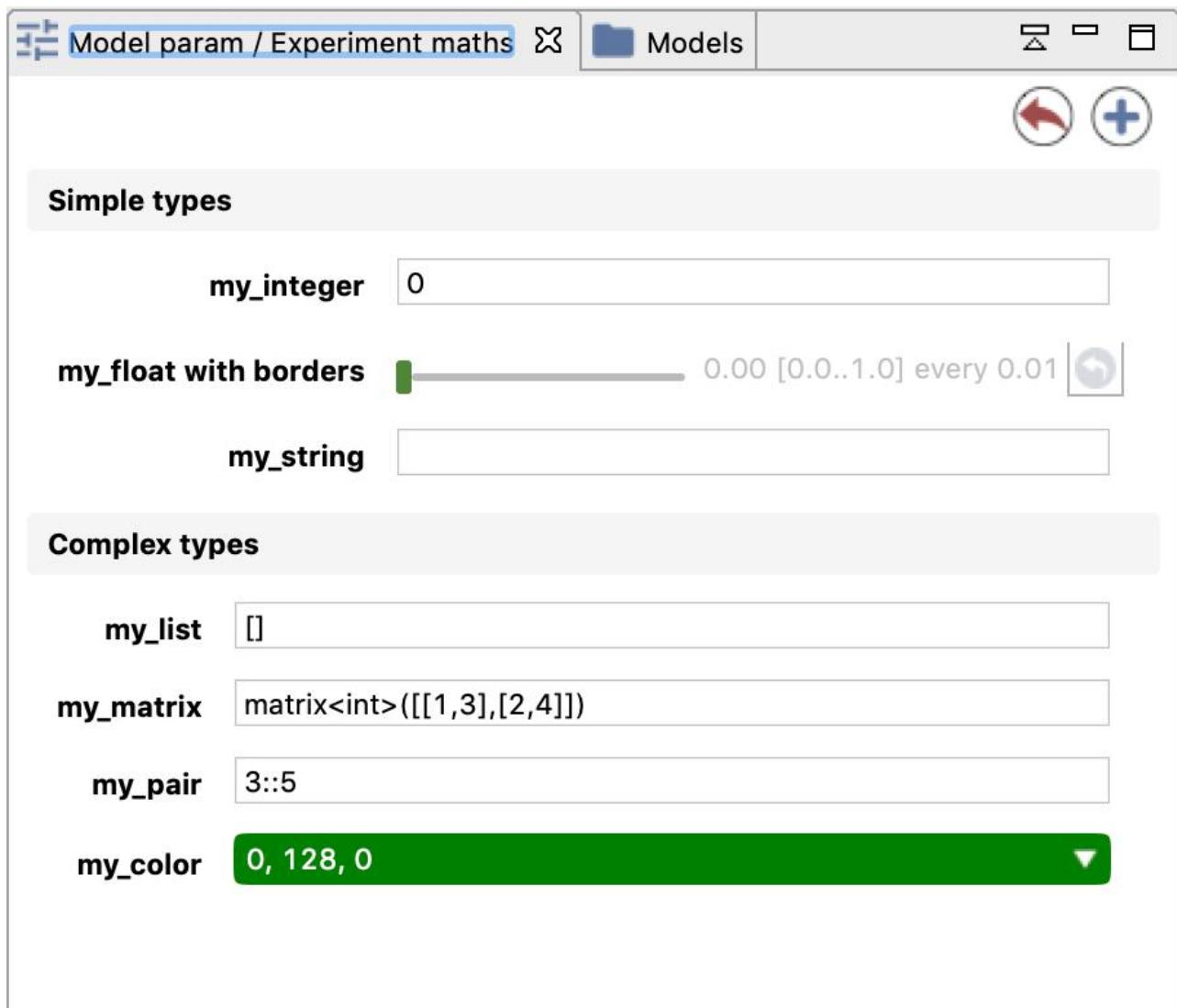
It important to notice that all modifications made in the parameters will be taken into account in case of simulation reload only. Launch of a new experiment from the model perspective will erase the modifications.

Table of contents

- [Parameters View](#)
 - [Parameters View](#)
 - [Modification of parameters values](#)

Parameters View

The modeler can [define 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: e.g. for string parameters, a simple text box will be displayed whereas a color selector will be available for color parameters. It can also depend on the way the parameter is defined: an integer or a float parameter will be displayed with a slider if its min and max values are defined, and a simple text field otherwise. The parameter's value displayed is the initial value provided to the variables associated with the parameters in the model.

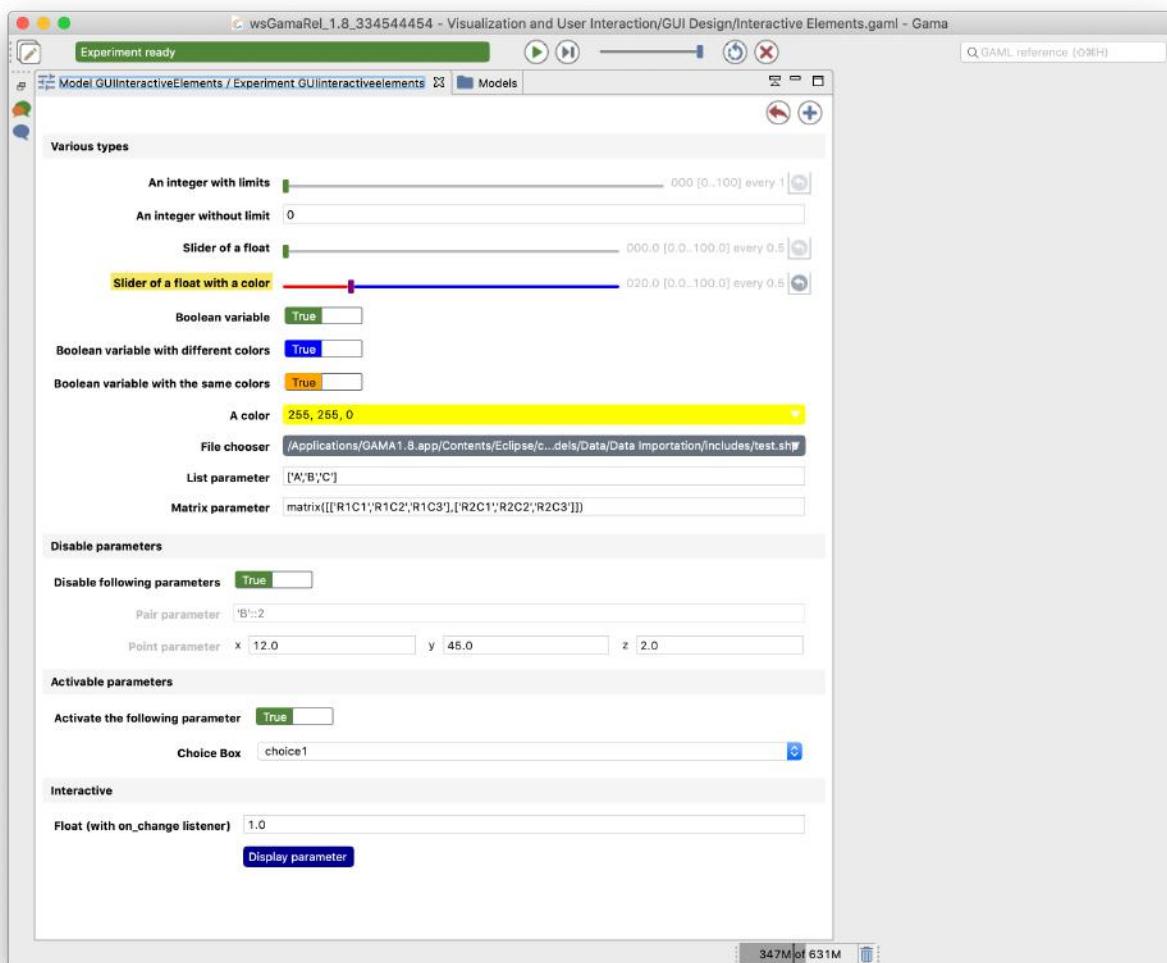


The above parameters view is generated from the following code:

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

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.

The model `Library models > Visualization and User Interaction > GUI Design > Interactive Elements.gaml` exemplifies all the possible way of displaying parameters (and other interactive elements). Even interactive elements (buttons or parameters will a behavior associated with a value change) can be added to the Parameter View.



Modification of parameters values

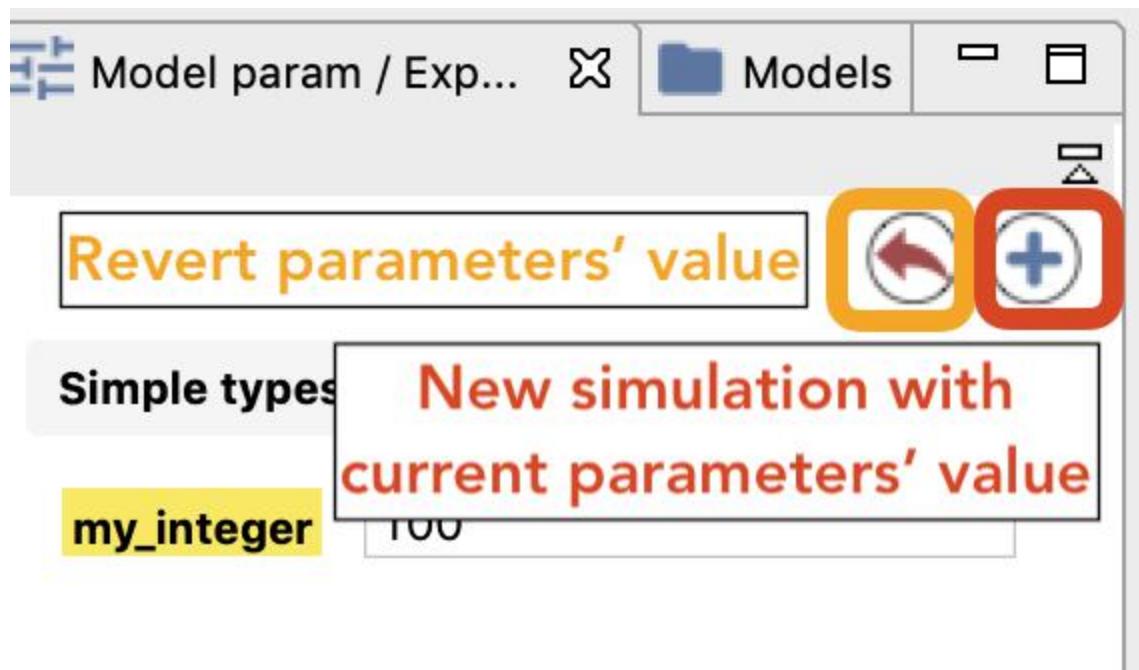
The modeler can modify the parameter values. After modifying the parameter values, you can reload the simulation by clicking on the top-right circular arrow button.

It is important to understand that modification of a parameter value is immediately taken into account in the simulation: **the value of the variable in the model is modified. BUT** the effect on the simulation will depend on the use of this variable in the model:

- if the variable is used at initialization of the simulation (e.g. it contains the number of agents to be created), then a change of its value will not be visible in the simulation running as it is not used,
- if the variable is used during the simulation (e.g. the pheromones evaporation rate in ants models), a change in the parameter view will have an impact on the simulation behavior.

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

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



Version: 1.9.1

Inspectors and monitors

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

- agent browser
- agent inspector

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

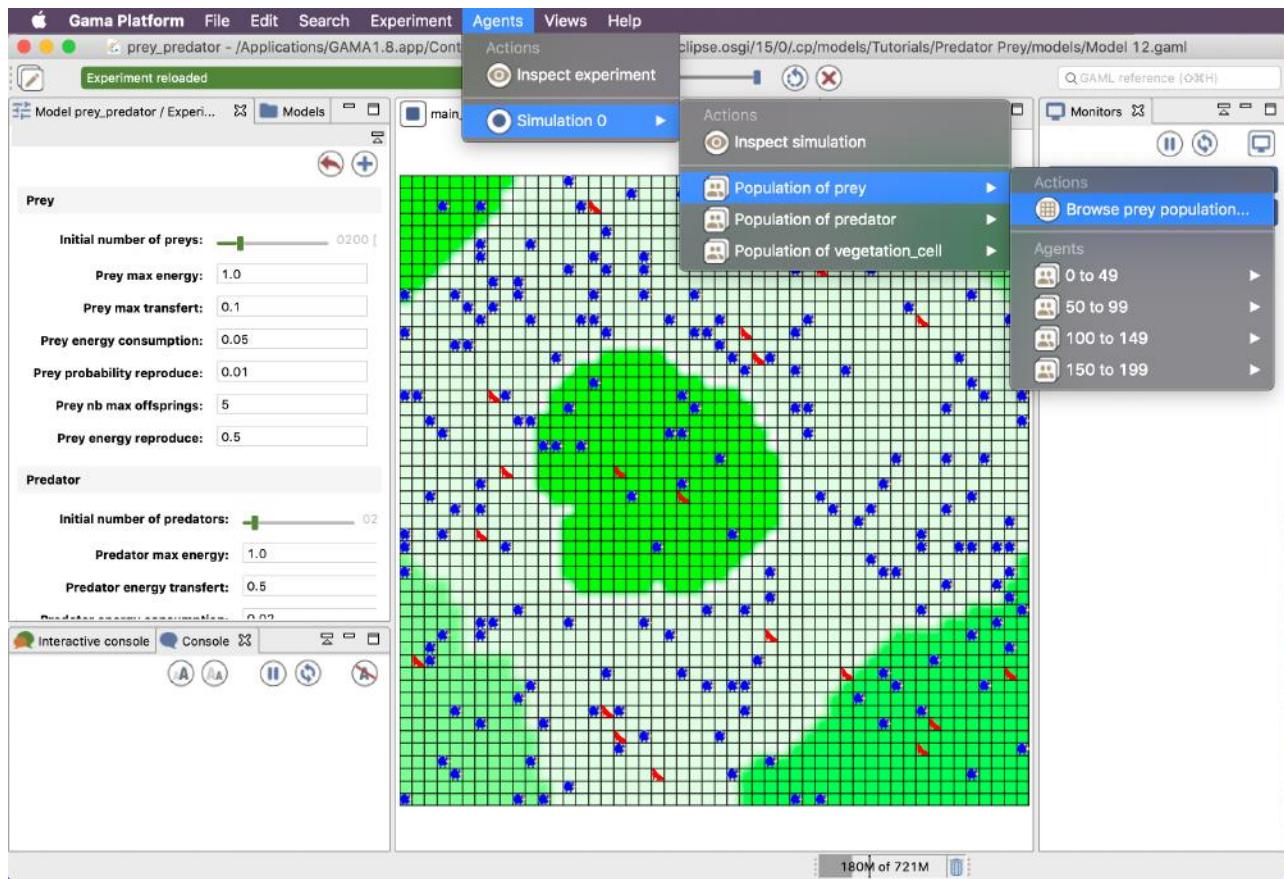
Table of contents

- [Inspectors and monitors](#)
 - [Agent Browser](#)
 - [Agent Inspector](#)
 - [Monitor](#)

Agent Browser

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

The agent browser is available through the **Agents** menu.



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

Browse(0): population of prey

Attributes	#	color	energy	energy_consum	energy_reproduce	max_energy	ma
agents	0	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	1	°blue	0.05	0.05	0.5	1.0	0.1
color	2	°blue	0.0480392156...	0.05	0.5	1.0	0.1
energy	3	°blue	0.0480392156...	0.05	0.5	1.0	0.1
energy_consum	4	°blue	0.05	0.05	0.5	1.0	0.1
energy_reproduce	5	°blue	0.0480392156...	0.05	0.5	1.0	0.1
host	6	°blue	0.05	0.05	0.5	1.0	0.1
location	7	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	8	°blue	0.0480392156...	0.05	0.5	1.0	0.1
max_energy	9	°blue	0.05	0.05	0.5	1.0	0.1
max_transfert	10	°blue	0.0480392156...	0.05	0.5	1.0	0.1
members	11	°blue	0.0480392156...	0.05	0.5	1.0	0.1
myCell	12	°blue	0.0480392156...	0.05	0.5	1.0	0.1
my_icon	13	°blue	0.0480392156...	0.05	0.5	1.0	0.1
name	14	°blue	0.05	0.05	0.5	1.0	0.1
nb_max_offsprings	15	°blue	0.05	0.05	0.5	1.0	0.1
peers	16	°blue	0.05	0.05	0.5	1.0	0.1
proba_reproduce	17	°blue	0.0480392156...	0.05	0.5	1.0	0.1
shape	18	°blue	0.0480392156...	0.05	0.5	1.0	0.1
size	19	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	20	°blue	0.05	0.05	0.5	1.0	0.1
	21	°blue	0.05	0.05	0.5	1.0	0.1
	22	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	23	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	24	°blue	0.05	0.05	0.5	1.0	0.1
	25	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	26	°blue	0.05	0.05	0.5	1.0	0.1
	27	°blue	0.05	0.05	0.5	1.0	0.1
	28	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	29	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	30	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	31	°blue	0.05	0.05	0.5	1.0	0.1
	32	°blue	0.05	0.05	0.5	1.0	0.1
	33	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	34	°blue	0.05	0.05	0.5	1.0	0.1
	35	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	36	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	37	°blue	0.05	0.05	0.5	1.0	0.1
	38	°blue	0.05	0.05	0.5	1.0	0.1
	39	°blue	0.0480392156...	0.05	0.5	1.0	0.1
	40	°blue	0.0480392156...	0.05	0.5	1.0	0.1

By clicking on the right mouse button on a line, it is possible to perform some actions on the corresponding agent (the same actions as when we right-click on it in a display).

The Browse view provides also two interesting additional features:

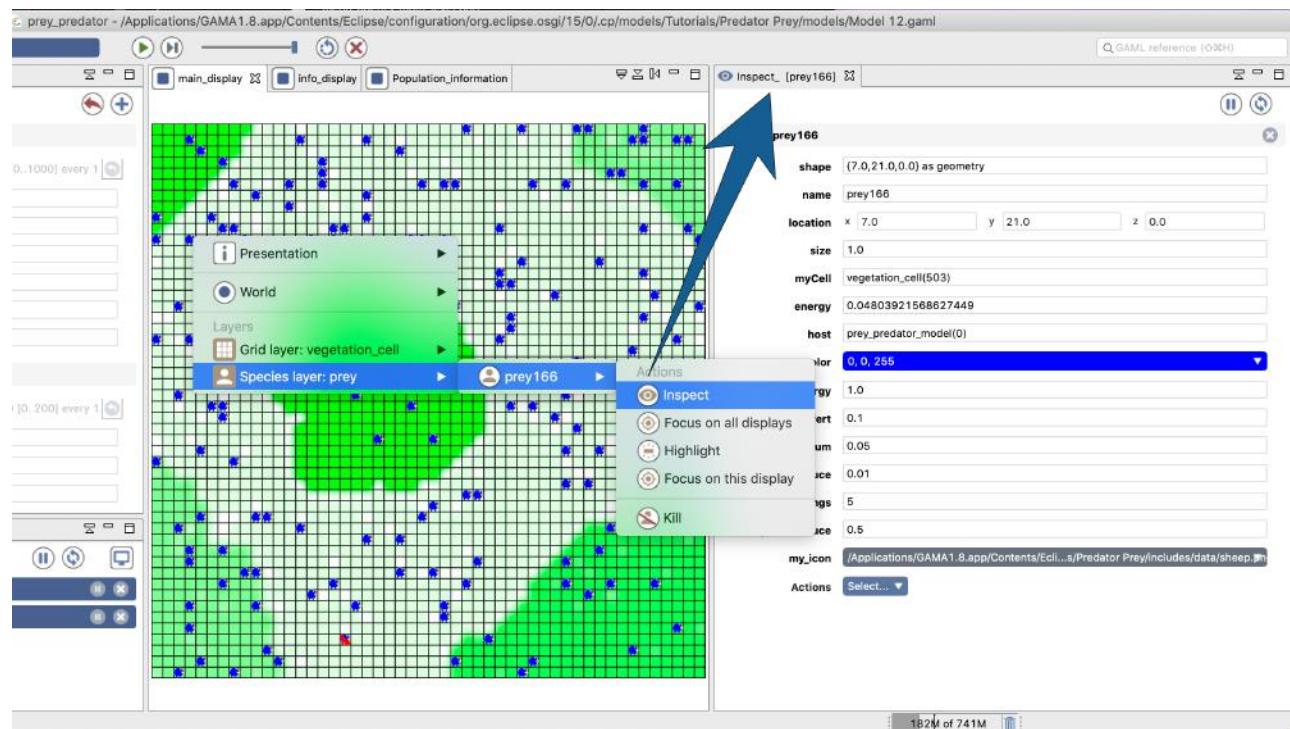
1. **Browse a species:** change the population displayed in the table.
2. **Save the agents and their attributes in a .csv file:** this allows the modeler to manipulate and analyze the agent population at will in external software.

Browse(3): population of prey

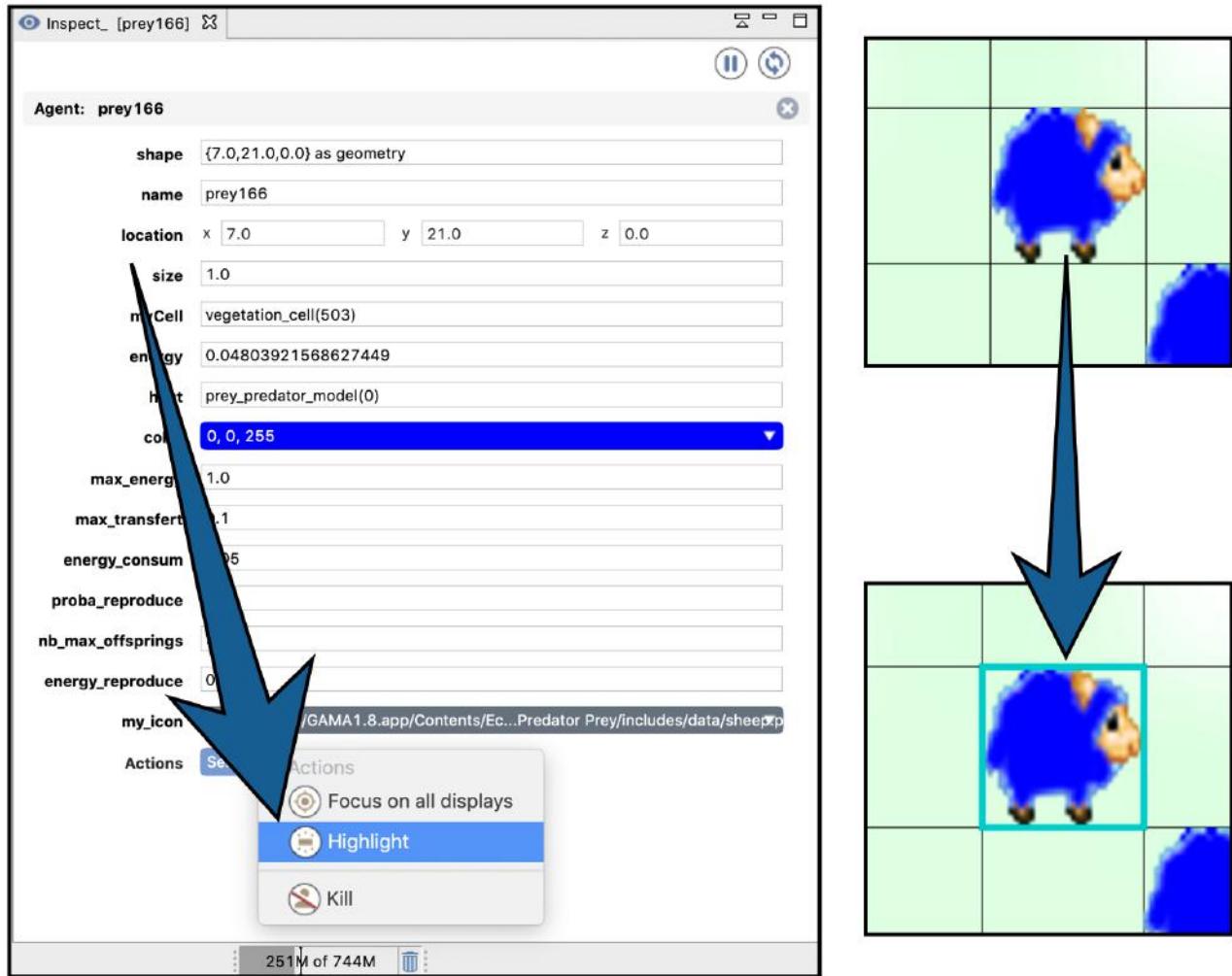
Attributes	#	color	energy	energy_cons
agents	0	°blue	0.0480392156...	0.05
color	1	°blue	0.05	0.05
energy	2	°blue	0.0480392156...	0.05
	3	°blue	0.0480392156...	0.05
	4	°blue	0.05	0.05
	5	°blue	0.0480392156...	0.05

Agent Inspector

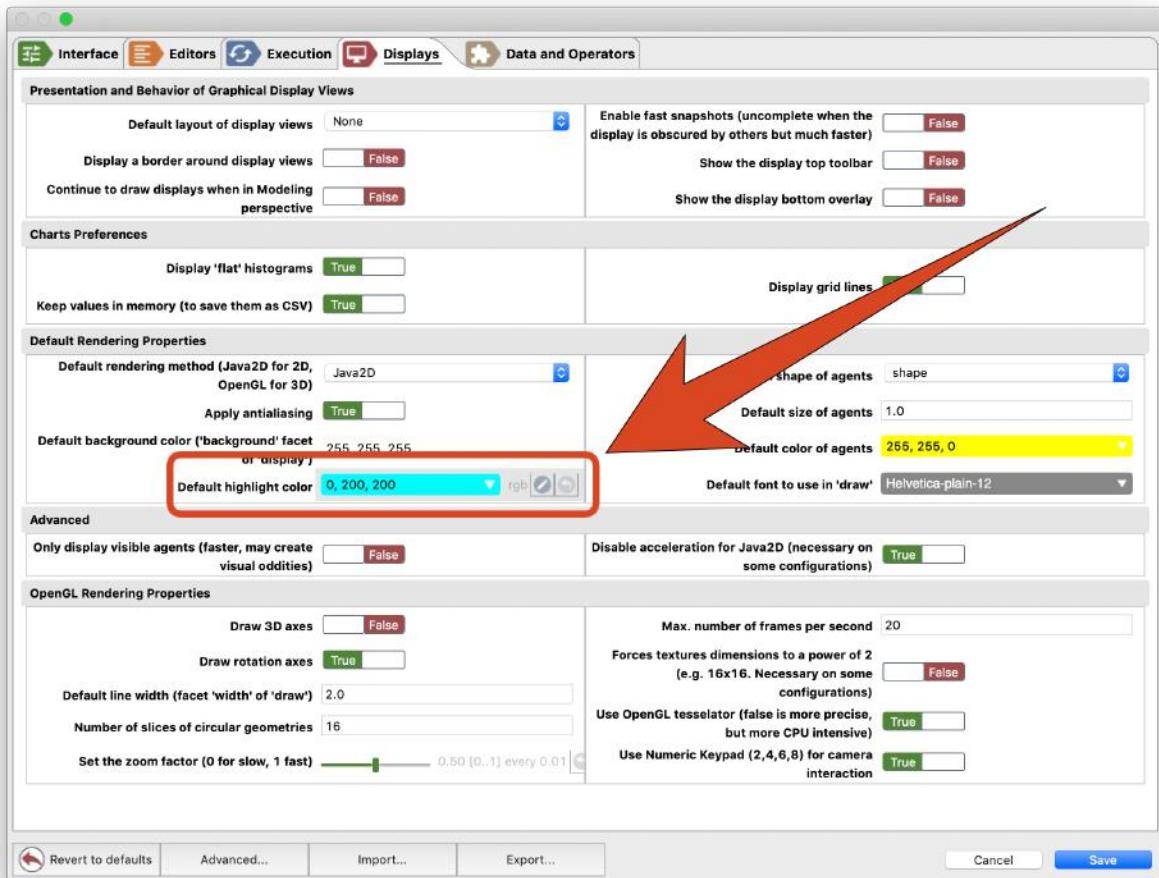
The agent inspector provides information about one specific agent. It also allows the modeler to change the values of its variables during the simulation. The agent inspector is available from the **Agents** menu, by right-clicking on a display, in the species inspector or when inspecting another agent.



It is possible to "highlight" the selected agent, to focus on it in all the displays, or to kill it.

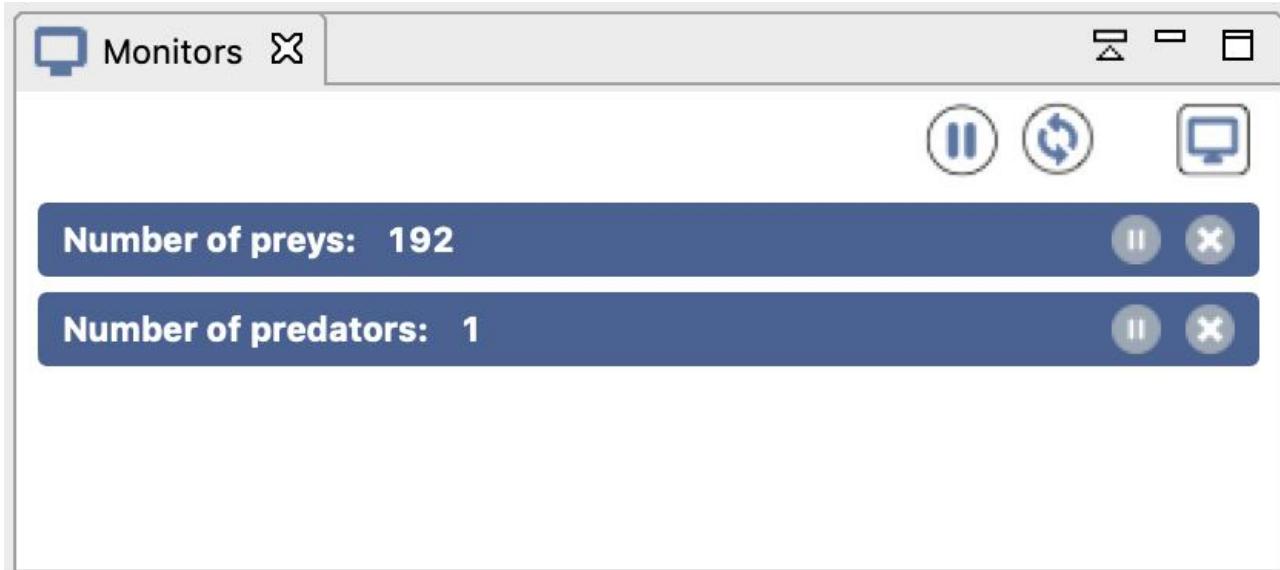


To change the color of the highlighted agent, go to Preferences/Display.



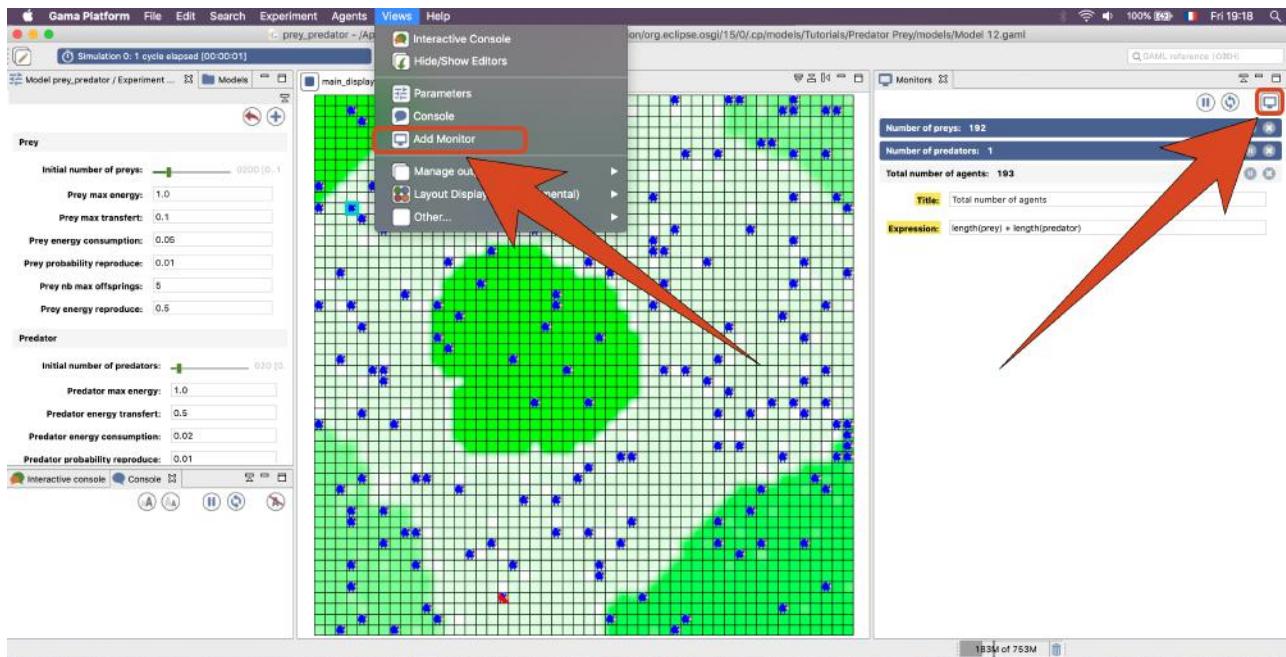
Monitor

Monitors allow the user to follow the value of a GAML expression. For instance, the following ones monitor the number of prey and predator agents during the simulation (the model is available in the Prey Predator tutorial). The monitor is updated at each simulation step.

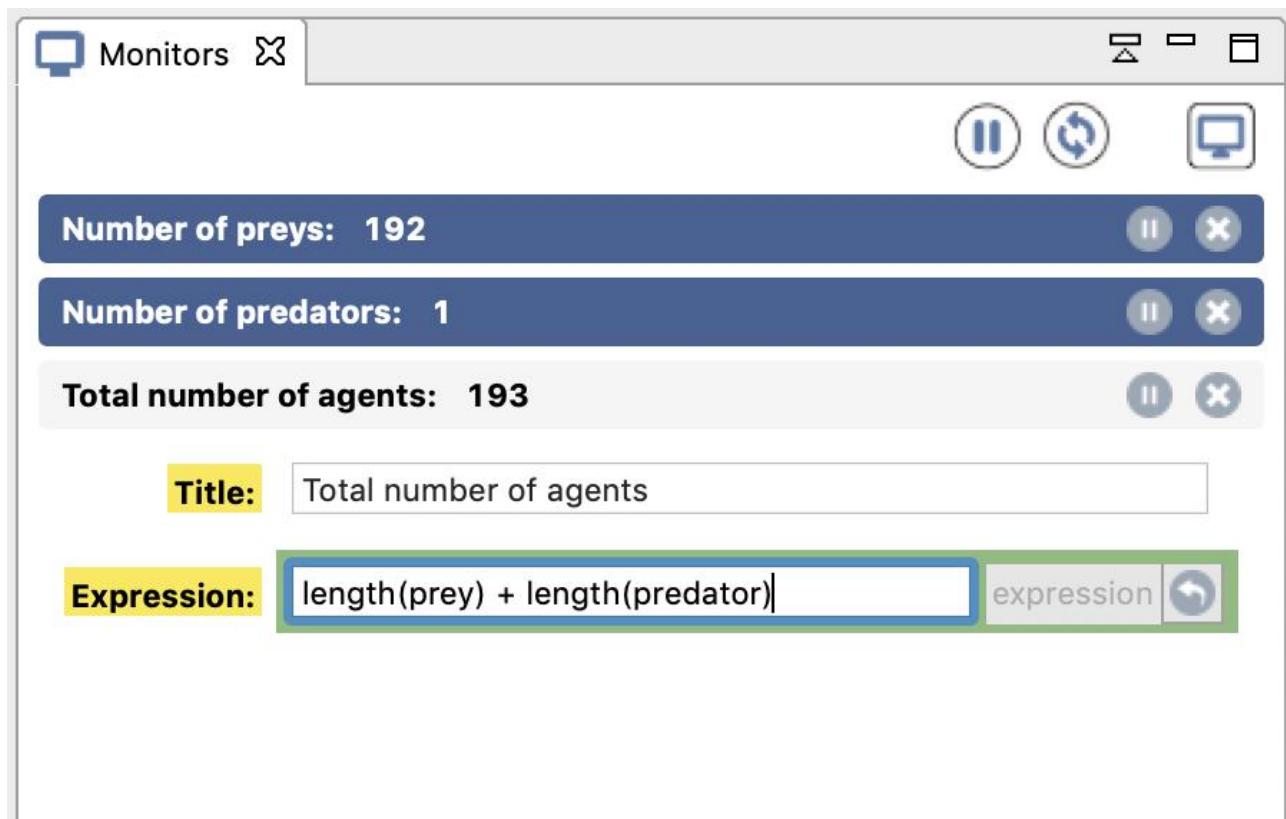


It is possible to define a monitor inside a model (see [this page](#)). It is also possible to define a monitor through the graphical interface.

To define a monitor, first choose **Add Monitor** in the **Views** menu (or by clicking on the icon in the Monitor view), then define the display legend and the expression to monitor. The expression is compiled when it is written in the text field: as long as the text field is surrounded by a red rectangle, it is incorrect. When the surrounding color becomes green, GAMA has accepted the expression and its value can be displayed in the monitor.



In the following example, we defined a monitor with the legend "Total number of agents" and its value is defined by the GAML expression computing the sum of the number of agents in each population: `length(prey) + length(predator)`.



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

Version: 1.9.1

Displays

GAMA allows modelers to define two kinds of displays in a GUI experiment:

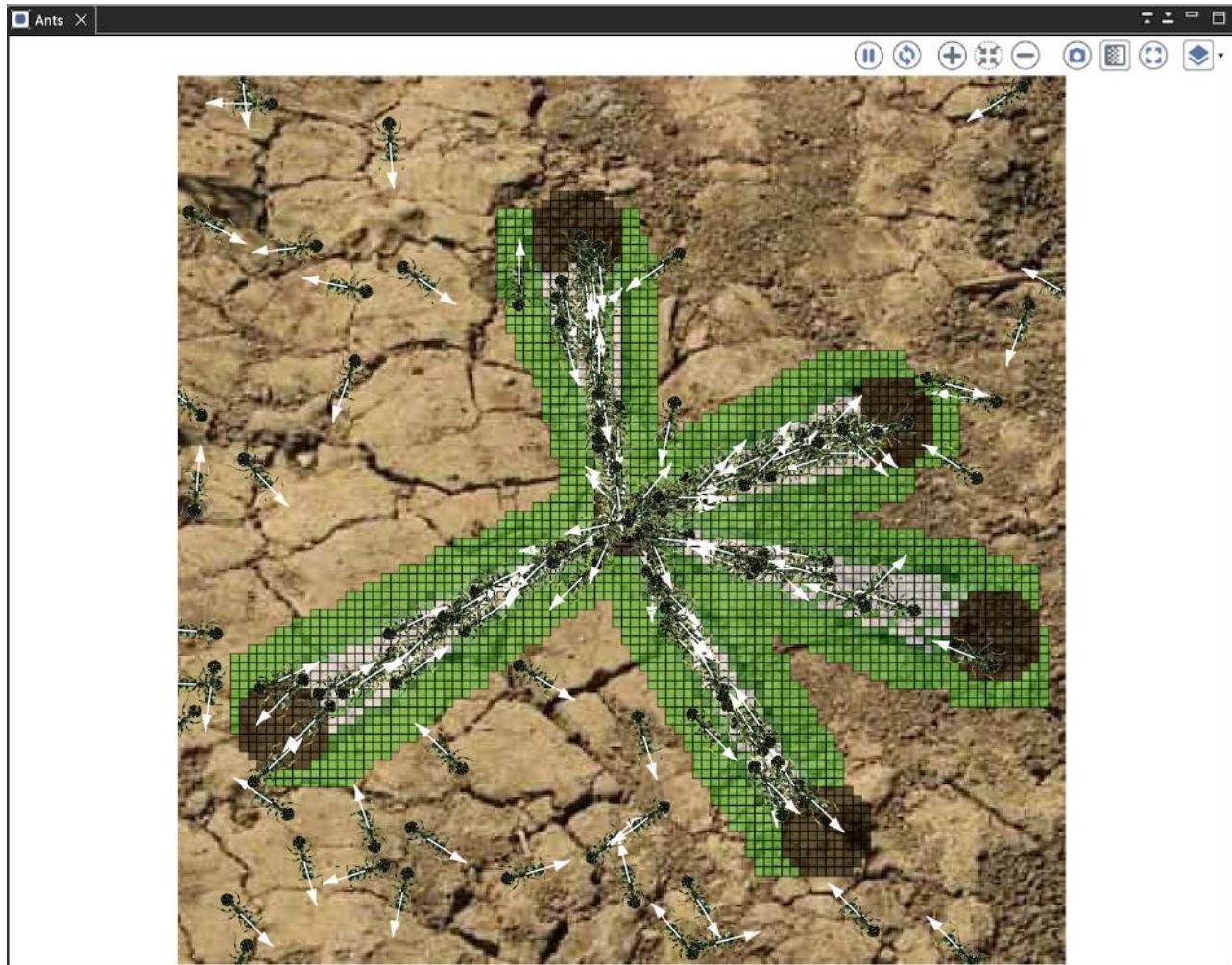
- java 2D displays
- OpenGL displays

These 2 displays allow modeler to display the same objects (agents, charts, texts ...). However, the OpenGL display offers extended features in particular in terms of 3D visualization and provides better performance for large scale simulation.

Classical displays (java2D)

The classical displays displaying any kind of content can be manipulated via the mouse (if no mouse event has been defined):

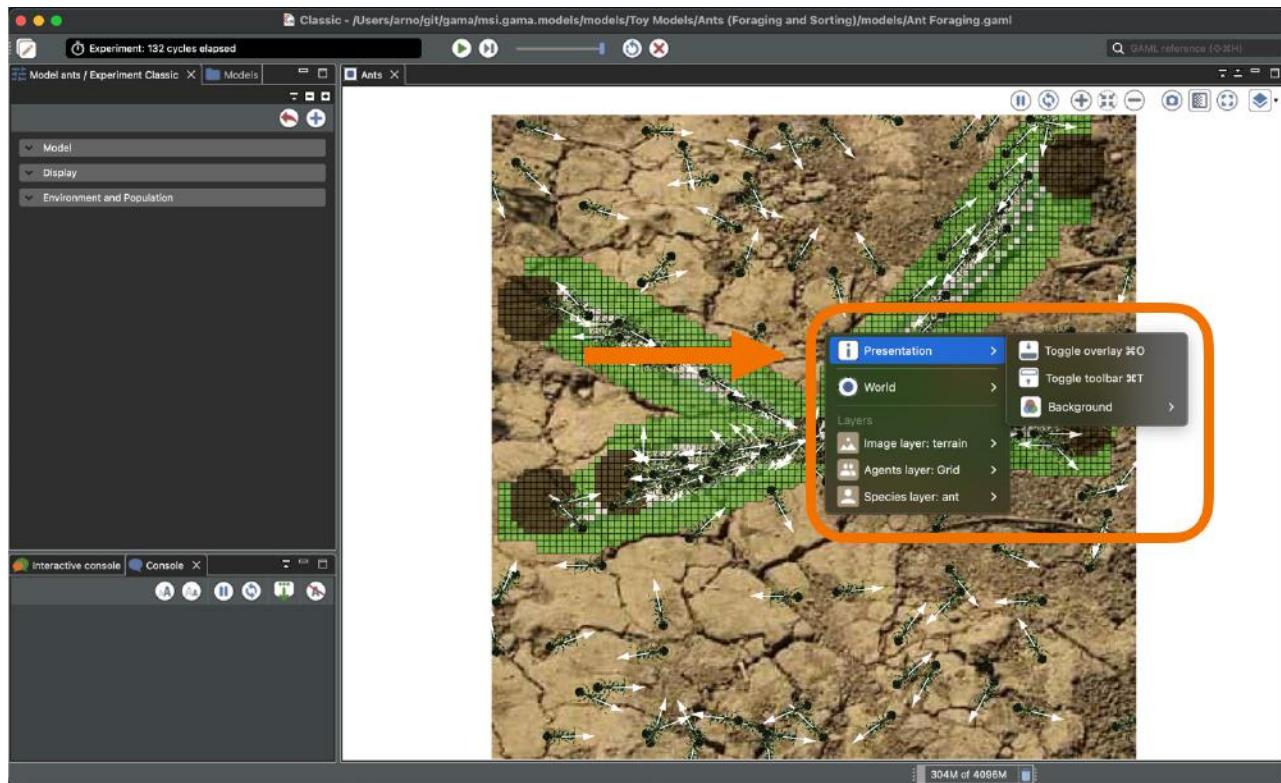
- the **mouse left** press and move allows to move the camera (in 2D),
- the **mouse right** click opens a context menu allowing the modeler to inspect displayed agents,
- the **wheel** allows the modeler to zoom in or out.



Each display provides several buttons to manipulate the display (from left to right):

- **Pause or resume the current view:** when pressed, the display will not be displayed anymore while the simulation is still running,
- **Synchronize,** when pressed, the display and the execution of the model are synchronized. Most of the time, this will reduce the speed of the simulation.
- **Zoom in,**
- **Zoom to fit view,**
- **Zoom out,**
- **Take a snapshot:** take a snapshot saved as a png image in the `snapshots` folder of the model folder.

- **Toggle antialias:** Antialiasing produces smoother outputs, but comes with a cost in terms of speed and memory used.
- **Toggle fullscreen ESC:** when pressed, the current view will be displayed in fullscreen. To exit this mode, press **ESC** key.
- **Browse through all displayed agents:** when pressed a [browse view will be open](#). Only the species displayed can be browsed.

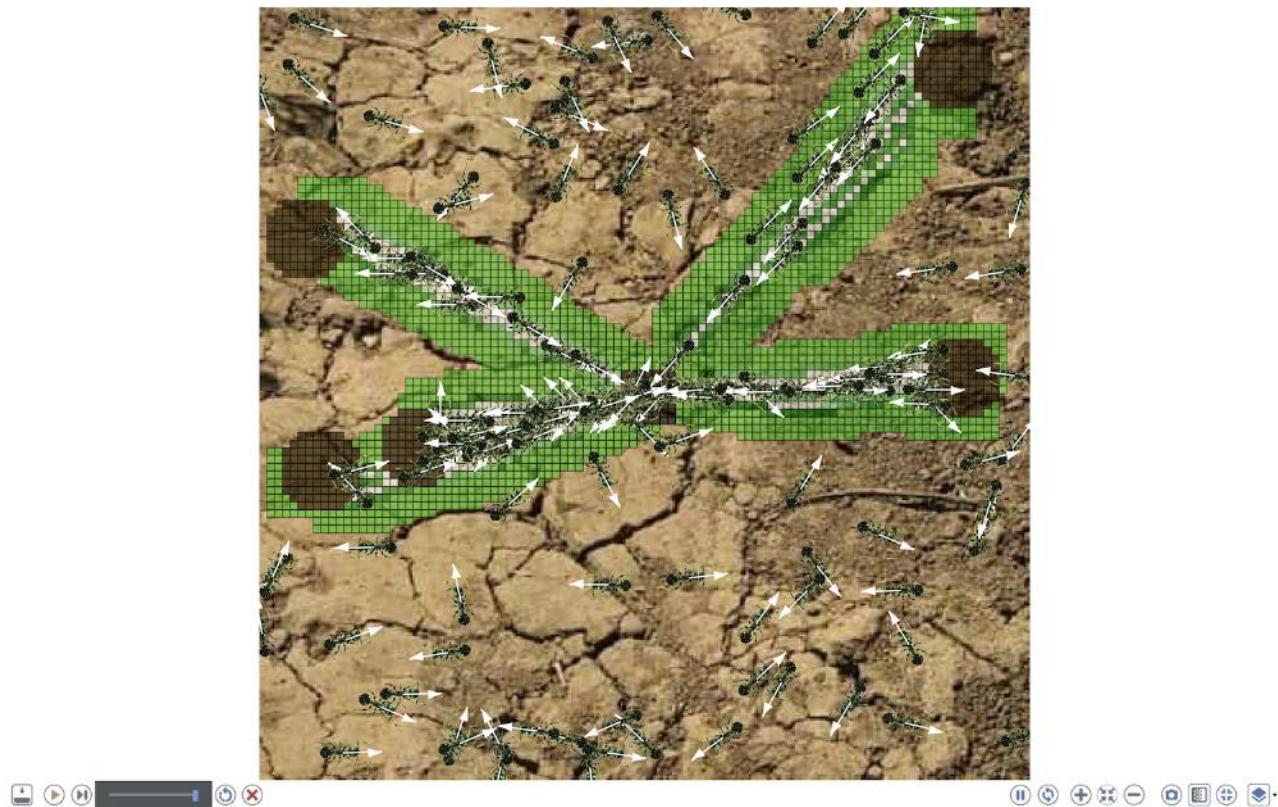


In addition to these commands, the contextual menu on the display provides three more commands (in "Presentation"):

- **Toggle overlay:** display/hide a semi-transparent toolbar on the bottom of the display, showing the coordinates of the mouse, the zoom, the number of fps (frame per second) of the simulation, and a scale (taking into account the zoom level).
- **Toggle toolbar:** display/hide the toolbar on the top of the display.

- **Background:** Change the background color.

When the View is displayed in **fullscreen mode**, the toolbar is now located in the bottom of the View and contain in addition to the previously detailed toolbar, the toggle side-control, and overlay controls and [controls of the experiment \(run, pause, step...\)](#).



OpenGL displays

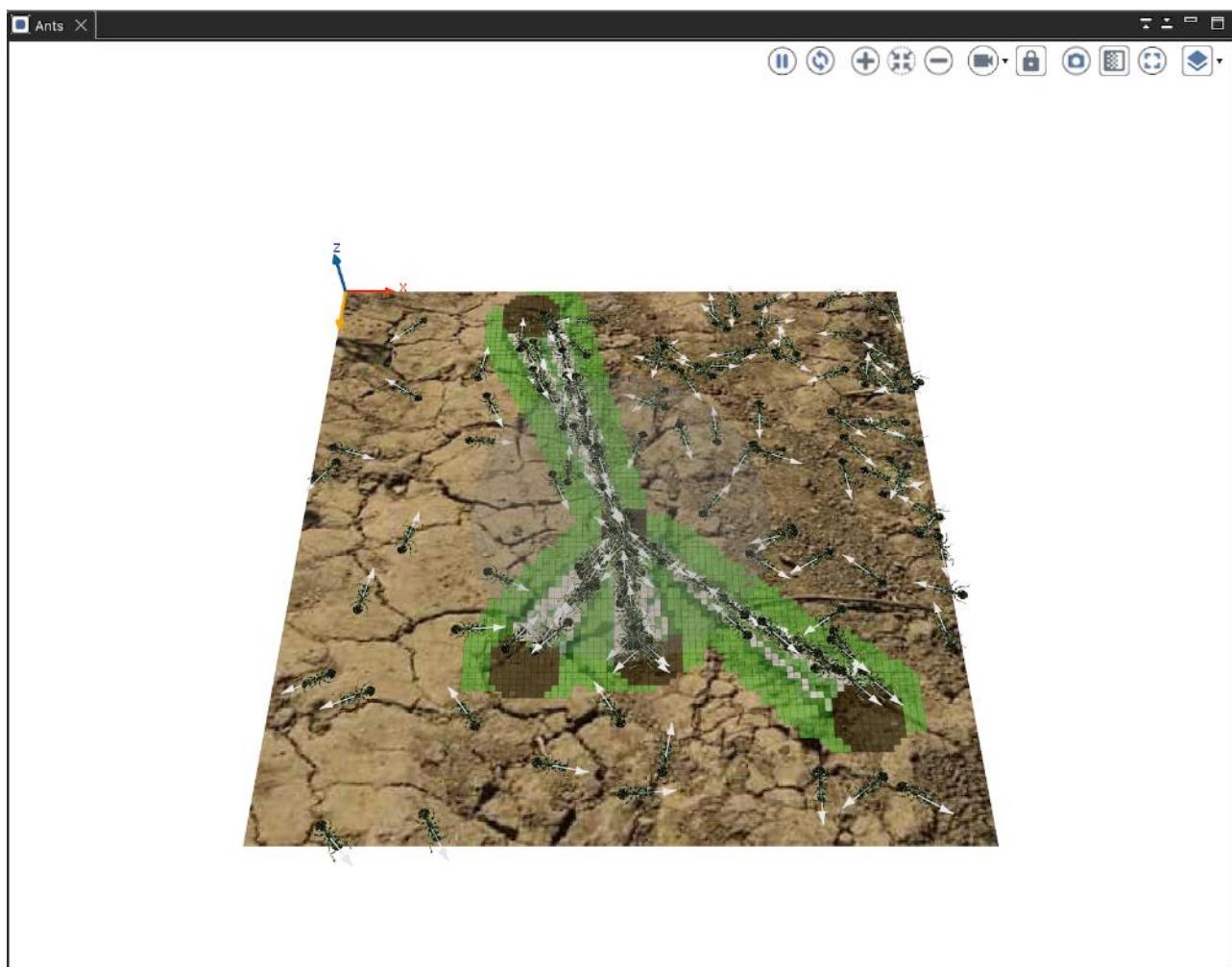
The OpenGL displays displaying offers all the feature provided by java2D but a 3D environnement:

- same behaviors with left-click, right-click and wheel than in the Java2D displays.
- `command` pressed (on Mac OS) or `Ctrl` (on Windows and Linux) + `Left-Click` pressed + mouse move: it controls the camera and modify its location/target/

orientation.

It opens many ways to visualize and understand your simulation(s) with most of the classical features provided by a 3D environment. More details and illustrations of those features can be found [here](#)

Any OpenGL display has the same menu and buttons as the classical Java2D displays. Nevertheless, the sidebar provides more options to manage camera and other options related to OpenGL displays management.



Camera commands

Key	Function
Double Click	Zoom Fit
+	Zoom In
-	Zoom Out
Up	Vertical movement to the top
Down	Vertical movement to the bottom
Left	Horizontal movement to the left
Right	Horizontal movement to the right
CTRL or CMD + Up	Rotate the model up (decrease the phi angle of the spherical coordinates)
CTRL or CMD + Down	Rotate the model down (increase the phi angle of the spherical coordinates)
CTRL or CMD + Left	Rotate the model left (increase the theta angle of the spherical coordinates)
CTRL or CMD + Right	Rotate the model right (decrease the theta angle of the spherical coordinates)

Key	Function
SPACE	Reset the pivot to the center of the envelope
KEYPAD 2,4,6,8	Quick rotation (increase/decrease phi/theta by 30°)
CTRL or CMD + LEFT_MOUSE	Makes the camera rotate around the model
ALT+LEFT_MOUSE	Begins Agent Selection using an ROI (Region of Interest)
SHIFT+LEFT_MOUSE	Draws an ROI on the display, allowing to maintain it across frames
SCROLL	Zoom-in/out to the current target (center of the sphere)
WHEEL CLICK	Reset the pivot to the center of the envelope

- **Keystone:** the keystone allows to modify the location of the 4 corner points of the environment bounding box. It can be used to project a simulation on a physical model as the projector can introduce some image distortions. Press **k** to enter in keystone mode, once the keystone is done press **k** to copy the value in the clipboard. You can now paste your keystone value as a facet in the display.

Version: 1.9.1

Batch Specific UI

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

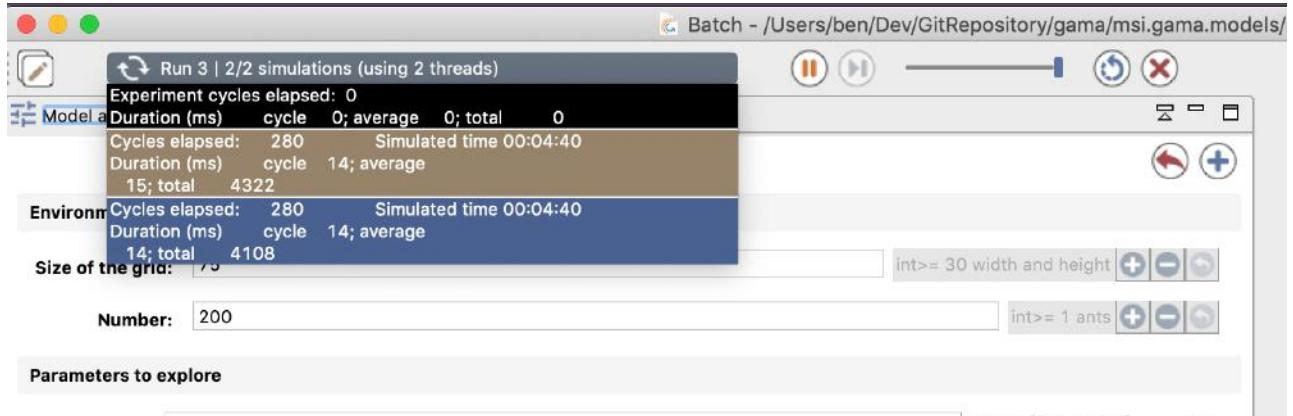
Table of contents

- [Batch Specific UI](#)
 - [Information bar](#)
 - [Batch UI](#)

Information bar

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

- The **run** number: One run corresponds to N executions of simulation with one given parameters values (N is an integer given by the facet `repeat` in the definition of a batch [experiment](#)). The number of runs is chosen by the [exploration method](#)).
- The **simulation** number: the number of replications done (and the number of replications specified with the `repeat` facet);
- The number of **thread**: the number of threads used for the simulation.



Batch UI

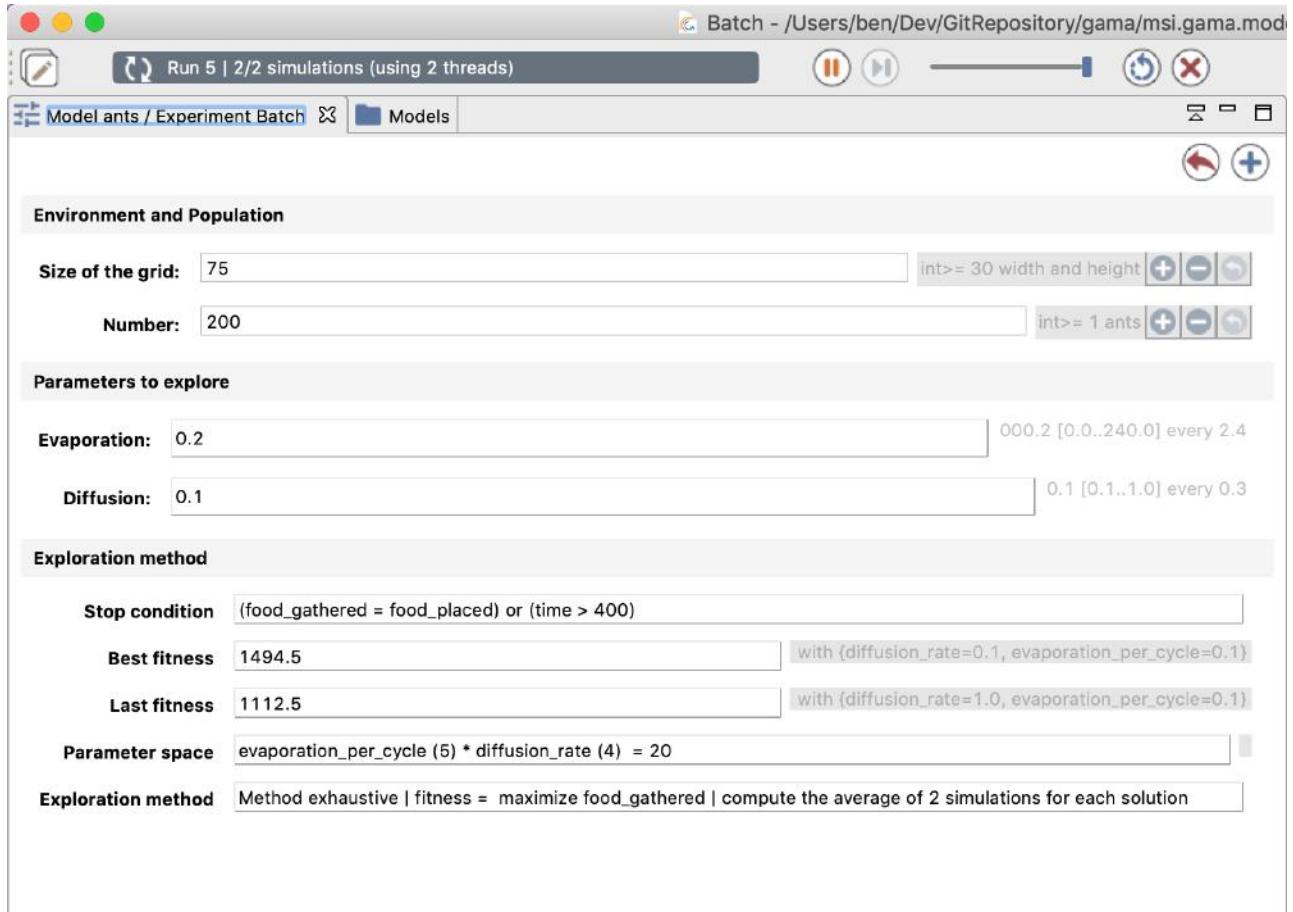
The parameters view is also a bit different in the case of a Batch UI:

- it shows both the parameters of the experiment, with a distinction between the ones that will be explored and the ones that will not.
- it also shows the state of the exploration. The provided information will depend on [the exploration method](#).

The following interface is generated given the following `experiment` (the exploration method is here the `exhaustive one`):

```
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_per_cycle 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 summarises all model parameters and the parameters given to the exploration method:

- **Environment and Population:** displays all the model parameters that should not be explored. Those parameters must be initialized with a fixed value when they are defined in the `experiment`.
- **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 displays information about the exploration method and the stop condition. It displays the size of the parameter space in the case of the exhaustive method, and different parameters (e.g. mutation or crossover probability...) for other methods. Finally, the best and the last fitnesses found are

shown, along with the associated parameter sets.

The following interface corresponds to the same experiment as previously, but with the [genetic](#) exploration method.

```
experiment Batch type: batch repeat: 2 keep_seed: true until:  
(food_gathered = food_placed) or (time > 400) {  
    // [Parameters]  
    method genetic maximize: food_gathered;  
}
```

The screenshot shows the AnyLogic software interface with the 'Experiment Batch' tab selected. The window is divided into several sections:

- Environment and Population:** Includes fields for 'Size of the grid' (75) and 'Number' (200).
- Parameters to explore:** Includes fields for 'Evaporation' (0.2) and 'Diffusion' (1.0).
- Exploration method:** Contains settings for the genetic algorithm:
 - Stop condition:** (food_gathered = food_placed) or (time > 400)
 - Best fitness:** 1225.5 (with {diffusion_rate=1.0, evaporation_per_cycle=1.0})
 - Last fitness:** 1225.5 (with {diffusion_rate=1.0, evaporation_per_cycle=1.0})
 - Parameter space:** evaporation_per_cycle (5) * diffusion_rate (4) = 20
 - Exploration method:** Method genetic | fitness = maximize food_gathered - the average of 2 simulations for each solution
 - Mutation probability:** 0.1
 - Crossover probability:** 0.7
 - Population dimension:** 3
 - Preliminary number of generations:** 1.0
 - Max. number of generations:** 20.0

Version: 1.9.1

Errors View

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

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

When we unfold the error, to have an idea of its location in the code. In addition clicking on one of the lines should highlight the corresponding line in the code.

The screenshot shows the NetLogo interface with the title bar "Model ants / Experiment Classic". The "Models" tab is selected. In the top right corner, there is an "Errors" button with a red exclamation mark icon. A modal window is open, displaying the error message: "1 occurrence in ant_grid6356 at cycle 0: Division by zero". Below the message, a stack trace is shown:

```
0  in int i <- int(3 / 0):
1  in int i <- int(3 / 0):
2  in ask food_places {
3  in ask food_places {
4  in loop times: number_of_food_places {
5  in init {
6  in agents ant grid6356. Simulation 0. Classic0
```

One of the most current (and sometimes the most mysterious) error is linked to an empty agent (with the value `nil`) on which we want to access to one of its attributes. It is expressed by `Cannot evaluate ATTRIBUTE_NAME as the target agent is nil` or `Java nil`. In this case, modelers have to check carefully their codes to be sure that all the agent variables have a not nil value.

The screenshot shows the NetLogo interface with the title bar "Model ants / Experiment Classic". The "Models" tab is selected. In the top right corner, there is an "Errors" button with a red exclamation mark icon. A modal window is open, displaying the error message: "1 occurrence in ant_grid5275 at cycle 0: Cannot evaluate name as the target agent is nil". Below the message, a stack trace is shown:

```
0  in write nil ant.name :
1  in write nil ant.name :
2  in agent ant grid5275
```

Version: 1.9.1

Running Headless

What is GAMA Headless

The headless mode gives the possibility 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. 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.

Different headless modes

1. The first and oldest way, called **Legacy mode** and detailed [here](#), consists in explicitly writing your full experiment plan (i.e each simulation you want to run, with each parameter sets) in an XML file. This way of using the Headless was the first implementation of the headless inside GAMA.
2. The second way, called **Headless Batch** and detailed on this [page](#), allows launching a [GAML batch experiment](#) in headless mode (i.e. without having to open GAMA's interface). This way is the most natural way to use the headless as it works exactly like in GUI Batch mode.
3. The last way, called **Headless Server** and described [there](#), let you open an interactive GAMA headless server on which you can dynamically send experiments to run. This last mode is interesting for using GAMA as back-end of

other project like web projects.

General knowledge about using GAMA Headless

There are two ways to run a GAMA experiment in headless mode: using a dedicated bash wrapper (recommended) or directly from the command line.

Bash Wrapper (recommended)

The wrapper file can be found in the `headless` directory located inside [Gama's installed folder](#). It is named `gama-headless.sh` on macOS and Linux, or `gama-headless.bat` on Windows.

You can start using it like so :

```
./gama-headless.sh [m/c/hpc/v] [launchingMode]
```

with:

- general headless options [-m/c/hpc/v]:
 - `-m memory` : memory allocated to gama (e.g. `-m 8g` to set it at 8GiB)
 - `-c` : console mode, the simulation description could be written with the `stdin`
 - `-hpc nb_of_cores` : limit to a specific number of cores the number of simulation running in parallel (eg. `-hpc 3` to limit GAMA at using 3 cores/ running 3 simulation at a time)
 - `-v` : verbose mode. trace are displayed in the console
- *launchingMode* will depend on which headless mode you'll use and explained in

following pages

You also can display general help on every options with this command:

```
./gama-headless.sh -help
```

Which, for release 1.9.0, will output:

```
*****
* GAMA version 1.9.0 *
* http://gama-platform.org *
* (c) 2007-2022 UMI 209 UMMISCO IRD/SU & Partners *
*****  
Welcome to Gama-platform.org version GAMA 1.9.0  
  
sh ./gama-headless.sh [Options]  
  
List of available options:  
  === Headless Options ===  
    -m [mem]                      -- allocate memory (ex 2048m)  
    -c                            -- start the console to write  
  xml parameter file  
    -v                            -- verbose mode  
    -hpc [core]                   -- set the number of core  
  available for experimentation  
    -socket [socketPort]          -- start socket pipeline to  
  interact with another framework  
    -p                            -- start pipeline to interact  
  with another framework  
  === Infos ===  
    -help                         -- get the help of the command  
  line  
    -version                      -- get the the version of gama  
  === Library Runner ===  
    -validate                     -- invokes GAMA to validate  
  models present in built-in library and plugins
```

Java Command (hard)

As GAMA is developed in Java, you can start the Headless mode by load appropriate bundle and starting it like this:

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

with:

- `$GAMA_CLASSPATH`: contains the relative or absolute path of jars inside the GAMA plugin directory and jars created by users
- *options* as explained above and in following pages

Note that we recommend you to open bash wrapper to have more detailed about how we imagine starting GAMA in headless mode.

Version: 1.9.1

Headless Batch

Getting started

This headless mode is the *Batch* one.

The advantage of this mode is how easily it is to prepare and launch, contrarily to the [Headless Legacy](#), this mode does not need any other file than the GAML file holding the experiment of type **batch**.

You can run your gama experiment with a command similar to this:

```
./gama-headless.sh [option] -batch experimentName /path/to/file.gaml
```

- with:
 - `-batch`: the flag that indicates it is a batch exploration
 - `experimentName`: the name of your batch experiment in the following file
 - `/path/to/file.gaml`: the path (relative or absolute) to the batch experiment

Simulation Output

Unfortunately, this mode can't save output data automatically, the actual way to do is saving wanted data inside CSV files from your model.

Calling GAMA headless on Windows

The example below assumes that your GAMA application is in folder `D:\software\` and your project (model) file is in folder `D:\my_models\`

Windows PowerShell

- You can open Windows PowerShell, change your directory to the headless folder and run gama-headless command:

```
cd D:\software\GAMA_1.9.0_Windows_with_JDK\headless\  
.gama-headless.bat -batch Optimization  
D:\my_models\predatorPrey\predatorPrey.gaml
```

Command Prompt

- You can open Command Prompt, change your directory to the headless folder and run gama-headless command:

```
cd D:\software\GAMA_1.9.0_Windows_with_JDK\headless\  
gama-headless.bat -batch Optimization  
D:\my_models\predatorPrey\predatorPrey.gaml
```

Python Script

- Your python script will have the following lines of code, mainly using the `os` package to run the native system commands

```
import os
```


Version: 1.9.1

Headless Server

Running a Gama Headless server:

Before doing anything, make sure that you possess the rights to create files and directories at the location you are running gama-server because it will need it to create workspaces on the fly.

From the release

Go to the `headless` directory in your Gama installation folder and run the script `gama-headless.sh` (or `gama-headless.bat`) with the argument `-socket` followed by the port number you want your Gama server to run on.

For example on Mac OS you could do:

```
cd Gama.app/Contents/headless
```

to move to the right directory, then run the script to listen on port `6868` with:

```
gama-headless.sh -socket 6868
```

From the command-line tool

The users who installed gama through a `.deb` file or `aur` have access to the command `gama-headless` and thus only need to open a terminal and run

```
gama-headless -socket 6868
```

to run a Gama server on the port `6868`.

From the source code

In Eclipse, instantiate a headless server by running `msi.gama.headless.id4_full` with the following argument `-os ${target.os} -ws ${target.ws} -arch ${target.arch} -nl ${target.nl} -socket 6868` (you can specify any other port)

From docker

First ensure to pull the official docker image

```
docker pull gamaplatform/gama:<version>
```

Then, run the container with the image you just pulled.

Do not forget to (1) expose the port you're starting your server on, and (2) mount your workspace inside the started container as below :

```
docker run -v <path/to/your/workspace>:/working_dir -p 6868:6868 gamaplatform/gama:<version> -socket 6868
```

For more informations, please refer to [Docker's official documentation](#) or [GAMA image's repository](#).

Connection

To connect to gama-server as a client you just need to access the following address: `ws://<ip>:<port>`. For example if you try to connect to a gama-server running on your current computer and started with the command `gama-headless -socket 6868`, you will have to connect to `ws://localhost:6868`.

Once a client is connected, gama-server will send a json object of type `ConnectionSuccessful`.

General description of interactions

Once connected, you can ask gama-server to execute different commands to control the execution of different simulations.

If you close your client application (or just close the socket on client-side) **gama-server will destroy all running simulations** of that client, so you have to keep your client alive.

For every command treated by gama-server, it will send back a json object describing if the command has been executed correctly or if there was a problem. If an unexpected exception is raised in gama-server, it will try to send the connected clients a json-object describing it. The same goes if a simulation throws an exception/error while running, the client that asked for it to run will receive it as a json-object.

In addition, the client can ask gama-server to receive (or not) the different outputs of a simulation: `write` statements, dialogs, status-bar changes etc. they will be sent as they come, in a specific json wrapper.

Available commands

All the commands sent to gama-server must be formatted as a `json` object.

The available commands are:

The `exit` command

This command is used to kill gama-server. It is triggered by sending a json object formatted as follows to the server

```
{
  "type": "exit"
}
```

Answer from gama-server

It is the only command that won't send back a json object.

The `load` command

This command is used to ask the server to initialize a specific experiment in a gaml file on the server's file-system. It is triggered by sending a json object formatted as follows to the server:

```
{  
    "type": "load",  
    "model": "<gaml_file_path>",  
    "experiment": "<experiment_name>",  
    "console": "<console>", //optional  
    "status": "<status>", //optional  
    "dialog": "<dialog>", //optional  
    "runtime": "<runtime>", //optional  
    "parameters": "<params>", //optional  
    "until": "<end_condition>", //optional  
}
```

The `model` parameter indicates the path of the experiment file on the server's file-system, and `experiment` is the actual name of the experiment to run. The four parameters `console`, `status`, `dialog` and `runtime` are booleans used to determine if the messages from respectively the console, the status-bar, the dialogs and the runtime errors should be redirected to the client. They are optional as per default `console` and `runtime` are set to `true` and the two others to `false`. You can add an array of parameters that will be used to initialize the experiment's variables with the values you picked. The value of `parameters` should be formatted as follows:

```
[  
    {  
        "type": "<type of the first parameter>",  
        "value": "<value of the first parameter>",  
        "name": "<name of the first parameter in the gaml file>"  
    },  
    {  
        "type": "<type of the second parameter>",  
        "value": "<value of the second parameter>",  
        "name": "<name of the second parameter in the gaml file>"  
    },  
    ...  
]
```

You can also add an ending condition to your simulation with the parameter `until`, the condition must be expressed in gaml.

Answer from gama-server

The `content` field of the response json sent by gama-server after processing this command will directly contain the `experiment_id` value stored as a string. The experiment id should be used in all the other commands to refer to that specific experiment in order to control it.

The play command

This command is used to actually run an experiment already initialized. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "play",  
  "exp_id": "<experiment_id>",  
  "sync": "<synchronized>", //optional  
}
```

The `experiment_id` is used to identify the experiment to play, and the optional `sync` is a boolean used in the case where there was an end condition defined in the `load` command, if it is true, gama-server will not send a response to the command, but only a end of simulation message once the condition is reached, if it's false gama-server will send both the response to the command and the `SimulationEnded` message.

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it. In case where the end condition is reached, a message of type `SimulationEnded` is sent to the client with an empty `content`.

The pause command

This command is used to pause a running experiment. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "pause",  
  "exp_id": "<experiment_id>"  
}
```

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it.

The step command

This command is used to process one (or a defined number of) step(s) of a simulation that has already been loaded. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "step",  
  "exp_id": "<experiment_id>",  
  "nb_step": "<number_of_steps>", //optional  
  "sync": "<synchronized>", // optional  
}
```

As usual `exp_id` refers to the experiment you want to apply the command to. The `nb_step` parameter indicates how many steps you want to execute, if you do not give that parameter gama-server will execute one step. The `sync` parameter indicates whether gama-server must wait for the end of the step(s) to send back a success message (when its value is true), or just plan the step(s) and send one directly after (when its value is false), this parameter can be ignored and will be interpreted as if it were `false`.

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it.

The `stepBack` command

This command is used to rollback the simulation one (or a defined number of) step(s) back. This command only works on experiments of type `memorize`. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "stepBack",  
  "exp_id": "<experiment_id>",  
  "nb_step": "<number_of_steps>", //optional  
  "sync": "<synchronized>", // optional  
}
```

The parameters are exactly the same as in the `step` command.

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it.

The `stop` command

This command is used to stop (kill) a running experiment. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "stop",  
  "exp_id": "<experiment_id>",  
}
```

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it.

The `reload` command

This command is used to reload an experiment. The experiment will be stop and the initialization process run again. You can use this command to change the simulation parameters or the ending condition. It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "reload",  
  "exp_id": "<experiment_id>",  
  "parameters": "<params>", //optional  
  "until": "<end_condition>", //optional  
}
```

Just like for the `load` command, the `parameters` and the `until` parameters are optional and must follow the same formatting.

Answer from gama-server

This command has an empty `content` field in the response json sent by gama-server after processing it.

The `expression` command

This command is used to ask the server to evaluate a `gaml` expression having an experiment as context.
It is triggered by sending a json object formatted as follows to the server

```
{  
  "type": "expression",  
  "exp_id": "<experiment_id>",  
  "expr": "<expression to evaluate>"  
}
```

For example if you want to know the number of agents of species `people` currently present in the simulation represented by the id `123`, you could send this command to gama-server:

```
{  
  "type": "expression",  
  "exp_id": "123",  
  "expr": "length(people)"  
}
```

Answer from gama-server

If the command is executed successfully by gama-server the `content` field of the response json will directly contain the result of the evaluated expression as a string.

Gama-server messages

All messages send by gama-server follow a json architecture that is formatted as follows:

```
{  
  "type": "some string describing the type of message",  
  "content": "a field containing everything additional information for the message", //It can be a  
  string, an int or a json object  
  "exp_id": "contains the experiment id (as a string) to which this message is linked to", //Optional,
```

Messages types

All messages have in common a `type` field that informs the client of the type of message sent. The different types possibles are:

- `ConnectionSuccessful`: Used when a client connected without any problem to gama-server
- `SimulationStatus`: Signals a message representing a simulation status
- `SimulationStatusInform`: Signals a message representing a simulation inform status
- `SimulationStatusError`: Signals a message representing a simulation error status
- `SimulationStatusNeutral`: Signals a message representing a simulation neutral status
- `SimulationOutput`: Signals a message as would be written in the console by a `write` statement in gama with an interface
- `SimulationDebug`: Signals a message as would be written in the console by a `debug` statement in gama with an interface
- `SimulationDialog`: Signals a message representing what would be a dialog in gama with an interface
- `SimulationErrorDialog`: Signals a message representing what would be an dialog in gama with an interface
- `SimulationError`: Signals a message representing an error raised in a running simulation
- `RuntimeError`: Signals a message representing an exception raised in gama-server while trying to process a command
- `GamaServerError`: Signals a message representing an unknown exception raised in gama-server (can be unrelated to any command)
- `MalformedRequest`: Signals that a command sent by the client doesn't follow the expected format (lack of parameter, wrong type etc.)
- `CommandExecutedSuccessfully`: Signals that a command sent by the client was executed without any problem on gama-server
- `SimulationEnded`: Signals that a running simulation **reached its end condition** and stopped. Beware if the simulation stops for another reason, this message won't be send.
- `UnableToExecuteRequest`: Signals that a command cannot be executed, though it may be formatted correctly. It mainly occurs when trying to execute a command on a simulation that is not currently running.

Connection related answers

When your client is connected correctly to gama-server, a message is sent. Its `type` is `ConnectionSuccessful` with an empty content. In case of problem, the client may receive a message of `type GamaServerError` or just get a timeout/broken connection message at the socket level.

Command answers

For every command described in the [commands section](#), the client will received a json answer formatted as follows:

```
{  
  "type": "some string describing the type of message",  
  "content": "a field containing every additional information for the message", //It can be a string, an  
  int or a json object, depending on the type of message, it could also be empty
```

So for example if you send an expression command to gama, with an `experiment_id` of value **2** and you want to evaluate the expression `length(people)` to know the number of agent **people** in that simulation. You may receive an answer looking like this:

```
{  
  "type": "CommandExecutedSuccessfully", //The type indicates that everything went normally  
  "content": 102, //There are 102 agents of the species people in your simulation at the time of  
  //evaluation  
  "command": //The description of the command you sent, as interpreted by gama-server and turned into a  
  //json  
  {  
    "type": "expression",  
    "exp_id": "2",  
    "expr": "length(people)",  
  }  
}
```

The `command` field is very useful for clients that run multiple simulations and commands at the same time, as it can be used to retrace which command the message responds to. **Note:** The `command` field contains all the parameters of the command sent by the client, including those that are not useful for GAMA to execute the command, you can thus use it to store more data, like an internal id used by the client, some kind of counter etc..

In case there is an error resulting from the processing of your command, you may receive an error message of type:

- **MalformedRequest** if you forgot a mandatory parameter to execute the command or gave objects that couldn't be de-serialized. The list of required parameters will be sent as a string in the `content` field.
- **UnableToExecuteRequest** if you are trying to execute a command on a simulation that is not currently running or some other problem of "logic". You will find more informations in the `content` field.
- **RuntimeError** and **GamaServerError** if while executing your command, an exception happens, either in gaml code for **RuntimeError** or in gama's code for **GamaServerError**. The exception's description will be given in the `content` field, as a json object containing the error message and the stack trace.

There is no `exp_id` field in those messages, because it is already included in all the `command` fields that are related to an experiment.

errors and exceptions

In addition to the error messages you can receive when directly requesting to execute a command (`MalformedRequest`, `UnableToExecuteRequest`) described in the `command answers` section, or the network errors that can be raised for external problems, it is possible that gama-server encounters an exception while running. In that case gama-server will send a json message formatted as described in the `Gama-server messages` section, the two different types would either be `GamaServerError`, `RuntimeError` or `SimulationError` and the `content` field would be filled as follows:

```
{  
  "exception": "The java class of the exception",  
  "message": "The message describing the problem",  
  "stack": [],//The stack trace of the exception given as a list of strings  
}
```

the `exp_id` and `command` fields would be present if possible, depending on where the exception happens.

Simulations outputs

As mentioned in the [introduction](#) and the description of the [load command](#). You can ask gama-server to redirect the simulation's outputs. There are 3 different types of output produced by a simulation that you can chose to redirect or not:

- the messages in the dialogs
- the messages in the status-bar
- the messages in the gama console Each has an associated boolean that you have to set to `true` in the [load command](#) in order to have it redirected to the client.

The output messages are sent directly to the client as soon as they are asked by the simulation. The format of the output messages follows the usual [message format](#). The `exp_id` will always be filled with the current experiment id, the 'command' field won't be present. The different types of messages possible are:

- for dialog messages: `SimulationDialog` and `SimulationErrorDialog` respectively for normal dialogs and error dialogs
- for status messages: `SimulationStatusNeutral`, `SimulationStatusError`, `SimulationStatusInform`, `SimulationStatus`
- for console messages: `SimulationOutput` for the messages written with the `write` statement and `SimulationDebug` for the ones written with the `debug` statement.

The `content` field will be formatted as follows:

- for dialog messages, it's directly a string containing the message
- for status messages:

```
{  
  "message": "the status message",  
  "icon": "the name/path of the associated icon", //only present for some SimulationStatus and  
  SimulationStatusInform messages  
  "color":  
  {  
    "r": "red value",  
    "g": "green value",  
    "b": "blue value",  
  }, // The background color in the status-bar, only present in some SimulationStatus messages  
}
```

- for console messages:

```
{  
  "message": "the message as it would be written in the console",  
  "color":  
  {  
    "r": red_value,  
    "g": green_value,  
    "b": blue value,  
  }, // The text color  
  "cycle": simulation_cycle, // the cycle of the simulation at the moment the debug statement is
```

Python wrapper

A python package is available to interact with Gama server as a client, you can find it [here](#). It will take care of formatting the queries to the server and receiving the answers. You simply have to install the package into your python environment with the command `pip install gama-client` and then import `gama_client` into your python files. For more information follow the `README.md` available on the package's github.

Javascript Client

There is also a javascript client being developed in this repository [gama.client](#)

Hello World Visualization in MapBox

- Clone the repository [gama.client](#)
- In `js/gama_client.js` edit the following variable `ABSOLUTE_PATH_TO_GAMA` to your local path (e.g `var ABSOLUTE_PATH_TO_GAMA = '/Users/arno/';`)
- open `index.html` in a browser

Hello World Message example

- In `js/simple_syntax.js` edit the following variable `modelPath` to your model's path
- open `syntax.html` in a browser

Troubleshooting

crash on `load` command

It is possible that gama-server starts and accepts connections, but crashes when receiving a `load` command with a message of the type:

```
java.lang.Exception: java.lang.NoClassDefFoundError: org/eclipse/core/resources/ResourcesPlugin
    at org.java_websocket.server.WebSocketServer$WebSocketWorker.run(WebSocketServer.java:1093)
Caused by: java.lang.NoClassDefFoundError: org/eclipse/core/resources/ResourcesPlugin
    at msi.gama.lang.gaml.indexer.GamlResourceIndexer.<clinit>(GamlResourceIndexer.java:54)
    at msi.gama.lang.gaml.resource.GamlResource.doLinking(GamlResource.java:362)
    at org.eclipse.xtext.resource.XtextResource.updateInternalState(XtextResource.java:304)
    at org.eclipse.xtext.resource.XtextResource.updateInternalState(XtextResource.java:292)
    at msi.gama.lang.gaml.resource.GamlResource.updateInternalState(GamlResource.java:308)
    at org.eclipse.xtext.resource.XtextResource.doLoad(XtextResource.java:182)
    at org.eclipse.xtext.linked.lazy.LazyLinkingResource.doLoad(LazyLinkingResource.java:115)
    at org.eclipse.emf.ecore.resource.impl.ResourceImpl.load(ResourceImpl.java:1563)
    at org.eclipse.emf.ecore.resource.impl.ResourceImpl.load(ResourceImpl.java:1342)
    at org.eclipse.emf.ecore.resource.impl.ResourceSetImpl.demandLoad(ResourceSetImpl.java:259)
    at org.eclipse.emf.ecore.resource.impl.ResourceSetImpl.demandLoadHelper(ResourceSetImpl.java:274)
    at org.eclipse.xtext.resource.XtextResourceSet.getResource(XtextResourceSet.java:266)
```

This issue arises because gama-server tries to create a workspace for your experiment but does not have the appropriate rights to do it. It can be the case in windows if you run gama-server directly from the `headless` directory from the installation folder (protected by default) and that you are not an Admin

Version: 1.9.1

Headless Legacy

Getting started

This headless mode is the *Legacy* one. So, if you are already familiar with headless from version 1.8.1 or older, nothing changed.

This mode relies on writing an explicit simulation plan in XML file. Those simulations, mostly for legacy reason, have to be of type `gui` (which is pretty counterintuitive, but this type let you set the parameter value to each simulation independently), but you don't have to write it explicitly as it's the default type for an experiment.

You can generate a first XML file corresponding to an existing experiment with the following command:

```
./gama-headless.sh -xml experimentName /path/to/inputFile.gaml /path/to/outputFile.xml
```

- with:
 - `-xml`: the flag asking the headless to generate a XML file well-formatted for our experiment
 - `experimentName`: the name of the `experiment` you want to run in headless
 - `/path/to/inputFile.gaml`: the path (relative or absolute) to your GAML file containing the experiment you want to run
 - `/path/to/outputFile.xml`: the path (relative or absolute) to the generated XML file

You can see more in details the content of the generated XML file (applied on the model *Predator Prey*) in [the Experiment Input File part](#).

Once you finished preparing your XML file, you can run it with a command similar to the following one:

```
./gama-headless.sh /path/to/file.xml /path/to/generated/outputFolder
```

- with:
 - `/path/to/file.xml`: the path (relative or absolute) to the XML file containing the full exploration plan to run by the headless
 - `/path/to/generated/outputFolder`: the path (relative or absolute) which will be generated by GAMA and hold every output files (variables, snapshots, and console messages)

You can see result output folder in [the Simulation Output part](#)

Experiment Input File

The XML input file contains for example (you can find it next to the file `gama-headless.sh` at the path `samples/predatorPrey.xml`):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Experiment_plan>
    <Simulation experiment="prey_predatorExp" finalStep="1000" id="0"
seed="1.0" sourcePath=".//samples/predatorPrey/predatorPrey.gaml">
        <Parameters>
            <Parameter name="Nb Preys: " type="INT" value="200"
var="nb_preys_init"/>
            <Parameter name="Prey max energy: " type="FLOAT"
value="1.0" var="prey_max_energy"/>
            <Parameter name="Prey max transfert: " type="FLOAT"
```

NB: Several simulations can be determined in one experiment plan. These simulations are run in parallel according to the number of allocated cores.

Simulation

```
<Simulation experiment="prey_predatorExp" finalStep="1000" id="0"  
seed="1.0" sourcePath=".samples/predatorPrey/predatorPrey.gaml">
```

- with:
 - `experiment` (**required**): determines which experiment should be run on the model. This experiment should exist, otherwise, the headless mode will exit.
 - `finalStep` (**required**): determines the number of simulation steps you want to run.
 - `id` (**required**): permits to prefix output files for experiment plans with huge simulations.
 - `seed` (optional): permits to set the `seed` value of the simulation.
 - `sourcePath` (**required**): contains the relative or absolute path to read the gaml model.
 - `until` (optional): defines a stop condition in GAML. It can be combined with the `finalStep` facet (in this case a simulation will finish when the stop condition is fulfilled or when the final step is reached).

Parameters

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

```
<Parameter name="Nb Preys: " type="INT" value="200"  
var="nb_preys_init"/>
```

- with:

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

NB: You need to set at least one of the attributes `name` or `var` in your `Parameter` tag

Outputs

One line per output value you want to retrieve. Outputs can be the name 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 framerate="1" id="1" name="Number of predators"/>
... with the name of a (built-in) variable defined in the
experiment itself...
<Output framerate="1" id="2" name="main_display"/>
```

- with:
 - `framerate` (**required**): the frequency of the monitoring (each step, every 2 steps, every 100 steps...).
 - `id` (optional): permits to prefix output files for simulation with huge outputs
 - `name` (**required**): name of the output in the 'output'/'permanent' section in the experiment or name of the experiment/model attribute to retrieve
 - `output_path` (optional): change the output directory where snapshot images are saved (for display output only!)

NB: the lower the framerate value is, the longer the experiment.

NB2: 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/
out
└── console-outputs-0.txt
└── simulation-outputs0.xml
└── snapshot
    ├── main_display0-0.png
    ├── main_display0-1.png
    ├── main_display0-2.png
    ├── main_display0-3.png
    ├── main_display0-4.png
    └── ...
```

- with:
 - `console-outputs-<simulationId>.xml`: containing every message written in GAMA's console
 - `simulation-outputs<simulationId>.xml`: containing variables' results in a XML format
 - `snapshot`: containing the snapshots (i.e. screenshots of gui displays) 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="0" >
    <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'>
        ...
    </Step>
</Simulation>

```

- With:
 - `<Simulation id="0" >`: tag containing results of the simulation. The `id` is set one set in the input file, in the **heading part**
 - `<Step id='0' >`: one block per step done containing the value of outputs variables. The `id` corresponds to the step number
 - `<Variable />` with:
 - `name`: name of the output
 - `value`: the current value of the model variable at the given step.

NB: The value of an output is repeated according to the framerate defined in the input experiment file.

NB2: The `value` of an output display gives the relative path to the generated image saved in '.png' format.

Calling GAMA headless legacy on Windows

The example below assumes that your GAMA application is in folder `D:\software\` and your project (model) file is in folder `D:\my_models\`. The data structure of the example model as in following. The example models can be found in the GAMA headless folder (`GAMA_1.9.0_Windows_with_JDK\headless\samples\predatorPrey`)

```
predatorPrey
├── includes
└── models
    ├── ...
    └── predatorPrey.gaml
```

In the `predatorPrey` model, we have a **GUI experiment** named `prey_predator`

Windows PowerShell

- You can open Windows PowerShell, change your directory to the headless folder and run `gama-headless` command:

```
cd D:\software\GAMA_1.9.0_Windows_with_JDK\headless\
.\gama-headless.bat -xml prey_predator
D:\my_models\predatorPrey\models\predatorPrey.gaml
D:\my_models\predatorPrey\models\predatorPrey.xml
.\gama-headless.bat D:\my_models\predatorPrey\models\predatorPrey.xml
D:\my_models\predatorPrey\results
```

Command Prompt

- You can open Command Prompt, change your directory to the headless folder `D:\software\GAMA_1.9.0_Windows_with_JDK\headless\` then run the commands:

```
gama-headless.bat -xml prey_predator  
D:\my_models\predatorPrey\models\predatorPrey.gaml  
D:\my_models\predatorPrey\models\predatorPrey.xml  
gama-headless.bat D:\my_models\predatorPrey\models\predatorPrey.xml  
D:\my_models\predatorPrey\results
```

Python Script

- Your python script will have the following lines of code, mainly using the `os` package to run the native system commands

```
import os  
os.chdir("D:\software\GAMA_1.9.0_Windows_with_JDK\headless")  
os.system("gama-headless.bat -xml prey_predator D:\my_models\  
predatorPrey\models\predatorPrey.gaml D:\my_models\predatorPrey\  
models\predatorPrey.xml")  
os.system("gama-headless.bat D:\my_models\predatorPrey\models\  
predatorPrey.xml D:\my_models\predatorPrey\results")
```

Version: 1.9.1

Preferences

Various preferences are accessible in GAMA to allow users and modelers to personalize their working environment, runtime options and simulation displays. This section reviews 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, as **default** behavior the preferences specific to GAMA will be shared, on the same machine, and for the same user, among all the workspaces managed by GAMA. If you want your preferences to be workspace specific you have to turn `use_global_preference_store` preferences to false using [tags](#)

Table of contents

- [Manage preferences](#)
 - [Preferences panel](#)
 - [Within model](#)
 - [Tags](#)
- [Preferences](#)
 - [Interface](#)
 - [Editors](#)
 - [Execution](#)
 - [Displays](#)
 - [Data and Operators](#)
 - [Manage preferences in GAML](#)

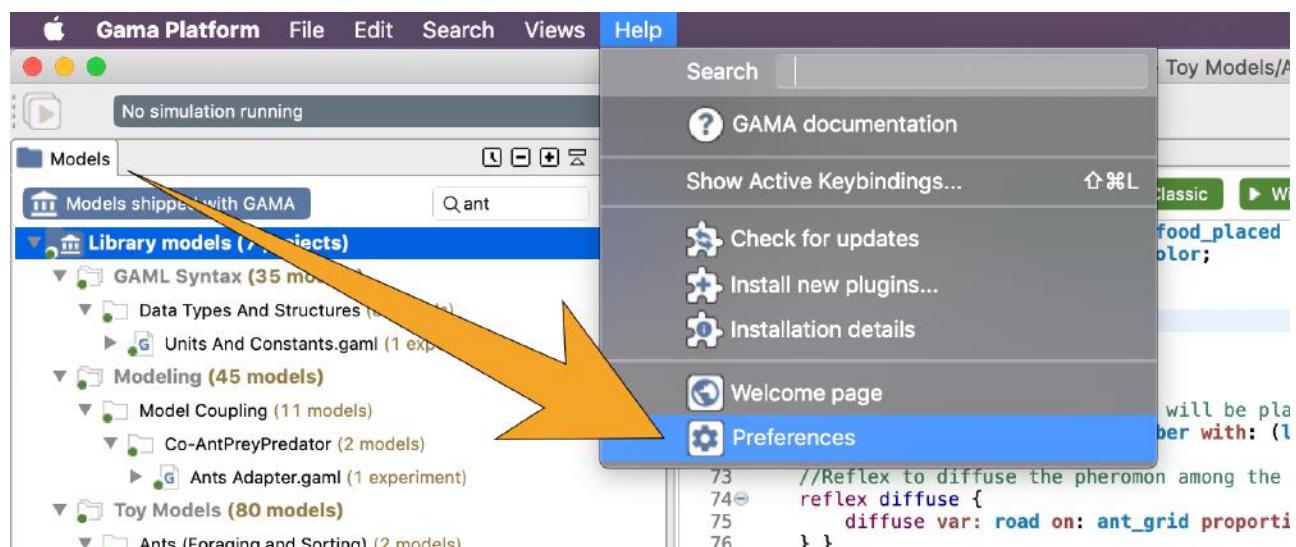
- Advanced Preferences

Manage preferences

There are three different ways to specify the preferences you want Gama to use. The first, and most intuitive way, is to open preference panel in the user interface of Gama. The second is to specify preferences within your model (global) using gaml syntax, like in this [example model code](#). Last and least intuitive for modelers, is to use tags when launching Gama either from git or command line. The three methods are exposed below.

Preferences panel

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.



Within model

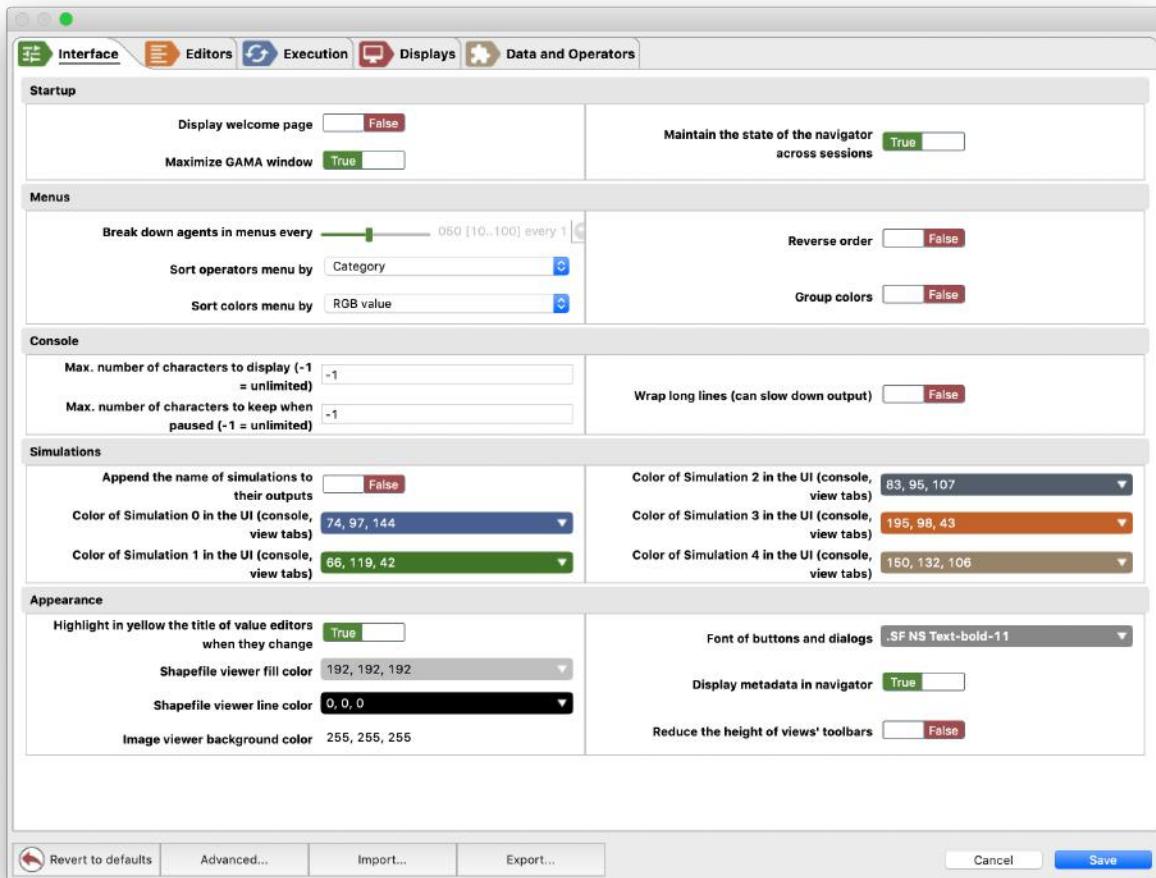
To modify one or several preferences for a specific execution of a model, one can use `gaml` references to the corresponding preferences and directly assign the desired value. For instance, if you want your batch experiment to use all core when simulation are running in parallel, you can add this line to the `init` block of your model `gama.pref_parallel_simulations_all <- true;`. The same can be done for every preferences. A non exhaustive list, with code example can be found [here](#); while having the mouse pointer over preferences in the preference panel, display the associated name.

Tags

The last way to customize preferences attached to Gama is to use `tags`. Those are passed to the Gama execution (when Gama is launched) using the `eclipse` syntax: `-Dname_of_the_preference=true/false` where `name_of_the_preference` is the name of the variable associated with the preference, and `true/false` one of the possible value for the preference (which is not limited to boolean but can be number, specific string or any value type). There are two main ways to input those tags: with Gama git using `eclipse` Run configuration, in tab called Arguments and when launching Gama from the `java` command line (e.g. see the [headless-game.sh](#) to have an example).

Interface

The Interface pane gathers all the preferences related to the appearance and behavior of the elements of the Graphical User Interface of GAMA.



• Startup

- **Display welcome page:** if true, and if no editors are opened, the [welcome page](#) is displayed when opening GAMA.
- **Maximize GAMA window:** if true, the GAMA window is open with the maximal dimensions at startup.
- **Maintain the state of the navigator across sessions:** if true, the context of the navigator (project opened, file selected...) will be saved when GAMA is closed and reloaded next start.

• Menus

- **Break down agents in menu every:** when [inspecting](#) a large number of agents, this preference sets how many should be displayed before the

decision is made to separate the population in sub-menus.

- **Sort operators menu by:** among [category, name], this preference sets how the operators should be displayed in the menu "Model" > "Operators" ([available only in Modeling perspective](#), when a model editor is active).
 - **Sort colors menu by:** among [RGB value, Name, Brightness, Luminescence], this sets how are sorted the colors in the menu "Model" > "Colors" ([available only in Modeling perspective](#), when a model editor is active).
 - **Reverse order:** if true, reverse the sort order of colors sets above.
 - **Group colors:** if true, the colors in the previous menu are displays in several sub-menus.
- **Console**
 - **Max. number of characters to display in the console (-1 means no limit)**
 - **Max. number of characters to keep when paused (-1 means no limit)**
 - **Wrap long lines (can slow down output)**
 - **Simulations**
 - **Append the name of simulations to their outputs:** if true, the name of the simulation is added after the name of the display or monitor (interesting in case of multi-simulations).
 - **Color of Simulation X in the UI (console, view tabs):** each simulation has a specific color. This is particularly interesting in case of a multi-simulations experiment to identify the displays of each simulation and its console messages.
 - **Appearance**
 - **Highlight in yellow the title of value editors when they change**
 - **Shapefile viewer fill color**
 - **Shapefile viewer line color**
 - **Image viewer background color:** Background color for the image viewer (when you select an image from the model explorer for example)
 - **Font of buttons and dialogs**

- **Display metadata in navigator:** if true, GAMA provides some metadata (orange, in parenthesis) after the name of files in the navigator: for a GAML model, it is the number of experiments; for data files, it depends on the kind of data: (for shapefiles) number of objects, CRS and dimensions of the bounding box, (for csv) the dimensions of the table, the delimiter, the data type ...

Editors

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



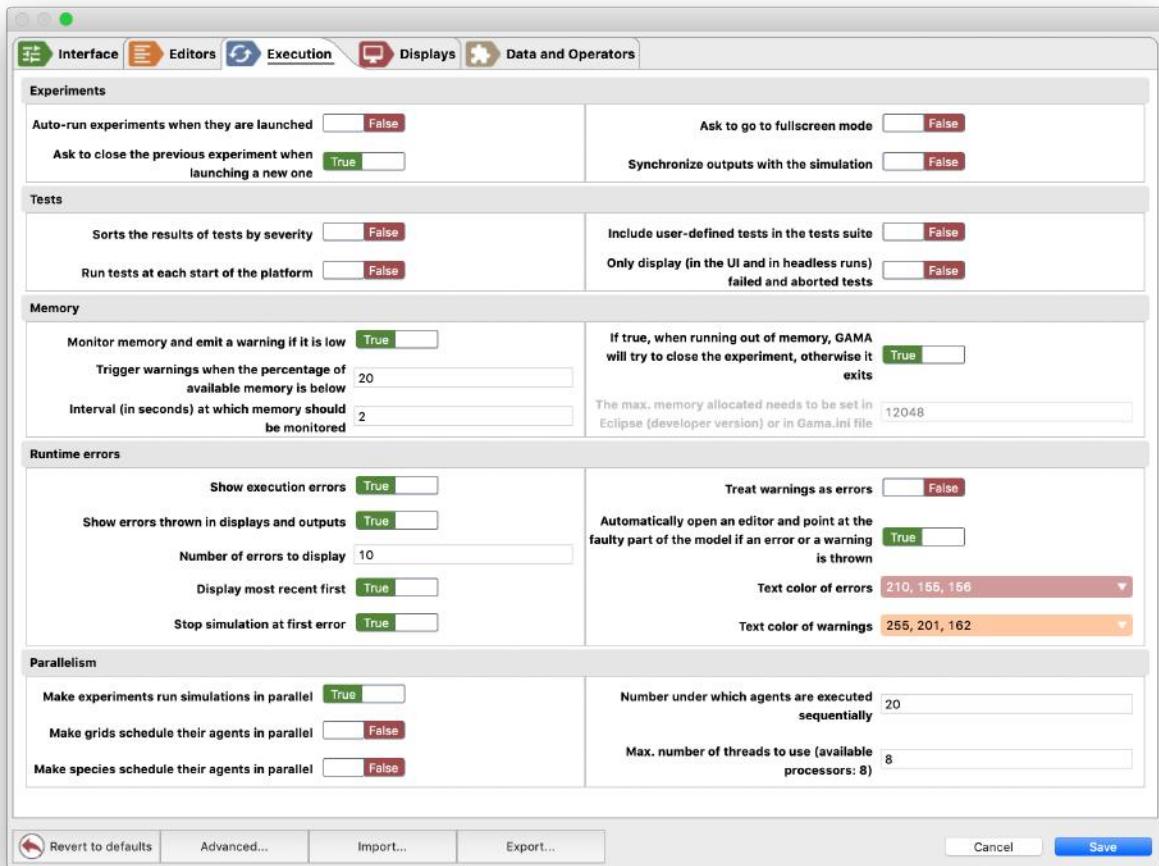
• Options

- **Show warning markers in the editor**: if false, the warning will only be available from the Validation View.
- **Show information markers in the editor**: if false, the information will only be available from the Validation View.
- **Save all editors when switching perspectives**
- **Hide editors when switching to simulation perspectives (can be overridden in the 'layout' statement)**

- **Applying formatting on save:** if true, every time a model file is saved, its code is formatted.
- **Save all model files before launching an experiment**
- **Drag files and resources as references in GAML files:** a GAML model file is dropped in another file as an import and other resources as the definition of a variable accessing to this resource.
- **Ask before saving each file**
- **Edition**
 - **Close curly brackets ({}):**
 - **Close square brackets ([]):**
 - **Close parentheses:**
 - **Turn on colorization of code sections:** if true, it activates the colorization of code blocks in order to improve the visual understanding of the code structure.
 - **Font of editors**
 - **Background color of editors**
 - **Mark occurrences of symbols:** if true, when a symbol is selected, all its other occurrences are also highlighted.
- **Syntax coloring:** this section allows the modeler to set the font and color of each GAML keyword kind in the syntax coloring (in any GAMA editor).

Execution

This pane gathers all the preferences related to the execution of experiments, memory management, the errors management, and the parallelism.



- **Experiments:** various settings regarding the execution of experiments.
 - **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.
 - **Ask to go to fullscreen mode:** if true, ask the modeler before switching to the fullscreen mode.
 - **Synchronize outputs with the simulation:** 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](#)).

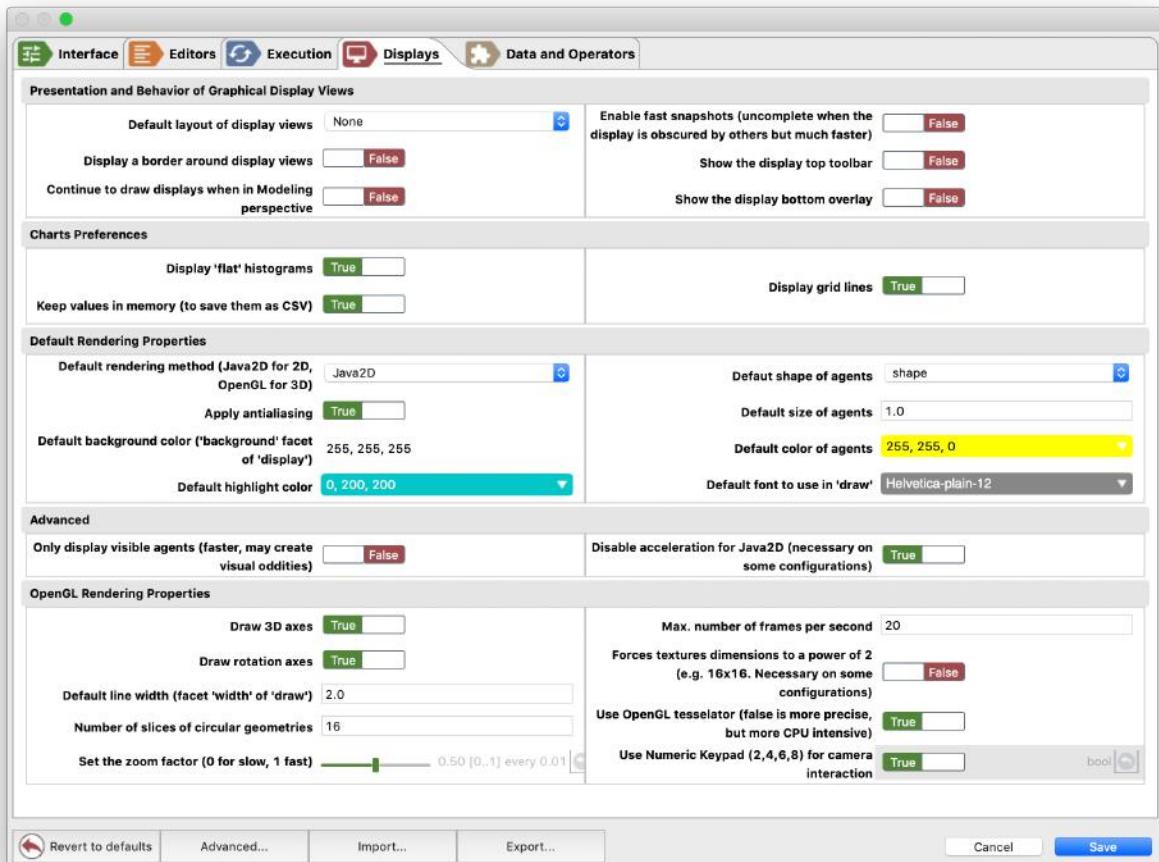
• Tests

- **Sorts the results of tests by severity**
 - **Run tests at each start of the platform**
 - **Include user-defined tests in the tests suite**
 - **Only display (in the UI and in headless runs) failed and aborted tests**
- **Memory:** a given amount of memory (RAM) is allocated to the execution of GAMA (it has to be set in the `Gama.ini` file). The allocated memory size should be chosen in accordance with the requirements of the model that is developed and the other applications running in your OS.
 - **Monitor memory and emit a warning if it is low:** a warning will appear during an experiment run when the memory is low.
 - **Trigger warnings when the percentage of available memory is below**
 - **Interval (in seconds) at which memory should be monitored**
 - **If true, when running out of memory, GAMA will try to close the experiment, otherwise it exits**
- **Runtime errors:** how to manage and consider simulation errors.
 - **Show execution errors:** whether errors should be displayed or not.
 - **Show errors thrown in displays and outputs:** the code defined inside the `aspect` block of a species will be executed each time the agents are repainted in a display. In particular, when the displays are not synchronized, some errors can occur due to some inconsistency between the model and the display (e.g. drawing a dead agent). As a consequence, the code executed inside an aspect should be limited as much as possible.
 - **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).

- **Automatically open an editor and point at the faulty part of the model if an error or a warning is thrown**
- **Text color of errors**
- **Text color of warnings**
- **Parallelism:** various settings regarding the parallel execution of experiments.
 - **Make experiments run simulations in parallel:** if true, in the case of a multi-simulations experiment, the simulation will be executed in parallel (note that the number of simulations that can be executed in parallel will depend on the number of threads to use).
 - **Make grids schedule their agents in parallel:** the agents of grid species will be executed in parallel. Depending on the model, this could increase the simulation speed, but the modeler cannot have any control over the execution order of the agents.
 - **Make species schedule their agents in parallel**
 - **Number under which agents are executed sequentially**
 - **Max. number of threads to use (available processors: 8)**

Displays



- **Presentation and Behavior of Graphical Display Views**

- **Default layout of display views:** among [None, stacked, Split, Horizontal, Vertical]. When an experiment defines several displays, they are by default (layout None) opened in the same View. This preference can set automatically this layout. A `layout` statement can also be used in `experiment` to redefine programmatically the layout of display views.
- **Display a border around display views**
- **Continue to draw displays when in Modeling perspective:** if true, when

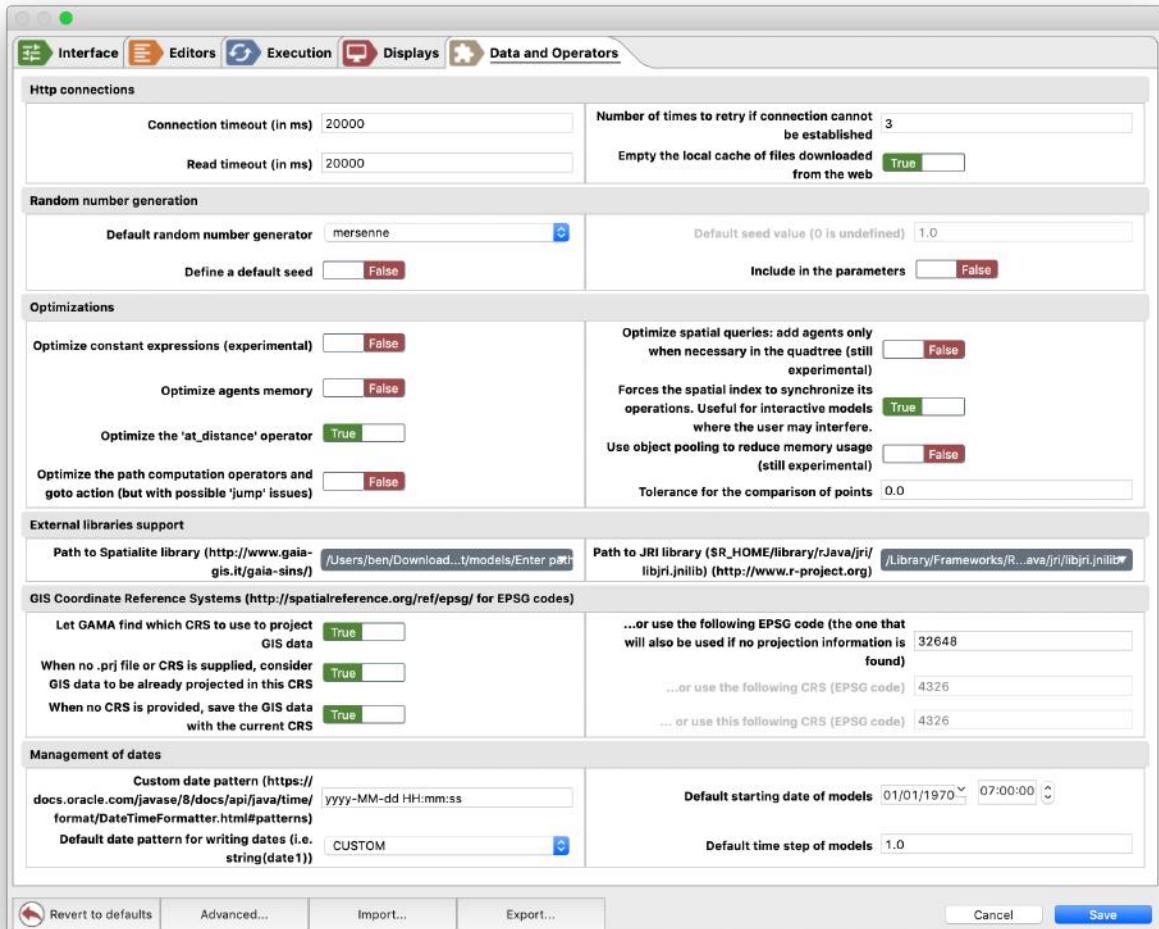
the simulation is running and the modeler chooses to switch to the Modeling perspective the displays are still updated. This is particularly relevant for displays showing plots of data over time.

- **Enable fast snapshots (uncomplete when the display is obscured but much faster)**
 - **Show the display top toolbar:** this could also be configured manually for each display (cf [displays related page](#)).
 - **Show the display bottom overlay:** this could also be configured manually for each display (cf [displays related page](#)).
- **Charts Preferences**
 - **Display 'flat' histograms:** if false, the histograms are displayed in a 3D style.
 - **Keep values in memory (to save them as csv)**
 - **Display grid lines:** in charts (and in particular `series`), if true, a grid is displayed in background.
 - **Default Rendering Properties:** various properties of displays
 - **Default rendering method (JavaED fro 2D, OpenGL for 3D):** use either 'Java2D' or 'OpenGL' if nothing is specified in the [declaration of a display](#).
 - **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.
 - **Default shape of agents:** a choice between `shape` (which represents the actual geometrical shape of the agent) and geometrical operators (`circle`, `square`, `triangle`, `point`, `cube`, `sphere` etc.) as default shape to display agents when no `aspect` is defined.

- **Default size of agents:** what size to use. This expression must be constant.
 - **Default color of agents:** what color to use.
 - **Default font to use in 'draw'**
- **Advanced:**
 - **Only display visible agents (faster, may create visual oddities)**
 - **Disable acceleration for Java2D (necessary on some configurations)**
- **OpenGL Rendering Properties:** various properties specific to OpenGL-based displays
 - **Draw 3D axes:** if true, the shape of the world and the 3 axes are drawn
 - **Draw rotation axes:** if true, a sphere appears when rotating the scene to illustrate the rotations.
 - **Default line width (facet `width` of `draw`):** the value is used in [draw statement](#) that draws a line without specifying the `width` facet.
 - **Number of slices of circular geometries:** when a circular geometry (circle, sphere, cylinder) is displayed, it needs to be discretized in a given number of slices.
 - **Set the zoom factor (0 for slow, 1 fast):** this determines the speed of the zoom (in and out), and thus its precision.
 - **Max. number of frames per second**
 - **Forces textures dimension to a power of 2 (e.g. 16x16. Necessary on some configurations)**
 - **Use OpenGL tessellator (false is more precise, but more CPU intensive)**
 - **Use Numeric Keypad (2,4,6,8) for camera interaction:** use these numeric keys to [make quick rotations](#).

Data and Operators

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



• Http connections

- **Connection timeout (in ms)**: set the connection timeout when the model tries to access a resource on the web. This value is used to decide when to give up the connection try to an HTTP server in case of response absence.
- **Read timeout (in ms)**: similar to connection timeout, but related to the time GAMA will wait for a response in case of reading demand.
- **Number of times to retry if connection cannot be established**
- **Empty the local cache of files downloaded from the web**: if true, after having downloaded the files and used them in the model, the files will be deleted.

- **Random Number Generation:** all the options pertaining to generating random numbers in simulations
 - **Default 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 (0 is undefined):** the value of this default seed
 - **Include in the parameter:** whether the choice of generator and seed is included by default in the [parameters views](#) of experiments or not.
- **Optimizations**
 - **Optimize constant expressions (experimental):** whether expressions considered as constants should be computed and replaced by their value when compiling models. Allows to save memory and speed, but may cause some problems with complex expressions.
 - **Optimize agents memory:** whether the memory used by agents is reduced (or not) when their structure appears to be simple: no sub-agents, for instance, because no sub-species is defined.
 - **Optimize the 'at_distance' operator:** an optimisation that considers the number of elements on each side and changes the loop to consider the fastest case.
 - **Optimize the path computation operators and goto action (but with possible 'jump' issues):** when an agent is not already on a path, simplifies its choice of the closest segment to choose and makes it jump directly on it rather than letting it move towards the segment.
 - **Optimize spatial queries: add agents only when necessary in the quadtree (still experimental):** if no queries is conducted against a species of agents, then it is not necessary to maintain them in the global quad tree.
 - **Forces the spatial index to synchronize its operations. Useful for interactive models where the user may interfere.:** when true, forces the quadtree to use concurrent data structures and to synchronize reads and

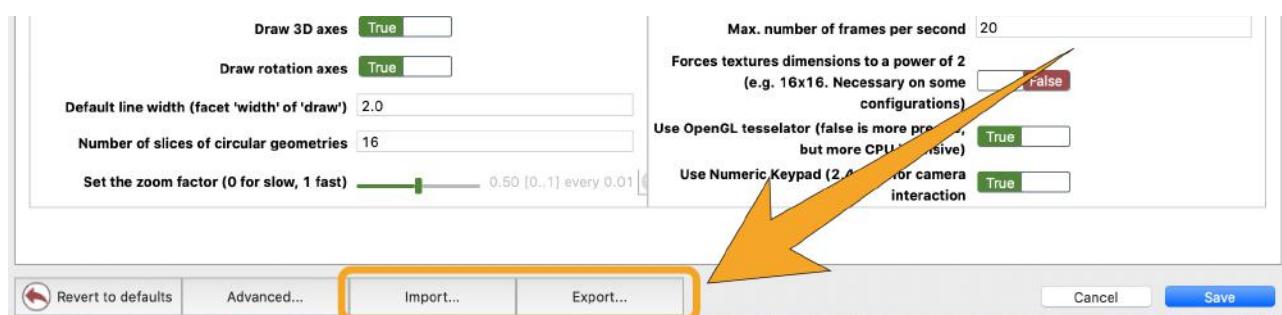
writes, allowing users to interact with the simulation without raising concurrent modification errors.

- **Use object pooling to reduce memory usage (still experimental):** when true, tries to reuse the same common objects (lists, maps, etc.) over and over rather than creating new ones.
- **Tolerance for the comparison of points:** depending on the way they are computed, 2 points who should be the same, could not be equal. This preference allows to be more tolerant in the way points are compared.
- **External libraries support**
 - **Path to Spatialite (<http://www.gaia-gis.it/gaia-sins/>):** the path toward the spatial extension for the SQLite database.
 - **Path to JRI library (\$R_HOME/library/rJava/jri/libjri.jnilib) (<http://www.r-project.org>):** when we need to couple GAMA and R, we need to set properly the path toward this file.
- **GIS Coordinate Reference Systems (<http://spatialreference.org/ref/epsg/> for EPSG codes):** 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 `true` (i.e. GAMA will always try to find the relevant CRS, and, if none can be found, will fall back one the one provided below)
 - **...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. If the option above is `false`, then the use of this CRS will be enforced in all models. Otherwise, GAMA will first try to find the most relevant CRS and then fall back on this one.
 - **When no .prj file or CRS is supplied, consider GIS data to be already projected in the CRS:** 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.
- Management of dates: some preferences for default values related to [the dates in GAMA](#).
 - Custom date pattern (<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns>)
 - Default date pattern for writing dates (i.e. `string(date1)`)
 - Default starting date of models: set the default value of the [global variable starting_date](#).
 - Default time step of models: define the default duration of a simulation step, i.e. the value of the variable `step` ([by default, it is set to 1s](#)).

Manage preferences in GAML

All these preferences can be accessed (set or read) directly in a GAML model. To share your preferences with others (e.g. when you [report an issue](#)), you can simply export your preferences in a GAML model. Importing preferences will set your preferences from an external GAML file.



When you export your preferences, the GAML file will look like the following code. It

contains 2 experiments: one to display all the preferences in the console and the other one to set your preferences will the values written in the model.

```
model preferences

experiment 'Display Preferences' type: gui {
init {
    //Append the name of simulations to their outputs
    write sample(gama.pref_append_simulation_name);

    //Display grid lines
    write sample(gama.pref_chart_display_gridlines);

    //Monitor memory and emit a warning if it is low
    write sample(gama.pref_check_memory);

    //Max. number of characters to keep when paused (-1 = unlimited)
    write sample(gama.pref_console_buffer);

    //Max. number of characters to display (-1 = unlimited)
    write sample(gama.pref_console_size);

    //Wrap long lines (can slow down output)
    write sample(gama.pref_console_wrap);

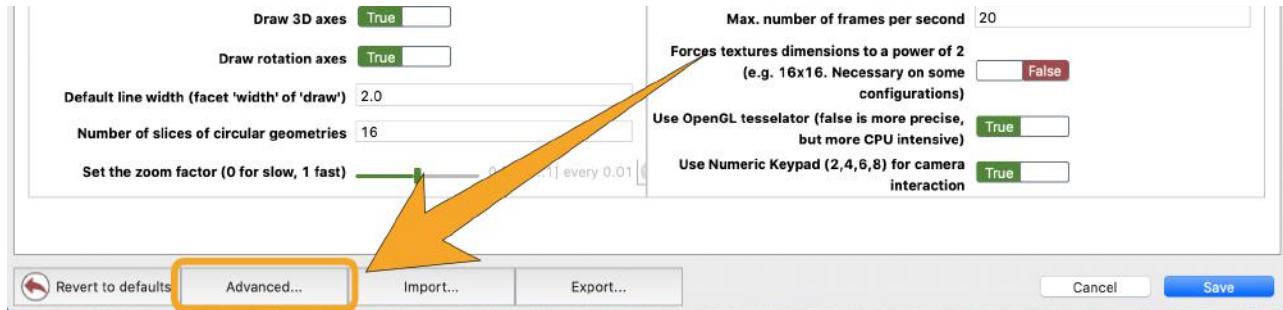
    //Custom date pattern (https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns)
    write sample(gama.pref_date_custom_formatter);
}

// ...
```

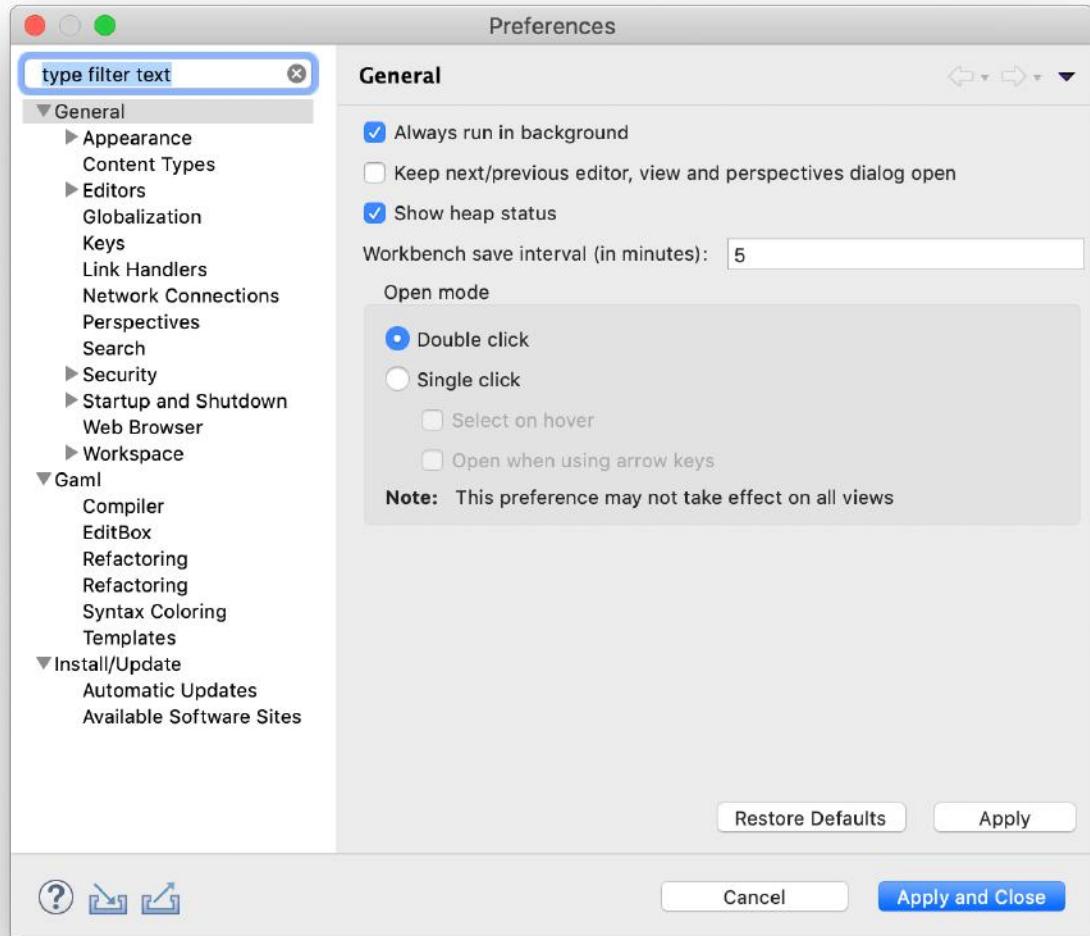
Advanced Preferences

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](#)).



Version: 1.9.1

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 OpenGL display crash GAMA

In case GAMA crashes whenever trying to display an OpenGL display or a Java2D, and you are running Ubuntu 21.10 (or earlier), it probably means that you're using **Wayland** as Display backend. You can fix it by running in a terminal `export GDK_BACKEND=x11` and launch GAMA from this same terminal. This workaround is described [here](#) and in [Issue 3373](#).

Wrong dark theme

GAMA have trouble managing custom GTK theme (specially dark ones, see [this issue](#)). The simplest solution is to explicit change the theme to the default Adwaiata as an override of the environment variable.

Change desktop application

Simply edit the file at `/usr/share/applications/gama-platform.desktop` and add `GTK_THEME=Adwaita` on the line starting by `Exec=`. You should have something like this :

```
Exec=env GTK_THEME=Adwaita GDK_BACKEND=x11 /opt/gama-platform/Gama
```

Save the file wait a few seconds and restart GAMA normally.

Note, if you want to force the dark mode, add this instead `GTK_THEME=Adwaita:dark`

From command line (hard)

If you are starting GAMA from the command line, use this command :

```
GTK_THEME=Adwaita /path/to/Gama
```

or this one to use the dark theme variant :

```
GTK_THEME=Adwaita:dark /path/to/Gama
```

On macOS

First launch of GAMA should be in GUI mode

When GAMA has just been downloaded and installed, it needs to be first launched in its GUI version before using it in the headless mode. If it is first launched in the headless mode, GAMA will be damaged and the installed version needs to be

removed and re-installed.

Detached displays "vanish" when moved to a secondary monitor (see #3670)

This is a known bug in Eclipse as well, only on macOS. The only workaround consists in (1) detaching the display as usual on the same monitor than GAMA; (2) pressing F3 to display all active windows on screen; (3) grabbing and moving "by hand" the window corresponding to the detached display to the second monitor. It will then work as usual.

On Windows

Problem with some Radeon graphics cards and OpenGl display

Some Radeon graphics cards may cause GAMA to crash when using OpenGl displays. The best solutions in this case are either to switch to java2D display or, if the computer is equipped with two graphics cards, to specify that the other graphics card should be used for GAMA (see [here](#)). Sometimes just setting the second GPU as recommended for GAMA won't be enough and Windows will still try to run it from the Radeon chipset, you can try setting the second GPU as the default GPU for everything, for example with NVIDIA cards, in NVidia control panel you can set it as `prefered graphics processor` in the `Global settings` tab. Alternatively, it has been reported that installing the latest version of `AMD Software: Adrenalin edition` (last tried successfully with 22.10.1) solved the problem but at the cost of very slower rendering.

Problem with java2D displays

For high-DPI screens, it is possible to observe an offset in java2D displays (not centered, not taking the whole panel, with an erroneous mouse location) with some scaling ratios. [Changing the scaling factor](#) to 100%, 125%, 150% or 200% should solve the problem.

General display problems (blurry icons, strange experiment displays, dark icons, blurry text, different appearance on second screen etc.)

In some computer we noticed numerous display problems those are hard to reproduce because they depend on a specific mix between hardware and software. If you ever encounter that kind of issue, there are two ways for you to try and act on them: the high DPI settings and the Windows scaling ratio.

High DPI settings

Most of those problems can be solved by setting the right high DPI settings in Windows. To do so, go to your `Gama.exe` file and right click on it. There chose `Properties` and then click on the `Compatibility` tab. Finally click on the `Change high DPI settings` button:

G Gama.exe Properties X

Security	Details	Previous Versions
General	Compatibility	Digital Signatures

If this program isn't working correctly on this version of Windows, try running the compatibility troubleshooter.

[Run compatibility troubleshooter](#)

[How do I choose compatibility settings manually?](#)

Compatibility mode

Run this program in compatibility mode for:

Windows 8 ▾

Settings

Reduced color mode

8-bit (256) color ▾

Run in 640 x 480 screen resolution

Disable fullscreen optimizations

Run this program as an administrator

Register this program for restart

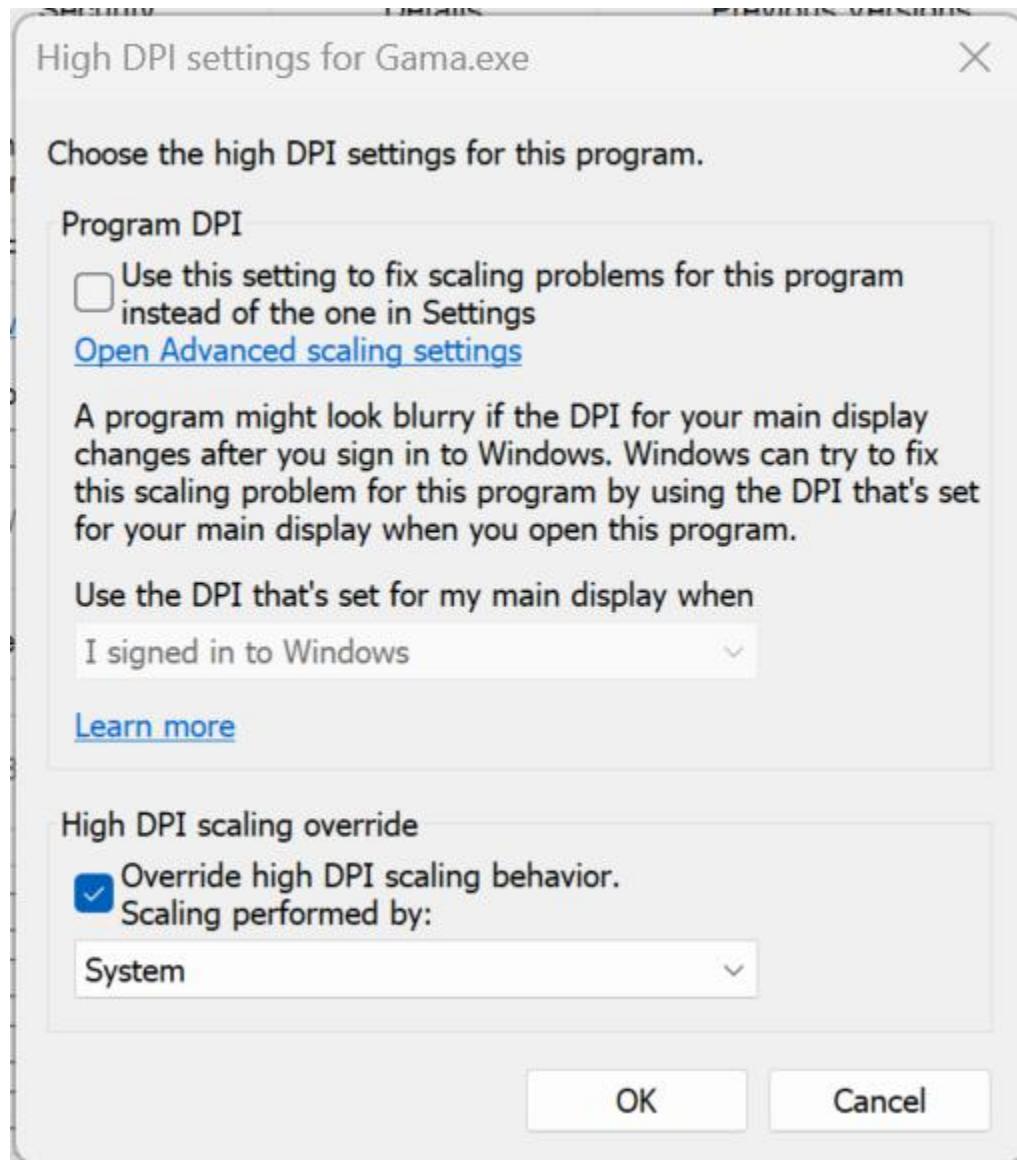
Use legacy display ICC color management

Change high DPI settings

 Change settings for all users

OK Cancel Apply

A new window opens, if you installed gama through the installer you should see the **High DPI scaling override** option checked and the **System** value selected.



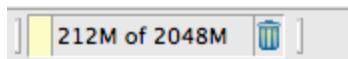
If it's not the case, you can try to set it and then in the properties window click on **Apply** and try to run gama again to see if there's some improvement. If not you can try with different values or to play with the **Program DPI** setting too.

Scaling ratio

Many of those issues are related to the scaling ratio you are using in your windows. If the previous tip didn't work, you can try to play a bit with your scaling ratios to see if there's any improvement. In general, if you have problems we recommend that you stick to the 100% or 200% as those are the values that works the best from our experience. If you have multiple displays and experience problems when moving gama from one to another, we also recommend that you use the same scaling ratio for the two displays. If not possible, setting as you main monitor the one were gama is going to run could also solve some issues.

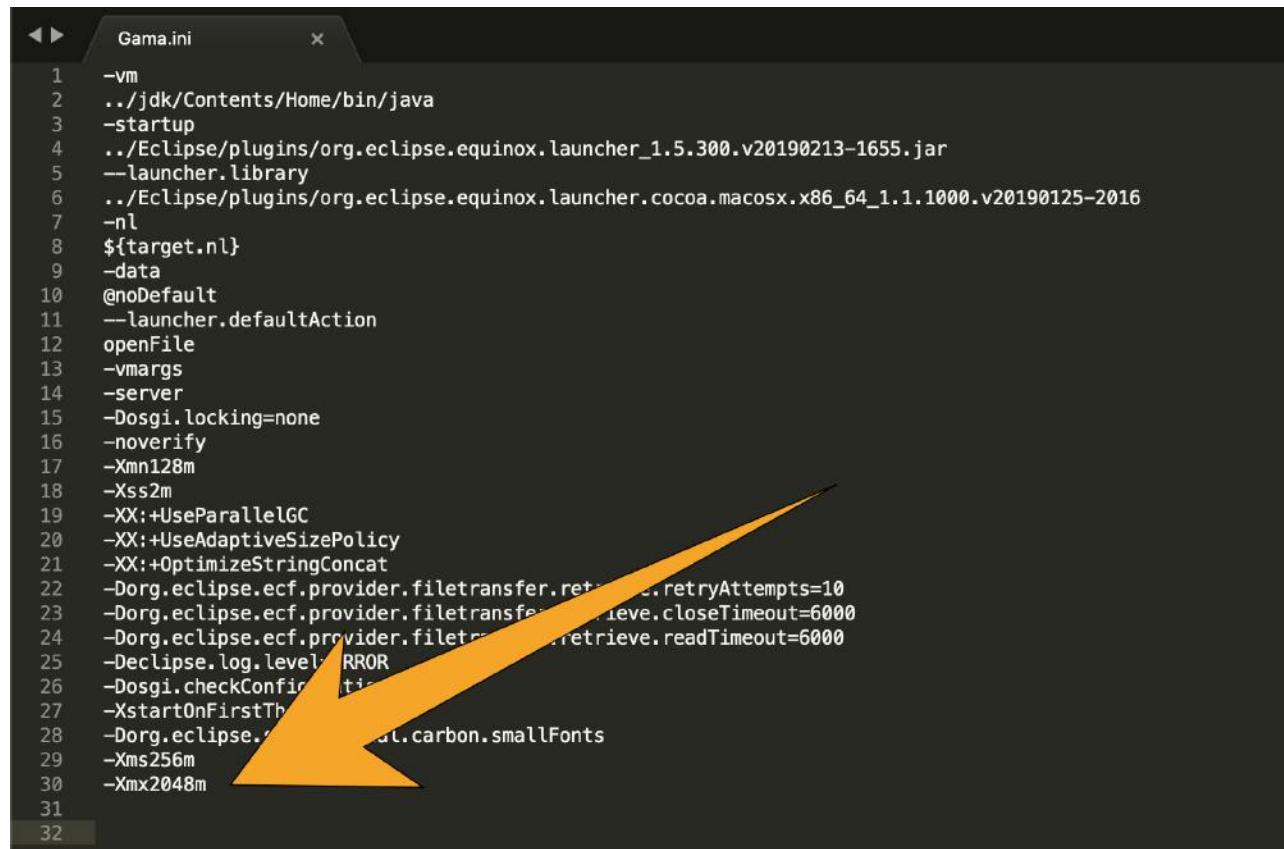
Memory problems

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 2 and 4GB of memory to run "normally" and launch small models. Memory problems are easy to detect: in 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 Mac OS X, you have to right-click on `Gama.app`, choose "Display Package Contents...", and you will find `Gama.ini` in `Contents/Eclipse`. 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 -vm
2 ../jdk/Contents/Home/bin/java
3 -startup
4 ../Eclipse/plugins/org.eclipse.equinox.launcher_1.5.300.v20190213-1655.jar
5 --launcher.library
6 ../Eclipse/plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.1000.v20190125-2016
7 -nl
8 ${target.nl}
9 -data
10 @noDefault
11 --launcher.defaultAction
12 openFile
13 -vmargs
14 -server
15 -Dosgi.locking=none
16 -noverify
17 -Xmn128m
18 -Xss2m
19 -XX:+UseParallelGC
20 -XX:+UseAdaptiveSizePolicy
21 -XX:+OptimizeStringConcat
22 -Dorg.eclipse.ecf.provider.filetransfer.retrieve.retryAttempts=10
23 -Dorg.eclipse.ecf.provider.filetransfer.retrieve.closeTimeout=6000
24 -Dorg.eclipse.ecf.provider.filetransfer.retrieve.readTimeout=6000
25 -Declipse.log.level=RROR
26 -Dosgi.checkConfigurations=true
27 -XstartOnFirstThread
28 -Dorg.eclipse.swt.carbon.smallFonts
29 -Xms256m
30 -Xmx2048m
31
32
```

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

Charting problems

By default the charts of a running experiment are only updated when you are in the experiment view. Therefore if you want to be able to run an experiment and plot its results while still working on the code of a model, you should make sure that the option `Continue to draw displays when in Modeling perspective` is set to true in the `Presentation and Behavior of Graphical Display Views` section of the `Display` tab in the settings.

Installation is broken

It may happen that after switching from one GAMA version to another, or after installing a plugin, something breaks your GAMA installation completely and uninstalling/reinstalling won't solve the problem. To fix this, you can go to your home directory and find the `.eclipse` (hidden) folder. For example on Windows it would be at:

```
C:\Users\username\.eclipse
```

There you will find a list of directories all starting with `org.` and one directory with the name starting with a number followed by the system you are using, for example for Windows it could be : `306334380_win32_win32_x86_64`, for linux `1164258503_linux_gtk_x86_64` etc. That directory contains the list of plugins and some config files that are persistent from one version to another, you can rename it (to `306334380_win32_win32_x86_64-backup` for example) to keep a track of what was your configuration before, and run GAMA again. GAMA should then create a new clean directory with the basic configuration and no plugin installed, which should solve configuration related problems.

Silent error when saving a file

In certain configurations, when using the [save statement](#) an error can happen while the simulation is running and trying to perform the save. To prevent those, make sure that you specified the format in which to save your data (`csv`, `text`, `json` etc.) with the `format` facet.

Saving SHP file raises an error

If you encounter a runtime error while trying to save an SHP file multiple times, especially if the message is something like `Java error: I/O error ... FileNotFoundException...`, you can try going into your gama preferences and go to `Data and Operators` -> `Optimizations` -> `In-memory shapefile mapping [...]` and set it to false.

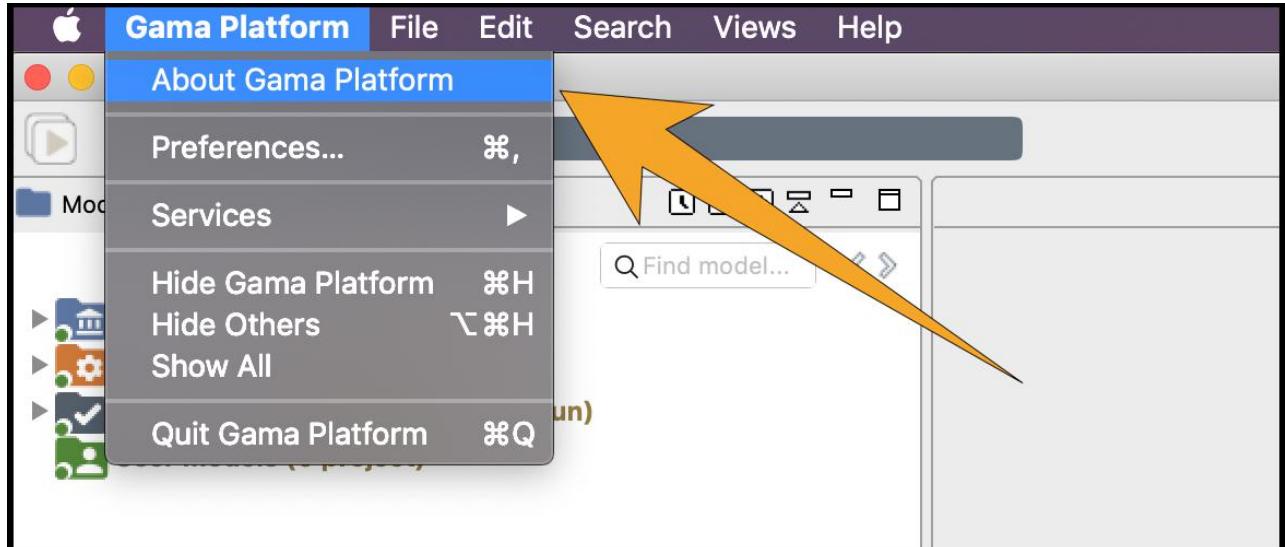
Submitting an Issue

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

- A complete description of the problem and how it occurred.
- The GAMA model or code you are having trouble with. If possible, attach a complete model.
- Screenshots or other files that help describe the issue.

Two files may be particularly interesting to attach to your issue: the [configuration](#)

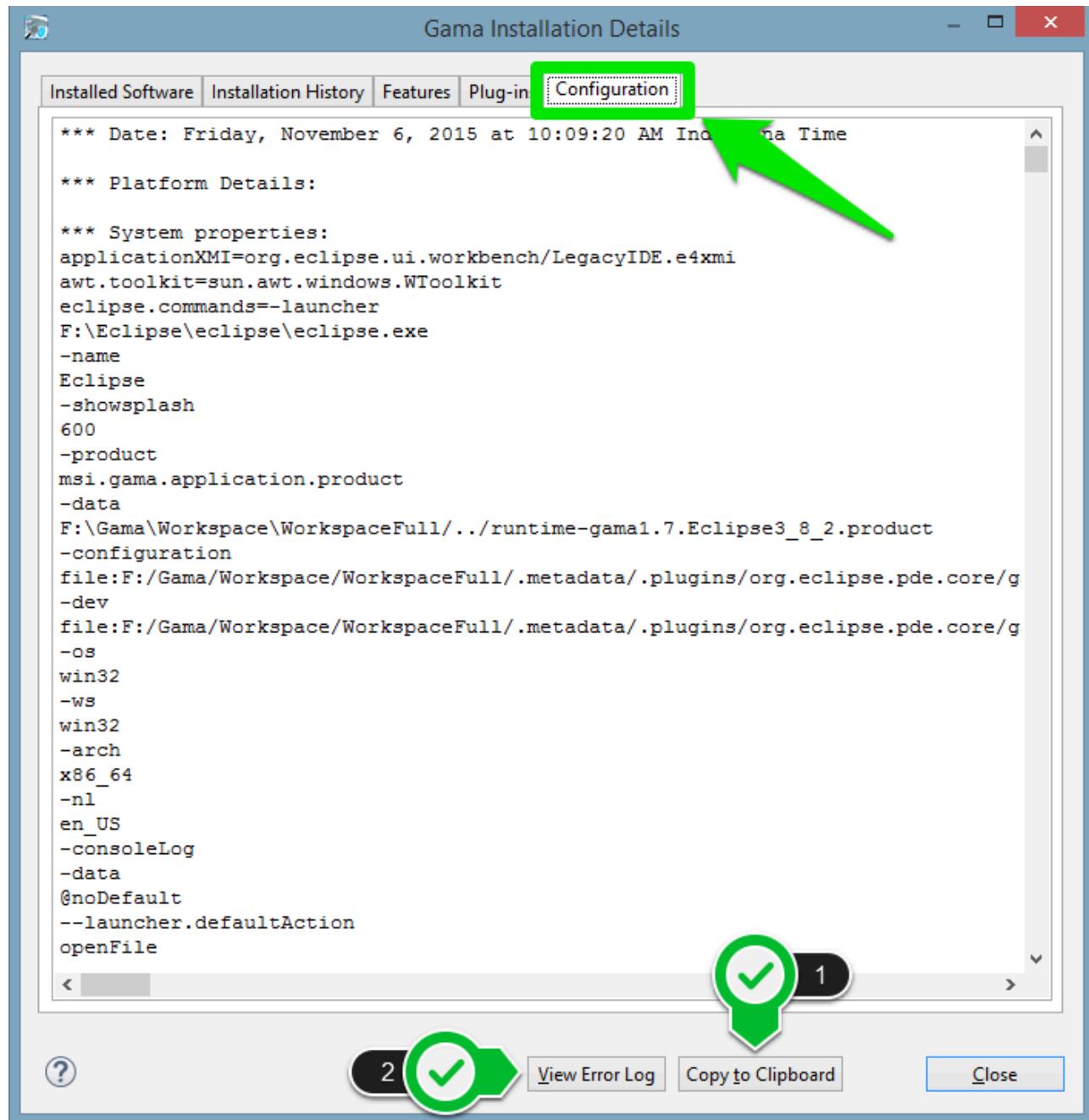
details and the **error log**. Both can be obtained quite easily from within GAMA itself in a few steps. First, click the "About GAMA..." menu item (under the "Gama Platform" menu on Mac OS X, "Help" menu on Linux & Windows)



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



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

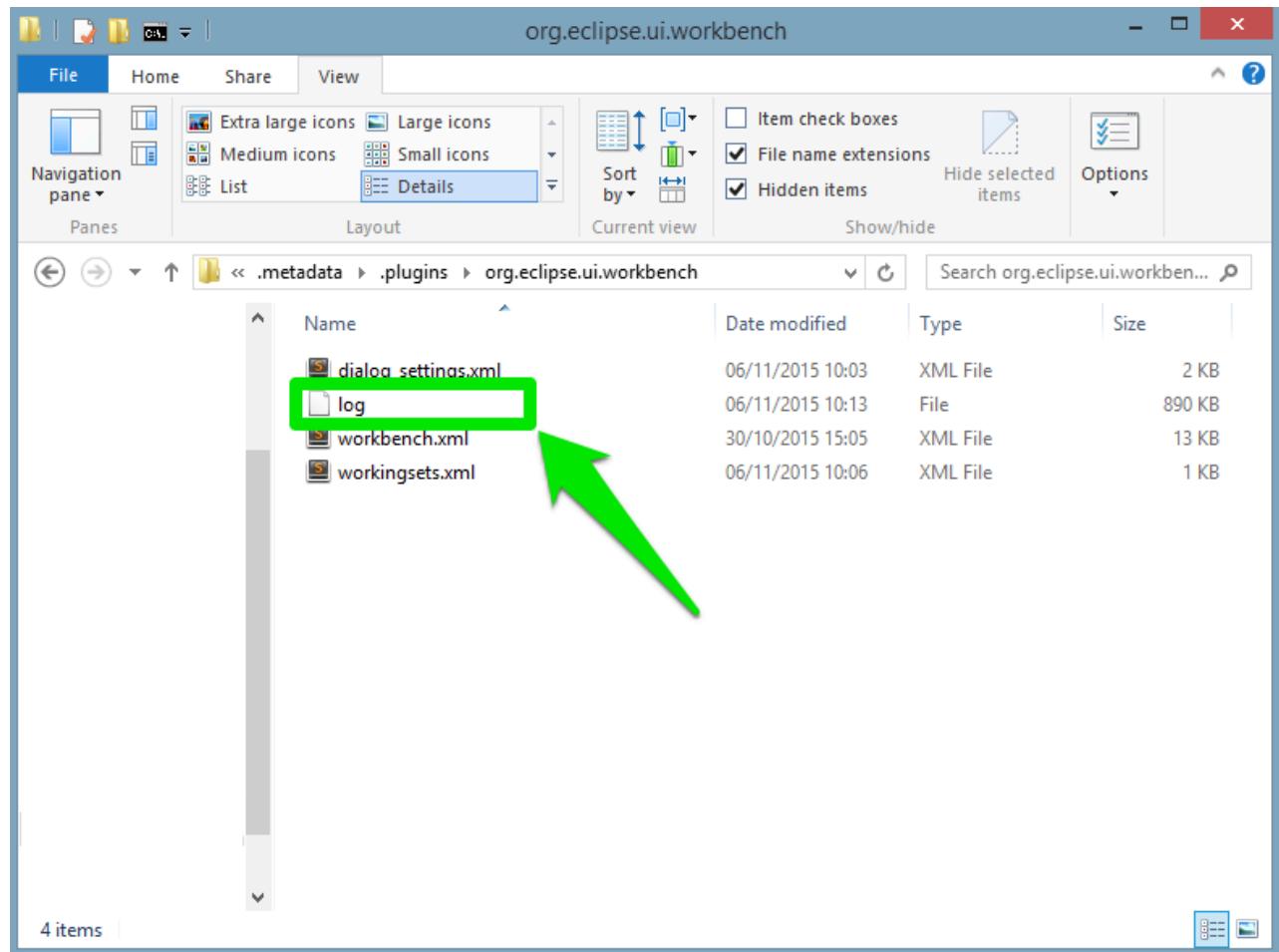


To provide complete information about the status of your system at the time of the error, you can

- (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 (usingTextEdit, 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.





>

Version: 1.9.1

Learn GAML Step by Step

This large progressive tutorial has been designed to help you to learn **GAML (GAmma Modeling Language)**. It will cover the main part of the possibilities provided by GAML, and guide you to learn some more.

How to proceed to learn better?

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

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

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

Version: 1.9.1

Introduction

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

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

This orientation has several conceptual consequences among which at least two are of immediate practical interest for modelers:

- Since simulations, or experiments, are represented by agents, GAMA is bound to support high-level *model compositionality*, i.e. the definition of models that can use other models as *inner agents*, leveraging multi-modeling or multi-paradigm modeling as particular cases of composition.

- The *visualization* of models can be expressed by *models of visualization*, composed of agents entirely dedicated to visually represent other agents, allowing for a clear *separation of concerns* between a simulation and its representation and, hence, the possibility to play with multiple representations of the same model at once.

Table of contents

- Key Concepts (Under construction)
 - Lexical semantics of GAML
 - Translation into a concrete syntax
 - Vocabulary correspondance with the object-oriented paradigm as in Java
 - Vocabulary correspondance with the agent-based paradigm as in NetLogo

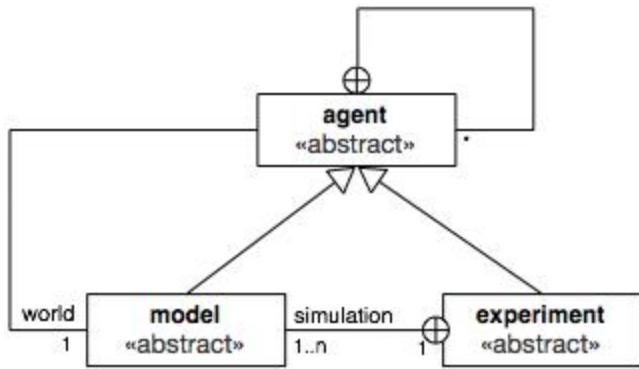
Lexical semantics of GAML

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

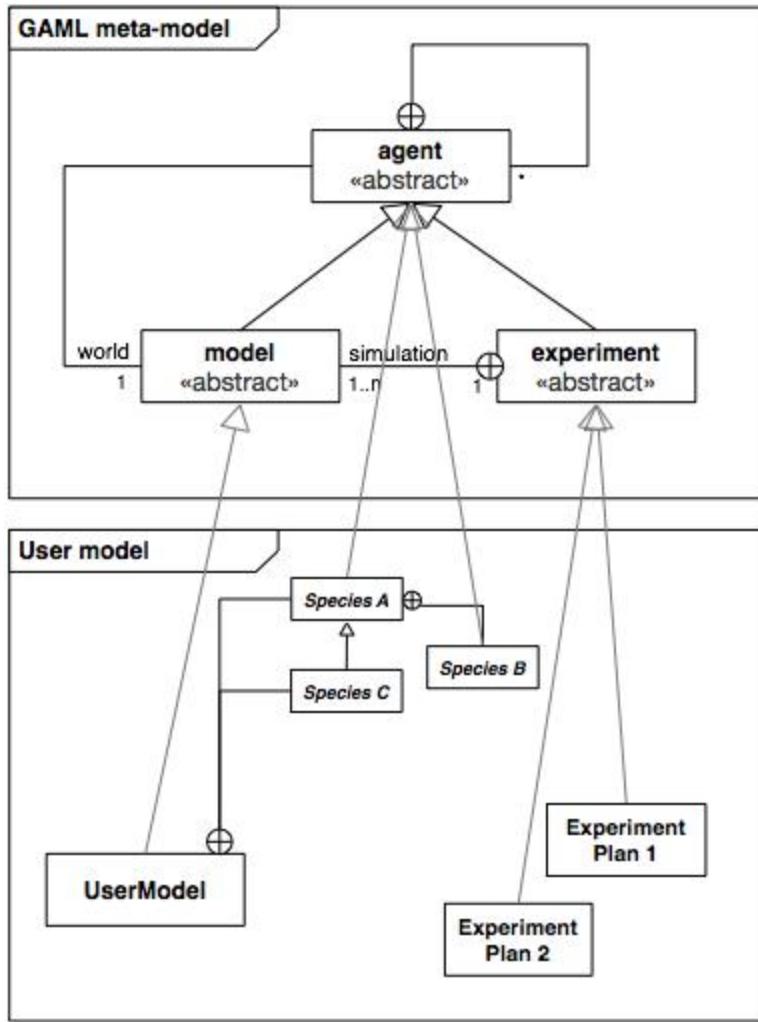
1. The role of GAML is to support modelers in writing **models**, which are specifications of **simulations** that can be executed and controlled during **experiments**, themselves specified by **experiment plans**.
2. The **agent-oriented** modeling paradigm means that everything "active" (entities of a model, systems, processes, activities, like simulations and experiments) can be represented in GAML as an **agent** (which can be thought of as a computational component owning its own data and executing its own behavior, alone or in interaction with other agents).
3. Like in the object-oriented paradigm, where the notion of *class* is used to supply

a specification for *objects*, agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they know*), **actions** (*what they can do*), **behaviors** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology** (*how they are connected*) or **schedule** (*in which order and when they should execute*).

4. Any **species** can be nested in another **species** (called its *macro-species*), in which case the **populations** of its instances will imperatively be hosted by an instance of this *macro-species*. A **species** can also inherit its properties from another **species** (called its *parent species*), creating a relationship similar to *specialization* in object-oriented design. In addition to this, **species** can be constructed in a compositional way with the notion of **skills**, bundles of **attributes** and **actions** that can be shared between different species and inherited by their children.
5. Given that all **agents** are specified by a **species**, **simulations** and **experiments** are then instances of two species which are, respectively, called **model** and **experiment plan**. Think of them as "specialized" categories of species.
6. The relationships between **species**, **models** and **experiment plans** are codified in the meta-model of GAML in the form of a framework composed of three abstract species respectively called **agent** (direct or indirect parent of all **species**), **model** (parent of all **species** that define a model) and **experiment** (parent of all **species** that define an experiment plan). In this meta-model, instances of the children of **agent** know the instance of the child of **model** in which they are hosted as their **world**, while the instance of **experiment plan** identifies the same agent as one of the **simulations** it is in charge of. The following diagram summarizes this framework:



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, an *experiment* agent is created. Its behavior, specified by its *experiment plan*, will create simulations agents (instance of the user *model*) and execute them. Recursively, the initialization of a simulation agent will create the agent population of the species defined in the model. Each of these agents, when they are created, can create the population of their micro-species...

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 correspondence (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 correspondence 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)

GAML	Java
statement	statement
type	type
parametric type	generics

Vocabulary correspondence 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

GAML	NetLogo
behavior	collection of global functions applied to one breed
aspect	only one, mixed with the behavior
skill	-
statement	primitive
type	type
parametric type	-



Version: 1.9.1

Start with GAML

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

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



Version: 1.9.1

Organization of a model

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

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

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

Table of contents

- Model Header (model species)
- Import gaml file
- Species declarations
- Experiment declarations
- Basic skeleton of a model

Model Header (*model species*)

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

```
model name_of_the_model
```

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

```
model dummy
```

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

Import gaml file

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

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

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

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

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

When importing models using the first form, all the declarations of the model(s) imported will be merged with those of the current model (in the order with which the import statements are declared, i.e. the latest definitions of global attributes or behaviors superseding the previous ones).

The second form is reserved for [using models as *micro-models* of the current model](#). This possibility is still experimental in the current version of GAMA.

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

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

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

Species declarations

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

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

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

Regular [species](#) can be declared with the keyword `species`. You can declare several regular species, and they all have to be named. A species defines its [attributes](#), [actions](#) and [behaviors](#) and [aspects](#).

```
species nameOfSpecies {  
    // definition of your species attributes, actions and behaviors  
    and aspects  
}
```

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

```
global {  
    // definition of global attributes, actions, behaviors  
}  
  
species A {...}  
  
species B {...}
```

is completely equivalent to:

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

    species A {...}

    species B {...}
}
```

Experiment declarations

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

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

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

experiment second_experiment {
    // definition of parameters (inputs)

    // definition of output
}
```

Note that you have four types of experiments:

- A `GUI experiment` allows you to display a graphical interface with input parameters and outputs. It is declared with the following structure:

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

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

```
experiment batch_experiment type:batch {  
    [parameters]  
    [exploration method]  
    [...]  
}
```

- A **Test experiment** allows you to write unit tests on a model (used to ensure its quality). It is declared with the following structure:

```
experiment test_experiment type:test autorun: true {  
    [setup]  
    [tests]  
    [...]  
}
```

- A **memorize experiment** allows you to store each step of the simulation in memory and to backtrack to previous steps. It is declared with the following structure:

```
experiment test_experiment type:memorize {  
    [parameters]  
    [output]  
    [...]
```

Basic skeleton of a model

Here is the basic skeleton of a model :

```
model name_of_the_model

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

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

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

Don't forget this structure! This will be the basis for all the models you will create from now.



Version: 1.9.1

Basic programming concepts in GAML

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

Index

- [Variables](#)
 - [Basic types](#)
 - [The point type](#)
 - [A word about dimensions](#)
- [Declare variables using facet](#)
- [Operators in GAMA](#)
 - [Logical operators](#)
 - [Comparison operators](#)
 - [Type casting operators](#)
 - [Other operators](#)
- [Conditional structures](#)
- [Loop](#)
- [Manipulate containers](#)
- [Random values](#)

Variables

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

```
typeName myVariableName;
```

Basic types

All the "basic" types are present in GAML: `int`, `float`, `string`, `bool` (see [the data type page](#) for information about all the available datatype). The operator for the affectation in GAML is `<-` (the operator `=` is used to test the equality).

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

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

```
model firstModel

global {
```

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

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

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

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

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

The point type

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

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

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

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

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

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

A world about dimensions

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

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

Declare variables using facet

Facets are used to describe the behavior of a variable during its declaration, by

adding the keyword `facet` just after the variable name, followed by the value you want for the facet (or also just after the initial value). See [the page related to the variable declaration for all the facets](#).

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

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

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

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

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

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

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

Operators in GAMA

In GAML language, you can use a lot of different operators. **An operator is a**

function, i.e. a way to get the result of a computation. All of them are listed in this [page](#), but here are the most useful ones:

Mathematical operators

The basic arithmetical operators, such as `+(add)`, `-(subtract)`, `*(multiply)`, `/(divide)`, `^(power)` are used this way:

```
// FirstOperand BinaryOperator SecondOperand  
//    --> ex: 5 * 3; // return 15  
  
int fif <- 5 * 3;
```

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

```
// UnaryOperator(Operand)  
//    --> ex: sqrt(49); // return 7.0  
  
float sq <- sqrt(49);
```

Logical operators

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

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

Comparison operators

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

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

Type casting operators

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

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

Other operators

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

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

Some others are used in a more intuitive way:

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

Conditional structures

You can write conditionals with `if/else` in GAML:

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

GAML also accepts ternary operator:

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

Loop

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

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

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

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

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

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

```
loop i from: 0 to: 5 {
    write i;
}
// the output will be 0 - 1 - 2 - 3 - 4 - 5
```

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

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

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

```
loop i over: [0, 2, 4] {
```

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

Manipulate containers

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

How to declare a list?

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

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

How to know the number of elements of a list?

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

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

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

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

How to get an element from a list?

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

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

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

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

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

How to know if an element exists in a list?

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

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

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

For those operation, you can use dedicated statements. The statements `add` and `put` are used to insert/modify an element, while the statement `remove` is used to remove an element. Here are some examples of how to use those 3 statements with the most common facets:

```

list<int> list_int <- [1,5,7,6,7];

remove from:list_int index:1; // remove the 2nd element of the list
write list_int; // the output is : [1,7,6,7]
remove item:7 from:list_int; // remove the 1st occurrence of 7
write list_int; // the output is : [1,6,7]

add item:9 to: list_int at: 2; // add 9 in the 3rd position
write list_int; // the output is : [1,6,9,7]
add 0 to: list_int; // add 0 in the last position
write list_int; // the output is : [1,6,9,7,0]

put 3 in: list_int at: 0; // put 3 in the 1st position
write list_int; // the output is : [3,6,9,7,0]
put 2 in: list_int key: 2; // put 2 in the 3rd position
write list_int; // the output is : [3,6,2,7,0]

```

Note that the `+` and `-` operators can be used to add an element at the end of a list and to remove the last element of a list:

```

list<int> list_int <- [1,5,7,6,7];

list_int <- list_int + 8;
write list_int; // the output is : [1,5,7,6,7,8]
list_int <- list_int - 7;
write list_int; // the output is : [1,5,7,6,8]

```

How to add 2 lists?

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

```

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

```

How to browse a list?

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

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

How to filter a list?

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

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

Other useful operators for the manipulation of lists:

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

Random values

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

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

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

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

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

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

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

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



Version: 1.9.1

Manipulate basic species

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

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



Version: 1.9.1

The global species

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

Index

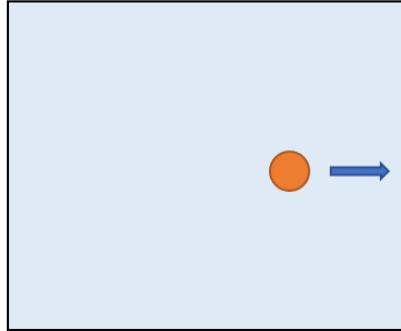
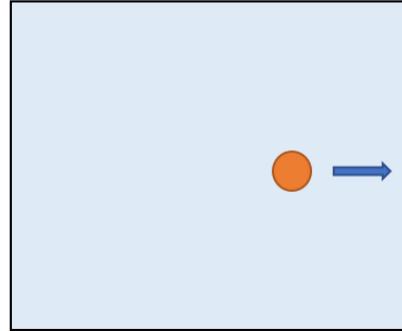
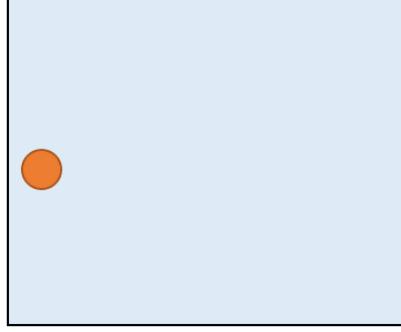
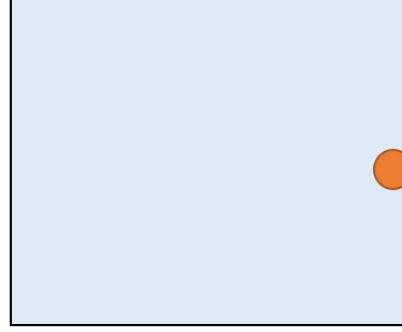
- Declaration
- Environment Size
- Built-in Attributes
- Built-in Actions
- The init statement

Declaration

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

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

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

	<code>torus:true</code>	<code>torus:false</code>
Init situation		
Final situation		

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

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

Directly in the `global` scope, you have to declare all your global attributes (can be seen as "static members" in Java or C++). To declare them, proceed exactly as for

declaring basic variables. Those attributes are accessible wherever you want inside the species scope.

Environment size

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

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

Note that the final shape of the world will always be a rectangle equal to the envelope of the geometry provided.

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

Built-in attributes

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

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

experiment

- contains the `experiment` agent that has created this simulation agent.

cycle

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

To learn more about time, please read the [recipe about dates](#).

step

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

To learn more about time, please read the [recipe about dates](#).

```
global {  
    ...  
    float step <- 10 #h;
```

time

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

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

To learn more about time, please read the [recipe about dates](#).

starting_date and current_date

- date, represent the starting date (resp. the current date) of the simulation. The **current_date** is updated from the **starting_date** by the value **step** at each simulation step.

To learn more about time, please read the [recipe about dates](#).

duration

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

total_duration

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

average_duration

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

machine_time

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

seed

- float, the seed of the random number generator. It will influence the set of random numbers that will be generated all over the simulation. 2 simulations of a model with the same parameters' values should behave identically when the seed is set to the same value. If it is not redefined by the modeler, it will be chosen randomly.

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.

pause

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

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

die

- stops the simulation (in fact it kills the simulation).

```
global {  
    ...  
    reflex halting when: empty (agents) {
```

But beware, it will not kill the simulation, instead the simulation will continue running with a dead global species (so everything will close and nothing will happen except for the cycles still increasing). If you want to completely stop the simulation you will have to call the `die` action of the `host` of the global species.

```
global {  
    ...  
    reflex halting when: empty (agents) {  
        ask host {  
            do die;  
        }  
    }  
}
```

Other actions

Other built-in actions are defined for the model species, just as in [any other regular species](#).

The `init` statement

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

Version: 1.9.1

Regular species

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

Index

- Declaration
- Built-in Attributes
- Species built-in Attributes
- Built-in Actions
- The init statement
- The aspect statement
- Instantiate an agent

Declaration

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

```
species my_specie {  
}
```

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

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

Built-in attributes

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

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

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

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

All those built-in attributes are attributes of an agent (an instance of a species).

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

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

Species built-in Attributes

Species have also their own attributes, which can be accessed with the following syntax (read-only) :

```
name_of_your_species.attribute_you_want
```

Here is the list of those attributes:

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

As an example, the following code illustrates all these attributes:

```
model NewModel

global {
    init {
        create my_species ;
    }
}

species my_species {
    int att1;

    init {
        create my_micro_species;

        write species(self).name;           // write in the console: my_species
        write species(self).attributes;
        // write in the console:
        ['name','shape','location','peers','host','agents','members','att1','my_micro_species']
        write species(self).actions;
    }
}
```

Built-in actions

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

```
species my_species{
    reflex being_killed {
        do debug("I will disappear from the simulation");
        do die;
    }
}
```

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

The 2 actions `_init_` and `_step_` are very important, as they allow the modeler to change totally the agents' dynamics:

- when the action `_init_` is defined in a species, it will be called instead of the `init` block.
- when the action `_step_` is defined in a species, it will be called at each simulation step instead of the species' behaviors (e.g. instead of the reflexes blocks).

The init statement

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

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

The aspect statement

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

```
species my_species {  
    aspect standard_aspect {  
    }  
}
```

You can then define your aspect by using the statement `draw`. You can then choose a geometry for your aspect, an image, a text (facet `text`), and its color (facet `color`)... It is common to have several `draw` statement in an `aspect` to enrich its display. We invite you to read the documentation about the `draw` statement to know more about.

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

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

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

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

Instantiate an agent

As already said quickly in the last session, the instantiation of the agents is most often in the `init` scope of the `global` species (this is not mandatory of course. You can instantiate your agents from an action/behavior of any species). Use the statement `create` to instantiate an agent:

- The first element given to the `create` statement (i.e. the facet `species`) is used to specify which species you want to instantiate.
- The facet `number` is used to tell how many agents you want to create.
- The facet `with` is used to specify some default values for some attributes of your instance. For example, you can specify the location.

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

species my_species {
    int vA;
}
```

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

```
model display_one_agent

global{
    float worldDimension <- 50#m;
    geometry shape <- square(worldDimension);

    init{
        point center <- {worldDimension/2,worldDimension/2};
        create my_species number: 1 with: (location:center);
    }
}

species my_species {
    aspect standard_aspect {
        draw circle(1#m);
```




Version: 1.9.1

Defining actions and behaviors

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

Index

- Action
 - Declare an action
 - Call an action
- Behavior
- Example

Action

Declare an action

An action is a function or procedure run by an instance of species. An action can return a value (in that case, the type of return has to be specified just before the name of the action), or not (in that case, you just have to put the keyword `action` before the name of the action). The former ones are often named **functions**, whereas the latter ones are named **procedures** in many programming languages.

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

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

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

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

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

Call an action

To call an action, it depends whether you want to get the returned value or not:

- to call a procedure (without getting any returned value): you have to use the statement `do`.
- to call a function and thus get the returned value, you need to use `any_agent`

`action(arguments)` and assigned this value to a variable.

You can use the statement `do` in different ways:

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

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

- With parenthesis: after specifying the name of your action, you can specify the values of your arguments in the same order they were declared, between parenthesis (just as if you used an operator):

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

We encourage you to use the second way.

To catch the returned value, you have to skip the `do` statement, and store the value directly in a temporary variable:

```
int var1 <- my_action(5,5.1);
// or
int var1 <- my_action(argA: 5, argB: 5.1);
```

Behavior

A behavior, or reflex, is a set of statements which is called **automatically** at each time step by an agent. Note that, a behavior is linked to an **architecture**; the reflex-based architecture is the default one, others can be used `with the controls facet of the species``.

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

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

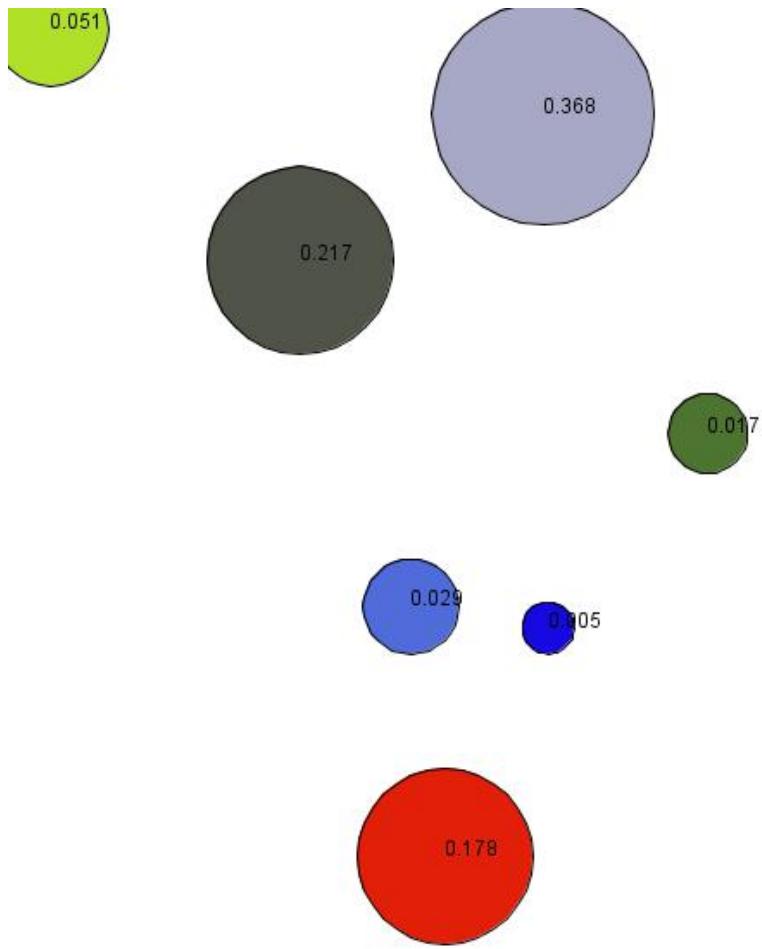
```
reflex my_reflex when: flip(0.5) {  
    write ("Executing the conditional reflex");  
    // statements...  
}
```

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

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

Example

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



Here is one of the multiple possible implementations:

```
model burst_the_baloon

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

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

    reflex endSimulation when: nbBalloonDead>10 {
        do pause;
    }
}
```




> Learn GAML step by step

> Global Species

> Interaction between agents

Version: 1.9.1

Interaction between agents

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

Index

- [The ask statement](#)
- [Pseudo variables](#)
- [Some useful interaction operators](#)
- [Example](#)

The ask statement

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

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

```
species my_species {
```

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

```
species my_species {
    list<agent> targets;

    reflex update {
        ask targets {
            // statements
        }
    }
}
```

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

```
species my_species {
    list<agent> targets;

    reflex update {
        ask targets as:my_species {
            // statements
        }
    }
}
```

- If you want to interact with all the agents of a species (note that the name of the species can be used in the `ask`, and in many other situations, as the population of this species, i.e. the list of agents instance of this species):

```
species my_species {
    reflex update {
        ask other_species {
            // statements
        }
    }
}
```

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

```
ask other_species.population
```

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

```
species my_specie {  
    reflex update {  
        ask agents of-species my_specie {  
            // statements  
        }  
    }  
}
```

Pseudo-variables

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

```
species speciesA {  
    init {  
        name <- "speciesA";  
    }  
  
    reflex update {  
        ask speciesB {  
            write name; // output : "speciesB"  
        }  
    }  
}
```

Now, if we introduce a third species, we can write an `ask` statement inside another.

```
species speciesA {
    init {
        name <- "speciesA";
    }

    reflex update {
        ask speciesB {
            write self.name; // output : "speciesB"
            write myself.name; // output : "speciesA"
            ask speciesC {
                write self.name; // output : "speciesC"
                write myself.name; // output : "speciesB"
            }
        }
    }
}

species speciesB {
    init {
        name <- "speciesB";
    }
}

species speciesC {
    init {
        name <- "speciesC";
    }
}
```

Nb: try to avoid multiple imbrications of `ask` statements. Most of the time, there is another way to do the same thing.

Some useful interaction operators

The operator `at_distance` can be used to know the list of agents that are in a certain distance from another agent.

```
species my_species {
    reflex update {
        list<agent> neighbors <- agents at_distance(5);
        // neighbors contains the list of all the agents located at a
        distance <= 5 from the caller agent.
    }
}
```

The operator `closest_to` returns the closest agent of a position among a container.

```
species my_species {
    reflex update {
        agent agentA <- agents closest_to(self);
        // agentA contains the closest agent from the caller agent.
        agent agentB <- other_specie closest_to({2,3});
        // agentB contains the closest instance of other_specie from the
        location {2,3}.
    }
}

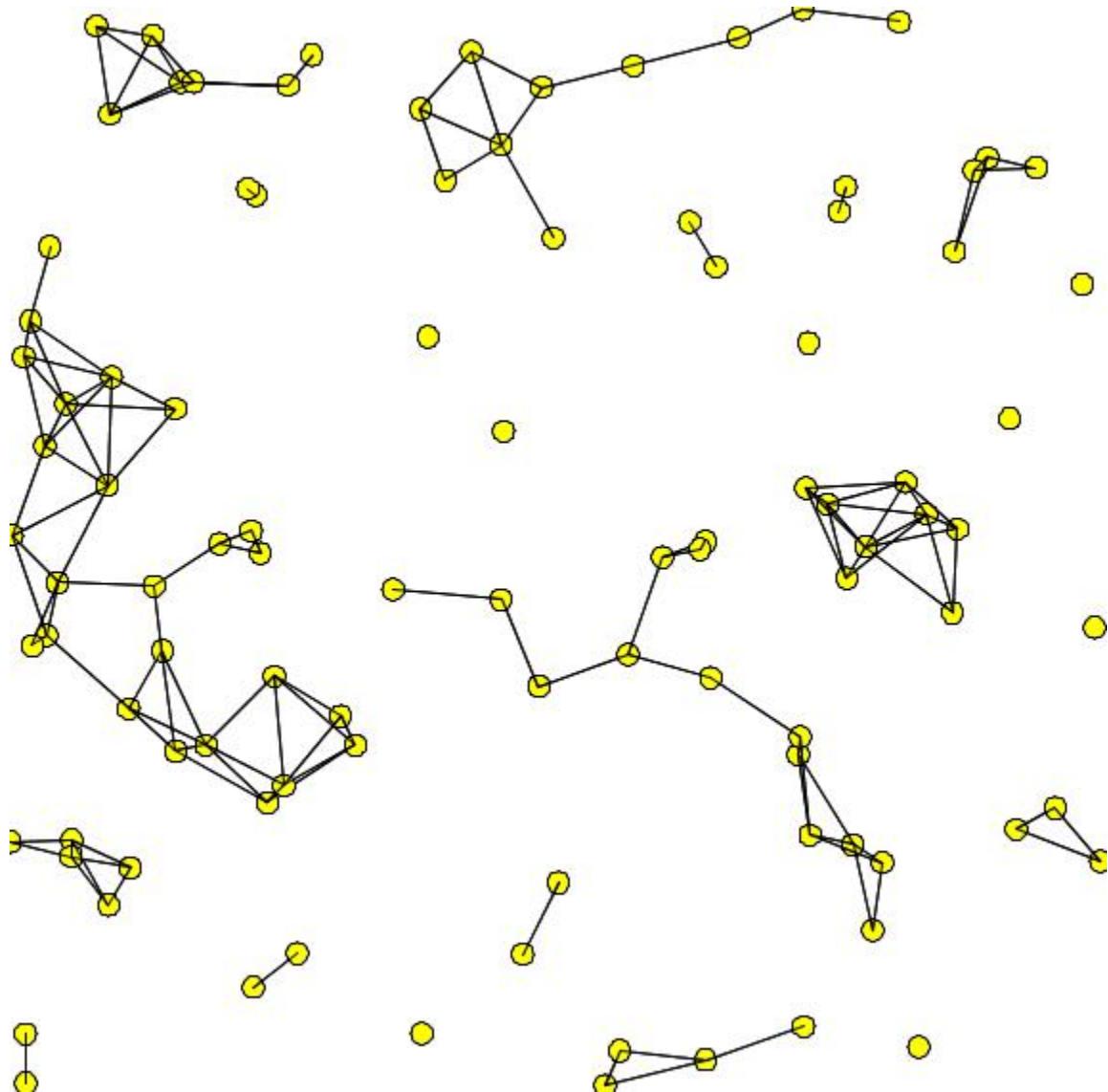
species other_specie { }
```

Example

To practice those notions, here is a short basic example. Let's build a model with a given number of agents with a circle display (keep in mind that their shape has kept

its default value: a point). They can move randomly on the environment (i.e. here move can be understood as changing its location), and when they are close enough from another agent, a line is displayed between them. This line is destroyed when the distance between the two agents is too important.

Hint: use the operator `polyline` to construct a line. List the points between angle brackets [and].



Here is one example of implementation:

```

model connect_the_neighbors

global{
    float speed <- 0.2;
    float distance_to_intercept <- 10.0;
    int number_of_circle <- 100;

    init {
        create my_species number:number_of_circle;
    }
}

species my_species {
    reflex move {
        location <-
{location.x+rnd(-speed,speed),location.y+rnd(-speed,speed)};
    }

    aspect default {
        draw circle(1) color: #yellow border: #black;
        ask my_species at_distance(distance_to_intercept) {
            draw polyline([self.location,myself.location]) color:#black;
        }
    }
}

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

```



Version: 1.9.1

Attaching Skills

GAMA allows the modeler to increase the capabilities of the GAMA agents by attaching skills to them through the facet `skills`. Skills are built-in modules that provide a set of related built-in attributes and built-in actions (in addition to those already proposed by GAMA) to the species that declare them. The [list of the available skills can be found on the dedicated page](#).

Index

- [The moving skill](#)
- [Other skills](#)
- [Example of implementation](#)

Skills

A declaration of skill is done by filling the `skills` facet in the species definition:

```
species my_species skills: [skill1,skill2] {  
}
```

A very useful and common skill is the `moving` skill.

```
species my_species skills: [moving] {  
}
```

Once your species has the moving skill, it earns automatically the following attributes: `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `follow`, `wander` and `wander_3D`.

Attributes:

- `speed` (float) designs the speed of the agent, in m/s.
- `heading` (int) designs the heading of an agent in degrees, which means that is the maximum angle the agent can turn around each step.
- `destination` (point) is the updated destination of the agent, with respect to its speed and heading. It's a read-only attribute, you can't change its value.

Actions:

`follow`

moves the agent along a given path passed in the arguments.

- returns: path
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `path` (path): a path to be followed.
- `move_weights` (map): Weights used for the moving.
- `return_path` (boolean): if true, return the path followed (by default: false)

`goto`

moves the agent towards the target passed in the arguments.

- returns: path
- `target` (agent,point,geometry): the location or entity towards which to move.

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **on** (graph): graph that restrains this move
- **recompute_path** (boolean): if false, the path is not recomputed even if the graph is modified (by default: true)
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weights used for the moving.

move

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- **bounds** (geometry,agent): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

wander

Moves the agent towards a random location at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

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

- **amplitude** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- **bounds** (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

wander_3D

Moves the agent towards a random location (3D point) at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- returns: path
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **amplitude** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- **z_max** (int): the maximum altitude (z) the geometry can reach
- **bounds** (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

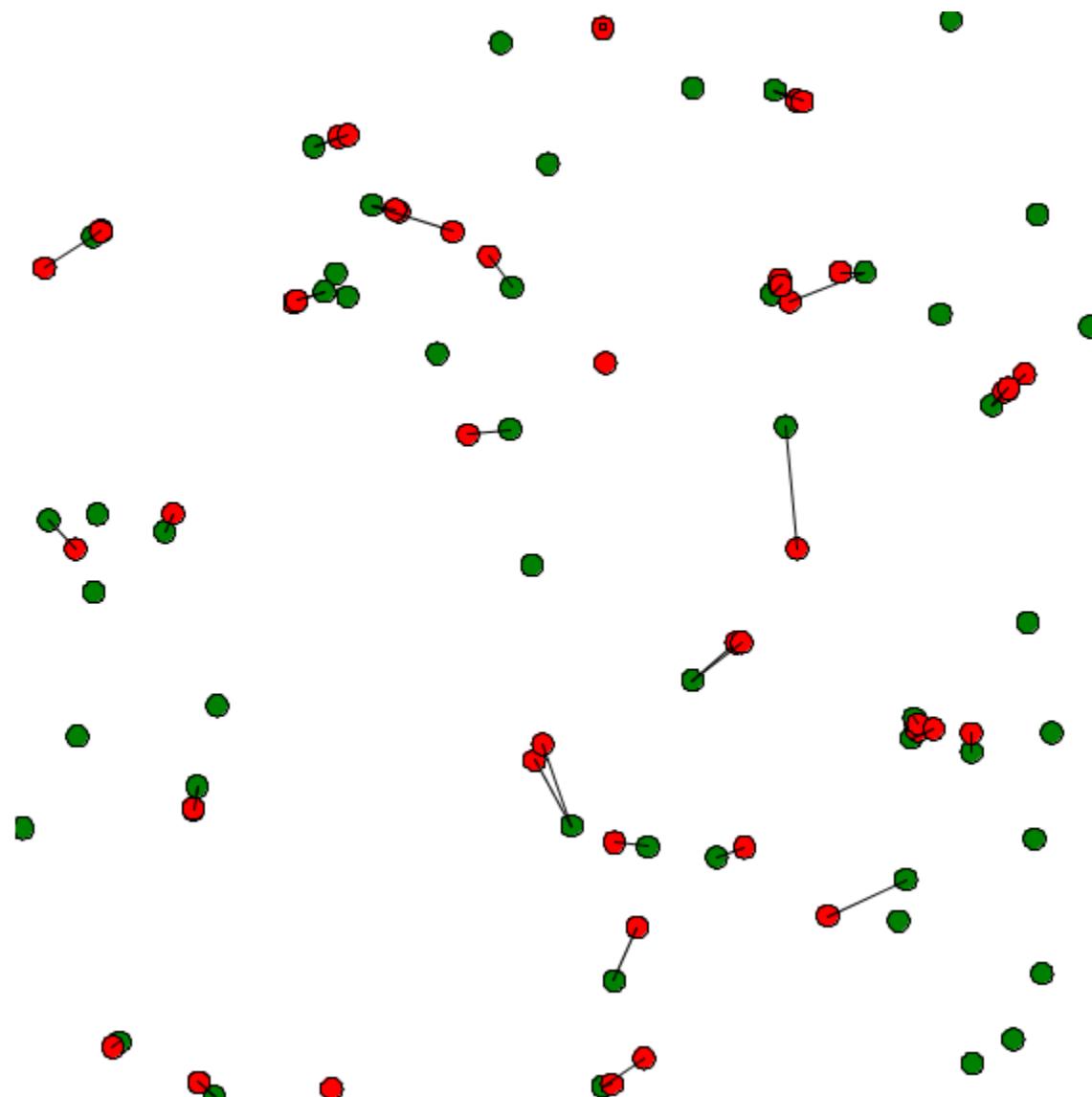
Other skills

A lot of other skills are available. Some of them can be [built in skills](#), integrated by default in GAMA, other are linked to [additional plugins](#).

This is the list of skills: `Advanced_driving`, `communication`, `driving`, `GAMASQL`, `graphic`, `grid`, `MDXSKILL`, `moving`, `moving3D`, `physical3D`, `skill_road`, `skill_road`, `skill_road_node`, `SQLSKILL`

Example

We can now build a model using the skill moving. Let's design 2 species, one is "species_red", the other is "species_green". Species_green agents are moving randomly with a certain speed and a certain heading. Species_red agents wait for a species_green agent to be in a certain range of distance. Once it is the case, the agent move toward the species_green agent. A line link the red_species agent and its target.



Here is an example of implementation:

```
model green_and_red_species

global{
    float distance_to_intercept <- 10.0;
    int number_of_green_species <- 50;
    int number_of_red_species <- 50;

    init {
        create speciesA number:number_of_green_species;
        create speciesB number:number_of_red_species;
    }
}

species speciesA skills:[moving] {
    init {
        speed <- 1.0;
    }
    reflex move {
        do wander amplitude: 90.0;
    }
    aspect default {
        draw circle(1) color:#green border: #black;
    }
}

species speciesB skills:[moving] {
    speciesA target;
    init {
        speed <- 0.0;
        heading <- 90.0;
    }
    reflex search_target when: target=nil {
        ask speciesA at_distance(distance_to_intercept) {
            myself.target <- self;
        }
    }
    reflex follow when: target!=nil {
```


Version: 1.9.1

Inheritance

As for many object-oriented programming languages, inheritance can be used in GAML. It is used to structure better your code when you have some complex models. It is, for example, useful when you have defined two different species with many attributes and behaviors in common: you can factorize everything in common in a **parent** species, and let in the **child species** only the differences between these 2 species. Notice that behaviors and actions defined in a parent species can be redefined in a children species, in order to code a difference in terms of behavior, or simply execute the action of the parent and complete it with some other statements.

Index

- Mother species / child species
- Virtual actions
- Get all the subspecies from a species

Mother species/child species

To make a species inherit from a mother species, you have to add the facet `parent`, and specify the mother species.

```
species mother_species { }

species child_species parent: mother_species { }
```

Thus, all the attributes, actions and reflex of the mother species are inherited to the child species.

```
species mother_species {  
    int attribute_A;  
    action action_A {}  
}  
  
species child_species parent: mother_species {  
    init {  
        attribute_A <- 5;  
        do action_A;  
    }  
}
```

If the mother species has a particular skill, its children will inherit all the attributes and actions.

```
species mother_species skills:[moving] {}  
  
species child_species parent:mother_species {  
    init {  
        speed <- 2.0;  
    }  
    reflex update {  
        do wander;  
    }  
}
```

You can redefine an action or a reflex by declaring an action or a reflex with the same name.

In the redefined action, it is common to call the action of the mother species with some specific parameters or to add more computations. To this purpose, you need to use:

- `invoke` instead of `do` to call an action (procedure) of the mother species.
- `super.action_name()` to call an action (function) of the mother species.

```

species animal {
    int age <- 0;

    action grow {
        age <- age + 1;
    }

    int get_age {
        return age;
    }
}

species cat parent: animal {
    action grow {
        invoke grow(); // call the action growth of the mother species
    }

    write "I am a cat and I grow up";
}

    int get_age {
        return super.get_age() * 7; // call the action get_age from the
    // mother species animal
    // 1 year is 7 year for cats
}
}

```

Virtual action

You have also the possibility to declare a virtual action in the mother species, which means an action without implementation, by using the facet `virtual`. Note that, when using `virtual` facet, the statement has to start by `action` and not a return type. If the action is expecting to return a value you need to add the `type` facet:

```
action virtual_action virtual: true;

action vistual_action_with_return_value virtual: true type: any_type;
```

When you declare an action as virtual in a species, this species becomes abstract, which means you cannot instantiate agent from it. All the children of this species have to implement this virtual action.

```
species virtual_mother_species {
    action my_action virtual:true;
}

species child_species parent: virtual_mother_species {
    action my_action {
        // some statements
    }
}
```

Get all the subspecies from a species

If you declare a `mother` species, you create a `child` agent, then `mother` will return the population of agents `mother` and **not** the population of agents `child`, as it is shown in the following example:

```
global {
    init {
        create child number: 2;
        create mother number: 1;
    }
    reflex update {
        write length(mother); // will write 1 and not 3
    }
}
```

We remind you that `subspecies` is a built-in attribute of the agent. Using this attribute, you can easily get all the subspecies agents of the mother species by writing the following GAML function:

```
global
{
    init {
        create child number: 2;
        create mother number: 1;
    }
    reflex update {
        write length(get_all_instances(mother)); // will write 3 (1+2)
    }
    list<agent> get_all_instances(species<agent> spec) {
        return spec.population + spec.subspecies accumulate
        (get_all_instances(each));
    }
}

species mother {}

species child parent: mother {}
```

The operator `of_generic_species` can also be used to filter a list of agents and get all the agents of a given species or of its children species. As a consequence, in the previous example, to count all the agents of `mother` and `child` species you can only write:

```
write length(agents of_generic_species mother);
```

Version: 1.9.1

Defining advanced species

In the previous chapter, we saw how to declare and manipulate **regular species** and the **global species** (as a reminder, the instance of the **global species** is the **world agent**).

We will now see that GAMA provides you the possibility to declare some special species, such as **grids** or **graphs**, with their own built-in attributes and their own built-in actions. We will also see how to declare **mirror species**, which is a "copy" of a regular species, in order to give it an other representation. Finally, we will learn how to represent several agents through one unique agent, with **multi-level architecture**.

Version: 1.9.1

Grid Species

A grid is a particular species of agents. Indeed, a grid is a set of agents that share a grid topology (until now, we only saw species with continuous topology). Like other agents, a grid species can have attributes, attributes, behaviors, aspects. However, contrary to regular species, **grid agents are created automatically at the beginning of the simulation**. It is thus not necessary to use the `create` statement to create them. Moreover, in addition to classic built-in variables, `grid` comes with a set of additional built-in variables.

Index

- Declaration
- Built-in attributes
- Access to a cell
- Display grid
- Grid from a matrix
- Example

Declaration

Instead of using the `species` keyword, use the keyword `grid` to declare a grid species. The grid species has exactly the same facets as the regular species, plus some others. To declare a grid, you can specify the number of columns and rows first (another possibility to declare the `grid` is detailed below when (we create a grid

from a matrix)[GridSpecies#grid-from-a-matrix]). You can do it in two different ways:

- Using the two facets `width` and `height` to fix the number of cells (the size of each cell will be determined thanks to the environment dimension).

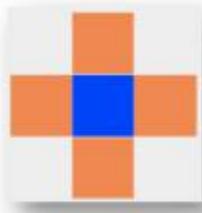
```
grid my_grid width: 8 height: 10 {  
    // my_grid has 8 columns and 10 rows  
}
```

- Using the two facets `cell_width` and `cell_height` to fix the size of each cell (the number of cells will be determined thanks to the environment dimension).

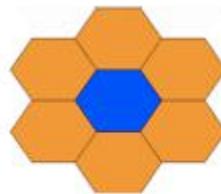
```
grid my_grid cell_width: 3 cell_height: 2 {  
    // my_grid has cells with dimension 3m width by 2m height  
}
```

By default, a grid is composed of 100 rows and 100 columns.

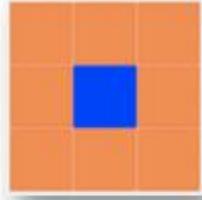
Another facet exists for grid only, very useful. It is the `neighbors` facet, used to determine how many neighbors each cell has. You can choose among 3 values: 4 (Von Neumann), 6 (hexagon) or 8 (Moore).



4 : Von Neumann



6 : Hexagon



8 : Moore

A grid can also be provided with specific facets that allow to optimize the computation time and the memory space, such as `use_regular_agents`, `use_individual_shapes` and `use_neighbors_cache`. Please refer to [the GAML](#)

Reference for more explanation about those particular facets.

Built-in attributes

grid_x

This variable stores the column index of a cell.

```
grid cell width: 10 height: 10 neighbors: 4 {  
    init {  
        write "my column index is:" + grid_x;  
    }  
}
```

grid_y

This variable stores the row index of a cell.

```
grid cell width: 10 height: 10 neighbors: 4 {  
    init {  
        write "my row index is:" + grid_y;  
    }  
}
```

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: #black ;
        }
    }
}
```

neighbors

The **neighbors** built-in variable returns the list of cells at a distance of 1. This list obviously depends on the neighbor type defined in the **grid** statement (4,6, or 8).

```
grid my_grid {
    reflex writeNeighbors {
        write neighbors;
    }
}
```

grid_value

The **grid_value** built-in variable is used when initializing a grid from grid file (see later). It contains the value stored in the data file for the associated cell. It is also used for the 3D representation of DEM.

"Missing" attribute

Information that is commonly asked a cell agent is the set of agents located inside it.

This information is not stored in the agent, but can be computed using the `inside` operator:

```
grid cell width: 10 height: 10 neighbors: 4 {  
    list<bug> bugs_inside -> {bug inside self};  
}
```

Access to a cell

There are several ways to access a specific cell:

- by a location: by casting a location variable (of type `point`) to a cell, GAMA will compute the cell that covers the given location:

```
global {  
    init {  
        write "cell at {57.5, 45} :" + cell({57.5, 45});  
    }  
}  
  
grid cell width: 10 height: 10 neighbors: 4 { }
```

- by the row and column indexes: like matrix, it is possible to directly access to a cell from its indexes

```
global {  
    init {  
        write "cell [5,8] :" + cell[5, 8];  
    }  
}  
  
grid cell width: 10 height: 10 neighbors: 4 { }
```

- The operator `grid_at` also exists to get a particular cell. You just have to specify the index of the cell you want (in x and y):

```
global {
    init {
        agent cellAgent <- cell grid_at {5, 8};
        write "cell [5,8] :" + cellAgent;
    }
}

grid cell width: 10 height: 10 neighbors: 4 { }
```

Display Grid

You can easily display your grid in your experiment as followed:

```
experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            grid MyGrid;
        }
    }
}
```

The grid will be displayed, using the color you defined for each cell (with the `color` built-in attribute). You can also show the border of each cell by using the facet `lines` and choosing a color:

```
display MyDisplay type: opengl {
    grid MyGrid lines: #black;
}
```

Another way to display a grid will be to define an aspect in your grid agent (the same way as for a [regular species](#)), and add your grid as a regular species in the display of in your experiment and thus by specifying its aspect:

```
grid MyGrid {
    aspect firstAspect {
        draw square(1);
    }
    aspect secondAspect {
        draw circle(1);
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            species MyGrid aspect: firstAspect;
        }
    }
}
```

Beware: do not use this second display when you have large grids: it is much slower.

Grid from a matrix

An easy way to load some values in a grid is to use matrix data. A `matrix` is a type of container (we invite you to learn some more about this useful type [here](#)). Once you have declared your matrix, you can set the values of your cells using the `ask` statement :

```
global {
    init {
        matrix data <- matrix([[0,1,1],[1,2,0]]);
```

Declaring larger matrix in GAML can be boring as you can imagine. You can load your matrix directly from a csv file with the operator `matrix` (used for the construction of the matrix).

```
file my_file <- csv_file("path/file.csv", "separator");
matrix my_matrix <- matrix(my_file);
```

You can try to read the following csv :

```
0,0,0,0,0,0,0,0,0,0,0
0,0,0,1,1,1,1,1,0,0,0
0,0,1,1,0,0,0,1,1,0,0
0,1,1,0,0,0,0,0,0,0,0
0,1,1,0,0,1,1,1,1,0,0
0,0,1,1,0,0,1,1,1,0,0
0,0,0,1,1,1,1,0,1,0,0
0,0,0,0,0,0,0,0,0,0,0
```

With the following model:

```
model import_csv

global {
    file my_csv_file <- csv_file("../includes/test.csv", ", ");
    init {
        matrix data <- matrix(my_csv_file);
        ask my_gama_grid {
            grid_value <- float(data[grid_x,grid_y]);
            write data[grid_x,grid_y];
        }
    }
}

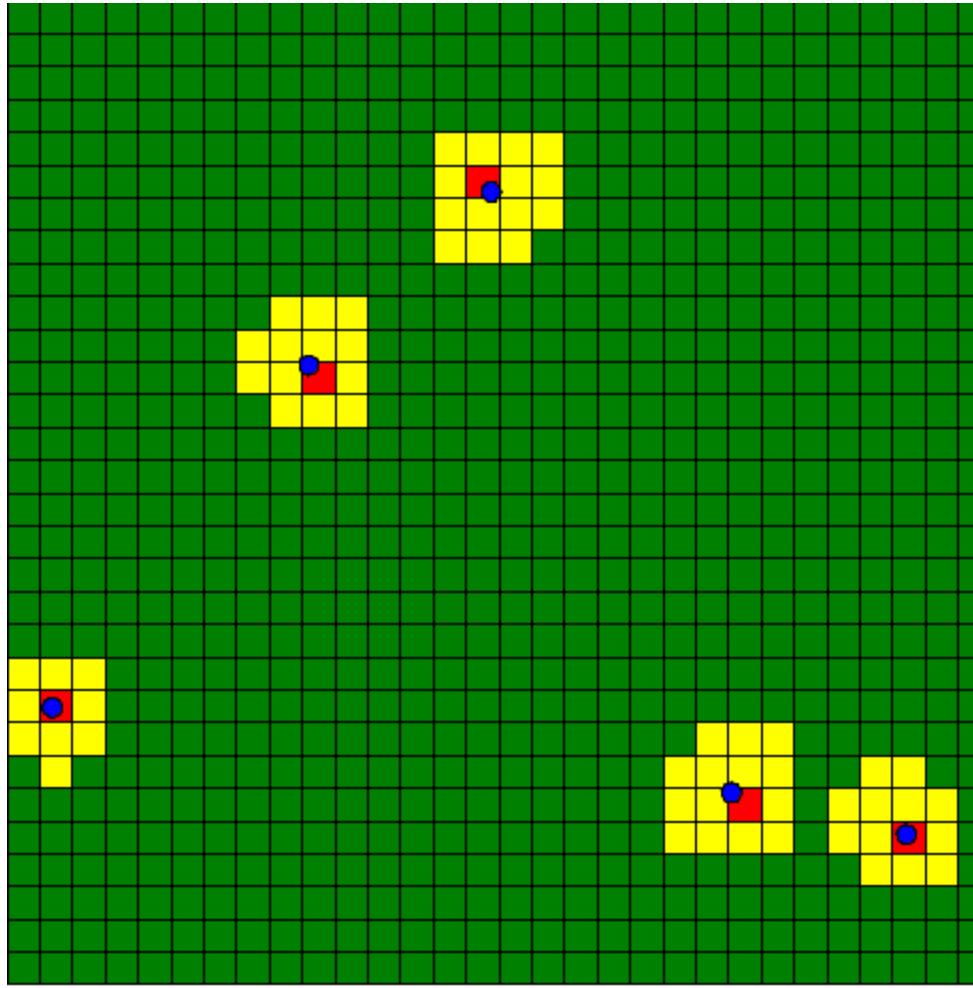
grid my_gama_grid width: 11 height: 8 {
```

For more complicated models, you can read some other files, such as ASCII files (asc), DEM files. In this case, the creation of the grid is even easier as the dimensions of the grid can be read from the file with the `file` facet:

```
grid my_grid from: my_asc_file {  
}
```

Example

To practice a bit those notions, we will build a quick model. A "regular" species will move randomly on the environment. A grid is displayed, and its cells becomes red when an instance of the regular species is waking inside this cell, and yellow when the regular agent is in the surrounding of this cell. If no regular agent is on the surrounding, the cell turns green.



Here is an example of implementation:

```
model my_grid_model

global{
    float max_range <- 5.0;
    int number_of_agents <- 5;
    init {
        create my_species number: number_of_agents;
    }

    reflex update {
        ask my_species {
            do wander amplitude: 180.0;
        }
    }
}
```




Version: 1.9.1

Graph Species

Using a graph in a model enables to easily show and manage interactions between agents: this can be the link between 2 points in space linked by a road or (non-spatial) interactions that exist between friends agents. It can also be a powerful tool to compute the shortest path between 2 points in space.

A graph is a concept that has several implementations in GAML: (i) it can be a datatype (`graph`), that can be created from a road shapefile (e.g. using `as_edge_graph`), (ii) it can be implemented as node and edge species, and (iii) it is a `topology`, i.e. an organisation of agents that influences the way distance and neighborhood is computed.

Index

- Declaration
 - Declare a graph with handmade agents
 - Declare a graph by using an geometry file
 - Declare a graph with nodes and edges
- Useful operators with graph
 - Knowing the degree of a node
 - Get the neighbors of a node
 - Compute the shortest path
 - Control the weight in graph
- Example

Declaration

Declare a graph with handmade agents

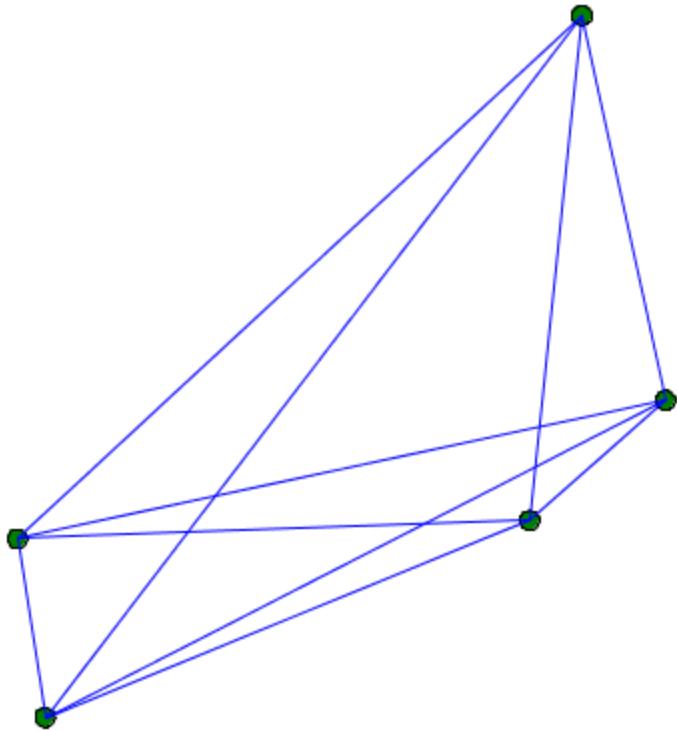
To instantiate this `graph`, we rely on the two built-in species `graph_node` and `base_edge`. We need to define our own node and edge species as children species of the 2 built-in species. First, the node species must inherit from the abstract species `graph_node`, then the method `related_to` must be redefined and finally an auxiliary species that inherits from `base_edge` used to represent the edges of the generated graph must be declared. A graph node is an abstract species that must redefine one method called `related_to`.

```
species my_node parent: graph_node edge_species: edge_agent{
    bool related_to(my_node other){
        return true;
    }
}

species edge_agent parent: base_edge {
```

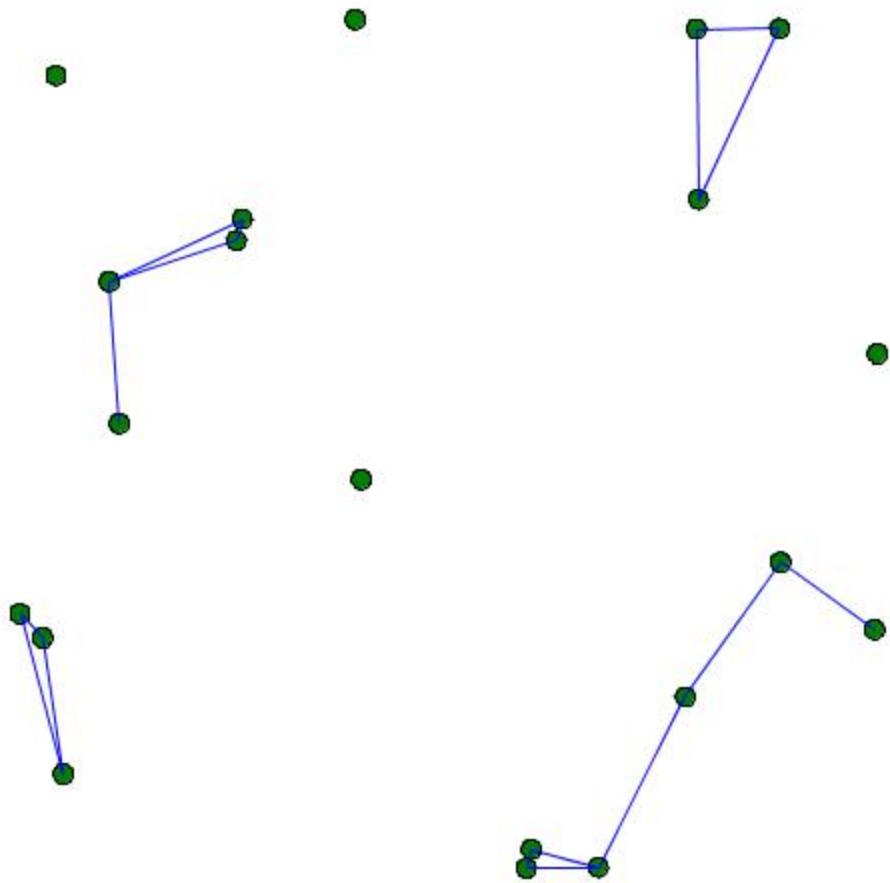
The method `related_to` returns a boolean and takes the agents from the current species as argument. If the method returns true, the two agents (the current instance and the one as argument) will be linked. The method is automatically called for each agent with each other agent of the given species in argument. Note that in the following example, `related_to` returns always true, so each agent will be linked to each other agent: we will get a complete graph.

```
model NewModel
```



You can, for example, link 2 agents when they are closer than a certain distance. Beware: The topology used in graph species is the graph topology and not the continuous topology. You can force the use of the continuous topology with the action `using` as follow:

```
bool related_to(my_node other){  
    using topology(world) {  
        return (self.location.distance_to other.location < 20);  
    }  
}
```



The abstract mother species `graph_node` has an attribute `my_graph`, with the type `graph`. The `graph` type represents a graph composed of vertices linked with edges. This type has built-in attributes such as `edges` (the list of all the edges agents), or `vertices` (the list of all the vertices agents).

Declare a graph by using a geometry file

In most cases, you will have to construct a graph from an existing file (example: a "shp" file). In that case, you will have to first instantiate a species from the shape file (with the `create` statement, using the facet `from`). Then, you will have to construct a graph from the agent, using one of the available operators such as `as_edge_graph`.

```

model load_shape_file

global {
    file roads_shapefile <- file("../includes/road.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;

    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

experiment main_experiment type:gui{
    output {
        display map {
            species road aspect:geom;
        }
    }
}

```

Declare a graph with nodes and edges

Another way to create a graph is building it manually nodes by nodes, and then edges by edges, without using agent structures. Use the `add_node` operator and the `add_edge` operator to do so. Here is an example of how to do:

```

list<point> nodes <- [];
graph my_graph <- graph([]);

```

Using this solution, my_graph can have two types: it can be an a-spatial graph, or a spatial graph. The spatial graph will have a proper geometry, with segments that follow the position of your graph (you can access to the segments by using the built-in "segments"). The a-spatial graph will not have any shape.

```
global {
    graph my_spatial_graph<-spatial_graph([]);
    graph my_aspatial_graph<-graph([]);

    init {
        point node1 <- {0.0,0.0};
        point node2 <- {10.0,10.0};

        my_spatial_graph <- my_spatial_graph add_node(node1);
        my_spatial_graph <- my_spatial_graph add_node(node2);
        my_spatial_graph <- my_spatial_graph add_edge(node1::node2);
        write my_spatial_graph.edges;

        // the output is [polyline ([{0.0,0.0,0.0},{10.0,10.0,0.0}])]
        my_aspatial_graph <- my_aspatial_graph add_node(node1);
        my_aspatial_graph <- my_aspatial_graph add_node(node2);
        my_aspatial_graph <- my_aspatial_graph add_edge(node1::node2);
        write my_aspatial_graph.edges;
        // the output is [{0.0,0.0,0.0}::{10.0,10.0,0.0}]
    }
}
```

Useful operators with graph

Knowing the degree of a node

The operator `degree_of` returns the number of edges attached to a node. To use it, you have to specify a graph (on the left side of the operator), and a node (on the right side of the operator).

The following code (to put inside the node species) displays the number of edges attached to each node:

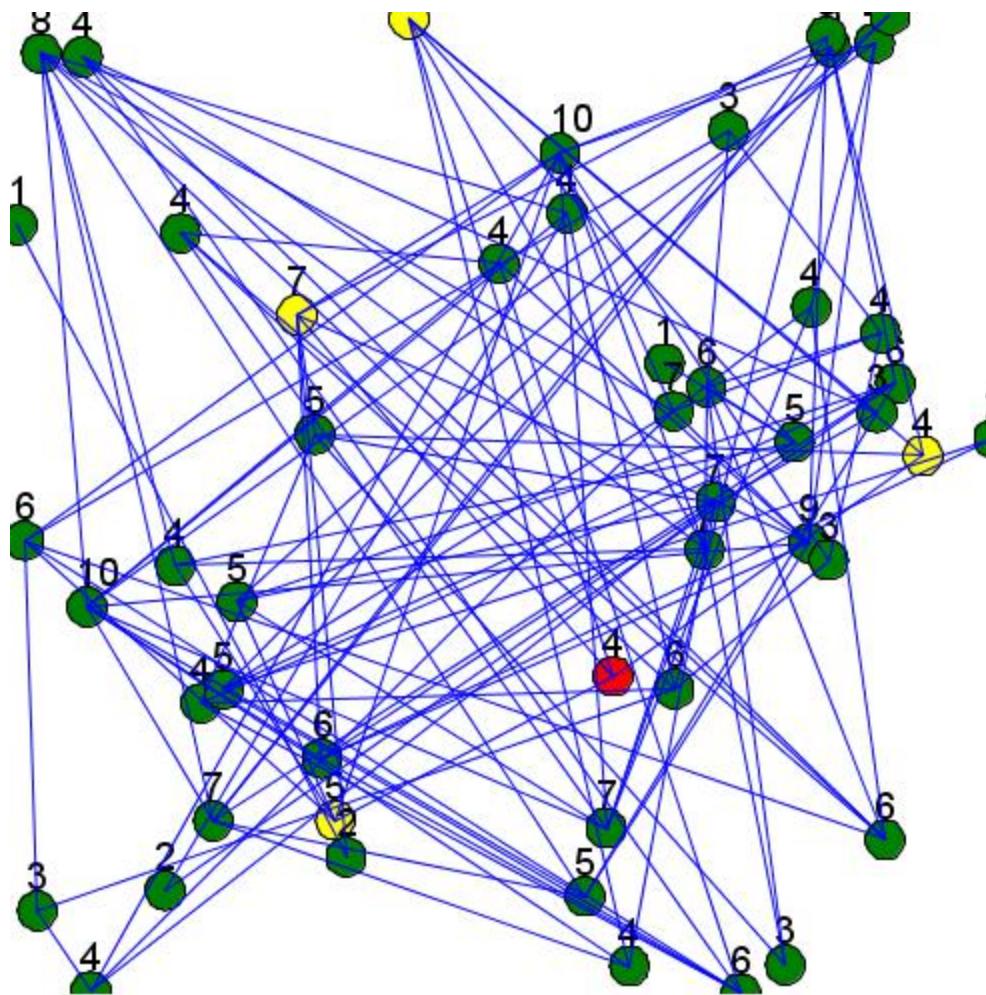
```
aspect base {
    draw string(my_graph degree_of node(5)) color:# black;
}
```

Get the neighbors of a node

To get the list of neighbors of a node, you should use the neighbors_of operator. On the left side of the operator, specify the graph you are using, and on the right side, specify the node. The operator returns the list of nodes located at a distance inferior or equal to 1, considering the graph topology.

```
species graph_agent parent: graph_node edge_species: edge_agent
{
    list<graph_agent> list_neighbors <- list<graph_agent>(my_graph
neighbors_of (self));
}
```

Here is an example of a model using those two previous concepts (a random node is chosen each step, displayed in red, and his neighbors are displayed in yellow):



```
model graph_model

global {
    int number_of_agents <- 50;

    init {
        create my_node number: number_of_agents;
    }

    reflex update {
        ask one_of(my_node) {
            status <- 2;
            do update_neighbors;
        }
    }
}
```

Compute the shortest path

To compute the shortest path to go from a point to another one, pick a source and a destination among the vertices you have for your graph. Store those values as point type.

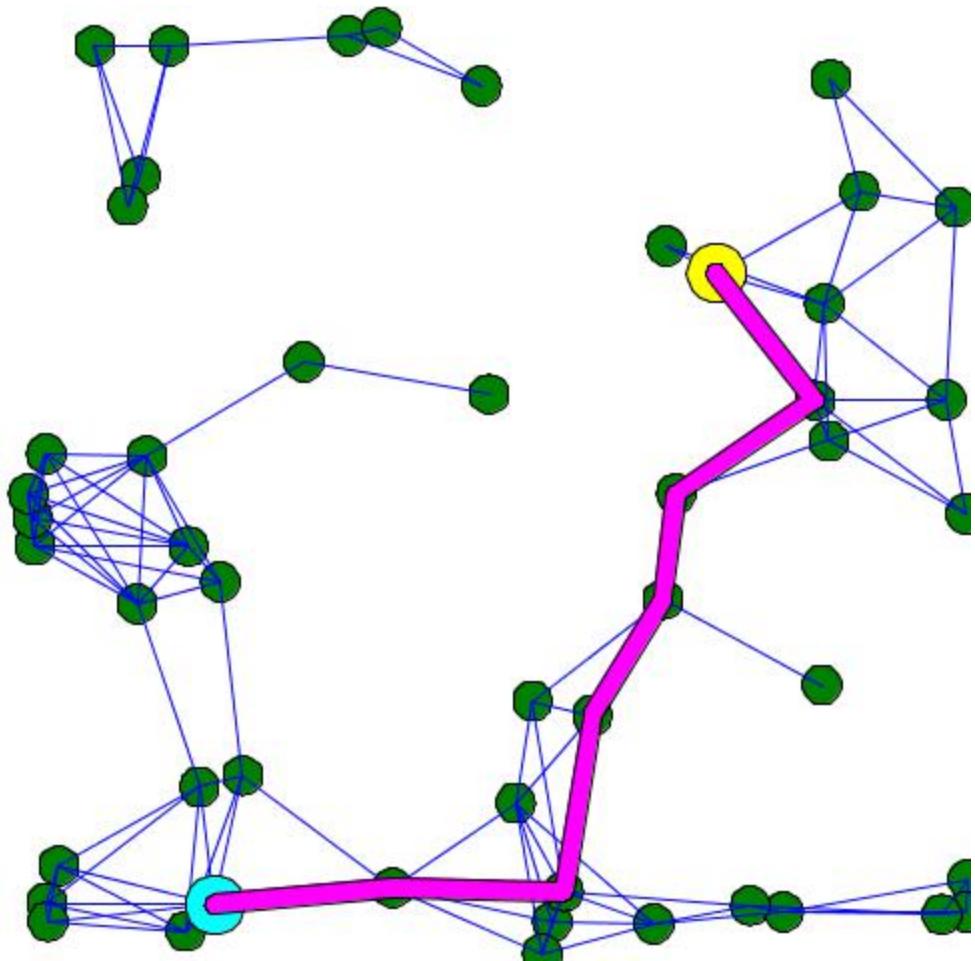
```
point source;
point destination;
source <- point(one_of(my_graph.vertices));
destination <- point(one_of(my_graph.vertices));
```

Then, you can use the operator `path_between` to return the shortest path. To use this operator, you have to give the graph, then the source point, and the destination point. This operator returns a [path type object](#).

```
path shortest_path;
shortest_path <- path_between (my_graph, source, destination);
```

Another operator exists, `paths_between`, that returns a list of shortest paths between two points. Please read the documentation to learn more about this operator.

Here is an example of code that shows the shortest path between two points of a graph:



```
model graph_model

global {
    int number_of_agents <- 50;
    point source;
    point target;
    graph the_graph;
    path shortest_path;

    init {
        create my_node number: number_of_agents;
    }

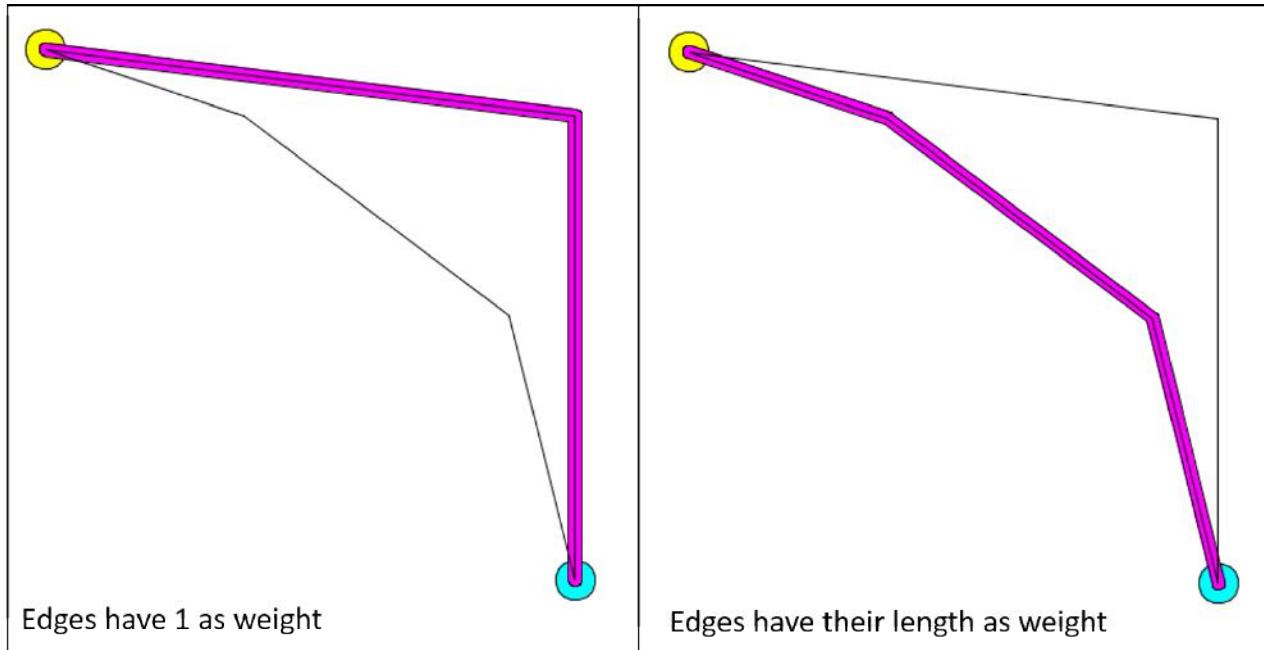
    reflex pick_two_points {
```

Control the weight in graph

You can add a map of weight for the edges that compose the graph. Use the operator `with_weights` to put weights in your graph. The graph has to be on the left side of the operator, and the map has to be on the right side. In the map, you have to put edges as key, and the weight for that edge as value. One common use is to put the distance as weight:

```
my_graph <- my_graph with_weights (my_graph.edges as_map  
(each::geometry(each).perimeter));
```

The calculation of the shortest path can change according to the weight you choose for your edges. For example, here is the result of the calculation of the shortest path when all the edges have 1 as weight value (it is the default graph topology), and when the edges have their length as weight.



Here is an example of implementation:

```

model shortest_path_with_weight

global {
    graph my_graph<-spatial_graph([]);
    path shortest_path;
    list<point> nodes;

    init {
        add point(10.0,10.0) to:nodes;
        add point(90.0,90.0) to:nodes;
        add point(40.0,20.0) to:nodes;
        add point(80.0,50.0) to:nodes;
        add point(90.0,20.0) to:nodes;

        loop nod over: nodes {
            my_graph <- my_graph add_node(nod);
        }
    }

    my_graph <- my_graph add_edge (nodes at 0::nodes at 2);
    my_graph <- my_graph add_edge (nodes at 2::nodes at 3);
    my_graph <- my_graph add_edge (nodes at 3::nodes at 1);
    my_graph <- my_graph add_edge (nodes at 0::nodes at 4);
    my_graph <- my_graph add_edge (nodes at 4::nodes at 1);

    // comment/decomment the following line to see the difference.
    my_graph <- my_graph with_weights (my_graph.edges as_map
(each)::geometry(each).perimeter));

    shortest_path <- path_between(my_graph,nodes at 0, nodes at 1);
}
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            graphics "shortest path" {
                if (shortest_path != nil) {
                    draw circle(3) at: point(shortest_path.source) color:
#yellow;

```




Version: 1.9.1

Mirror species

A mirror species is a species whose population is automatically managed with respect to another species. Whenever an agent is created or destroyed from the other species, an instance of the mirror species is created or destroyed. Each of these 'mirror agents' has access to its reference agent (called its target). Mirror species can be used in different situations but the one we describe here is more oriented towards visualization purposes.

Index

- Declaration
- Example

Declaration

A mirror species can be defined using the mirrors facet 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, one wants to place the mirror agent at the same location as the reference species. This can be achieved by simply adding the following lines in the mirror species:

```
species B mirrors: A{
    point location <- target.location update: target.location;
}
```

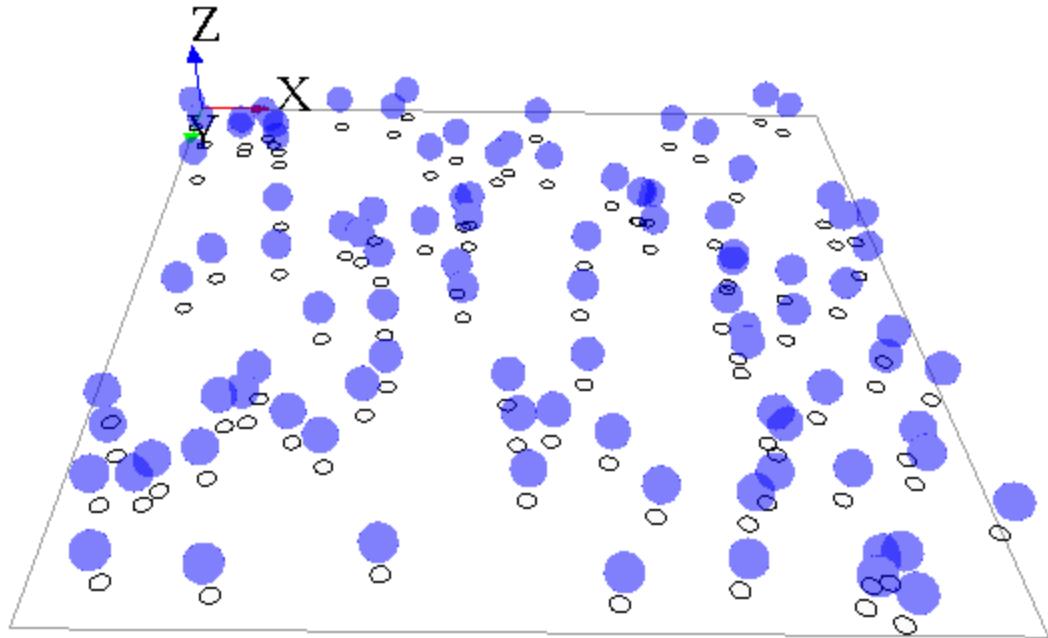
target is a built-in attribute of a mirror species. It refers to the instance of the species tracked.

In the same spirit, any attribute of a reference species can be reached using the same syntax. For instance, if the species A has an attribute called **attribute1** of type **int** it is possible to get this attribute from the mirror species B using the following syntax:

```
int value <- target.attribute1;
```

Example

To practice a bit with the mirror notion, we will now build a simple model displaying a species A (aspect: white circle) moving randomly, and another species B (aspect: blue sphere) with the species A location on x and y, with an upper value for the z-axis.



Here is an example of an implementation for this model:

```
model Mirror

global {
    init{
        create A number:100;
    }
}

species A skills:[moving]{
    reflex update{
        do wander;
    }
    aspect base{
        draw circle(1) color: #white border: #black;
    }
}
species B mirrors: A{
```




Version: 1.9.1

Multi-level architecture

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

Index

- Declaration of micro-species
- Access to micro-agents / host agent
- Representation of an entity as different types of agent
- Dynamic migration of agents
- Example

Declaration of micro-species

A species can have other species as **micro-species**. The micro-species of a species is declared inside the species' declaration.

```
species macro_species {  
    species micro_species_in_group {  
    }  
}
```

In the above example, `micro_species_in_group` is a micro-species of `macro_species`. An agent of `macro_species` can have agents `micro_species_in_group` as micro-agents. Agents of `micro_species_in_group` have an agent of `macro_species` as `host` agent.

As the species `micro_species_in_group` is declared inside the species `macro_species`, `micro_species_in_group` will return a list of `micro_species_in_group` agent inside the given `macro_species` agent.

```
global {
    init {
        create macro_species number:5;
    }
}

species macro_species  {
    init {
        create micro_species_in_group number: rnd(10);
        write "the macro species agent named "+name+" contains
"+length(micro_species_in_group)+" agents of micro-species.";
    }
}

species micro_species_in_group { }

experiment my_experiment type: gui { }
```

In the above example, we create 5 macro-species agents, and each one creates a random number of inner micro-species agents. We can see that `micro_species_in_group` refers to the list of micro-species agents inside the given macro-species agent.

Access to micro-agents, host agent

To access micro-agents (from a macro-agent), and to host agent (from a micro-agents), you have to use two built-in attributes.

The `members` built-in attribute is used inside the macro-agent, to get the list of all its micro-agents.

```
species macro_species {  
    init {  
        create first_micro_species number: 3;  
        create second_micro_species number: 6;  
        write "the macro-agent named "+name+" contains "+length(members)+"  
micro-agents."  
    }  
  
    species first_micro_species { }  
  
    species second_micro_species { }  
}
```

The `host` built-in attribute is used inside a micro-agent to get its host macro-agent.

```
species macro_species {  
  
    micro_species_in_group micro_agent;  
  
    init {  
        create micro_species_in_group number: rnd(10);  
        write "the macro-agent named "+name+" contains "+length(members)+"  
micro-agents."  
    }  
}
```

NB: We already said that the `world` agent is a particular agent, instantiated just once. In fact, the world agent is the host of all the agents. You can try to get the host for a regular species agent, you will get the `world` agent itself (named as you named your model). You can also try to get the members of your `world` (from the global scope for example), and you will get the list of the agents presents in the world.

```
global {
    init {
        create macro_species number:5;
        write "the world has "+length(members)+" members.";
    }
}

species macro_species {
    init {
        write "the macro agent named "+name+" is hosted by "+host;
    }
}
```

Representation of an entity as different types of agent

The multi-level architecture is often used in order to represent an entity through different types of agent. For example, an agent "bee" can have a behavior when it is alone, but when the agent is near from a lot of agents, he can changes his type to "bee_in_swarm", defined as a micro-species agent of a macro-species "swarm" agent. Another example: an agent "pedestrian" can have a certain behavior when walking on the street, and then change his type to "pedestrian_in_building" when he is in a macro-agent "building".

You have then to distinguish two different species to define your micro-species:

- The first can be seen as a regular species (it is the "bee" or the "pedestrian" for instance). We will name this species as "micro_species".
- The second is the real micro-species, defined inside the macro-species (it is the "bee_in_swarm" or the "pedestrian_in_building" for instance). We will name this species as "micro_species_in_group". This species has to inherit from the "micro_species" in order to allow migrations between `micro_species_in_group` and `micro_species`.

```
species micro_species { }

species macro_species {
    species micro_species_in_group parent: micro_species { }
}
```

Dynamic migration of agents

In our example about bees, a "swarm" entity is composed of nearby flying "bee" entities. When a "bee" entity approaches a "swarm" entity, this "bee" entity will become a member of the group. To represent this, the modeler lets the "bee" agent change its species to "bee_in_swarm" species. The "bee" agent hence becomes a "bee_in_swarm" agent. To change species of an agent, we can use one of the following statements: `capture`, `release`, `migrate`.

The statement `capture` is used by the "macro_species" to capture one (or several) "micro_species" agent(s), and turn it (them) to a "micro_species_in_group". You can specify which agent (or list of agents) you want to capture by passing them as the first argument of the statement `capture`. The facet `as` is used to cast the agent(s) from "micro_species" to the species "micro_species_in_group". You can use the facet `returns` to get the newly captured agent(s).

```
capture agents_of_micro_species as: micro_species_in_group;
```

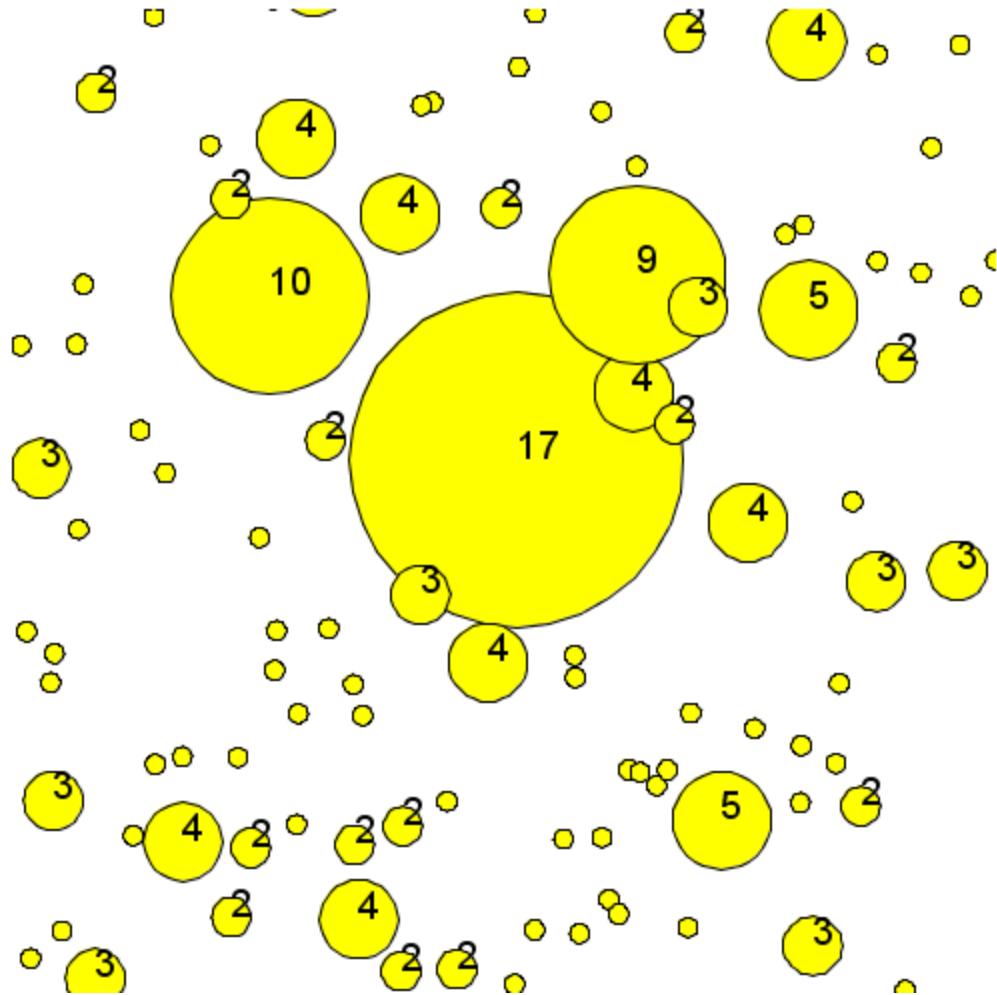
The statement `release` is used by a "macro_species" agent to release one (or several) "micro_species_in_group" agent(s), and turn it (them) to a "micro_species". You can specify which agent (or list of agents) you want to release by giving them as the first argument of the `release` statement. The facet `as` is used to cast the agents from "micro_species_in_group" species to "micro_species" species. The facet `in` is used to specify the new host (by default, it is the host of the "macro_species"). You can use the facet `returns` to get the newly released agent(s).

```
release agents_of_micro_species_in_group as: micro_species in: world;
```

The statement `migrate`, less used, permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Read the GAML Reference to learn more about this statement.

Example:

Here is an example of micro_species that gather together in macro_species when they are close enough.



```
model multilevel

global {
    int release_time <- 20;
    int capture_time <- 100;
    int remaining_release_time <- 0;
    int remaining_capture_time <- capture_time;
    init {
        create micro_species number:200;
    }
    reflex reflex_timer {
        if (remaining_release_time=1) {
            remaining_release_time <- 0;
            remaining_capture_time <- capture_time;
        }
    }
}
```


Version: 1.9.1

Defining GUI Experiment

When you execute your simulation, you will often need to display some information. For each simulation, you can define some inputs, outputs and behaviors:

- The inputs will be composed of parameters manipulated by the user for each simulation.
- The behaviors will be used to define behavior executed at each step of the **experiment**.
- The outputs will be composed of displays, monitors and inspectors. They will be defined inside the scope `output`. The definition of their layout can also be set with the `layout` statement.

A typical GUI experiment code follows this pattern:

```
experiment exp_name type: gui {  
    [input]  
    [behaviors]  
    output {  
        layout [layout_option]  
        [display statements]  
        [monitor statements]  
    }  
}
```

Types of experiments

You can define four types of experiments (through the facet `type`):

- **gui** experiments (the default type) are used to play an experiment and displays its outputs. It is also used when the user wants to interact with the simulation.
- **batch** experiments are used to play an experiment several times (usually with other input values), used for model exploration. We will [come back to this notion a bit further in the tutorial](#).
- **test** experiments are used to [write unit tests](#) on a model (used to ensure its quality).
- **memorize** experiments are GUI experiments in which [the simulation state is kept in memory and the user can backtrack to any previous step](#).

Experiment attributes

Inside experiment scope, you can access to some built-in attributes which can be useful, such as `minimum_cycle_duration`, to force the duration of one cycle.

```
experiment my_experiment type: gui {
    float minimum_cycle_duration <- 2.0#minute;
}
```

In addition, the attribute `simulations` contain the list of all the simulation agents that are running in the current experiment. Whereas the attribute `simulation` represents a single simulation, the last element of the simulation list.

Experiment facets

Finally, in the case of a GUI experiment, the facets `autorun` and `benchmark` can be used as follows:

```
experiment name type: gui autorun: true benchmark: true { }
```

When `autorun` is set to `true` the launch of the experiment will be followed automatically by its run. When `benchmark` is set to true, GAMA records the number of invocations and running time of the statements and operators of the simulations launched in this experiment. The results are automatically saved in a csv file in a folder called 'benchmarks' when the experiment is closed.

Other built-ins are available, to learn more about, go to the page [experiment built-in](#).

Defining displays layout

A `layout` can be added to `output` to specify the layout of the various displays defined below (e.g. `#none`, `#split`, `#stack`, `#vertical` or `#horizontal`). It will also define which elements of the interface are displayed: `parameters`, `navigator`, `editors`, `consoles`, `toolbars`, `tray`, or `tabs` facets (expecting a boolean value). You will find more detailed information in the [statement's documentation](#)

Defining elements of the GUI experiment

In this part, we will focus on the **gui experiments**. We will start with learning how to **define input parameters**, then we will study the outputs, such as **displays**, **monitors and inspectors**, and **export files**. We will finish this part with how to define **user commands**.



Version: 1.9.1

Defining Parameters

When playing a simulation, you have the possibility to define input parameters, in order to change them and replay the simulation. Defining parameters allows to make the value of a global variable definable by the user through the user graphic interface.

Index

- [Defining parameters](#)
- [Additional facets](#)

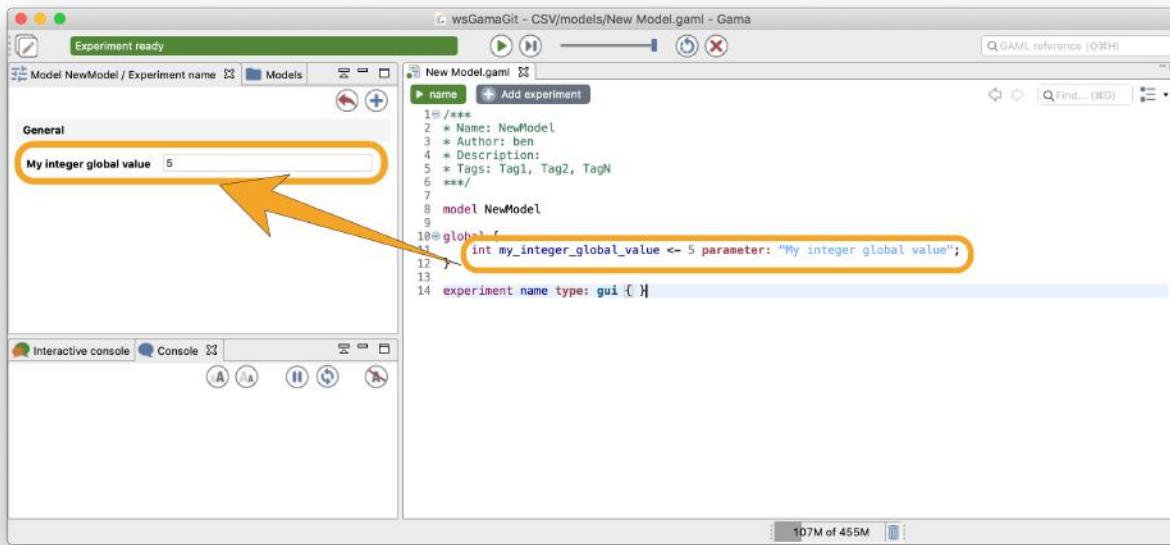
Defining parameters

You can define parameters inside the global scope when defining your global variables with the facet `parameter` (**this way of defining parameters is not the recommended one**, as it makes the variable a parameter of all the experiments that will be defined and does not offer the possibility to redefine its initial and possible values in several ways in each experiment):

```
global {
    int my_integer_global_value <- 5 parameter: "My integer global
value";
}
```

When launching your experiment, the parameter will appear in your "Parameters"

panel, with the name you chose for the `parameter` facet.



You can also define your parameter inside the experiment (**recommended**), using the statement `parameter`. You have to specify first the name of your parameter, then the name of the global variable through the facet `var`.

```
global {
    int my_integer_global_value <- 5;
}

experiment MyExperiment type: gui {
    parameter "My integer global value" var:my_integer_global_value;
}
```

NB: This variable has to be initialized with a value. If you do not want to initialize your value in the `global` block, you can initialize the value directly in the `parameter` statement, using the facet `init` or `<-`.

```
global {
    int my_integer_global_value;
}

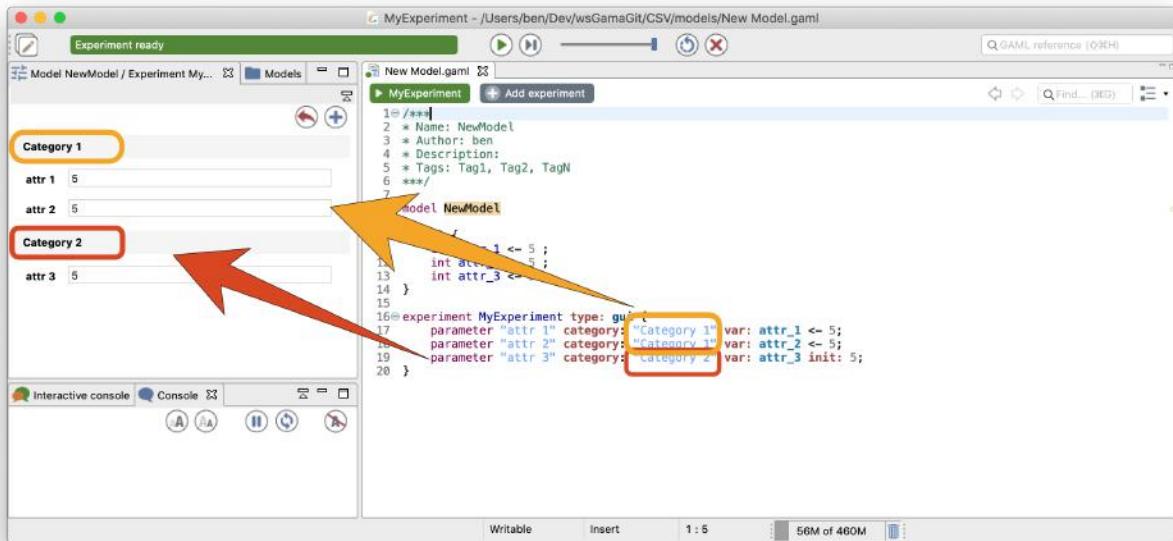
experiment MyExperiment type: gui {
    parameter "My integer global value" var: my_integer_global_value
    init: 5;
}
```

Additional facets

You can use some facets to arrange your parameters. For example, you can categorize your parameters under a label, using the facet `category`:

```
global {
    int attr_1 <- 5 ;
    int attr_2 <- 5 ;
    int attr_3 <- 5 ;
}

experiment MyExperiment type: gui {
    parameter "attr 1" category: "Category 1" var: attr_1 <- 5;
    parameter "attr 2" category: "Category 1" var: attr_2 <- 5;
    parameter "attr 3" category: "Category 2" var: attr_3 init: 5;
}
```



You also can add some facets such as `min`, `max`, `step` or `among` to improve the declaration of the parameter (and define the possible values the parameter can take).

```

global {
    string fruit <- "none" ;
    string vegetable <- "none";
    int integer_variable <- 5;
}

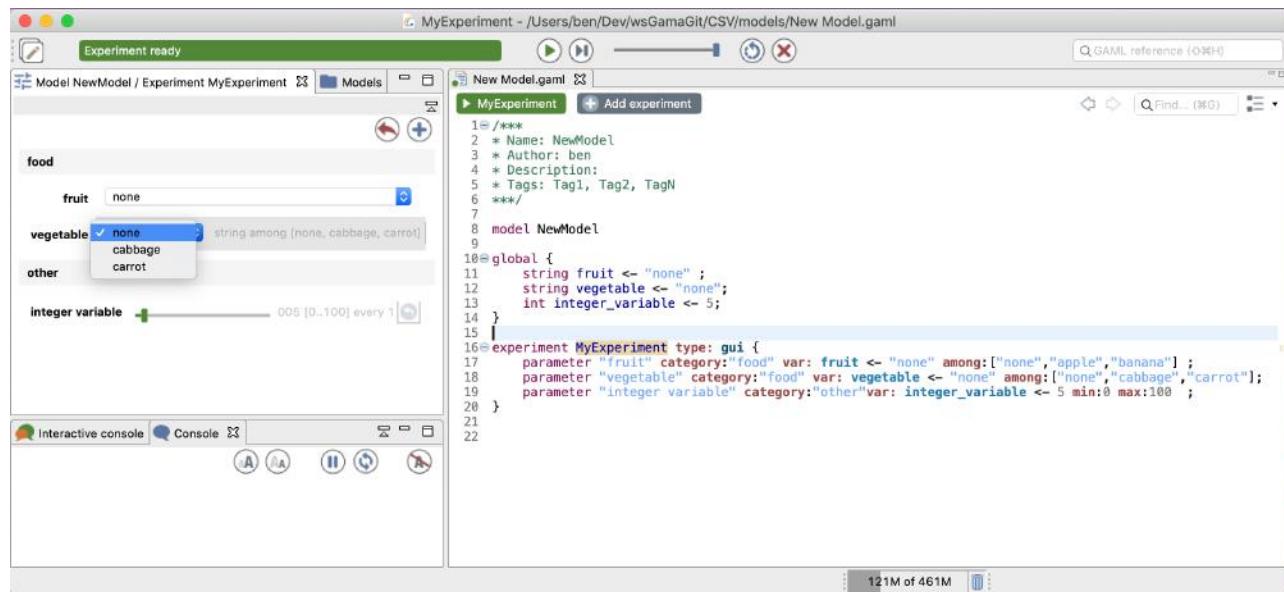
experiment MyExperiment type: gui {
    parameter "fruit" category:"food" var: fruit <- "none"
among:["none", "apple", "banana"] ;
    parameter "vegetable" category:"food" var: vegetable <- "none"
among:["none", "cabbage", "carrot"];
    parameter "integer variable" category:"other"var:
integer_variable <- 5 min:0 max:100 step:5;
}

```

We can notice that the parameters will not appear graphically in the same way if they

are defined with a set of possible values (with `among`) or with a range of possible values (defined by a `min`, `max` and a `step`).

The definition of the initial value and of the possible values can be set in the `global` or in the `experiment` depending on the aim of this limitation: for example if a variable has a maximum value set to 1 in the global, this limitation can be used in the model in order that the variable value does not exceed this value. If the maximum boundary is set in the experiment, some executions of the model can be done without it...



Version: 1.9.1

Defining displays (Generalities)

Index

- [Displays and layers](#)
- [Organize your layers](#)
- [Example of layers](#)
 - [species layer](#)
 - [grid layer](#)
 - [agents layer](#)
 - [image layer](#)
 - [graphics layer](#)

Displays and layers

A display is one of the graphical outputs of your simulation. You can define several displays related to what you want to represent from your model execution. To define a display, use the keyword `display` inside the `output` scope, and specify a name (`name` facet).

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

Other facets are available when defining your display:

- Use `background` to define a color for your background:

```
display "my_display" background: #red
```

- Use `refresh` if you want to refresh the display when a condition is true (to refresh your display every number of steps, use the operator `every`)

```
display "my_display" refresh:every(10)
```

- You can choose between two types of displays, by using the facet `type`:

- `java2D` displays will be used when you want to have 2D visualization. It is used for example when you manipulate charts. This is the default value for the facet type.
- `opengl` displays allows you to have 3D visualization.

You can save the display on the disk, as a png file, in the folder `name_of_model/models/snapshots`, by using the facet `autosave`. This facet takes a boolean as argument (to allow or not to save each frame) or a point to define the size of your image (note that when no unit is provided, the unit is `#px` (pixel)).

```
display my_display autosave: true type: java2D {}
```

The complete list of the display's facets are [available in the documentation](#)

Each display can be decomposed in one or several layers. Most of the time, all the layers are superimposed and cover all the environment space. In a 3D (OpenGL) display the layers can be split in order to be visualized separately (cf [the page about displays](#)).

Organize your layers

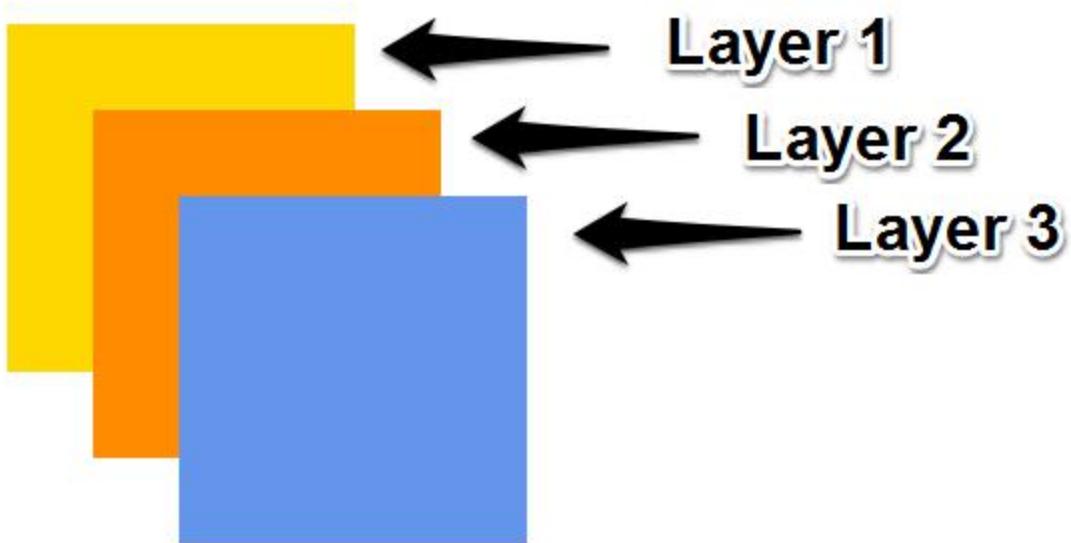
In one 2D display, you will have several types of layers, giving what you want to display in your model. You have a large number of layers available. You already know some of them, such as `species`, `agents`, `grid`, but other specific layers such as `image` (to display image) and `graphics` (to freely draw shapes/geometries/texts without having to define a species) are also available

Each layer will be displayed in the same order as you declare them. The last declared layer will be above the others. As a consequence, any layer can hide elements of the lower levels.

Thus, the following code:

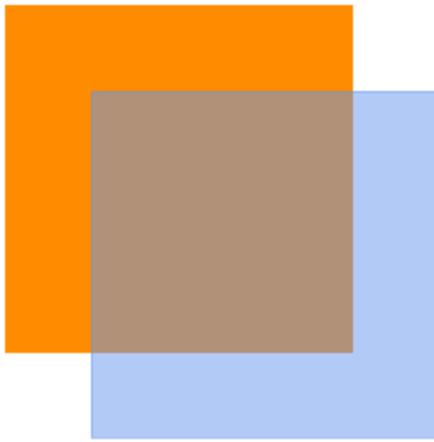
```
experiment expe type: gui {
    output {
        display my_display {
            graphics "layer1" {
                draw square(20) at: {10,10} color: #gold;
            }
            graphics "layer2" {
                draw square(20) at: {15,15} color: #darkorange;
            }
            graphics "layer3" {
                draw square(20) at: {20,20} color: #cornflowerblue;
            }
        }
    }
}
```

Will have this output:



Most of the layers have a `transparency` facet that you can use in order to see the layers which are under.

```
experiment expe type:gui {  
    output {  
        display my_display {  
            graphics "layer1" {  
                draw square(20) at:{10,10} color:#darkorange;  
            }  
            graphics "layer2" transparency:0.5 {  
                draw square(20) at:{15,15} color:#cornflowerblue;  
            }  
        }  
    }  
}
```



To specify a position and a size for your layer, you can use the `position` and the `size` facets.

The `position` facet is used with a point type, between `{0, 0}` and `{1, 1}`, which corresponds to the position of the upper left corner of your layer in percentage of the display's dimensions. Thus, if you choose the point `{0.5, 0.5}`, the upper left corner of your layer will be in the center of your display. By default, this value is `{0, 0}` which corresponds to the top-left corner.

The `size` facet is used with a point type, between `{0, 0}` and `{1, 1}` also. It corresponds to the size occupied by the layer in percentage of the display's dimensions. By default, this value is `{1, 1}` which represents 100% of the width and height available.

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" position:{0,0} size:{0.5,0.8} {
                draw shape color:#darkorange;
            }
            graphics "layer2" position:{0.3,0.1} size:{0.6,0.2} {
```



NB: `displays` can have a `background`, while `graphics` can't. If you want to put a background for your `graphics`, a solution can be to draw the `shape` of the world (which is, by default, a square 100m*100m).

A lot of other facets are available for the various layers. Please read the documentation of `graphics` for more information.

Example of layers

species layer

`species` allows the modeler to display all the agents of a given species in the current display. In particular, the modeler can choose the aspect used to display them.

Please read the documentation about [species](#) statement if you are interested.

grid layer

Similarly to [species](#), [grid](#) allows the modeler to display all the agents of a given species in the current display, but only in the case where the species is a grid. The [lines](#) color can be specified. The inner color of the cells is determined by the [color](#) built-in attribute of grid agents. This is an optimized way of displaying the grid agents (compared to the species layers).

Please read the documentation about [grid](#) agents if you are interested.

agents layer

[agents](#) allows the modeler to display only the agents that fulfill a given condition.

Please read the documentation about [agents](#) statement if you are interested.

image layer

[image](#) allows the modeler to display an image (e.g. as the background of a simulation).

Please read the documentation about [image](#) statement if you are interested.

graphics layer

[graphics](#) allows the modeler to freely draw shapes/geometries/texts without having to define a species.

Please read the documentation about [graphics](#) statement if you are interested.



Version: 1.9.1

Defining 3D Displays

OpenGL display

To use an OpenGL display, we have to define the attribute `type` of the display with `type:opengl` in the chosen `display` of your model (or use the preferences->display windows to use it by default):

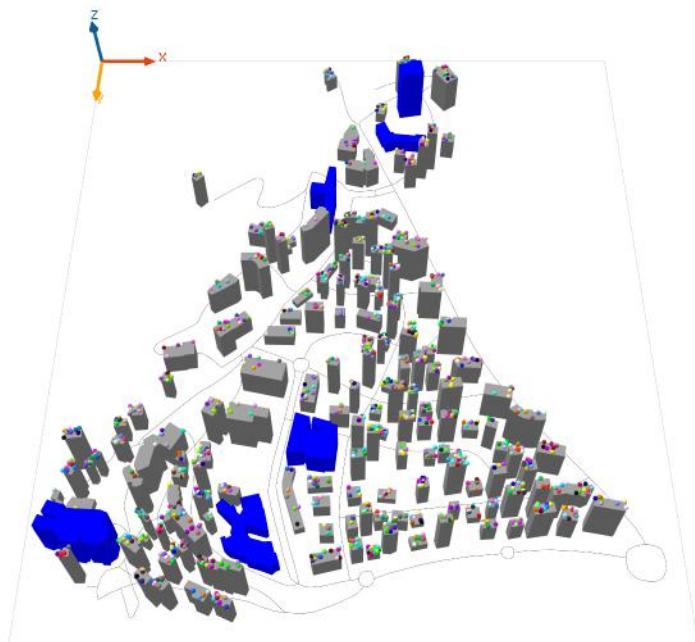
```
output {
    display DisplayName type: opengl {
        species mySpecies;
    }
}
```

The OpenGL display shares most of the features that the java2D offers and that are described [here](#). Using 3D display offers much more options to draw and show a simulation. A layer can be positioned and scaled in a 3D world. It is possible to superpose layers on different z value and display different information on the model at different positions on the screen.

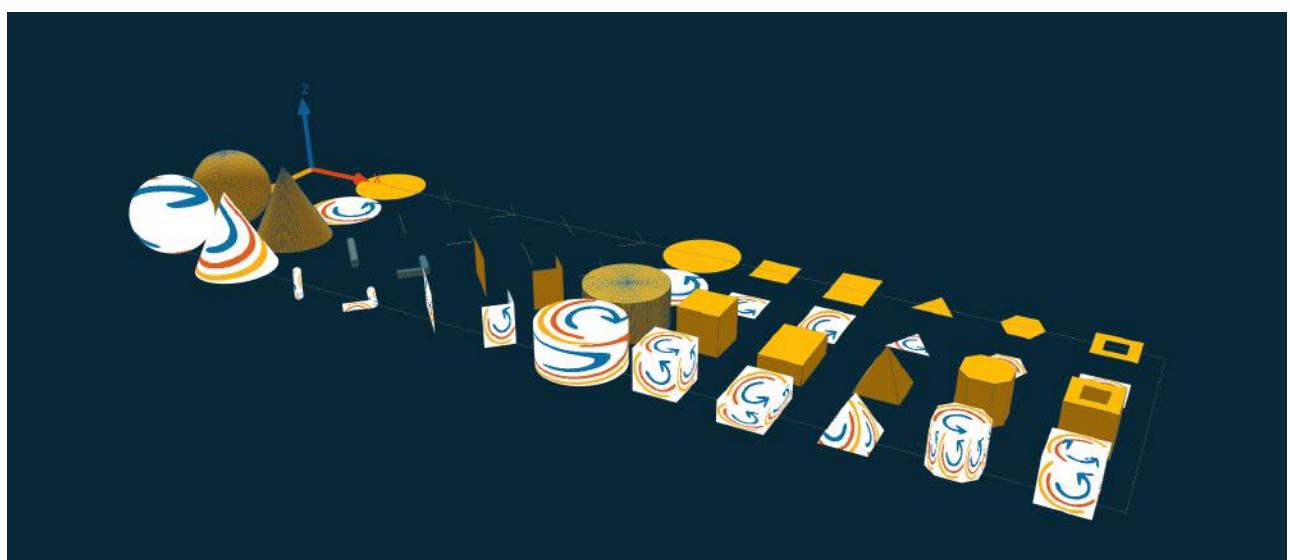
Most of the features offered by GAMA in 3D can be found as model example in the model library in the [Visualization and User Interaction/3D Visualization](#)

Such as:

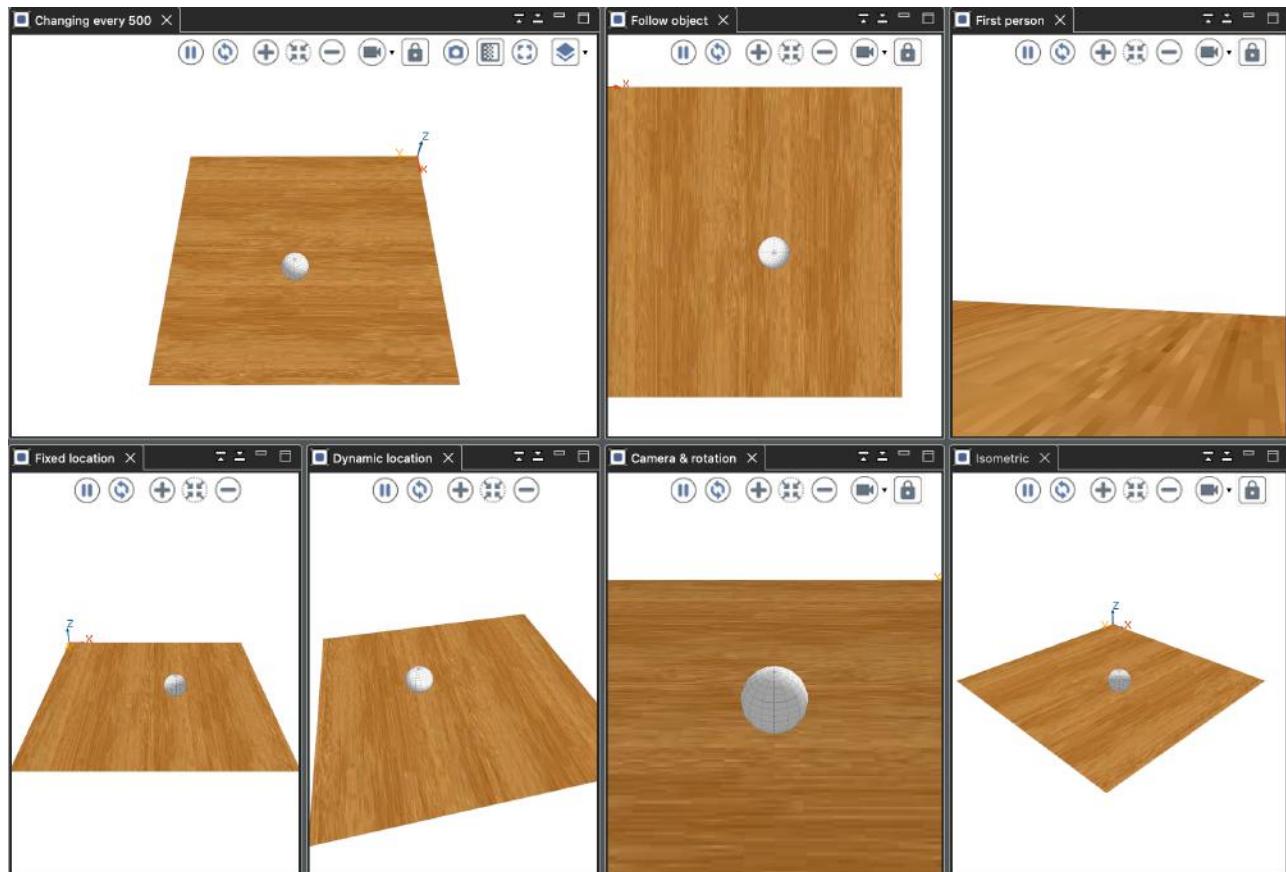
- 3D Model: Creating a simple model with building in 3D from a GIS file extruded in [Building Elevation.gaml](#)



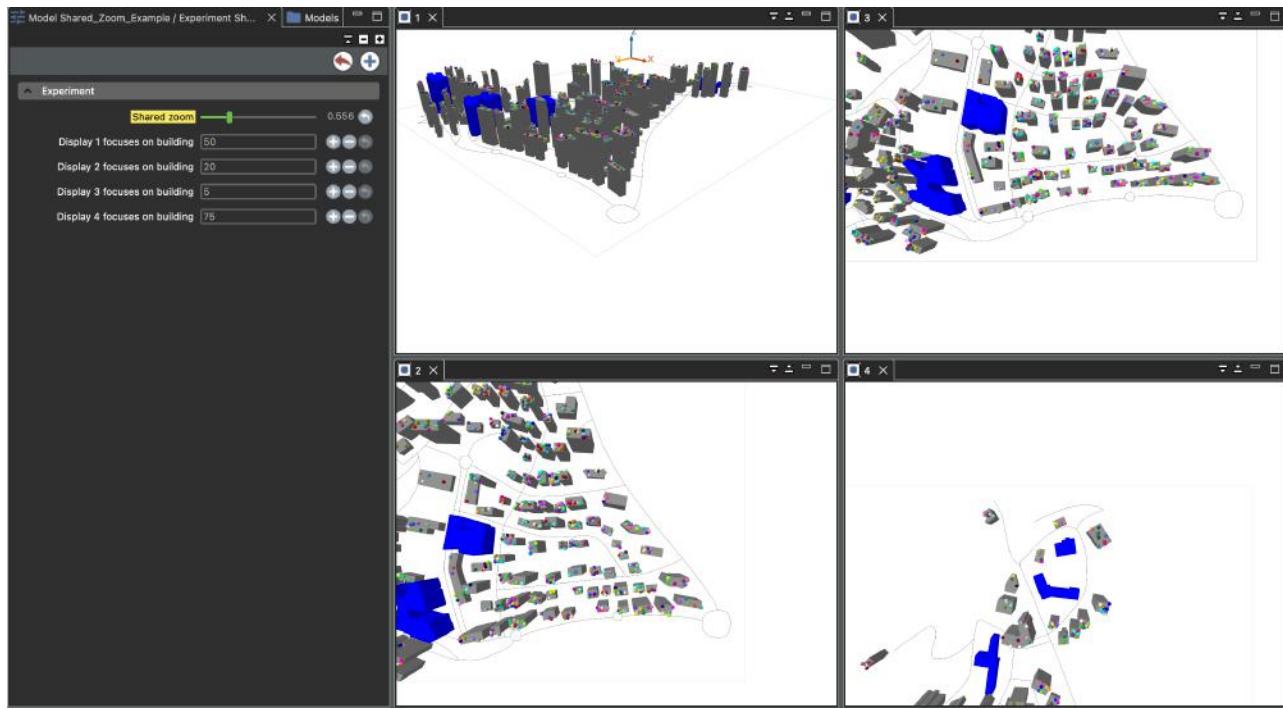
- Built-in 3D shapes supported by GAMA are described in [Built-In Shapes.gaml](#)



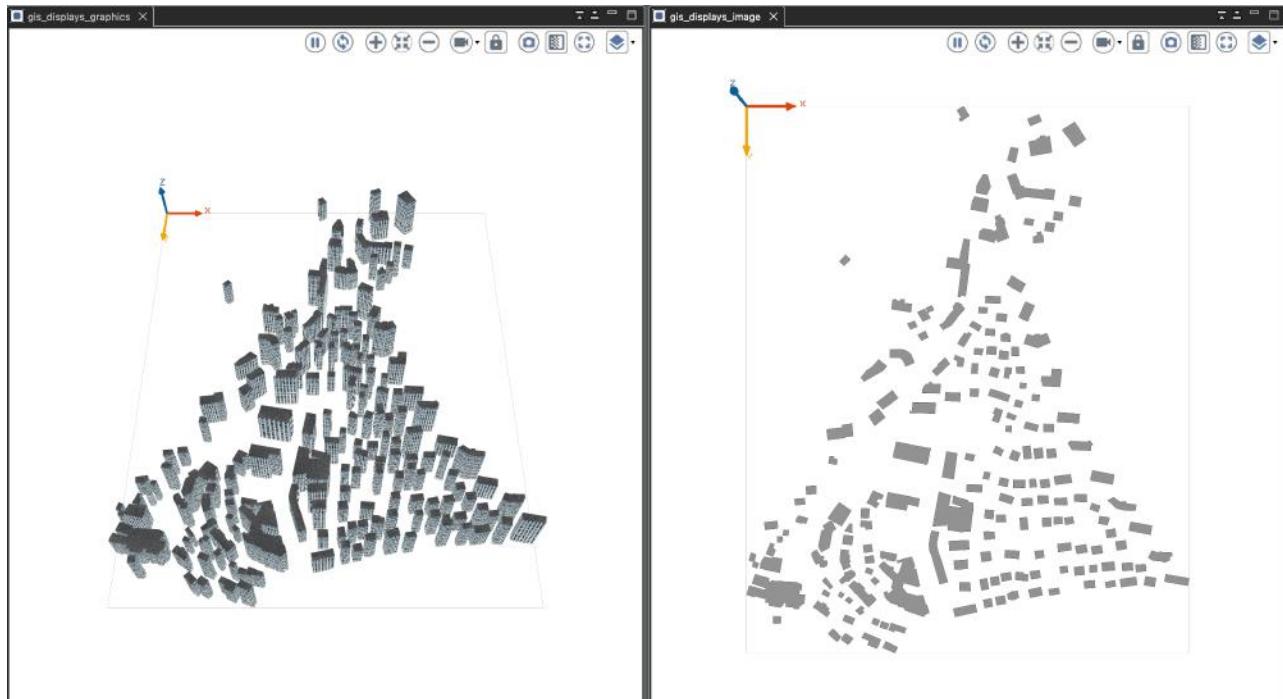
- Features related to camera and the way to manipulate it are found in [Camera Definitions.gaml](#)



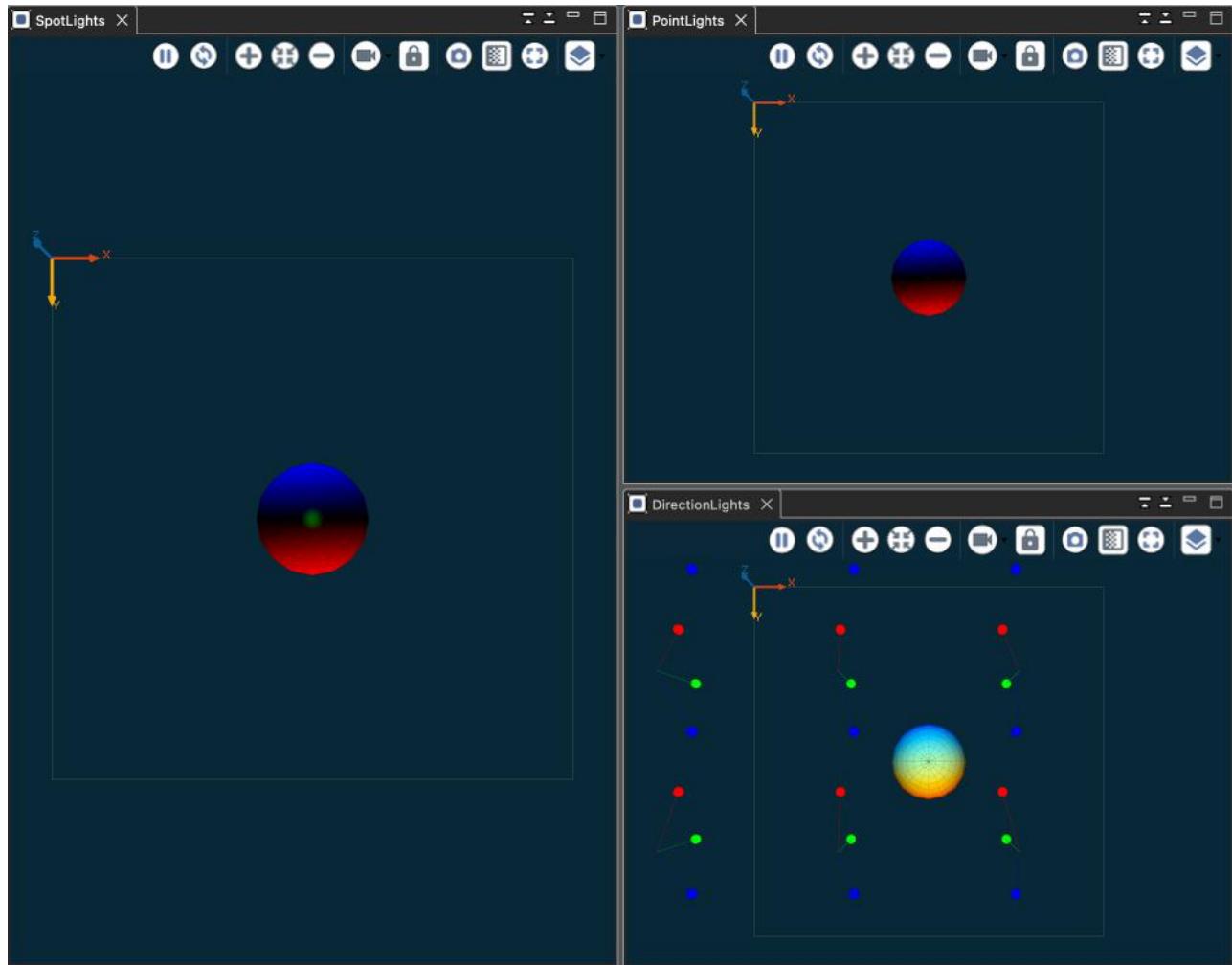
- Different point of view can be described on the same simulation and shared by different displays in [Camera Shared Zoom.experiment](#)



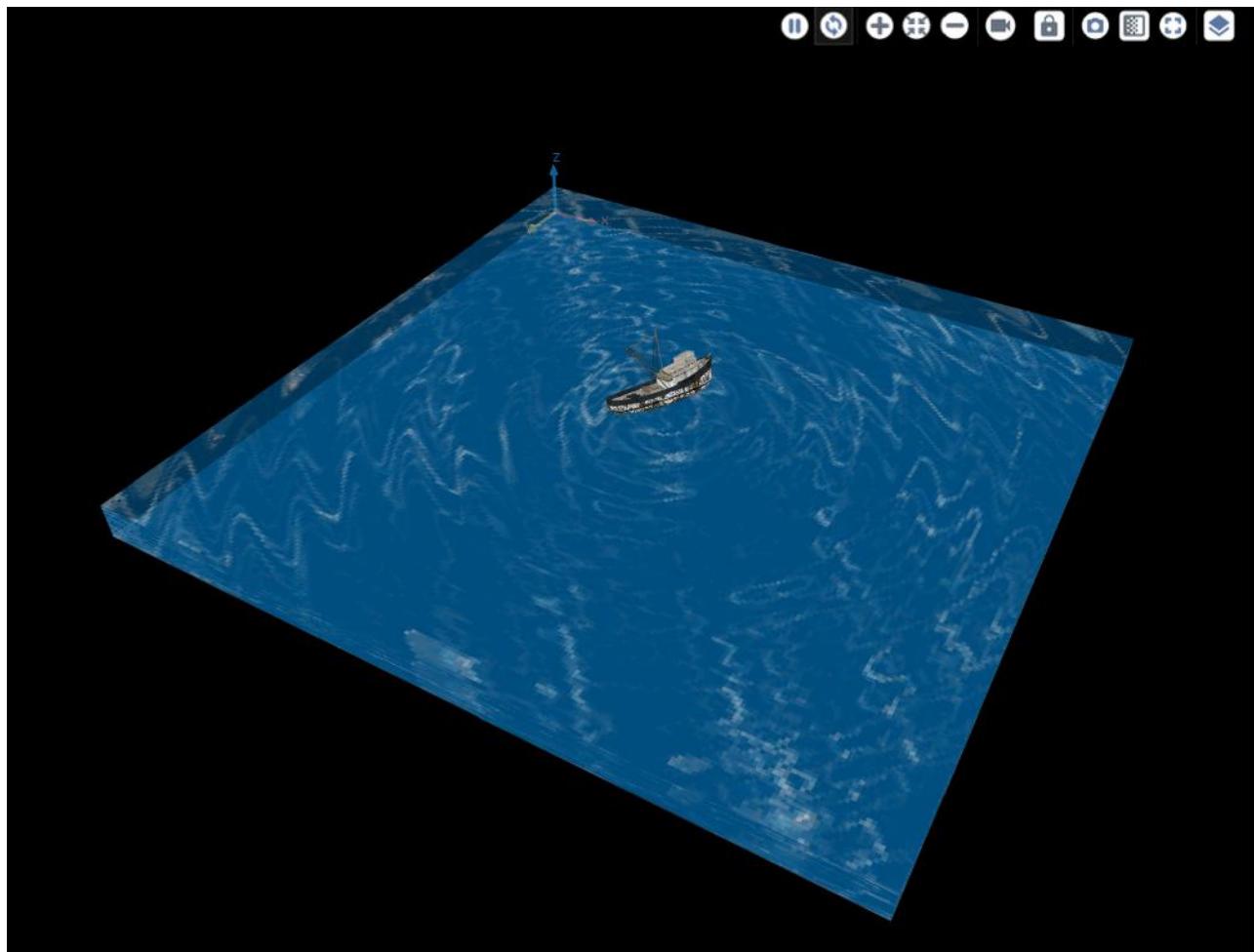
- Any GIS file can be visualized in 3D and a texture can be applied to the 3D shape in [GIS Visualization.gaml](#)



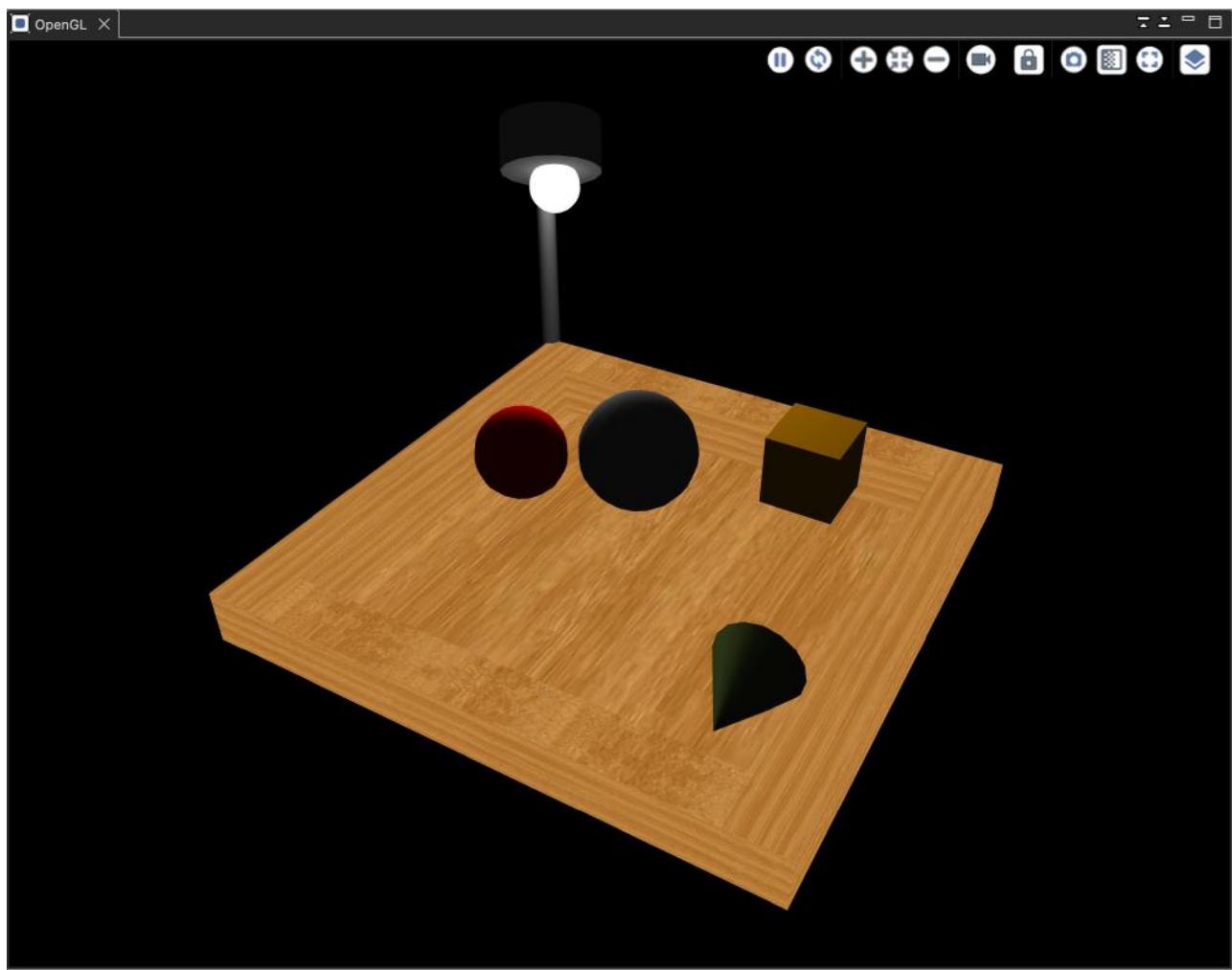
- GAMA is handling different kind of lighting such as spot lights and point lights as illustrated in [Lighting.gaml](#)



- [Moving 3D object.gaml](#) which shows how to draw a moving objet as a OBJ File and how to apply a 3D rotation on it



- Specular light can also be defined as illustrated in [Specular effects.gaml](#)





Version: 1.9.1

Defining Charts

To visualize results and make analysis about your model, you will certainly have to use charts. You can define several types of charts in GAML among histograms, pie, series, radar, heatmap... For each type, you will have to determine the data you want to highlight.

Index

- [Define a chart](#)
- [Data definition](#)
- [Various types of charts](#)
 - [pie](#)
 - [series](#)
 - [histogram](#)
 - [xy](#)
 - [heatmap](#)
 - [radar](#)
 - [scatter](#)
 - [box_whisker](#)
- [Other charts possibilities](#)

Define a chart

To define a chart, we have to use [the chart statement](#). A chart has to be named (with the `name` facet), and the type has to be specified (with the `type` facet). The value of the

`type` facet can be `histogram`, `pie`, `series`, `scatter`, `xy`, `radar`, `heatmap` or `box_whisker`. A chart has to be defined inside a display.

```
experiment my_experiment type: gui {
    output {
        display "my_display" {
            chart "my_chart" type: pie {

            }
        }
    }
}
```

`chart` can be configured by setting by facets: in particular the labels in x and y-axis can be set (`x_serie_labels`, `y_serie_labels`), axes colors (`axes`), a third axis can be added...

After declaring your chart, you have to define the data you want to display in your chart.

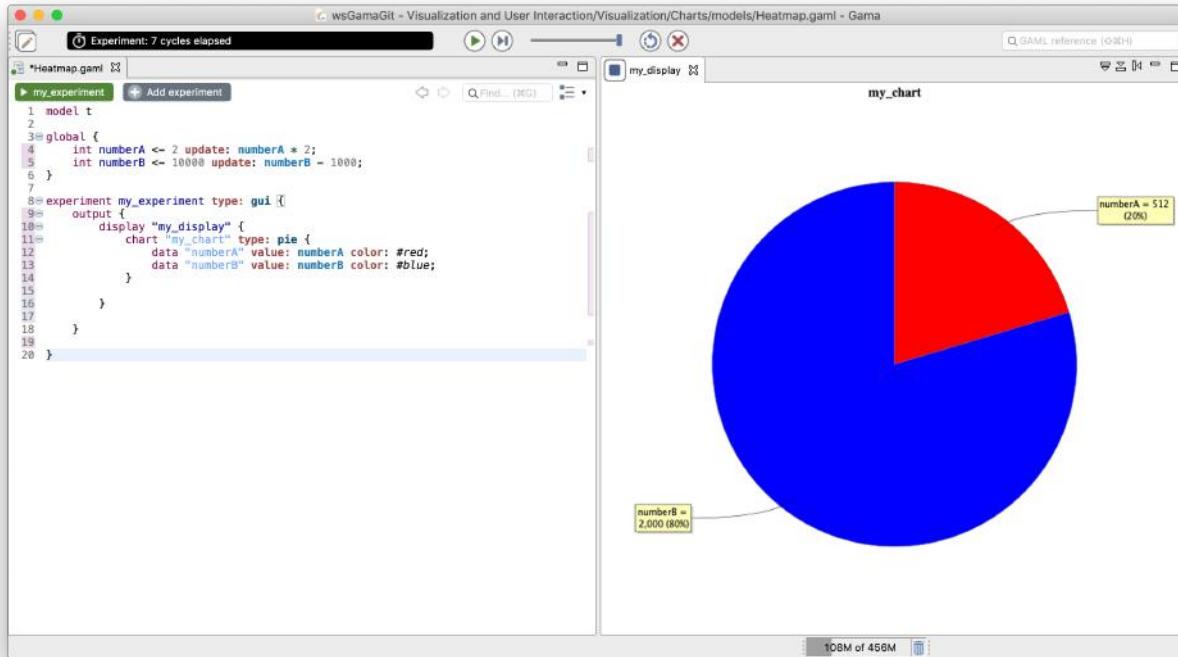
Data definition

Data can be specified with:

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

The `data` statement is used to specify which expression will be displayed. You have to give your data a name (that will be displayed in your chart), the value of the expression you want to follow (using the `value` facet). You can add some optional facets such as `color` to specify the color of your data.

```
global {
```



The `datalist` statement is used to write several `data` statements in one statement. Instead of giving simple values, `datalist` is expecting value lists. The previous chart is thus equivalent to the following one using the `datalist` statement:

```

display "my_display2" {
    chart "my_chart2" type: pie {
        datalist ["numberA", "numberB"] value: [numberA, numberB] color:
        [#red, #blue];
    }
}

```

`datalist` is particularly suitable in the case where the number of data series to plot can change during the simulation. As an example, when we want to plot the evolution of an attribute value for each agent (and new agents are created), we need to use this statement. As an example, in the following model, we want to plot the `energy` of each `people` agent. Each simulation step one agent is created.

```

global {

    init {
        create people number:15;
    }

    reflex population_growth when: length(people) < 50 {
        create people number:1;
    }
}

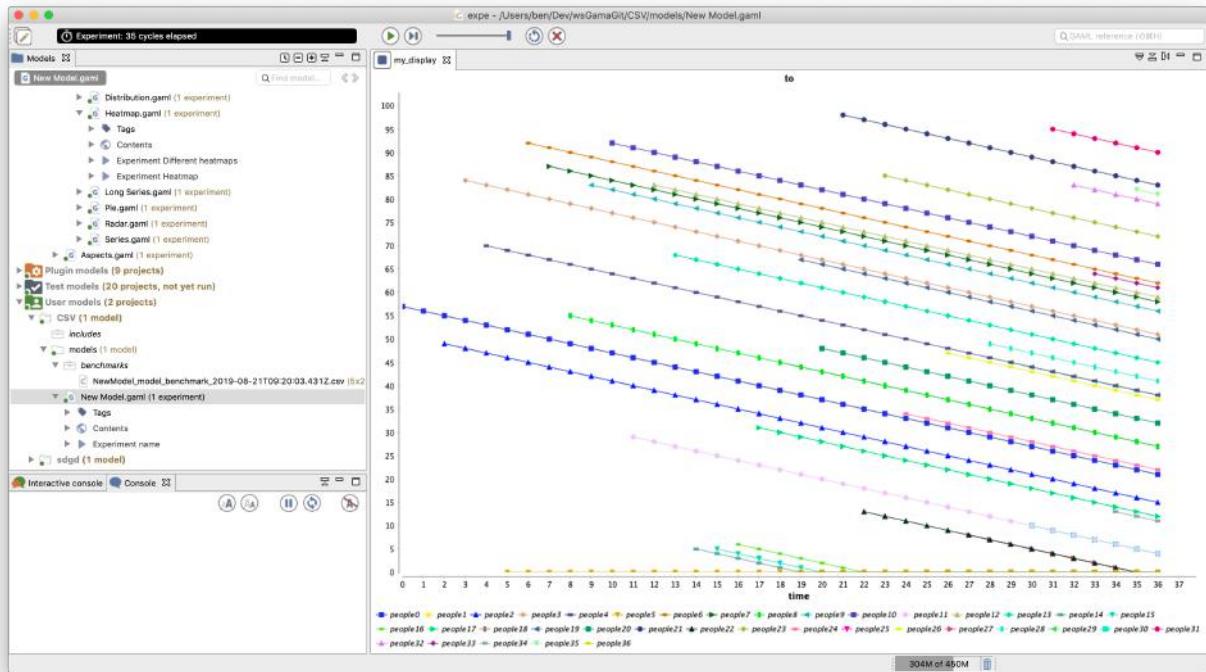
species people {

    int energy <- rnd(100) min:0;
    rgb color <- rnd_color(255);

    reflex aging {
        energy <- energy - 3;
    }
}

experiment my_experiment type: gui {
    float minimum_cycle_duration <- 0.1;
    output {
        display "my_display" {
            chart "my_chart" type: series {
                datalist people collect (each.name) value: people
collect (each.energy) color: people collect (each.color) ;
            }
        }
    }
}

```



`datalist` provides you some additional facets you can use. If you want to learn more about them, [please read the documentation](#).

Various types of charts

As we already said, you can display several types of graphs: the histograms, the pies, the series, the radars, heatmap...

pie

The `pie` chart shows on a single pie diagram the ratio of each data series over the sum of all the series. It has already been illustrated above.

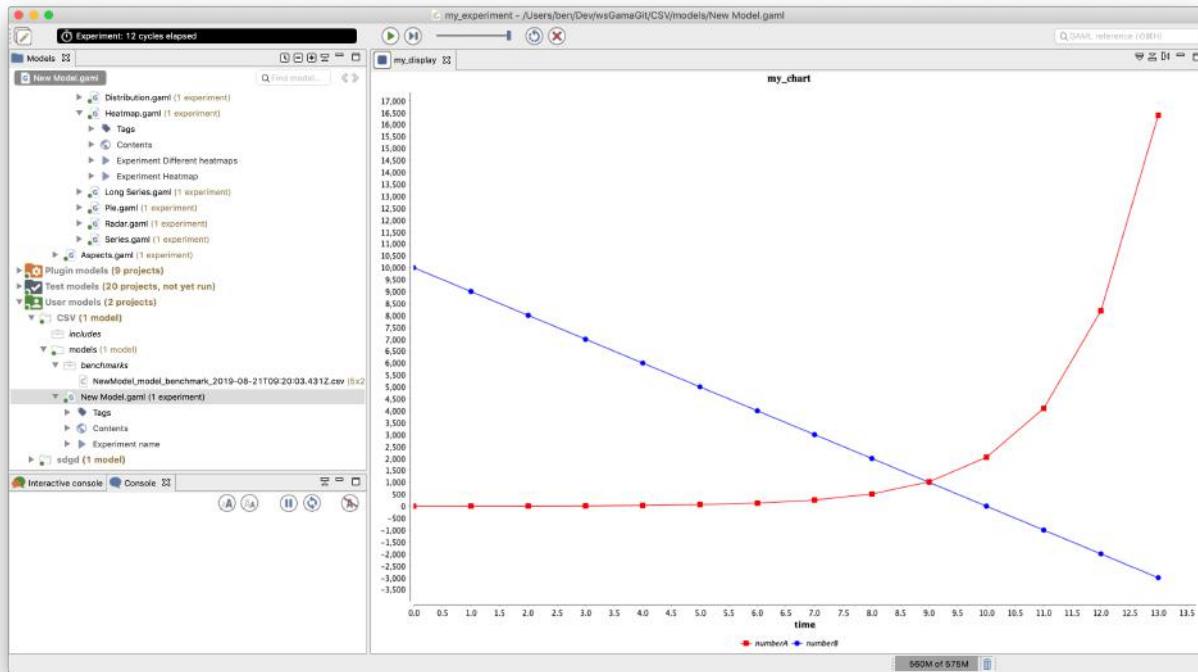
series

The `series` type is perhaps the most basic plot: it displays in an x-y coordinates space

the value of each data series over time (simulation step): the x-axis displays the simulation step, the y-axis represents the value of the data series. The previously defined `pie` chart, can be displayed using a `series` simply by changing the chart `type`.

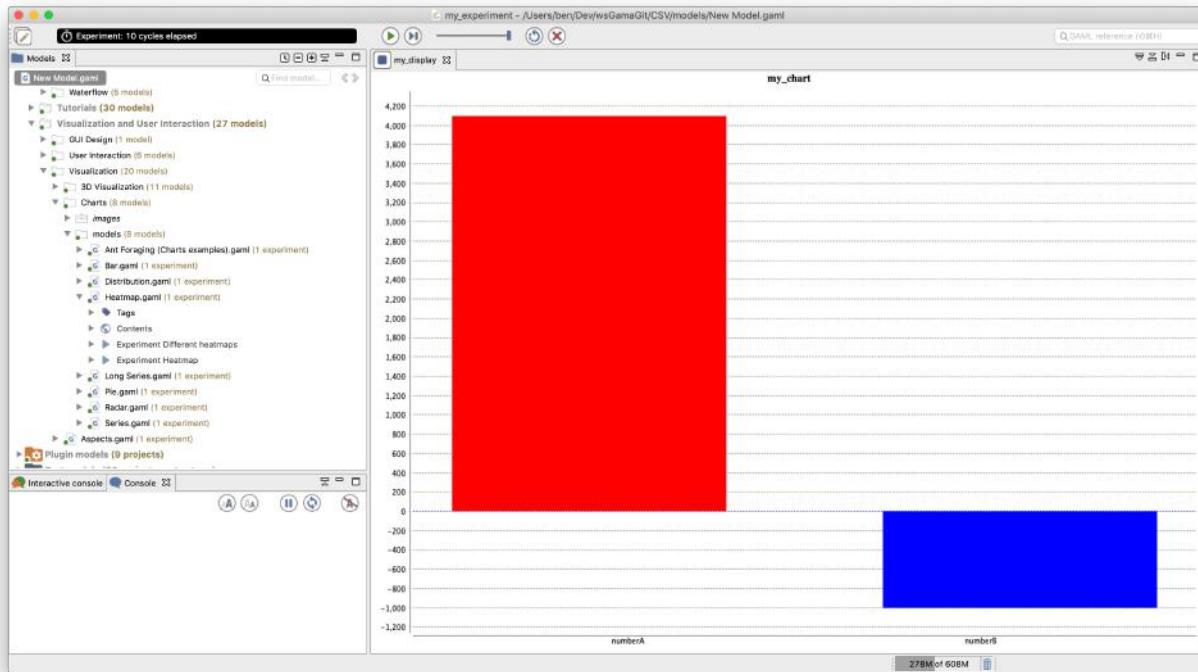
```
global {
    int numberA <- 2 update: numberA*2;
    int numberB <- 10000 update: numberB-1000;
}

experiment my_experiment type: gui {
    float minimum_cycle_duration <- 0.1;
    output {
        display "my_display" {
            chart "my_chart" type: series {
                data "numberA" value: numberA color: #red;
                data "numberB" value: numberB color: #blue;
            }
        }
    }
}
```



histogram

The `histogram` charts represent with bars the value of several data series. The previous example can be displayed with a `histogram` chart.



Histograms are often used to display the distribution of a value inside a population. For example, let consider a population of agents representing human beings with an `age` attribute. The following model illustrates the plot of the age distribution over the population. We used the operator `distribution_of` to compute the distribution to plot: here we display the number of people agent in 20 ranges computed among the ages between 0 and 100.

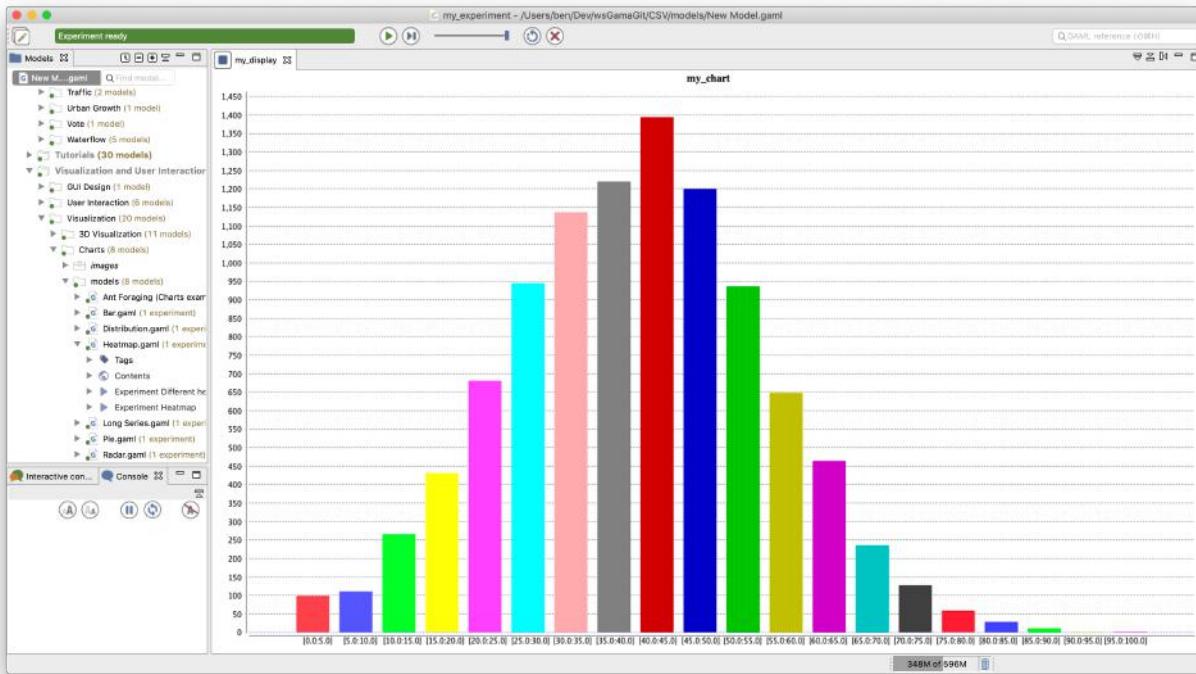
```

model NewModel

global {
    init {
        create people number: 10000;
    }
}

species people {
    float age <- gauss(40.0, 15.0);
}

```

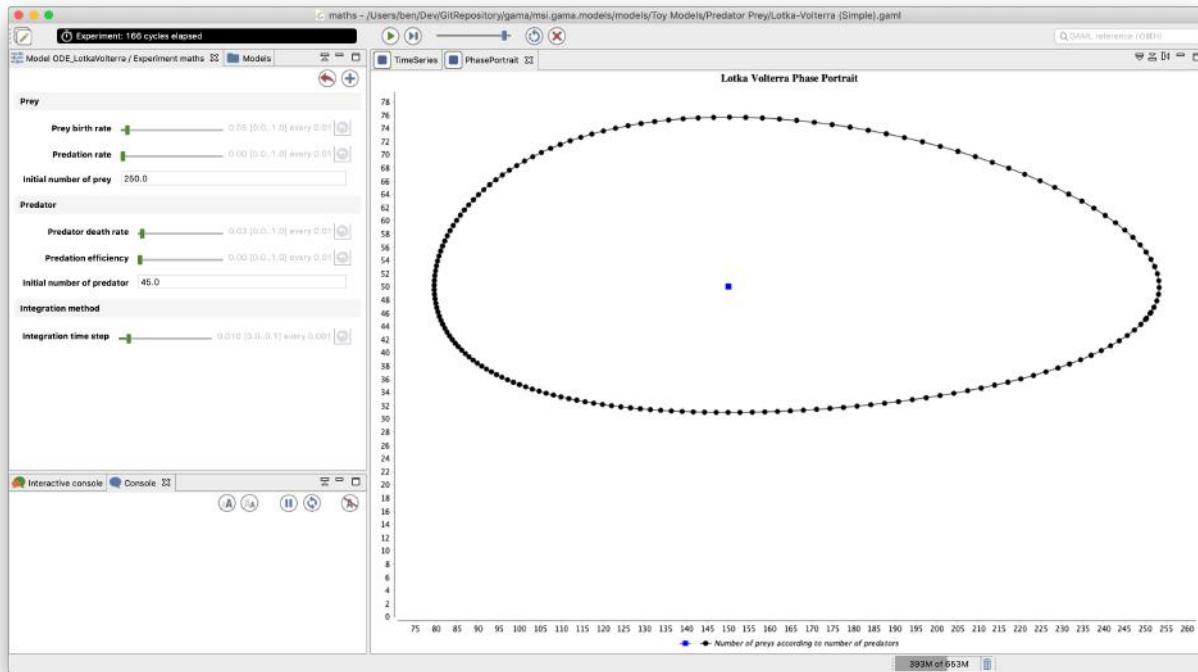


Note that the facet `reverse_axes` (with `true` value) can be added to the `chart` statement to display horizontal bars.

xy

The `xy` displays are used when we want to display a value in function of another one (instead of plotting a value in function of the time): in this case, the x-axis does not represent the time in general. It can be used for example to plot a phase portrait, e.g. in the Lotka-Volterra model (prey-predator model) in which we want to plot the number of preys according to the number of predators. The code for the chart is then:

```
display PhasePortrait {
    chart "Lotka Volterra Phase Portrait" type: xy {
        data 'Preys/Predators' value:
        {first(LotkaVolterra_agent).nb_prey,
        first(LotkaVolterra_agent).nb_predator} color: #black ;
    }
}
```

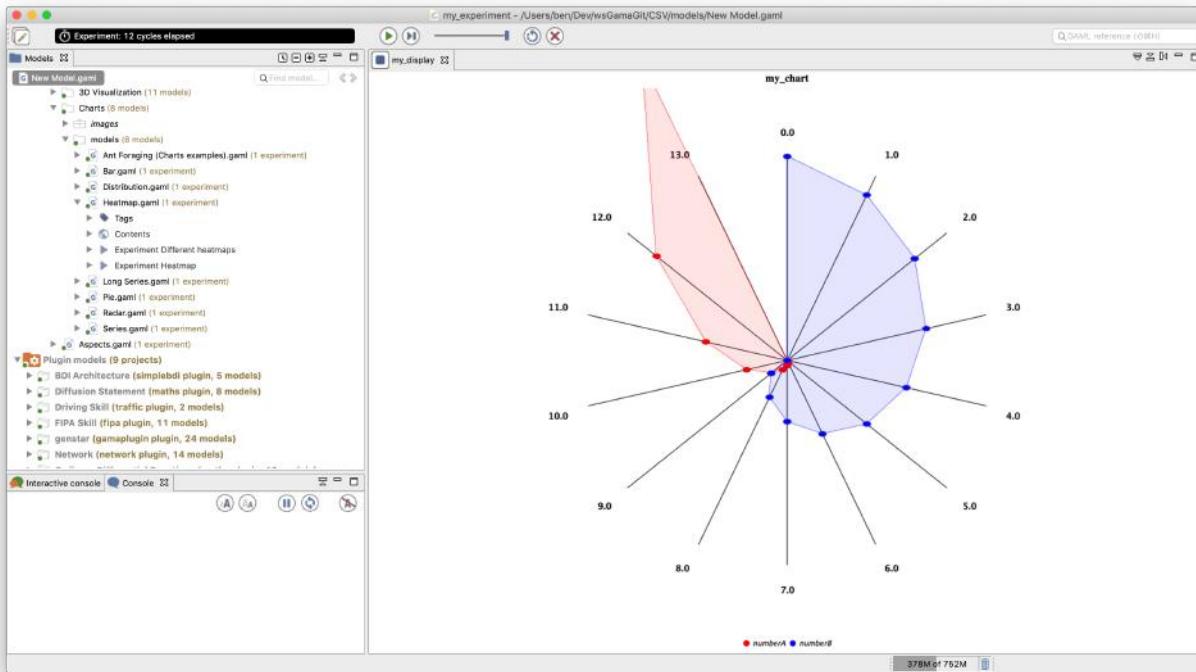


radar

A **radar** chart displays the evolution of expression over time in a kind of circular representation: the radar representation. If reuse the example describes previously and used in the previous types of charts, we get the following adapted model:

```
global {
    int numberA <- 2 update: numberA^2;
    int numberB <- 10000 update: numberB-1000;
}

experiment my_experiment type: gui {
    float minimum_cycle_duration <- 0.1;
    output {
        display "my_display" {
            chart "my_chart" type: radar background: #white axes:#black {
                data "numberA" value: numberA color: #red accumulate_values: true;
            }
        }
    }
}
```



heatmap

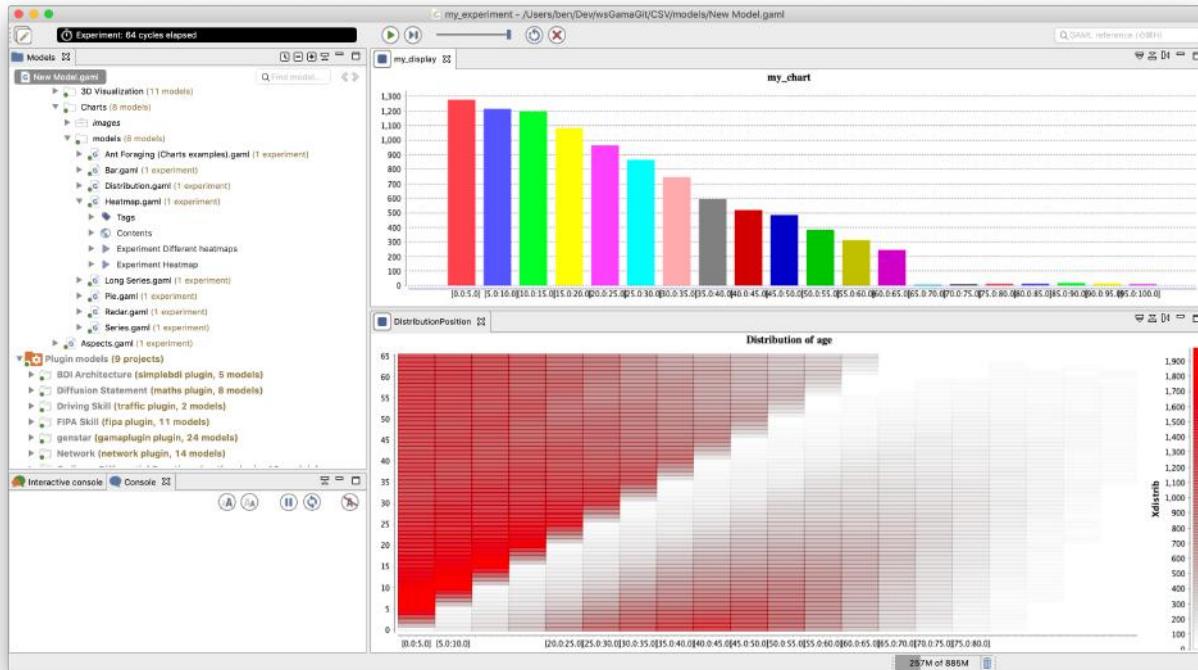
The heatmap in GAMA is close to a stack of histograms charts (allowing to keep a view of the evolution of values over time), representing the height of the bars by color in a gradient.

Let consider the model of a human population characterized by their age. We had a population dynamic: at each step, their age is incremented by 1. They also have a probability to die at each step (that increases with their age). When an agent dies, it creates a new agent with an age equals to 0.

```
model NewModel

global {
    init {
        create people number: 10000;
    }
}
```

We thus displayed the evolution of the age distribution using both a histogram chart (for the instantaneous distribution) and a heatmap display to key a track of the evolution over time. In the heatmap, the left Y-axis represents the time (the simulation step number); as a consequence 1 line represents the state at 1 simulation step. The x-axis represents the various ranges of the distribution (same meaning as for histograms). The right Y-axis shows the meaning of the color gradient.

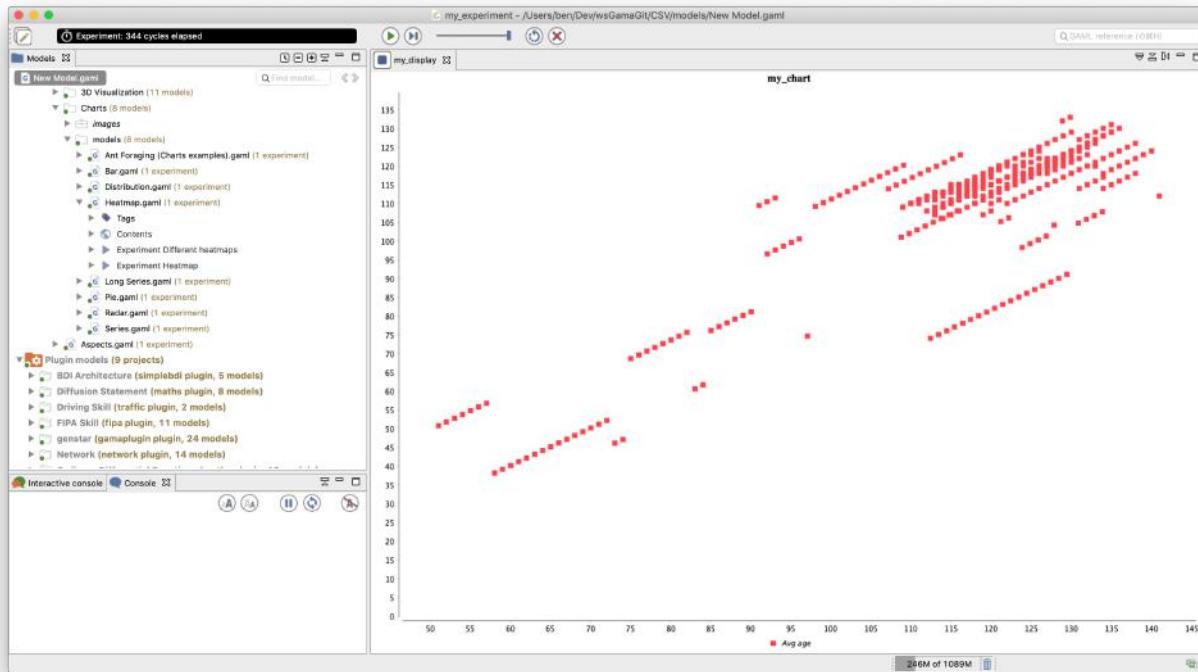


scatter

The scatter chart allows us to represent in a 2D space the "spatial distribution" of a set of values. As an example, it allows us to plot the `age` of all the `people` agents: the X-axis represents the possible age value and not the time as in a `series` charts.

Here is an example of a chart of type `scatter` on the previous model example:

```
experiment my_experiment type: gui {
```



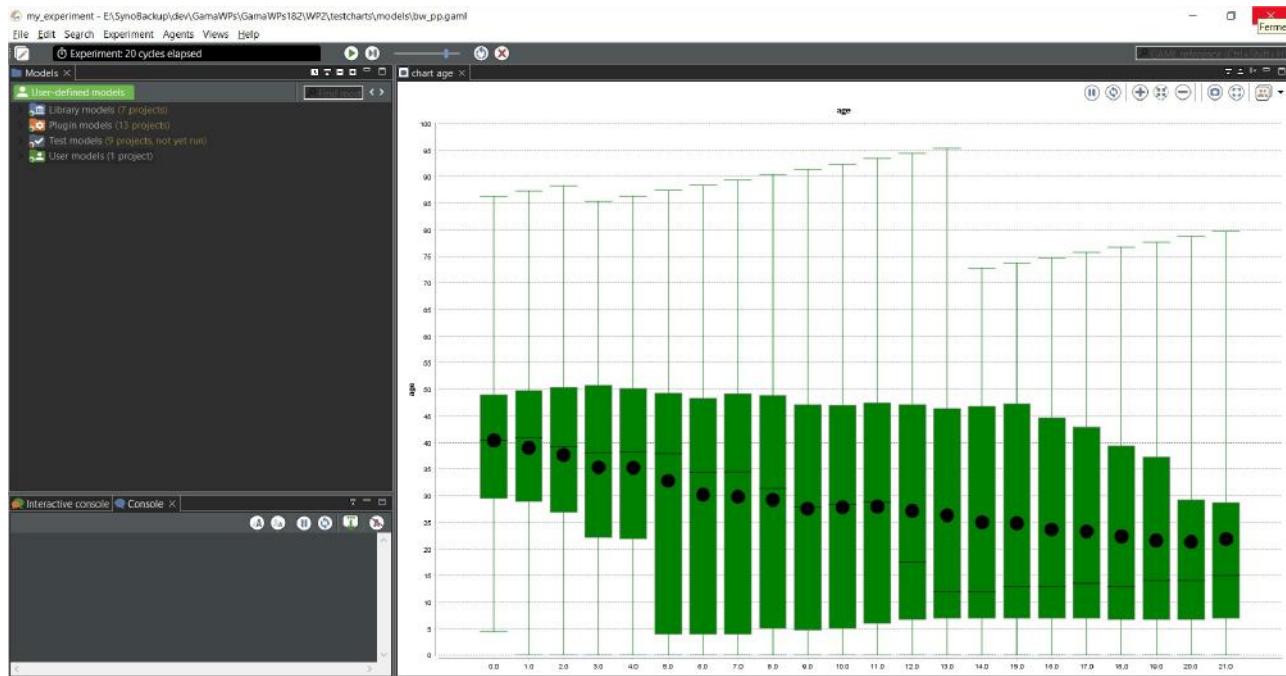
box_whisker

The `box_whisker` charts represent the distribution of a value. A circle for the average, a horizontal line for the median, a filled bar (the "box") for the top 75% and the bottom 25% and a line (the "whisker") for the maximum and minimum values.

For example, let consider again a population of agents representing human beings with an `age` attribute (with an aging mechanism). The following model illustrates the plot of the age distribution over the population.

```
model NewModel

global {
    init {
        create people number: 100;
    }
}
```



Note that the facet `series_label_position` (with `yaxis` value) int the `chart` statement is used to display the serie label ("age") on the y axis. With more series, we can display the labels as a legend (with the 'legend' value).

Other charts possibilities

The `chart`, `data` and `datalist` come with a huge number of additional facets, allowing you to design advanced result display. We can mention here some of them.

Error values

Just as a box plot, drawing error values around a value, allows the user to visually identify a value (e.g. a mean value) and the distribution of this value around it. The `y_err_values` facet of `data` can be used to show in the data plot and a range around it (e.g. the min and max value of an expression in the agent population).

In this example, we plot the average age of agents in the population, with the minimum and maximum value. Here is only the experiment code related to the model shown in

the previous parts.

```
experiment my_experiment type: gui {
    float minimum_cycle_duration <- 0.1;
    output {
        display "my_display" {
            chart "my_chart" type: series {
                data "average age" value: people mean_of each.age color: #red
                y_err_values: [people min_of each.age,people
max_of each.age];
            }
        }
    }
}
```



Version: 1.9.1

Defining monitors and inspectors

Other outputs can be very useful to study better the behavior of your agents.

Index

- [Define a monitor](#)
- [Define an inspector](#)

Define a monitor

A [monitor](#) allows to follow the value of an arbitrary expression in GAML. It will appear, in the User Interface, in a small window on its own and be recomputed every time step (or according to its [refresh](#) facet).

Definition of a monitor:

```
monitor monitor_name value: an_expression refresh: boolean_statement;
```

with:

- [value:](#) mandatory, the expression whose value will be displayed by the monitor.

- `refresh`: bool statement, optional: the new value is computed if the bool statement returns true.

Example:

```
experiment my_experiment type: gui {
    output {
        monitor monitor_name value: cycle refresh: every(1#cycle);
    }
}
```

NB: you can also declare monitors during the simulation, by clicking on the button "Add new monitor", and specifying the name of the variable you want to follow.

Define an inspector

During the simulation, the user interface of GAMA provides the user the possibility to [inspect an agent](#), or a group of agents. But you can also define the inspector you want directly from your model, as an output of the experiment.

Use the statement `inspect` to define your inspector, in the output scope of your GUI experiment. The inspector has to be named (using the facet `name`), a value has to be specified (with the `value` facet).

```
inspect "inspector_name" value: the_value_you_want_to_display;
```

Note that you can inspect any type of species (regular species, grid species, even the world...) or agent.

The optional facet `type` is used to specify the type of your inspector. 2 values are possible:

- `agent` (default value) if you want to display the information as a regular [agent inspector](#). Note that if you want to inspect a large number of agents, this can take a lot of time. In this case, prefer the other type `table`
- `table` if you want to display the information as an [agent browser](#).

The optional facet `attributes` is used to filter the attributes you want to display in your inspector.

Beware: only one agent inspector (`type: agent`) can be used for an experiment. Besides, you can add as many agent browsers (`type: table`) as you want for your experiment.

Example of implementation:

```
model new

global {
    init {
        create my_species number:3;
    }
}

species my_species {
    int int_attr <- 6;
    string str_attr <- "my_value";
    string str_attr_not_important <- "blabla";
}

grid my_grid_species width: 10 height: 10 {
    int rnd_value <- rnd(5);
}

experiment my_experiment type:gui {
    output {
        inspect "my_species_inspector" value: my_species attributes:
        ["int_attr", "str_attr"];
    }
}
```

Another statement, `browse`, is doing a similar thing, but prefer the `table` type (if you want to browse an agent species, the default type will be the `table` type).



Version: 1.9.1

Defining export files

The Save Statement

Allows to save data in a file. The type of file can be "shp", "json" and "kml" for vector spatial data (agents and geometries), "asc" and "geotiff" for raster spatial data (grid), "image" for image, "dimacs", "dot", "gexf", "graphml", "gml" and "graph6" for graphs, "text" and "csv". The `save` statement can be used in an init block, a reflex, an action or in a user command.

Facets

- `attributes`, optional, expects any type in [map, list] - Allows to specify the attributes of a shape file or GeoJson file where agents are saved. Can be expressed as a list of string or as a literal map. When expressed as a list, each value should represent the name of an attribute of the shape or agent. The keys of the map are the names of the attributes that will be present in the file, the values are whatever expressions needed to define their value.
- `crs`, optional, expects any type - the name of the projection, e.g. `crs:"EPSG:4326"` or its EPSG id, e.g. `crs:4326`. [Here](#) a list of the CRS codes (and EPSG id)
- `data`, optional, expects any type - the data that will be saved to the file
- `header`, optional, expects bool - an expression that evaluates to a boolean, specifying whether the save will write a header if the file does not exist
- `rewrite`, optional, expects bool - a boolean expression specifying whether to erase the file if it exists or append data at the end of it. Only applicable to "text" or "csv" files. Default is true

- `to`, optional, expects string - an expression that evaluates to a string, the path to the file, or directly to a file
- `format`, optional, a string representing the format of the output file (e.g. `shp`, `asc`, `geotiff`, `png`, `text`, `csv`). If the file extension is non ambiguous in facet 'to:', this format does not need to be specified. However, in many cases, it can be useful to do it (for instance, when saving a string to a .pgw file, it is always better to clearly indicate that the expected format is 'text').
- `type`, optional, deprecated, use `format` instead.

Usages

- Its simple syntax is:

```
save data to: output_file format: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->" + name + ":" + location) to:  
"save_data.txt" format: text;
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" format: csv;
```

- To save the geometries of all the agents of a species into a shapefile or a geojson (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" format: shp  
attributes: [name::"nameAgent", location::"locationAgent"] crs:
```

- To save a grid into a geotiff or a asc file (the value considered will be the "grid_value" attribute of the cell):

```
save cell to:"../results/grid.tif" format:geotiff;  
save cell to:"../results/grid.asc" format:asc;
```

- To save a grid into an image file:

```
save cell to:"../results/grid.png" format:image;
```

Export files in an experiment

When the modeler wants to save data at each simulation step, it is recommended to use the `save` statement either in the model itself or in a `reflex` of the `experiment` (the syntax and the use are similar in all the cases).

The use of `save` in `experiment` is mandatory when we want to save a value related to several simulations running in parallel (e.g. the average of a value over several simulations). It is in particular in `batch experiments` to save a value at the end of simulations.

Autosave

Image files can be exported also through the `autosave` facet of the display, as explained in [this previous part](#).



Version: 1.9.1

Defining user interaction

During the simulation, GAML provides you the possibility to define some function the user can execute during the execution. In this chapter, we will see how to define buttons to execute action during the simulation, how to catch click event, and how to use the user control architecture.

Index

- [Catch Mouse Event](#)
- [Define User command](#)
 - [... in the GUI Experiment scope](#)
 - [... in global or regular species](#)
 - [user_location](#)
 - [user_input](#)
- [User Control Architecture](#)

Catch Mouse Event

You can catch mouse event during the simulation using the statement `event`. This statement has 2 required facets:

- `name` (identifier) : Specify which event do you want to trigger (among the following values : `mouse_down`, `mouse_up`, `mouse_move`, `mouse_enter`, `mouse_exit` or any alphanumeric symbol/key of the keyboard, such as, `'a'`,

' b' ...).

- **action** (identifier) : Specify the name of the global action to call.

```
event mouse_down action: my_action;
```

The `event` statement has to be defined in the `experiment/output/display` scope. Once the event is triggered, the global action linked will be called. The action linked cannot have arguments. To get the location of the mouse click, the `#user_location` can be used; to get the agents on which the mouse has clicked, you can use spatial query (e.g. `my_species overlapping #user_location`).

```
global
{
    action my_action
    {
        write "do action";
    }
}

species my_species
{

}

experiment my_experiment type: gui
{
    output
    {
        display my_display
        {
            species my_species;
            event mouse_down action: my_action;
        }
    }
}
```

Define User command

Anywhere in the global block, in a species or in an (GUI) experiment, `user_command` statements can be implemented. They can either call directly an existing action (with or without arguments) or be followed by a block that describes what to do when this command is run.

Their syntax can be (depending of the modeler needs) either:

```
user_command cmd_name action: action_without_arg_name;  
//or  
user_command cmd_name action: action_name with: [arg1::val1,  
arg2::val2];  
//or  
user_command cmd_name {  
    // statements  
}
```

For instance:

```
user_command kill_myself action: die;  
//or  
user_command kill_myself action: some_action with: [arg1::5, arg2::3];  
//or  
user_command kill_myself {  
    do die;  
}
```

Defining User command in GUI Experiment scope

The user command can be defined directly inside the GUI experiment scope. In that

case, the implemented action appears as a button in the top of the parameter view.

Here is a very short code example :

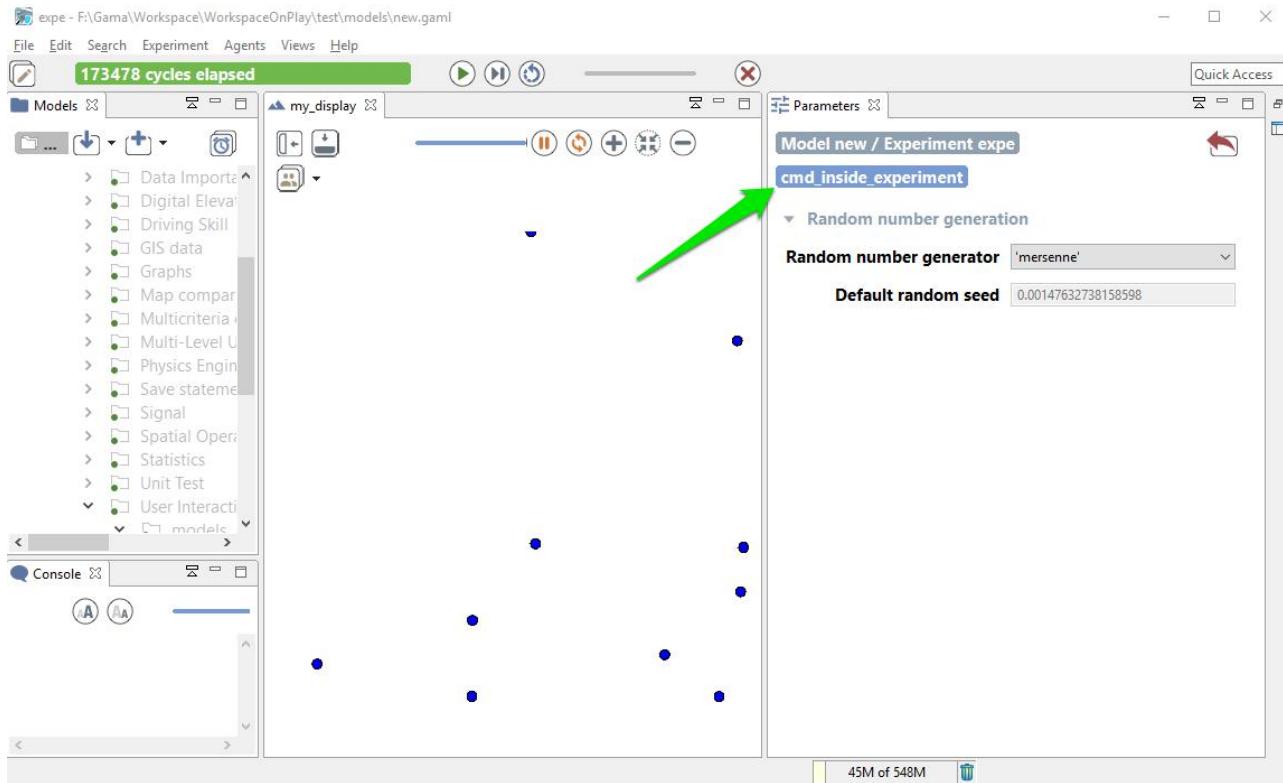
```
model quick_user_command_model

global {
    action createAgent
    {
        create my_species;
    }
}

species my_species {
    aspect base {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    user_command cmd_inside_experiment action:createAgent;
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

And here is screenshots of the execution :



Defining User command in a global or regular species

The user command can also be defined inside a species scope (either global or regular one). Here is a quick example of model :

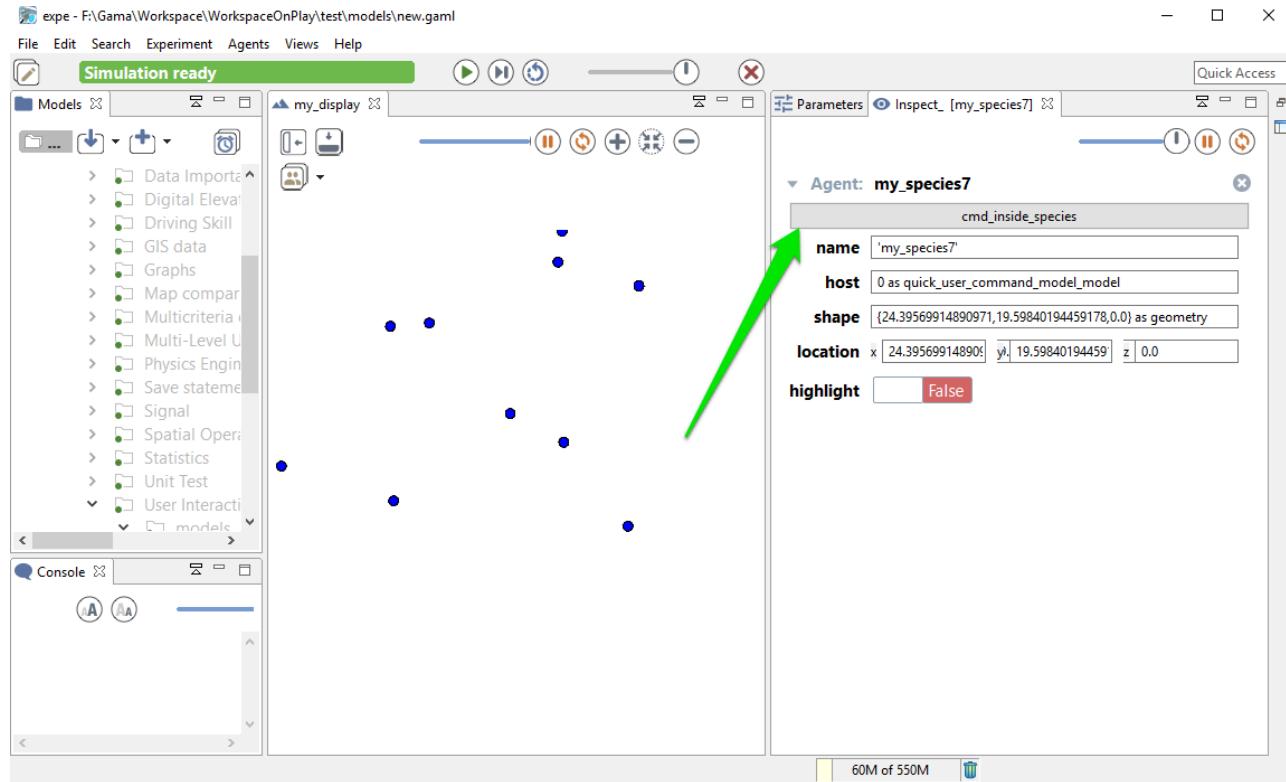
```
model quick_user_command_model

global {
    init {
        create my_species number:10;
    }
}

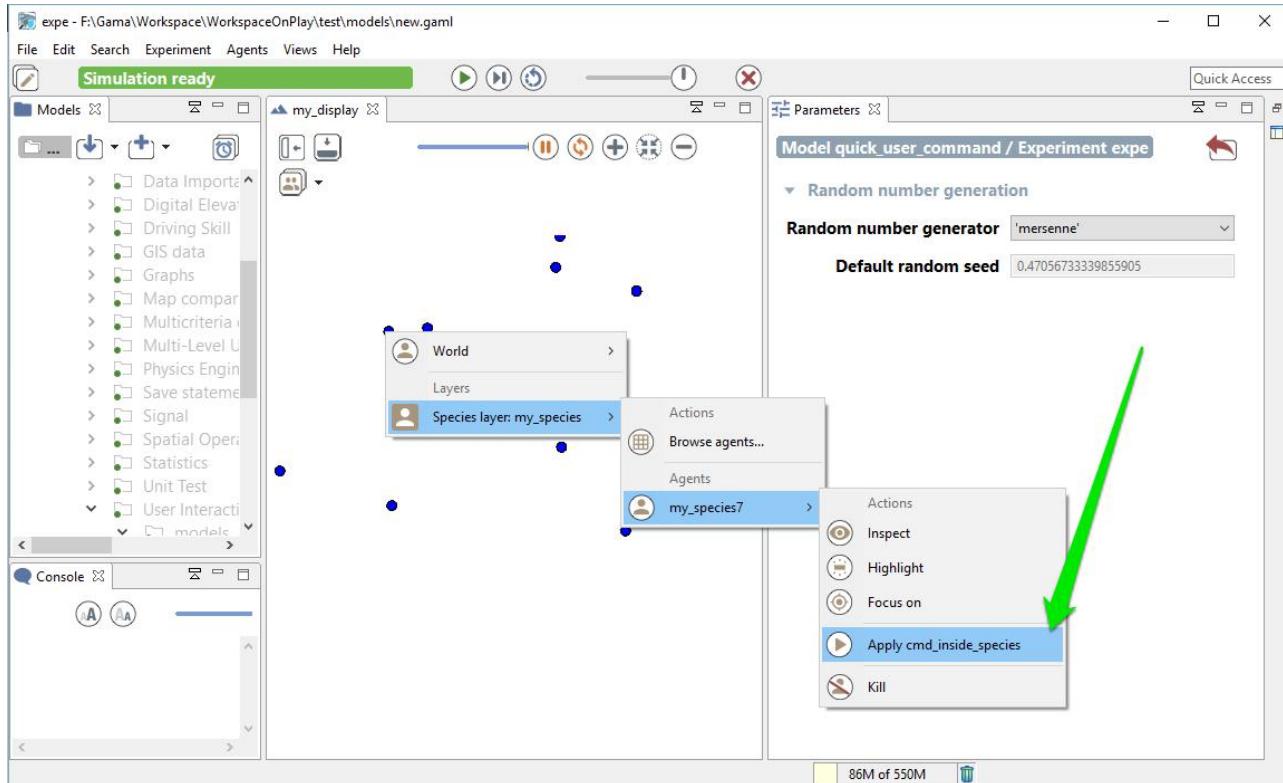
species my_species {
    user_command cmd_inside_experiment action:die;
    aspect base {
```

During the execution, you have 2 ways to access to the action:

- When the agent is inspected, they appear as buttons above the agents' attributes



- When the agent is selected by a right-click in a display, these commands 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

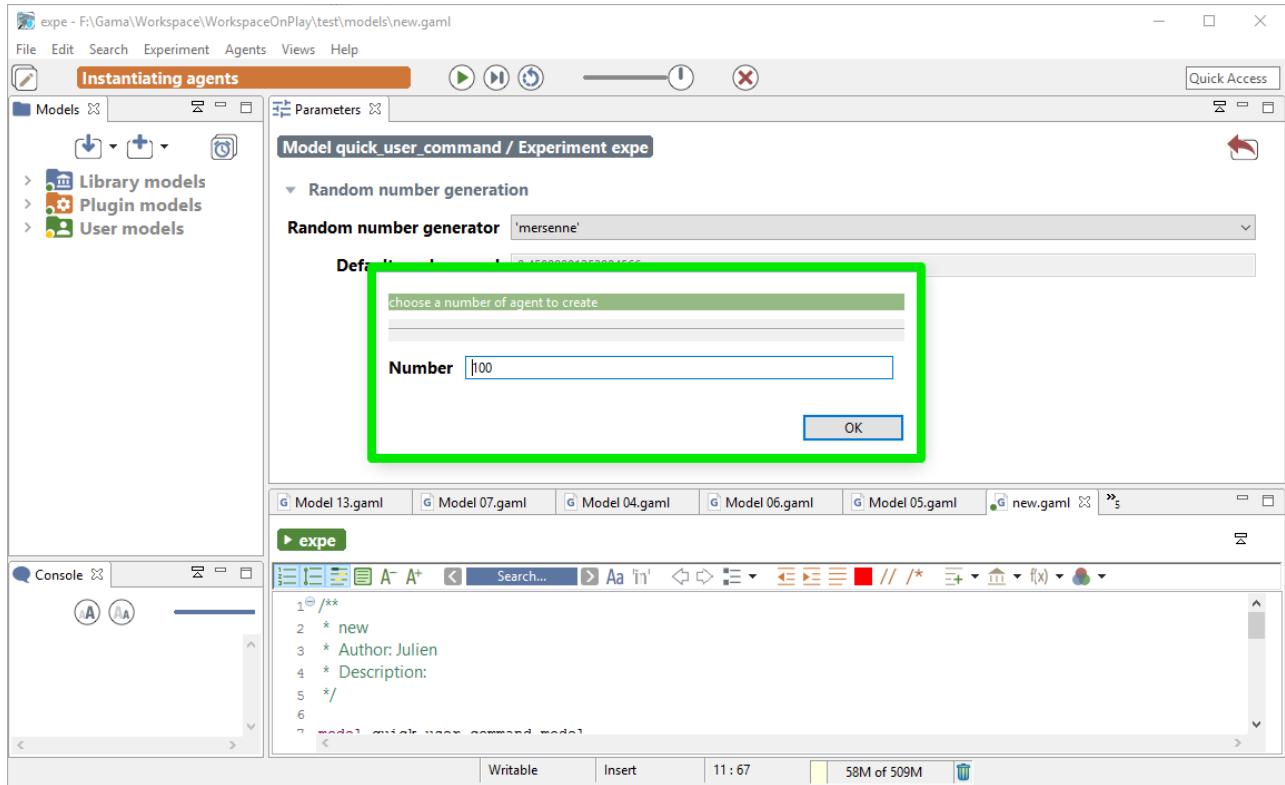
As it is also, sometimes, necessary to ask the user for some values (not defined as parameters), the `user_input` unary operator has been introduced. This operator takes a map [string::value] as argument (the key is the name of the chosen parameter, the value is the default value), displays a dialog asking the user for these values, and returns the same map with the modified values (if any). You can also add a text as first argument of the operator, which will be displayed as a title for your dialog popup. The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section like the following one to force the user to input new values instead of relying on the initial values of parameters.

Here is an example of implementation:

```
model quick_user_command_model

global {
    init {
        map values <- user_input("Choose a number of agent to
```

When running this model, you will first have to input a number:



User Control Architecture

The other way to define user interaction is to use user control architecture. Please jump directly to the section [user control architecture](#) if you want to learn more about this point.



> Learn GAML step by step

> Exploring Models

Version: 1.9.1

Exploring Models

We just learnt how to launch GUI Experiments from GAMA. A GUI Experiment will start with a particular set of input, compute several outputs, and will stop at the end (if asked).

In order to explore models (by automatically running the Experiment using several configurations to analyze the outputs), a first approach is to run several simulations from the same experiment, considering each simulation as an agent. A second approach, much more efficient for larger explorations, is to run an other type of experiment : the **Batch Experiment**.

We will start this part by learning how to **run several simulations** from the same experiment. Then, we will see how **batch experiments** work, and we will focus on how to use those batch experiments to explore models by using **exploration methods**.



Version: 1.9.1

Run Several Simulations

To explore a model, the easiest and the most intuitive way to proceed is running several simulations with several parameter value, and see the differences from the output. GAMA provides you the possibility to launch several simulations from the GUI.

Create a simulation

Let's remind you that in GAMA, everything is an **agent**. We already saw that the "**world**" **agent** is the **agent of the model**. The model is thus a **species**, called modelName_model :

```
model toto // <- the name of the species is "toto_model"
```

New highlight of the day : an **Experiment** is also an agent ! It's a special agent which will instantiate automatically an agent from the model species. You can then perfectly create agents (*model* agents) from your experiment, using the statement **create** :

```
model multi_simulations // the "world" is an instance of the
"multi_simulations_model"

global {
}

experiment my_experiment type:gui {
```

This sort model will instantiate 2 simulations (two instance of the model) : one is created automatically by the experiment, and the second one is explicitly created through the statement `create`.

To simplify the syntax, you can use the built-in attribute `simulation` of your **experiment**. When you have a model called "multi_simulations", the two following lines are strictly equal :

```
create multi_simulations_model;
create simulation;
```

As it was the case for creating regular species, you can specify the parameters of your agent during the creation through the facet `with:` :

```
model multi_simulations

global {
    rgb bgd_color;
}

experiment my_experiment type:gui {
    parameter name:"background color:" var:bgd_color init:#blue;
    init {
        create simulation with:[bgd_color::#red];
    }
    output {
        display "my_display" background:bgd_color{}
    }
}
```

Manipulate simulations

Generate simulations on-the-fly

When you think the simulations as agents, it gives you a lot of new possibilities. You can for example create a reflex from your experiment, asking to create simulations **during the experiment execution !**

The following short model for example will create a new simulation at each 10 cycles :

```
model multi_simulations

global {
    init {
        write "new simulation created ! Its name is "+name;
    }
}

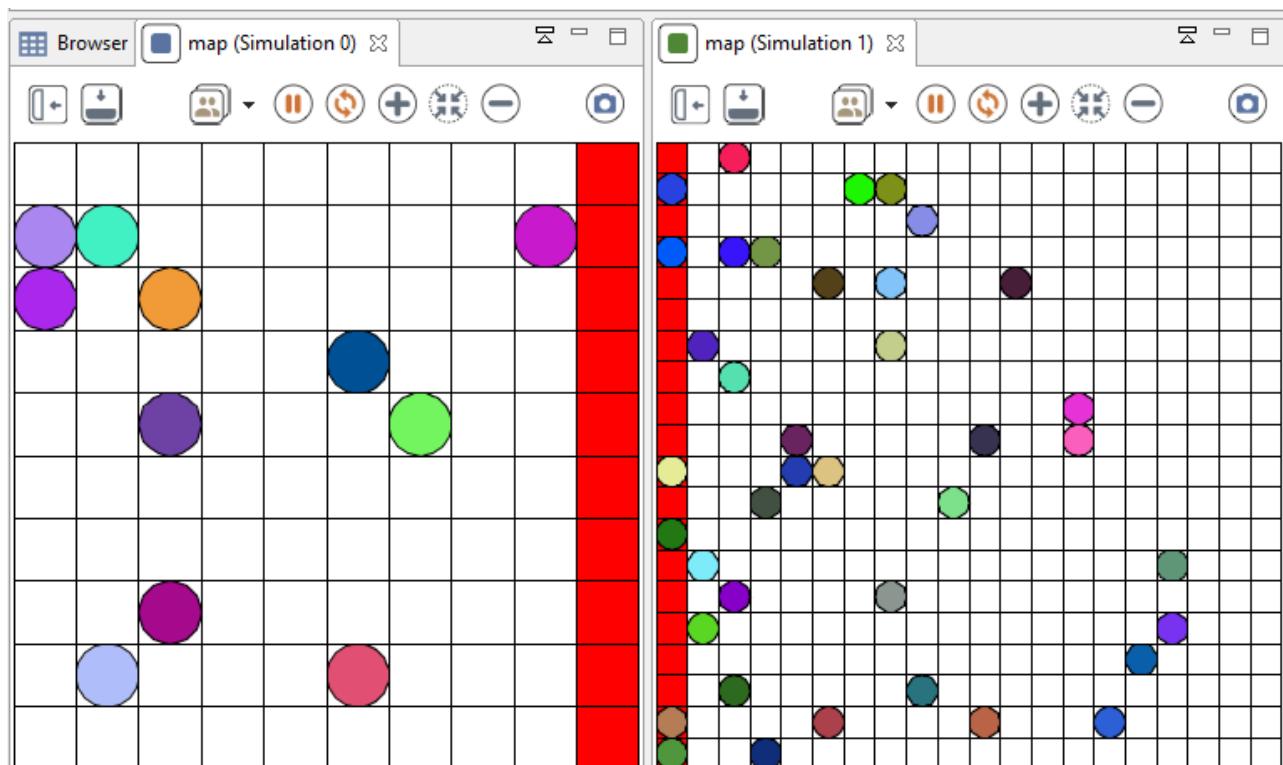
experiment my_experiment type:gui {
    init {}
    reflex when:(mod(cycle,10)=0 and cycle!=0) {
        create simulation;
    }
    output {}
}
```

You may ask, what is the purpose of such a thing ? Well, with such a short model, it is not very interesting, for sure. But you can imagine running a simulation, and if the simulation reaches a certain state, it can be closed, and another simulation can be

run instead with different parameters (a simulation can be closed by doing a "do die" on itself).

Communication between simulations

You can also imagine to run two simulations, and to communicate from one to another through the experiment, as it is shown in this easy model, where agents can move from one simulation to another :



```
model smallWorld

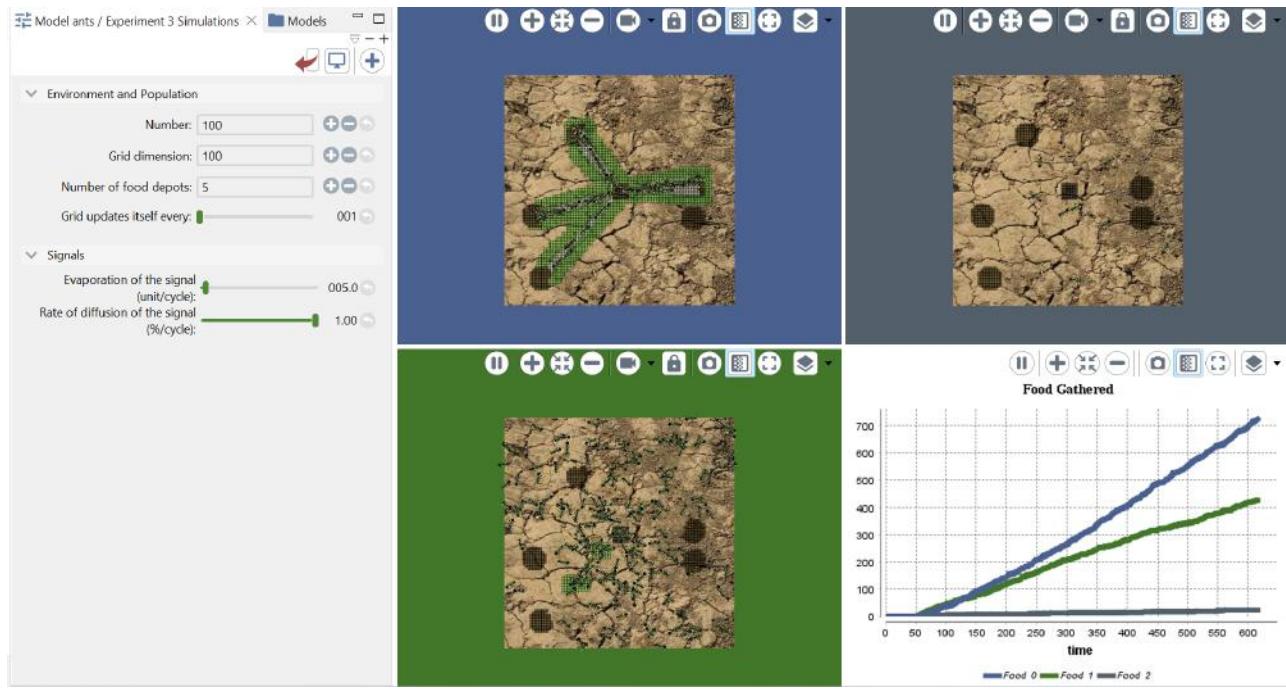
global {
    int grid_size <- 10;
    bool modelleft <- true;
    int id<- 0;
    int nb_agents <- 50;
```

For more complex communication example you can look into the example models from the `network` plugin> `TCP Server And Client Example .gaml` runs a first simulation that will create two servers (one of them being the lead server), then the experiment will create a new simulation with different parameters, and this simulation will itself create two clients, after this, clients and servers from both simulations will communicate together through the TCP protocol. The same behavior can be found in the `WebSocket Server And Client Example .gaml` model but with the use of `websocket` instead of TCP.

Going a step further, `TCP Teleportation.gaml` recreates the same behavior as the `Small world` example from the previous statement but this time the whole agent is serialized and de-serialized from one simulation to another then sent through TCP. And this time you can chain as many simulation as you want thanks to an experiment parameter. `1 Pong Teleportation (send agent).gaml` exhibits the same behavior but uses the `MQTT` protocol.

Permanent displays

Here is an other example of application available in the model library under the name `Ant Foraging.gaml` with the experiment `3 Simulations`. In this simulation, we run 3 times the Ant Foraging model, with different parameters and plot in a unique graph the evolution of the total gathered food by each model.



This graph is done thanks to the `permanent` block:

```
permanent {
    display Comparison background: #white {
        chart "Food Gathered" type: series {
            loop s over: simulations {
                data "Food " + int(s) value: s.food_gathered color:
s.color marker: false style: line thickness: 5;
            }
        }
    }
}
```

This block defines a display that is linked to the experiment only, it can thus access all the currently running simulations with the list `simulations`. From there you can define displays to summarize what's happening in all the simulations, display some statistics on them etc.

Random seed

Defining the seed from the model

If you run several simulations, you may want to use the same seed for each one of those simulations (to compare the influence of a certain parameter, in exactly the same conditions).

Let's remind you that `seed` is a built-in attribute of the model. You than just need to specify the value of your seed during the creation of the simulation if you want to fix the seed :

```
create simulation with:[seed::10.0];
```

You can also specify the seed if you are inside the `init` scope of your `global` agent.

```
global {
    init {
        seed<-10.0;
    }
}
```

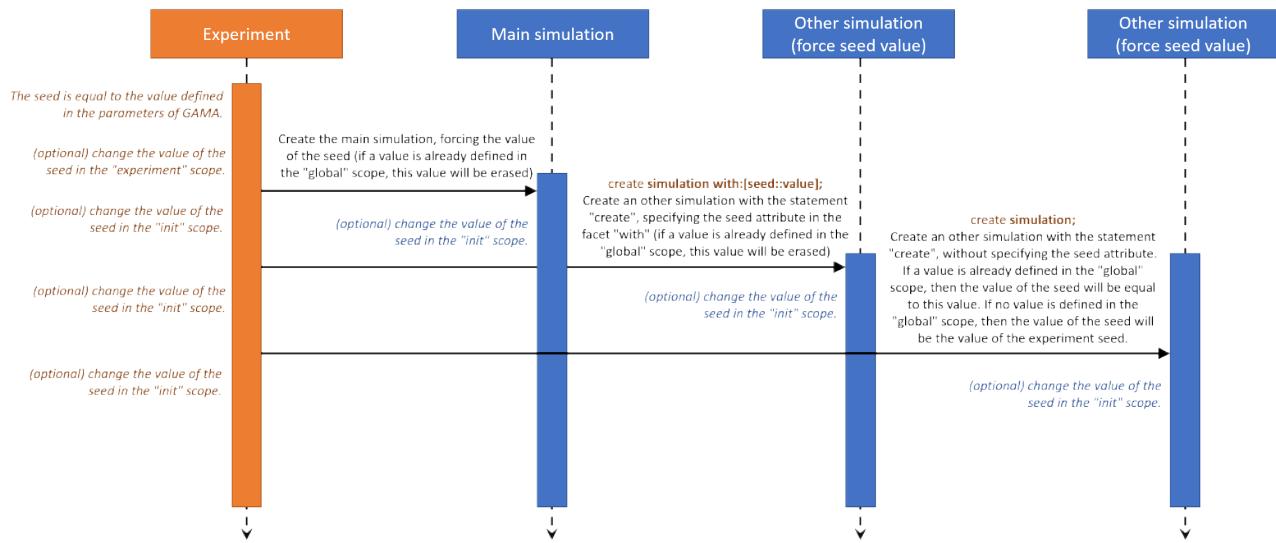
Notice that if you affect the value of your seed built-in directly in the global scope, the affection of the parameters (for instance specified with the facet `with` of the statement `create`), and the "init" will be done after will be done at the end.

Defining the seed from the experiment

The experiment agent also have a built-in attribute `seed`. The value of this seed is

defined in your [simulation preferences](#). The first simulation created is created **with the seed value of the experiment**.

The following sequence diagram can explain you better how the affectation of the seed attribute works :



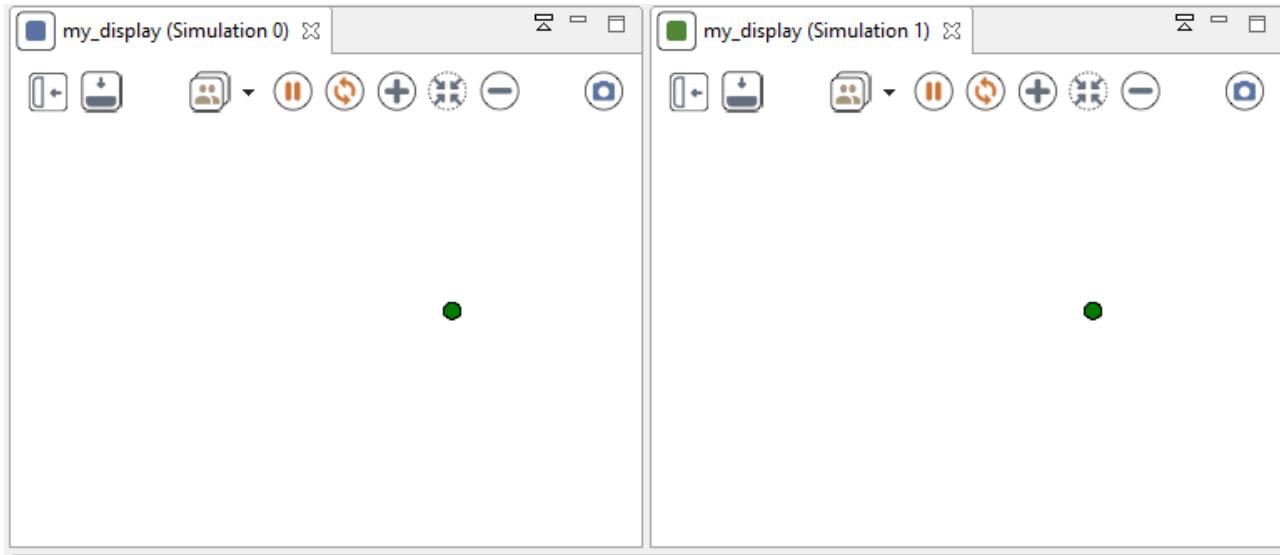
The affectation of an attribute is always done in this order : (1) the attribute is affected with a specific value in the species scope. If no attribute value is specified, the value is a default value. (2) if a value is specified for this attribute in the `create` statement, then the attribute value is affected again. (3) the attribute value can be changed again in the `init` scope.

Run several simulations with the same random numbers

The following code shows how to run several simulations with a specific seed, determined from the experiment agent :

```
model multi_simulations
```

When you run this simulation, their execution is exactly similar.



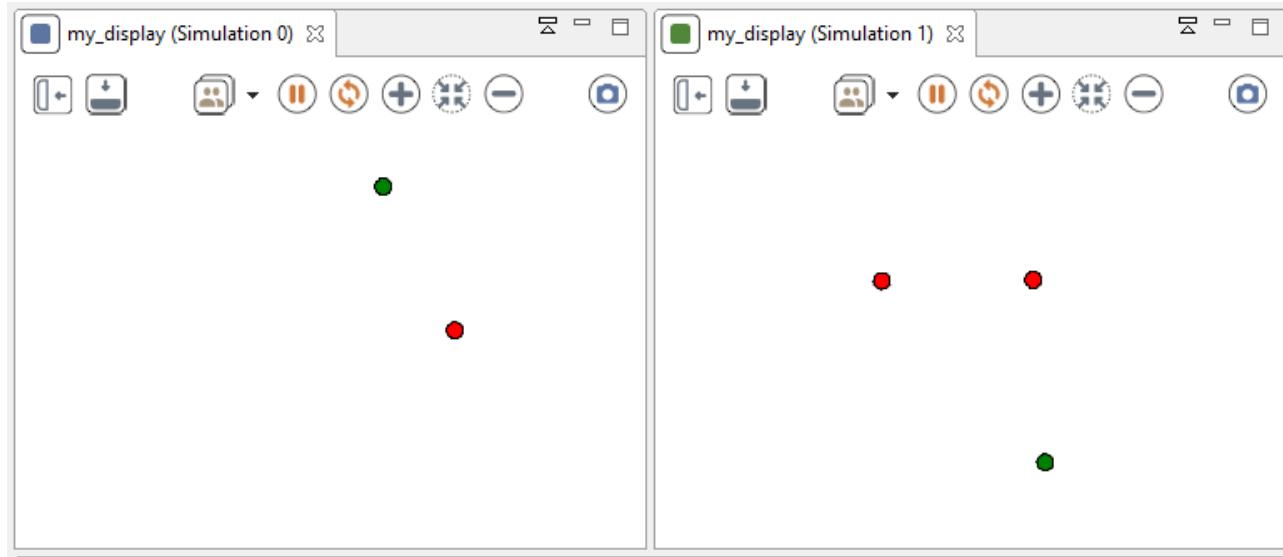
Let's try now to add a new species in this model, and to add a parameter to the simulation for the number of agents created for this species.

```
model multi_simulations

global {
    int number_of_speciesB <- 1;
    init {
        create my_speciesA;
        create my_speciesB number:number_of_speciesB;
    }
}

species my_speciesA skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#green;
    }
}
```

Then you run the experiment, you may find something strange...



Even if the first step seems ok (the green agent and one of the two red agent is initialized with the same location), the simulation differs completely. You should have expected to have the same behavior for the green agent in both of the simulation, but it is not the case. The explanation of this behavior is that a random number generator has generated more random numbers in the second simulation than in the first one.

If you don't understand, here is a short example that may help you to understand better :

```
model multi_simulations

global {
    int iteration_number <- 1;
    reflex update {
        float value;
        loop times:iteration_number {
            value<-rnd(10.0);
            write value;
        }
    }
}
```

The output will be something like that :

```
7.67003069780383
cycle 0 in experiment multi_simulations_model0 : 7.67003069780383
7.67003069780383
0.22889843360303863
cycle 0 in experiment multi_simulations_model1 : 0.22889843360303863
0.22889843360303863
cycle 1 in experiment multi_simulations_model0 : 0.22889843360303863
4.5220913306263855
0.8363180333035425
cycle 1 in experiment multi_simulations_model1 : 0.8363180333035425
4.5220913306263855
cycle 2 in experiment multi_simulations_model0 : 4.5220913306263855
5.460148568140819
4.158355846617511
cycle 2 in experiment multi_simulations_model1 : 4.158355846617511
0.8363180333035425
cycle 3 in experiment multi_simulations_model0 : 0.8363180333035425
1.886091659169562
4.371253083874633
cycle 3 in experiment multi_simulations_model1 : 4.371253083874633
```

Which means :

Cycle	Value generated in simulation 0	Value generated in simulation 1
1	7.67003069780383	7.67003069780383
		0.22889843360303863
2	0.22889843360303863	4.5220913306263855
		0.8363180333035425

Cycle	Value generated in simulation 0	Value generated in simulation 1
3	4.5220913306263855	5.460148568140819
		4.158355846617511

When writing your models, you have to be aware of this behavior. Remember that each simulation has its own random number generator.

Change the RNG

The RNG (random number generator) can also be changed : `rng` is a string built-in attribute of the experiment (and also of the model). You can choose among the following rng :

- mersenne (by default)
- cellular
- java

The following model shows how to run 4 simulations with the same seed but with some different RNG :

```
model multi_simulations

global {
    init {
        create my_species number:50;
    }
}

species my_species skills:[moving] {
    reflex update {
        do wander;
```




Version: 1.9.1

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. [Exploration methods are detailed in this page.](#)

A Batch experiment is defined by:

```
experiment exp_title type: batch until: condition {  
    [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 simulation. 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) Specifies the number of simulations replications for each

parameter configuration (a set of values assigned to the parameters). This means that several simulation will be run with the same parameter values, however a different random seed will be used for the pseudo-random number generator for each simulation. This allows to get some statistical power from the experiments conducted for stochastic models. The default value is 1.

- `keep_seed`: (boolean) If true, the same series of random seeds will be used from one parameter configuration to another. The default value is false.

```
experiment my_batch_experiment type: batch repeat: 5 keep_seed: true
until: (cycle = 300) {
    [parameter to explore]
    [exploration method]
}
```

Action `_step_` and reflexes

As for any species, `experiment` can define as many `reflex` as needed. In a `batch` experiment, they will be executed at the end of each bunch of simulations (set of replications) for a given parameters configuration. Note that at the experiment level, you have access to all the species and all the global variables and to all the simulations (variable `simulations`).

To be complete, each experiment (as any agent) will call at each step (i.e. the end of the replications set) the `_step_` action: this action is in charge of executing the behavior of the experiment agent, that is by default the execution of each of its `reflex`. It is possible to redefine the action `_step_`, but this should be used with care since this inhibits the reflexes.

For instance, the following experiment runs the simulation 5 times, and saves the people agents in a single shapefile at the end of the 5 simulations.

```

experiment 'Run 5 simulations' type: batch repeat: 5 keep_seed: true
until: ( time > 1000 ) {
    int cpt <- 0;

    reflex save_people {
        save people type:"shp" to:"people_shape" + cpt + ".shp" with:
        [is_infected::"INFECTED", is_immune::"IMMUNE"];
        cpt <- cpt + 1;
    }
}

```

The same can be done using the `action _step_ {` instead of `reflex save_people {`.

But if now we want to save information from the 5 simulations and save 1 shapefile per replication, we need to use the built-in attribute `simulations`. To save 1 shapefile per simulation run, we thus need to write:

```

experiment 'Run 5 simulations' type: batch repeat: 5 keep_seed: true
until: ( time > 1000 ) {
    reflex end_of_runs {
        int cpt <- 0;
        ask simulations {
            save people type: "shp" to: "result/people_shape" + cpt +
            ".shp" with: [is_infected::"INFECTED", is_immune::"IMMUNE"];
            cpt <- cpt + 1;
        }
    }
}

```

If now we want to save in a file aggregated values over the five simulations, such as the average number of infected people over the five simulations, we need to write:

```
experiment 'Run 5 simulations' type: batch repeat: 5 keep_seed: true
```

Permanent

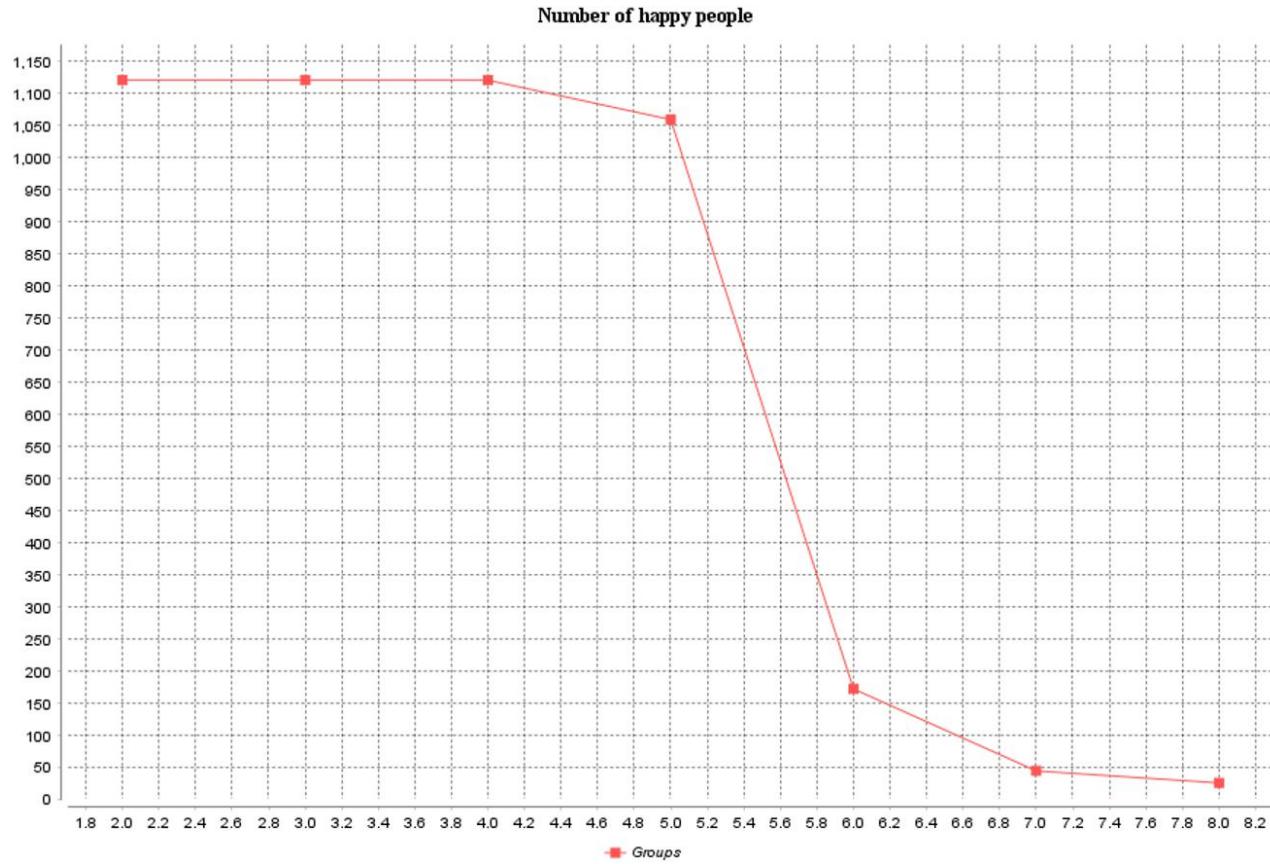
The `permanent` statement allows the modeler to define an output block that will not be re-initialized at the beginning of each simulation but will be filled instead at the end of each simulation. For instance, this `permanent` section will plot for each simulation the end value of the `food_gathered` variable (defined as a global variable in the model).

```
permanent {
    display Ants background: #white refresh: every(1#cycle) {
        chart "Food Gathered" type: series {
            data "Food" value: food_gathered;
        }
    }
}
```

It can be particularly useful when exploring values of parameters to plot their influence over some metric. For example here is a model based on Thomas Schelling's model of residential segregation (described in many library models) where we use batch to explore the parameter `number_of_groups`. For each value of this parameter we will run 10 simulations to mitigate the impact of randomness. We will then use the `permanent` statement to display the average happiness after 50 simulation steps in function of the number of groups in which the population is split:

```
/**
 * Name: NewModel
 * Based on the internal empty template.
 * Author: baptiste
 * Tags:
 */
```

Which gives us a result that looks something like this:



Parameter sets in parallel

There is an option in the `Preferences...` menu of Gama to allow multiple replications to be executed in parallel, that is to fully use assigned cores to computation. In that case, `permanent` and `reflex` blocks in the `experiment` will only be triggered once at the end of the whole set of simulations, rather than after each set of replications. Therefor, this option should only be used when doing none GUI batch experiment



Version: 1.9.1

Exploration calibration methods

Several methods are currently available in GAMA to explore and calibrate your simulation model.

Table of contents

- The `method` statement
- Exploration, analysis and calibration methods in a nutshell
- The exploration method
 - Exhaustive sampling: `factorial`
 - Random sampling: `uniform`
 - Latin Hypercube sampling: `latinhypercube`
 - Orthogonal sampling: `orthogonal`
- Analysis methods
 - Analysis of stochasticity: `stochanalyse`
 - Sobol analysis: `sobol`
 - Morris analysis: `morris`
 - Beta analysis: `betad`
- Calibration methods
 - Hill Climbing: `hill_climbing`
 - Simulated Annealing: `annealing`

- Tabu Search: `tabu`
- Reactive Tabu Search: `reactive_tabu`
- Genetic Algorithm: `genetic`
- Particle Swarm Optimization: `pso`

The `method` statement

The optional `method` statement controls the algorithm which drives the batch.

If this element is omitted, the batch will run an `exploration` method with default facets, see the [exploration section](#) for more details.

Examples of the use of `method` statement:

```
method exploration;
```

another examples with custom options

```
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";
```

Rationals behind using batch methods

Overall Gama provides three uses of batch `method` for [exploration](#), [analysis](#) and [calibration](#):

- The first type stands for classical exploration of simulation models by launching simulations for a given set of parameters. The purpose of such approach is to better understand the behavior of the model exploring different scenarios corresponding to a set of points in the parameter space. See the section dedicated to [exploration](#).
- The set of methods dedicated to analysis are meant to better understand how the model outputs are determined by stochastic processes and input parameters, what usually is referred to as Sensitivity Analysis. See the section dedicated to [analysis](#)
- The last set of methods are used to choose a satisfying set of parameter value to achieve as close as possible desired outputs, what usually is referred to as Calibration. See the section dedicated to [calibration](#)

Exploration methods

Exploration is the simplest and the most intuitive way to get a better understanding of the behavior of a model simulation across various input conditions. Basically it consists in launching simulations replicates (using facet `repeat`) for a parameter set (a vector of parameter values), retrieve outcomes, going to explore another parameter set, retrieve outcomes, and so on so force. The list or set of points in the parameter space to explore depend on the sampling algorithms used, via the `sampling` facet. Gama provides 6 different algorithms, `saltelli`, `morris`, `latinhypercube`, `orthogonal`, `uniform` and `factorial`, among which 4 are detailed in the sections below. Another aspect is the outcome of the exploration: it is up to the modelers to define, in the model or the experiment how outputs of the simulation should be saved. Most detailed outputs will be a record of any variable of interest step by step for each simulation, or an aggregate value over the course of a simulation for each simulation and at the most, only an aggregate value over each simulations replications. It depends on what you want to observe and how you want to statistically explore results of simulation. For state of the art exploration strategy

for agent-based simulation, see for instance Borgonovo et al., Sensitivity analysis of agent-based models: a new protocol. Comput Math Organ Theory 28, 52–94 (2022).

Generic method facets (i.e. parameters):

- `sample`: number of points to explore in the parameter space. Optional, when omitted default value is 124.
- `sampling`: the methods used to automatically draw points from the parameter space. Optional, when omitted default sampling is `factorial`
- `with`: the explicit list of points (map) to explore. Optional, when omitted no default value, when used the two previous facet are bypassed.
- `from`: the explicit list of points to explore, encoded in a csv file where the line corresponds to one point in the parameter space, each column a parameter value. Optional, when omitted no default value, when used the two previous facet are bypassed.

Examples:

When you want to explicitly define the points in the parameter space to explore, just put them (a point is a map of parameter name as key and parameter value as value) in a list

```
method exploration with: [
    ["parameter1":0.1, "parameter2": 0.01],
    ["parameter1":0.5, "parameter2": 0.2],
    ["parameter1":1.0, "parameter2": 0.05],
];
```

An alternative is to put the list of points in a csv file, where each lines will stands for a point in the parameter space, each column defining a parameter.

```
experiment exploration type: batch keep_seed:true until:( time > 5000
```

When no precise hypothesis is made on the configuration of the parameter space, Gama provides built-in exploration strategies using the `sampling` facet:

```
experiment exploration type: batch keep_seed:true until:( time > 5000
) repeat:40 {
    method exploration sample:100 sampling:uniform;
}
```

In the next sub-sections we detail the various sampling methods modelers can use as exploration strategies.

Exhaustive exploration of the parameter space: factorial sampling

This is the default batch exploration sampling algorithm. It explores all the combination of parameter values in a sequential way.

Example:

```
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 parameters. 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)
- ...

random exploration of the parameter space: uniform sampling

Provides a quick and simple way to explore the parameter space, drawing uniformly each points. The drawing algorithm treat parameters in two distinctive ways: when `min` and `max` facet of the parameter is used, the value is drawn uniformly within the boundaries (included); when a list/set of values is defined, the value is drawn uniformly within this list/set. Each parameter value is treated independently. The algorithm select as many points as defined by the `sample` facet.

Example:

```
experiment exploration type: batch until:( time > 5000 ) repeat:40 {
    method exploration sample:100 sampling:uniform;
}
```

latin hypercube sampling

The procedure of Latin Hypercube Sampling (LHS) works upon a grid view of the parameter space: each parameter is divided into n slices, with n based on `sample` facet, making up a matrix with d dimensions, with d the number of parameters. At first iteration, the procedure select one slice per parameter, draw a value for each ones (within the boundaries of the slice) and exclude each selected slice; then procedure continue until no more slices remains. For a simple yet more precise definition, see the [wikipedia](#) page.

Example:

```
experiment exploration type: batch until:( time > 5000 ) repeat:40 {  
    method exploration sample:100 sampling:latinhypercube;  
}
```

orthogonal sampling

The procedure behind orthogonal sampling is an extension of LHS optimizing the density of points over the parameter space. It means that, based on the a grid view constraint, it will try to minimize the distance between each points. see the [wikipedia](#) page of LHS.

Example:

```
experiment exploration type: batch until:( time > 5000 ) repeat:40 {  
    method exploration sample:100 sampling:orthogonal;  
}
```

Analysis Methods

Analysis methods provide statistical insights into the understanding of simulation model behavior. Contrary to mere exploration procedure, analysis methods refer in Gama to as statistical analysis of outputs based on input parameters; that is, sensitivity analysis. Gama embeds two types of such analysis : one to gauge the [impact of stochasticity](#) and three other options to evaluate the contribution of parameters, i.e. [Sobol](#), [Morris](#) and [Beta_d](#).

The first method relies on statistical tests to evaluate how much simulations output generated with the exact same input parameter values are correlated. Consequently

modelers can decide, based on a correlation threshold of their choice, how many simulation replications they need to build coherent aggregated outputs (i.e. one that do not rely too much on stochastic discretion of results). More explanation is provided in the corresponding section. The other set of methods focuses on contribution of parameters over outputs variability.

Gama provides three different measurements of inputs contribution to outputs variability : the Sobol, Morris and Beta indexes. Each cover different aspect of output variability and provide various insight on parameters contribution. For instance Sobol depicts weights, whereas Morris proposes valence (i.e. positive or negative impacts) on parameters contributions. Beta index gains insight based on the general distribution of results, while the two other look at variance of results. More information to be found on corresponding sub-sections.

There is generic parameters of such methods, in particular the targeted `outputs` which are to be determined for all analysis methods. However `sobol` and `morris` methods have specific facets, while `stochanalyse` and `betad` don't.

Common facets of all analysis methods:

- `outputs`: the list of output variables to analyse through the chosen method. Mandatory.
- `report`: the path to the file where the results of the chosen method will be written. Mandatory.
- `results`: the path to the file where the variable designated in `outputs` and the corresponding parameter values (point) will be written. Optional, when omitted no raw result report will be written.

Stochastic Analysis: `stochanalyse`

This method embeds three different index to measure the impact of stochastic processes. For each of them, given thresholds make it possible to outline the number

of replicates required to satisfy statistical criteria such as p-value or student test. The three indicators are:

- student t test (see [Lee et al., 2015](#))
- standard error (see [Bobachev and Morris, 2010](#))
- coefficient of variation (see this [blog post](#) for neat concise explanation)

Each measure is made n times, one for each simulation point, with n the value of facet `sample`, and m replicates for each point, with m the value of facet experiment `repeat`. It means that, we provide n statistical tests, for which we have m instance.

The tests are conducted with 2 replicates, then 3, 4, etc. up to m ; repeated n times. Tests gives you correlation, error and variation indexes, for 2 to m replicates, with various build-in thresholds to decide if test is successful on not. Those tests are repeated n times to ensure that the test is not only processed on a particular configuration of the simulation, for example a point in the parameter space that emphasis or decrease stochastic processes.

Useful facets:

- `sample`: the number of point of the parameter space to build. Optional, when omitted default value is 1.
- `sampling`: the sampling algorithm used to draw `sample` number of points
- `from`: a file with corresponding points of scenarios that might have higher stochastic impact. Optional.
- `with`: a given list of points (map). Optional.

Exemple

```
experiment stoch type: batch until: time > end_cycle repeat:40
keep_simulations:false {
    method stochanalyse outputs:["num_dead"] results:"Results/
stochanalysis.txt" sample:3;
```

In this example, 40 x 3 simulations will be run; stochastic indexes will be computed over 3 different point in the parameter space and failure/success computed from 2 to 40 replicates.

Sobol Analysis: `sobol`

This is an implementation of the Sobol sensitivity analysis exploration. It is based on the implementation of the algorithm provided by <http://moeaframework.org> under the GPL GNU licence.

Rational behind the Sobol sensitivity analysis can be found in Saltelli article ([https://doi.org/10.1016/S0010-4655\(02\)00280-1](https://doi.org/10.1016/S0010-4655(02)00280-1)). To put it simple, the procedure randomly draw N x P points in the parameter space (with N defined by the `sample` parameter and P the number of parameters to explore), execute the set of associated simulation and compute first, second and total ordered sensitivity indexes.

Intuitively, those values give an estimated contribution of the parameters to the variability of one or several outputs (the `outputs` list in parameter of the method).

Useful facet:

- `sample`, the size of the sample N for the sobol sequence. Mandatory.

Example:

```
experiment Sobol type: batch keep_seed:true until:( time > 5000 ) {
    method sobol outputs:["num_dead"] sample:100 report:"Results/
    sobol.txt" results:"Results/sobol_raw.csv";
}
```

Morris Analysis: `morris`

The corresponding method is an implementation of the Morris sensitivity analysis

exploration. It is a Java re-implementation of the R package [sensitivity](#) and proposes two main feature : the sampling method and a sensitivity analysis, both attached to the statement `morris` (both as the name of an analysis method and a keyword to define a sampling method).

Rational behind the Morris sensitivity analysis can be found in the [Morris's seminal article](#); more resources (including open access ones) can be found [here](#). Compared to Sobol analysis, Morris does not include interaction effect between parameters but provide a positive/negative valence to the impact of parameters in addition to the magnitude of the contribution. In short, Morris analysis the contribution of parameter one factor at a time (OFAT), making it possible to elicit most impactful parameters, also giving insight on the direction of contribution of parameters, that is increasing the value of one parameter with a negative Morris index, will impact negatively the value of the outputs of interest.

Useful facet:

- `sample`: the size of the sample for the sobol sequence. Mandatory.
- `level`: the number of times the Morris sampling will slightly change the value of parameters to evaluate sensitivity of outputs. Optional, when omitted default value is 4

Example:

```
experiment Morris type: batch keep_seed:true until:( time > 5000 ) {  
    method morris outputs:["num_dead"] sample:100 level:4  
    report:"Results/morris.txt" results:"Results/morris_raw.csv";  
}
```

Beta Analysis: betad

This method corresponds to the proposed sensitivity analysis by [Baucells and](#)

Borgonovo. Contrary to both Sobol and Morris SA methods, `betad` is not based on results variance but rely on the global distribution of results; which means that it includes full interactions between parameters in the assessment of one factor impact on the whole distribution of outputs. To put it simply, beta distribution evaluates how much variation on a parameter input will increase the maximum absolute differences

Most important contribution of this method is to provide a third evaluation which may push or hinder results from Morris and Sobol. Used together, the three method provides solid understanding of input contribution over outputs of interests.

Useful facet:

- `sample`: the size of the sample for the sobol sequence. Mandatory.
- `sampling`: contrary to sobol and morris, no particular sampling algorithm had been defined to fit `beta^d` index computation. Optional, when omitted default is factorial.

Example:

```
experiment Morris type: batch keep_seed:true until:( time > 5000 ) {  
    method betad outputs:["num_dead"] sample:100 sampling:orthogonal  
    report:"Results/betad.txt" results:"Results/beta_raw.csv";  
}
```

Calibration Methods

Calibration is the third exploration feature of the batch experiments. Conversely to previous exploration methods, calibration will try to find the best combination of inputs to obtain best matching for desired outputs. In short, calibration is an optimization of input parameter values in order to be as close as possible to desired outputs. To elicit how close the input allows to be relatively to requested output, the

algorithm uses a fitness measure, that is a variable, defined in the model, which evaluate the 'distance' between a specific simulation outputs and the requested one. It is up to modelers to define this variable within the model keeping in mind that the fitness could be minimize (with 0 a perfect match) or maximized (can always be improved).

Specific facets dedicated to calibration methods are:

- `minimize` or `maximize` (mandatory for optimization methods): a facet defining the expression to be optimized.
- `aggregation` (optional): the possible values are `min` or `max` (string). 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 aggregated fitness can be turned to the minimum or the maximum of the obtained fitness values using this facet.
- other parameters that are specific to the exploration method (optional): see below for a description of these facets.

Hill Climbing: `hill_climbing`

This algorithm is an implementation of the Hill Climbing algorithm. [See the Wikipedia article for a more detailed explanation](#). This is a local search method that tries at each step, given a solution `s`, to find a solution `s'` in the neighborhood of `s` that increases (or decreases depending on the aim of the exploration) the fitness. This method is more efficient than the global exploration to find an optimum, but with the risk of finding a local optimum, whereas a global optimum could exist.

Algorithm:

```
Initialization of an initial solution s
iter = 0
```

Method facets (i.e. parameters):

- `iter_max`: number of iterations before stopping the exploration.

Example:

```
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: annealing

This algorithm is an implementation of the Simulated Annealing algorithm. See the [Wikipedia article](#) for more details. This is a global search method able to find an approximation of a global optimum. The idea is close to the one of slow cooling: given a solution, the algorithm will look for a better one in its neighborhood. This size of the neighborhood (represented by the temperature) will decrease over the execution of the algorithm.

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
```

Method facets (i.e. 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:

```
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: `tabu`

This algorithm is an implementation of the Tabu Search algorithm. [See the Wikipedia article](#) for more details. This is a local search method. To avoid the issue of local optimum, two additional principals are added: (i) *worsening*, i.e. the algorithm can sometimes choose a worse solution, (ii) *prohibitions*, i.e. solutions that have already been explored will become **tabu** in order to avoid that the algorithm considers them repeatedly.

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 facets (i.e. parameters):

- `iter_max`: number of iterations.
- `tabu_list_size`: size of the tabu list.

Example:

```
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: `reactive_tabu`

This algorithm is a simple implementation of the Reactive Tabu Search algorithm (Battiti et al., 1993). This Reactive Tabu Search is an enhanced 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_ini`: 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.

Example:

```
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
```

Genetic Algorithm: `genetic`

This is a simple implementation of Genetic Algorithms (GA). [See the Wikipedia article](#) for more details. 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 to this selection. Most of them are based on 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 equal 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 equal 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 mutate 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 facets (i.e. 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.

Example:

```
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;  
}
```

Particle Swarm Optimization: `pso`

This is an implementation of the Partical Swarm Optimization algorithme (PSO). See the [Wikipedia article](#) for more details. It solves a problem by having a population of candidate solutions, here dubbed particles, and moving these particles around in the search-space according to simple mathematical formula over the particle's position and velocity. Each particle's movement is influenced by its local best known position, but is also guided toward the best known positions in the search-space, which are

updated as better positions are found by other particles. This is expected to move the swarm toward the best solutions.

Method facets (i.e. parameters):

- `iter_max`, number of iterations.
- `num_particles`, number of particles.
- `weight_cognitive`, weight for the cognitive component.
- `weight_inertia`, weight for the inertia component.
- `weight_social`, weight for the social component.

Example:

```
experiment PSO type: batch keep_seed: true repeat: 3 until: ( time > 5000 ) {  
    parameter 'Infection rate' var: infection_rate min: 0.1 max:0.5 step:0.01;  
    parameter 'Probability of dying:' var: dying_proba min: 0.01 max: 0.2 step:0.01;  
    method pso num_particles: 3 weight_inertia:0.7 weight_cognitive: 1.5 weight_social: 1.5 iter_max: 5 minimize: num_dead ;  
}
```

Version: 1.9.1

Optimizing Models

Now you are becoming more comfortable with GAML, it is time to think about how the runtime works, to be able to run some more optimized models. Indeed, if you already tried to write some models by yourself using GAML, you could have noticed that the execution time depends a lot of how you implemented your model!

We will first present you in this part some **runtime concepts** (and present you the species facet `scheduler`), and we will then show you some **tips to optimize your models** (how to increase performances using `scheduler`, grids, displays and how to choose your operators).



Version: 1.9.1

Runtime Concepts

When a model is being simulated, a number of algorithms are applied, for instance, to determine the order in which to run the different agents, or the order in which the initialization of agents is performed, etc. This section details some of them, which can be important when building models and understanding how they will be effectively simulated.

Table of contents

- [Simulation initialization](#)
- [Agents Creation](#)
- [Agents Step](#)
- [Schedule Agents](#)

Simulation initialization

Once the user launches an experiment, GAMA starts by creating an experiment agent that will manage the initialization of the simulation(s). For each simulation, first, it creates a [world](#) agent.

It initializes all its attributes with their init values. This includes its [shape](#) (that will be used as the environment of the simulation).

If a species of type [grid](#) exists in the model, agents of this species are created.

Finally, the `init` statement of the `global` is executed. It should include the creation of all the other agents of [regular species](#) of the simulation. After their creation and initialization, they are added in the list `members` the `world` (that contains all the micro-agent of the `world`).

Agents Creation

Except `grid` agents, other agents are created using the `create` statement. It is used to allocate memory for each agent and to initialize all its attributes.

If no explicit initialization exists for an attribute, it will get the default value corresponding to its [type](#).

The initialization of an attribute can be located at several places in the code; they are executed in the following order (which means that, if several ways are used, the attribute will finally have the value of the last applied one):

- in the attribute declaration, using the `init` or `<-` facet.
- using the `from:` or `with` facet of the `create` statement.
- in the `init` block of the species.
- in the embedded block of the `create` statement.

Agents Step

When an agent is asked to *step*, it means that it is expected to:

- update its variables (facet `update` in the variable declaration),
- run its behaviors (reflex, state...),
- *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...
    }
}

```

Notice that, using architecture to manage the behavior of agents, is only a possibility provided by GAMA to ease the development of a model. Modelers who need precise control on the agents' step can:

- redefine the `_step_` action of the species, in order to explicit how the agents will behave,
- implement no behavior in the species (but only `action`). The execution of agents can thus be controlled from a reflex of the `global` that can control the execution of each of them.

Schedule Agents

The global scheduling of agents is then simply the application of this previous *step* to the *experiment agent*, keeping in mind that this agent has only one micro-population (of simulation agents, each instance of the model species), and that the simulation(s) inside this population contain(s), in turn, all the "regular" populations of agents of the model.

To influence this schedule, then, one possible way is to change the way populations compute their lists of agents to schedule, which can be done in a model by providing custom definitions to the `schedules` facet of one or several species.

A practical application of this facet is to reduce simulation artifacts created by the default scheduling of populations, which is sequential (i.e. their agents are executed in turn in their order of creation). To enable 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 completely random scheduling, that will eliminate the sequential scheduling of populations, by defining a custom species acting as a scheduler of the agents (that will be executed after the `world` agent):

```
global {...}

species scheduler schedules: shuffle(A + B + C);

species A schedules: [] {...}
species B schedules: [] {...}
species C schedules: [] {...}
```

It is important to suppress the population-based scheduling to avoid having agents being scheduled 2 times (one time in the custom definition, one time by their population). Note that it is not necessary to create a scheduler agent.

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 {...}

species scheduler schedules: 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 at 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 definition 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'
```

Note that `schedules` facet will not be taken into account when it is added to the `global`. It is thus not possible to unschedule the `world` agent.

Version: 1.9.1

Optimizing Models

This page aims at presenting some tips to optimize the memory footprint or the execution time of a model in GAMA.

Note: since GAMA 1.6.1, some optimizations have become obsolete because they have been included in the compiler. They have, then, been removed from this page. For instance, writing 'rgb(0,0,0)' is now compiled directly as '#black'.

Table of contents

- [Benchmarking](#)
- [Scheduling](#)
- [Grid](#)
 - [Optimization Facets](#)
 - [use_regular_agents](#)
 - [use_individual_shapes](#)
- [Operators](#)
 - [List operators](#)
 - [first_with](#)
 - [where / count](#)
 - [Spatial operators](#)
 - [container of agents in closest_to, at_distance, overlapping, inside](#)
 - [Accelerate with a first spatial filtering](#)
- [Displays](#)
 - [shape](#)

- circle vs square / sphere vs cube
- OpenGL refresh facets

Benchmarking

In order to optimize a model, it is important to exactly know which part of the model take times. It is thus possible to benchmark the global execution of the model using the `benchmark` facet of the `experiment` statement: it will produce a global report on the number of times any keyword has been used and how long has been spent to execute them.

```
experiment Benchmarking type: gui benchmark: true { }
```

`benchmark` is also a statement that can be used on a block of codes to evaluate its execution. In addition, it provides the possibility to run the block of code several times to get more accurate results.

```
global {
    init {
        create people number: 300;
    }

    reflex neighbourhood {
        benchmark "Benchmark of closest_to operator" repeat: 100 {
            ask people {
                do get_closest_people;
            }
        }
    }
}
```

Finally the manual way to evaluate the execution of a code could be using the

`machine_time` built-in global variable that gives the current time in milliseconds.

Then to compute the time taken by a statement, a possible way is to write:

```
float t <- machine_time;  
// here a block of instructions that you consider as "critical"  
// ...  
write "duration of the last instructions: " + (machine_time - t);
```

Scheduling

If you have a species of agents that, once created, are not supposed to do anything more (i.e. no behavior, no reflex, their actions triggered by other agents, their attributes being simply read and written by other agents), such as a "data" grid, or agents representing a "background" (from a shape file, etc.), consider using the `schedules: []` facet on the definition of their species. This trick allows to tell GAMA to not schedule any of these agents.

```
grid my_grid height: 100 width: 100 schedules: [] {  
    ...  
}
```

The `schedules` facet is dynamically computed (even if the agents are not scheduled), so, if you happen to define agents that only need to be scheduled every x cycles, or depending on a condition, you can also write `schedules` to implement this. For instance, the following species will see its instances scheduled every 10 steps and only if a certain condition is met:

```
species my_species schedules: (every 10) ? (condition ? my_species :  
[]) : [] {  
    ...
```

In the same way, modelers can use the frequency facet to define when the agents of a species are going to be activated. By setting this facet to 0, the agents are never activated.

```
species my_species frequency: 0 {  
    ...  
}
```

Grid

Optimization Facets

In this section, we present some facets that allow to optimize the use of grid (in particular in terms of memories). Note that all these facet can be combined (see the Life model from the Models library).

use_regular_agents

If false, then a special class of agents is used. This special class of agents used less memories but has some limitation: the agents cannot inherit from a "normal" species, they cannot have sub-populations, their name cannot be modified, etc.

```
grid cell width: 50 height: 50 use_regular_agents: false ;
```

use_individual_shapes

If false, then only one geometry is used for all agents. This facet allows to gain a lot of memory, but should not be used if the geometries of the agents are often activated (for instance, by an aspect).

```
grid cell width: 50 height: 50 use_individual_shapes: false ;
```

Parallel execution

The `grid` statement can also specify whether the agents of the grid are computed in parallel, using the facet `parallel`. This could increase (depending on the computation) the execution time.

Operators

List operators

`first_with`

It is sometimes necessary to randomly select an element of a list that verifies a given condition. Many modelers use the `one_of` and the `where` operators to do this:

```
bug one_big_bug <- one_of (bug where (each.size > 10));
```

Whereas it is often more optimized to use the `shuffle` operator to shuffle the list, then the `first_with` operator to select the first element that verifies the condition:

```
bug one_big_bug <- shuffle(bug) first_with (each.size > 10);
```

`where / count`

It is quite common to want to count the number of elements of a list or a container that verify a condition. The obvious to do it is:

```
int n <- length(my_container where (each.size > 10));
```

This will however create an intermediary list before counting it, and this operation can be time consuming if the number of elements is important. To alleviate this problem, GAMA includes an operator called **count** that will count the elements that verify the condition by iterating directly on the container (no useless list created):

```
int n <- my_container count (each.size > 10);
```

Spatial operators

container of agents in **closest_to**, **at_distance**, **overlapping**, **inside**

Several spatial query operators (such as **closest_to**, **at_distance**, **overlapping** or **inside**) allow to restrict the agents being queried to a container of agents. For instance, one can write:

```
agent closest_agent <- a_container_containing_agents closest_to self;
```

This expression is formally equivalent to :

```
agent closest_agent <- a_container_containing_agent with_min_of (each  
distance_to self);
```

But it is much faster **if your container is large**, as it will query the agents using a spatial index (instead of browsing through the whole container). Note that in some cases, when you have a small number of agents, the first syntax will be faster. The same applies to the other operators.

Now consider a very common case: you need to restrict the agents being queried,

not to a container, but to a species (which, actually, acts as a container in most cases). For instance, you want to know which predator is the closest to the current agent. If we apply the pattern above, we would write:

```
predator closest_predator <- predator with_min_of (each distance_to  
self);
```

or

```
predator closest_predator <- list(predator) closest_to self;
```

But these two operators can be painfully slow if your species has many instances (even in the second form). In that case, always prefer using **directly** the species as the left member:

```
predator closest_ predator <- predator closest_to self;
```

Not only is the syntax clearer, but the speed gain can be phenomenal because, in that case, the list of instances is not used (we just check if the agent is an instance of the left species).

However, what happens if one wants to query instances belonging to 2 or more species? If we follow our reasoning, the immediate way to write it would be (if predator 1 and predator 2 are two species):

```
agent closest_agent <- (list(predator1) + list(predator2)) closest_to  
self;
```

or, more simply:

```
agent closest_agent <- (predator1 + predator2) closest_to self;
```

The first syntax suffers from the same problem than the previous syntax: GAMA has to browse through the list (created by the concatenation of the species populations) to filter agents. The solution, then, is again to use directly the species, as GAMA is clever enough to create a temporary "fake" population out of the concatenation of several species, which can be used exactly like a list of agents, but provides the advantages of a species population (no iteration made during filtering).

Accelerate `closest_to` with a first spatial filtering

The `closest_to` operator can sometimes be slow if numerous agents are concerned by this query. If the modeler is just interested in a small subset of agents, it is possible to apply a first spatial filtering on the agent list by using the `at_distance` operator.

For example, if the modeler wants first to do a spatial filtering of 10m:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self;
```

To be sure to find an agent, the modeler can use a test statement:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self;
if (closest_agent = nil) {closest_agent <- predator1 closest_to
self;}
```

Displays

shape

It is quite common to want to display an agent as a circle or a square. A common mistake is to mix up the shape to draw and the geometry of the agent in the model. If the modeler just wants to display a particular shape, he/she should not modify the agent geometry (i.e. its `shape` attribute, 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

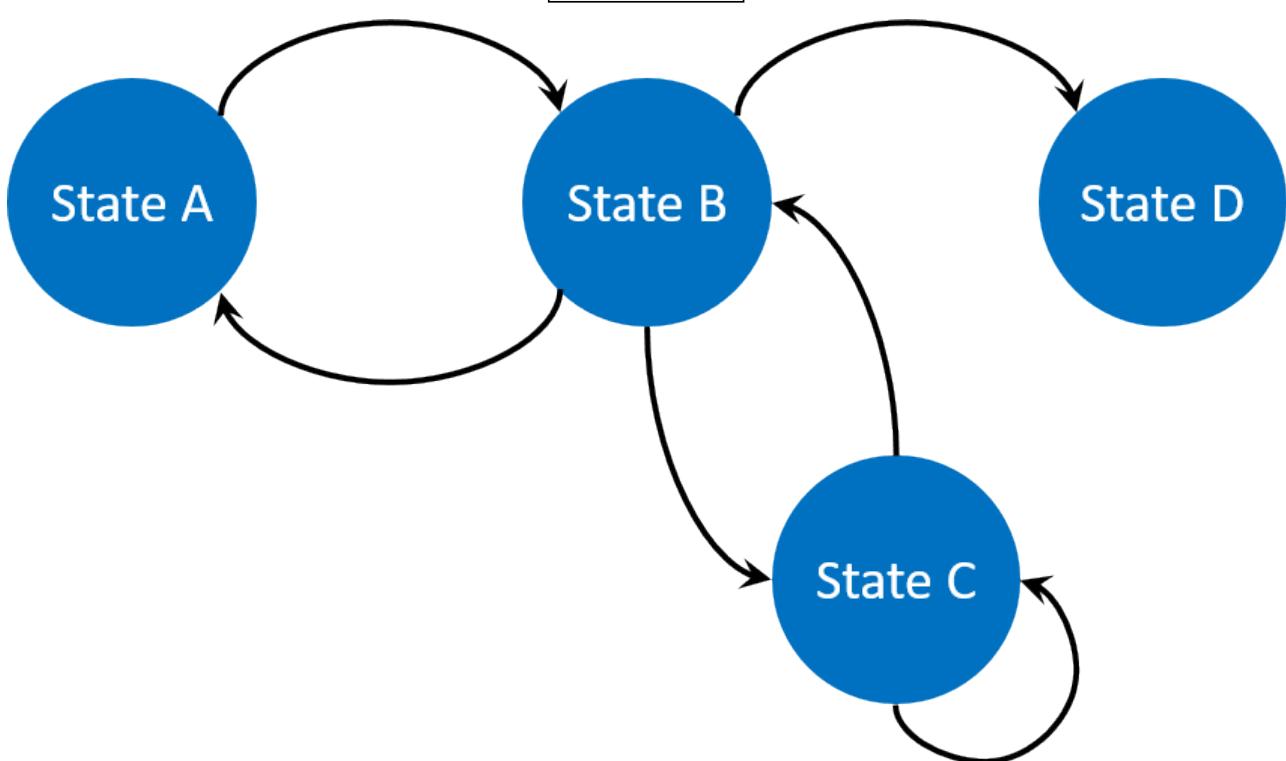
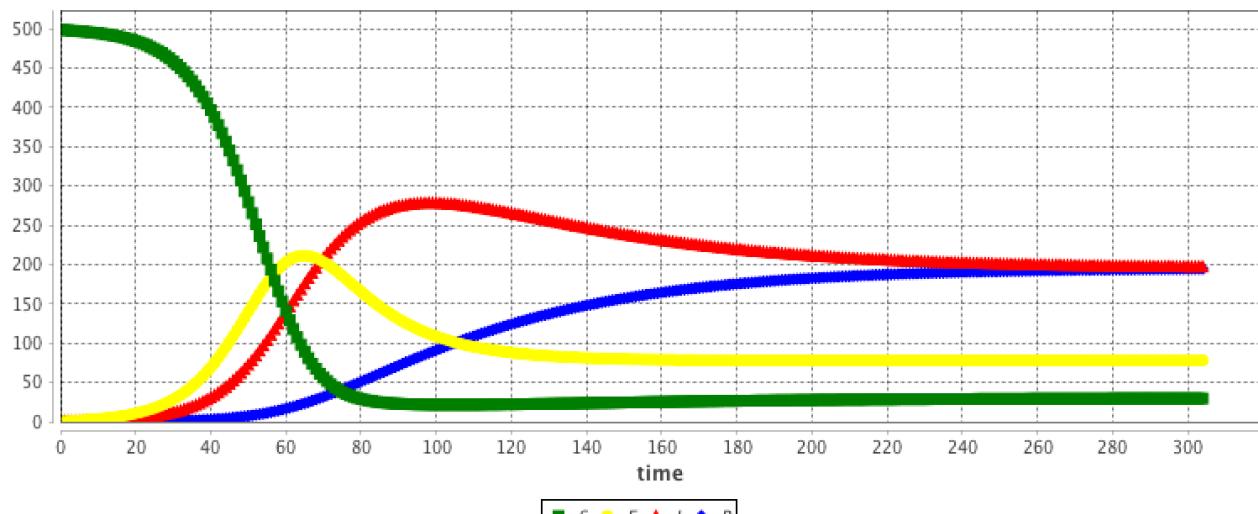
In OpenGL display, it is possible to specify that it is not necessary to refresh a layer

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;
}
```

Version: 1.9.1

Multi-Paradigm Modeling



Multi-paradigm modeling is a research field focused on how to define a model semantically. From the beginning of this step by step tutorial, our approach is based on [behavior](#) (or reflex), for each agents. In this part, we will see that GAMA provides other ways to implement your model, using several control architectures. Sometime, it will be easier to implement your models choosing other paradigms.

In a first part, we will see how to use some [control architectures](#) which already exist in GAML, such as [finite state machine architecture](#), [task based architecture](#) or [user control architecture](#). In a second part, we will see an approach based on mathematical modeling through use of [differential equations](#).



Version: 1.9.1

Control Architectures

GAMA allows the modeler to attach built-in control architecture to agents.

These control architectures will give the possibility to the modeler to use for a species a specific control architecture in addition to the [common behavior structure](#). Note that only one control architecture can be used per species.

The attachment of a control architecture to a species is done through the facets [control](#).

For example, the given code attaches the `fsm` control architecture to the dummy species.

```
species dummy control: fsm {  
}
```

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

- [fsm](#): finite state machine based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is in one single state. Such an agent needs to have one initial state (the state in which it will be at its initialization)
- [weighted_tasks](#): task-based control architecture. At any given time, only the task 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.
- [rules](#): rules-based control architecture. This architecture uses a set of rules, that will be executed if a given condition is fulfilled and in an order defined by a priority.
- [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.

An [exhaustive list of the architectures available with GAMA](#) provides all the variables and additional actions provided by an architecture.

Index

- [Finite State Machine](#)
 - [Declaration](#)
 - [state statement](#)
- [Task Based](#)
 - [Declaration](#)
 - [task](#)
- [User Control Architecture](#)
 - [user_only, user_first, user_last](#)

- Additional attribute
- user_panel
- Other Control Architectures

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 executes 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 statement

Facets

- `initial`: a boolean expression, indicates the initial state of the agent (only one state with `initial` set to true is allowed in a species).
- `final`: a boolean expression, indicates the final state of the 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 statecharts: 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";
    }
}
```

The dummy agents start at *state1*. At each simulation step, they have a probability of 0.5 to change their state to *state2*. If they do not change their state to *state2*, they have a probability of 0.2 to change their state to *state3*. In *state2*, at each simulation step, they have a probability of 0.5 to change their state to *state1*. At last, in *step3*, at each simulation step, they have a probability of 0.5 to change their state to *state1*. If they do not change their state to *state1*, they have a probability of 0.2 to change their state to *state2*.

Here a possible result that can be obtained with one dummy agent:

```
0:dummy0->state1
0:dummy0->transition to state1
1:dummy0->state2
2:dummy0->state2
2:dummy0->leave state2
2:dummy0->transition to state1
3:dummy0->state1
3:dummy0->transition to state1
4:dummy0->state2
5:dummy0->state2
5:dummy0->leave state2
5:dummy0->transition to state1
6:dummy0->state1
7:dummy0->state3
8:dummy0->state2
```

Task-Based

GAMA integrated several **task-based** control architectures. Species can define any number of tasks within their body. At any given time, only one or several tasks are executed according to the architecture chosen:

- `weighted_tasks`: in this architecture, only the task with the maximal weight is

executed.

- `sorted_tasks`: in this architecture, the tasks are all executed in the order specified by their weights (biggest first)
- `probabilistic_tasks`: this architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen each step.

Declaration

Using one of the task architectures for a species requires to use the **control** facet:

```
species dummy control: weighted_tasks {  
    ...  
}
```

```
species dummy control: sorted_tasks {  
    ...  
}
```

```
species dummy control: probabilistic_tasks {  
    ...  
}
```

task statement

Facets

Besides a sequence of statements like `reflex`, a task contains the following additional facet:

- **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
```

Rules-based architecture

The behavior of an agent with the rules-based architecture can contain `reflex` and `rule` statements. The `reflex` block will always be executed first. Then the rules are fired (executed) when their condition becomes true and in the order defined by their decreasing priorities.

Declaration

Using the rules-based architectures for a species requires to use the `control` facet:

```
species dummy control: rules {  
    ...  
}
```

rule statement

facets

- `when`: (boolean), the condition that needs to be fulfilled to execute the rule.
- `priority`: (float), an optional priority for the rule, which is used to sort activable rules and run them in that order.

Definition

As `reflex`, a `rule` is a sequence of statements that can be executed, at each time step, by the agent. They are executed if and only if their condition expression (`when` facet) is fulfilled. Among all the rules that fulfill their condition, the tasks are executed in the decreasing order of their priority (`priority` facet).

For example, consider the following example:

```
species simple_rules_statements control: rules {  
  
    int priority_of_a <- 0 update: rnd(100);  
    int priority_of_b <- 0 update: rnd(100);  
  
    reflex show_priorities {  
        write "Priority of rule a = " + priority_of_a + ", priority of  
rule b = " + priority_of_b;  
    }  
  
    rule a when: priority_of_a < 50 priority: priority_of_a {  
        write "  Rule a fired with priority: " + priority_of_a;  
    }  
  
    rule b when: priority_of_b > 25 priority: priority_of_b {  
        write "  Rule b fired with priority: " + priority_of_b;  
    }  
}
```

At each simulation step, first, the agents update the priority values associated with the rules. The `reflex` will first display these values. Then the conditions are evaluated and the rules that can be executed are executed in their priority order.

Here a possible result:

```
Priority of rule a = 38, priority of rule b = 32  
  Rule a fired with priority: 38  
  Rule b fired with priority: 32  
Priority of rule a = 91, priority of rule b = 32  
  Rule b fired with priority: 32  
Priority of rule a = 37, priority of rule b = 2  
  Rule a fired with priority: 37  
Priority of rule a = 77, priority of rule b = 90  
  Rule b fired with priority: 90
```

User Control Architecture

`user_only`, `user_first`, `user_last`

A specific type of control architecture has been introduced to allow users to take control of an agent during the course of the simulation. When the user gets control of the agent, a control panel will appear in the interface. This architecture can be invoked using three different keywords: `user_only`, `user_first`, `user_last`.

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

If the control chosen is `user_first`, it means that the user-controlled panel is opened first, and then the agent has a chance to run its "own" behaviors (reflexes, essentially, or "init" in the case of a "user_init" panel). If the control chosen is `user_last`, it is the contrary.

Additional attribute

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.

`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 `state` or `reflex`. 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.

`user_command` 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.

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 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. (The full model is provided in the GAMA model library.)

```
species user control:user_only {
    user_panel default initial: true {
        transition to: "Basic Control" when: every (10 #cycles) and
!advanced_user_control;
        transition to: "Advanced Control" when: every(10 #cycles) and
advanced_user_control;
    }
}
```

The panel marked with the `initial: true` facet will be the one run first when the agent is supposed to run. If none is marked, the first panel (in their definition order) is chosen.

A special panel called `user_init` will be invoked only once when initializing the agent if it is defined. If no panel is described or if all panels are empty (i.e. no `user_command`), 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.

Other Control Architectures

Some other control architectures are available in additional plugins. For instance, [BDI \(Belief, desire, intention\) architecture](#) is available. Feel free to read about it if you want to learn more.

Do you need some other control architectures for your model? Feel free to make your suggestion to the team of developers through the [mailing list](#). Remember also that GAMA is an open-source platform, you can design your own control architecture easily. Go to the section [Community/contribute](#) if you want to jump into coding!

Using Differential Equations

Introduction

ODEs (Ordinary Differential Equations) models are often used in physics, chemistry, biology, ecology and epidemiology. They allow tracking continuous changes of a system, and offer the possibility of a mathematical analysis. The possibility to find a numerical solution (for a given Cauchy problem) of first order differential equations has been implemented in Gama.

In population dynamics, systems of ODEs are used to describe the macroscopic evolution over time of a population, which is usually split into several compartments. The state of the population is described by the number of individuals in each compartment. Each equation of the ODE system describes the evolution of the number of individuals in a compartment. In such an approach, individuals are not taken into account individually, with own features and behaviors. On the contrary, they are aggregated and only the population density is considered.

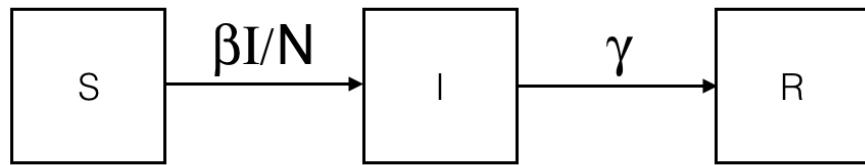
Compartmental models are widely used to represent the spread of a disease in a population, with a large variety of models derived from the classical Kermack-McKendrick model, often referred to as the SIR model. More information about compartmental models in epidemiology can be found [here](#).

In SIR class models, the population is split into 3 (or more) compartments: S (Susceptible), I (Infected), R (Recovered). It is not usually possible to find an analytical

solution of the ODE system, and an approximate solution has to be found, using various numerical schemes (such as Euler, Runge-Kutta, Dormand-Prince...)

Example of a SIR model

In the SIR model, the population is split into 3 compartments: S (Susceptible), I (Infected), R (Recovered). Susceptible individuals become infected (move from compartment S to I) at a rate proportional to the size of both S and I populations. People recover (are removed from compartment I) at a constant rate. This can be represented by the following Forrester diagram: boxes represent compartments and arrows are flows. Arrows hold the rate of a compartment population flowing to another compartment.



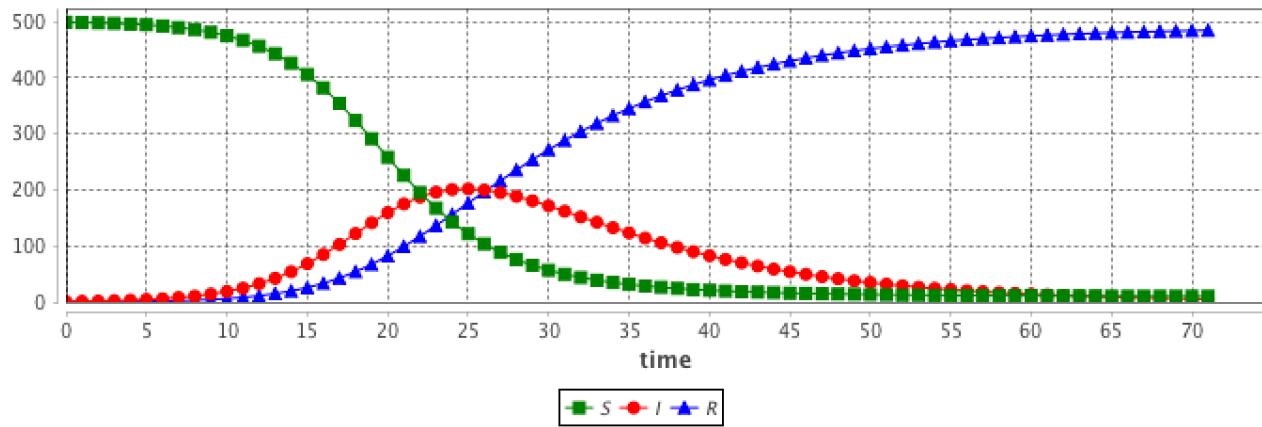
The corresponding ODE system contains one equation per compartment. For example, the I compartment evolution is influenced by an inner (so positive) flow from the S compartment and an outer (so negative) flow to the R compartment.

$$\left\{ \begin{array}{l} \frac{dS}{dt} = -\frac{\beta IS}{N} \\ \frac{dI}{dt} = \frac{\beta IS}{N} - \gamma I \\ \frac{dR}{dt} = \gamma I \end{array} \right.$$

Given an initial condition (initial values) at time t=0, such as

- S = 499
- I = 1
- R = 0
- beta = 0.4
- gamma = 0.1
- h = 0.1

one can obtain the evolution of the evolution of S, I and R over time, by integrating the ODE system using a numerical scheme.



Why and when can we use ODE in agent-based models?

ODE models are often used when a system can be considered at a macro (population) level, i.e when the individual variability has little influence on dynamics at the aggregated level.

It is relevant to use ODE in agent-based models in several cases:

- Large scale agent-based models require many resources to run and a very large computation time. For example, if we want to implement a model describing the worldwide epidemic spread and the impact of air traffic on it, we cannot simulate the 7 billion people. Instead we can represent only cities with airports and airplanes as agents. In this case, cities are entities represented by a compartment, on which a SIR class epidemiological model can be run, using an ODE system. Such a model combines some advantages of agent-based models (detailed description of the disease spread from one city to another with the plane agents) and mathematical modeling (good description of an epidemiological dynamics at a city level, using fewer resources and less computation time).

- Some processes may be easier to manipulate at the aggregated level, for several reasons: 1) a global description of a system may turn sometimes more informative than a detailed one, 2) a detailed description may require to fit too many parameters for which there is no sufficient data, in that case it is easier to fit a global model with less parameters, and 3) when one wants to keep a low number of parameters, in order to avoid overfitting or to optimize [Akaike Information Criterion](#).
- Some systems are better described with a continuous dynamics than with a discrete one. This is the case for many physical or biological systems (physics laws such as gravity or water dynamics, biological processes such as respiration or cell growth). Coupling ABM and ODE model allow considering individual/discrete processes along with continuous processes.
- Some models already exist in an ODE version, and could be coupled to another model in Gama without having to rewrite an Agent-Based version of the model.

Use of ODE in a GAML model

A stereotypical use of ODE in a GAMA agent-based model is to describe species where some agents attribute evolution is described using an ODE system.

As a consequence, the GAML language has been increased by two main concepts (as two statements):

- equations can be written with the `equation` statement. An `equation` block is composed of a set of `diff` statement describing the evolution of species attributes.
- an equation can be numerically integrated using the `solve` statement

Additionaly, Gama provides an intuitive, flexible and natural framework to build

ODE models, since an ODE system may be split among several entities. For example, if there are three species (resp. agent) involved in a common dynamics, it is possible to declare each equation inside the corresponding species (resp. agent) definition. An example is shown [below](#).

Defining and solving an ODE system

Defining an ODE system with `equation`

Defining a new ODE system needs to define a new `equation` block in a species. As an example, the following `eqSI` system describes the evolution of a population with 2 compartments (S and I) and the flow from S to I compartments:

```
species userSI {
    float t ;
    float I ;
    float S ;
    int N ;
    float beta<-0.4 ;
    float h ;

    equation eqSI {
        diff(S,t) = -beta * S * I / N ;
        diff(I,t) = beta * S * I / N ;
    }
}
```

This equation has to be defined in a species with `t`, `S` and `I` attributes. `beta` (and other similar parameters) can be defined either in the specific species (if it is specific to each agent) or in the `global` if it is a constant.

Note: it is mandatory to declare a differentiation variable (here `t`) as an attribute in

the species. It is automatically updated using the `solve` statement and contains the time elapsed in the equation integration.

solve an ODE system

Once the equation or system of equations has been defined, it is necessary to execute a `solve` statement inside a reflex in order to numerically integrate the ODE system. The reflex is executed at each cycle, and the values of the attributes used in the equations (`S` and `I` in the previous example) are updated with the values obtained by integrating the system between the start time end and end time of the current cycle.

```
reflex solving {
    solve eqSI method: #rk4 step_size: h;
}
```

Several numerical schemes are available to solve the ODE system. More details about the numerical schemes and the `solve` syntax are provided [below](#).

Alternative way to define an ODE system

Split a system into several agents

An equation system can be split into several species and each part of the system are synchronized using the `simultaneously` facet of `equation`. The system split into several agents can be integrated using a single call to the `solve` statement. Notice that all the `equation` definition must have the same name.

For example, the SI system presented above can be defined in two different species `S_agt` (containing the equation defining the evolution of the S value) and `I_agt` (containing the equation defining the evolution of the I value). These two equations are linked using the `simultaneously` facet of the `equation` statement. This facet expects a set of agents. The integration is called only once in a simulation step, e.g. in the `S_agt` agent.

```

global {
    int N <- 1000;
    float hRK4 <- 0.01;
}

species S_agt {
    float t ;
    float Ssize ;

    equation evol simultaneously: [ I_agt ] {
        diff(Ssize, t) = (- sum(I_agt accumulate [each.beta *
each.Isize]) * self.Ssize / N);
    }

    reflex solving {solve evol method: #rk4 step_size: hRK4 ;}
}

species I_agt {
    float t ;
    float Isize ; // number of infected
    float beta ;

    equation evol simultaneously: [ S_agt ] {
        diff(Isize, t) = (beta * first(S_agt).Ssize * Isize / N);
    }
}

```

The interest is that the modeler can create several agents for each compartment, which different values. For example in the SI model, the modeler can choose to

create 1 agent `S_agt` and 2 agents `I_agt`. The `beta` attribute will have different values in the two agents, in order to represent 2 different strains.

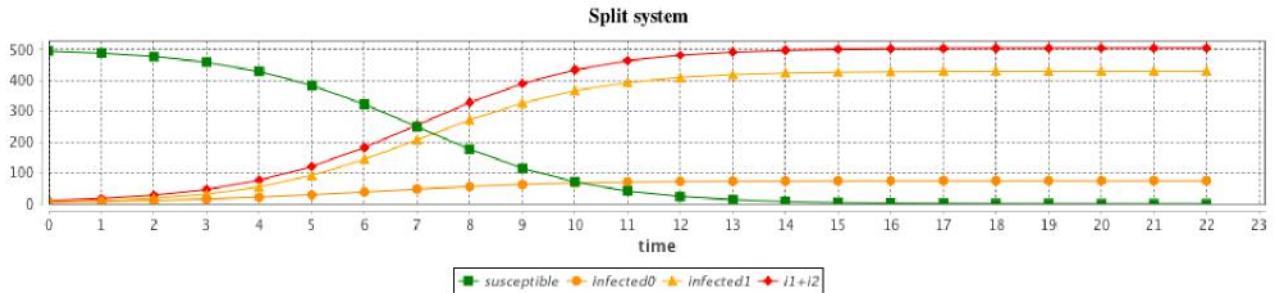
```
global {
    int number_S <- 495 ; // The number of susceptible
    int number_I <- 5    ; // The number of infected
    int nb_I <- 2;
    float gbeta  <- 0.3 ; // The parameter Beta

    int N <- number_S + nb_I * number_I ;
    float hRK4 <- 0.1 ;

    init {
        create S_agt {
            Ssize <- float(number_S) ;
        }
        create I_agt number: nb_I {
            Isize <- float(number_I) ;
            self.beta <- myself.gbeta + rnd(0.5) ;
        }
    }
}
```

The results are computed using the RK4 (Runge-Kutta 4) method with:

- `number_S = 495`
- `number_I = 5`
- `nb_I = 2`
- `gbeta = 0.3`
- `hKR4 = 0.1`



Important note: the `solve` statement must be called once and only once per cycle. In this example, it is executed in the 'solving' reflex of the only `S_agt` agent. There is no 'solving' reflex in the `I_agt` agents: since the equations definitions blocks are connected through the `simultaneously` facet, there equation blocks will be integrated by the `S_agt` agent. Note also that if there were several `S_agt` agents, with the same definition of the 'solving' reflex, the `solve` statement would be executed several times, which could result in wrong results. To ensure that it is called only once, the 'solving' reflex should be rewritten. For example, it is possible to write this:

```
reflex solving when: (int(self)=0) {solve evol method: #rk4
step_size: hRK4 ;}
```

More details

Details about the `solve` statement

The `solve` statement can have a huge set of facets (see [this page](#) for more details). The basic use of the `solve` statement requires only the equation identifier. By default, the integration method is Runge-Kutta 4 with a fixed integration step of `0.005*step`, which means that each simulation step (cycle) is divided into 200 smaller integration steps that are used to simulate a continuous evolution of the system.

For fixed integration step numerical schemes such as Runge-Kutta 4, the length of the integration step is defined in the `step_size` facet. Increasing the integration step results in faster computation at the cost of accuracy.

Integration method with the `method` facet

Several integration methods can be used in the `method` facet. GAMA relies on the [Apache Commons Math library](#) to provide numerical schemes; it thus provides access to the various solvers integrated into the library. The list of all the solver is detailed [in this page, section 15.4 Available integrators](#). The GAML constants associated with each of them (to use in the `method` statement) are:)

- Fixed Step Integrators
 - `#Euler` for [Euler](#). It implements [the Euler integration method](#), which is mainly used for academic illustration of numerical schemes. It should not be used outside of this purpose due to its lack of precision (a very small integration step is required for an acceptable precision).
 - `#Midpoint` for [Midpoint](#)
 - `#rk4` for [Runge-Kutta 4](#). It implements [the Runge-Kutta 4 integration method](#). It provides a faster convergence than the Euler method, and thus does not require very small integration steps. However the user has to determine manually the ideal integration step. For that reason, it is recommended to try first an adaptative stepsize integrator such as the Dorman-Prince 5(4) integration method.
 - `#Gill` for [Gill](#)
 - `#ThreeEighths` for [3/8](#)
 - `#Luther` for [Luther](#)
- Adaptive Stepsize Integrators
 - `#HighamHall54` for [Higham and Hall](#)
 - `#DormandPrince54` for [Dormand-Prince 5\(4\)](#) It implements [the Dorman-Prince 5\(4\) integration method](#). It is similar to the `ode45` in Matlab. This

method is based on the Runge-Kutta solvers family. It evaluates the error between the numerical solution and the analytic solution, and adapt the integration step in order to minimize it. It is recommended to try this method first, even if it may not be the best one in case of [stiff problems](#).

- `#dp853` for [Dormand-Prince 8\(5,3\)](#).
- `#GraggBulirschStoer` for [Gragg-Bulirsch-Stoer](#)
- `#AdamsBashforth` for [Adams-Bashforth](#)
- `#AdamsMoulton` for [Adams-Moulton](#)

Integration steps

The length of the integration step has a huge impact on precision: a smaller integration step means more evaluation points, which results in a better precision but a longer computation time. In order to improve the precision of the integration or its speed, the integration step can be set using the `step_size` facet for fixed steps methods.

- `step_size` (float): integration step, use with most fixed steps integrator methods (default value: `0.005*step`)

Adaptive stepsize integrators (e.g. `#DormandPrince54`) automatically determine and set the integration step according to a given error tolerance. Some of them also use different integration steps all over the computation, since parts of the solution that are stable enough do not require a very small integration step, while parts with high variations need more precision. Such integrators require thus more information, through the following mandatory facets:

- `min_step`, `max_step` (float): these 2 values define the range of variation for the integration step. As an example, we can use: `min_step:0.01 max_step:0.1`.
- `scalAbsoluteTolerance` and `scalRelativeTolerance` (float): they define the allowed absolute (resp. relative) error. As an example, we can use: `scalAbsoluteTolerance:0.0001 scalRelativeTolerance:0.0001`.

Synchronization between simulation and integration

The simulation and the integration are synchronized: if one simulation step represents 1 second, then one call of the `solve` statement will integrate over 1s in the ODE system. This means that the `step` attribute of the `global` has an impact on the integration. See below to observe this influence.

It is thus important to specify the unit of the parameters used in the ODE system (in particular relatively to time).

It is also important to notice that the integration step `step_size` will only control the precision of the integration. If `step` (of the `global`) is `1#s`, then after 1 call of `solve`, `1#s` has flowed in the equation system. If `step_size` is set to `1#s` or to `0.01#s` will not impact this fact. The only difference is that in the latter case, the solver made 100x more computations than in the former one (increasing the precision of the final result).

Additional facets

Here are additional facets that be added to the `solve` statement:

- `t0` (float): the first bound of the integration interval (default value: `cycle*step`, the time at the beginning of the current cycle.)
- `tf` (float): the second bound of the integration interval. Can be smaller than `t0` for a backward integration (default value: `cycle*step`, the time at the beginning of the current cycle.)

This might be useful more model coupling, when the system to integrate is not linked to the time evolution of the main simulation.

Intermediate results

In one simulation step, if the statement `solve` is called one time, several integration

steps will be done internally. Intermediate computation results can be accessed using the notation: `var[]` that returns the list of intermediate values of the variable `var` involved in a differential equation. As an example, with a SIR equation:

```
species agent_with_SIR_dynamic {
    int N <- 1500 ;
    int iInit <- 1;
    float t;
    float S <- N - float(iInit);
    float I <- float(iInit);
    float R <- 0.0;

    float alpha <- 0.2 min: 0.0 max: 1.0;
    float beta <- 0.8 min: 0.0 max: 1.0;
    float h <- 0.01;

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

    reflex solving {
        solve SIR method: #rk4 step_size: h ;

        write S[];
        write I[];
        write R[];
        write t[];
    }
}
```

We can use `S[]`, `I[]`, `R[]` and `t[]` to access the list of intermediate variables of these 4 variables. Since `S[]` is a list the first element can be accessed with `S[] [0]`.

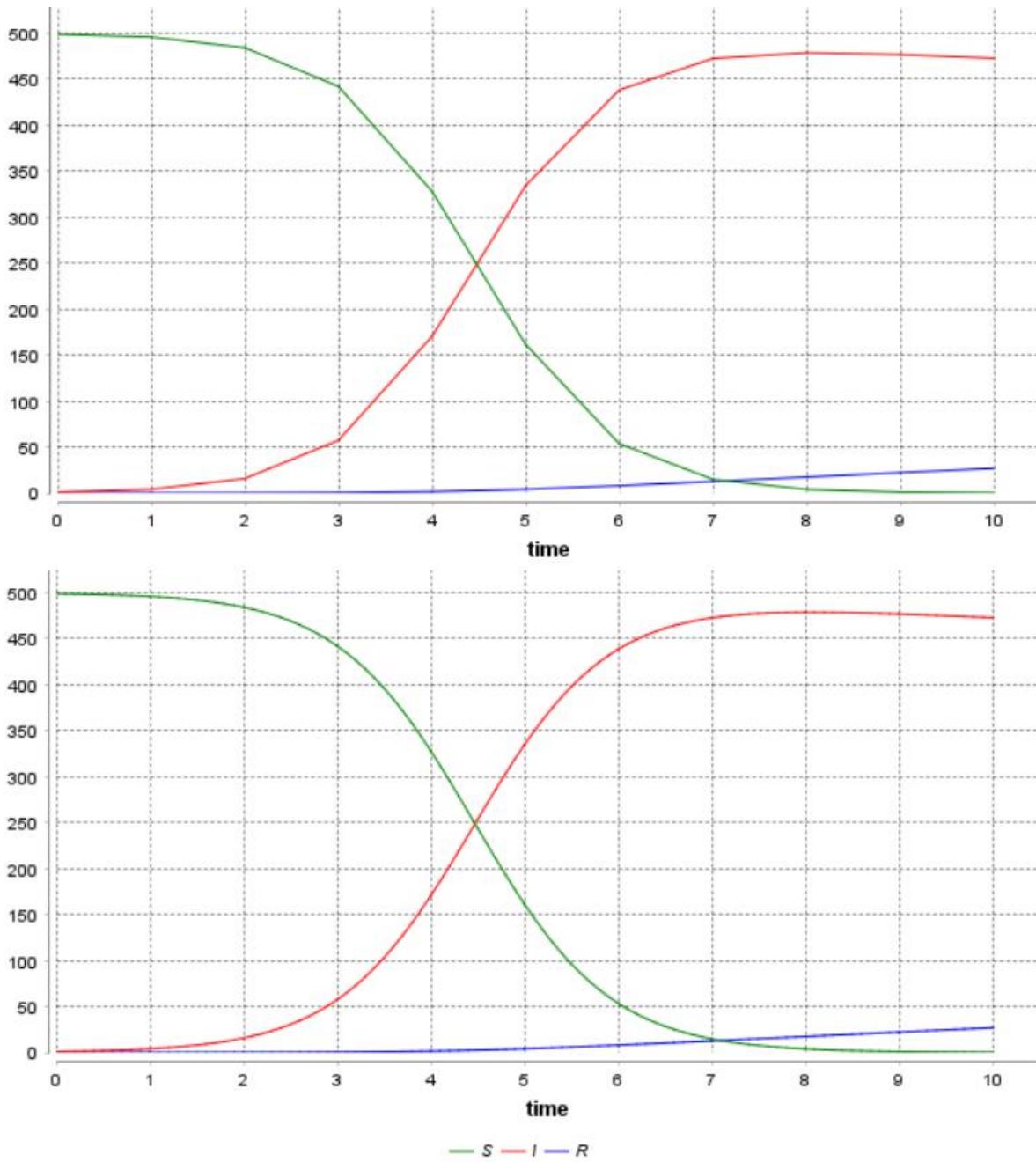
Note that the current value of a variable, i.e. `S`, equals to the last value of the list

```
S[]: S = last(S[]).
```

Accessing the intermediate values can be useful to provide smooth continuous charts. A way to do so is to provide the display with the full list of integration times and values, such as:

```
experiment continuous_display type: gui {
    output {
        display display_charts axes: false{
            chart 'SIR' type: series x_serie:
            first(agent_with_SIR_dynamic).t[] y_range: {0,1000} background:
            #white {
                data "S" value: first(agent_with_SIR_dynamic).S[]
                color: #green marker: false;
                data "I" value: first(agent_with_SIR_dynamic).I[]
                color: #red marker: false;
                data "R" value: first(agent_with_SIR_dynamic).R[]
                color: #blue marker: false;
            }
        }
    }
}
```

The following picture illustrates the result: the top subfigure shows the dynamics with discrete visualization and the bottom one the continuous curves.

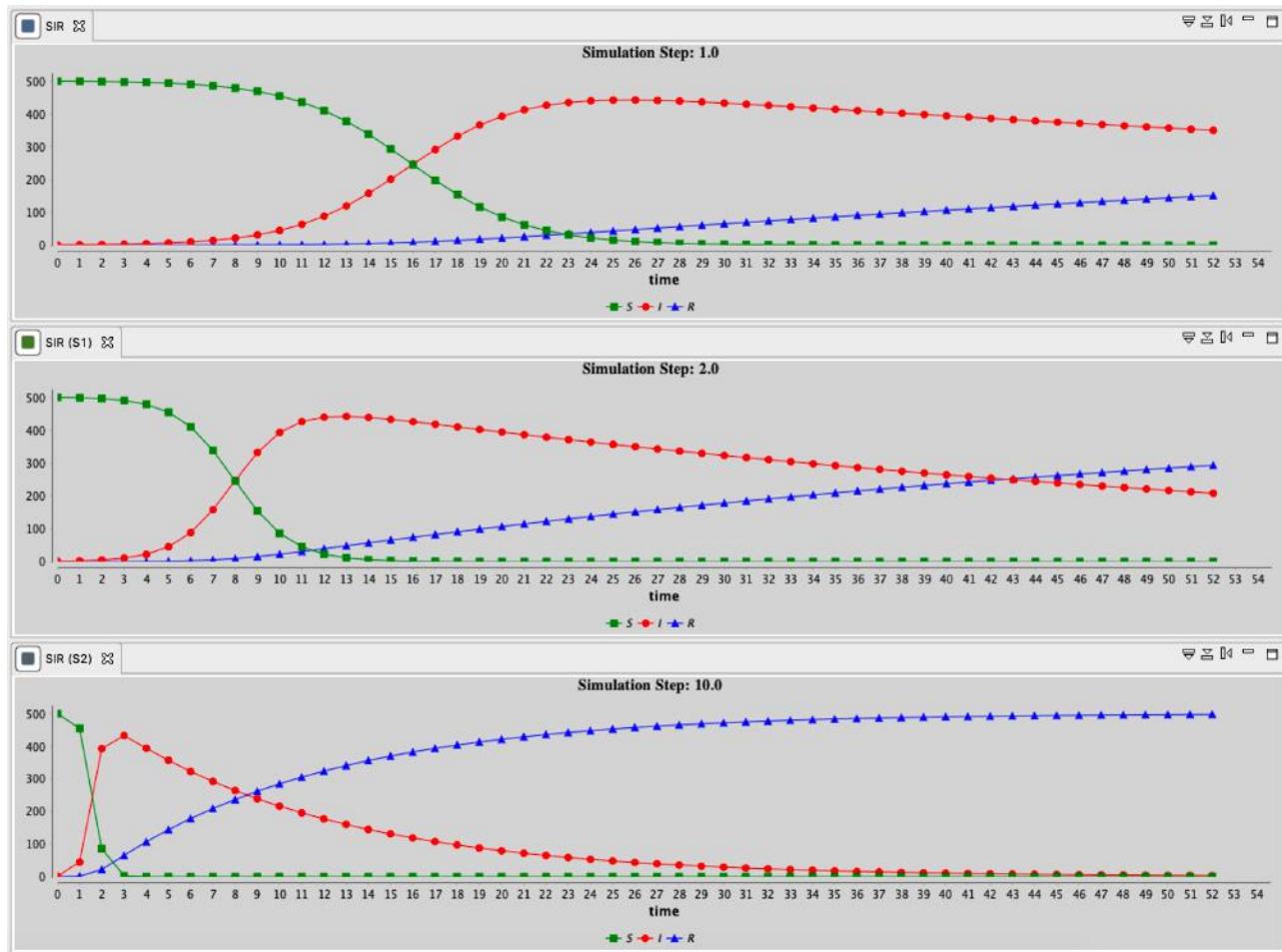


Example of the influence of the integration step

The `step` (of the `global`) and `step_size` (of `solve`) facets may have a huge influence on the results. `step_size` only has an impact on the result accuracy. The `step` facet changes the cycle duration and so the time scale and results in curves being horizontally scaled.

In the following image, the `step` facet has been change from 1.0 (first simulation) to 2.0 (second simulation). The dynamics are exactly the same, but they are viewed at different time scales.

The following image illustrates this impact, by calling (with 3 different values of `step`):



When changing this facet, be sure that the time scale of the ODE system remains consistent with the one of the other agents.



>

Recipes

Version: 1.9.1

Recipes

Understanding the [structure of models](#) in GAML and gaining some insight of [the language](#) is required, but is usually not sufficient to build correct models or models that need to deal with specific approaches (like [equation-based modeling](#)). This section is intended to provide readers with practical "how to"s on various subjects, ranging from the use of [database access](#) to the design of [agent communication languages](#). It is by no means exhaustive, and will progressively be extended with more "recipes" in the future, depending on the concrete questions asked by users.

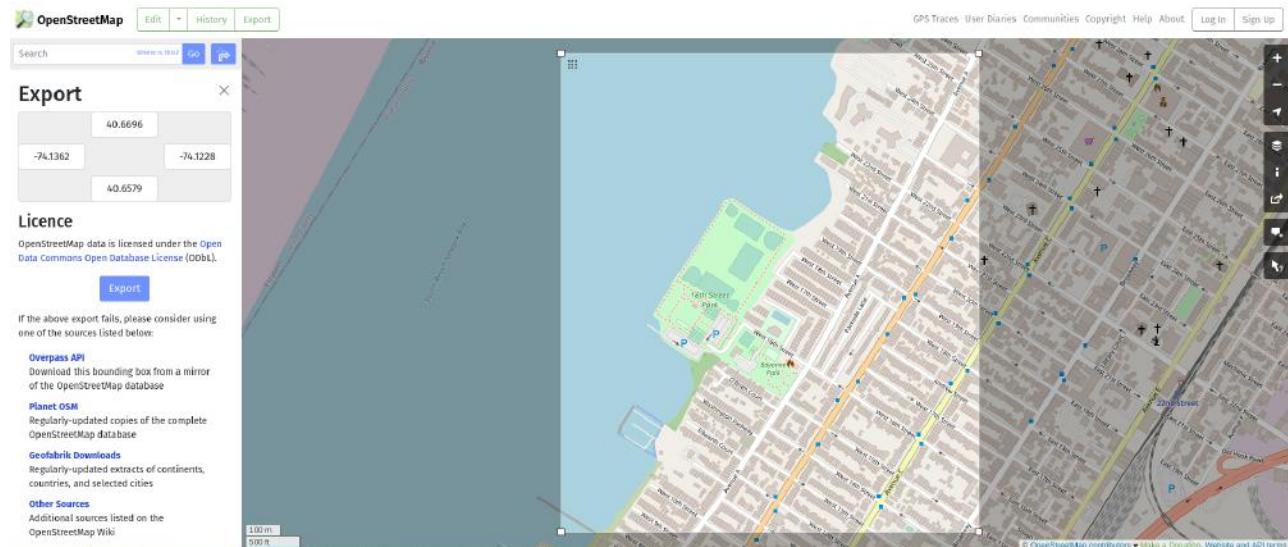
Version: 1.9.1

Manipulate OSM Datas

This section will be presented as a quick tutorial, showing how to proceed to manipulate OSM (Open street map) data, clean them and load them into GAMA. We will use the software [QGIS](#) to change the attributes of the OSM file.

Note that GAMA can read and import OpenStreetMap data natively and create agents from them. An example model is provided in the Model Library (Data Importation / OSM File Import.gaml). In this case, you will have to write a model to import, select data from OpenStreetMap before creating agents and then could export them into shapefiles, much easier to use in GAMA.

From the website [openstreetmap.org](#), we will choose a place (in this example, we will take a neighborhood in New York City). Directly from the website, you can export the chosen area in the osm format.

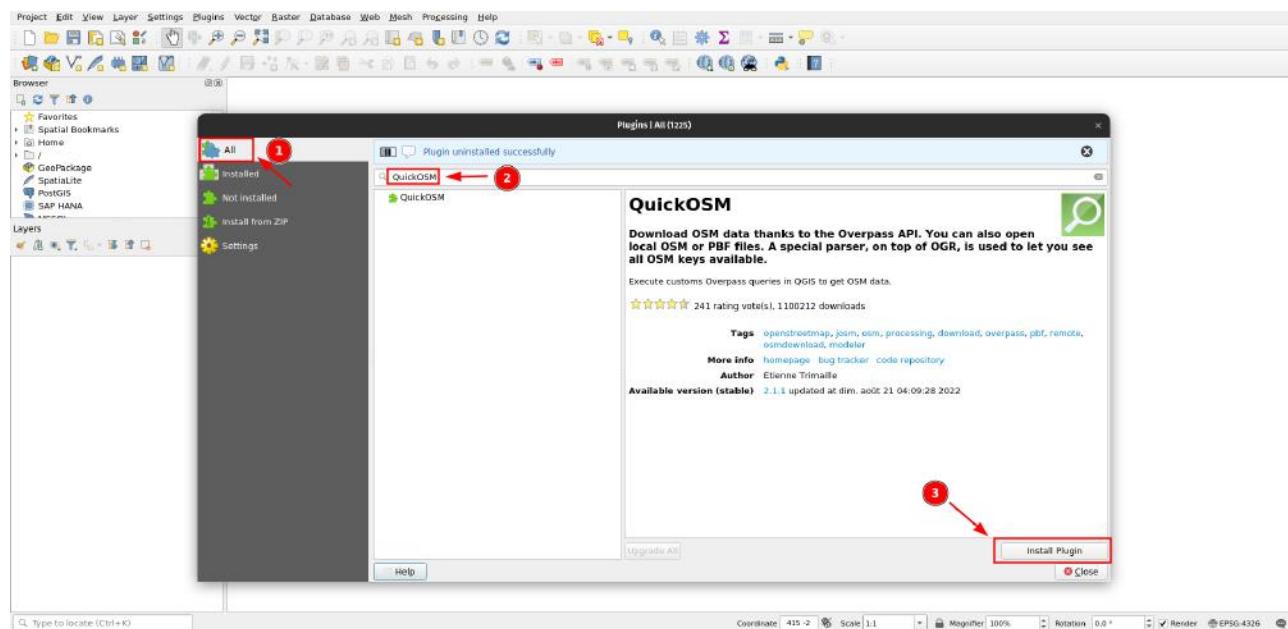


We have now to manipulate the attributes for the exported osm file. Several

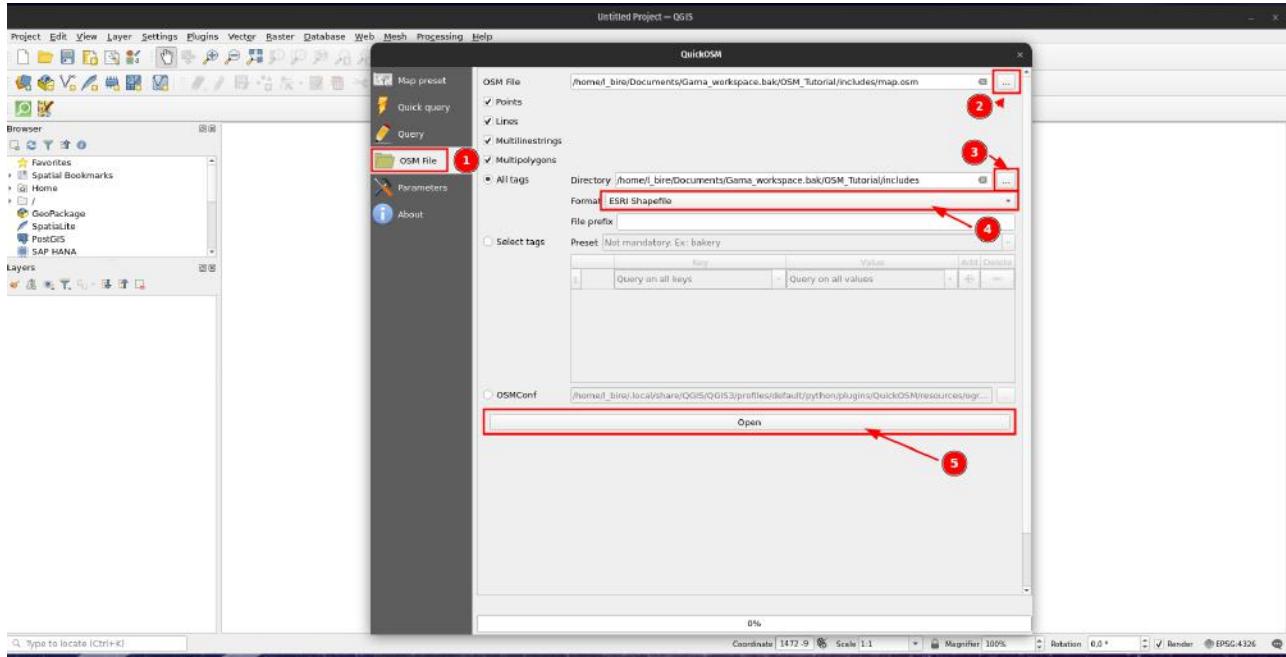
softwares can be used, but we will focus on [QGIS](#), which is totally free and provides a lot of possibilities in term of manipulation of data.

Once you have installed correctly QGIS, launch QGIS Desktop, and start to import the topology from the osm file.

QGis 3, the version we are using to build this tutorial, needs us to install a plugin to process files downloaded from OSM website. There are several versions of plugins allowing to do that. However, the most stable and simple one to use is QuickOSM. Go to your extension manager, select "All extensions" on the top left corner, look for "QuickOSM" extension and install it.



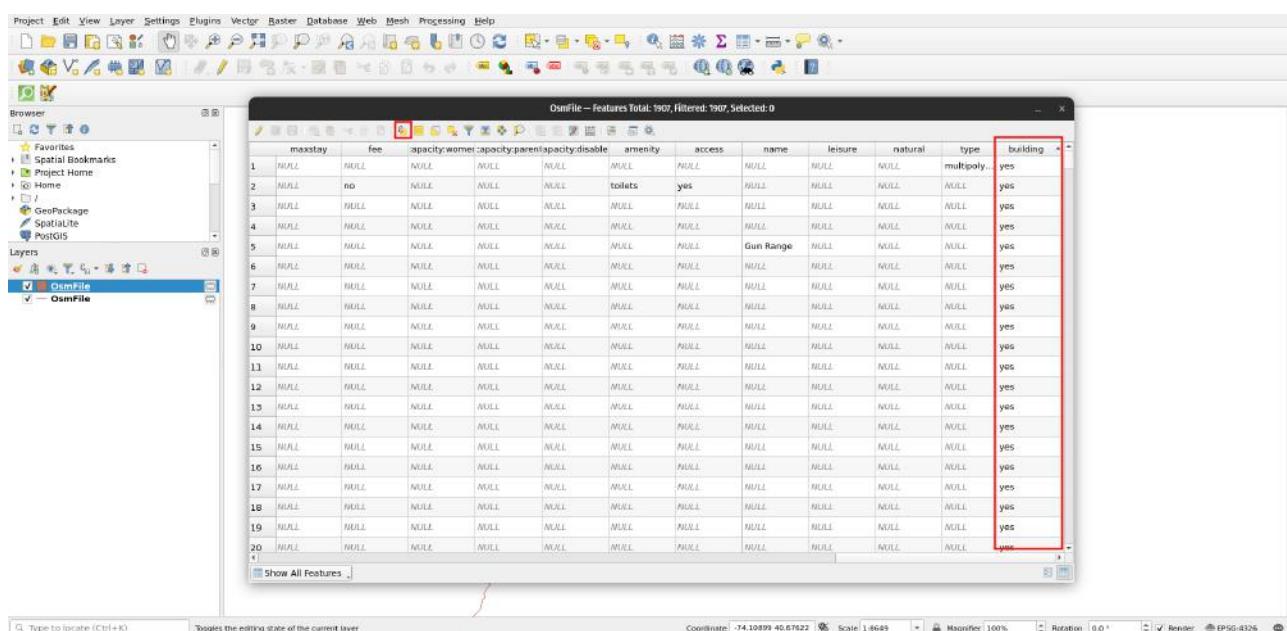
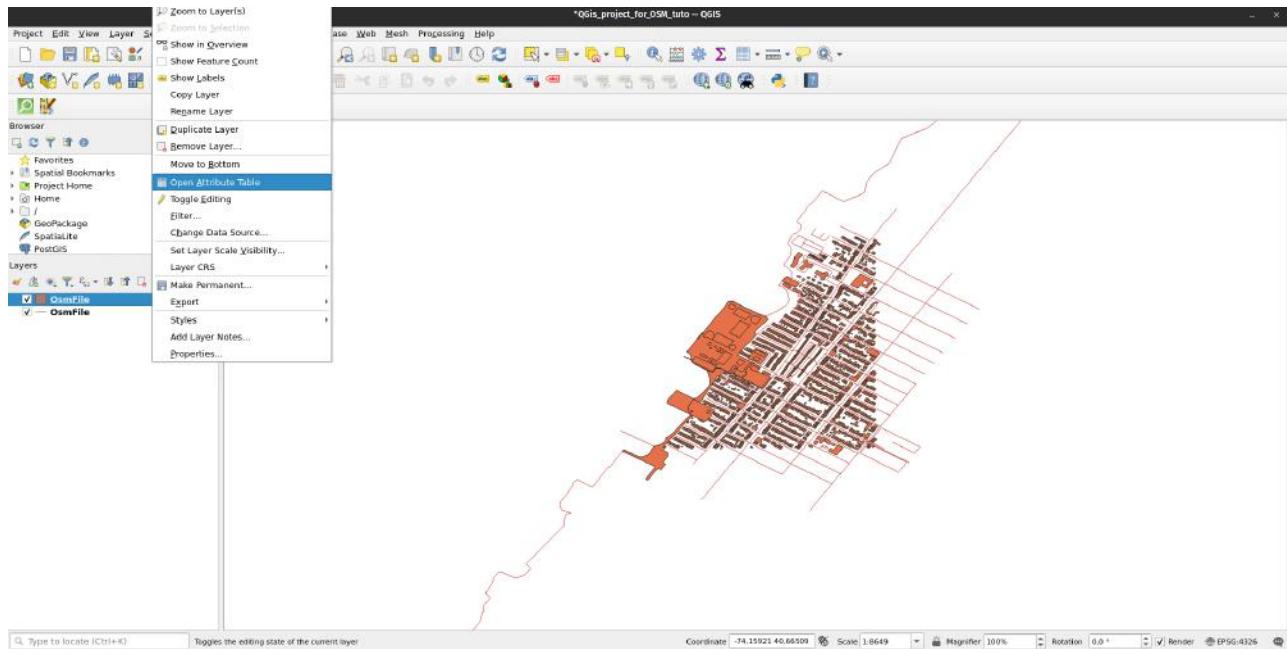
You should now be able to use the extension QuickOSM. Open its menu by entering the vector menu on the top QGis panel, go to QuickOSM, then select : 1) OSM file 2) Browse your xml file. 3) Browse to select the folder that will welcome the processed file through QGis (select the "includes" folder of your current GAMA project for more efficiency). 4) Select the format you want for your processed file (in the example, ESRI shapefile will give a .shp, very well processed by GAMA afterwards). 5) Open your file.

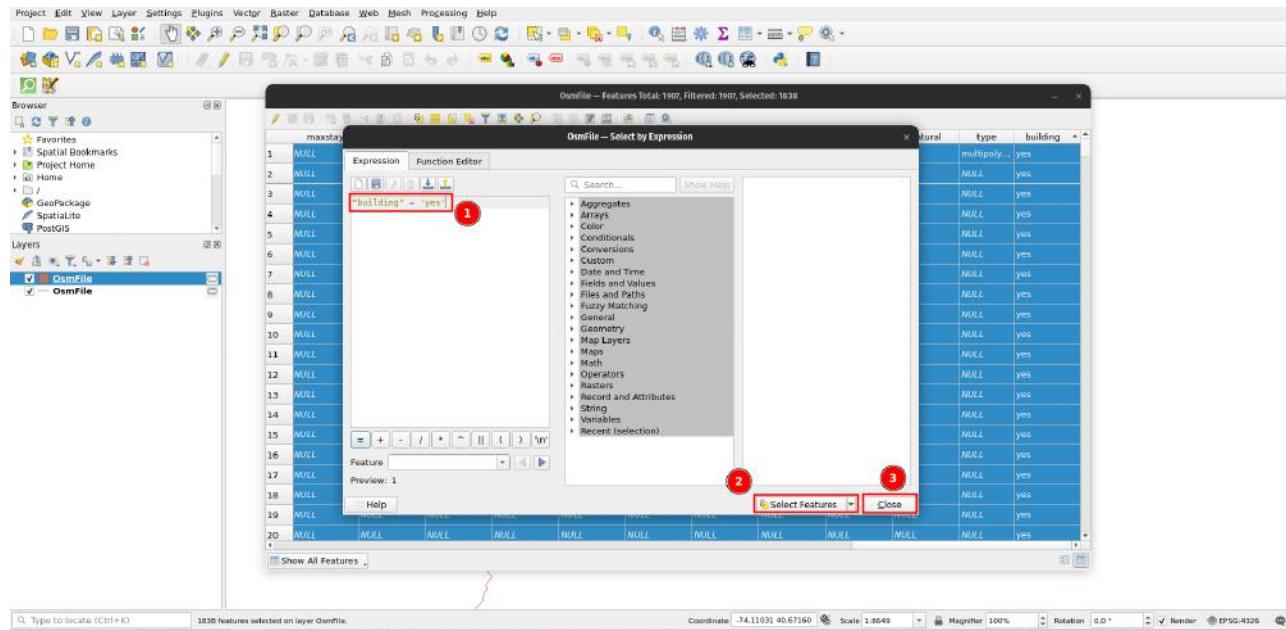


You will go back naturally to the main QGis UI and you downloaded OSM layer will be visible.

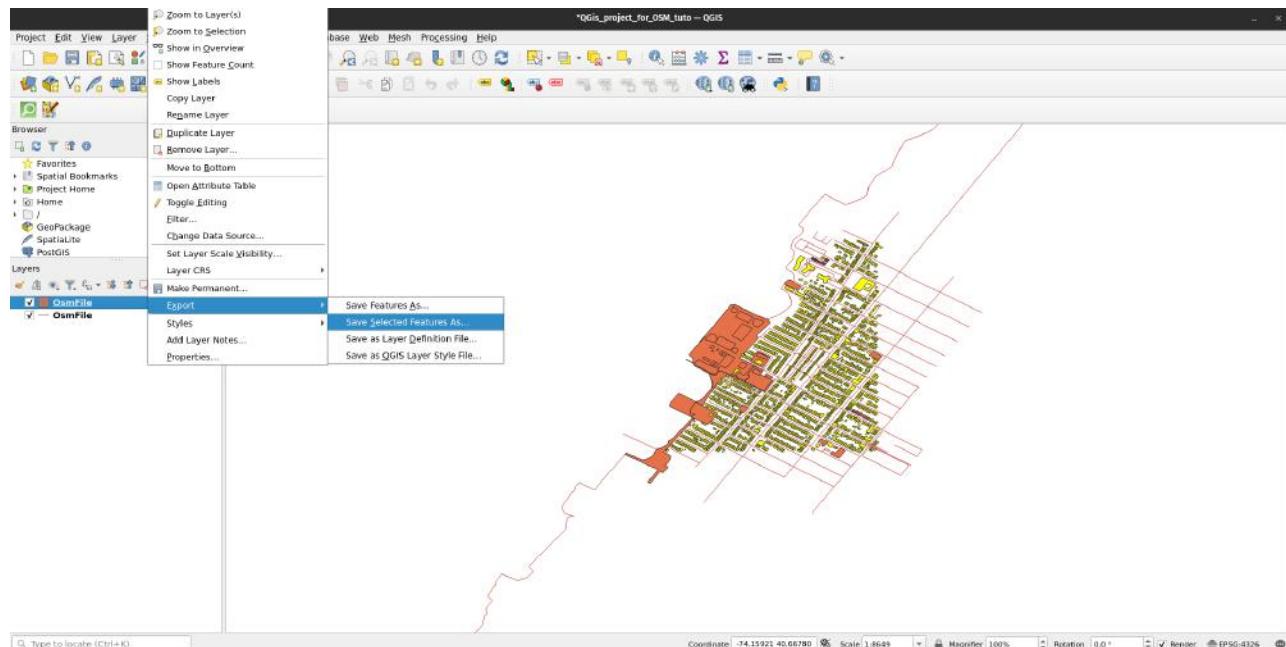
Before beginning the layer treatment process with QGis, please note that it is easier to use separated shapefiles for each entities if the objective is to use GAMA afterwards. Therefore, we recommend importing in GAMA separated shapefiles for each spatial entity you want to make appear in your model. Indeed, it is simpler to create one species for each spatial entity instead of having one species declined by different arributes. If you want to represent buildings and houses, it's recommended to have two separated files : one for buildings and one for houses.

First, we want buildings to be isolated from the other polygons. We can use the field 'building' to make an attribute selection to do that. By opening the attribute table of the file, we can check that this field exists and that every building has been attributed a "yes". If other information are given, replace the information by "yes" to facilitate the process of selection. Once it's done, go to the attribute selection menu on the top panel and type this in the dialogue box : "building" = 'yes'. Click on "select features" once it's done and close the window.



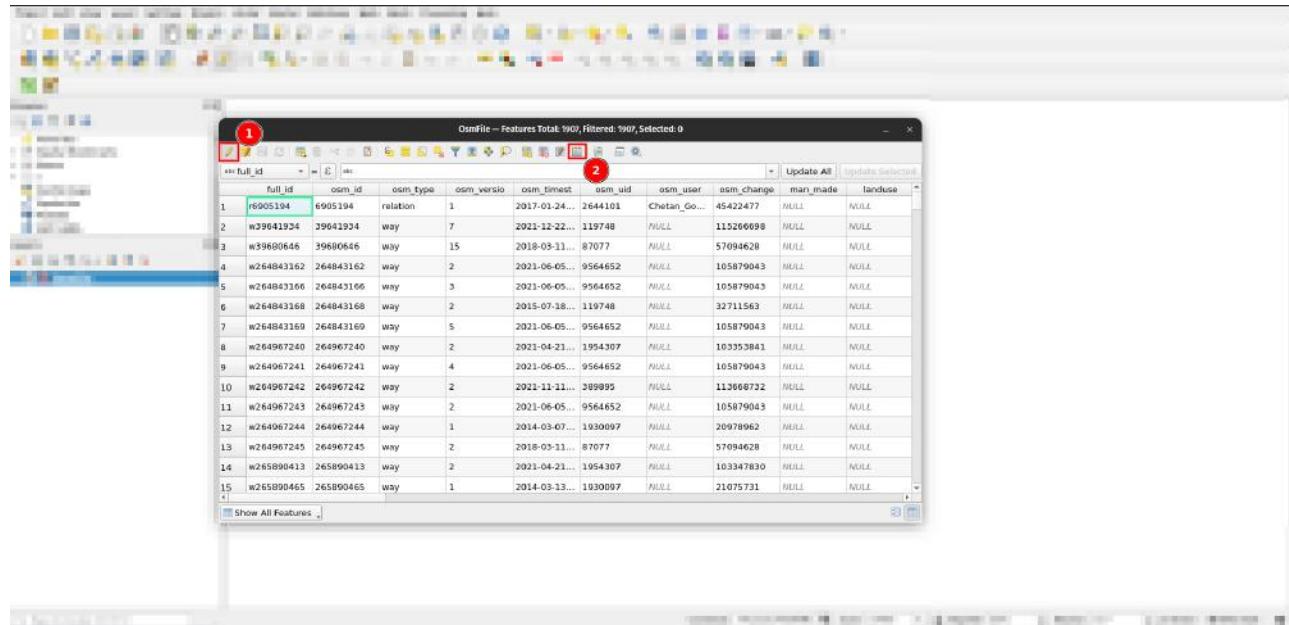


If typed correctly, QGis should understand that you want to select only polygons that are recognized as buildings. The buildings polygons should be highlighted in yellow now. Save the selected features as a new .shp file in your "includes" folder. We called ours "OSM_For_GAMA_Buildings". The new layer which just appeared should only comprise buildings of the area.

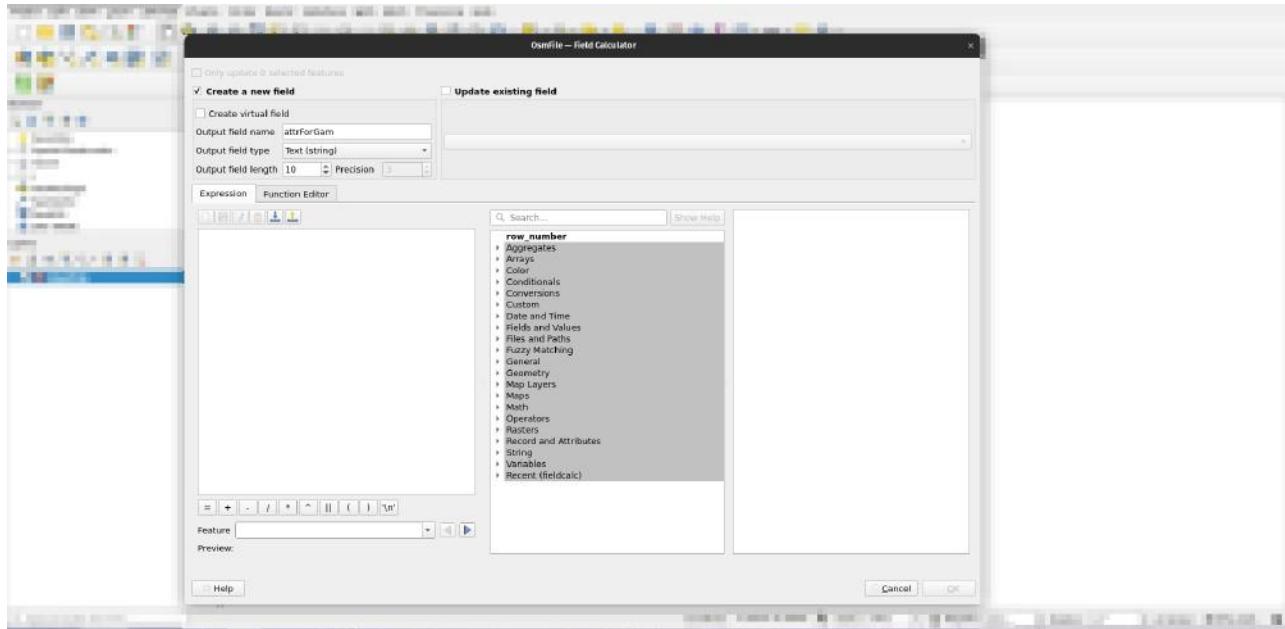


Then, we can create a new field for the new buildings layer to allow a better handling in GAMA platform do differentiate colors : first, enter you attribute table of the layer.

Then, go to edit mode (the pen icon on the top left corner) and select field calculator.



Stay on the left hand side, we will now set up the field's characteristics, you can copy what you see in the screenshot bellow. Pay attention to selecting "string" to the field type, otherwise you won't be able to get the proper format of attributes in the following steps of this tutorial.

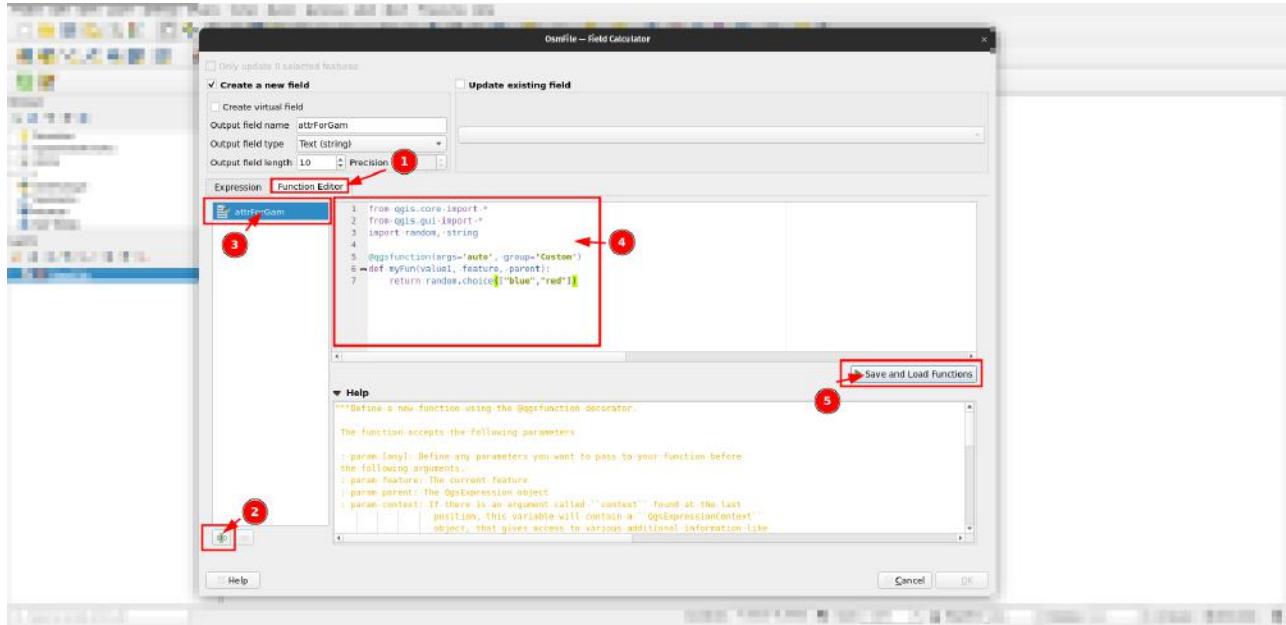


We want to create this attribute and associate to it variables that will be colors. We want to ask QGis to do that randomly on its own. Therefore, we have to provide the software a function. Go to the "function editor" tab, click on the "+" to add a new function file and write these lines down after having deleted the default help :

```
from qgis.core import *
from qgis.gui import *
import random, string

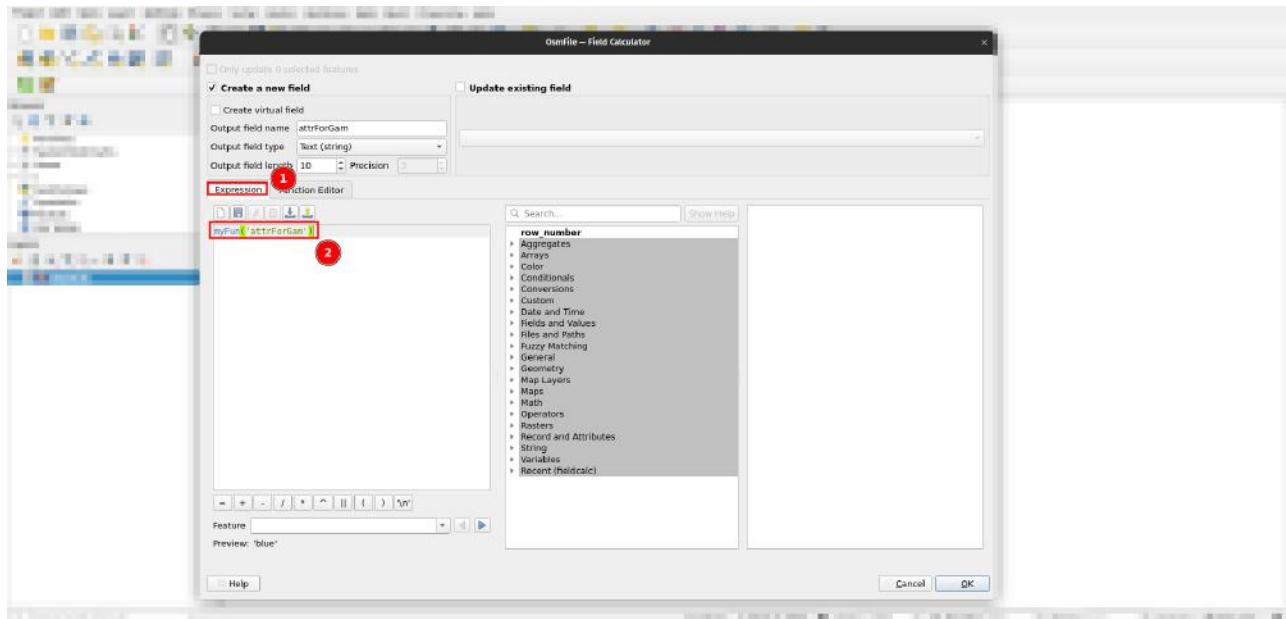
@qgsfunction(args='auto', group='Custom')
def myFun(value1, feature, parent):
    return random.choice(["blue", "red"])
```

Save and run the function using the proper button on the bottom right corner.



Then go to the expression type and call your function by typing :

`myFun('attrForGam')` Click on "Ok" which will get you back to your attribute table : you can now check the layer's attributes to see if the new field 'attrForGam' has been filled with random values "red" or "blue".



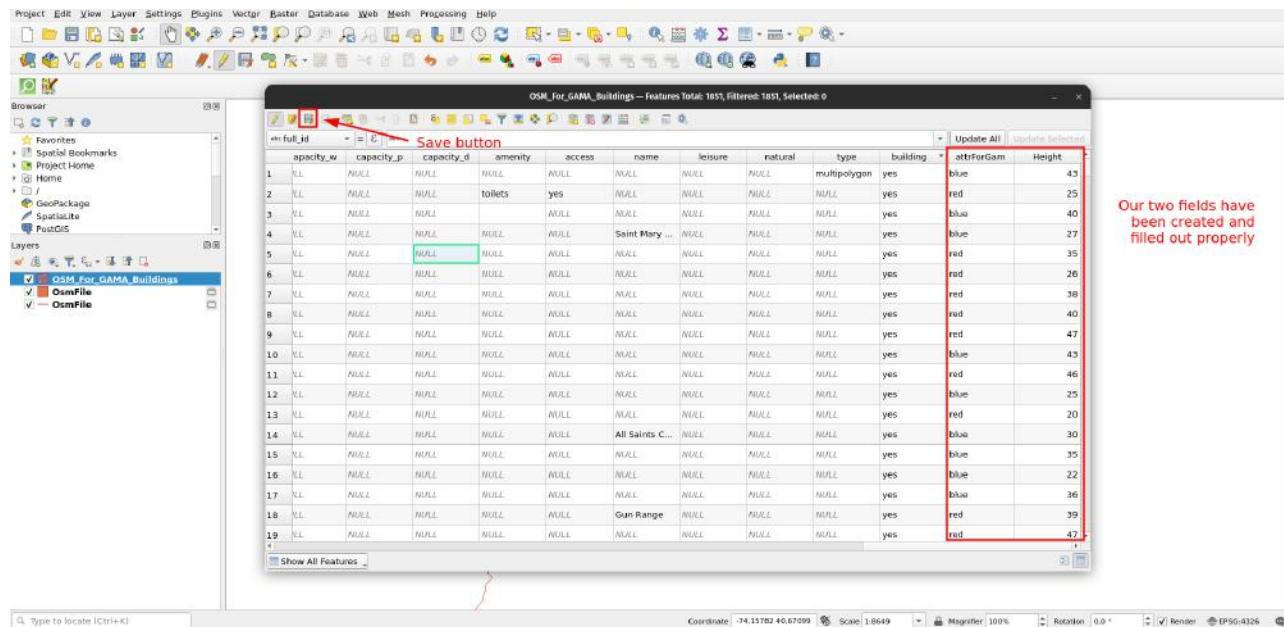
When you download data from OSM website, some fields might be missing. For

instance, the file we downloaded here doesn't include the buildings' height. To give realistic aspects to our model, we want here to call another function to make QGis create a "Height" field and associate automatically a height to buildings between 20 and 50 meters high. You can repeat the previous steps for the 'attrForGam' field to create a 'Height' field using the following code :

```
from qgis.core import *
from qgis.gui import *
import random, string

@qgsfunction(args='auto', group='Custom')
def myFunHeight(value1, feature, parent):
    return random.randrange(20, 50, 1)
```

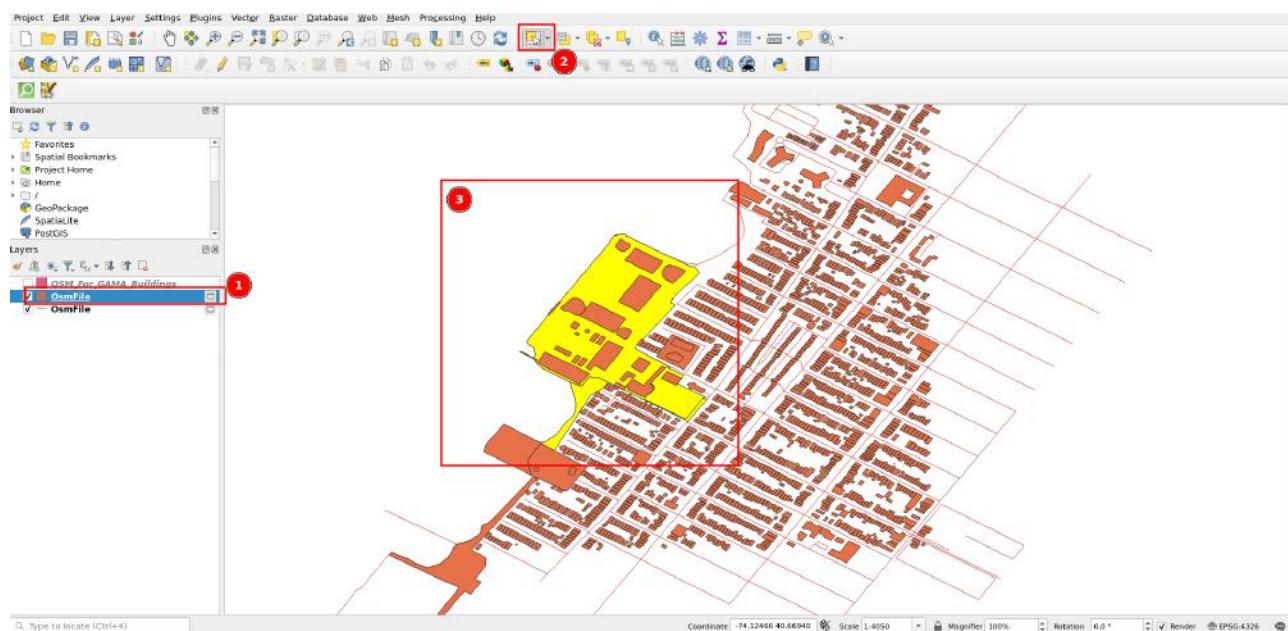
Don't forget to call this new function in the expression tab : `myFunHeight('Height')`. Then, check if the new field has been added and save the modifications of your attributes.



Our work on the buildings layer is done. You don't have to save it since QGis is automatically saving the modifications you do on your files (the one we previously

called "OSM_For_GAMA_Buildings"). The file for buildings is now ready to be used in GAMA for modelling. We now have to take care our other polygons and lines we downloaded.

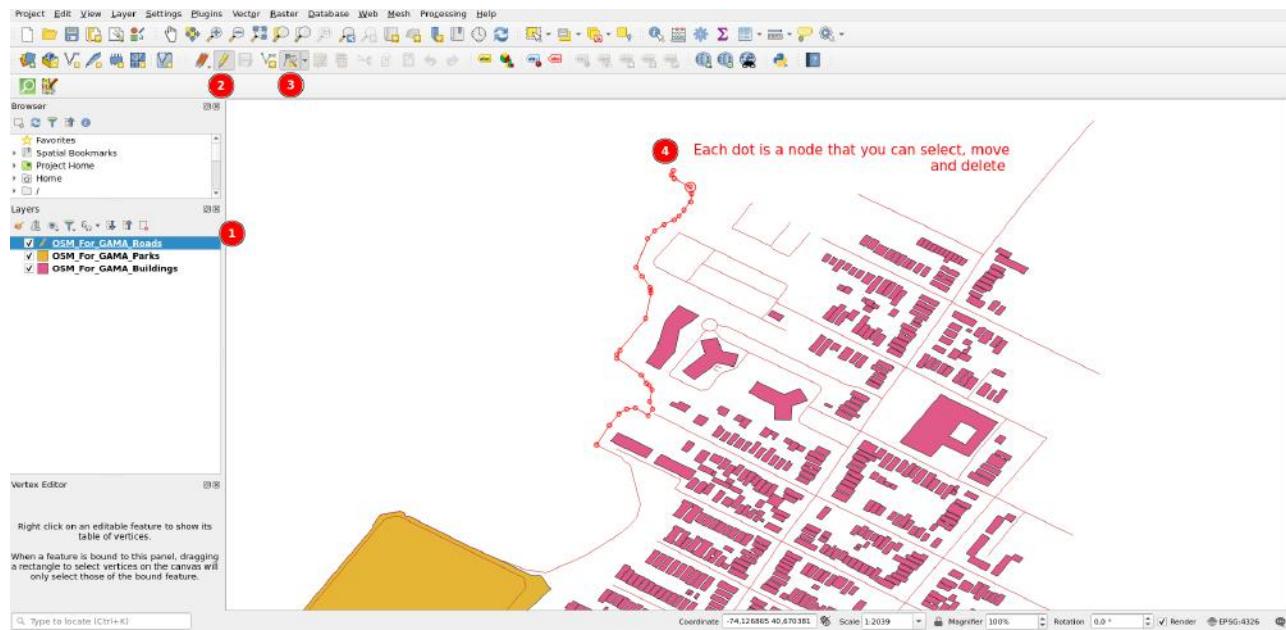
We now want to add the main natural elements to our model : parks. For this, we only have to select the few parks we have in the area thanks to the spatial selection tool provided by QGis. To do that, you have to use the original polygons layer you downloaded to make a spatial selection of the parks. To select several polygons using this tool, just press shift + left click on each polygons you are interested with. Before beginning the selection, locate where the parks are so that you are sure the polygons you select are the parks you want to represent. 1) Select the polygon layer. 2) Select the tool "select features by area or single click" on the top panel. 3) Select the right polygons using left click and pressing maj to select several polygons.



Then, save the selected polygons as a new shapefile which will only comprise parks areas. For this tutorial, we called the parks shapefile "OSM_For_GAMA_Parks".

Finally, we need roads for our possible agents to travel the city. The shapefile already exists from the OSM file we downloaded. It is possible to modify it using the edition

mode after selecting the lines layer, and delete the roads we don't want.



Don't forget to save your layer as a .shp file in your "includes" folder of your current GAMA project. For this tutorial, we call the roads shapefile "OSM_For_GAMA_Parks"

Please note that you can repeat these steps as many times as you want according to the level of details you need in your model. As OSM provides a large possibility of land use types, we cannot go over every one of them in this tutorial. The steps are the same as the ones described above.

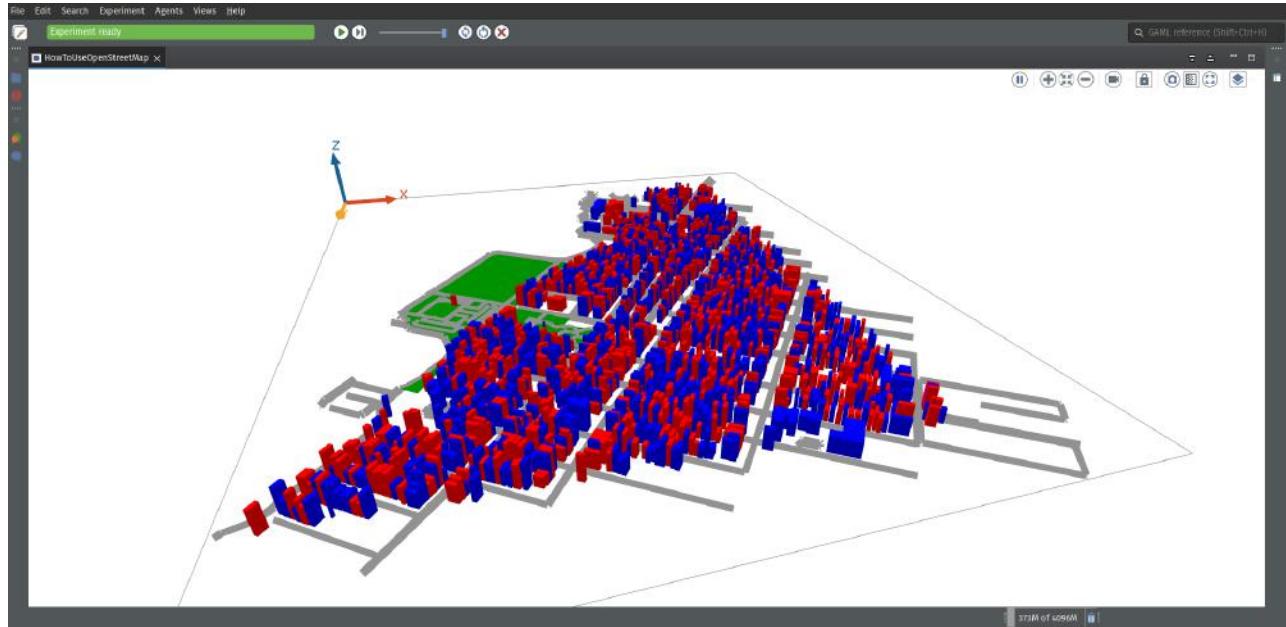
You can now import your three .shp files that should be in your "includes" folder of your current project.

```
model OSMtutorial

global {
    // Global variables related to the Management units

    file shapeFile1 <- file("../includes/OSM_For_GAMA_Buildings.shp");
    file shapeFile2 <- file("../includes/OSM_For_GAMA_Parks.shp");
```

Here is the result, with a special colorization of the different elements regarding the value of the attribute "attrForGama", an elevation regarding the value of the attribute "height", and basic species creation for roads and parks.



Version: 1.9.1

Implementing diffusion

GAMA provides you the possibility to represent and simulate the diffusion of a variable through a grid topology.

Index

- [Diffuse statement](#)
- [Diffusion with matrix](#)
 - [Diffusion matrix](#)
 - [Gradient matrix](#)
 - [Compute multiple propagations at the same step](#)
 - [Executing several diffusion matrix](#)
- [Diffusion with parameters](#)
- [Computation methods](#)
 - [Convolution](#)
 - [Dot Product](#)
- [Use mask](#)
 - [Generalities](#)
 - [Tips](#)
- [Pseudo code](#)

Diffuse statement

The statement to use for the diffusion is `diffuse`. It has to be used in a `grid` species. The `diffuse` uses the following facets:

- `var` (an identifier), (omissible) : the variable to be diffused
- `on` (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restituted to the neighboring cells, multiplied by the variation value (no signal loss). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (loss of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix masking the diffusion (matrix created from an image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- `matrix` (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- `min` (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- `propagation` (a label), takes values in {diffusion, gradient} represents both the way the signal is propagated and the way to treat multiple propagations of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect

to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is $S' = (S / N / \text{proportion}) - \text{variation}$. The intensity of S is then diminished by $S * \text{proportion}$ on P. In 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 neighbor 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 diffusion rate
- `radius` (int): a diffusion radius (in number of cells from the center)
- `variation` (float): an absolute value to decrease at each neighbor

To write a diffusion, you first have to declare a grid and declare a special attribute for the diffusion. You will then have to write the `diffuse` statement in another scope (such as the `global` scope for instance), which will permit the values to be diffused at each step. There, you will specify which variable you want to diffuse (through the `var` facet), on which species or list of agents you want the diffusion (through the `on` facet), and how you want this value to be diffused (through all the other facets, we will see how it works [with matrix](#) and [with special parameters](#) just after).

Here is the template of code we will use for the next following part of this page:

```
global {
    int size <- 64; // the size has to be a power of 2.
    cells selected_cells;

    // Initialize the emitter cell as the cell at the center of the
    word
    init {
        selected_cells <- location as cells;
    }
}
```

This model will simulate a diffusion through a grid at each step, affecting 1 to the center cell diffusing variable value. The diffusion will be seen during the simulation through a color code, and through the elevation of the cell.

Diffusion with matrix

A first way of specifying the behavior of your diffusion is using diffusion matrix. A diffusion matrix is a 2-dimension matrix $[n][m]$ with `float` values, where both `n` and `m` have to be **odd values**. The most often, diffusion matrices are square matrices, but you can also declare a rectangular matrix.

Example of matrix:

```
matrix<float> mat_diff <- matrix([
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]]);
```

In the `diffuse` statement, you then have to specify the matrix of diffusion you want in the facet `matrix`.

```
diffuse var: phero on: cells matrix:mat_diff;
```

Using the facet `propagation`, you can specify if you want the value to be propagated as a *diffusion* or as a *gradient*.

Diffusion matrix

A *diffusion* (the default value of the facet `propagation`) will spread the values to the neighbors' cells according to the diffusion matrix, and all those values will be added

together, as it is the case in the following example:

Diffusion matrix :

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

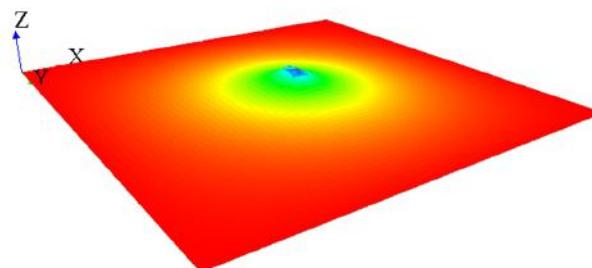
Step 0	Step 1	Step 2
0 0 0 0 0	0 0 0 0 0	0+1/9 *1/9 = 1/81 0+2*1/9 *1/9 = 2/81 0+3*1/9 *1/9 = 3/81 0+2*1/9 *1/9 = 2/81 0+1/9 *1/9 = 1/81
0 0 0 0 0	0 0 0 0 0	1/9*1/9 = 1/81 1/9*1/9 = 1/81 1/9*1/9 = 1/81 1/9*1/9 = 1/81 1/9*1/9 = 1/81
0 0 1 0 0	0 0 1 0 0	0+1/9 *1/9 = 1/81 0+2*1/9 *1/9 = 2/81 0+3*1/9 *1/9 = 3/81 0+2*1/9 *1/9 = 2/81 0+1/9 *1/9 = 1/81
0 0 0 0 0	0 0 0 0 0	1*1/9 = 1/9 1*1/9 = 1/9 1*1/9 = 1/9 1*1/9 = 1/9 1*1/9 = 1/9
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0

Note that the sum of all the values diffused at the next step is equal to the sum of the values that will be diffused multiply by the sum of the values of the diffusion matrix. That means that if the sum of the values of your diffusion matrix is larger than 1, the values will increase exponentially at each step. The sum of the value of a diffusion matrix is usually equal to 1.

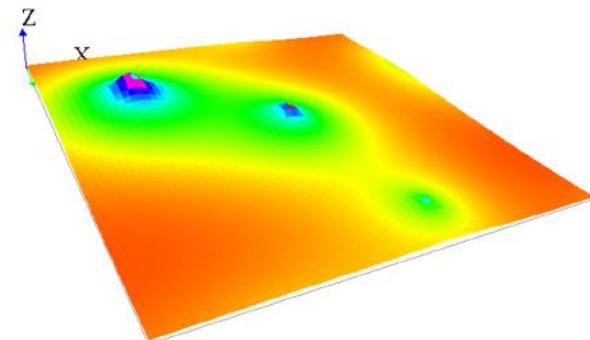
Here are some matrix examples you can use, played with the template model:

Uniform diffusion

```
matrix<float> math_diff_uniform <- matrix([
    [1/9,1/9,1/9],
    [1/9,1/9,1/9],
    [1/9,1/9,1/9]]);
```



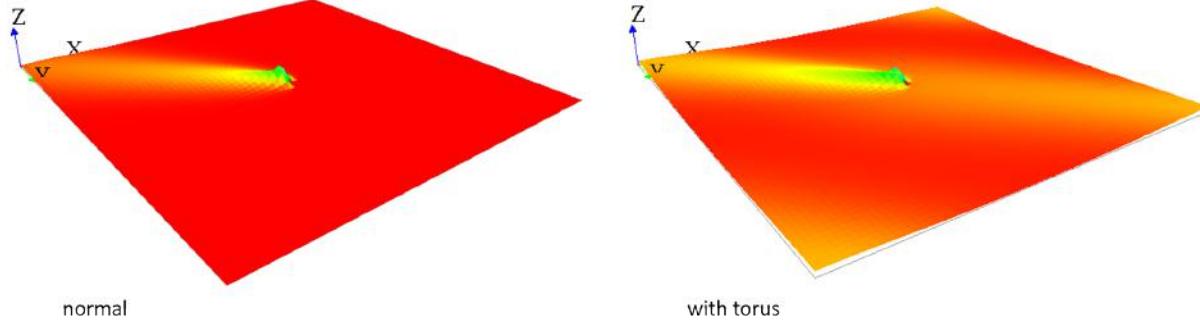
One emitter spot, no torus



several emitter spots, with torus

Anisotropic diffusion

```
matrix<float> math_diff_anisotropic <- matrix([
    [2/9, 2/9, 1/9],
    [2/9, 1/9, 0.0],
    [1/9, 0.0, 0.0]]);
```



Gradient matrix

A `gradient` (use `facet : propagation:gradient`) is another type of propagation. This time, only the larger value diffused will be chosen as the new one.

Gradient matrix :

1/8	1/8	1/8
1/8	1	1/8
1/8	1/8	1/8

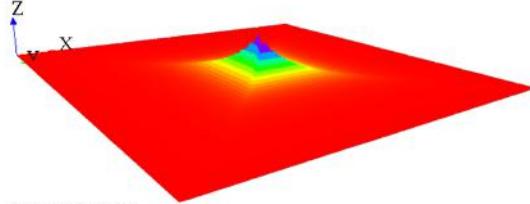
Step 0					Step 1					Step 2				
0	0	0	0	0	0	0	0	0	0	max(0, 1/8 *1/8 = 1/64)				
0	0	0	0	0	0	max(0, 1*1/8) = 1/8	0*1/8 +1*1/8 = 1/8	max(0, 1*1/8) = 1/8	0	max(1*1/8, 1 *1/8 = 1/8)				
0	0	1	0	0	0	max(0, 1*1/8) = 1/8	max(1, 1*1/8) = 1	max(0, 1*1/8) = 1/8	0	max(1, 1*1, 1 *1/8 = 1)				
0	0	0	0	0	0	max(0, 1*1/8) = 1/8	max(0, 1*1/8) = 1/8	max(0, 1*1/8) = 1/8	0	max(1*1/8, 1 *1/8 = 1/8)				
0	0	0	0	0	0	0	0	0	0	max(0, 1/8 *1/8 = 1/64)				

Note that unlike the *diffusion* propagation, the sum of your matrix can be greater than 1 (and it is the case, most often !).

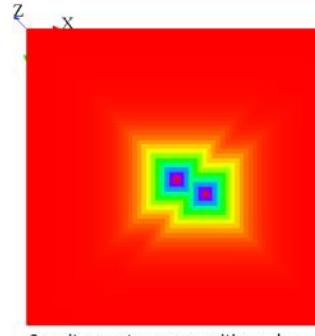
Here are some matrix examples with gradient propagation:

Uniform gradient

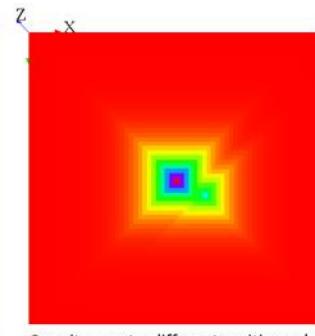
```
matrix<float> math_grad <- matrix([
    [3/4, 3/4, 3/4],
    [3/4, 1, 3/4],
    [3/4, 3/4, 3/4]]);
```



One emitter spot



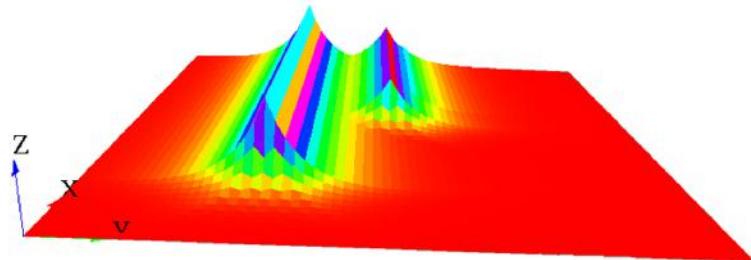
2 emitter spots, same emitting value



2 emitter spots, different emitting value

Irregular gradient

```
matrix<float> math_grad <- matrix([
    [2/4, 3/4, 3/4],
    [3/4, 1, 1],
    [2/4, 3/4, 3/4]]);
```



2 emitter spots, different emitting value

Compute multiple propagations at the same step

You can compute several times the propagation you want by using the facet `cycle_length`. GAMA will compute for you the corresponding new matrix and will apply it.

cycle_length:2



1/9	1/9	1/9		
1/9	1/9	1/9		
1/9	1/9	1/9		
			1/81	2/81
			2/81	3/81
			3/81	6/81
			6/81	1/9
			1/9	6/81
			6/81	3/81
			3/81	2/81
			2/81	1/81

Writing those two things are exactly equivalent (for diffusion):

```
matrix<float> mat_diff <- matrix([
    [1/81, 2/81, 3/81, 2/81, 1/81],
    [2/81, 4/81, 6/81, 4/81, 2/81],
    [3/81, 6/81, 1/9, 6/81, 3/81],
    [2/81, 4/81, 6/81, 4/81, 2/81],
    [1/81, 2/81, 3/81, 2/81, 1/81]]);
reflex diff {
    diffuse var: phero on: cells matrix:mat_diff;
```

and

```
matrix<float> mat_diff <- matrix([
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]]);
reflex diff {
    diffuse var: phero on: cells matrix:mat_diff cycle_length:2;
```

Executing several diffusion matrix

If you execute several times the statement `diffuse` with different matrix on the

same variable, their values will be added (and centered if their dimensions are not equal).

Thus, the following 3 matrices will be combined to create one unique matrix:

$$\begin{array}{|c|c|c|} \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array} + \begin{array}{|c|} \hline 1/9 \\ \hline 0 \\ \hline 1/9 \\ \hline \end{array} + \begin{array}{|c|} \hline 1/9 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 1/9 & 0 & 1/9 \\ \hline 0 & 0 & 0 \\ \hline 1/9 & 0 & 1/9 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array}$$

Diffusion with parameters

Sometimes writing diffusion matrix is not exactly what you want, and you may prefer to just give some parameters to compute the correct diffusion matrix. You can use the following facets in order to do that: `propagation`, `variation` and `radius`.

Depending on which `propagation` you choose, and how many neighbors your grid has, the propagation matrix will be computed differently. The propagation matrix will have the size: $\text{range}^2 + 1$.

Let's note **P** for the propagation value, **V** for the variation, **R** for the range and **N** for the number of neighbors.

- With diffusion propagation

For diffusion propagation, we compute following the following steps:

(1) We determine the "minimale" matrix according to N (if N = 8, the matrix will be

$\begin{bmatrix} [P/9, P/9, P/9] & [P/9, 1/9, P/9] & [P/9, P/9, P/9] \end{bmatrix}$). if N = 4, the matrix will be $\begin{bmatrix} [0, P/5, 0] & [P/5, 1/5, P/5] & [0, P/5, 0] \end{bmatrix}$).

(2) If R != 1, we propagate the matrix R times to obtain a $[2^*R+1][2^*R+1]$ matrix (same computation as for `cycle_length`).

(3) If V != 0, we subtract each value by $V^*DistanceFromCenter$ ($DistanceFromCenter$ depends on N).

Ex with the default values (P=1, R=1, V=0, N=8):

- **With gradient propagation**

The value of each cell will be equal to **P/POW(N,DistanceFromCenter)-DistanceFromCenter*V**. ($DistanceFromCenter$ depends on N).

Ex with R=2, other parameters default values (R=2, P=1, V=0, N=8):

$$\text{size} = 2^R + 1 = 5$$

$1/\text{pow}(8,2)-0*0 = 1/64$				
$1/\text{pow}(8,2)-0*0 = 1/64$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,2)-0*0 = 1/64$
$1/\text{pow}(8,2)-0*0 = 1/64$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,0)-0*0 = 1$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,2)-0*0 = 1/64$
$1/\text{pow}(8,2)-0*0 = 1/64$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,1)-0*0 = 1/8$	$1/\text{pow}(8,2)-0*0 = 1/64$
$1/\text{pow}(8,2)-0*0 = 1/64$				

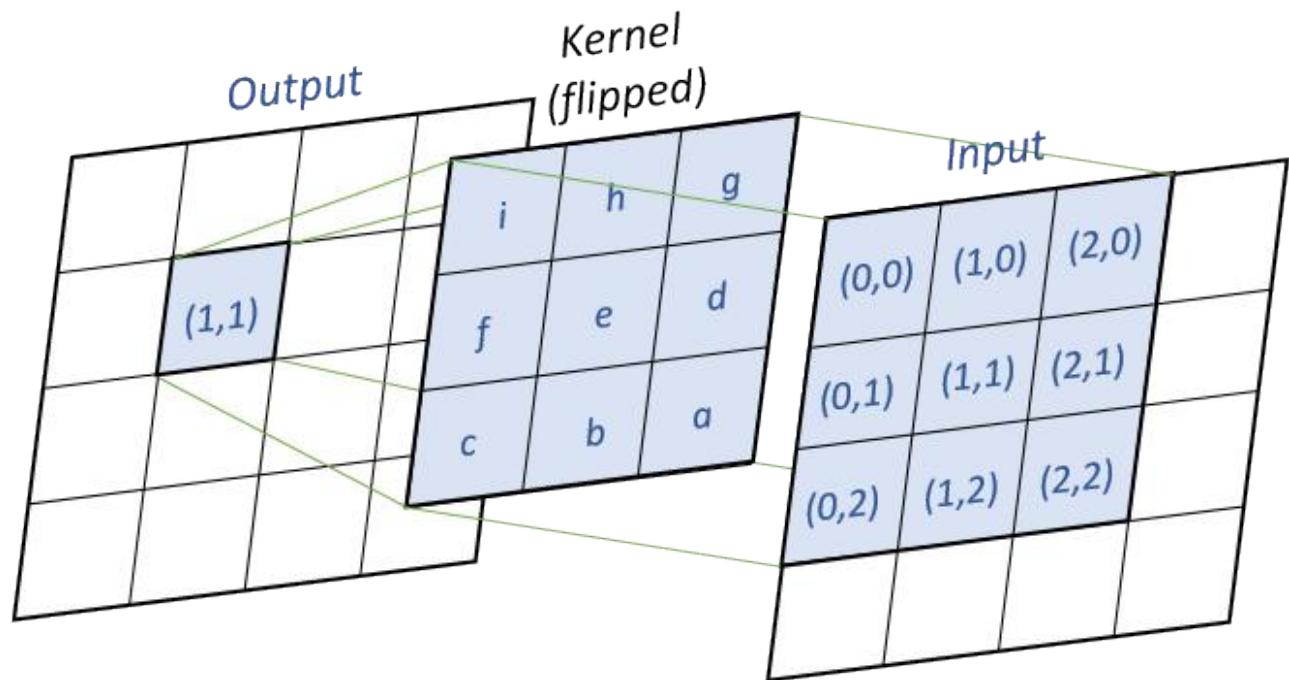
Note that if you declared a diffusion matrix, you cannot use those 3 facets (it will raise a warning). Note also that if you use parameters, you will only have a uniform matrix.

Computation methods

You can compute the output matrix using two computation methods by using the facet `method` : the dot product and the convolution. Note that the result of those two methods is exactly the same (except if you use the `avoid_mask` facet, the results can be slightly different between the two computations).

Convolution

`convolution` is the default computation method for diffusion. For every output cells, we will multiply the input values and the flipped kernel together, as shown in the following image :

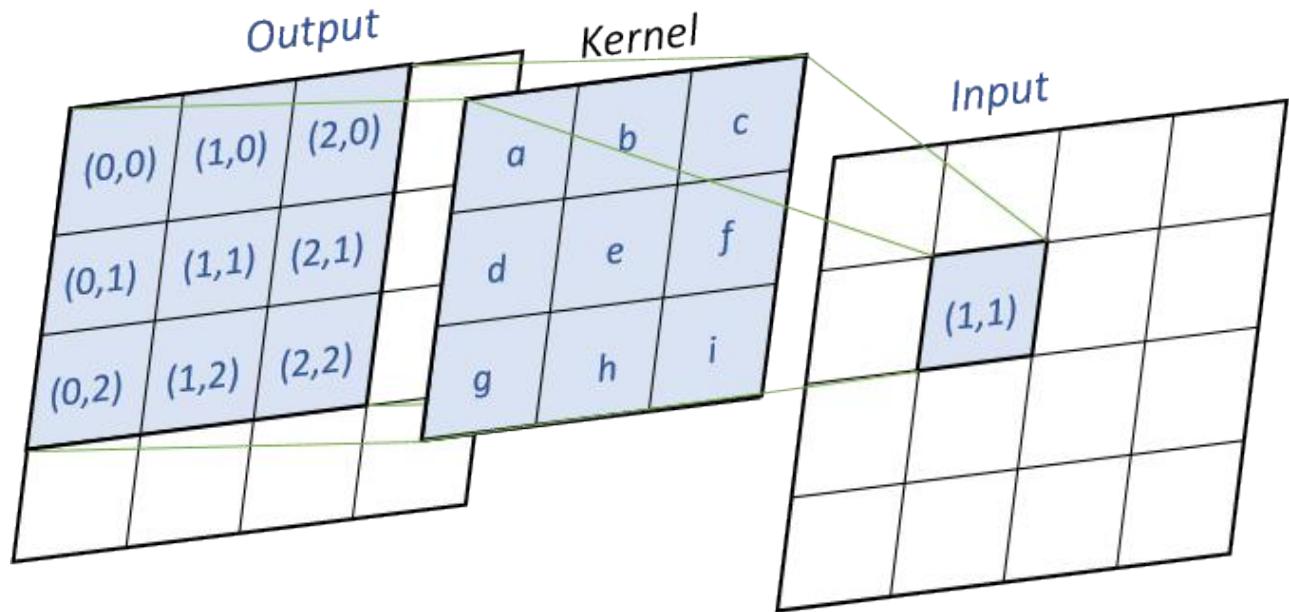


Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
    for (j = 0 ; j < y.nbCols ; j++)
        for (m = 0 ; m < k.nbRows ; m++)
            for (n = 0 ; n < k.nbCols ; n++)
                y[i,j] += k[k.nbRows - m - 1, k.nbCols - n - 1]
                    * x[i - k.nbRows/2 + m, j - k.nbCols/2 + n]
```

Dot Product

`dot_product` method will compute the matrix using a simple dot product between the matrix. For every input cells, we multiply the cell by the kernel matrix, as shown in the following image :



Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
    for (j = 0 ; j < y.nbCols ; j++)
        for (m = 0 ; m < k.nbRows ; m++)
            for (n = 0 ; n < k.nbCols ; n++)
                y[i - k.nbRows/2 + m, j - k.nbCols/2 + n] += k[m, n] * x[i, j]
```

Using a mask

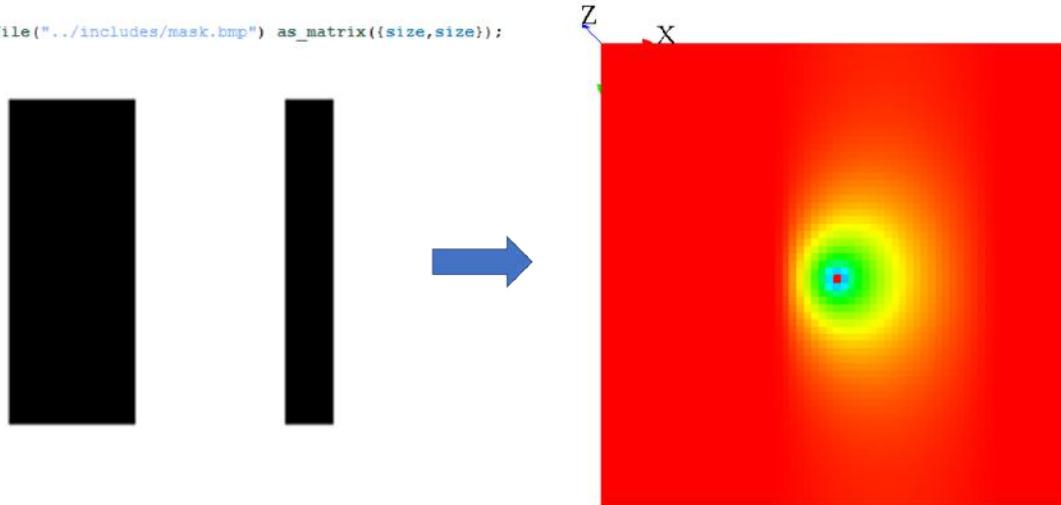
Generalities

If you want to propagate some values in a heterogeneous grid, you can use some mask to forbid some cells to propagate their values.

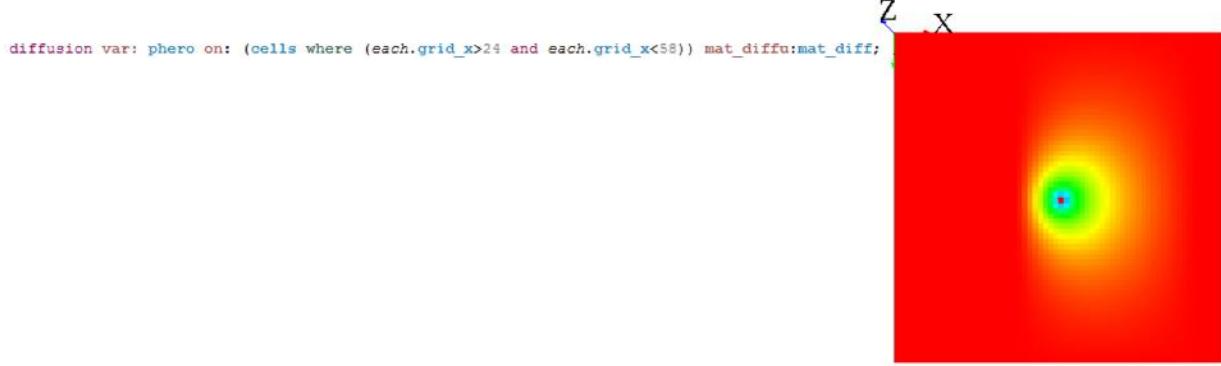
You can pass a matrix to the facet `mask`. All the values smaller than `-1` will not propagate, and all the values greater or equal to `-1` will propagate.

A simple way to use mask is by loading an image :

```
matrix mymask <- file("../includes/mask.bmp") as_matrix({size,size});
```



Note that when you use the `on` facet for the `diffuse` statement, you can choose only some cells, and not every cell. In fact, when you restrain the values to be diffuse, it is exactly the same process as if you were defining a mask.



Uniform diffusion only applied on
cells where : $24 < \text{grid_x} < 58$

When your diffusion is combined with a mask, the default behavior is that the non-masked cells will diffuse their values in **all** existing cells (that means, even the masked cells !). To change this behavior, you can use the facet `avoid_mask`. In that case, the value which was supposed to be affected to the masked cell will be redistributed to the neighboring non-masked cells.

Tips

Masks can be used to simulate a lot of environments. Here are some ideas for your models:

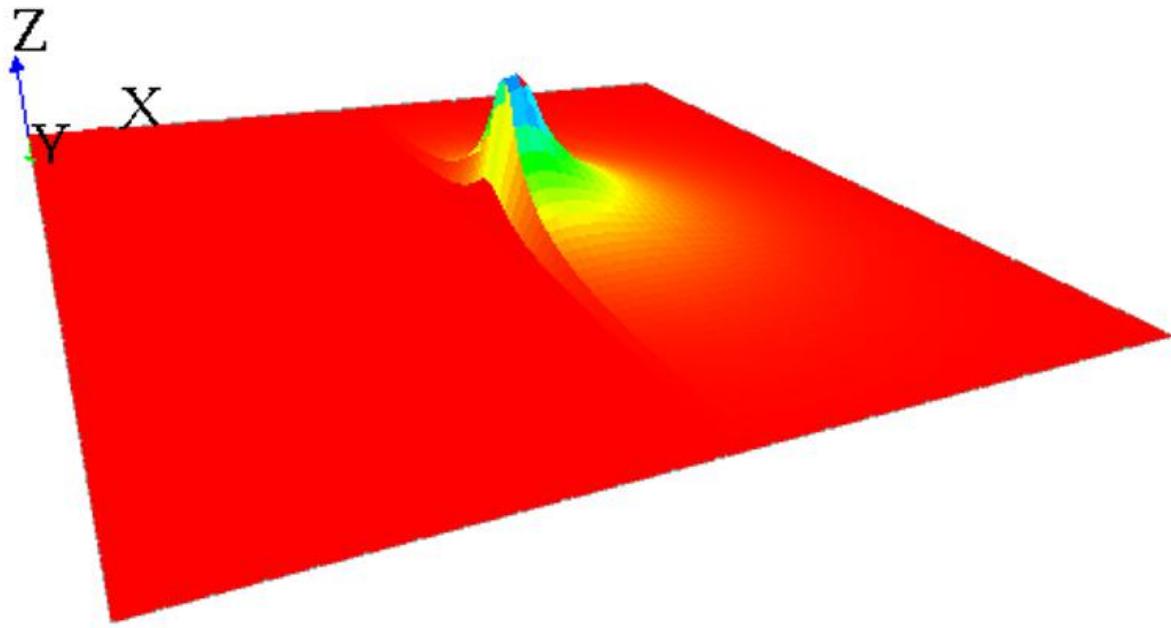
Wall blocking the diffusion

If you want to simulate a wall blocking a uniform diffusion, you can declare a second diffusion matrix that will be applied only on the cells where your wall will be. This diffusion matrix will "push" the values outside from himself, but conserving the values (the sum of the values of the diffusion still have to be equal to 1) :

```

matrix<float> mat_diff <- matrix([
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]]);

```

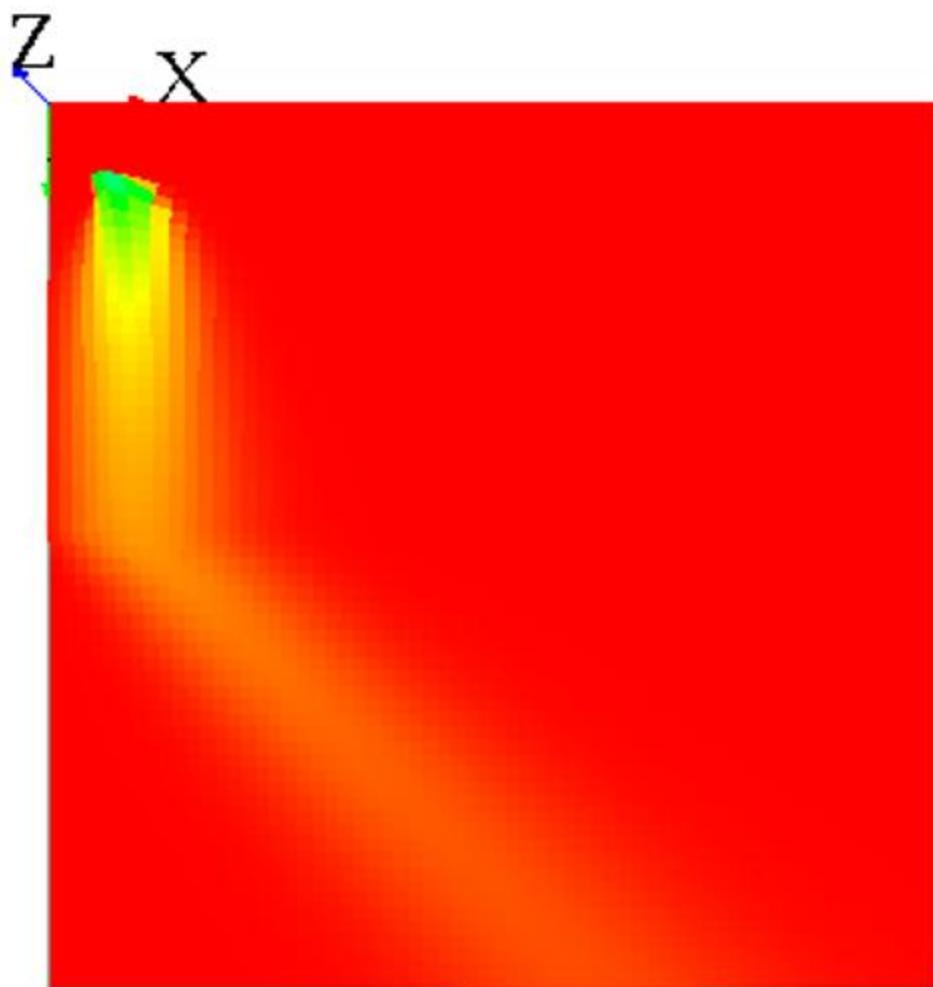


Note that almost the same result can be obtained by using the facet `avoid_mask`: the value of all masked cells will remain at 0, and the value which was supposed to be affected to the masked cell will be distributed to the neighboring cells. Notice that the results can be slightly different if you are using the `convolution` or the `dot_product` method: the algorithm of redistribution of the value to the neighboring cells is a bit different. We advise you to use the `dot_product` with the `avoid_mask` facet, the results are more accurate.

Wind pushing the diffusion

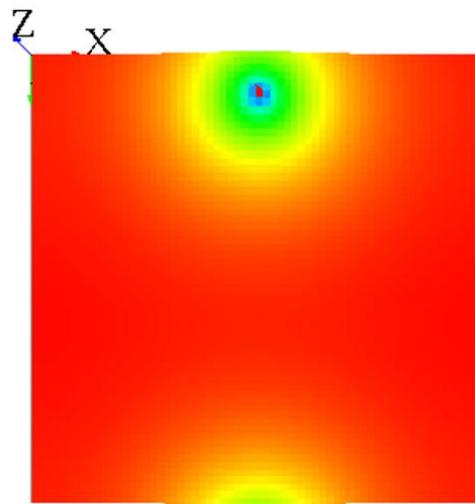
Let's simulate a uniform diffusion that is pushed by a wind from "north" everywhere in the grid. A wind from "west" as blowing at the top side of the grid. We will here have to build 2 matrices: one for the uniform diffusion, one for the "north" wind and one for the "west" wind. The sum of the values for the 2 matrices meant to simulate the wind will be equal to 0 (as it will be added to the diffusion matrix).

```
matrix<float> mat_diff <- matrix([
```

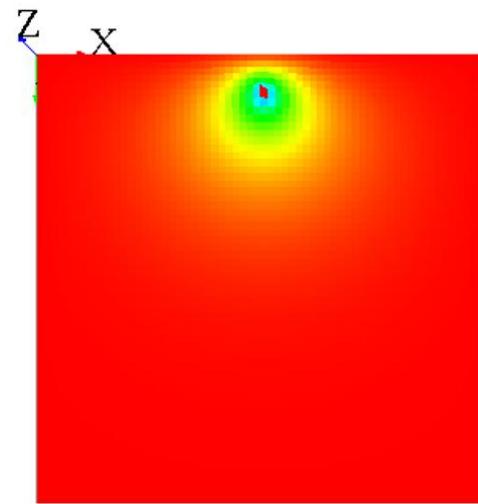


Endless world

Note that when your world is not a torus, it has the same effect as a *mask*, since all the values outside from the world cannot diffuse some values back :



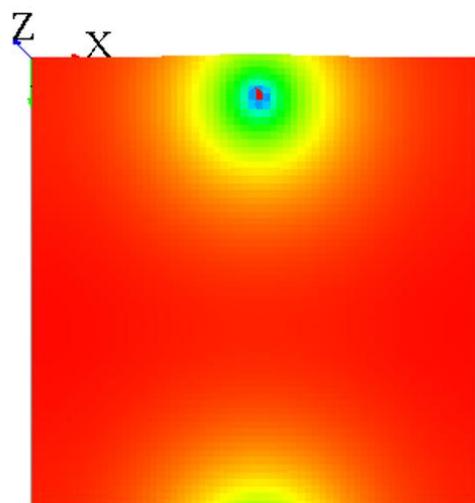
Uniform diffusion in a torus world
(after 500 cycles)



Uniform diffusion in a non-torus
world (after 500 cycles)

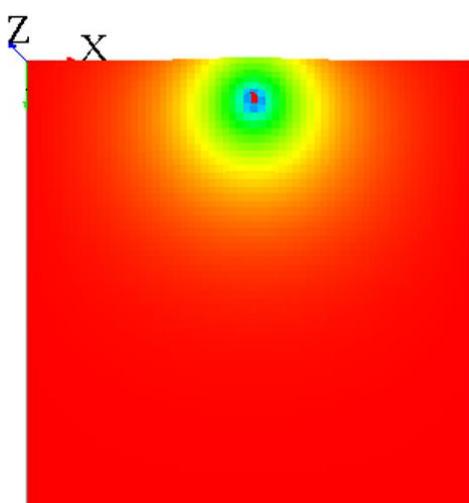
\neq

You can "fake" the fact that your world is endless by adding a different diffusion for the cells with `grid_x=0` to have almost the same result :



Uniform diffusion in a torus world
(after 500 cycles)

\approx



Uniform diffusion in a non-torus
world with simulation of endless
world (after 500 cycles)

```

matrix<float> mat_diff <- matrix([
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]]);

matrix<float> mat_diff_upper_edge <- matrix([
    [0.0, 0.0, 0.0],
    [1/9+7/81, 2/9+1/81, 1/9+7/81],
    [1/9, 1/9, 1/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_y>0))
matrix:mat_diff;
    diffuse var: phero on: (cells where(each.grid_y=0))
matrix:mat_diff_upper_edge;
}

```

Pseudo-code

This section is more for a better understanding of the source code.

Here is the pseudo-code for the computation of diffusion :

1) : Execute the statement `diffuse`, store the diffusions in a map (from class `DiffusionStatement` to class `GridDiffuser`) :

- Get all the facet values
- Compute the "real" mask, from the facet "mask:" and the facet "on:".
 - If no value for "mask:" and "on:" all the grid, the mask is equal to null.
- Compute the matrix of diffusion
 - If no value for "matrix:", compute with "nb_neighbors", "is_gradient", "proportion", "propagation", "variation", "range".
 - Then, compute the matrix of diffusion with "cycle_length".

2) : At the end of the step, execute the diffusions (class *GridDiffuser*) :

- For each key of the map,
 - Load the couple "var_diffu" / "grid_name"
 - Build the "output" and "input" array with the dimension of the grid.
 - Initialize the "output" array with -Double.MAX_VALUE.
 - For each value of the map for that key,
 - Load all the properties : "method_diffu", "is_gradient", "matrix", "mask", "min_value"
 - Compute :
 - If the cell is not masked, if the value of input is > min_value, diffuse to the neighbors.
 - If the value of the cell is equal to -Double.MAX_VALUE, replace it by input[idx] * matDiffu[i][j].
 - Else, do the computation (gradient or diffusion).
 - Finish the diffusion :
 - If output[idx] > -Double.MAX_VALUE, write the new value in the cell.

Version: 1.9.1

Using Database Access

Database features of GAMA provide a set of actions on Database Management Systems (DBMS). 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.

These features are implemented in two kinds of components: skill (`SQLSKILL`) and agent (`AgentDB`).

`SQLSKILL` and `AgentDB` provide almost the same features (the 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 and destroyed when it is killed.
- In contrast, an agent of a species with the `SQLSKILL` skill will open a connection each time it wants to execute a query. This means that each action will be composed of three running steps:
 - Make a database connection.
 - Execute an SQL statement.
 - Close database connection.

An agent with the `SQLSKILL` spends a lot of time to create/close the connection each time it needs to send a query; it saves the database connection (a DBMS often limits the number of simultaneous connections). In contrast, an `AgentDB`

agent only needs to establish one database connection that can be used for any action. Because it does not need to create and close the database connection for each action, therefore actions of `AgentDB` agents are executed faster than the 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 to create agents, define the environment, or analyze or store simulation results in RDBMS. The database features help us to have more flexibility in the 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
- PostgreSQL: **The GIS extension needs to be installed and activated in the database.**

Note that, MySQL and Postgres DBMSs require a dedicated server to work while SQLite only needs a file to be accessed.

All the actions are independent from the chosen DBMS. Only the connection

parameters are DBMS-dependent.

We chose to implement 3 main query actions:

- `select`: that will execute the SELECT SQL queries. It will return a result dataset.
- `insert`: that will execute the INSERT SQL queries. It will return the number of records that are affected by the SQL query.
- `executeUpdate`: that can execute any CREATE/INSERT/DROP/DELETE SQL queries (basically all the queries that do not return a dataset. It generalizes the `insert` action.

SQLSKILL skill

Define a species that uses the `SQLSKILL` skill

Example of declaration:

```
species toto skills: [SQLSKILL] {  
    //insert your descriptions here  
}
```

Agents with such a skill can use new actions defined in the skill. All these actions need information for the database connection.

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 *key::value* pairs with the following **keys**:

- `dbtype` (mandatory): 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"

(ignore case sensitive)

- `host` (optional): Host name or IP address of data server. It is absent when we work with SQLite.
- `port` (optional): Port of connection. It is not required when we work with SQLite.
- `database` (mandatory): Name of database. It is the file name including the path when we work with SQLite.
- `user` (optional): Username. It is not required when we work with SQLite.
- `passwd` (optional): Password. It is not required when we work with SQLite.
- `srid` (optional): 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.

Example: Definitions of connection parameters

```
// POSTGRES connection parameter
map <string, string> POSTGRES <- [
    'host'::'localhost',
    'dbtype'::'postgres',
    'database'::'BPH',
    'port'::'5432',
    'user'::'postgres',
    'passwd'::'abc'];

//SQLite
map <string, string> SQLITE <- [
    'dbtype'::'sqlite',
    'database'::'../includes/meteo.db'];

// MySQL connection parameter
map <string, string> MySQL <- [
    'host'::'localhost',
    'dbtype'::'MySQL',
    'database'::'', // it may be an empty string
```

Action `testConnection`: test a connection to a database

Syntax:

`testConnection (params: map <string, string>)` 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 the given name and password).
 - `false`: the agent cannot connect (either the server is not started, the database does not exist or the user/password are not correct).
- **Arguments:**
 - `params` (type = `map <string, string>`): map of connection parameters
- **Exceptions:** `GamaRuntimeException`

Example: Check a connection to a MySQL database.

```
// Needs to be executed in the context of an agent with the SQLSKILL skill.  
// MySQL is the connection parameters map defined above.  
if (testConnection(MySQL)){  
    write "Connection is OK" ;  
}else{  
    write "Connection is false" ;  
}
```

Action `select`: select data from a database

Syntax:

select (*params*: map <string, string>, *select*: string, *values*: 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 names.
 - The second element is a list of column types.
 - The third element is a data set.
- **Arguments:**
 - *params* (type = map<string, string>): map containing the connection parameters
 - *select* (type = string): A SQL query returning values, i.e. a SELECT query. The selection query can be a parametric query (i.e. it can contain question marks).
 - *values* (type = list): List of values that are used to replace question marks. 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 <- select(PARAMS, "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 <- select(params: PARAMS,
select: "SELECT temp_min FROM points where
```

Action `insert`: Insert data into a database

Syntax:

`insert (param: map<string,string>, into: string, columns: list<string>, values: 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<string,string>`): map containing the connection parameters. `into` (type = string): the table name. `columns` (type=`list<string>`): list of column names of the table. It is an optional argument. If it is not specified then all columns of table are selected. `values` (type=list): list of values that are used to insert into the table chosen 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']);
```

Action `executeUpdate`: Execution update commands

Syntax:

`executeUpdate` (`param: map<string,string>, updateComm: string, columns: list<string>, values: 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 or the update command fails then it throws a `GamaRuntimeException`. Otherwise, it returns an integer value.

- **Return:** int. It returns a number of records affected by the SQL query.
- **Arguments:**
 - `params` (type = `map<string,string>`): map containing the connection parameters,
 - `updateComm` (type = `string`): SQL query string. It may be one of the SQL commands: *create*, *update*, *delete* and *drop* with or without question marks.
 - `columns` (type=`list<string>`): list of column names of the 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` to execute some 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"
```

AgentDB

`AgentBD` is a built-in species, which supports behaviors that look like actions in `SQLSKILL` but differs in that it uses only one connection for several actions. It means that `AgentDB` creates a connection to the database and keeps that connection open for its later operations.

Define a species that is an inheritance of `AgentDB`

Example of declaration:

```
species agtDB parent: AgentDB {  
    //insert your descriptions here  
}
```

Action `connect`: Connect to a database

Syntax:

connect (*params: map<string,string>*) This action makes a connection to the database. If a connection is established then it will assign the connection object into a built-in attribute of the species (conn) otherwise it throws a GamaRuntimeException.

- **Return:** connection
- **Arguments:**
 - `params` (type = `map<string,string>`): 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 agtDB {
    do connect (params: POSTGRES);
}
```

Action `isConnected`: Check whether an agent is connected to a database

Syntax:

isConnected (*params: map<string,string>*) This action checks if an agent is connected to a database.

- **Return:** Boolean. If the agent is connected to a database then isConnected returns true; otherwise it returns false.
- **Arguments:**
 - `params` (type = *map<string,string>*): map containing the connection parameters

Example: Check whether the agents agtDB are connected.

```
ask agtDB {
    if (self isConnected){
        write "It already has a connection";
```

Action `close`: Close the current connection

Syntax:

`close` This action closes the current database connection of the current agent. If the agent does not have any database connection then it throws a GamaRuntimeException.

Example: close the connection of all the agtDB agents.

```
ask agtDB {  
    if (self.isConnected()){  
        do close;  
    }  
}
```

Action `getParameter`: Get connection parameters

Syntax:

`getParameter` This action returns the connection parameters of the current agent.

- **Return:** `map<string, string>`

Example:

```
ask agtDB {  
    if (self.isConnected()){  
        write "the connection parameter: " +self.getParameter();  
    }  
}
```

Set connection parameters

Syntax:

setParameter (*params: map<string,string>*) This action sets new values for connection parameters and closes the current connection of the agent. 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<string,string>*): map containing the connection parameters
- **Exceptions:** *GamaRuntimeException*

Example:

```
ask agtDB {
    if (self.isConnected()){
        do setParameter params: MySQL;
        do connect params: self.getParameter();
    }
}
```

Retrieve data from a database by using `AgentDB`

Because `AgentDB`'s connection to the database is kept alive, it can execute several SQL queries using only the `connect` action once. Hence `AgentDB` can do actions such as `select`, `insert`, `executeUpdate` with the same parameters as those of `SQLSKILL` *except for the **params** parameter which is always absent*.

Examples:

```
map<string, string> PARAMS <- ['dbtype':'sqlite',
'database':'../../includes/Student.db'];
ask agtDB {
    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 <- self.select("SELECT * FROM registration;");

    //update
    int n <- self.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 the environment and create agents

In GAMA, it is possible to initialize the simulations from data stored in a database: we can use the results of the `Select` action of `SQLSKILL` or `AgentDB` to create agents or to define the boundary of the environment in the same way we do with shape files. Further more, we can also save simulation data that are generated by the simulation including geometry data.

Note that GAMA only supports PostGIS and MySQL as spatial DBMS.

Define the boundary of the environment from the database

- **Step 1:** specify the SELECT query by declaring a map object with keys as below:
 - `dbtype` (mandatory): 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" (ignore case sensitive)
 - `host` (optional): Host name or IP address of data server.
 - `port` (optional): Port of connection.
 - `database` (mandatory): Name of database.
 - `user` (optional): Username.
 - `passwd` (optional): Password.
 - `srid` (optional): srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in *Preferences->External* configuration.

- `select` (mandatory): selection query.

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 the boundary of the environment by using the map object defined in the first step (in the `global` block of the model).

```
geometry shape <- envelope(BOUNDS);
```

Note: We can do the same way if we work with MySQL 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 database data. We can do as below:

- **Step 1:** Define a species with `SQLSKILL` or `AgentDB`

```
species DB_accessor skills: SQLSKILL {
    //insert your descriptions here
}
```

- **Step 2:** Define a connection and selection parameters

```
global {
    map<string, string> PARAMS <- [ //'srid':'32648', // optional
        'host':'localhost',
        'dbtype':'postgis',
        'database':'spatial_db',
        'port':'5432',
        'user':'postgres',
        'passwd':''];
    string QUERY <- "SELECT type, ST_AsEWKB(geom) as geom FROM
buildings;";
}
```

- **Step 3:** Create species by using selected results

```
init {
    create DB_accessor {
        create buildings from: select(PARAMS, QUERY)
            with:[ nature::"type", shape::"geom"];
    }
    ...
}
```

Save Geometry data to database

Saving agents in a database will be simply a set of insertion into the database. We can do as below:

- **Step 1:** Define a species with SQLSKILL or AgentDB

```
species DB_accessor skills: SQLSKILL {
    //insert your descriptions here
```

- **Step 2:** Define a connection and create GIS database and tables

```

global {
    map<string, string> PARAMS <- [ // 'srid'::'4326', // optional
        'host'::'localhost', 'dbtype'::'postgres', 'database'::'spatial_db',
        'port'::'5432', 'user'::'postgres', 'passwd'::'' ];

    init {
        create DB_accessor ;
        ask DB_accessor {
            if (self.testConnection(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 the GIS database

```
ask building {
  ask DB_Accessor {
    do insert(params: PARAMS,
              into: "buildings",
              columns: ["name", "type", "geom"],
              values: [myself.name, myself.type, myself.shape];
  }
}
```

Version: 1.9.1

Using FIPA ACL

GAMA allows modelers to provide agents the capability to communicate with other agents using [FIPA](#) Communication Acts (such as inform, request, call for proposal...) and [Interaction Protocols](#) (such [Contract Net Interaction Protocol](#), [Request Interaction Protocol](#)).

To add these capabilities to the chosen species, the modeler needs to attach the [fipa skill](#): it adds to agents of the species some additional attributes (e.g. the list of messages received) and available actions (e.g. the possibility to send messages given the chosen Communication Act).

The exhaustive list of available Communication Acts and Interaction Protocols is available from the technical description of the [fipa skill page](#). Examples can be found in the model library bundled with GAMA ([Plugin models / FIPA Skill](#)).

Table of Contents

- Main steps to create a conversation using FIPA Communication Acts and Interaction Protocols
- Attach the fipa skill to a species
- Initiate a conversation
- Receive messages
- Reply to a received message
- The [message](#) data type
- The [conversation](#) data type

Main steps to create a conversation using FIPA Communication Acts and Interaction Protocols

1. Attach the skill `fipa` to the agents' species that need to use Communication Acts
2. An initiator agent starts a conversation with some agents: it chooses the Interaction Protocol and starts it by sending the first Communication Acts of the protocol
3. Each agent involved in the conversation needs to check its received messages and respond to them by choosing the appropriate Communication Act.

Attach the `fipa` skill to a species

To attach the `fipa` skill to a species, the modeler has to add it in the `skills` facet of the `species` statement ([in a way similar to any other skill](#)).

```
species any_species skills: [fipa] {  
    ...  
}
```

Agents of any species can communicate in the same conversation. The only constraint is that they need to have the capabilities to receive and send messages, i.e. to have the skill `fipa`.

Species can have several attached skills: a single species can be provided with both the `moving` and `fipa` skills (and any other ones).

This skill adds to every agent of the species:

- some additional attributes:
 - `conversations` is the list of the agent's current conversations,
 - `mailbox` is the list of messages of all types of performatives,
 - `requests`, `informs`, `proposes`... are respectively the list of the 'request', 'inform', 'propose' performative messages.
- some additional actions, such as:
 - `inform`, `accept_proposal`... that replies a message with an 'inform' (respectively 'accept_proposal' performative message).
 - `start_conversation` that starts a conversation with a chosen interaction protocol.
 - `end_conversation` that replies a message with an 'end_conversation' performative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsibility of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol.
 - `reply` that 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, please use the 'action' with the same name as the 'performative'. For example, to reply a message with a 'request' performative message, the modeler should use the 'request' action.

Initiate a conversation

An interaction using an Interaction Protocol starts with the creation of a conversation by an agent, using the `start_conversation` action.

The modeler specifies the chosen **protocol** (facet `protocol`), **list of participants** (facet `to`), **communication act** (facet `performative`) and **message** (facet `contents`).

```

species Initiator skills: [fipa] {
    reflex send_propose_message when: (time = 1) {
        do start_conversation to: [p] protocol: 'fipa-propose'
    performative: 'propose' contents: ['Go swimming?'] ;
    }
}

```

Receive messages

Each agent (with the `fipa` skill) is provided with several "mailbox" attributes filtering the various received messages by communication act: e.g. `proposes` contains the list of the received messages with the "Propose" communication act.

Receiving a message consists thus in looking at each message from the mailbox, and acting in accordance with its contents, participants...

Important remark: once the `contents` field of a received message has been read, it is removed from all the lists it appears in.

```

species Initiator skills: [fipa] {
    reflex read_accept_proposals when: !(empty(accept_proposals)) {
        write name + ' receives accept_proposal messages';
        loop i over: accept_proposals {
            write 'accept_proposal message with content: ' +
        string(i.contents);
        }
    }
}

species Participant skills: [fipa] {
    reflex accept_proposal when: !(empty(proposes)) {
        message proposalFromInitiator <- proposes at 0;

        do accept_proposal message: proposalFromInitiator contents:
    }
}

```

Remark:

- To test that the agent has received a new message is simply done by testing whether the dedicated mailing box contains messages.
- To get a message, the modeler can either loop over the message list to get all the messages or get a message by its index in the message box.

Reply to a received message

Given the message it has received, an agent can reply using the appropriate Communication Act (using the appropriate action). It simply has to specify the message to which it replies and the content of the reply.

Note that it does not need to specify the receiver as it is contained in the message.

```
species Participant skills: [fipa] {
    reflex accept_proposal when: !(empty(proposes)) {
        message proposalFromInitiator <- proposes at 0;

        do accept_proposal message: proposalFromInitiator contents:
        ['OK! It \\'s hot today!'] ;
    }
}
```

End a conversation

When a conversation is made in the scope of an Interaction Protocol, it is ended automatically when the last Communicative Act has been sent.

In the case of a 'no-protocol conversation', it is the responsibility of the modeler to explicitly send the `end_conversation` message to mark the end of a conversation/

interaction protocol.

When a conversation ends, it is automatically removed from the list `conversations`.

The `message` type

The agents' mailbox is defined as a list of messages. Each message is a GAML object of type `message`. An exhaustive description of this type is provided in the dedicated [GAML Data Types page](#).

A `message` object is defined by a set of several fields, such as:

- `contents` (type `unknown`): the content of the message
- `sender` (type `unknown`): the sender of the message. In the case where the sender is an agent, it is possible to get the corresponding agent with `agent(m.sender)` (where `m` is the considered message).
- `unread` (type `bool`): specify whether the message has been read.
- `emission_timestamp` (type `int`)
- `reception_timestamp` (type `int`)

The `conversation` data type

The agents' `conversations` contain the list of the conversations in which the agent takes part. Each conversation is a GAML object of type `conversation` that contains the list of messages exchanged, the protocol, initiator... An exhaustive description of this type is provided in the dedicated [GAML Data Types page](#).

A `conversation` object is defined by a set of several fields, such as:

- `messages` (type = list of messages): the list of messages that compose this conversation

- `protocol` (type = string): the name of the protocol followed by the conversation
- `initiator` (type = agent): the agent that has initiated this conversation
- `participants` (type = list of agents): the list of agents that participate in this conversation
- `ended` (type = bool): whether this conversation has ended or not

Version: 1.9.1

Using BEN (simple_bdi)

Introduction to BEN

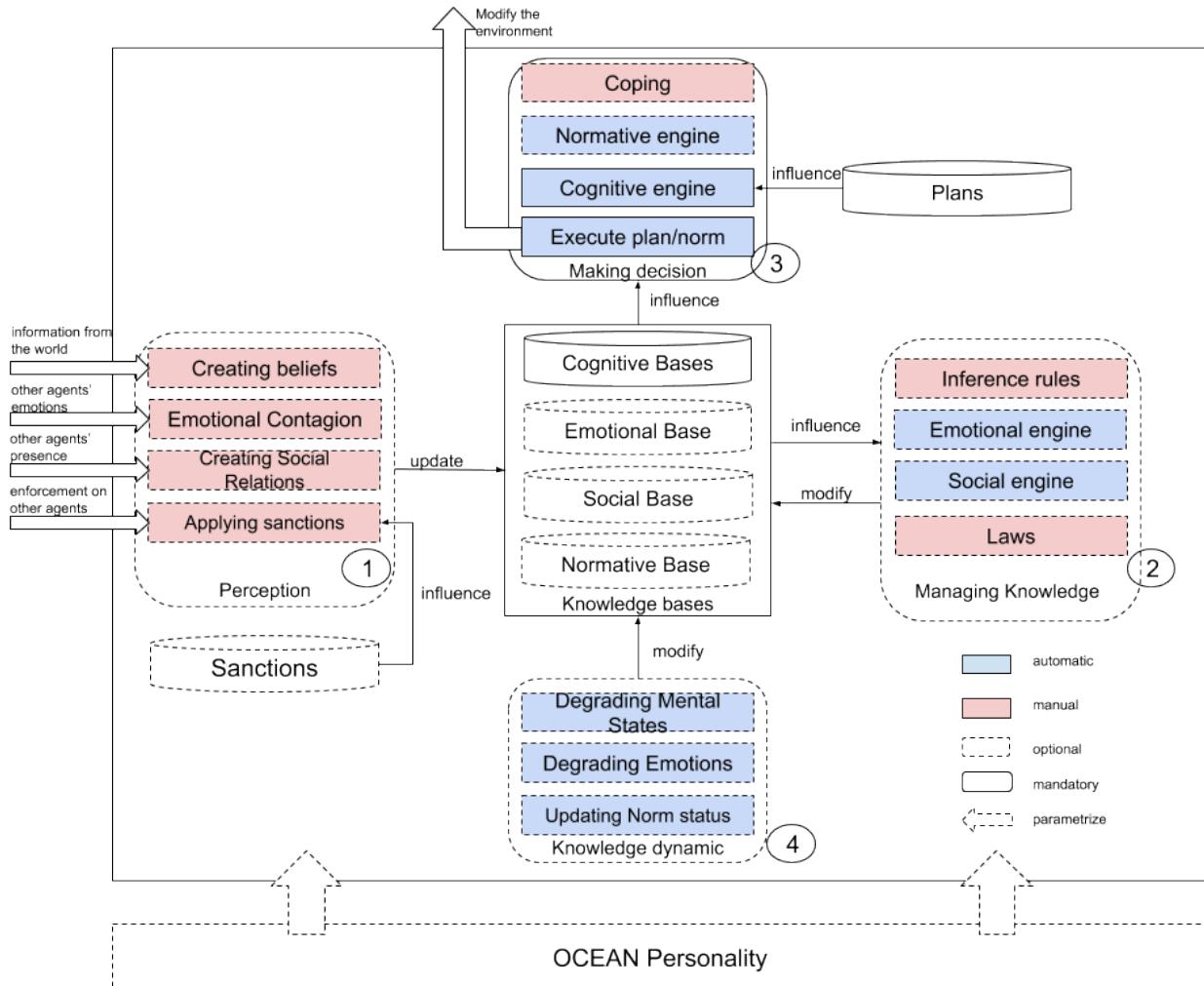
BEN (Behavior with Emotions and Norms) is an agent architecture providing social agents with cognition, emotions, emotional contagion, personality, social relations, and norms. This work has been done during the Ph.D. of Mathieu Bourgais, funded by the ANR ACTEUR.

The BEN architecture is accessible in GAMA through the use of the `simple_bdi` architecture when defining agents. This page indicates the theoretical running of BEN as well as the practical way it has been implemented in GAMA.

This page features all the descriptions for the running of the BEN architecture. This page is updated with the version of BEN implemented in GAMA. To get more details on its implementation in GAMA, see [operators related to BDI](#), [BDI tutorial](#) or [BDI built-in architecture reference](#).

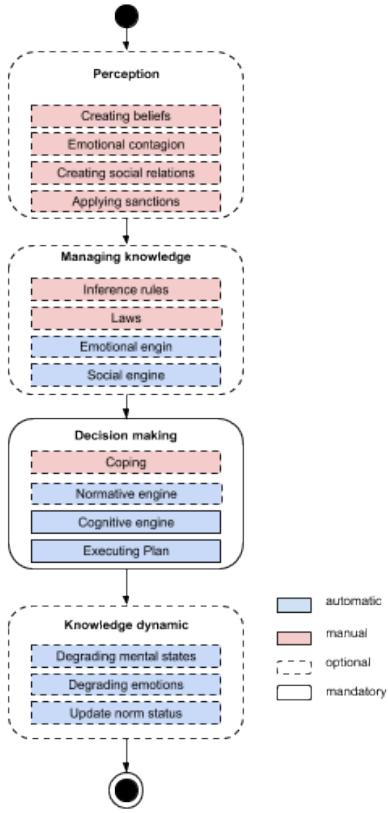
The BEN architecture

The BEN Architecture used by agents to make a decision at each time step is represented by the image right below:



Each social agent has its own instance of the BEN architecture to make a decision. The architecture is composed of 4 main parts connected to the agent's knowledge bases, seated on the agent's personality. Each part is made up of processes that are automatically computed (in blue) or which need to be manually defined by the modeler (in pink). Some of these processes are mandatory (in solid line) and some others are optional (in dotted line). This modularity enables each modeler to only use components that seem pertinent to the studied situation without creating heavy and useless computations.

The Activity diagram bellow shows the order in which each module and each process is activated. The rest of this page explains in details how each process from each module works and what is the difference between the theoretical architecture and its implementation.



Predicates, knowledge and personality

In BEN, an agent represents its environment through the concept of predicates.

A predicate represents information about the world. This means it may represent a situation, an event or an action, depending on the context. As the goal is to create behaviors for agents in a social environment, that is to say taking actions performed by other agents into account with facts from the environment in the decision making process, an information P caused by an agent j with an associated list of value V is represented by $\mathbf{P}_j(V)$. A predicate \mathbf{P} represents an information caused by any or none agent, with no particular value associated. The opposite of a predicate P is defined as **not P**.

In GAML, the simplebdi architecture adds a new type called `_predicate` which is made of a name (mandatory), a map of values (optional) an agent causing it (optional) and a truth value (optional, by default at true). To manipulate these predicates, there are operators like `set_agent_cause`, `set_truth`, `with_values` and `add_values` to modify the corresponding attribute of a given predicate (`with_values` changes all the map of values while `add_values` enables to add a new value without changing the rest of the map). These values can be accessed with operators `get_agent_cause`, `get_truth`, `get_values`. An operator `not` is also defined for predicates.

Below is an example of how to define predicates in GAML:

```
predicate a <- new_predicate("test");
predicate b <- new_predicate("test", ["value1"]::10];
predicate c <- new_predicate("test",agentBob);
predicate d <- new_predicate("test",false);
predicate e <- new_predicate("test",agenBob,false);
```

Cognitive mental states

Through the architecture, an agent manipulates cognitive mental states to make a decision; they constitute the agent's mind. A cognitive mental state possessed by the agent i is represented by $M_i(PMEM, Val, Li)$ with the following meaning:

- **M**: the modality indicating the type of the cognitive mental state (e.g. a belief).
- **PMEm**: the object with which the cognitive mental state relates. It can be a predicate, another cognitive mental state, or an emotion.
- **Val**: a real value which meaning depends on the modality.
- **Li**: a lifetime value indicating the time before the cognitive mental state is forgotten.

A cognitive mental state with no particular value and no particular lifetime is written $M_i(PMEM)$. $Val[M_i(PMEM)]$ represents the value attached to a particular cognitive mental state and $Li[M_i(PMEM)]$ represents its lifetime.

The cognitive part of BEN is based on the BDI paradigm (Bratman, 1987) in which agents have a belief base, a desire base and an intention base to store the cognitive mental states about the world. In order to connect cognition with other social features, the architecture outlines a total of 6 different modalities which are defined as follows:

- **Belief**: represents what the agent knows about the world. The value attached to this mental state indicates the strength of the belief.
- **Uncertainty**: represents an uncertain information about the world. The value attached to this mental state indicates the importance of the uncertainty.
- **Desire**: represents a state of the world the agent wants to achieve. The value attached to this mental state indicates the priority of the desire.
- **Intention**: represents a state of the world the agent is committed to achieve. The value attached to this mental state indicates the priority of the intention.
- **Ideal**: represents an information socially judged by the agent. The value attached to this mental state indicates the praiseworthiness value of the ideal about P. It can be positive (the ideal about P is praiseworthy) or negative (the ideal about P is blameworthy).

- **Obligation:** represents a state of the world the agent has to achieve. The value attached to this mental state indicates the priority of the obligation.

In GAML, mental states are manipulated thanks to add, remove and get actions related to each modality: `add_belief`, `remove_belief`, `get_belief`, `add_desire`, `remove_desire` ... Then, operators enables to acces or modify each attribute of a given mental state: `get_predicate`, `set_predicate`, `get_strength`, `set_strength`, `get_lifetime`, `set_lifetime`, etc.

Below is an exemple of code in GAML concerning cognitive mental states:

```
reflex testCognition{
    predicate a <- new_predicate("test");
    do add_belief(a,strength1,lifetime1);
    mental_state b <- get_uncertainty(a);
    int c <- get_lifetime(b);
}
```

Emotions

In BEN, the definition of emotions is based on the OCC theory of emotions (Ortony, 90). According to this theory, an emotion is a valued answer to the appraisal of a situation. Once again, as the agents are taken into consideration in the context of a society and should act depending on it, the definition of an emotion needs to contain the agent causing it. Thus, an emotion is represented by **Em_i(P,Ag,I,De)** with the following elements :

- **Em_i:** the name of the emotion felt by agent *i*.
- **P:** the predicate representing the fact about which the emotion is expressed.
- **Ag:** the agent causing the emotion.
- **I:** the intensity of the emotion.
- **De:** the decay withdrawal from the emotion's intensity at each time step.

An emotion with any intensity and any decay is represented by **Em_i(P,Ag)** and an emotion caused by any agent is written **Em_i(P)**. **I[Em_i(P,Ag)]** stands for the intensity of a particular emotion and **De[Em_i(P,Ag)]** stands for its decay value.

In GAML, emotions are manipulated thanks to `add_emotion`, `remove_emotion` and `get_emotion` actions and attributes of an emotion are manipulated with set and get operators (`set_intensity`, `set_about`, `set_decay`, `set_agent_cause`, `get_intensity`, `get_about`, `get_decay`, `get_agent_cause`).

Below is an exemple of code in GAML concerning emotions:

```

reflex testEmotion{
    predicate a <- new_predicate("test");
    do add_emotion(new_emotion("hope",a));
    do add_emotion(new_emotion("joy",intesity1,a, decay1));
    float c <- get_intensity(get_emotion(new_emotion("joy",a)));
}

```

Social relations

As people create social relations when living with other people and change their behavior based on these relationships, BEN architecture makes it possible to describe social relations in order to use them in agents' behavior. Based on the research carried out by (Svennevig, 2000), a social relation is described by using a finite set of variables. Svennevig identifies a minimal set of four variables: liking, dominance, solidarity, and familiarity. A trust variable is added to interact with the enforcement of social norms. Therefore, in BEN, a social relation between agent i and agent j is expressed as $R_{i,j}(L,D,S,F,T)$ with the following elements:

- **R**: the identifier of the social relation.
- **L**: a real value between -1 and 1 representing the degree of liking with the agent concerned by the link. A value of -1 indicates that agent j is hated, a value of 1 indicates that agent j is liked.
- **D**: a real value between -1 and 1 representing the degree of power exerted on the agent concerned by the link. A value of -1 indicates that agent j is dominating, a value of 1 indicates that agent j is dominated.
- **S**: a real value between 0 and 1 representing the degree of solidarity with the agent concerned by the link. A value of 0 indicates that there is no solidarity with agent j , a value of 1 indicates a complete solidarity with agent j .
- **F**: a real value between 0 and 1 representing the degree of familiarity with the agent concerned by the link. A value of 0 indicates that there is no familiarity with agent j , a value of 1 indicates a complete familiarity with agent j .
- **T**: a real value between -1 and 1 representing the degree of trust with the agent j . A value of -1 indicates doubts about agent j while a value of 1 indicates complete trust with agent j . The trust value does not evolve automatically in accordance with emotions.

With this definition, a social relation is not necessarily symmetric, which means $R_{i,j}(L,D,S,F,T)$ is not equal by definition to $R_{j,i}(L,D,S,F,T)$. **L[R_{i,j}]** stands for the liking value of the social relation between agent i and agent j , **D[R_{i,j}]** stands for its dominance value, **S[R_{i,j}]** for its solidarity value, **F[R_{i,j}]** represents its familiarity value and **T[R_{i,j}]** its trust value.

In GAML, social relations are manipulated with `add_social_link`, `remove_social_link` and `get_social_link` actions. Each feature of a social link is accessible with set and gt operators (`set_agent`, `get_agent`,

set_liking, get_liking, set_dominance, etc.)

Below is an exemple of code to manipulates social relations in GAML:

```
reflex testSocialRelations{
    do add_social_link(new_social_link(agentAlice));
    do add_social_link(new_social_link(agentBob, 0.5, -0.3, 0.2, 0.1));
    float val <- get_liking(get_social_link(new_social_link(agentBob)));
    social_link sl <- set_dominance(get_social_link(new_social_link(agentBob)), 0.3);
}
```

Personality and additional variables

In order to define personality traits, BEN relies on the OCEAN model (McCrae, 1992), also known as the big five factors model. In the BEN architecture, this model is represented through a vector of five values between 0 and 1, with 0.5 as the neutral value. The five personality traits are:

- **O**: represents the openness of someone. A value of 0 stands for someone narrow-minded, a value of 1 stands for someone open-minded.
- **C**: represents the consciousness of someone. A value of 0 stands for someone impulsive, a value of 1 stands for someone who acts with preparations.
- **E**: represents the extroversion of someone. A value of 0 stands for someone shy, a value of 1 stands for someone extrovert.
- **A**: represents the agreeableness of someone. A value of 0 stands for someone hostile, a value of 1 stands for someone friendly.
- **N**: represents the degree of control someone has on his/her emotions, called neurotism. A value of 0 stands for someones neurotic, a value of 1 stands for someone calm.

In GAML, these variables are build-in attributes of agents using the simplebdi *control architecture*. They are *called_openness*, *conscientiousness*, *extroversion*, *agreeableness* and *neurotism*. To use this personality to automatically parametrize the other modules, a modeler needs to indicate it as shown in the GAML example below:

```
species miner control:simple_bdi {
    ...
    bool use_personality <- true;
    float openness <- 0.1;
    float conscientiousness <- 0.2;
    float extroversion <- 0.3;
    float agreeableness <- 0.4;
    float neurotism <- 0.5;
```

With BEN, the agent has variables related to some of the social features. The idea behind the BEN architecture is to connect these variables to the personality module and in particular to the five dimensions of the OCEAN model in order to reduce the number of parameters which need to be entered by the user. These additional variables are:

- The probability to keep the current plan.
- The probability to keep the current intention.
- A charisma value linked to the emotional contagion process.
- An emotional receptivity value linked to the emotional contagion.
- An obedience value used by the normative engine.

With the cognition, the agent has two parameters representing the probability to randomly remove the current plan or the current intention in order to check whether there could be a better plan or a better intention in the current context. These two values are connected to the consciousness components of the OCEAN model as it describes the tendency of the agent to prepare its actions (with a high value) or act impulsively (with a low value).

- Probability Keeping Plans = $C^{1/2}$
- Probability Keeping Intentions = $C^{1/2}$

For the emotional contagion, the process (presented later) requires charisma (Ch) and emotional receptivity (R) to be defined for each agent. In BEN, charisma is related to the capacity of expression, which is related to the extroversion of the OCEAN model, while the emotional receptivity is related to the capacity to control the emotions, which is expressed with the neuroticism value of OCEAN.

- $Ch = E$
- $R = 1 - N$

With the concept of norms, the agent has a value of obedience between 0 and 1, which indicates its tendency to follow laws, obligations, and norms. According to research in psychology, which tried to explain the behavior of people participating in a recreation of the Milgram's experiment (Begue, 2015), obedience is linked with the notions of consciousness and agreeableness which gives the following equation:

- $obedience = ((C+A)/2)^{1/2}$

With the same idea, all the parameters required by each process are linked to the OCEAN model.

If a modeler wants to put a different value to one of these variables, he/she just need to indicate a new value manually. For the probability to keep the current plan and the probability to keep the current

intention, he/she also has to indicates it with a particular boolean value, as shown in the GAML example below:

```
species miner control: simple_bdi {  
    ...  
    bool use_personality <- true;  
    bool use_persistence <- true;  
    float plan_persistence <- 0.3;  
    float intention_persistence <- 0.4;  
    float obedience <- 0.2;  
    float charisma <- 0.3;  
    float receptivity <- 0.6;  
    ...  
}
```

Perception

The first step of BEN is the perception of the environment. This module is used to connect the environment to the knowledge of the agent, transforming information from the world into cognitive mental states, emotions or social links but also used to apply sanctions during the enforcement of norms from other agents.

Below is an example of code to define a perception in GAML:

```
perceive target: fireArea in: 10{  
    ...  
}
```

The first process in this perception consists of **adding beliefs** about the world. During this phase, information from the environment is transformed into predicates which are included in beliefs or uncertainties and then added to the agent's knowledge bases. This process enables the agent to update its knowledge about the world. From the modeler's point of view, it is only necessary to specify which information is transformed into which predicate. The addition of a belief $Belief_A(X)$ triggers multiple processes :

- it removes $Belief_A(not X)$.
- it removes $Intention_A(X)$.
- it removes $Desire_A(X)$ if $Intention_A(X)$ has just been removed.
- it removes $Uncertainty_A(X)$ or $Uncertainty_A(not X)$.
- it removes $Obligation_A(X)$. \end{itemize}

In GAML, the *focus* statement eases the use of this process. Below is an example that adds a belief and an uncertainty with the focus statement during a perception:

```
perceive target: fireArea in: 10{
    focus id:"fireLocation" var:location strength:10.0;
    //is equivalent to ask myself {do
    add_belief(new_predicate("fireLocation", ["location_value":::myself.location],10.0);}
    focus id:"hazardLocation" var:location strength:1.0 is_uncertain:true;
    //is equivalent to ask myself {do
    add_uncertainty(new_predicate("hazardLocation", ["location_value":::myself.location],1.0);}
}
```

The **emotional contagion** enables the agent to update its emotions according to the emotions of other agents perceived. The modeler has to indicate the emotion triggering the contagion, the emotion created in the perceiving agent and the threshold of this contagion; the charisma (Ch) and receptivity (R) values are automatically computed as explained previously. The contagion from agent i to agent j occurs only if $Ch_i \times R_j$ is superior or equal to the threshold, which value is 0.25 by default. Then, the presence of the trigger emotion in the perceived agent is checked in order to create the emotion indicated.

The intensity and decay value of the emotion acquired by contagion are automatically computed.

- If $Em_j(P)$ already exists:
 - $I[Em_j(P)] = I[Em_j(P)] + I[Em_i(P)] \times Ch_i \times R_j$
 - if $pEm_i(P) > I[Em_j(P)]$:
 - $De[Em_j(P)] = De[Em_i(P)]$
 - if $I[Em_j(P)] > I[Em_i(P)]$:
 - $De[Em_j(P)] = De[Em_j(P)]$
- If $Em_j(P)$ does not already exist:
 - $I[Em_j(P)] = I[Em_i(P)] \times Ch_i \times R_j$
 - $De[Em_j(P)] = De[Em_i(P)].$

In GAML, *emotional_contagion* statement helps to define an emotional contagion during a perception, as shown below:

```
perceive target: otherHumanAgents in: 10{
    emotional_contagion emotion_detected:fearFire threshold:contagionThreshold;
    //creates the detected emotion, if detected, in the agent doing the perception.
    emotional_contagion emotion_detected:joyDance emotion_created:joyPartying;
    //creates the emotion "joyPartying", if emotion "joyDance" is detected in the
    perceived agent.
}
```

During the perception, the agent has the possibility of **creating social relations** with other perceived agents. The modeler indicates the initial value for each component of the social link, as explained previously. By default, a neutral relation is created, with each value of the link at 0.0. Social relations can also be defined before the start of the simulation, to indicate that an agent has links with other agents at the start of the simulation, like links with friends or family members.

In GAML, the *socialize* statement help creating dynamicaly new social relations, as shown below:

```
perceive target:otherHumanAgents in: 10{
    socialize;
    //creates a neutral relation
    socialize dominance: -0.8 familiarity:0.2 when: isBoss;
    //example of a social link with precise values for some of its dimensions in a
    certain context
}
```

Finally, the agent may **apply sanctions** through the norm enforcement of other agents perceived. The modeler needs to indicate which modality is enforced and the sanction and reward used in the process. Then, the agent checks if the norm, the obligation, or the law, is violated, applied or not activated by the perceived agent. Notions of norms laws and obligations and how they work are explained later in this document.

A norm is considered violated when its context is verified, and yet the agent chose another norm or another plan to execute because it decided to disobey. A law is considered violated when its context is verified, but the agent disobeyed it, not creating the corresponding obligation. Finally, an obligation is considered violated if the agent did not execute the corresponding norm because it chose to disobey.

Below is an example of how to define an enforcement in GAML:

```
species miner skills: [moving] control:simple_bdi {
    ...
    perceive target: miner in: viewdist {
        myself.agent_perceived<-self;
        enforcement norm:"share_information" sanction:"sanctionToNorm" reward:"rewardToNorm";
    }

    sanction sanctionToNorm{
        do change_liking(agent_perceived, -0.1);
    }

    sanction rewardToNorm{
        do change_liking(agent_perceived, 0.1);
    }
}
```

Managing knowledge bases

The second step of the architecture, corresponding to the module number 2, consists of managing the agent's knowledge. This means updating the knowledge bases according to the latest perceptions, adding new desires, new obligations, new emotions or updating social relations, for example.

Modelers have to use **inference rules** for this purpose. These rules are triggered by a new belief, a new uncertainty or a new emotion, in a certain context, and may add or remove any cognitive mental state or emotion indicated by the user. Using multiple inference rules helps the agent to adapt its mind to the situation perceived without removing all its older cognitive mental states or emotions, thus enabling the creation of a cognitive behavior. These inference rules enable to link manually the various dimensions of an agent, for example creating desires depending on emotions, social relations and personality.

In GAML, the *rule* statement enables to define inference rules:

```
species miner skills: [moving] control: simple_bdi {  
    ...  
    perceive target: miner in: viewdist {  
        ...  
    }  
    ...  
    rule belief: new_predicate("testA") new_desire: new_predicate("testB");  
}
```

Using the same idea, modelers can define **laws**. These laws enable the creation of obligations in a given context based on the newest beliefs created by the agent through its perception or its inference rules. The modeler also needs to indicate an obedience threshold and if the agent's obedience value is below that threshold, the law is violated. If the law is activated, the obligation is added to the agent's cognitive mental state bases. The definition of laws makes it possible to create a behavior based on obligations imposed upon the agent.

Below is an example of the definition of a *law* statement in GAML:

```
law belief: new_predicate("testA") new_obligation:new_predicate("testB")  
threshold:thresholdLaw;
```

Emotional engine

BEN enables the agent to get emotions about its cognitive mental states. This **addition of emotions** is

based on the OCC model (Ortony, 1990) and its logical formalism (Adam, 2007), which has been proposed to integrate the OCC model in a BDI formalism.

According to the OCC theory, emotions can be split into three groups: emotions linked to events, emotions linked to people and actions performed by people, and emotions linked to objects. In BEN, as the focus is on relations between social agents, only the first two groups of emotions (emotions linked to events and people) are considered.

The twenty emotions defined in this paper can be divided into seven groups depending on their relations with mental states: emotions about beliefs, emotions about uncertainties, combined emotions about uncertainties, emotions about other agents with a positive liking value, emotions about other agents with a negative liking value, emotions about ideals and combined emotions about ideals. All the initial intensities and decay value are computed using the OCEAN model and the value attached to the concerned mental states.

The emotions about beliefs are joy and sadness and are expressed this way:

- **Joy_i(P_j,j)** = Belief_i(P_j) \& Desire_i(P)
- **Sadness_i(P_j,j)** = Belief_i(P_j) \& Desire_i(not P)

Their initial intensity is computed according to the following equation with N the neuroticism component from the OCEAN model:

- $I[Em_i(P)] = V[Belief_i(P)] \times V[Desire_i(P)] \times (1+(0,5-N))$

The emotions about uncertainties are fear and hope and are defined this way:

- **Hope_i(P_j,j)** = Uncertainty_i(P_j) \& Desire_i(P)
- **Fear_i(P_j,j)** = Uncertainty_i(P_j) \& Desire_i(not P)

Their initial intensity is computed according to the following equation:

- $I[Em_i(P)] = V[Uncertainty_i(P)] \times V[Desire_i(P)] \times (1+(0,5-N))$

Combined emotions about uncertainties are emotions built upon fear and hope. They appear when an uncertainty is replaced by a belief, transforming fear and hope into satisfaction, disappointment, relief or fear confirmed and they are defined this way:

- **Satisfaction_i(P_j,j)** = Hope_i(P_j,j) \& Belief_i(P_j)
- **Disappointment_i(P_j,j)** = Hope_i(P_j,j) \& Belief_i(not P_j)
- **Relief_i(P_j,j)** = Fear_i(P_j,j) \& Belief_i(not P_j)

- **Fear confirmed_i(P_j,j)** = Fear_i(P_j,j) \& Belief_i(P_j)

Their initial intensity is computed according to the following equation with Em'_i(P) the emotion of fear/hope.

- I[Em_i(P)] = V[Belief_i(P)] x I[Em'_i(P)]

On top of that, according to the logical formalism (Adam, 2007), four inference rules are triggered by these emotions:

- The creation of **fear confirmed** or the creation of **relief** will replace the emotion of **fear**.
- The creation of **satisfaction** or the creation of **disappointment** will replace a **hope** emotion.
- The creation of **satisfaction** or **relief** leads to the creation of **joy**.
- The creation of **disappointment** or **fear confirmed** leads to the creation of **sadness**.

The emotions about other agents with a positive liking value are emotions related to emotions of other agents which are in the social relation base with a positive liking value on that link. They are the emotions called "happy for" and "sorry for" which are defined this way :

- **Happy for_i(P,j)** = L[R_{i,j}] > 0 \& Joy_j(P)
- **Sorry for_i(P,j)** = L[R_{i,j}] > 0 \& Sadness_j(P)

Their initial intensity is computed according to the following equation with A the agreeableness value from the OCEAN model.

- I[Em_i(P)] = I[Em_j(P)] x L[R_{i,j}] x (1-(0,5-A))

Emotions about other agents with a negative liking value are close to the previous definitions, however, they are related to the emotions of other agents which are in the social relation base with a negative liking value. These emotions are resentment and gloating and have the following definition:

- **Resentment_i(P,j)** = L[R_{i,j}] < 0 \& Joy_j(P)
- **Gloating_i(P,j)** = L[R_{i,j}] < 0 \& Sadness_j(P)

Their initial intensity is computed according to the following equation. This equation can be seen as the inverse of Equation \eqref{eqIntensEmo4}, and means that the intensity of resentment or gloating is greater if the agent has a low level of agreeableness contrary to the intensity of "happy for" and "sorry for".

- I[Em_i(P)] = I[Em_j(P)] x |L[R_{i,j}]| x (1+(0,5-A))

Emotions about ideals are related to the agent's ideal base which contains, at the start of the simulation,

all the actions about which the agent has a praiseworthiness value to give. These ideals can be praiseworthy (their praiseworthiness value is positive) or blameworthy (their praiseworthiness value is negative). The emotions coming from these ideals are pride, shame, admiration and reproach and have the following definition:

- **Pride_i(P_i,i)** = Belief_i(P_i) \& Ideal_i(P_i) \& V[Ideal_i(P_i)] > 0
- **Shame_i(P_i,i)** = Belief_i(P_i) \& Ideal_i(P_i) \& V[Ideal_i(P_i)] < 0
- **Admiration_i(P_j,j)** = Belief_i(P_j) \& Ideal_i(P_j) \& V[Ideal_i(P_j)] > 0
- **Reproach_i(P_j,j)** = Belief_i(P_j) \& Ideal_i(P_j) \& V[Ideal_i(P_j)] < 0

Their initial intensity is computed according to the following equation with O the openness value from the OCEAN model:

- $I[Em_i(P)] = V[Belief_i(P)] \times |V[Ideal_i(P)]| \times (1+(0,5-O))$

Finally, combined emotions about ideals are emotions built upon pride, shame, admiration and reproach. They appear when joy or sadness appear with an emotion about ideals. They are gratification, remorse, gratitude and anger which are defined as follows:

- **Gratification_i(P_i,i)** = Pride_i(P_i,i) \& Joy_i(P_i)
- **Remorse_i(P_i,i)** = Shame_i(P_i,i) \& Sadness_i(P_i)
- **Gratitude_i(P_j,j)** = Admiration_i(P_j,j) \& Joy_i(P_j)
- **Anger_i(P_j,j)** = Reproach_i(P_j,j) \& Sadness_i(P_j)

Their initial intensity is computed according to the following equation with Em'_i(P) the emotion about ideals and Em"_i(P) the emotion about beliefs.

- $I[Em_i(P)] = I[Em'_i(P)] \times I[Em''_i(P)]$

In order to keep the initial intensity of each emotion between 0 and 1, each equation is truncated between 0 and 1 if necessary.

The initial decay value for each of these twenty emotions is computed according to the same equation with Deltat a time step which enables to define that an emotion does not last more than a given time:

- $De[Em_i(P)] = N \times I[Em_i(P)] \times Deltat$

To use this automatic computation of emotion, a modeler need to activate it as shown in the GAML example below :

```

species miner control:simple_bdi {
    ...
    bool use_emotions_architecture <- true;
    ...
}

```

Social Engine

When an agent already known is perceived (i.e. there is already a social link with it), the social relationship with this agent is updated automatically by BEN. This update is based on the work of (Ochs, 2009) and takes the agent's cognitive mental states and emotions into account. In this section, the **automatic update of each variable of a social link** $R_{i,j}(L,D,S,F,T)$ by the architecture is described in details; the trust variable of the link is however not updated automatically.

- **Liking:** according to (Ortony, 1991), the degree of liking between two agents depends on the valence (positive or negative) of the emotions induced by the corresponding agent. In the emotional model of the architecture, *joy* and *hope* are considered as positive emotions (*satisfaction* and *relief* automatically raise *joy* with the emotional engine) while *sadness* and *fear* are considered as negative emotions (*fear confirmed* and *disappointment* automatically raise *sadness* with the emotional engine). So, if an agent i has a positive (resp. negative) emotion caused by an agent j , this will increase (resp. decrease) the value of appreciation in the social link from i concerning j .

Moreover, research has shown that the degree of liking is influenced by the solidarity value \cite{smith2014social}. This may be explained by the fact that people tend to appreciate people similar to them.

The computation formula is described with the following equation with $mPos$ the mean value of all positive emotions caused by agent j , $mNeg$ the mean value of all negative emotions caused by agent j and a_L a coefficient depending of the agent's personality, indicating the importance of emotions in the process, and which is described below.

- $L[R_{i,j}] = L[R_{i,j}] + |L[R_{i,j}]| (1 - |L[R_{i,j}]|) S[R_{i,j}] + a_L (1 - |L[R_{i,j}]|)(mPos - mNeg)$
- $a_L = 1 - N$
- **Dominance :** (Keltner, 2001) and (Shiota, 2004) explain that an emotion of fear or sadness caused by another agent represent an inferior status. But (Knutson, 1996) explains that perceiving fear and sadness in others increases the sensation of power over those persons.

The computation formula is described by the following equation with mSE the mean value of all negative emotions caused by agent i to agent j , mOE the mean value of all negative emotions caused by agent j to

agent i and a_D a coefficient depending on the agent's personality, indicating the importance of emotions in the process.

- $D[R_{i,j}] = D[R_{i,j}] + a_D (1 - |D[R_{i,j}]|)(mSE - mOE)$
- $a_D = 1 - N$
- **Solidarity:** The solidarity represents the degree of similarity of desires, beliefs, and uncertainties between two agents. In BEN, the evolution of the solidarity value depends on the ratio of similarity between the desires, beliefs, and uncertainties of agent i and those of agent j . To compute the similarities and oppositions between agent i and agent j , agent i needs to have beliefs about agent j 's cognitive mental states. Then it compares these cognitive mental states with its own to detect similar or opposite knowledge.

On top of that, negative emotions tend to decrease the value of solidarity between two people. The computation formula is described by the following equation with sim the number of cognitive mental states similar between agent i and agent j , opp the number of opposite cognitive mental states between agent i and agent j , NbKnow the number of cognitive mental states in common between agent i and agent j , mNeg the mean value of all negative emotions caused by agent j , as₁ a coefficient depending of the agent's personality, indicating the importance of similarities and oppositions in the process, and as₂ a coefficient depending of the agent's personality, indicating the importance of emotions in the process.

- $S[R_{i,j}] = S[R_{i,j}] + S[R_{i,j}] \times (1 - S[R_{i,j}]) \times (as_1 (sim - opp) / (NbKnow) - as_2 mNeg))$
- $as_1 = 1 - O$
- $as_2 = 1 - N$
- **Familiarity:** In psychology, emotions and cognition do not seem to impact the familiarity. However, (Collins, 1994) explains that people tend to be more familiar with people whom they appreciate. This notion is modeled by basing the evolution of the familiarity value on the liking value between two agents. The computation formula is defined by the following equation.
- $F[R_{i,j}] = F[R_{i,j}] \times (1 + L[R_{i,j}])$

The trust value is not evolving automatically in BEN, as there is no clear and automatic link with cognition or emotions. However, this value can evolve manually, especially with sanctions and rewards to social norms where the modeler can indicate a modification of the trust value during the enforcement process.

To use this automatic update of social relations, a modeler need to activate it as shown in the GAML example below:

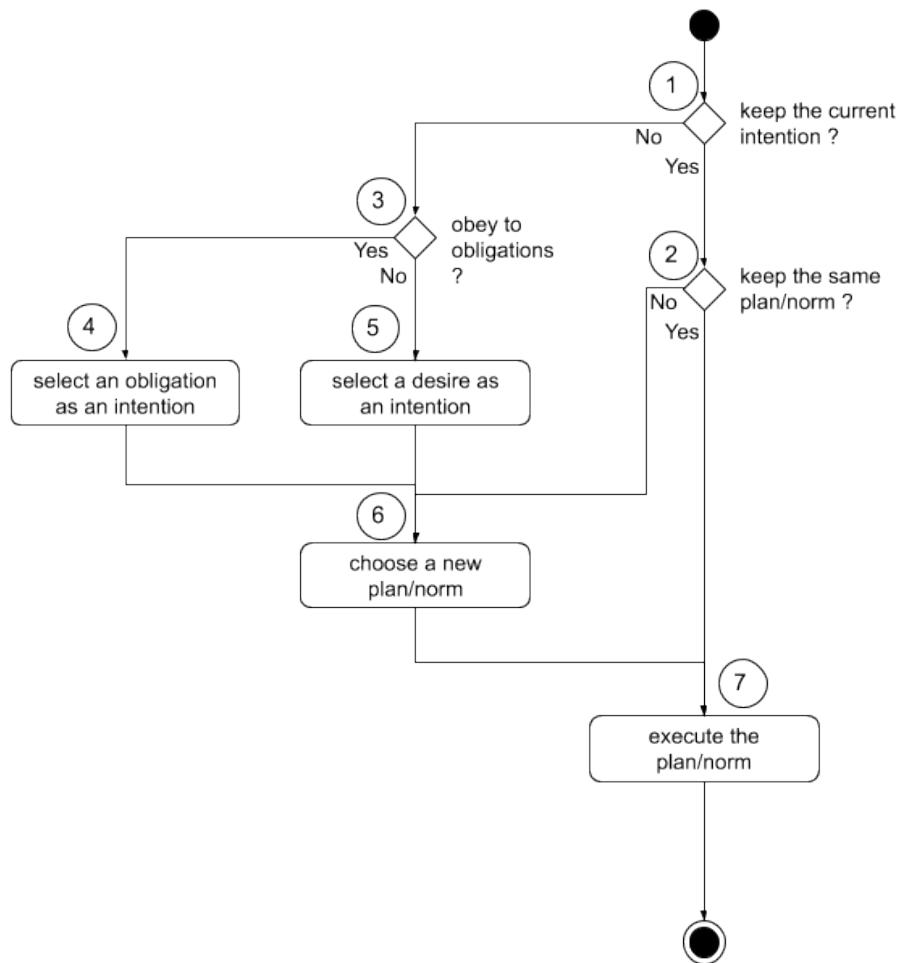
```

species miner control: simple_bdi {
    ...
    bool use_social_architecture <- true;
    ...
}

```

Making Decision

The third part of the architecture is the only one mandatory as it is where the agent makes a decision. A cognitive engine can be coupled with a normative engine to chose an intention and a plan to execute. The complete engine is summed up in the figure below:



The decision-making process can be divided into seven steps:

- **Step 1:** the engine checks the current intention. If it is still valid, the intention is kept so the agent may continue to carry out its current plan.
- **Step 2:** the engine checks if the current plan/norm is still usable or not, depending on its context.
- **Step 3:** the engine checks if the agent obeys an obligation taken from the obligations corresponding to a norm with a valid context in the current situation and with a threshold level lower than the agent's obedience value as computed in Section 4.1.
- **Step 4:** the obligation with the highest priority is taken as the current intention.
- **Step 5:** the desire with the highest priority is taken as the current intention.
- **Step 6:** the plan or norm with the highest priority is selected as the current plan/norm, among the plans or norms corresponding to the current intention with a valid context.
- **Step 7:** the behavior associated with the current plan/norm is executed.

Steps 4, 5 and 6 do not have to be deterministic; they may be probabilistic. In this case, the priority value associated with obligations, desires, plans, and norms serves as a probability.

In GAML, a modeler may indicate the use of a probabilistic or deterministic cognitive engine with the variable `probabilistic_choice`, as shown in the example code below:

```
species miner control: simple_bdi {
    ...
    bool probabilistic_choice <- true;
    ...
}
```

Defining plans

The modeler needs to define action plans which are used by the cognitive engine, as explained earlier. These plans are a set of behaviors executed in a certain context in response to an intention. In BEN, a plan owned by agent i is represented by $\text{Pl}_i(\text{Int}, \text{Cont}, \text{Pr}, \text{B})$ with:

- **Pl:** the name of the plan.
- **Int:** the intention triggering this plan.
- **Cont:** the context in which this plan may be applied.
- **Pr:** a priority value used to choose between multiple plans relevant at the same time. If two plans are relevant to the same priority, one is chosen at random.
- **B:** the behavior, as a sequence of instructions, to execute if the plan is chosen by the agent.

The context of a plan is a particular state of the world in which this plan should be considered by the agent making a decision. This feature enables to define multiple plans answering the same intention but

activated in various contexts.

Below is an example for the definition of two plans answering the same intention in different contexts in GAML:

```
species miner control: simple_bdi skills: [moving]{
    ...
    plan evacuationFast intention: in_shelter emotion: fearConfirmed priority:2 {
        color <- #yellow;
        speed <- 60 #km/#h;
        if (target = nil or noTarget) {
            target <- (shelter with_min_of (each.location distance_to location)).location;
            noTarget <- false;
        } else {
            do goto target: target on: road_network move_weights: current_weights
        recompute_path: false;
            if (target = location) {
                do die;
            }
        }
    }

    plan evacuation intention: in_shelter finished_when: has_emotion(fearConfirmed){
        color <-#darkred;
        if (target = nil or noTarget) {
            target <- (shelter with_min_of (each.location distance_to location)).location;
            noTarget <- false;
        } else {
            do goto target: target on: road_network move_weights: current_weights
        recompute_path: false;
            if (target = location) {
                do die;
            }
        }
    }
    ...
}
```

Defining norms

A normative engine may be used within the cognitive engine, as it has been explained above. This normative engine means choosing an obligation as the current intention and selecting a set of actions to answer this intention. Also, the concept of social norms is modeled as a set of action answering an intention, which an agent could disobey. tention and selecting a set of actions to answer this intention. Also, the concept of social norms is modeled as a set of action answering an intention, which an agent could disobey.

In BEN, this concept of behavior which may be disobeyed is formally represented by a norm possessed by agent i **No_i(Int,Cont,Ob,Pr,B,Vi)** with:

- **No**: the name of the norm.
- **Int**: the intention which triggers this norm.
- **Cont**: the context in which this norm can be applied.
- **Ob**: an obedience value that serves as a threshold to determine whether or not the norm is applied depending on the agent's obedience value (if the agent's value is above the threshold, the norm may be executed).
- **Pr**: a priority value used to choose between multiple norms applicable at the same time.
- **B**: the behavior, as a sequence of instructions, to execute if the norm is followed by the agent.
- **Vi**: a violation time indicating how long the norm is considered violated once it has been violated.

In GAML, a norm is defined as follows:

```
species miner control: simple_bdi {
    ...
    //this first norm answer an intention coming from an obligation
    norm doingJob obligation:has_gold finished_when: has_belief(has_gold)
threshold:thresholdObligation{
        if (target = nil) {
            do add_subintention(has_gold,choose_goldmine, true);
            do current_intention_on_hold();
        } else {
            do goto target: target ;
            if (target = location) {
                goldmine current_mine<- goldmine first_with (target = each.location);
                if current_mine.quantity > 0 {
                    gold_transported <- gold_transported+1;
                    do add_belief(has_gold);
                    ask current_mine {quantity <- quantity - 1;};
                } else {
                    do add_belief(new_predicate(empty_mine_location,
["location_value"]::target));
                    do remove_belief(new_predicate(mine_at_location,
["location_value"]::target));
                }
                target <- nil;
            }
        }
    }

    //this norm may be seen as a "social norm" as it answers an intention not coming from an
    //obligation but may be disobeyed
    norm share_information intention:share_information threshold:thresholdNorm
```

Dynamic knowledge

The final part of the architecture is used to create a temporal dynamic to the agent's behavior, useful in a simulation context. To do so, this module automatically degrades mental states and emotions and updates the status of each norm.

The **degradation of mental states** consists of reducing their lifetime. When the lifetime is null, the mental state is removed from its base. The **degradation of emotions** consists of reducing the intensity of each emotion stored by its decay value. When the intensity of an emotion is null, the emotion is removed from the emotional base.

In GAML, if a mental state has a lifetime value or if an emotion has an intensity and a decay value, this degradation process is done automatically.

Finally, **the status of each norm is updated** to indicate if the norm was activated or not (if the context was right or wrong) and if it was violated or not (the norm was activated but the agent disobeyed it). Also, a norm can be violated for a certain time which is updated and if it becomes null, the norm is not violated anymore.

These last steps enable the agent's behavior's components to automatically evolve through time, leading the agents to forget a piece of knowledge after a certain amount of time, creating dynamics in their behavior.

Conclusion

The BEN architecture is already implemented in GAMA and may be accessed by adding the simple_bdi control architecture to the definition of a species.

A tutorial may be found with the [BDI Tutorial](#).

Version: 1.9.1

Advanced Driving Skill

This page aims at presenting how to use the advanced driving skill in models.

The use of the advanced driving skill requires to use 3 skills:

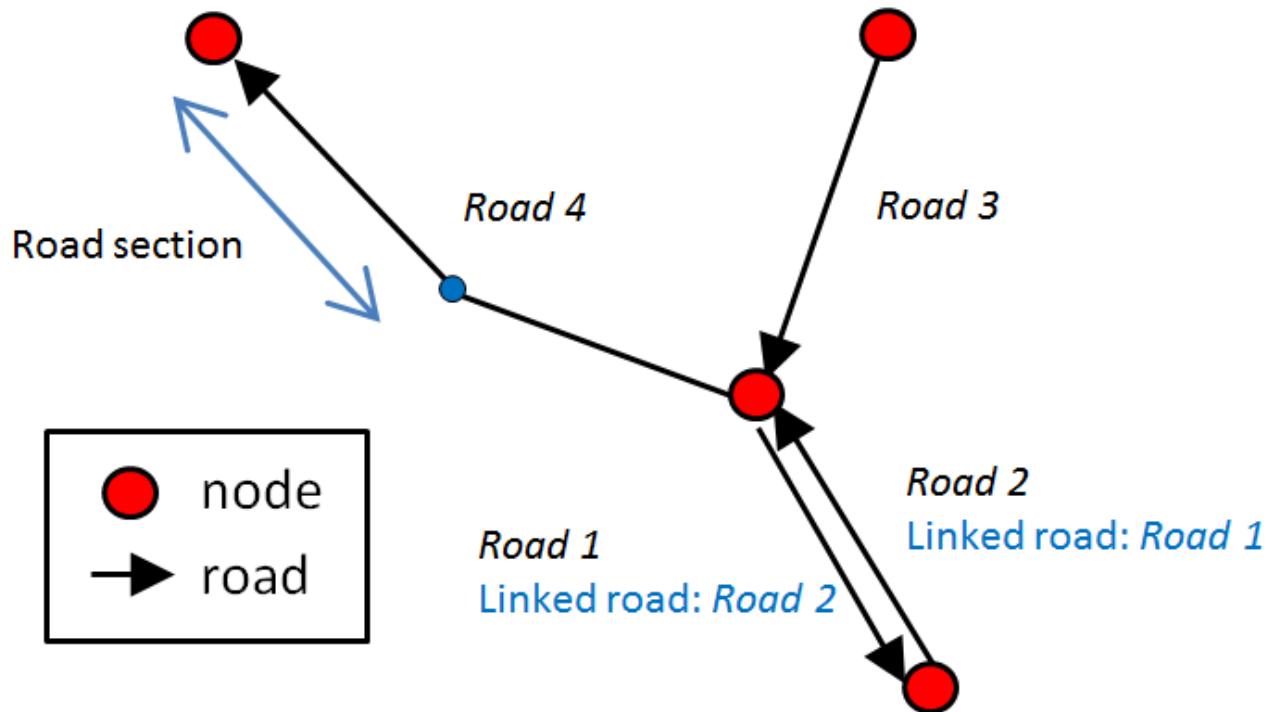
- **Advanced driving skill:** dedicated to the definition of the driver species. It provides the driver agents with variables and actions allowing to move an agent on a graph network and to tune its behavior.
- **Road skill:** dedicated to the definition of roads. It provides the road agents with variables and actions allowing to registers agents on the road.
- **Road node skill:** dedicated to the definition of nodes. It provides the node agents with variables allowing to take into account the intersection of roads and the traffic signals.

Table of contents

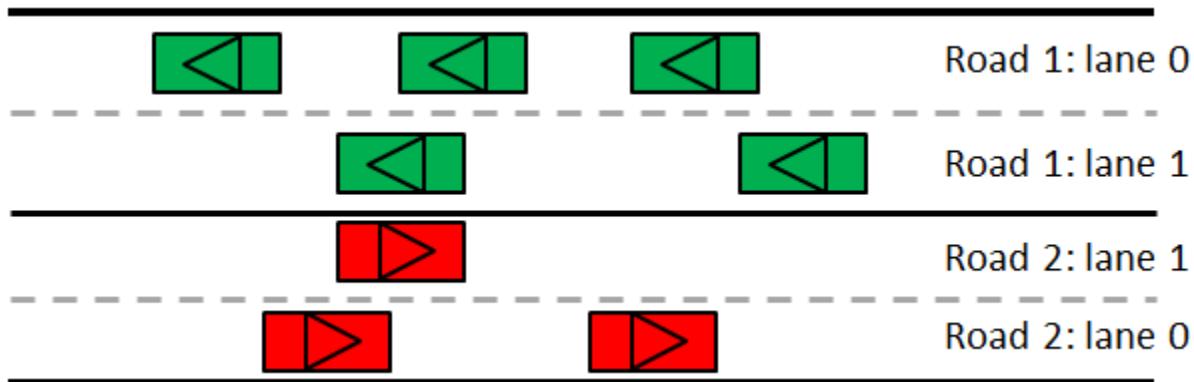
- Advanced Driving Skill
 - Structure of the network: road and road node skills
 - Advanced driving skill
 - Application example

Structure of the network: road and road_node skills

The advanced driving skill is versatile enough to be usable with most of classic road GIS data, in particular, OSM data. We use a classic format for the roads and nodes. Each road is a polyline composed of road sections (segments). Each road has a target node and a source node. Each node knows all its input and output roads. A road is considered as directed. For bidirectional roads, 2 roads have to be defined corresponding to both directions. Each road will be the `linked_road` of the other. Note that for some GIS data, only one road is defined for bidirectional roads, and the nodes are not explicitly defined. In this case, it is very easy, using the GAML language, to create the reverse roads and the corresponding nodes (it only requires a few lines of GAML).



A road can be composed of several lanes and the vehicles will be able to change at any time its lane. What a lane represents will depend a lot on the context of application. Typically, if in developed countries, the lanes are most of times well defined, in many other countries this notion is much more abstract. For example in Vietnam where the main means of locomotion is the motorcycle, a lane can designate a "place" for a motorcycle and thus be much narrower than classical lanes. Another property of the road that will be taken into account is the maximal authorized speed on it. Note that even if the user of the plug-in has no information about these values for some of the roads (the OSM data are often incomplete), it is very easy using the GAML language to fill the missing value by a default value. It is also possible to change these values dynamically during the simulation (for example, to take into account that after an accident, a lane of a road is closed or that the speed of a road is decreased by the authorities).



The **road skill** (`skill_road`) provides the road agents with several variables that will define the road properties:

- `num_lanes` : integer, number of lanes.
- `maxspeed` : float; maximal authorized speed on the road.
- `linked_road` : road agent; reverse road (if there is one).
- `source_node` : node agent; source node of the road.
- `target_node` : node agent; target node of the road.

It provides as well the road agents with read-only variables:

- `agents_on`: list of list (of driver agents); for each lane, the list of driver agents on the road.
- `all_agents`: list (of driver agents): the list of agents on the road.

The **road node skill** (`skill_road_node`) provides the road node agents with several variables that will define the road node properties:

- `roads_in`: list of road agents; the list of road agents that have this node for target node.
- `roads_out`: list of road agents; the list of road agents that have this node for source node.
- `stop`: list of list of road agents; list of stop signals, and for each stop signal, the list of concerned roads.
- `priority_roads`: list of road agents: the list of priority roads.

It provides as well the road agents with one read-only variable:

- `block`: map: key: driver agent, value: list of road agents; the list of driver agents blocking the node, and for each agent, the list of concerned roads.

Advanced driving skill

A vehicle is first characterized by its location, a 3D-point (coordinate) that represents the centroid of the vehicle. The actual geometry of the vehicle is not taken into account. However, the size of a vehicle is determined by two attributes:

`vehicle_length` and `num_lanes_occupied`. Indeed, if we go back to our Vietnamese example where the lanes are defined according to the size of the motorcycles, we can consider that a motorcycle will occupy one lane, but that a car, which is much wider, will occupy two.

Each vehicle agent has also a planned trajectory that consists of a succession of edges. When the vehicle agent enters a new edge, it first chooses its lane according to the traffic density, with a bias for the rightmost lane. The movement on an edge is inspired by the Intelligent Driver Model. The drivers have the possibility to change their lane at any time (and not only when entering a new edge). The lane-changing model is inspired from the MOBIL model.

The **advanced driving skill** (`advanced_driving`) provides the driver agents with several variables that will define the car properties and the personality of the driver:

- `final_target`: point; final location that the agent wants to reach (its goal).
- `vehicle_length`: float; length of the vehicle.
- `num_lanes_occupied`: float; the number of lanes occupied by the vehicle.
- `max_acceleration`: float; maximal acceleration of the vehicle.
- `max_speed`: float; maximal speed of the vehicle.
- `right_side_driving`: boolean; do drivers drive on the right side of the road?
- `speed_coeff`: float; coefficient that defines if the driver will try to drive above or below the speed limits.
- `safety_distance_coeff`: float; coefficient for the security distance. The security distance will depend on the driver speed and on this coefficient.
- `proba_lane_change_up`: float; probability to change lane to an upper lane if necessary (and if possible).
- `proba_lane_change_down`: float; probability to change lane to a lower lane if necessary (and if possible).
- `proba_use_linked_road`: float; probability to take the reverse road if necessary (if there is a reverse road).
- `proba_respect_priorities`: float; probability to respect left/right (according to the driving side) priority at intersections.
- `proba_respect_stops`: list of float; probabilities to respect each type of stop signals (traffic light, stop sign...).

- `proba_block_node`: float; probability to accept to block the intersecting roads to enter a new road.
- `lane_change_coldown`: float; the duration that a vehicle must wait before changing lanes again
- `max_safe_deceleration`: float; the maximum deceleration that the vehicle is willing to induce on its back vehicle when changing lanes. Known as the parameter 'b_save' in the MOBIL lane changing model
- `min_safety_distance`: float; the minimum distance of the vehicle's front bumper to the leading vehicle's rear bumper, known as the parameter s0 in the Intelligent Driver Model
- `lane_change_limit`: int; the maximum number of lanes that the vehicle can change during a simulation step
- `acc_gain_threshold`: float; the minimum acceleration gain for the vehicle to switch to another lane, introduced to prevent frantic lane changing. Known as the parameter 'a_th' in the MOBIL lane changing model
- `linked_lane_limit`: int; the maximum number of linked lanes that the vehicle can use; the default value is -1, i.e. the vehicle can use all available linked lanes
- `ignore_oneway`: bool; if set to `true`, the vehicle will be able to violate one-way traffic rule
- `lowest_lane`: int; the lane with the smallest index that the vehicle is in
- `acc_bias`: float; the bias term used for asymmetric lane changing, parameter 'a_bias' in MOBIL
- `allowed_lanes`: list of int; a list containing possible lane index values for the attribute lowest_lane
- `time_headway`: float; the time gap that to the leading vehicle that the driver must maintain. Known as the parameter 'T' in the Intelligent Driver Model
- `delta_idm`: float; the exponent used in the computation of free-road acceleration in the Intelligent Driver Model
- `max_deceleration`: float; the maximum deceleration of the vehicle. Known as

the parameter 'b' in the Intelligent Driver Model

- **politeness_factor**: float; determines the politeness level of the vehicle when changing lanes. Known as the parameter 'p' in the MOBIL lane changing model

It provides as well the driver agents with several read-only variables:

- **speed**: float; speed expected according to the road **max_value**, the car properties, the personality of the driver and its **real_speed**.
- **real_speed**: float; real speed of the car (that takes into account the other drivers and the traffic signals).
- **current_path**: path (list of roads to follow); the path that the agent is currently following.
- **current_road**: agent; the road on which the agent is driving on.
- **lowest_lane**: agent; the index of the lowest lane occupied.
- **current_target**: point; the next target to reach (sub-goal). It corresponds to a node.
- **targets**: list of points; list of locations (sub-goals) to reach the final target.
- **current_index**: integer; the index of the current goal the agent has to reach.
- **using_linked_road**: boolean; is the agent on the linked road?

Of course, the values of these variables can be modified at any time during the simulation. For example, the probability to take a reverse road (**proba_use_linked_road**) can be increased if the driver is stuck for several minutes behind a slow vehicle.

In addition, the advanced driving skill provides driver agents with several actions:

- **compute_path**: arguments: a graph and a target node. This action computes from a graph the shortest path to reach a given node.
- **drive**: no argument. This action moves the driver on its current path according to the traffic condition and the driver properties (vehicle properties and driver

personality). The `drive_random` make the agent drives on a road and chooses randomly a new road at each intersection.

The `drive` action works as follow: while the agent has the time to move (`remaining_time > 0`), it first defines the speed expected. This speed is computed from the `max_speed` of the road, the current `real_speed`, the `max_speed`, the `max_acceleration` and the `speed_coef` of the driver.

Then, the agent moves toward the current target and compute the remaining time. During the movement, the agents can change lanes. If the agent reaches its final target, it stops; if it reaches its current target (that is not the final target), it tests if it can cross the intersection to reach the next road of the current path. If it is possible, it defines its new target (target node of the next road) and continues to move.

The function that defines if the agent crosses or not the intersection to continue to move works as follow: first, it tests if the road is blocked by a driver at the intersection (if the road is blocked, the agent does not cross the intersection). Then, if there is at least one stop signal at the intersection (traffic signal, stop sign...), for each of these signals, the agent tests its probability to respect or not the signal (note that the agent has a specific probability to respect each type of signals). If there is no stopping signal or if the agent does not respect it, the agent checks if there is at least one vehicle coming from a right (or left if the agent drives on the left side) road at a distance lower than its security distance. If there is one, it tests its probability to respect this priority. If there is no vehicle from the right roads or if it chooses to do not respect the right priority, it tests if it is possible to cross the intersection to its target road without blocking the intersection (i.e. if there is enough space in the target road). If it can cross the intersection, it crosses it; otherwise, it tests its probability to block the node: if the agent decides nevertheless to cross the intersection, then the perpendicular roads will be blocked at the intersection level (these roads will be unblocked when the agent is going to move).

Concerning the movement of the driver agents on the current road, the agent moves

from a section of the road (i.e. segment composing the polyline) to another section according to the maximal distance that the agent can moves (that will depend on the remaining time). For each road section, the agent first computes the maximal distance it can travel according to the remaining time and its speed. Then, the agent computes its security distance according to its speed and its `safety_distance_coeff`. While its remaining distance is not null, the agent computes the maximal distance it can travel (and the corresponding lane), then it moves according to this distance (and update its current lane if necessary). If the agent is not blocked by another vehicle and can reach the end of the road section, it updates its current road section and continues to move.

The computation of the maximal distance an agent can move on a road section consists of computing for each possible lane the maximal distance the agent can move. First, if there is a lower lane, the agent tests the probability to change its lane to a lower one. If it decides to test the lower lane, the agent computes the distance to the next vehicle on this lane and memorizes it. If this distance corresponds to the maximal distance it can travel, it chooses this lane; otherwise, it computes the distance to the next vehicle on its current lane and memorizes it if it is higher than the current memorized maximal distance. Then if the memorized distance is lower than the maximal distance the agent can travel and if there is an upper lane, the agents test the probability to change its lane to an upper one. If it decides to test the upper lane, the agent computes the distance to the next vehicle on this lane and memorizes it if it is higher than the current memorized maximal distance. At last, if the memorized distance is still lower than the maximal distance it can travel if the agent is on the highest lane and if there is a reverse road, the agent tests the probability to use the reverse road (linked road). If it decides to use the reverse road, the agent computes the distance to the next vehicle on the lane 0 of this road and memorizes the distance if it is higher than the current memorized maximal distance.

More details about the driving skill can be found [here](#)

Version: 1.9.1

Manipulate Dates

Managing Time in Models

If some models are based on an abstract time - only the number of cycles is important - others are based on a real time. To this purpose, GAMA provides some tools to manage time.

First, GAMA allows the modeler to define the duration of a simulation step. It provides access to different time variables. At last, since GAMA 1.7, it provides a date variable type and some global variables allowing to use a real calendar to manage time.

Definition of the step and use of temporal unity values

GAMA provides three important [global variables to manage time](#):

- `cycle` (int - not modifiable): the current simulation step - this variable is incremented by 1 at each simulation step
- `step` (float - can be modified): the duration of a simulation step (in seconds). By default, the duration is one second.
- `time` (float - not modifiable): the current time spent since the beginning of the simulation - this variable is computed at each simulation step by: $\text{time} = \text{cycle} * \text{step}$.

The value of the cycle and time variables are shown in the top left (green rectangle) of the simulation interface. Clicking on the green rectangle allows to display either the number cycles or the time variable. Concerning this variable, it is presented following a years - months - days - hours - minutes - seconds format. In this presentation, every month is considered as being composed of 30 days (the different number of days of months are not taken into account).

Concerning step global variable, the variable can be modified by the modeler. A classic way of doing it consists of reediting the variable in the global section:

```
global {  
    float step <- 1 #hour;  
}
```

In this example, each simulation step will represent 1 hour. This time will be taken into account for all actions based on time (e.g. moving actions).

Note that the value of the `step` variable should be given in seconds. To facilitate the definition of the step value and of all expressions based on time, GAMA provides [different built-in constant variables accessible with the "#"](#) symbol:

- `#s` : second - 1 second
- `#mn` : minute - 60 seconds
- `#hour` : hour - 60 minutes - 3600 seconds
- `#day` : day - 24 hours - 86400 seconds
- `#week` : week - 7 days - 604800 seconds
- `#month` : month - 30 days - 2592000 seconds
- `#year` : year - 12 month - 3.1104E7 seconds

The date variable type and the use of a real calendar

Since GAMA 1.7, it is possible to use a real calendar to manage the time. For that, the modeler has only to define the starting date of the simulation. This variable is of type `date` which allows him/her to represent a date and time. A date variable has several attributes:

- `year` (int): the year component of the date
- `month` (int): the month component of the date
- `day` (int): the day component of the date
- `hour` (int): the hour component of the date
- `minute` (int): the minute component of the date
- `second` (int): the second component of the date
- `day_of_week` (int): the day of the week
- `week_of_year` (int): the week of the year

Several ways can be used to define a date. The simplest one consists in using a list of int values: [year,month of the year,day of the month, hour of the day, minute of the hour, second of the minute]

```
date my_date <- date([2010,3,23,17,30,10]); // the 23th of March 2010,  
at 17:30:10
```

Another way consists in using a string with the good format. The following one is perhaps the most complete, with year, month, day, hour, minute, second and also the time zone.

```
date my_date <- date("2010-3-23T17:30:10+07:00");
```

But the following ones can also be used:

```
// without time zone:  
my_date3 <- date("2010-03-23 17:30:10");  
//Dates (without time)  
my_date3 <- date("20100323");  
my_date3 <- date("2010-03-23");  
// Dates using some patterns:  
my_date3 <- date("03 23 2010", "MM dd yyyy");  
my_date3 <- date("01 23 20", "HH mm ss");
```

Note that the current (real) date can be accessed through the `#now` built-in variable (variable of type date).

In addition, GAMA provides different useful operators working on dates. For instance, it is possible to compute the duration in seconds between 2 dates using the `" - "` operator. The result is given in seconds:

```
float d <- starting_date - my_date;
```

It is also possible to add or subtract a duration (in seconds) to a date:

```
write "my_date + 10: " + (my_date + 10);  
write "my_date - 10: " + (my_date - 10);
```

At last, it is possible to add or subtract a duration (in years, months, weeks, days, hours, minutes, seconds) to a date:

```

write "my_date add_years 1: " + (my_date add_years 1);
write "my_date add_months 1: " + (my_date add_months 1);
write "my_date add_weeks 1: " + (my_date add_weeks 1);
write "my_date add_days 1: " + (my_date add_days 1);
write "my_date add_hours 1: " + (my_date add_hours 1);
write "my_date add_minutes 1: " + (my_date add_minutes 1);
write "my_date add_seconds 1: " + (my_date add_seconds 1);

write "my_date subtract_years 1: " + (my_date subtract_years 1);
write "my_date subtract_months 1: " + (my_date subtract_months 1);
write "my_date subtract_weeks 1: " + (my_date subtract_weeks 1);
write "my_date subtract_days 1: " + (my_date subtract_days 1);
write "my_date subtract_hours 1: " + (my_date subtract_hours 1);
write "my_date subtract_minutes 1: " + (my_date subtract_minutes 1);
write "my_date subtract_seconds 1: " + (my_date subtract_seconds 1);

```

Date variables in the model

For the modelers, two global date variables are available:

- `starting_date`: date considered as the beginning of the simulation (by default the starting date is `1970-01-01 07:00:00`).
- `current_date`: current date of the simulation.

Defining a value of the `starting_date` allows to change the normal time management of the simulation by a more realistic one (using a calendar):

```

global {
    date starting_date <- date([1979,12,17,19,45,10]);
}

```

When a value is set to this variable, the `current_date` variable is automatically initialized with the same value. However, at each simulation step, the `current_date`

variable is incremented by the `step` variable. The value of the `current_date` will replace the value of the time variable in the top left green panel.

Note that you have to be careful when a real calendar is used, the built-in constants `#month` and `#year` should not be used as there are not consistent with the calendar (where month can be composed of 28, 29, 30 or 31 days).

Version: 1.9.1

Implementing light

When using OpenGL display, GAMA provides you the possibility to manipulate one or several lights, making your display more realistic. Most of the following screenshots will be taken with the following short example gaml:

```
model test_light

grid cells {
    aspect base {
        draw square(1) at:{grid_x,grid_y} color:#white;
    }
}

experiment my_experiment type:gui{
    output {
        display my_display type: opengl background: #darkblue {
            species cells aspect: base;
            graphics "my_layer" {
                draw square(100) color:#white at:{50,50};
                draw cube(5) color:#lightgrey at:{50,30};
                draw cube(5) color:#lightgrey at:{30,35};
                draw cube(5) color:#lightgrey at:{60,35};
                draw sphere(5) color:#lightgrey at:{10,10,2.5};
                draw sphere(5) color:#lightgrey at:{20,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,60,2.5};
                draw cone3D(5,5) color:#lightgrey at:{55,10,0};
                draw cylinder(5,5) color:#lightgrey at:{10,60,0};
            }
        }
    }
}
```

Index

- [Light generalities](#)
- [Default light](#)
- [Custom lights](#)

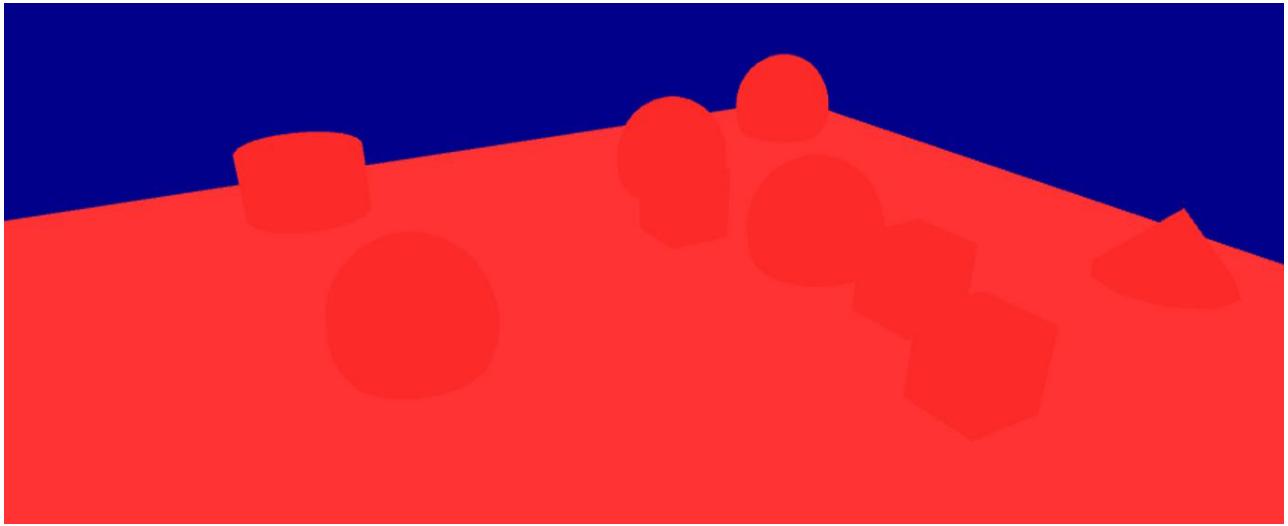
Light generalities

Before going deep into the code, here is a quick explanation about how light works in OpenGL. First of all, you need to know that there are 3 types of lights you can manipulate: the **ambient light**, the **diffuse light** and the **specular light**. Each "light" in OpenGL is in fact composed of those 3 types of lights.

Ambient light

The **ambient light** is the light of your world without any lighting. If a face of a cube is not stricken by the light rays, for instance, this face will appear totally black if there is no ambient light. To make your world more realistic, it is better to have ambient light. Ambient light has then no position or direction. It is equally distributed to all the objects of your scene.

Here is an example of our GAML scene using only ambient light (color red) (see below [how to define ambient light in GAML](#)):



Diffuse light

The **diffuse light** can be seen as the light rays: if a face of a cube is stricken by the diffuse light, it will take the color of this diffuse light. You have to know that the more perpendicular the face of your object will be to the light ray, the more lightened the face will be.

A diffuse light has then a direction. It can have also a position. You have 2 categories of diffuse light: the **positional lights**, and the **directional lights**.

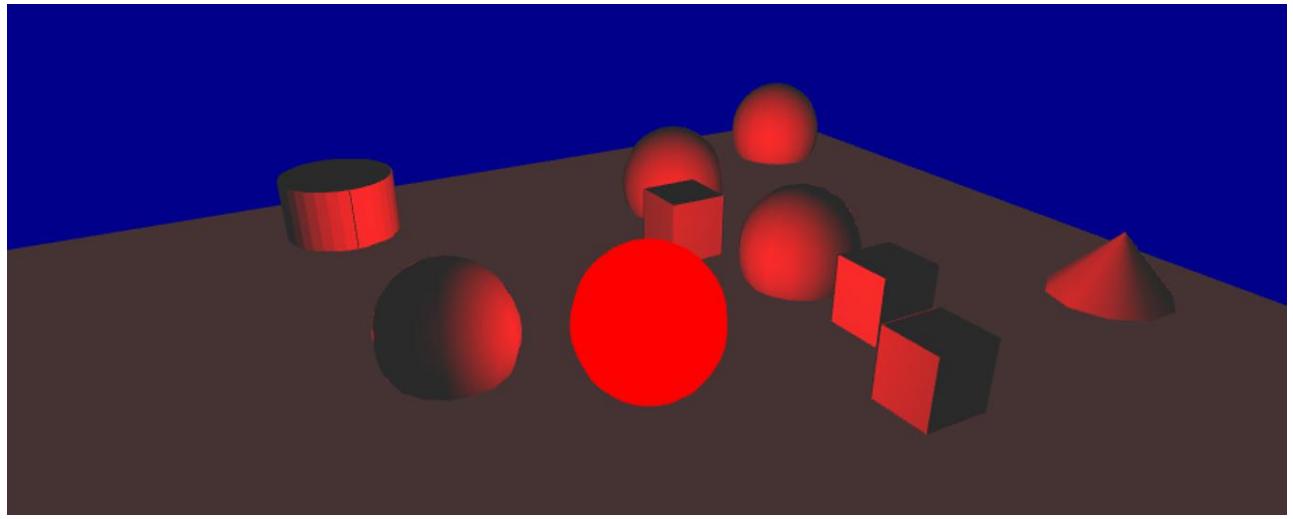
Positional lights

Those lights have a position in your world. It is the case of **point lights** and **spot lights**.

- **Point lights**

Point lights can be seen as a candle in your world, diffusing the light equally in all the direction.

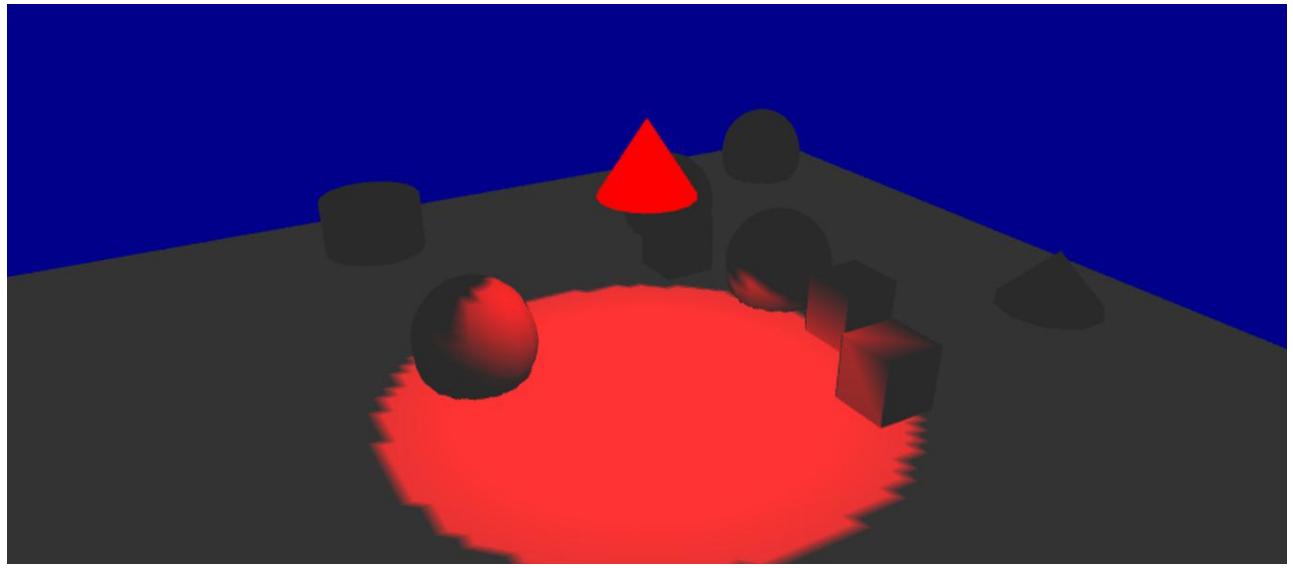
Here is an example of our GAML scene using only diffuse light, with a point light (color red, the light source is displayed as a red sphere) :



- **Spot lights**

Spot lights can be seen as a torch light in your world. It needs a position, and also a direction and an angle.

Here is an example of our GAML scene using only diffusion light, with a spot light (color red, the light source is displayed as a red cone) :



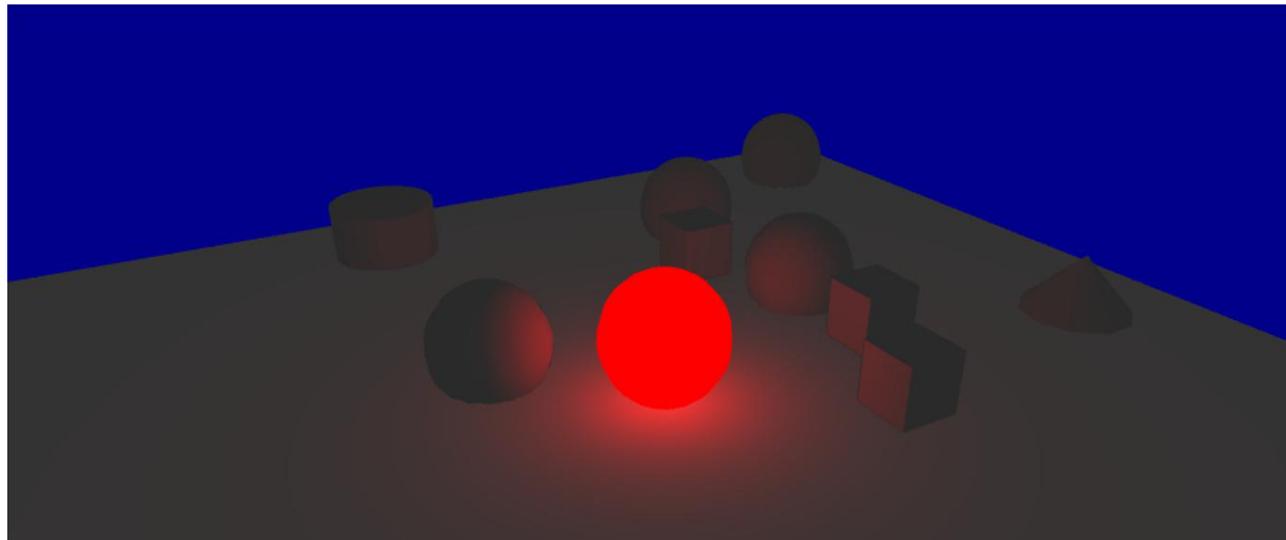
Positional lights, as they have a position, can also have an attenuation according to the distance between the light source and the object. The value of positional lights

are computed with the following formula:

```
diffuse_light = diffuse_light * ( 1 / (1 + constante_attenuation +
linear_attenuation * d + quadratic_attenuation * d))
```

By changing those 3 values (constante_attenuation, linear_attenuation and quadratic_attenuation), you can control the way light is diffused over your world (if your world is "foggy" for instance, you may turn your linear and quadratic attenuation on). Note that by default, all those attenuations are equal to 0.

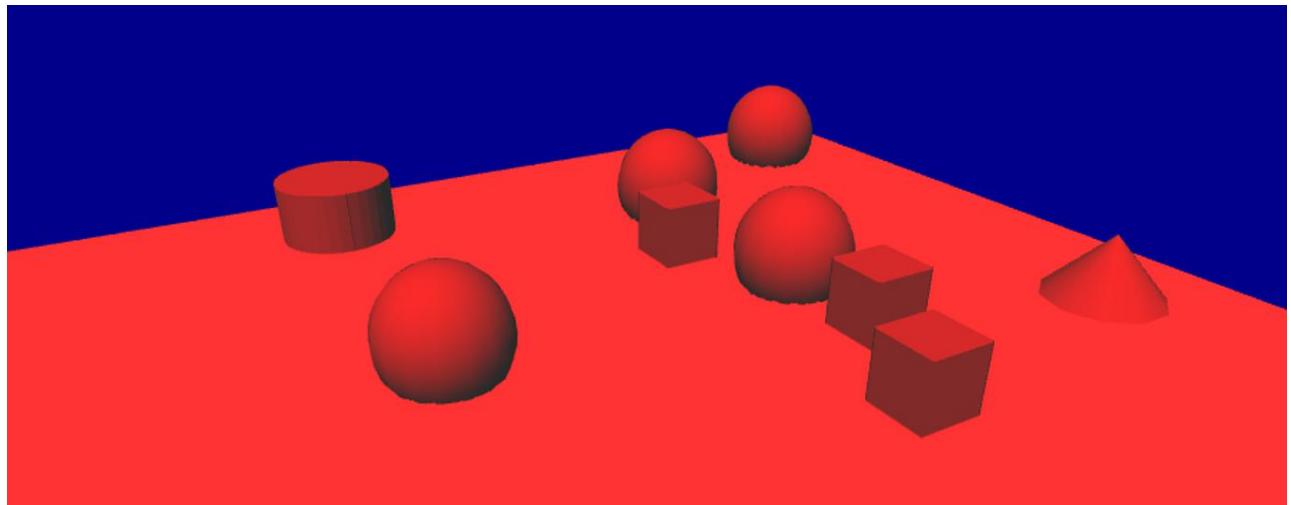
Here is an example of our GAML scene using only diffusion light, with a point light with linear attenuation (color red, the light source is displayed as a red sphere):



Directional lights

Directional lights have no real "position": they only have a direction. A directional light will strike all the objects of your world in the same direction. An example of directional light you have in the real world would be the light of the sun: the sun is so far away from us that you can consider that the rays have the same direction and the same intensity wherever they strike. Since there is no position for directional lights, there is no attenuation either.

Here is an example of our GAML scene using only diffusion light, with a directional light (color red) :



Specular light

This is a more advanced concept, giving an aspect a little bit "shiny" to the objects stricken by the specular light. It is used to simulate the interaction between the light and a special material (ex: wood, steel, rubber...). This specular light is not implemented yet in GAMA, only the two others are.

Default light

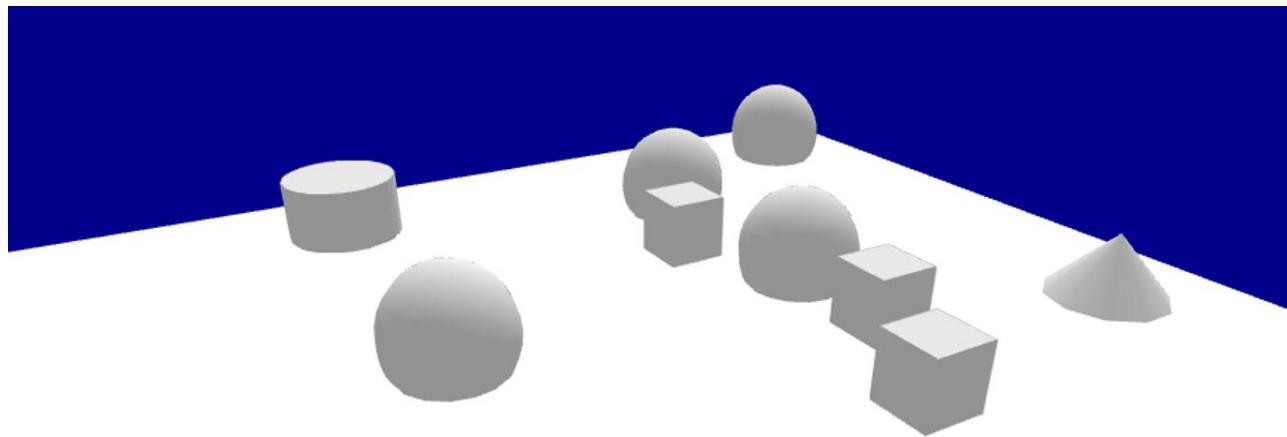
In your OpenGL display, without specifying any light, you will have only one light, with those following properties :

Those values have been chosen in order to have the same visual effect in both OpenGL and java2D displays, when you display 2D objects, and also to have a nice "3D effect" when using the OpenGL displays. We chose the following setting by default:

- The ambient light value: `rgb(127,127,127,255)`

- diffuse light value: `rgb(127,127,127,255)`
- type of light: `direction`
- direction of the light: `(0.5,0.5,-1);`

Here is an example of our GAML scene using the default light:



Custom lights

In your OpenGL display, you can create several lights, giving them the properties you want.

In order to add lights, or modifying the existing lights, you have to use the statement `light` inside your `display` scope:

```
experiment my_experiment type:gui {
    output {
        display "my_display" type:opengl {
            light "my_light";
        }
    }
}
```

A name has to be declared for the light. Through this facet, you can specify which light you want. Once you are manipulating a light through the `light` statement, the light is turned on. To switch off the light, you have to add the facet `active`, and turn it to `false`. The light you are declaring through the `light` statement is, in fact, a "diffuse" light. You can specify the color of the diffuse light through the facet `intensity` (by default, the color will be turned to white). Another very important facet is the `type` facet. This facet accepts a value among `#direction`, `#point` and `#spot`.

Ambient light

The ambient light can be set when declaring a light, using the `#ambient` constant, through the facet `intensity`:

```
experiment my_experiment type: gui {
    output {
        display "my_display" type: opengl {
            light #ambient intensity: 100;
        }
    }
}
```

Note for developers: Note that this ambient light is set to the `GL_LIGHT0`. This `GL_LIGHT0` only contains an ambient light, and no either diffuse nor specular light.

Declaring direction light

A direction light, as explained in the first part, is a light without any position. Instead of the facet `position`, you will use the facet `direction`, giving a 3D vector.

Example of implementation:

```
light "my_direction_light" type: #direction direction: {1,1,1}
intensity: #red;
```

Declaring point light

A point light will need a facet `position`, in order to give the position of the light source.

Example of implementation of a basic point light:

```
light "my_point_light" type: #point location: {10,20,10} intensity:
#red;
```

You can add, if you want, a custom attenuation of the light, through the facets `linear_attenuation` or `quadratic_attenuation`.

Example of implementation of a point light with attenuation :

```
light "my_point_light" type: #point location: {10,20,10} intensity:
#red linear_attenuation: 0.1;
```

Declaring spot light

A spot light will need the facet `position` (a spot light is a positional light) and the facet `direction`. A spot light will also need a special facet `spot_angle` to determine the angle of the spot (by default, this value is set to 45 degree).

Example of implementation of a basic spot light:

```
light "my_spot_light" type: #spot location:
{0,0,100}direction:{0.5,0.5,-1} intensity: #red angle: 20;
```

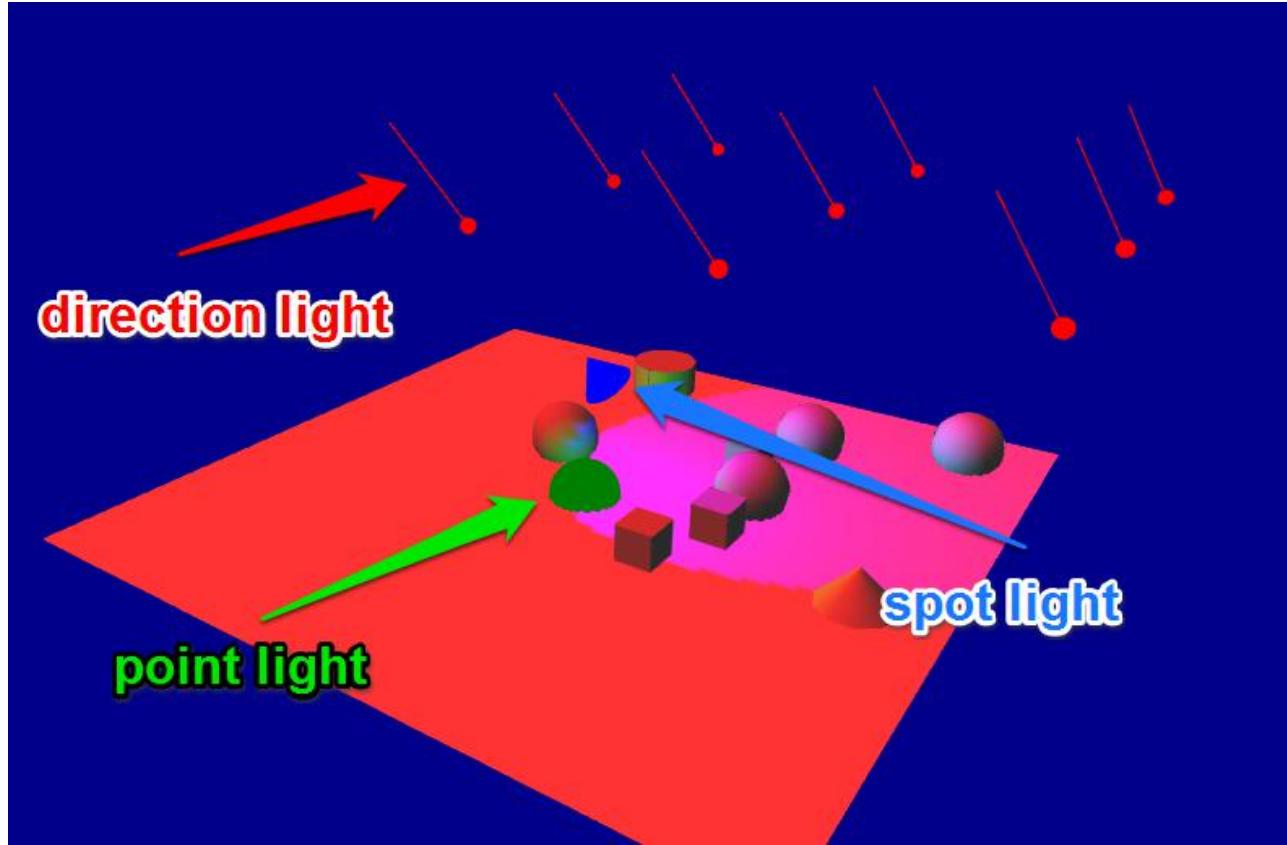
Same as for point light, you can specify an attenuation for a spot light.

Example of implementation of a spot light with attenuation:

```
light "my_spot_light" type:#spot location:{0,0,100}
direction:{0.5,0.5,-1} intensity:#red angle:30 linear_attenuation:
0.1;
```

Note that when you are working with lights, you can display your lights through the facet `show` (of `light`) to help you to implement your model. The three types of lights are displayed differently:

- The **point** light is represented by a sphere with the color of the diffuse light you specified, in the position of your light source.
- The **spot** light is represented by a cone with the color of the diffuse light you specified, in the position of your light source, the orientation of your light source. The size of the base of the cone will depend on the angle you specified.
- The **direction** light, as it has no real position, is represented with arrows a bit above the world, with the direction of your direction light, and the color of the diffuse light you specified.



Note for developers: Note that, since the GL_LIGHT0 is already reserved for the ambient light (only !), all the other lights (from 1 to 7) are the lights from GL_LIGHT1 to GL_LIGHT7.

Version: 1.9.1

Using Comodel

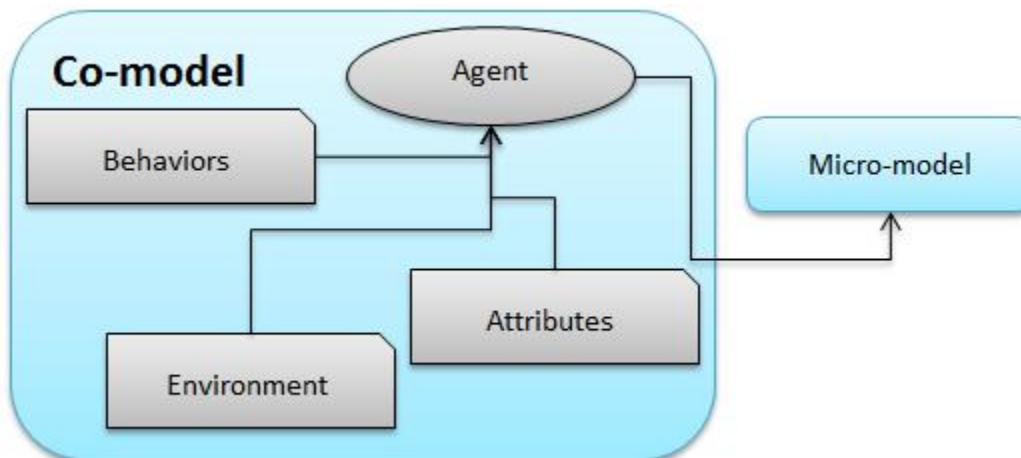
Introduction

In the trend of developing a complex system of multi-disciplinary, composing and coupling models are days by days becoming the most attractive research objectives. GAMA is supporting the co-modeling and co-simulation which are supposed to be a common coupling infrastructure.

Example of a Comodel

A Comodel is a model, especially an agent-based model, composed of several sub-models, called **micro-models**. A comodel itself could be also a micro-model of another comodel. From the point of view of a micro-model, the comodel is called a **macro-model**.

A micro-model must be imported, instantiated, and life-controlled by a macro-model.



Why and when can we use Comodel?

Co-models can definitely be very useful when the whole model can be decomposed in several sub-models, each of them representing, in general, a dynamics of the whole model, and that interact through some entities of the model. In particular, it allows several modelers to develop the part of the model dedicated to their expertise field, to test it extensively, before integrating it inside the whole model (where integration tests should not be omitted!).

Use of Comodel in a GAML model

The GAML language has evolved by extending the import section. The old importation told the compiler to merge all imported elements into one model, but the new one allows modelers to keep the elements coming from imported models separately from the caller model.

Definition of a micro-model

Defining a micro-model of comodel is to import an existing model with an alias name. The syntax is:

```
import <path to the GAML model> as <identifier>
```

The identifier is then become the new name of the micro-model.

As an example taken from the model library, we can write:

```
import "Prey Predator Adapter.gaml" as Organism
```

Instantiation of a micro-model

After the importation and giving an identifier, micro-model must be explicitly instantiated. It could be done by the `create` statement.

```
create <micro-model identifier> . <experiment name> [optional parameter];
```

The `<experiment name>` is an experiment inside micro-model. This syntax will generate some experiment agents and attach an implicit simulation.

Note: The creation of several instances is not multi-simulation, but multi-experiment. Modelers could create an experiment with multi-simulation by explicitly do the init inside the experiment scope.

As an example taken from the model library, we can write:

```
global {
    init {
        //instantiate three instant of micro-model PreyPredator
        create Organism.Simple number: 3 with: [shape::square(100),
preyinit::10, predatorinit::1] ;
    }
}
```

Control micro-model life-cycle

A micro-model can be controlled as any normal agent by asking the corresponding identifier, and also be destroyed by the `do die;` statement. And it can be recreated any time we need.

```
ask (<micro-model identifier> . <experiment name> at <number>) .
simulation {
    ...
}
```

More generally, to schedule all the created simulations, we can do:

```
reflex simulate_micro_models {
    // ask all simulation do their job
    ask (Organism.Simple collect each.simulation) {
        do _step_;
    }
}
```

Visualization of the micro-model

The micro-model species could display in comodel with the support of agent layer

```
agents "name of layer" value: (<micro-model> . <experiment name> at
<number>).<get List of agents>;
```

As an example:

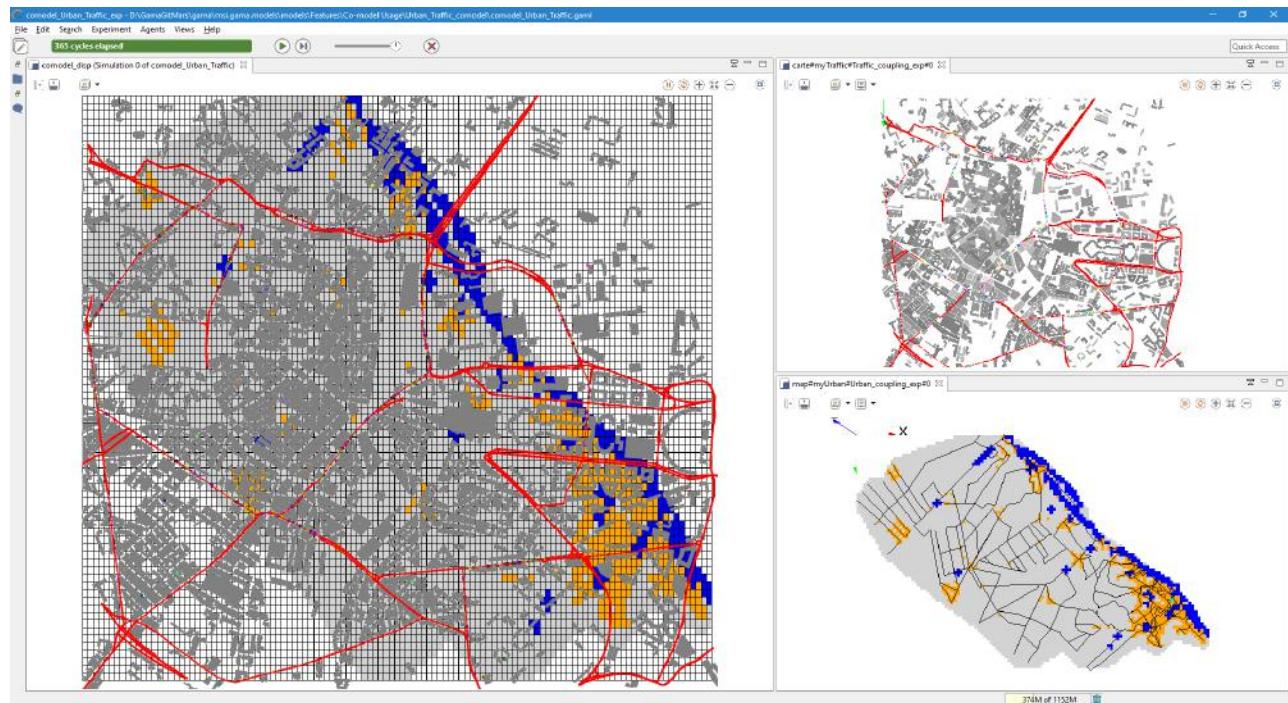
```
display "Comodel display" {
    agents "agentprey" value: (Organism.Simple accumulate
each.get_prey());
    agents "agentpredator" value: (Organism.Simple accumulate
each.get_predator());
}
```

More details

Example of the comodel

The following illustrations are taken from the model library provided with the GAMA platform.

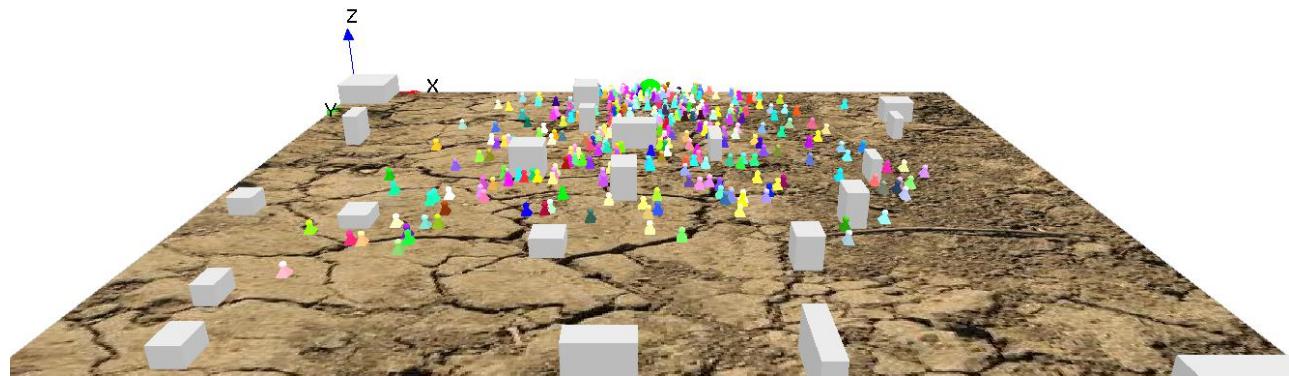
Urbanization model with a Traffic model



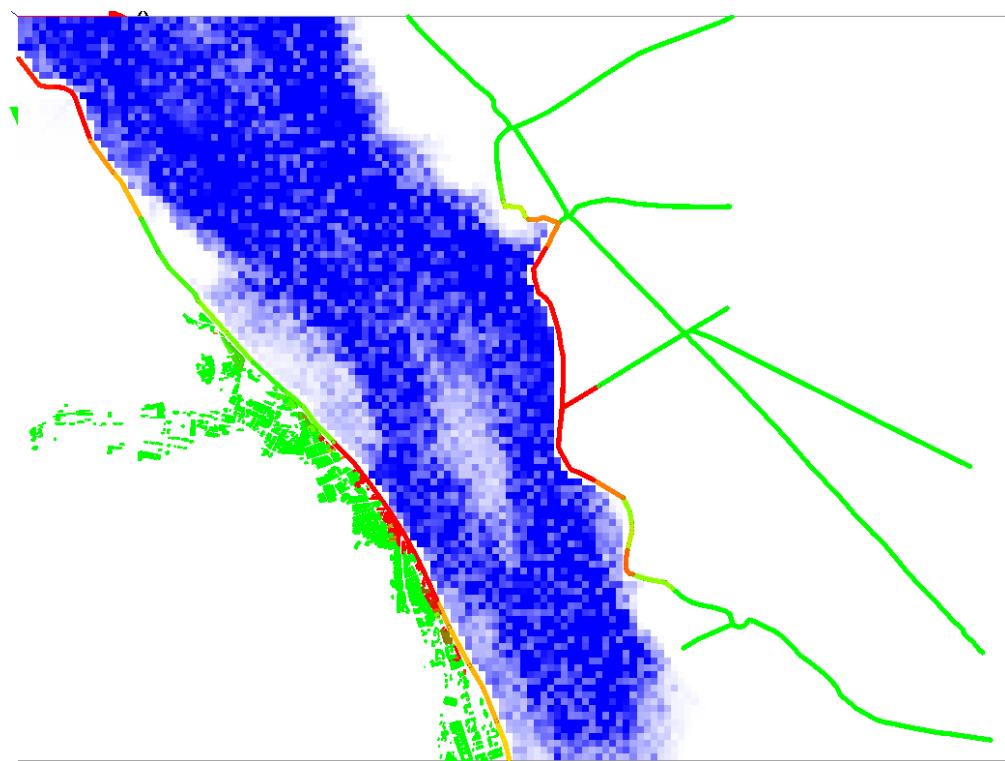
Flood model with Evacuation model

The aim of this model is to couple the two existing models: Flood Simulation and Evacuation.

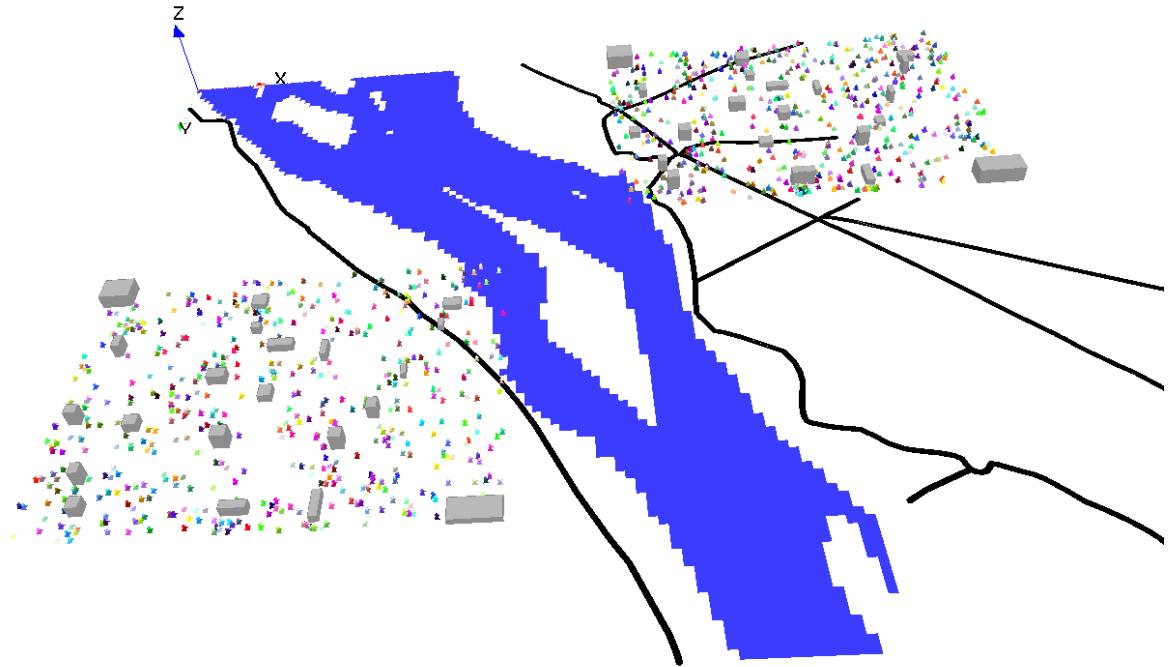
Toy Models/Evacuation/models/continuous_move.gaml



Toy Models/Flood Simulation/models/Hydrological Model.gaml



The comodel explores the effect of a flood on an evacuation plan:



Simulation results:



Version: 1.9.1

Save and Restore simulations

Last version of GAMA has introduced new features to save the state of a simulation at a given simulation cycle. This has two main applications:

- The possibility to save the state of a simulation
- The possibility to restore a simulation from this file.
- The possibility to go backward to an older state of a simulation.

Save a simulation

```
experiment saveSimu type: gui {  
  
    reflex store when: cycle = 5 {  
        write "===== START SAVE + self " + " - " + cycle  
;  
        write "Save of simulation : " +  
        save_simulation('saveSimu.gsim');  
        write "===== END SAVE + self " + " - " + cycle  
;  
    }  
  
    output {  
        display main_display {  
            species road aspect: geom;  
            species people aspect: base;  
        }  
    }  
}
```

Restore a simulation

```
experiment reloadSavedSimuOnly type: gui {  
  
    action _init_ {  
        create simulation from:  
        saved_simulation_file("saveSimu.gsim");  
    }  
  
    output {  
        display main_display {  
            species road aspect: geom;  
            species people aspect: base;  
        }  
    }  
}
```

Memorize simulation

```
model memorize  
  
global {  
    init{  
        create people number:1;  
    }  
}  
  
species people skills: [moving] {  
  
    init{  
        location <- {50, 50};  
    }  
    reflex movement {
```


Version: 1.9.1

Using network

Introduction

GAMA provides features to allow agents to communicate with other agents (and other applications) through network and to exchange messages of various types (from simple number to agents). To this purpose, the `network` skill should be used on agents intending to use these capabilities.

Notice that in this communication, roles are asymmetric: the simulations should contain a server and some clients to communicate. Message exchanges are made between agents through this server. 6 protocols are supported (TCP, UDP, MQTT, HTTP, Websocket, Arduino):

- **when TCP, UDP or Websocket protocols are used:** Agents can be either clients or server depending on the needs of the simulation.
- **when the MQTT protocol is used:** all the agents are clients and the server is an external software. A free solution (ActiveMQ) can be freely downloaded from: <http://activemq.apache.org>.
- **when HTTP is used:** the agents can interact with webpages/webservices through raw GET, POST, PUT, and DELETE requests
- **when arduino is used:** one agent of the simulation can connect to an Arduino as a client.

Which protocol to use ?

In the GAMA network, 6 kinds of protocol can be used. Each of them has a particular

purpose.

- **MQTT**: this is the default protocol that should be used to make agents of various GAMA instances to communicate through a MQTT server (that should be run as an external application, e.g. ActiveMQ that can be downloaded from: <http://activemq.apache.org/>),
- **UDP**: this protocol should be limited to fast (and unsecured) exchanges of small pieces of data from GAMA to an external application (for example, mouse location from a Processing application to GAMA, c.f. model library),
- **TCP and Websocket**: these protocols can be used both to communicate between GAMA agents in a simulation or between GAMA and an external application.
- **HTTP requests**: this protocol should be used to communicate with an external webservice.
- **Arduino**: this protocol should be used to communicate with an arduino device

Disclaimer

In all the models using any network communication, the server should be launched before the clients. As a consequence, when TCP, Websocket or UDP protocols are used, a model creating a server agent should always be run first. Using MQTT protocol, the external software server should be launched before running any model using it.

Declaring a network species

To create agents able to communicate through a network, their species should have the skill `network`:

```
species Networking_Client skills: [network] {  
    ...  
}
```

A list exhaustive of the additional attributes and available actions provided by this skill are described here: [network skill preference page](#).

Creation of a network agent

The network agents are created as any other agents, but (in general) at the creation of the agents, the connection is also created, using the `connect` built-in action:

```
create Networking_Client {  
    do connect to: "localhost" protocol: "tcp_client" port: 3001  
    with_name: "Client";  
}
```

Each protocol has its specificities regarding the connection:

- **TCP:**
 - `protocol`: the 2 possible keywords are `tcp_server` or `tcp_client`, depending on the wanted role of the agent in the communication.
 - `port`: traditionally the port `3001` is used.
 - `raw`: false by default, it is better to turn it to `true` when communicating with external applications as it will remove all the wrapper informations used for communication inside gama.
- **Websocket:**
 - `protocol`: the 2 possible keywords are `websocket_server` or `websocket_client`, depending on the wanted role of the agent in the communication.

- **port**: traditionally the port 3001 is used.
- **raw**: false by default, it is better to turn it to true when communicating with external applications as it will remove all the wrapper informations used for communication inside gama.
- **UDP:**
 - **protocol**: the 2 possible keywords are `udp_server` or `udp_emitter`, depending on the wanted role of the agent in the communication.
 - **port**: traditionally the port 9876 is used.
- **MQTT:**
 - **protocol**: MQTT is the default protocol value (if no value is given, MQTT will be used)
 - **port**: traditionally the port 1883 is used (when ActiveMQ is used as the server application)
 - **admin** and **password**: traditionally the default login and password are "admin" (when ActiveMQ is used as the server application)
- **HTTP requests:**
 - **protocol**: the only keyword to use is `http`.
 - **port**: traditionally the port 80 is used for http connections and 443 for https.

Note: if no connection information is provided with the MQTT protocol (no `port`), then GAMA connects to an MQTT server provided by the GAMA community (for test purpose only!).

Sending messages

To send any message, the agent has to use the `send` action:

```
do send to: "server" contents: name + " " + cycle + " sent to server";
```

The network skill in GAMA allows the modeler to send simple string messages between agents but also to send more complex objects (and in particular agents). In this case, the use of the MQTT protocol is highly recommended.

```
do send to: "receiver" contents: (9 among NetworkingAgent);
```

Receiving messages

Asynchronous reading

The messages sent by other agents are received in the `mailbox` attribute of each agent. So to get its new message, the agent has simply to check whether it has a new message (with action `has_more_message()`) and fetch it (that gets it and remove it from the mailing box) with the action `fetch_message()`.

```
reflex fetch when: has_more_message() {
    message mess <- fetch_message();
    write name + " fecth this message: " + mess.contents;
}
```

Note that when an agent is received, the fetch of the message will recreate the agent in the current simulation.

Alternatively, the `mailbox` attribute can be directly accessed (notice that the `mailbox` is a list of messages):

```
reflex receive {
    if (length(mailbox) > 0) {
        write mailbox;
    }
}
```

Synchronous reading

In certain cases you need to wait for a message from another application to continue the execution of your simulation. To do so, you can use the `fetch_message_from_network` action to force the mailbox to refresh (which normally is only done once per cycle) until you receive a message:

```
reflex fetch {
    write "waiting for server to send data";
    loop while: !has_more_message() {
        do fetch_message_from_network;
    }

    //This second loop will only be reached once a message has been
    found into the agent's mailbox
    loop while: has_more_message() {
        message s <- fetch_message();
        write "at cycle: " + cycle + ", received from server: " +
        s.contents;
    }
}
```

Broadcasting a message to all the agents' members of a given group

Each time an agent creates a connection to another agent as a client, a way to communicate with it is stored in the `network_groups` attribute. So an agent can use this attribute to broadcast messages to all the agents with whom it can communicate:

```
reflex broad {
    loop id over: network_groups {
        do send to: id contents: "I am Server " + name + " I give
order to " + id;
    }
}
```

To go further:

- [network skill reference page](#).
- example models can be found in the GAMA model library, in: [Plugin models > Network](#).

Version: 1.9.1

Headless mode for dummies

Overview

This tutorial presents the headless mode usage of GAMA. We will execute the Predator-Prey model, already presented in [this tutorial](#). Headless mode is documented in its dedicated part, here, we focus on the definition of an experiment plan, where the model is run several times. We only consider the shell script execution, not the java command execution.

In headless-mode, GAMA can be seen as any shell command, whose behavior is controlled by passing arguments to it. You must provide 2 arguments :

- an **input experiment file**, used to describe the execution plan of your model, its inputs and the expected outputs.
- an **output directory**, where the results of the execution are stored

Headless-mode is a little more technical to handle than the general GAMA use-case, and the following commands and code have been solely tested on a Linux Ubuntu 22.04 machine with the default GAMA 1.9.0 (installer version, with embedded JDK).

You may have to perform some adjustments (such as paths definition) according to your machine, OS, java and GAMA versions and so on.

Setup

GAMA version

Headless mode is frequently updated by GAMA developers, so you have to get the very latest build version of GAMA.

You can download it here <https://github.com/gama-platform/gama/releases> Be sure to pick the **Continuous build** version (The name looks like `GAMA1.7_Linux_64_02.26.17_da33f5b.zip`) and **not** the major release, e.g.

`GAMA1.7_Linux_64.zip`. Big note on Windows OS (maybe on others), GAMA must be placed outside of several sensible folders (Program Files, Program Filesx64, Windows). RECOMMENDED: Place GAMA in Users Folder of windows OS.

gama-headless.sh script setup

The `gama-headless.sh` script can be found under the `headless` directory, in GAMA installation directory e.g. :
`~/GAMA/headless/`

Modifying the script (a little bit)

The original script looks like this :

```

#!/bin/bash
memory=2048m
declare -i i

i=0
echo ${!i}

for ((i=1;i<=$#;i=$i+1))
do
if test ${!i} = "-m"
then
    i=$i+1
    memory=${!i}
else
    PARAM=$PARAM\ ${!i}
    i=$i+1
    PARAM=$PARAM\ ${!i}
fi
done

echo ****
echo ** GAMA version 1.9.0
echo ** http://gama-platform.org
echo ** (c) 2007-2022 UMI 209 UMMISCO IRD/UPMC & Partners
echo ****
passWork=.work$RANDOM

java -cp ./plugins/org.eclipse.equinox.launcher*.jar -Xms512m -Xmx$memory -Djava.awt.headless=true
org.eclipse.core.launcher.Main -application msi.gama.headless.id4 -data $passWork $PARAM $mfull
$outputFile
rm -rf $passWork

```

Notice the final command of the script `rm -rf $passWork`. It is intended to remove the temporary file used during the execution of the script. For now, we should comment this command, in order to check the logs if an error appears:

```
#rm -rf $passWork
```

Setting the experiment file

Headless mode uses a XML file to describe the execution plan of a model. An example is given in the [headless mode documentation page](#).

The script looks like this : **N.B. this version of the script, given as an example, is deprecated**

```

<?xml version="1.0" encoding="UTF-8"?>
<Experiment_plan>
    <Simulation id="2" sourcePath="./predatorPrey/predatorPrey.gaml" finalStep="1000"
experiment="predPrey">
        <Parameters>
            <Parameter name="nb_predator_init" type="INT" value="53" />
            <Parameter name="nb_preys_init" type="INT" value="621" />
        </Parameters>

```

As you can see, you need to define 3 things in this minimal example:

- Simulation: its id, path to the model, finalStep (or stop condition), and name of the experiment
- Parameters name, of the model for *this* simulation (i.e. Simulation of id= 2)
- Outputs of the model: their id, name, type, and the rate (expressed in cycles) at which they are logged in the results file during the simulation

We now describe how to constitute your experiment file.

Experiment File: Simulation

id

For now, we only consider one single execution of the model, so the simulation `id` is not critical, let it unchanged. Later example will include different simulations in the same experiment file. Simulation `id` is a string. Don't introduce weird symbols into it.

sourcePath

`sourcePath` is the relative (or absolute) path to the model file you want to execute headlessly.

Here we want to execute the [fourth model of the Predator Prey tutorial suite](#), located in `~/GAMA/plugins/msi.gama.models_1.7.0.XXXXXXXXXXXXXX/models/Tutorials/Predator Prey/models` (with XXXXXXXXXXXX replaced by the number of the release you downloaded)

So we set `sourcePath="..../plugins/msi.gama.models_1.7.0.201702260518/models/Tutorials/Predator Prey/models/Model 07.gaml"` (Remember that the headless script is located in `~/GAMA/headless/`)

Depending on the directory you want to run the `gama-headless.sh` script, sourcePath must me modified accordingly. Another workaround for shell more advanced users is to define a `$GAMA_PATH`, `$MODEL_PATH` and `$OUTPUT_PATH` in `gama-headless.sh` script. Don't forget the quotes `"` around your path.

finalStep

The duration, in cycles, of the simulation.

experiment

This is the name of (one of) the experiment statement at the end of the model code.

In our case there is only one, called `prey_predator` and it looks like this :

```
experiment prey_predator type: gui {  
    parameter "Initial number of preys: " var: nb_preys_init min: 1 max: 1000 category: "Prey" ;
```

So we are now able to constitute the entire Simulation tag:

```
<Simulation id="2" sourcePath="~/GAMA/plugins/msi.gama.models_1.7.0.201702260518/models/Tutorials/  
Predator Prey/models/Model 01.gaml" finalStep="1000" experiment="prey_predator">
```

N.B. the numbers after `msi.gama.models` (the number of your GAMA release actually) have to be adapted to your own release of GAMA number. The path to the GAMA installation directory has also to be adapted of course.

Experiment File: Parameters

The parameters section of the experiment file describes the parameters names, types and values to be passed to the model for its execution.

Let's say we want to fix the number of preys and their max energy for this simulation. We look at the experiment section of the model code and use their **title**. The title of a parameter is the name that comes right after the `parameter` statement. In our case, the strings "Initial number of preys: " and "Prey max energy: " (Mind the spaces, quotes and colon)

The parameters section of the file would look like :

```
<Parameters>  
  <Parameter name="Initial number of preys: " type="INT" value="621" />  
  <Parameter name="Prey max energy: " type="FLOAT" value="1.0" />  
</Parameters>
```

Any declared parameter can be set this way, yet you don't have to set all of them, provided they are initialized with a default value in the model (see the global statement part of the model code).

Experiment File: Outputs

Output section of the experiment file is pretty similar to the previous one, except for the `id` that have to be set for each of the outputs .

We can log some of the declared outputs : `main_display` and `number_of_preys`.

The outputs section would look like the following:

```
<Outputs>  
  <Output id="1" name="main_display" framerate="10" />  
  <Output id="2" name="Number of preys" framerate="1" />  
</Outputs>
```

Outputs must have an id, a name, and a framerate.

- `id` is a number that identifies the output
- framerate is the rate at which the output is written in the result file. It's a number of cycle of simulation (integer). In this example the display is saved every 10 cycle
- `name` is either the "title" of the corresponding monitor. In our case, the second output's is the title of the monitor "Number of preys", i.e. "Number of preys"

We also save a `display` output, that is an image of the simulation graphical display named `main_display` in the code of the model. These images is what you would have seen if you had run the model in the traditional GUI mode.

Execution and results

Our new version of the experiment file is ready :

```
<?xml version="1.0" encoding="UTF-8"?>
<Experiment_plan>
    <Simulation id="2" sourcePath="/absolute/path/to/your/model/file/Model_04.gaml"
finalStep="1000" experiment="prey_predator">
        <Parameters>
            <Parameter name="Initial number of preys: " type="INT" value="621" />
            <Parameter name="Prey max energy: " type="FLOAT" value="1.0" />
        </Parameters>
        <Outputs>
            <Output id="1" name="main_display" framerate="10" />
            <Output id="2" name="Number of preys" framerate="1" />
        </Outputs>
    </Simulation>
</Experiment_plan>
```

Execution

We have to launch the `gama-headless.sh` script and provide two arguments : the experiment file we just completed and the path of a directory where the results will be written.

Warning In this example ,we are lazy and define the source path as the absolute path to the model we want to execute. If you want to use a relative path, note that it has to be define relatively to the location of your **ExperimentFile.xml location** (and the location where you launched the script)

In a terminal, position yourself in the headless directory : `~/GAMA/headless/'.

Then type the following command :

```
gama-headless.sh -v ~a/path/to/MyExperimentFile.xml /path/to/the/desired/output/directory
```

And replace paths by the location of your ExperimentFile and output directory

You should obtain the following output in the terminal :

```
*****
* GAMA version 1.7.0 V7 *
* http://gama-platform.org *
* (c) 2007-2016 UMI 209 UMMISCO IRD/UPMC & Partners *
*****  

>GAMA plugin loaded in 2927 ms: msi.gama.core  

>GAMA plugin loaded in 67 ms: ummisco.gama.network  

>GAMA plugin loaded in 56 ms: simtools.gaml.extensions.traffic  

>GAMA plugin loaded in 75 ms: simtools.gaml.extensions.physics  

>GAMA plugin loaded in 1 ms: irit.gaml.extensions.test  

>GAMA plugin loaded in 75 ms: ummisco.gaml.extensions.maths  

>GAMA plugin loaded in 47 ms: msi.gaml.extensions.fipa  

>GAMA plugin loaded in 92 ms: ummisco.gama.serialize  

>GAMA plugin loaded in 49 ms: irit.gaml.extensions.database  

>GAMA plugin loaded in 2 ms: msi.gama.lang.gaml  

>GAMA plugin loaded in 1 ms: msi.gama.headless  

>GAMA plugin loaded in 103 ms: ummisco.gama.java2d  

>GAMA plugin loaded in 189 ms: msi.gaml.architecture.simplebdi  

>GAMA plugin loaded in 129 ms: ummisco.gama.opengl  

>GAMA building GAML artefacts>GAMA total load time 4502 ms.  

  in 714 ms  

cpus :8  

Simulation is running...  

.....  

Simulation duration: 7089ms
```

Results

The results are stored in the output directory you provided as the second argument of the script.

3 items have appeared:

- A `console_output.txt` file, containing the output of the GAMA console of the model execution if any
- a XML file `simulation-outputXX.xml`, where XX is the `id` number of your simulation. In our case it should be 2.
- the folder `snapshots` containing the screenshots coming from the second declared output: `main_display`. image name format is `main_display[id]_[cycle].png`.

The values of the monitor "Number of preys" are stored in the xml file `simulation-outputXX.xml`

Common error messages

`Exception in thread "Thread-7" No parameter named prey_max_energy in experiment prey_predator` Probably a typo in the name or the title of a parameter. check spaces, capital letters, symbols and so on.

`java.io.IOException: Model file does not exist: /home/ubuntu/dev/tutoGamaHeadless/../plugins/msi.gama.models_1` This may be a relative path mistake; try with absolute path.

`java.lang.NumberFormatException: For input string: "1.0"` This may be a problem of type declaration in the parameter section.

Going further

Experiments of several simulation

You can launch several simulation by replicating the simulation declaration in your ExperimentFile.xml and varying the values of the parameters. Since you will have to edit the experiment file by hand, you should do that only for a reasonable number of simulations (e.g. <10)

Design of experiments plans

For more systematic parameter values samples, you should turn towards a more adapted tool such as GAMAR, to generate a `ExperimentFile.xml` with a huge number of simulations.

Version: 1.9.1

Calling gama from another program

This tutorial presents an example for using Headless. The tutorial shows how to use Headless Legacy mode, Headless batch and Headless server. All the files related to this tutorial (images and models) are available in the Headless folder (headless/samples/predatorPrey).

1. Example using python with Headless legacy

```
import os

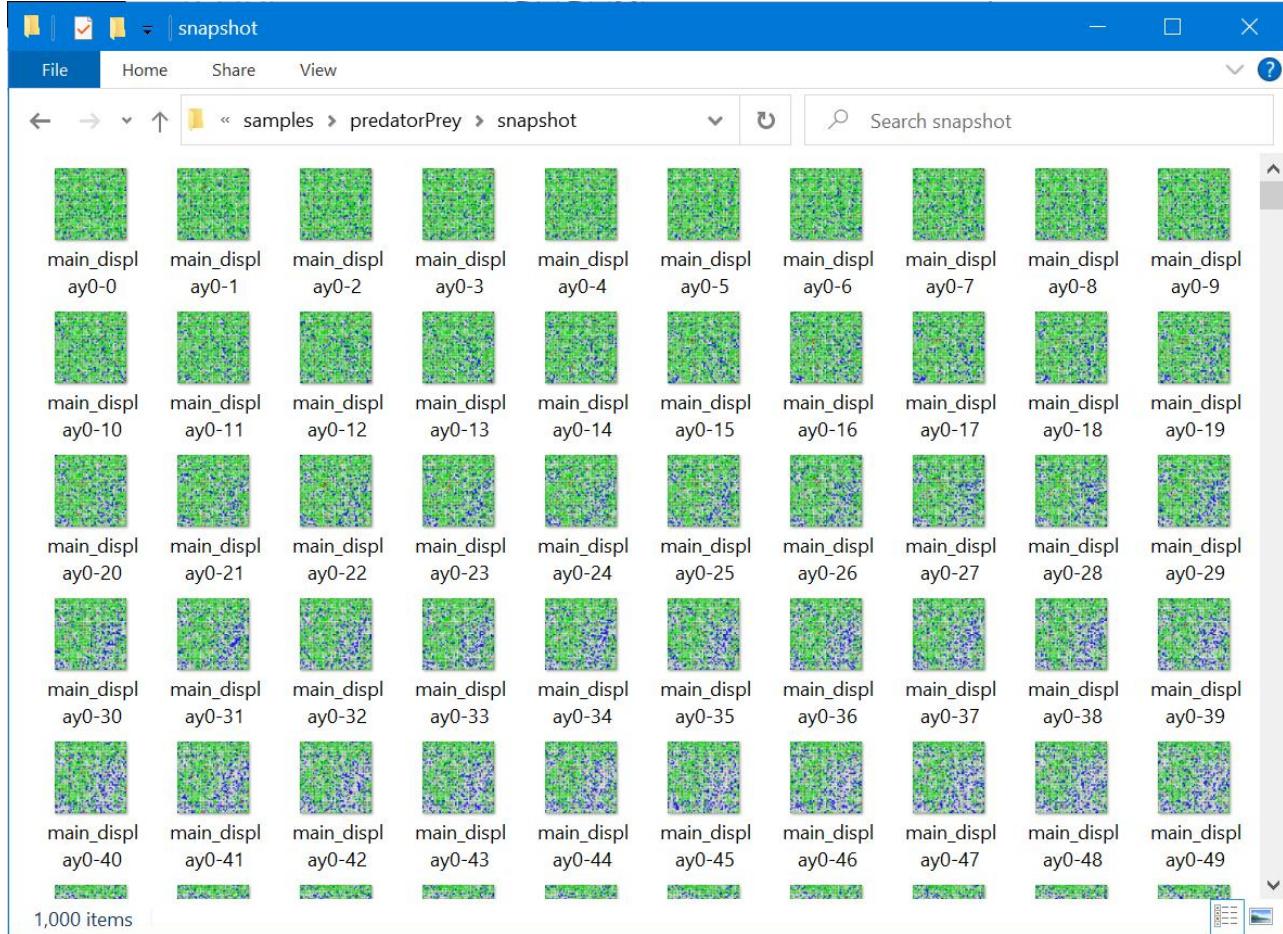
GAMA_folder_with_SDK = r"D:\software\GAMA_1.9.0_Windows_with_JDK\headless"

Model_file = GAMA_folder_with_SDK + r"\samples\predatorPrey\predatorPrey.gaml"

ExperimentName = "prey_predator"
XML_file = GAMA_folder_with_SDK + r"\samples\predatorPrey.xml"
Output_folder = GAMA_folder_with_SDK + r"\samples\predatorPrey"

os.chdir(GAMA_folder_with_SDK)
os.system("gama-headless.bat -xml " + ExperimentName + " " +
Model_file + " " + XML_file)
os.system("gama-headless.bat " + XML_file + " "+ Output_folder)
```

The results of the experiment is stored in the set folder. In which the snapshot for every step is also saved in the snapshot folder.



2. Example on using python with Headless batch

```
import os
GAMA_folder_with_SDK = r"D:\software\GAMA_1.9.0_Windows_with_JDK\
headless"

Model_file = GAMA_folder_with_SDK + r"\samples\predatorPrey\
predatorPrey.gaml"
```

3. Example on using python with Headless server

The legacy version allows you to access the headless feature of GAMA by controlling the model parameters and experiment plan from outside GAMA model file . The headless batch, allows you to access the headless feature of GAMA with the model parameters and experiment plans defined inside the GAMA model file. The headless server, allows you to not only to access the headless feature but also to interact with the currently running GAMA experiment. You can load, play, pause, reload, stop and exit an experiment with very specific commands as described [here](#).

The general sequence of operations is:

- Start the server from a command line `gama-headless.sh -socket 6868` , this opens the communication via port 6868 using websockets.
- Connect to the server from another application/script that supports interacting with websockets. e.g., python. See below to use a python wrapper.
- Start with the `Load` command to load an experiment and then use one of the specific commands as described [here](#) to construct a sequence of operations as required by your workflow.

Start the GAMA server

On your command line, execute the following command, you will find the gama-headless.sh in the headless folder inside your GAMA installation.

```
gama-headless.sh -socket 6868
```

Use the Python wrapper instead

The GAMA developers have made available an elegant python wrapper that simplifies using GAMA server with python scripts and is available [here](#). However if you are not a serious programmer and just want to use this tool, the following bare minimum code shall get you started and you can slowly add one command after another to build your sequence of operations to interact with the GAMA server. Before you can start, you have to install the wrapper. In your python environment, install the **gama-client** package with the command:

```
pip install gama-client
```

You can check that everything went well by opening a python console and try the following line:

```
from gama_client.base_client import GamaBaseClient
```

If you don't see any error messages, then the python wrapper has been installed correctly.

Bare minimum code

The whole interaction with the GAMA server is facilitated using the `asyncio` library in Python and our wrapper that we installed in the previous step. The discussion on use of `asyncio` is beyond the scope of this tutorial, so just take it as granted. This whole interaction can be considered a dialouge (two way communication) between the client (you/ your script) and the server (GAMA server). You send a command to the server, and the server sends back a message. You parse this message and its contents and construct the next command to interact with the server. This back and forth continues until you use the `exit` command or if an error occurs on the server.

Among all the messages sent by the server, as a beginner you should know about

these four main messages: `ConnectionSuccessful` (you connected to the server), `CommandExecutedSuccessfully` (your command was well received and executed), `UnableToExecuteRequest` (something is wrong with your model), `MalformedRequest` (something is wrong with your command format)

Just run the following python script and if all goes well, you are ready to use the GAMA server via python.

```
import asyncio
from gama_client.base_client import GamaBaseClient

async def message_handler(message):
    print("received message:", message)

async def main():
    client = GamaBaseClient("localhost", 6868, message_handler)
    await client.connect(False)

    while True:
        await asyncio.sleep(1)

if __name__ == "__main__":
    asyncio.run(main())
```

It gets even better !

A word of caution It is recommended that you slowly build on to the above script by adding `commands` step by step. You may use the script below as guidance to learn and to stay on course and not having to search a lot through the documentation. Blindly copy-pasting the code and changing the parameters without understanding is not advised.

The python wrapper makes it even easier for beginners. So easy that you just have to change values of the following 5 variables in the sample python script below to make use of GAMA server.

```
MY_SERVER_URL = "localhost"
MY_SERVER_PORT = 6868
GAML_FILE_PATH_ON_SERVER = r"D:\Gama\headless\samples\
predatorPrey\predatorPrey.gaml"
EXPERIMENT_NAME = "prey_predatorExp"
MY_EXP_INIT_PARAMETERS = [{"type": "int", "name": "nb_preys_init", "value": 100}]
```

Sample python script

```
import asyncio
from asyncio import Future
from typing import Dict

from gama_client.base_client import GamaBaseClient
from gama_client.command_types import CommandType
from gama_client.message_types import MessageType

experiment_future: Future
play_future: Future
pause_future: Future
expression_future: Future
step_future: Future
stop_future: Future

async def message_handler(message: Dict):
    print("received", message)
    if "command" in message:
        if message["command"]["type"] == CommandType.Load.value:
            experiment_future.set_result(message)
        elif message["command"]["type"] == CommandType.Play.value:
            play_future.set_result(message)
```


Version: 1.9.1

Writing Unit Tests in GAML

Unit testing is an essential instrument to ensure the quality of any software and it has been implemented in GAMA: this allows in particular that parts of the model are behaving as expected and that evolutions in the model do not introduce unexpected changes. To these purposes, the modeler can define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue.

Writing tests in GAML involves the use of 4 keywords:

- `assert` statement,
- `test` statement,
- `setup` statement,
- `type: test` facet of `experiment`.

In this unit testing tutorial, we intend to show how to write unit tests in GAML using the statement `test`.

What is `test` in GAML?

In GAML, the statement `test` allows the modeler to write a part of code lines to verify if portions of our GAML model are doing exactly what they are expected to do: this is done through the use of several assertions (using `assert` statements). This is

done independently from other parts of the model.

To write a typical GAML unit test, we can follow three steps:

1. Define a set of attributes to use within the test,
2. Write initialization instructions,
3. Write assertions.

The aim of using unit testing is to observe the resulting behavior of some parts of our model. If the observed behavior is consistent with the expectations, the unit test passes, otherwise, it fails, indicating that there is a problem concerning the tested part of the model.

Introduction to assertions

The basis of Unit tests is to check that given pieces of codes provide expected results. To this purpose, the modeler can write some basic tests that should be true: s/he thus asserts that such expression can be evaluated to true using the `assert` statement. Here are some examples of `assert` uses:

```
assert 1 + 1 = 2;
assert isGreater(5, 6) = false;
assert rnd(1.0) <= 1.0;
```

With the above statements, the modeler states the `1+1` is equal to `2`, `isGreater(5, 6)` is false (given the fact that `isGreater` is an action defined in a species) and `rnd(1.0)` always returns a value below 1.0.

`assert` can be used in any behavior statement (as an example in a `reflex`, a `state` or in a `test`). Note that, if they are written outside of a `test` and that the test is not fulfilled, then an exception is thrown during their execution.

As an example, the following model throws the exception: `Assert failed 3>4` (as obviously 3 is not greater than 4 and that the GAML `>` operator is properly implemented on this case).

```
model NewModel

global {
    init {
        assert 3 > 4;
    }
}

experiment NewModel type: gui {}
```

To be able to have a dashboard of the state of your model w.r.t. the unit tests, they need to be written in a `test` and the model launched with an experiment of type `test`.

How to write a GAML test?

A `test` statement can be used in any species (regular species, global or experiment species) everywhere a `reflex` can be used. Its aim is to gather several asserts in one block. If the tests are executed with any kind of experiment but `test`, they will be executed, but nothing is reported. With a `test` experiment, a kind of dashboard will be displayed.

So we will consider that we start by adding an `experiment` with `type` set to `test`. The following code shows an example.

```
experiment MyTest type: test autorun: true {
    ...
```

Let's consider the following GAML code:

```
model TestModel

global {
    init {
        create test_agent number: 1;
    }
}

species test_agent {
    bool isGreater (int p1, int p2) {
        if (p1 >= p2) {
            return true;
        } else {
            return false;
        }
    }

    test testsOK {
        assert isGreater(5, 6) = false;
        assert isGreater(6, 5) = true;
    }

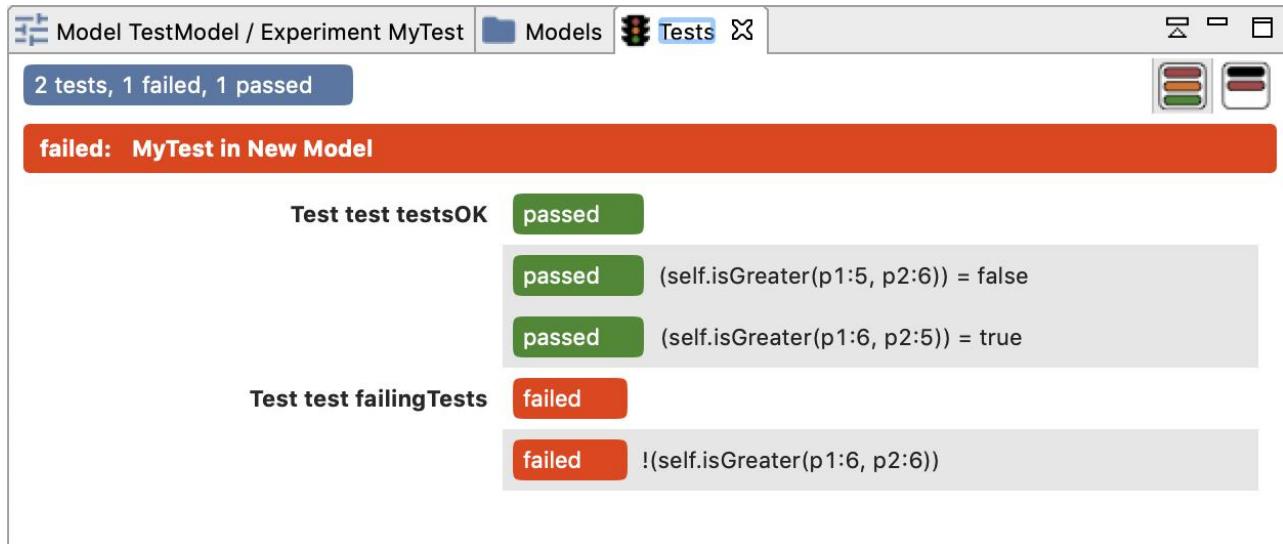
    test failingTests {
        assert ! isGreater(6, 6);
    }
}

experiment MyTest type: test autorun: true { }
```

In this example, the defined action, `isGreater`, returns `true` if a parameter `p1` is greater than a parameter `p2` and `false` if not. So to test it, we declare a unit test using `test` and add inside several `assert` statements. For instance, `assert isGreater(5, 6) = false;` will return `true` if the result of `isGreater(5, 6)` is really `false` and `false` if not. So, if the action `isGreater` is well-defined, it should

`return false`. Considering that "greater" and "greater and equal" should be two different functions, we add a test to check that `isGreater` does not return true in case of equality of its 2 operands. In this case, as the action is not-well implemented the test fails.

The following picture illustrates the GUI dashboard for unit tests, showing for each test and even each assert whether it passes or fails. Clicking on the button will display in the GAML editor the code line.



Use of the `setup` statement

In a species where we want to execute several tests, it is common to want to have the same initial states, in order to prevent the previous tests to have modified the tested object and thus altering the unit test results. To this purpose, we can add the `setup` statement in the species and use it to set the expected initial state of the object to be tested. It will be called before every `test`.

As an example, in the following model, we want to test the operator `translated_by` and `translated_to` on a point. As each of them will modify the point object to be tested, we add a `setup` to reinitialize it.

```

model TestModel

global {
    geometry loc <- {0,0};

    setup {
        loc <- {0,0};
    }

    test translate_to {
        loc <- loc translated_to {10,10};
        loc <- loc translated_to {10,10};
        assert loc.location = {10,10};
    }

    test translated_by {
        loc <- loc translated_by {10,10};
        loc <- loc translated_by {10,10};
        assert loc.location = {20,20};
    }
}

experiment MyTest type: test autorun: true { }

```

The test experiment

It is also possible to write tests in the `experiment`. The main idea is here to totally separate the model and its tests. As an example let's consider the following GAML code, which aims to test several GAML operators, related to the graph datatype:

```

model TestGraphs

global {
    graph the_graph;

```


Version: 1.9.1

Ensure model's reproducibility

There has been a huge effort made in GAMA development in order to ensure the reproducibility of the simulations, i.e. when several simulations of the same models are launched with the same random generator seed and same parameter values, they are supposed to provide the same results.

Nevertheless, GAMA provides several ways to speed up simulations runs, e.g. by making parallel the execution of some agents' behaviors. The use of parallelism may destroy the reproducibility of the simulations. More generally, there are many sources of uncertainty which can break this reproducibility.

How to ensure reproducibility of a model?

If you aim at reproducibility, you need to reduce as much as possible all the sources of uncertainty.

- Set the random number generator seed (explicitly set a value to the model's `seed` global attribute).
- Reduce the parallel execution of agents' behaviors.
 - remove all the explicitly parallel execution, in particular remove / set to false all the `parallel` facets (e.g. in the loop, ask...).
 - Set all of GAMA's settings regarding parallelization to false. You can find

them in the `Preferences` menu, then under the tab `Execution` at the section `Parallelism` to disable them globally, or you can set them to false only in your experiment with the corresponding variables as shown below:

```
experiment 'any exp' {
    init {
        //Make grids schedule their agents in parallel
        gama.pref_parallel_grids <- false;
        //Make experiments run simulations in parallel
        gama.pref_parallel_simulations <- true;
        //Make species schedule their agents in parallel
        gama.pref_parallel_species <- false;
    }
}
```

- Displays are computed independently of the simulation, and in parallel. Limit computation and model modifications in the aspects.
 - Remove any modification of the model in the aspects.
 - Do not use any random operators in the aspects (e.g. `rnd`, `one_of`, `any` ...).
- The use of asynchronous communications (using `network`) with external applications, the use of files (in particular if they are changed externally) can also modify the behavior of simulations
- As a safety measure, you can also set your random number generator to `mersenne` as others may not have been as much tested for reproducibility

Version: 1.9.1

Using extensions

The core GAMA software can be extended with some additional plugins, allowing the model to give more capabilities to agents (negotiation, using fuzzy logic, or Bayesian network) or providing connections to external softwares such as R or Matlab.

For instructions to install these additional plugins, interested readers can refer to the [dedicated page](#).

Selected plugins provided by the GAMA community

The update site located at the address <http://updates.gama-platform.org/experimental> contains new plugins for GAMA mainly developed by the GAMA community ([its Github repository is available here](#)). **As the name of the repository highlights it, these plugins are for most of them still in development, before integration in the kernel of GAMA.** In addition, there are a few eclipse plugins that also work with GAMA.

The following plugins have been tested and are still supported:

- **RJava**: to connect GAMA and R
- **Weka**: to connect GAMA and Weka
- **Matlab**: [to connect GAMA and Matlab](#)
- **Argumentation feature**: to allow agents to reason on an argument system
- **Bayesian Network feature**: to use Bayesian Network to make decision

- **Fuzzy logic:** to use fuzzy logic model to make decision
- **Launchpad:**
- **Camisole:**
- **ImageAnalysis:** to add image processing algorithms to gama
- **Mike and Hecras:**
- **MPI:**
- **QRCode:** to add primitives to encode/decode QRCodes in gaml
- **Switch project:**
- **Uml Generator:** to be able to generate uml from gaml models inside the GAMA IDE
- **Unity:** to connect GAMA to Unity
- **VR:**
- **Gaming:** to add more interactive types of displays
- **Remote.Gui:** to allow exposing some model parameters in order to interact with external application through a network communication
- **ifcfile:** to add support for ifc files in gaml
- **Netcdf:** to add support for the NetCDF file format in gaml
- **Webcam:** to add webcam handling primitives in gaml
- **Graphical editor:** to edit gaml models with graphical blocks instead of code
- **Markdown documentation:** to add the possibility to generate the markdown documentation of a model
- **Easy shell:** to add the eclipse [Easy shell](#) plugin to the GAMA IDE
- **git client:** to add the eclipse git client into the GAMA IDE

RJava plugin

This plugin allows the modeler to launch some computation on the [R](#) software. To this purpose, [R](#) should be installed on your computer and [GAMA](#) should be properly

configured.

This possible connection to R opens thus the possibility for the modeler to use all the statistical functions and libraries developed in this tool of reference. In addition, R scripts defined by the modeler can also be used directly from their GAMA model.

Toward participative simulations with `Remote.Gui` and `Gaming` plugins

There are more and more applications of GAMA for participative simulations ([LittoSim](#), [MarakAir](#), [HoanKiemAir](#)...). There was thus a need for new features to improve the possible interactions with simulations and the definition of the Graphical User Interface. The two plugins `Remote.Gui` and `Gaming` (available in the "Participative simulation" category) attempts to fill this need.

- `Remote.Gui` allows exposing some model parameters, in order that they can be modified through [a network](#). This allows, for example, to develop a remote application (e.g. Android application) to control the parameters' values during the simulation.
- `Gaming` allows the modeler to define displays that are much more interactive. This is used to define serious games in which the users can have a wide range of possible interactions with the simulation.

`Weka` and `Matlab` plugins

Similarly to `RJava`, `Matlab` and `weka` plugins allow the modeler to run computations on the `Matlab` and `Weka` software, taking advantages of all the possibilities of these softwares and of scripts defined by themselves.

Notice that the `Matlab` plugin requires MATLAB 2019a to be installed and activated on your computer.

The graphical editor

The graphical editor allows to create or edit existing GAMA models using only graphical elements, in a similar way to the [scratch programming language](#). You can find a complete overview of the plugin [here](#).

The Git client

This plugin gives you the possibility to have the same git integration in GAMA than in eclipse, with dedicated views and contextual menus directly in the IDE. For more information you can go to it's [dedicated documentation](#).

Version: 1.9.1

Calling R from GAMA models

Introduction

The R language is a powerful tool for statistical computing and graphics, and its community is very large in the world (See the [website](#)). Adding a support for the R language is one of our strong endeavors to accelerate many statistical and data mining tools integration into the GAMA platform.

Installing R and rJava

Install R on your computer

Please refer to the [R official website](#), or to [RStudio](#) if you want in addition a nice IDE.

install the rJava library in R

In the R (RStudio) console, write:

```
install.packages("rJava")
```

to install the library. To check that the install is correct, you load the library using `library(rJava)` (in the R console). If no error message appears, it means the installation is correct.

In case of trouble

For MacOSX

in recent versions you should first write in a terminal:

```
R CMD javareconf  
sudo ln -f -s $(/usr/libexec/java_home)/jre/lib/server/libjvm.dylib /usr/local/lib
```

For Linux

make sure you have the `default-jdk` and `default-jre` packages installed and then execute the command `sudo R CMD javareconf`

For Windows

make sure you have a `JAVA_HOME` and a `CLASSPATH` environment variable setup, if not you need to create and set them, for example:

```
JAVA_HOME="C:\Program Files\Java\OpenJDK17\  
CLASSPATH="C:\Program Files\Java\OpenJDK17\bin\"
```

Set the environment variable `R_HOME`

On Windows

set the environment variables as follows. `R_HOME` is the root directory where we can find the `library` folder in your `R` installation, so it should look like this:

```
R_HOME="C:\Program Files\R\R-4.2.2\"
```

`R_PATH` should point to the folder containing your `R` interpreter, the variable should be set with something similar to this (adapting with your R version and R installation path):

```
R_PATH="C:\Program Files\R\R-4.2.0\bin\x64"
```

On Linux

By default it should be `/usr/lib/R`, you can thus just append the line `R_HOME=/usr/lib/R` to your `/etc/environment` file and reboot your computer

On macOS

You need to create (or update) the file `environment.plist` in the folder: `~/Library/LaunchAgents/` (for the current user, note that this folder is a hidden folder) or in `/Library/LaunchAgents/` (for all users) It should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>my.startup</string>
  <key>ProgramArguments</key>
  <array>
    <string>sh</string>
    <string>-c</string>
    <string> launchctl setenv R_HOME /Library/Frameworks/R.framework/Resources/ </string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

Recommended

If the rJava library doesn't appear in the R library directory, copy the installed rJava library from where it was installed (with `install.packages("rJava")`) to the `library` folder in your `R_HOME`.

Updating the Path variable (Windows only)

In addition, on Windows you also **need** to add to your `Path` environment variable the path to your `R` binaries, by default located in `C:\Program Files\R\R-4.2.2\bin\x64` for `R-4.2.2 64bits`. The `Path` variable is a variable already created by Windows, so you just have to edit it to **add** a new path, no need to delete anything.

Configuration in GAMA

Linking the R connector

From GAMA 1.9.0, you need to specify the path to the R connector library in the GAMA launching arguments. To this purpose, you need to add to either:

1. the `GAMA.ini` file if you use the release version of GAMA
2. **or** the launching configuration (if you use the source code version) the following line:
(replace `PATH_TO_R` by the path to R, i.e. the value in `$R_HOME`):

- **on macOS:** `-Djava.library.path=PATH_TO_R/library/rJava/jri/rlibjri.jnilib`
- **on Windows:** `-Djava.library.path=PATH_TO_R\library\rJava\jri\`
- **on Linux:** `-Djava.library.path=PATH_TO_JRI`

As an example, under macOS, you need to add:

```
-Djava.library.path=/Library/Frameworks/R.framework/Resources/library/rJava/jri/
```

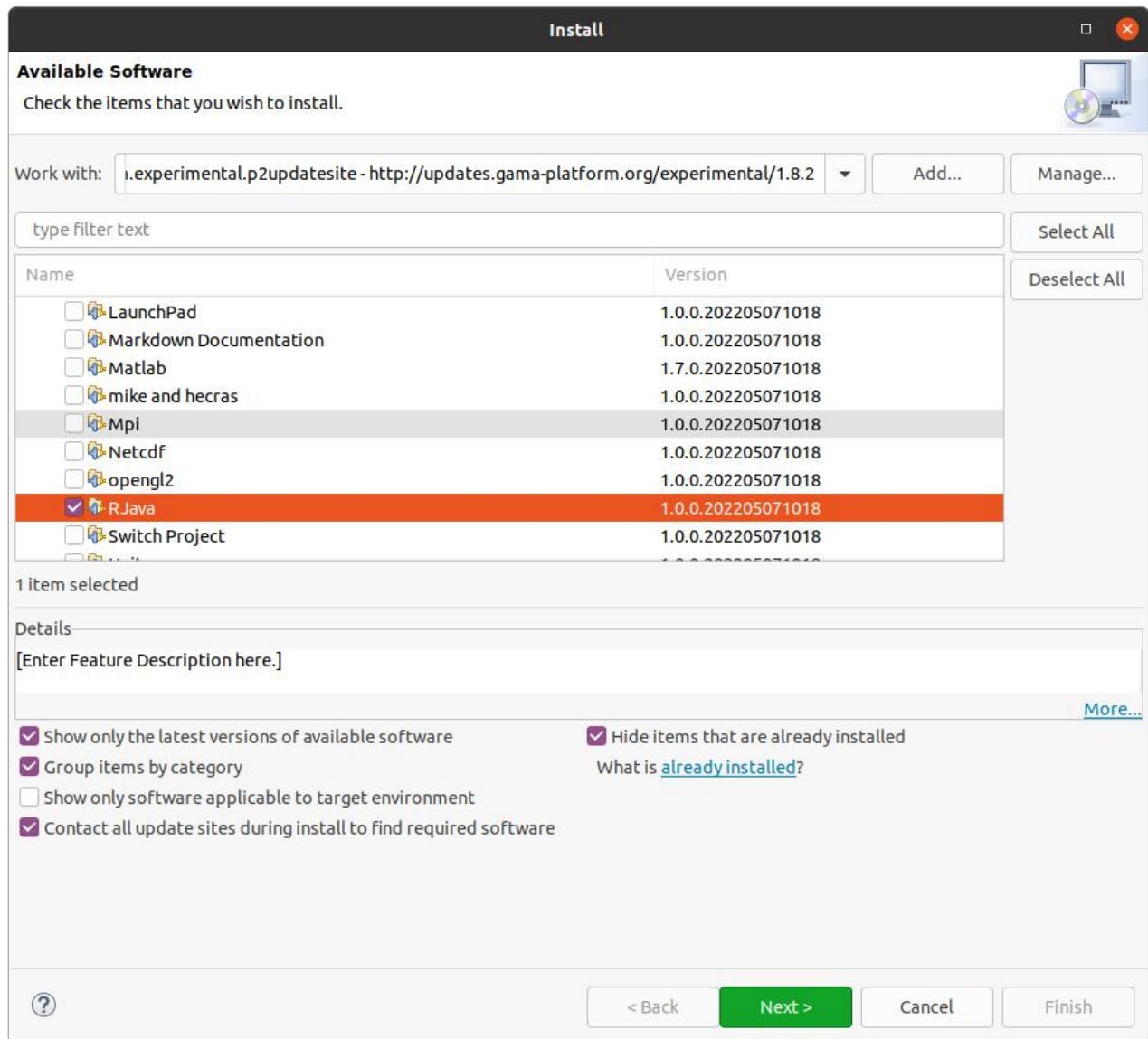
On Windows and Linux, the jri library could be in a different location than the `R_HOME`, for example on Linux by default it would be in:

```
-Djava.library.path=/home/user_name/R/x86_64-pc-linux-gnu-library/3.6/rJava/jri/
```

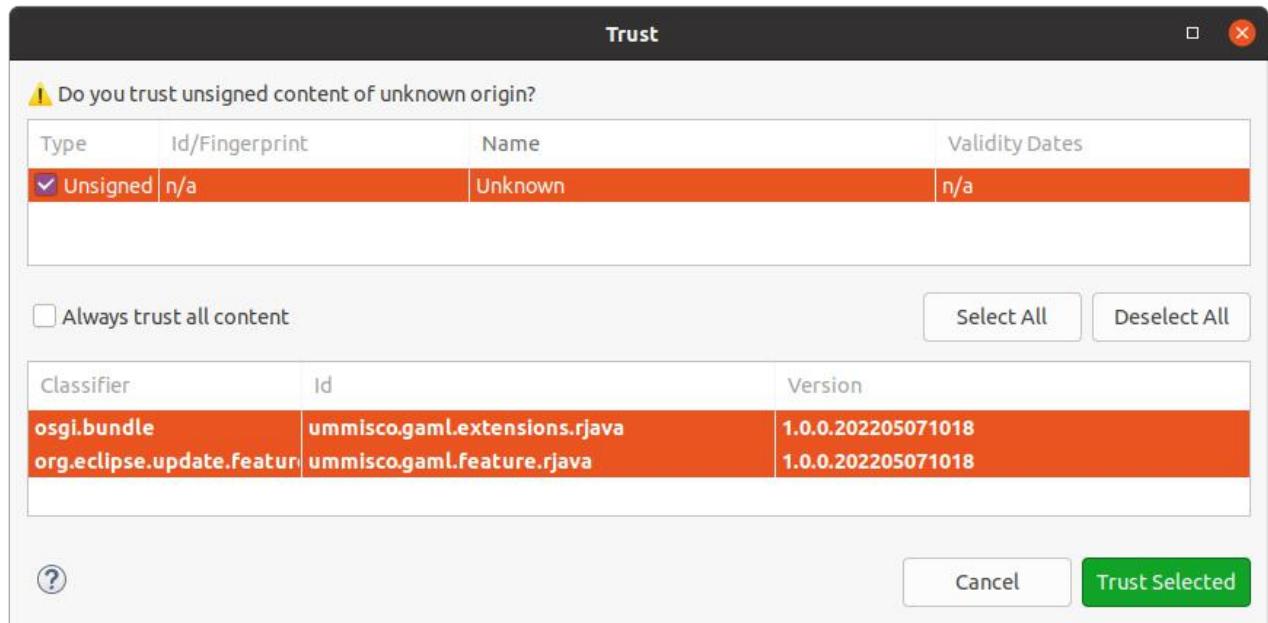
On Windows it can be located in the user's `AppData\local` or in `Documents\R`.

Installing the R plugin

Next you need to install the R plugin from Gama. To do it, select "Install new plugins..." in the "Help" menu of Gama. In the `Work with` drop down select the repository ending with "experimental/" followed by your Gama version. Once done, you need to select the plugin `rJava`, click on `next` and then `finish`.



After this, you could be asked to "trust" the plugin, simply select the first line and click on **Trust selected**



Finally, you will be asked to restart Gama, click on [Restart now](#).

For more details, readers can refer to the page dedicated to the [installation of additional plugins](#).

Calling R from GAML

Before computation

Any agent aiming at using R for some computation needs to be provided with the [RSkill](#).

Before calling any computation, this agent needs to start a connection with the R software.

As an example, if we want that the [global](#) agent can use R, we need to have the following minimal model:

```
global skills: [RSkill] {
    init {
        do startR;
    }
}
```

Computation

Evaluate an R expression

The `R_eval` operator can be used to evaluate any R expression. It can also be used to initialize a variable or call any function. It can return any data type (depending on the R output). As in an R session, the various evaluations are dependent on the previous ones.

Example:

```
global skills: [RSkill] {  
  
    init{  
        do startR;  
  
        write R_eval("x<-1");  
        write R_eval("rnorm(50,0,5)");  
    }  
  
}
```

Evaluate an R script

To evaluate an R script, stored in a (text) file, open the file and execute each of its lines.

```
global skills:[RSkill]{  
    file Rcode <- text_file("../includes/rScript.txt");  
    init{  
        do startR;  
        // Loop that takes each line of the R script and execute it.  
        loop s over: Rcode.contents{  
            unknown a <- R_eval(s);  
            write "R>"+s;  
            write a;  
        }  
    }  
}
```

Convert GAMA object to R object

To use GAMA complex objects into R functions, we need to transform them using the `to_R_data` operator: it transforms any GAMA object into a R object.

```
global skills:[RSkill] {
    init {
        do startR();

        string s2 <- "s2";
        list<int> numlist <- [1,2,3,4];
        write R_eval("numlist = " + to_R_data(numlist));
    }
}
```

Convert a species to a dataframe

Dataframe is a powerful R data type allowing to ease data manipulation... Dataframe can of course be defined at hand using R commands. But GAML provides the `to_R_dataframe` operator to directly transform a species of agents into a dataframe for future analysis.

```
global skills: [RSkill] {

    init{
        do startR();

        create people number: 10;

        do R_eval("df<- " + to_R_dataframe(people));
        write R_eval("df");
        write R_eval("df$flipCoin");
    }
}

species people {
    bool flipCoin <- flip(0.5);
}
```

Troubleshooting

It is possible that after installing everything, Gama works normally but crashes every time you try to use the skill `Rskill` without any error message. If that's the case, the problem is certainly that Gama is unable to load the `jri` library or its dependencies (other R packages). Make sure that the path you wrote in the `.ini` file is correct and that every environment variable is set with proper values.

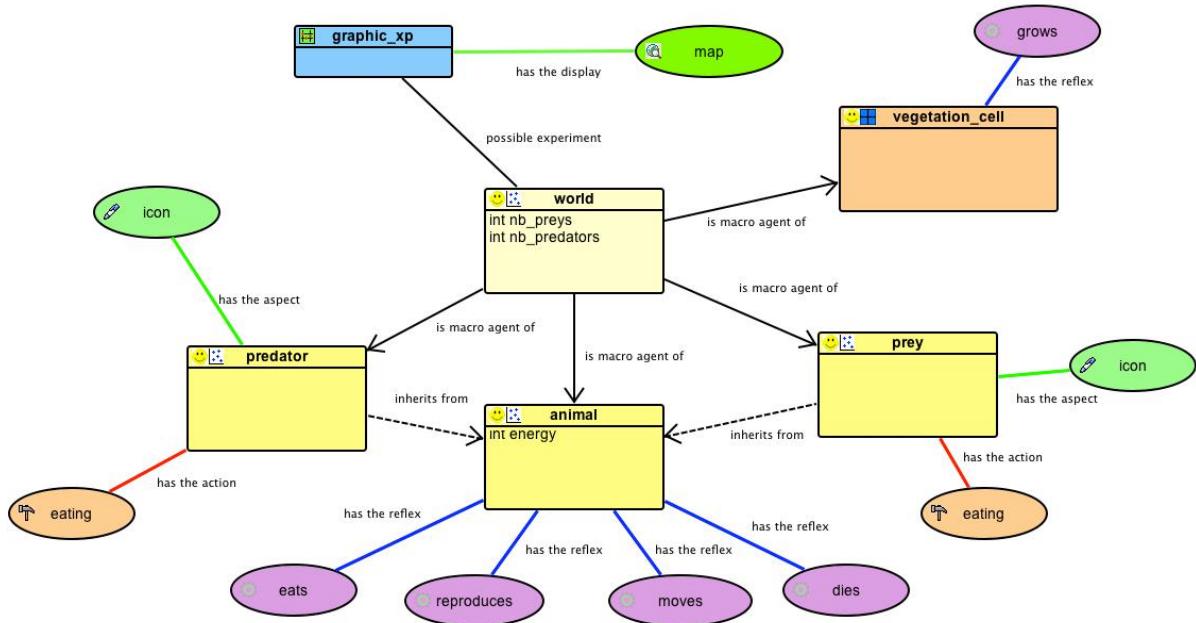
Also on windows, check that the `Path` variable contains the path to your `R` installation.

If after checking everything the problem is still there, you can try copying the `.dll` files at the `R_PATH` location and the `jri.dll` and paste them into your `JAVA_PATH` directory (the `bin` folder of your jdk).

Version: 1.9.1

The Graphical Editor

The graphical editor allows defining a GAMA model through a graphical interface (`gadl` files). It is based on the Graphiti Eclipse plugin. It allows as well to produce a graphical model (diagram) from a `gaml` model. A tutorial is available [here](#).



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.

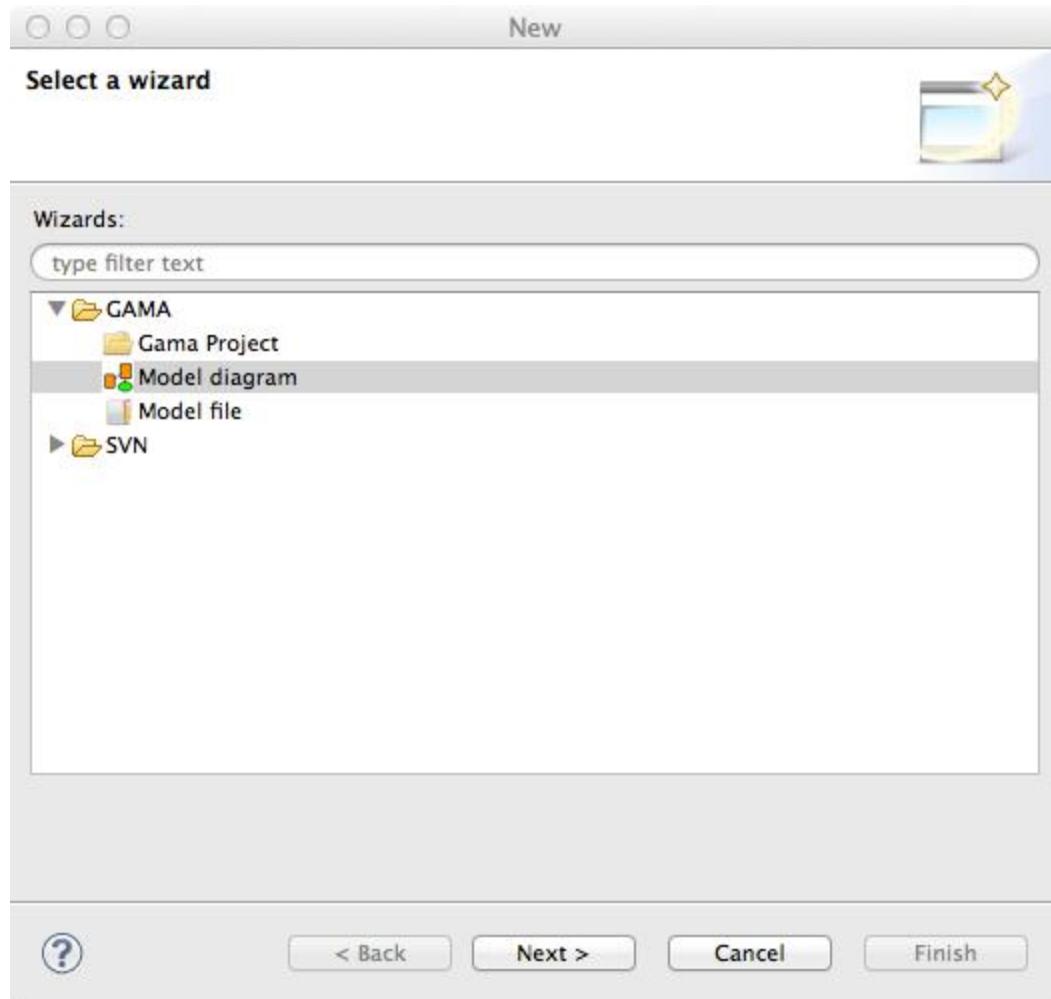
The graphical editor plug-in is called **Graphical_modeling** and is directly available

from the GAMA update site http://updates.gama-platform.org/graphical_modeling/1.9.0

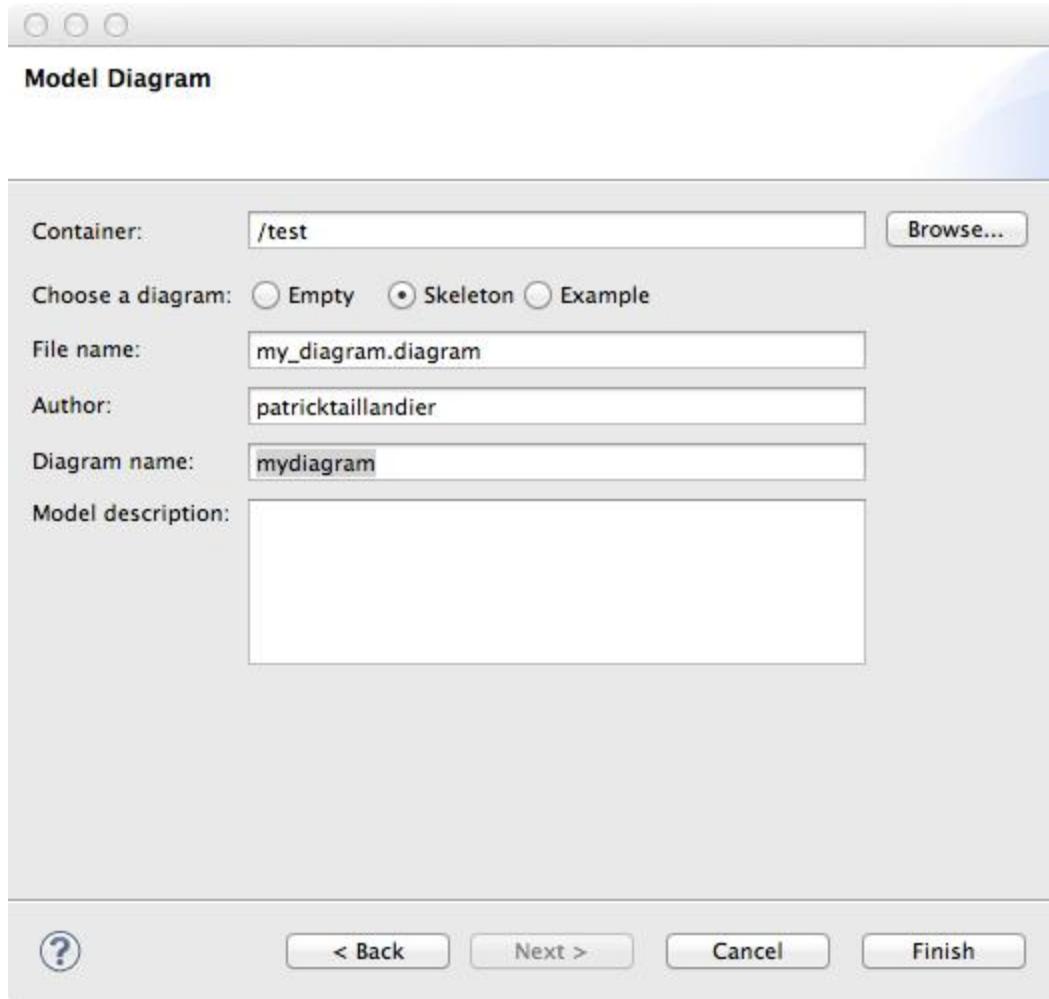
Note that the graphical editor is still under development. Updates of the plug-in will be added to the GAMA website. After installing the plug-in (and periodically), check for updates for this plug-in: in the "Help" menu, choose "Check for Updates" and install the proposed updates for the graphical modeling 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 [the section on model validation](#) for more precise information about these statuses.

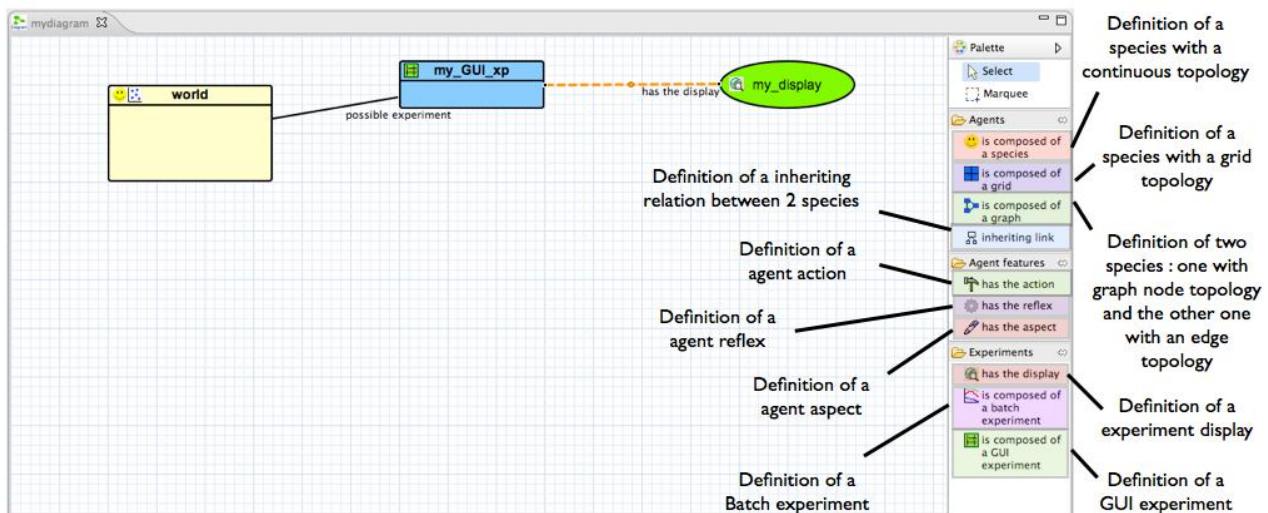
In its initial state, a model is always in the **Functional** state, which means it compiles without problems, but cannot be used to launch experiments. The **InError** state occurs when the file contains errors (syntactic or semantic ones).

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 the modeler 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

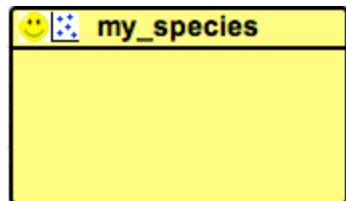
The following figure presents the editing framework:



Features

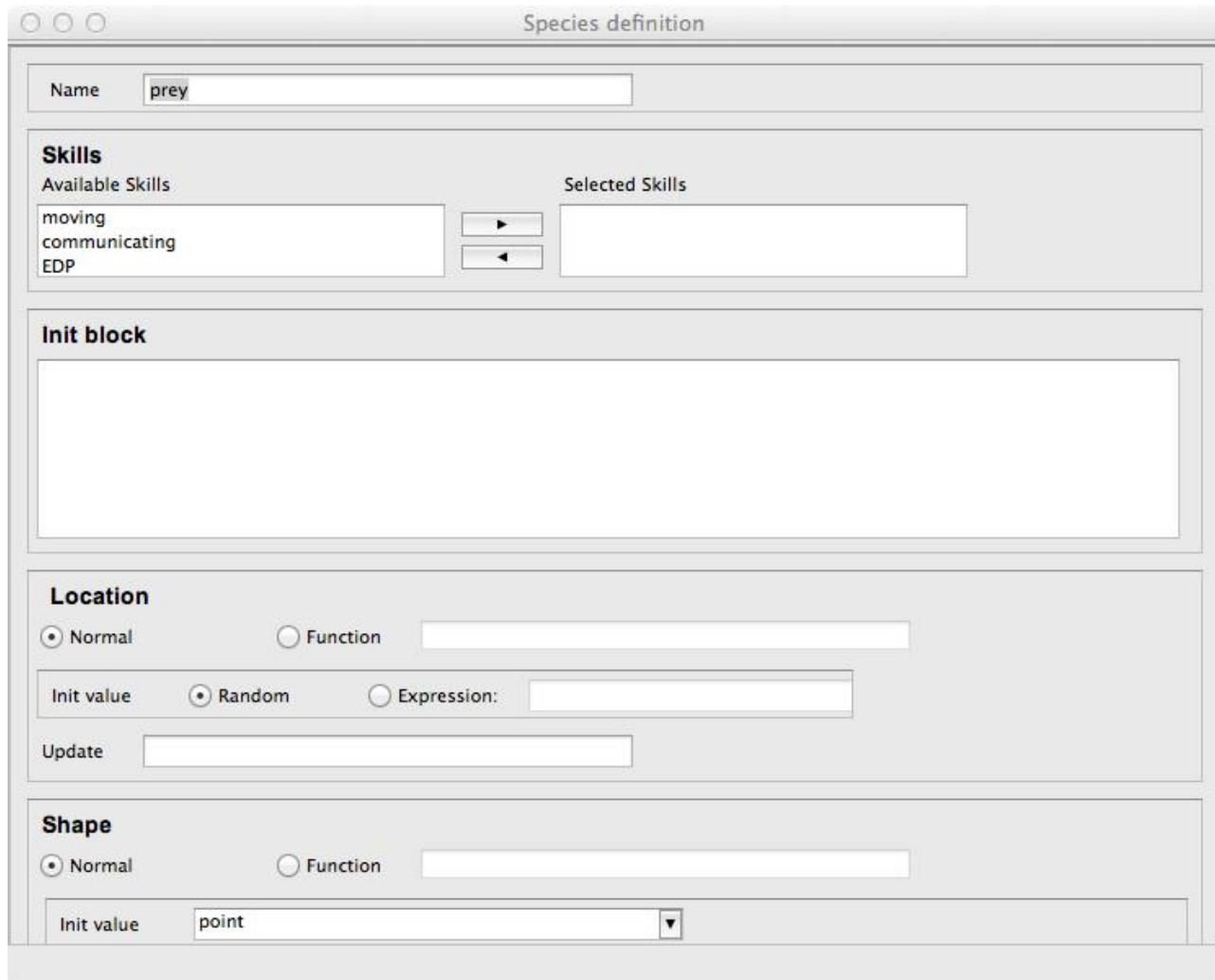
agents

species

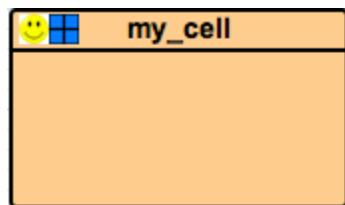


The species feature allows the modeler to define a species with a continuous topology. A species is always a micro-species of another species. The top-level (macro-species of all species) is the world species.

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



grid



The grid feature allows the modeler to define a species with a [grid topology](#). A grid is always a micro-species of another species.

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

- **target:** -

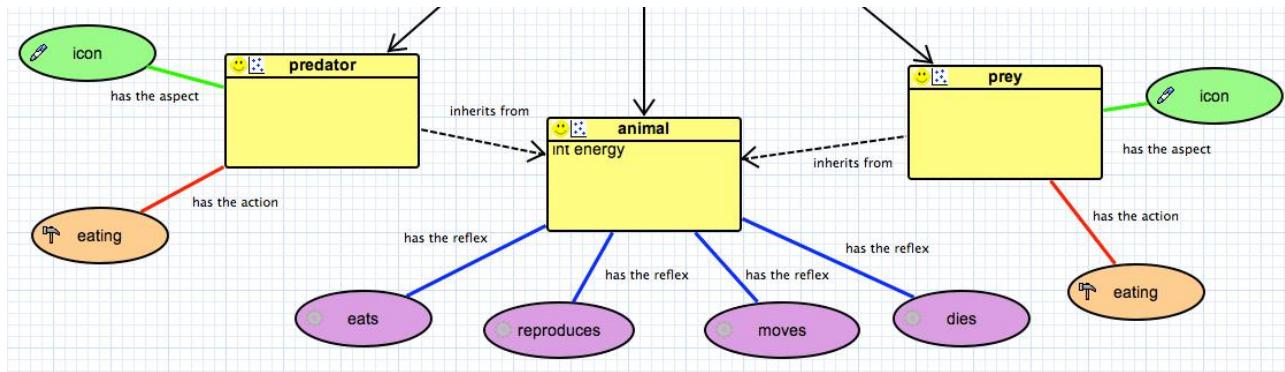
Species definition

Name	my_cell					
Skills						
Available Skills	Selected Skills					
moving communicating EDP	► ◀					
Init block						
<input type="radio"/> Yes <input checked="" type="radio"/> No <input type="radio"/> Expression: <input type="text"/>						
Grid properties						
Neighborhood	4 (square – von Neumann)					
Number of rows	100					
Number of columns	100					
Variables						
Name	Type	init value	update	function	min	max

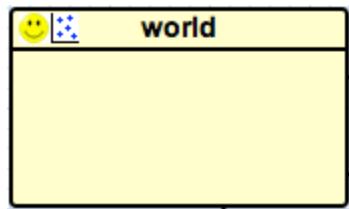
Inheriting link

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

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



world



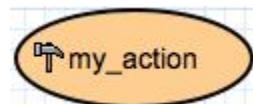
When a model is created, a **world** species is always defined. It represents the global part of the model. The **world** species, which is unique, is the top-level species. All other species are micro-species of the **world** species.

Species definition

Name	world					
Skills						
Available Skills	Selected Skills					
moving communicating EDP	► ◀					
Init block						
<input type="radio"/> Yes <input checked="" type="radio"/> No <input type="radio"/> Expression: []						
Bounds						
Value type	width-height					
Width	100.0					
Height	100.0					
Variables						
Name	Type	init value	update	function	min	max
[]	[]	[]	[]	[]	[]	[]

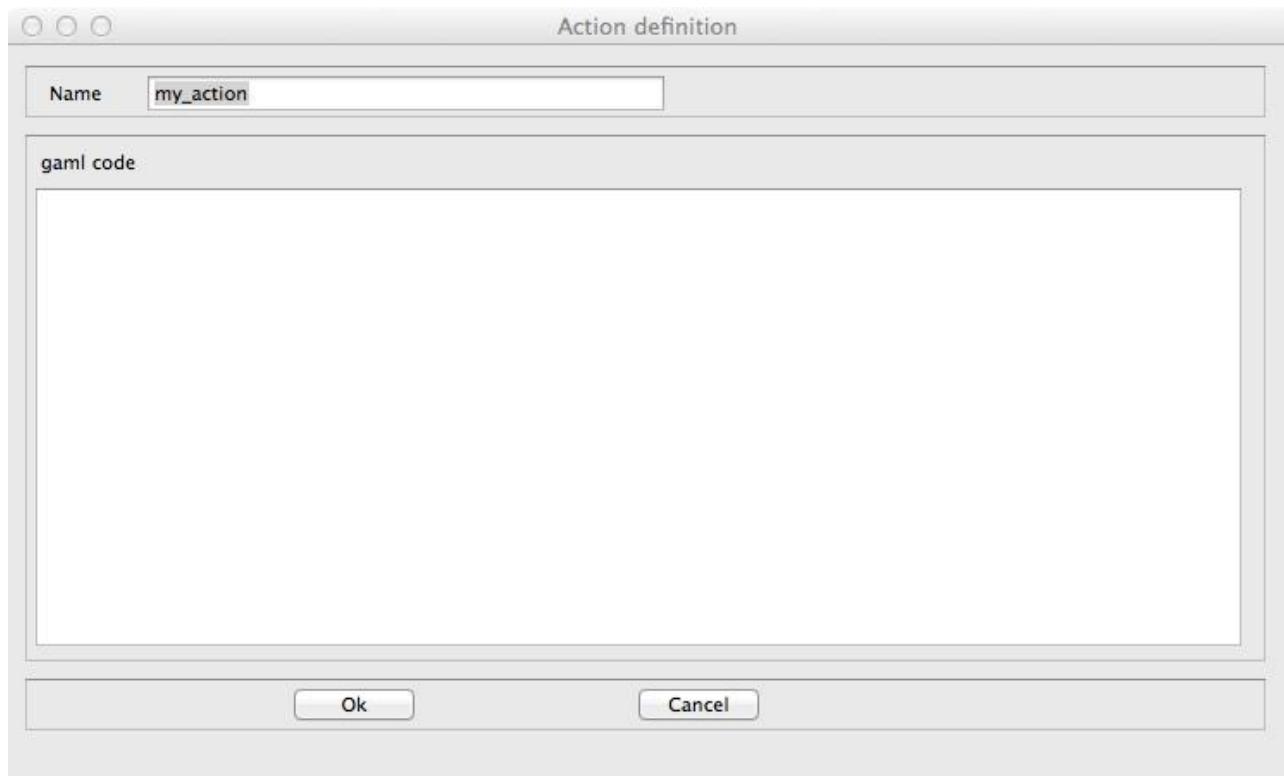
agent features

action

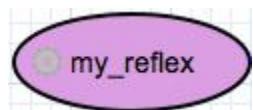


The action feature allows the modeler to define an action for a species.

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

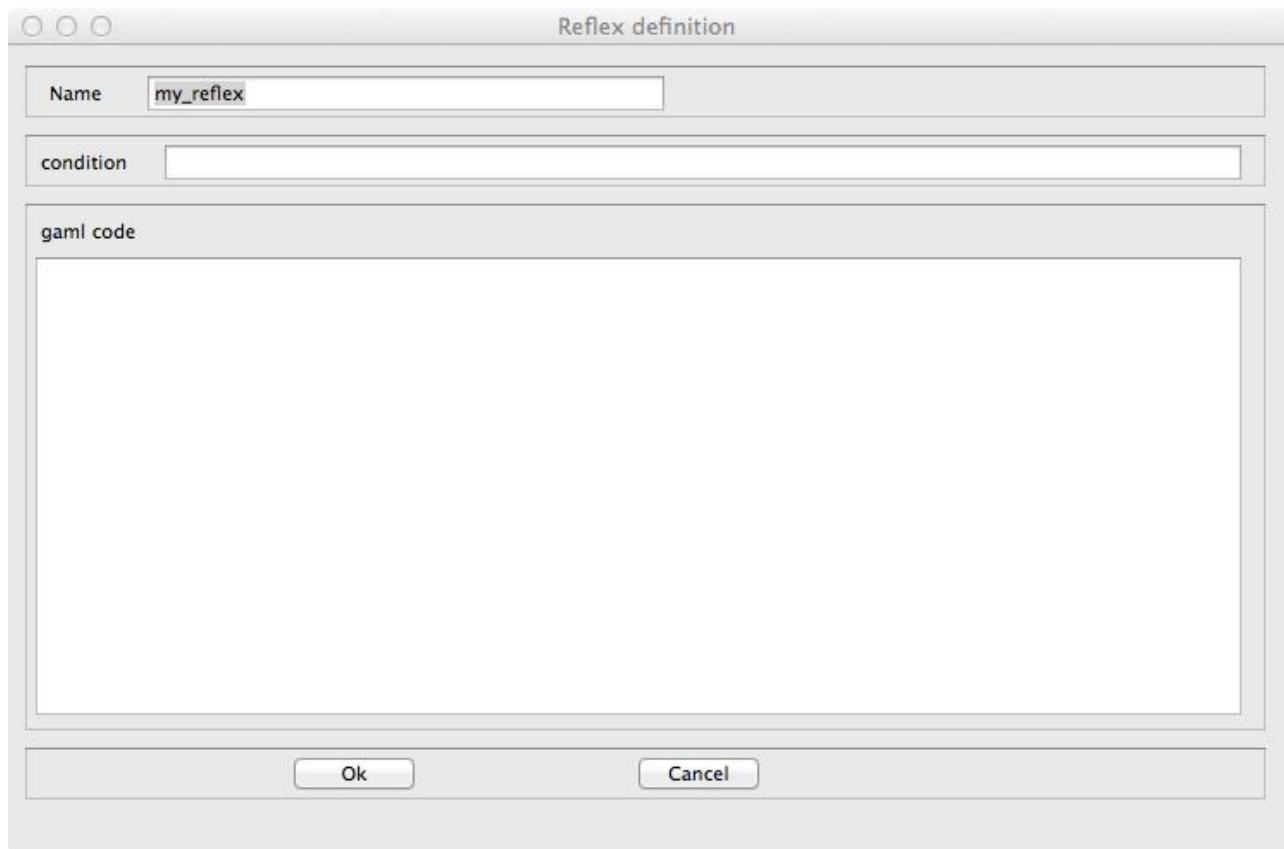


reflex



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

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



aspect



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

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

Aspect definition

Name

Layers

Layer1
Layer2

Add Edit Remove ▲ ▼

Ok Cancel

Edit Aspect Layer

Name

Shape ▾

Radius

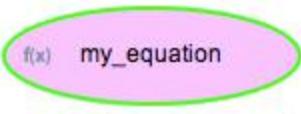
Color Constant Expression:

Empty

Rotate

OK Cancel

equation

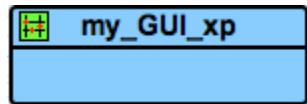


The equation feature allows the modeler to define an equation for a species.

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

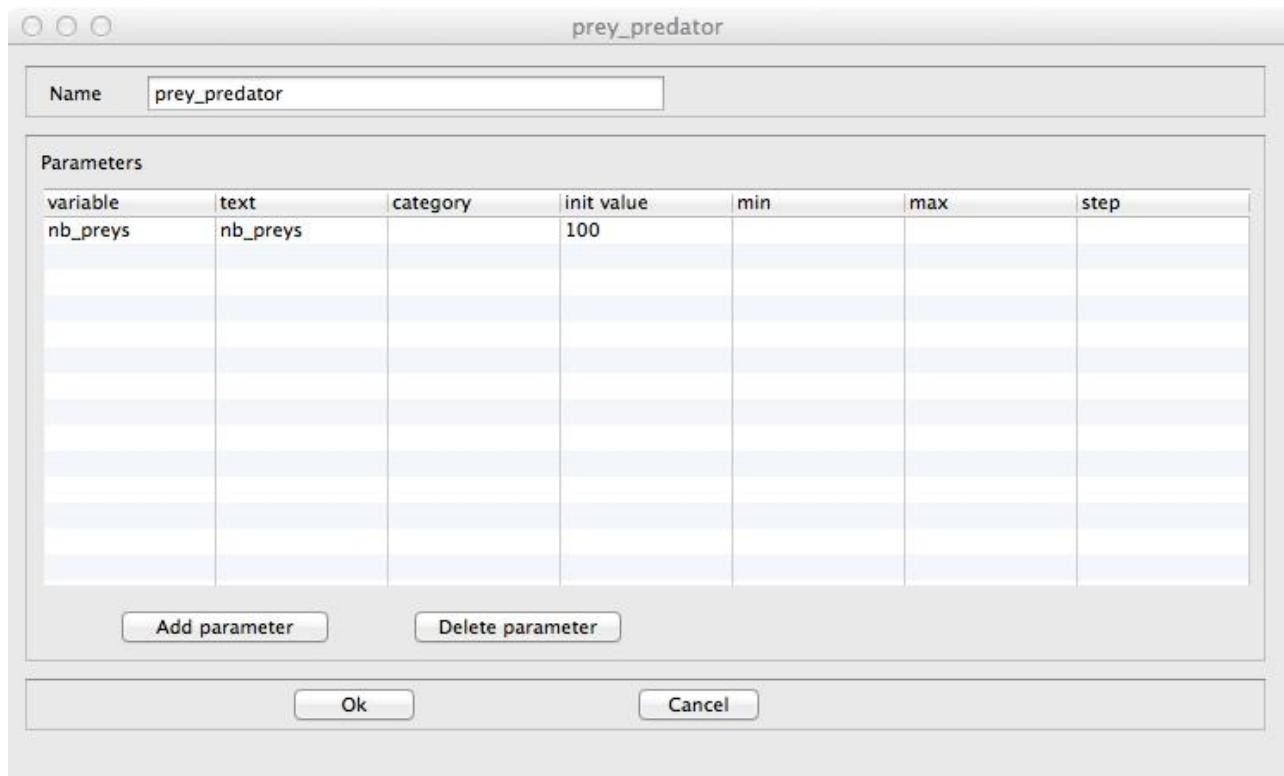
experiment

GUI experiment



The GUI Experiment feature allows the modeler to define a GUI experiment.

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

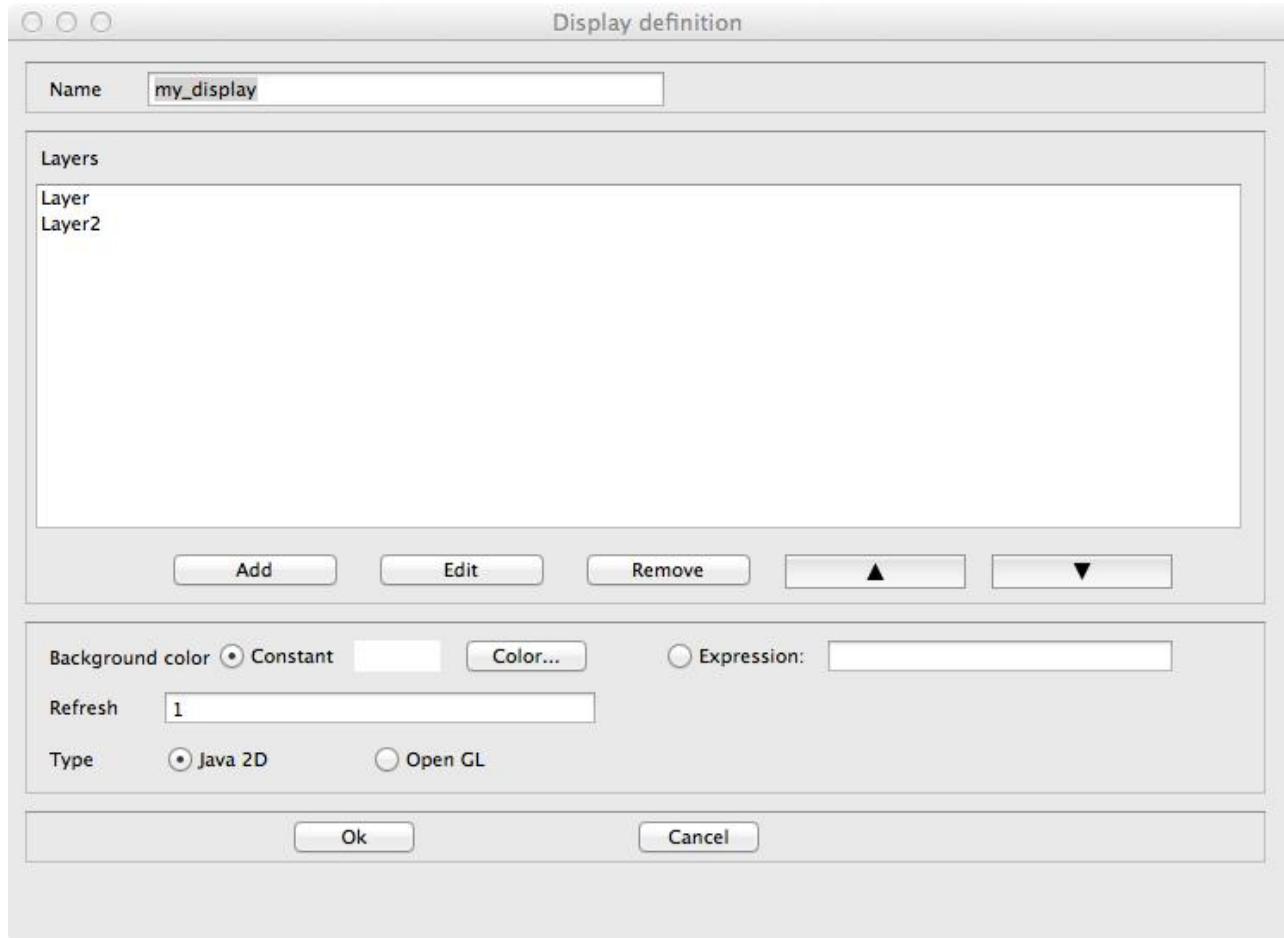


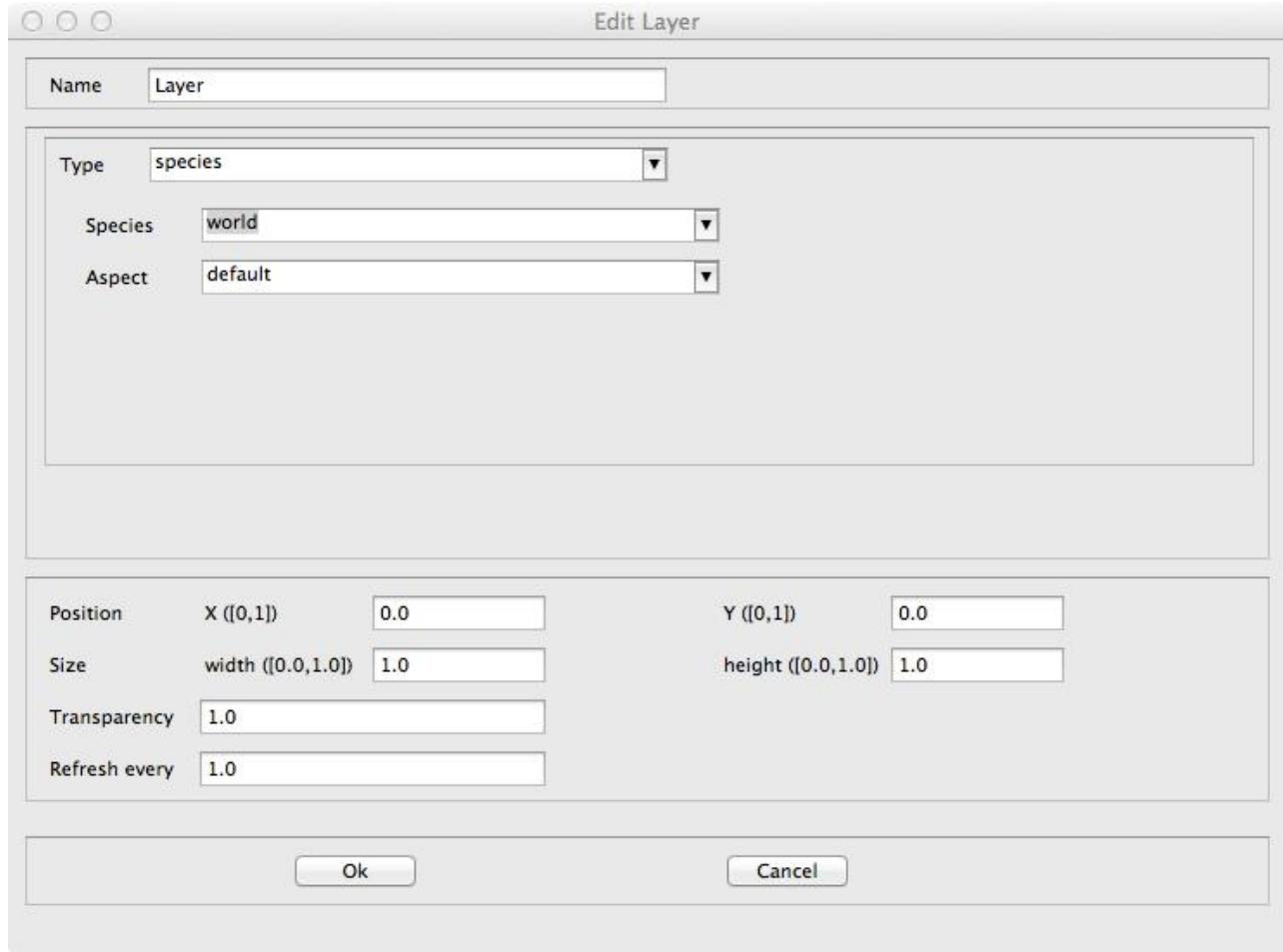
display



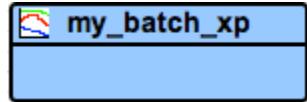
The display feature allows the modeler to define a display.

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





batch experiment

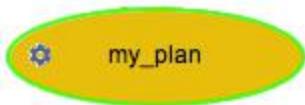


The Batch Experiment feature allows the modeler to define a Batch experiment.

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

BDI Architecture

Plan



The Plan feature allows the modeler to define a plan for a BDI species, i.e. a sequence of statements that will be executed in order to fulfill a particular intention.

- **source:** a species with a BDI architecture
- **target:** - s

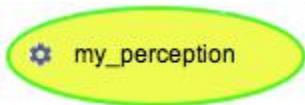
Rule



The Rule feature allows the modeler to define a rule for a BDI species, i.e. a function executed at each iteration to infer new desires or beliefs from the agent's current beliefs and desires.

- **source:** a species with a BDI architecture
- **target:** -

Perception

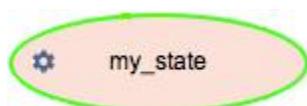


The Perception feature allows the modeler to define a perception for a BDI species, i.e. a function executed at each iteration that updates the agent's Belief base according to the agent perception.

- **source:** a species with a BDI architecture
- **target:** -

Finite State Machine Architecture

State

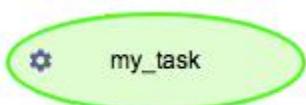


The State feature allows the modeler to define a state for a FSM species, i.e. sequence of statements that will be executed if the agent is in this state (an agent has a unique state at a time).

- **source:** a species with a finite state machine architecture
- **target:** -

Task-based Architecture

Task



The Task feature allows the modeler to define a task for a Tasked-based species, i.e. 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.

- **source:** a species with a task-based architecture
- **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).

Version: 1.9.1

Using Git from GAMA to version and share models

Install the Git client [Tested on the GAMA 1.9.0]

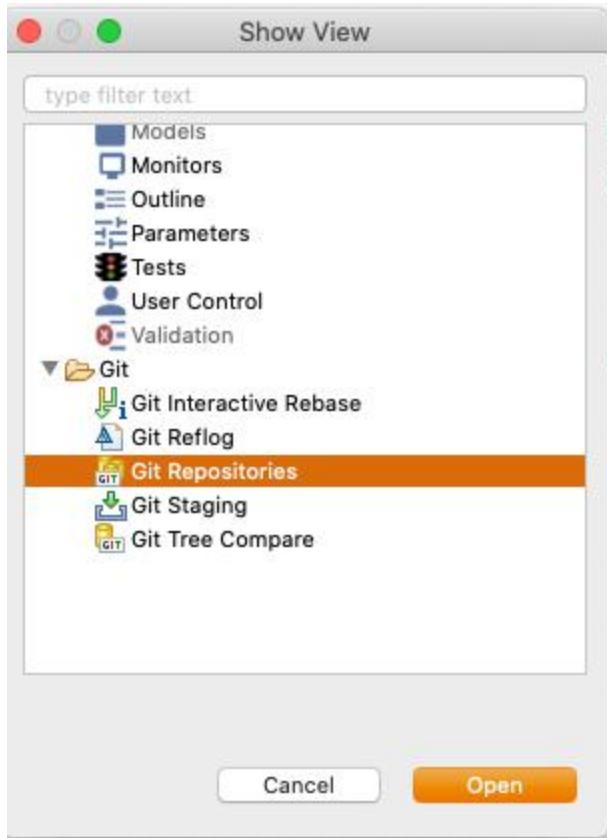
The Git client for GAMA needs to be installed as an external plugin.

1. Help > Install new plugins...
2. Add the following address in the text field "Work with":
`https://download.eclipse.org/egit/updates`. (press Enter key)
3. In the available plugins to install, choose `Git integration for Eclipse` > `Git integration for Eclipse`
4. Click on the Next button and follow the instructions (GAMA will be relaunched).

Open the Git view

To use Git in GAMA select Views -> Other... -> Show View -> Other...

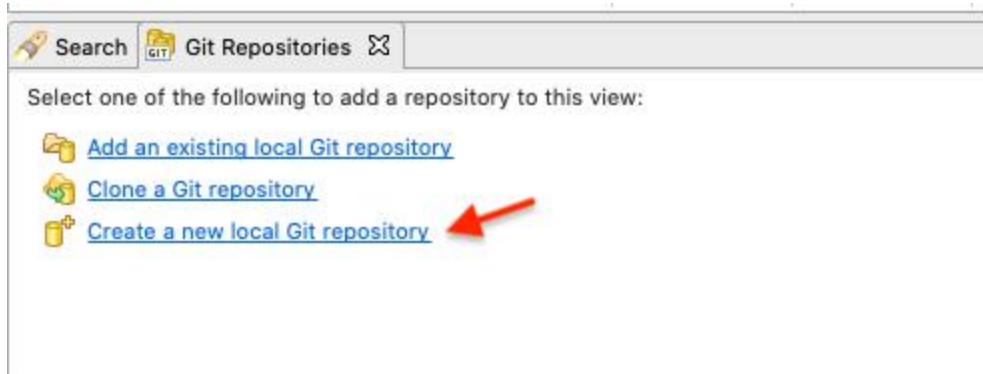
In the Show view window that appears select Git -> Git Repositories and click on *Open*.



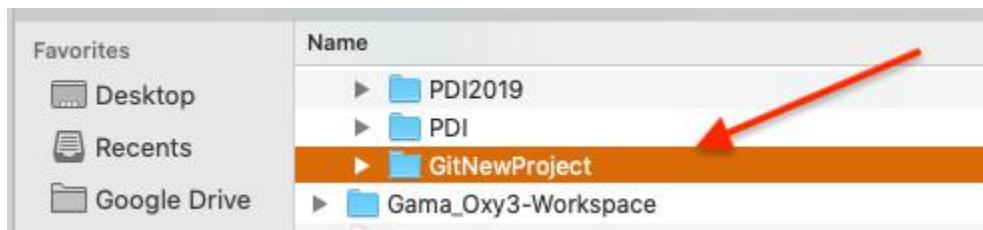
Create a Local Repository

With Git you can easily create local repositories to version your work locally. First, you have to create a GAMA project (e.g **GitNewProject**) that you want to share via your local repository.

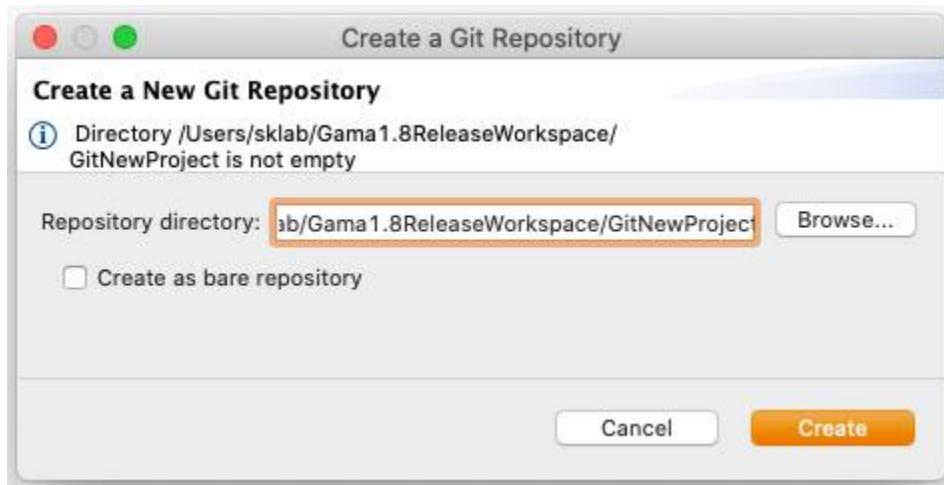
After you have created your GAMA project, go to the Git Repository view and click on *Create a new local Git repository*.



In the following window specify the directory for the new repository (select the folder of the created GAMA project - **GitNewProject** -), through the button Browse...



then hit the Create button.

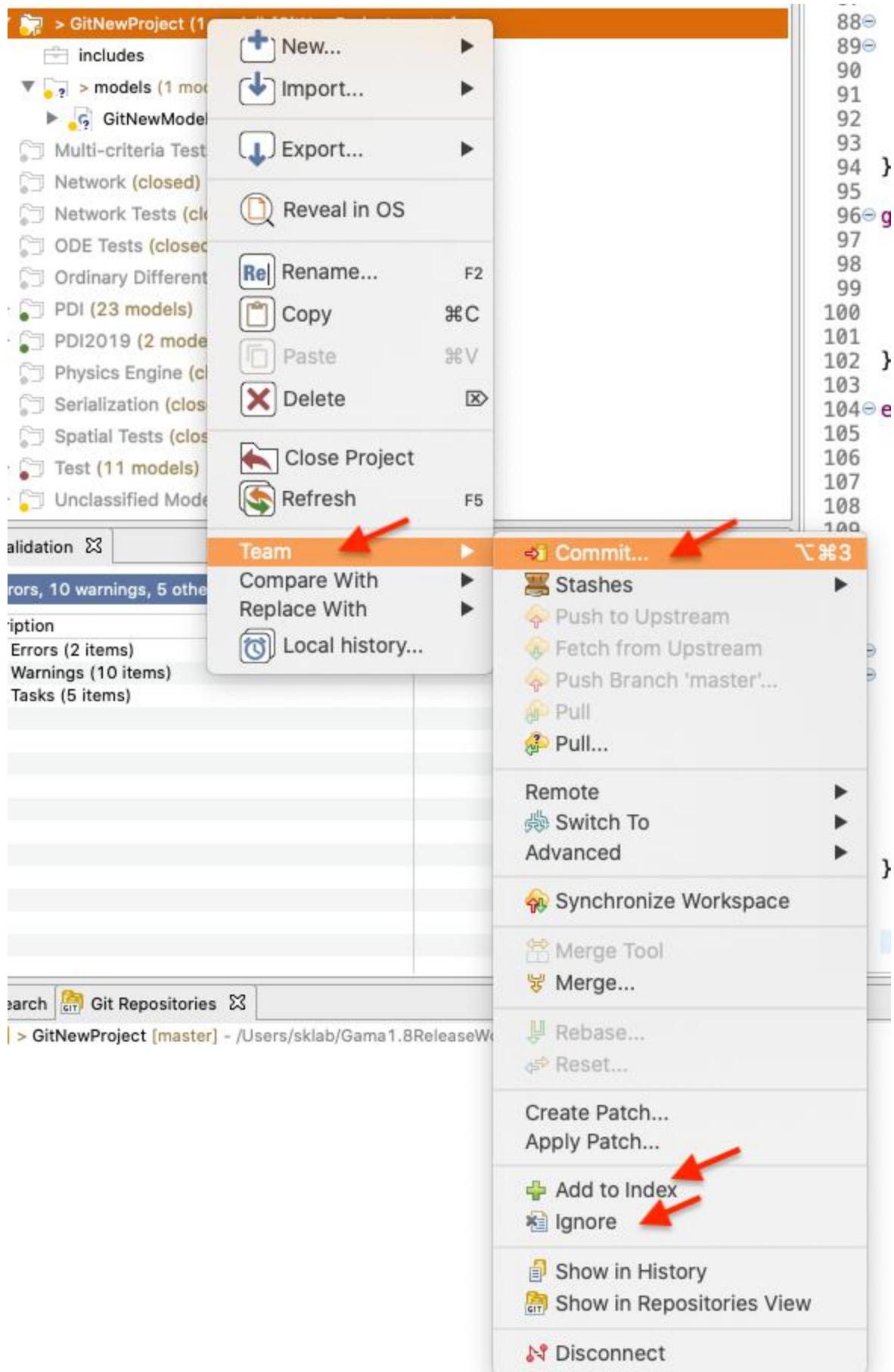


Now your local repository is created, you can add models and files into your GAMA project. As you selected the folder of the new created GAMA Project, the repository will not be empty. So, it will be initialized with all the folders and files of the GAMA

project. Note the changed icons: the project node will have a repository icon, the child nodes will have an icon with a question mark.



Before you can commit the files to your repository, you need to add them. Simply right click the shared project's node and navigate to Team -> Add to Index.



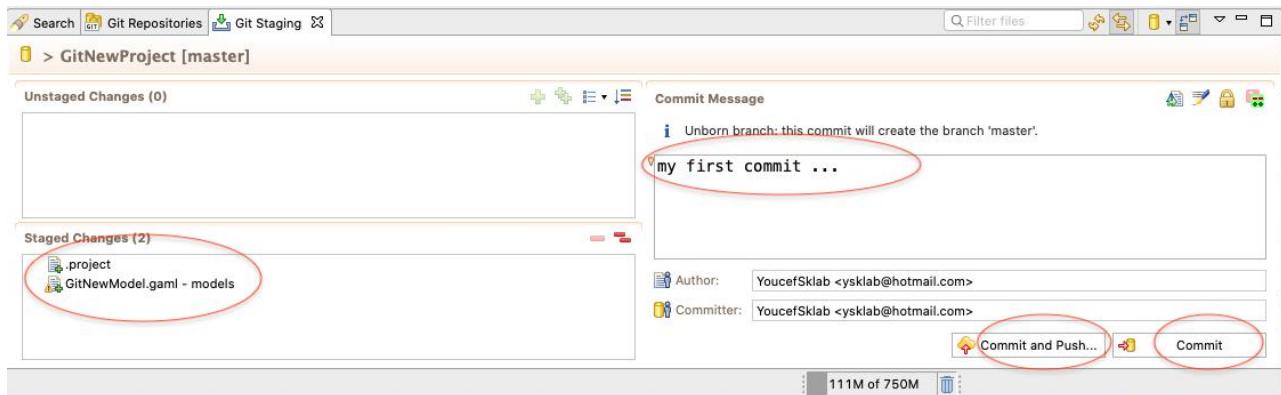
After this operation, the question mark should change to a plus symbol.



To set certain folders or files to be ignored by Git, right click them and select Team -> Ignore. The ignored items will be stored in a file called .gitignore, which you should add to the repository.

Commit

Now you can modify files in your project, save changes made in your workspace to your repository and commit them. You can do commit the project by right clicking the project node and selecting Team -> Commit... from the context menu. In the Commit wizard, all files should be selected automatically. Enter a commit message and hit the Commit button.



If the commit was successful, the plus symbols will have turned into repository icons.



After changing files in your project, a ">" sign will appear right after the icon, telling you the status of these files is dirty. Any parent folder of this file will be marked as dirty as well.



If you want to commit the changes to your repository, right click the project (or the files you want to commit) and select Team -> Commit... . Enter a commit message and click Commit to commit the selected files to your repository.

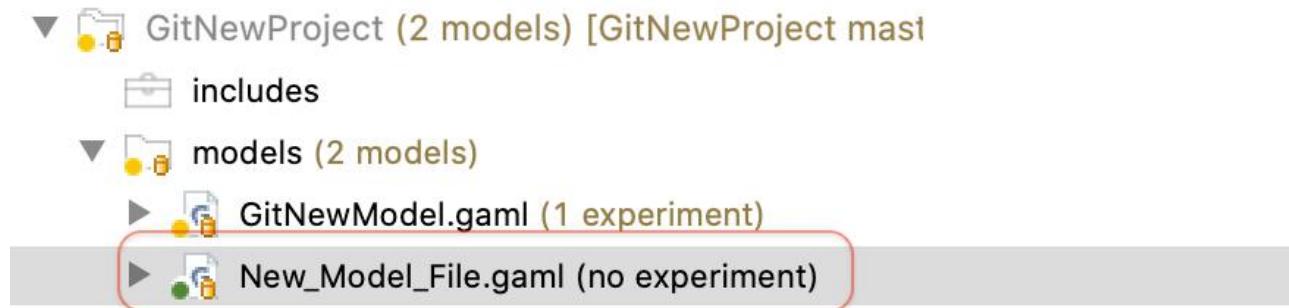
Add Files

To add a new file to the repository, you need to create it in your shared GAMA project first. Then, the new file will appear with a question mark.



Right click it and navigate to Team -> Add to Index. The question mark will turn into a plus symbol and the file will be tracked by Git, but it is not yet committed. In the next

commit, the file will be added to the repository and the plus symbol will turn into a repository icon.



Revert Changes

If you want to revert any changes, there are two options. You can compare each file you want to revert with the HEAD revision (or the index, or the previous version) and undo some or all changes done. Second, you can hard reset your project, causing any changes to be reverted.

Revert via Compare

Right click the file you want to revert and select Compare With -> HEAD Revision. This will open a comparison with the HEAD Revision, highlighting any changes done. You can revert several lines. select the line you want to revert and hit the Copy Current Change from Right to Left button (in the toolbar).

```

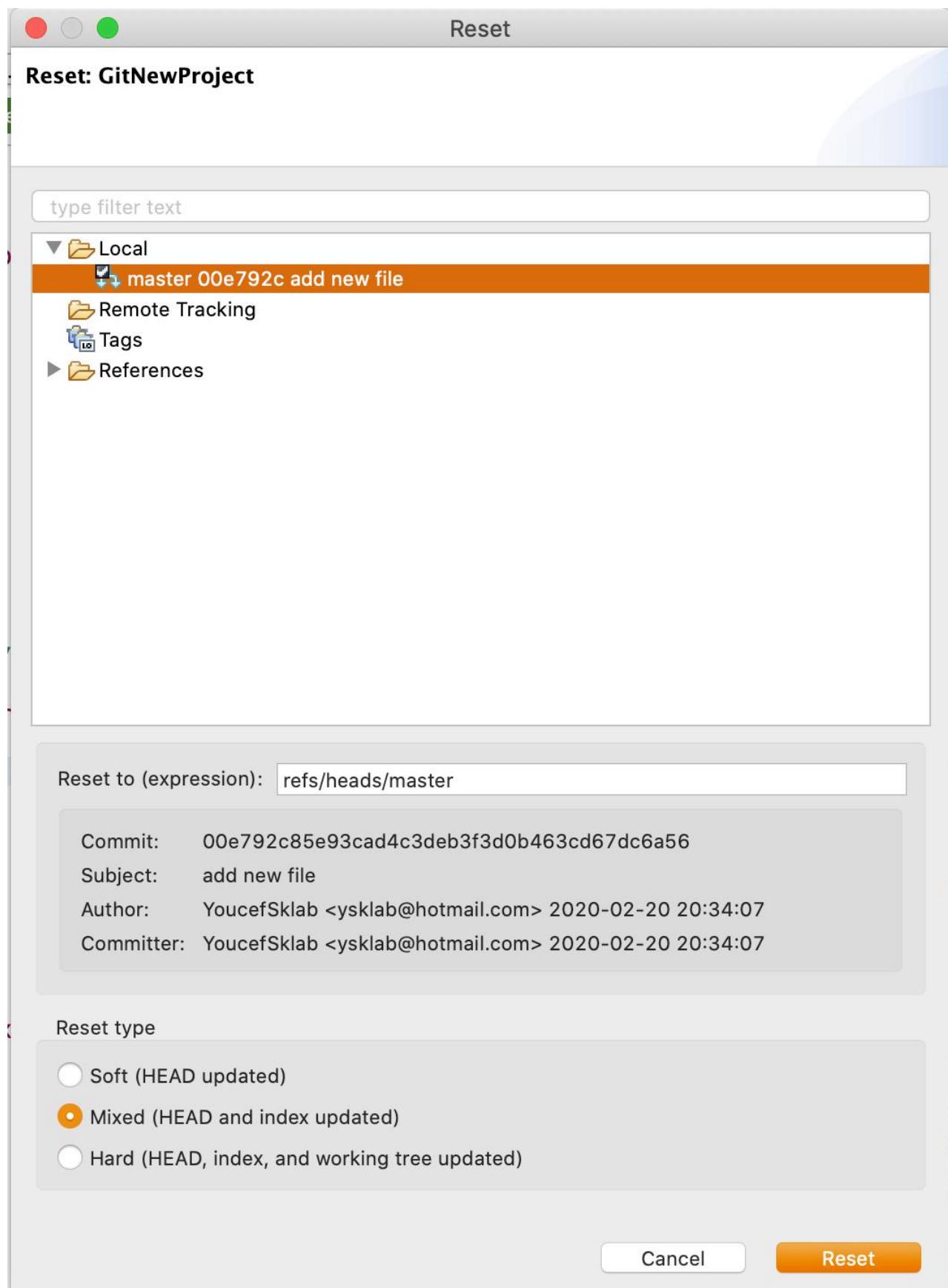
Local: GitNewModel.gaml
91     }
92     energy <- energy + energy_transfert ;
93   }
95 //
96
97 grid vegetation_cell width: 50 height: 50 neighbors: 4 {
98   float value_1 <- 12.0;
99   string var_name <- " ";
100  float maxFood <- 1.0 ;
101  float foodProd <- (rnd(1000) / 1000) * 0.01 ;
102  float food <- (rnd(1000) / 1000) max: maxFood update: food
103  rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 -
104    list<vegetation_cell> neighbours <- {self neighbors_at 2
105  }
106
107
108 experiment prey_predatorExp type: gui {
109   parameter "Nb Preys: " var: nb_preys_init min: 0 max: 10
110

GitNewModel.gaml 33c672f (YoussefSklab)
89   ask one_of (reachable_preys) {
90     do die ;
91   }
92   energy <- energy + energy_transfert ;
93 }
94 //
95 //
96
97 grid vegetation_cell width: 50 height: 50 neighbors: 4 {
98   float maxFood <- 1.0 ;
99   float food <- (rnd(1000) / 1000) * 0.01 ;
100  float food <- (rnd(1000) / 1000) max: maxFood update: food
101  rgb color <- rgb(int(255 * (1 - food)), 255, int(255 *
102    list<vegetation_cell> neighbours <- {self neighbors_at 2
103  }
104
105 experiment prey_predatorExp type: gui {
106   parameter "Nb Preys: " var: nb_preys_init min: 0 max: 10
107   parameter "Prey max energy: " var: prey_max_energy

```

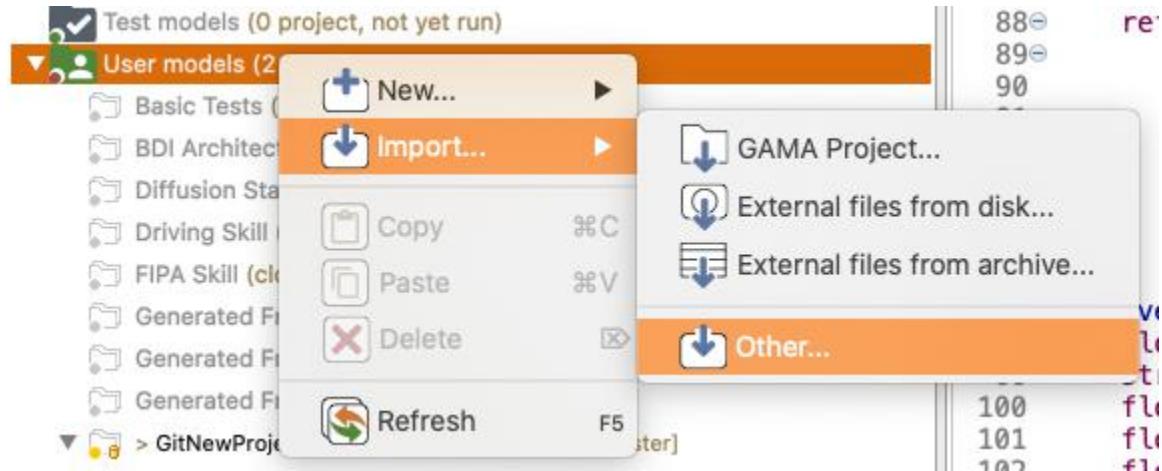
Revert via Reset

To reset all changes made to your project, right click the project node and navigate to Team -> Reset.... Select the branch you want to reset to (if you haven't created any other branches, there will be just one). Click the reset button. All changes will be reset to this branch's last commit. Be careful with this option as all last changes in your Gama Project will be lost.

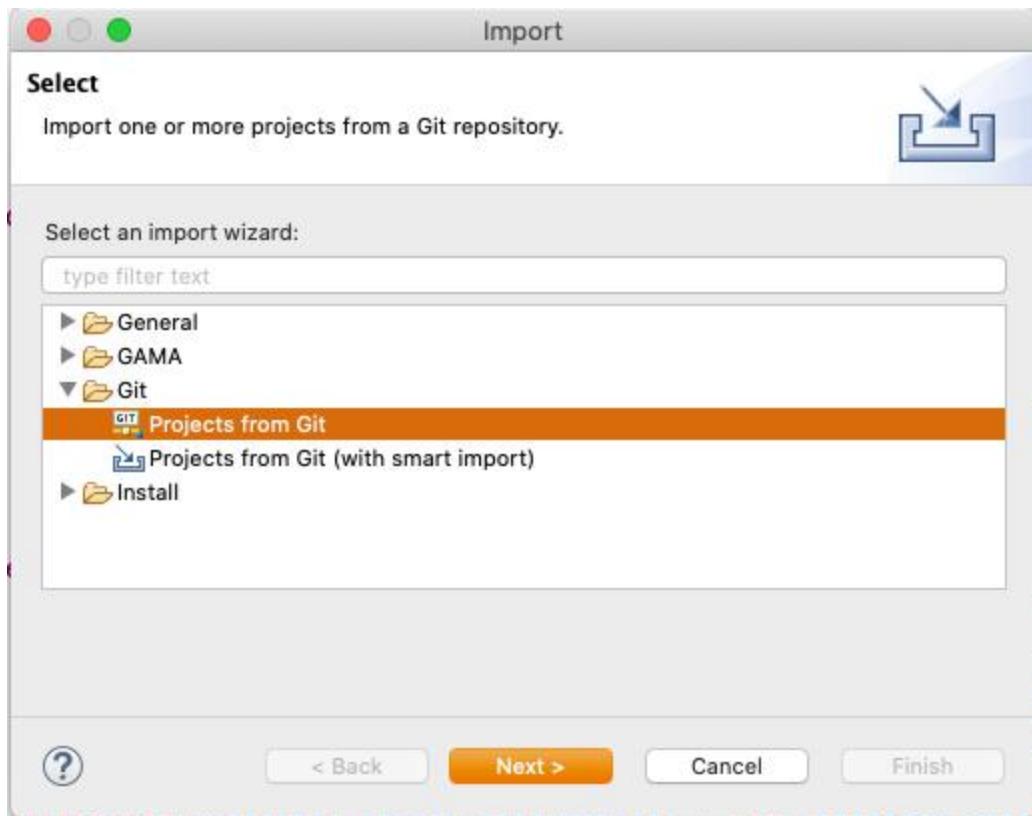


Clone Repositories

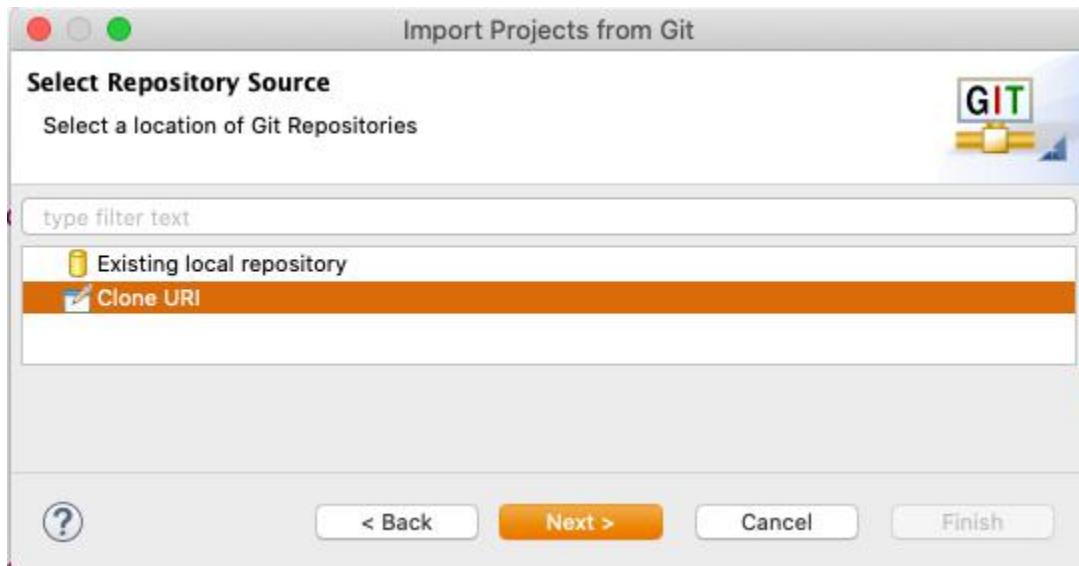
To checkout a remote project, you will have to clone its repository first. Open the GAMA Import wizard: right click the User models node -> Import... -> Other...



Select Git -> Projects from Git and click Next.

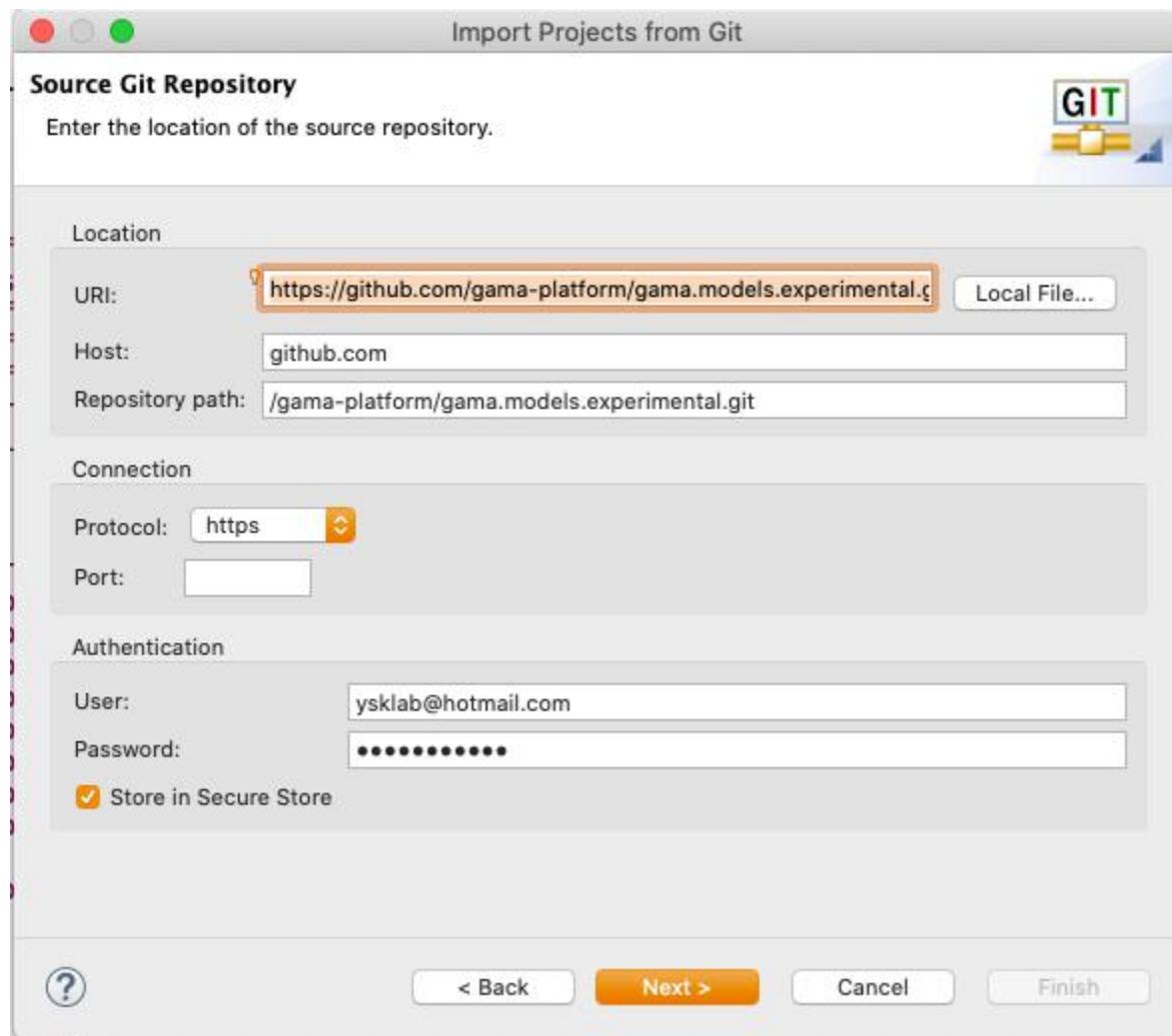


Select "Clone URI" and click Next.

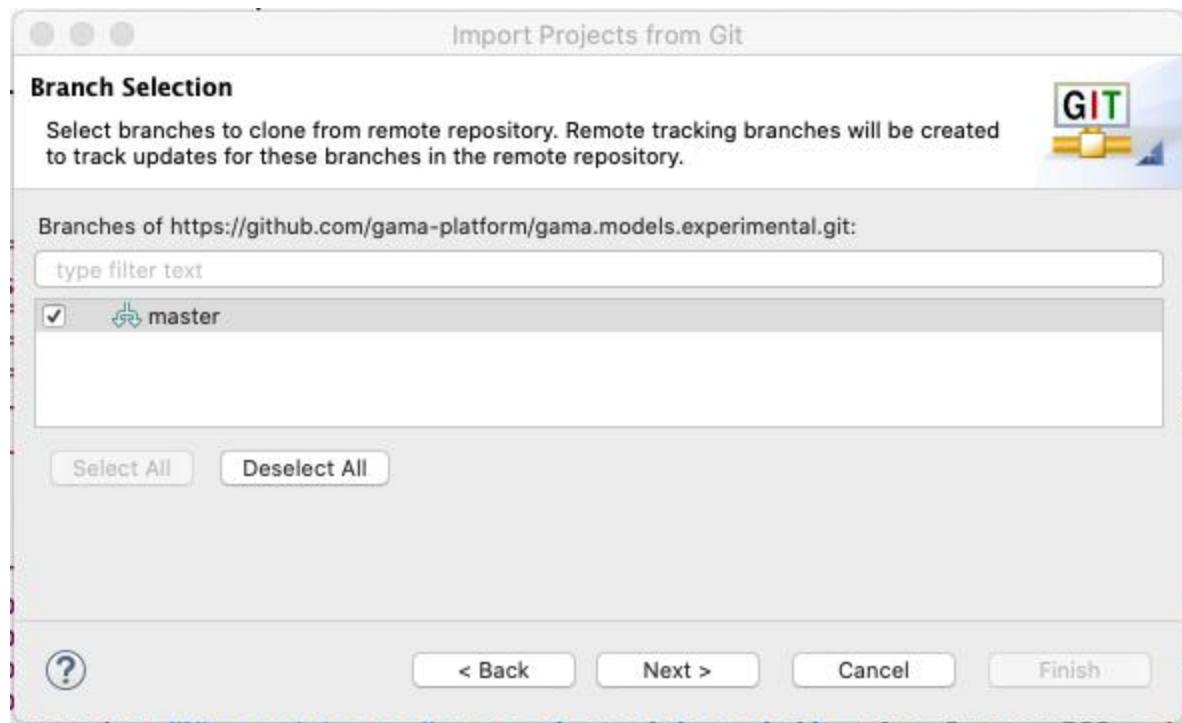


Now you will have to enter the repository's location. Entering the URI will automatically fill some fields. Complete any other required fields and hit Next (e.g,

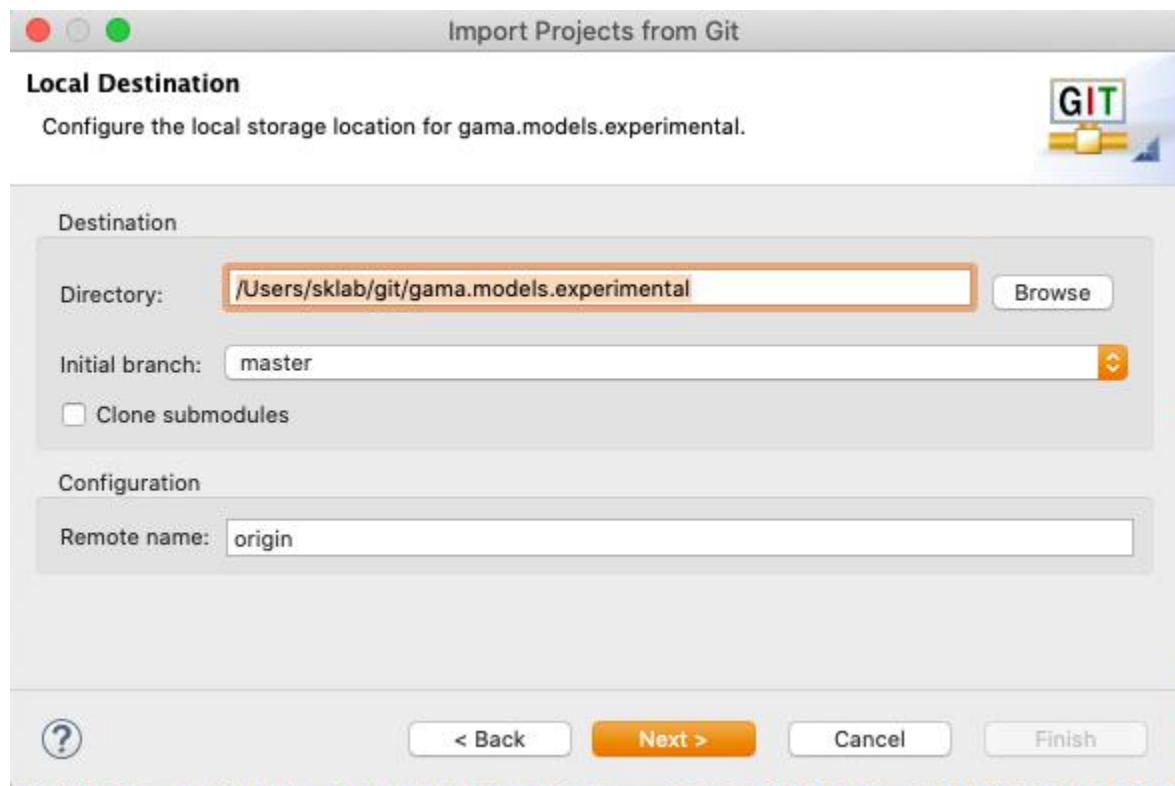
Authentification fields). If you use GitHub, you can copy the URI from the web page.



Select all branches you wish to clone and hit Next again.

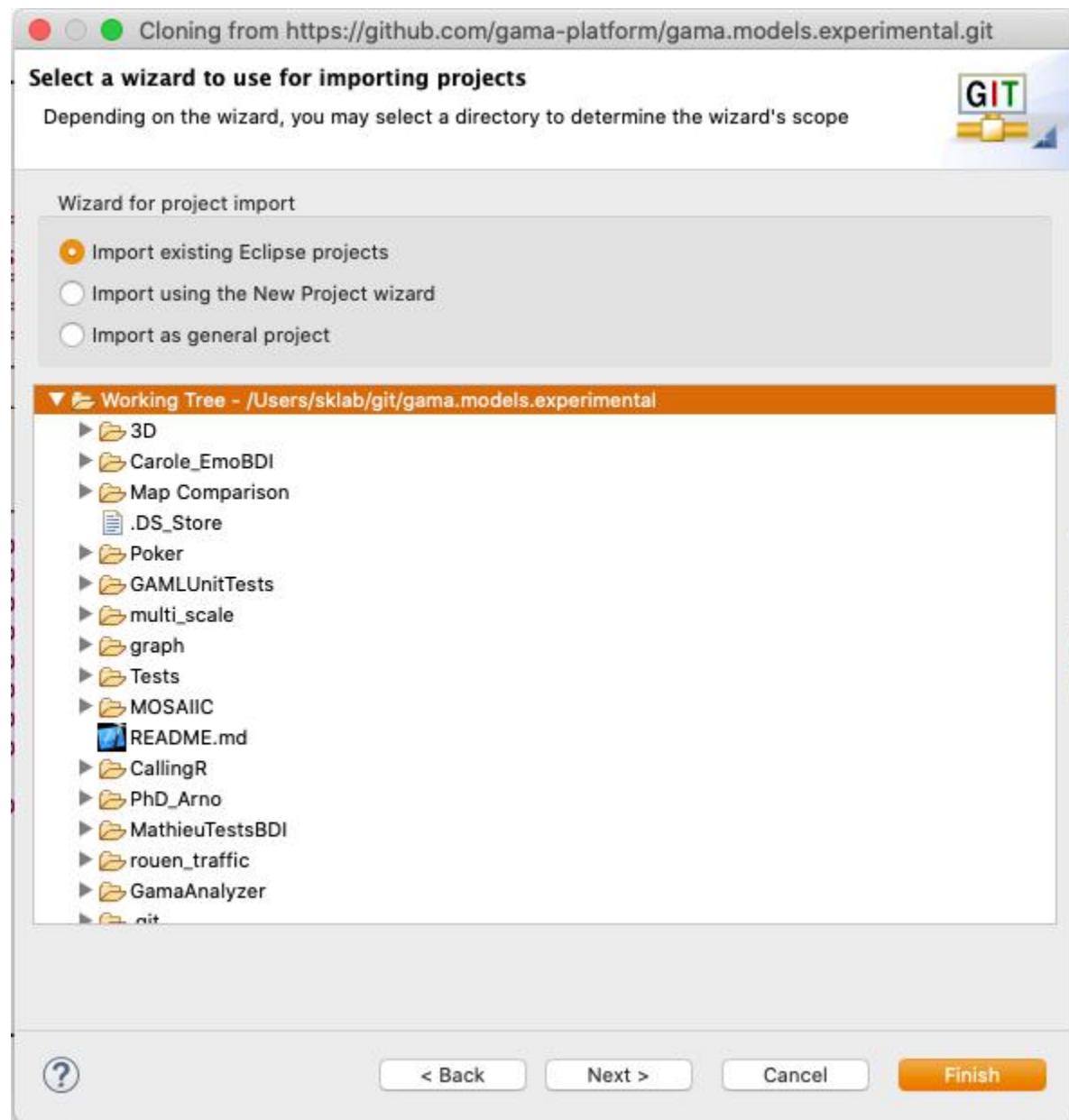


Hit next, then choose a local storage location to save the repository in.

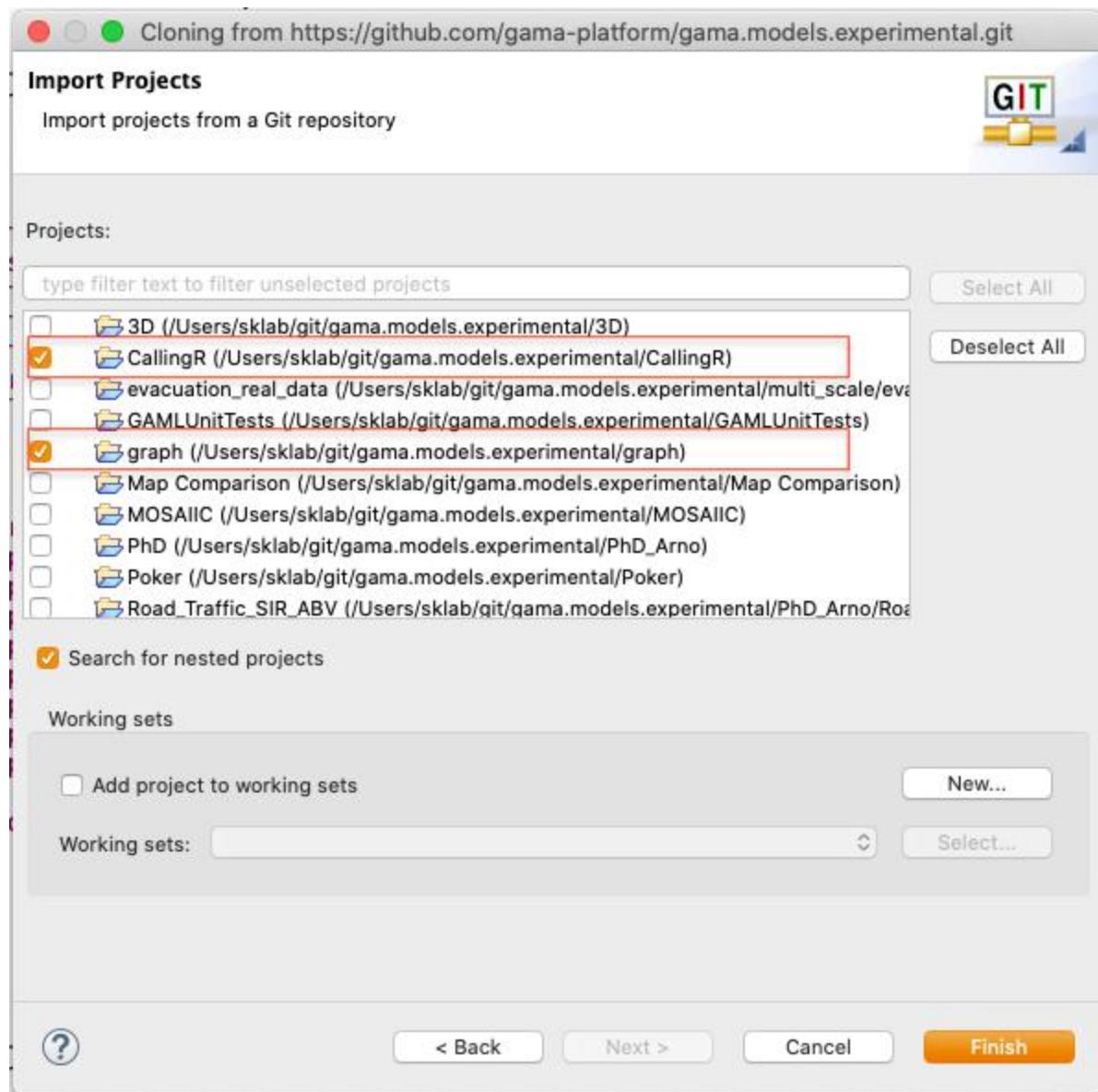


To import the projects, select the cloned repository and hit Next.

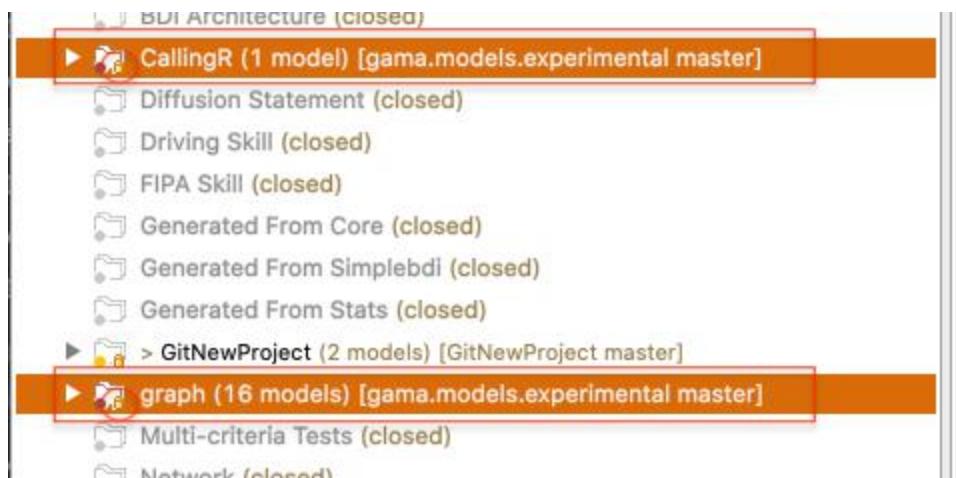
Select Import Existing Projects and hit Next.



In the following window, select all projects you want to import and click Finish.

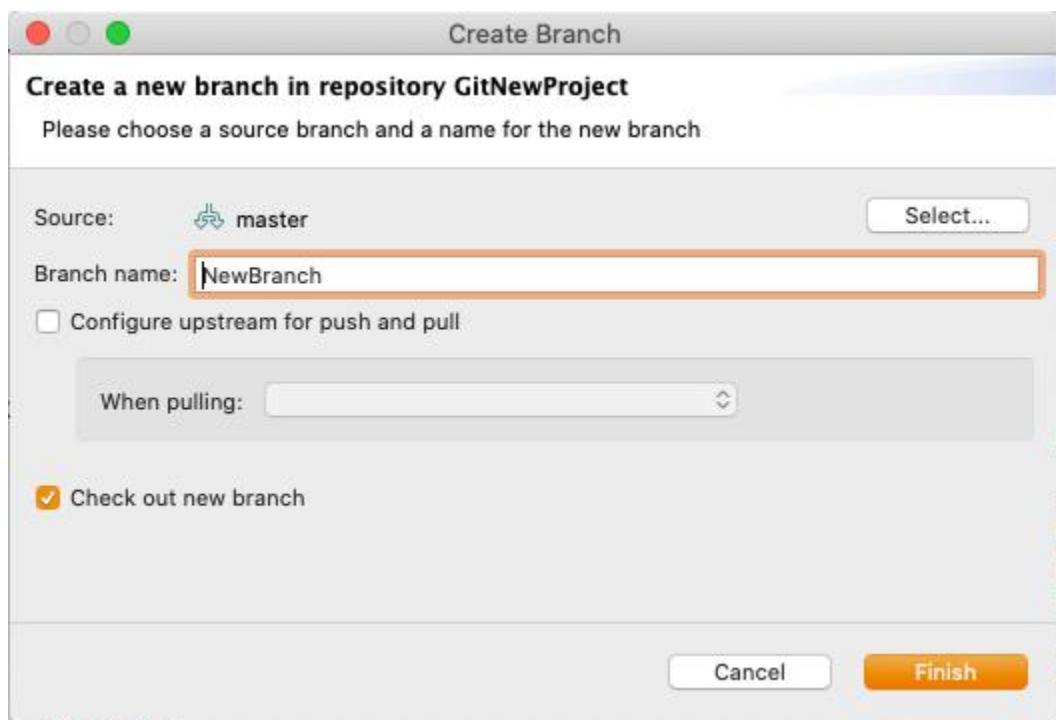


The projects should now appear in the Models Explorer. (Note the repository symbol in the icons indicating that the projects are already shared.)



Create Branches

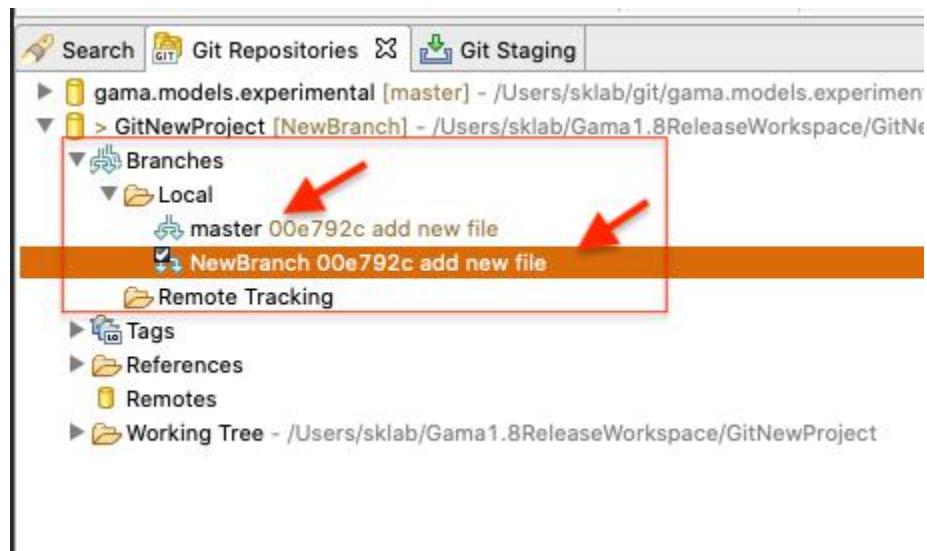
To create a new branch in your repository, right click your project and navigate to Team -> Switch to -> New Branch... from the context menu. Select the branch you want to create a new branch from, hit New branch and enter a name for the new branch.



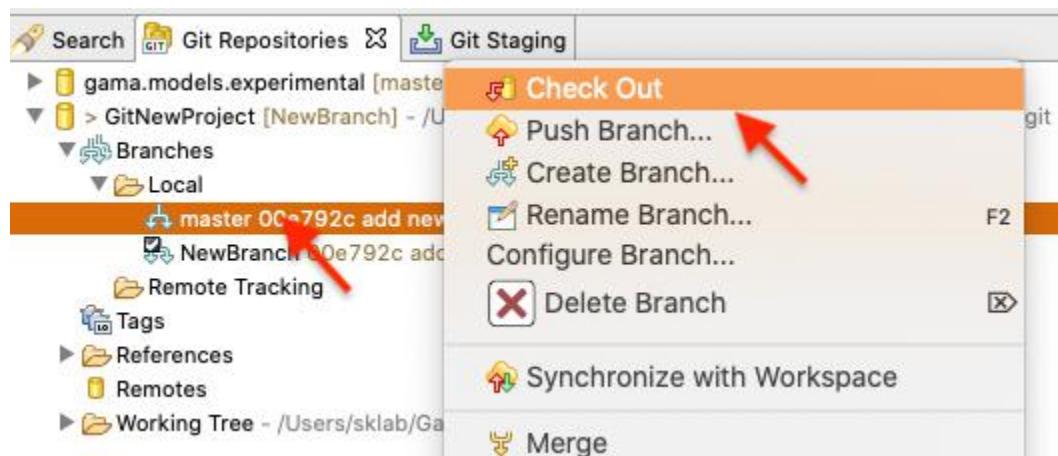
The new branch (NewBranch) should appear in the branch selection window.



You can see all the branches in the Git Repositories view.

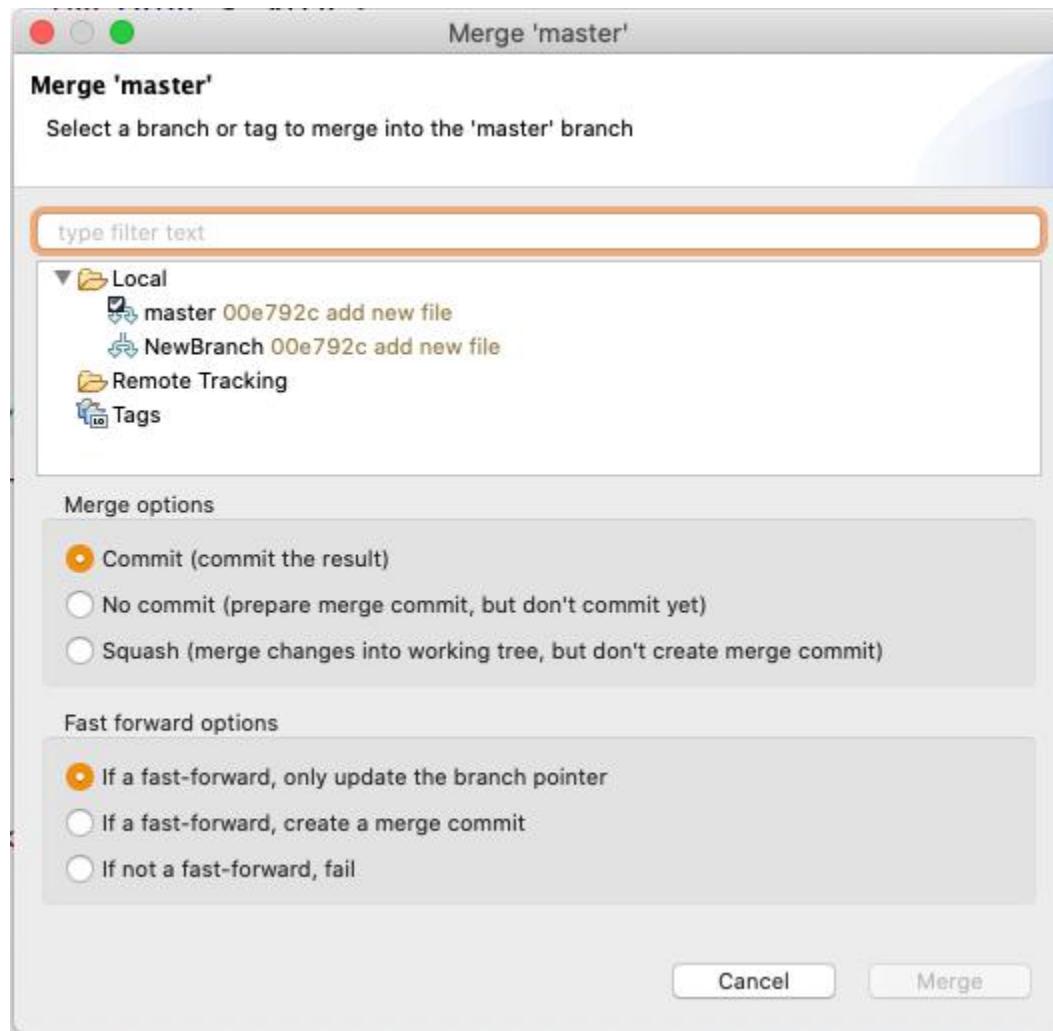


If you would like to checkout the a branch, select it and click Checkout.

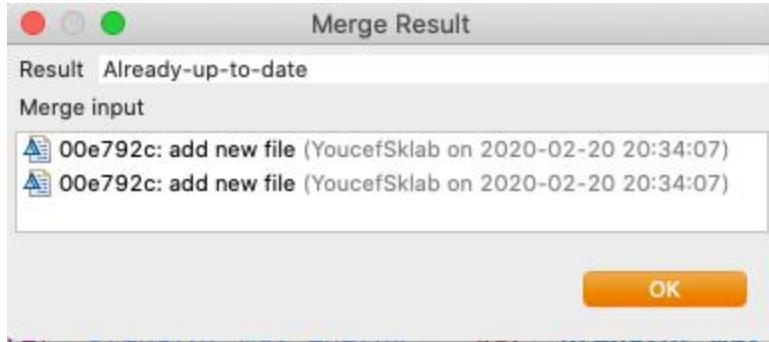


Merge

To merge one branch into another, right click the project node and navigate to Team -> Merge...



The merge will execute and a window will pop-up with the results. The possible results are Already-up-to-date, Fast-forward, Merged, Conflicting, Failed.



Note that a conflicting result will leave the merge process incomplete. You will have to resolve the conflicts and try again. When there are conflicting changes in the working project, the merge will fail.

Fetch and Pull

To update the remote branches when cloning remote repositories (Git creates copies of the branches as local branches and as remote branches) you will have to use Fetch. To perform a Fetch, select Team -> Fetch From... from the project's context menu.

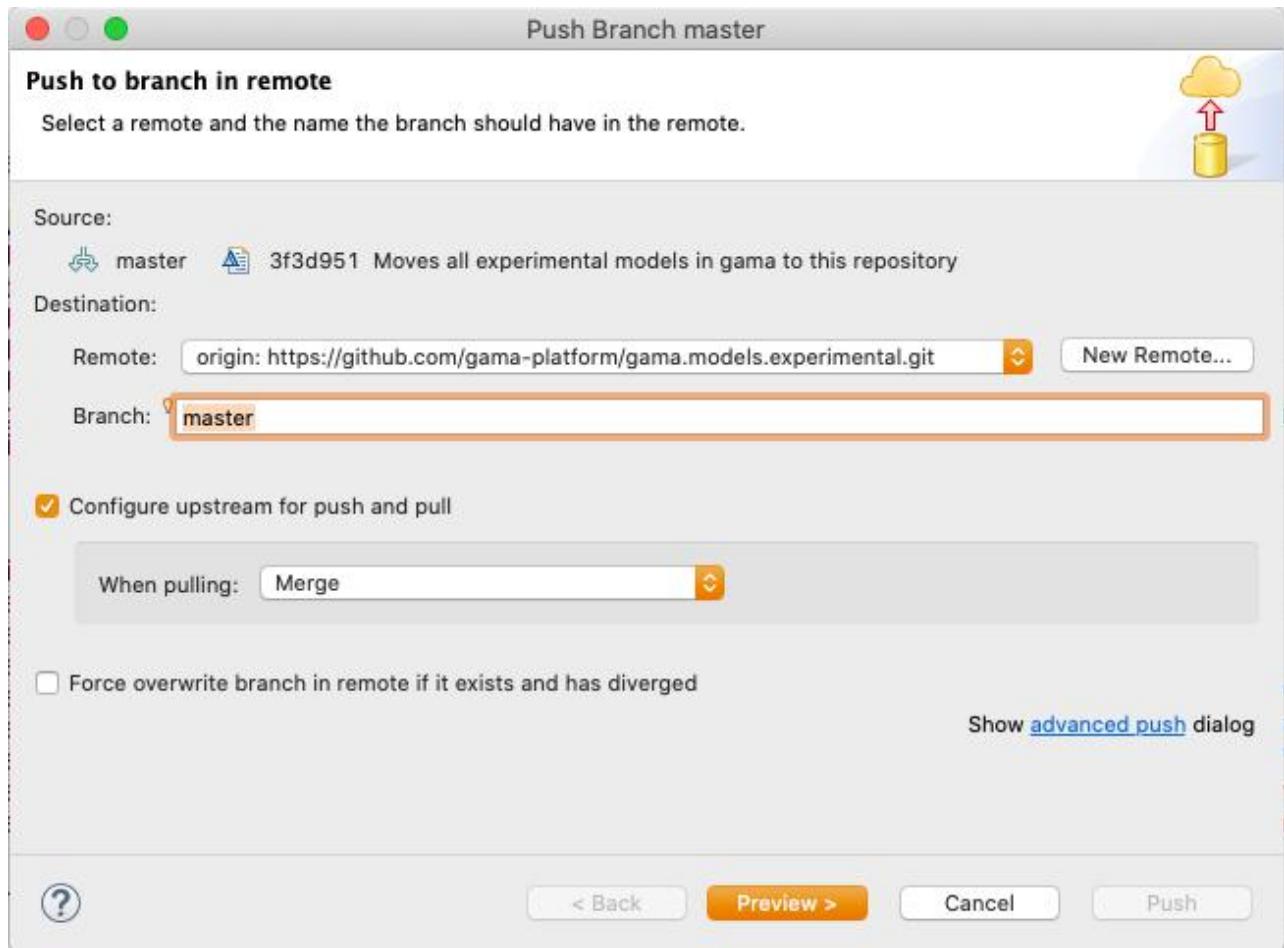
To update your local branches, you will have to perform a Merge operation after fetching.

Pull

Pull combines Fetch and Merge. Select Team -> Pull.

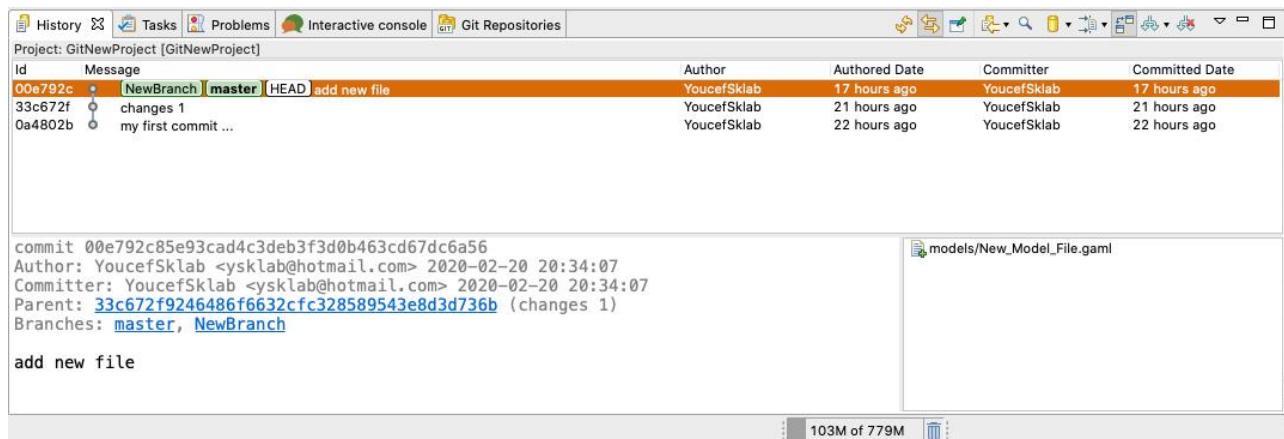
Push

Local changes made to your local branches can be pushed to remote repositories causing a merge from your branches into the branches of the remote repository (X pulls from Y is the same as Y pushes to X). The Push wizard is pretty much the same as the Fetch wizard.



History View

To show the repository history, right click it and select Team -> Show in History. This will open the History View, giving an overview of the commits and allowing you to perform several actions (creating branches/tags, revert, reset...).





>

Version: 1.9.1

GAML References

The GAML references describe in details all the keywords of the GAML language. In particular, they detail all the [expressions](#) (operators, units, literals...), [statements](#), [data types](#), [file types](#), [skills](#), [architectures](#), [built-in species](#)...

Index of keywords

The [Index](#) page contains the exhaustive list of the GAML keywords, with a link to a detailed description of each of them.



> GAML References

> Built-in Species

Version: 1.9.1

Built-in Species

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

It is possible to use in the models a set of built-in agents. These agents allow to directly use some advance features like clustering, multi-criteria analysis, etc. The creation of these agents are similar as for other kinds of agents:

```
create species: my_built_in_agent returns: the_agent;
```

So, for instance, to be able to use clustering techniques in the model:

```
create cluster_builder returns: clusterer;
```

Table of Contents

[agent](#), [AgentDB](#), [base_edge](#), [experiment](#), [graph_edge](#), [graph_node](#), [physical_world](#),

agent

Variables

- `host` (`agent`): Returns the agent that hosts the population of the receiver agent
- `location` (`point`): Returns the location of the agent
- `name` (`string`): Returns the name of the agent (not necessarily unique in its population)
- `peers` (`list`): Returns the population of agents of the same species, in the same host, minus the receiver agent
- `shape` (`geometry`): Returns the shape of the receiver agent

Actions

`_init_`

Returned type: `unknown`

`_step_`

Returned type: `unknown`

AgentDB

AgentDB is an abstract species that can be extended to provide agents with

capabilities to access databases

Variables

- **agents** (`list`): Returns the list of agents for the population(s) of which the receiver agent is a direct or undirect host
- **members** (`container`): Returns the list of agents for the population(s) of which the receiver agent is a direct host

Actions

`close`

Close the established database connection.

Returned type: `unknown` : Returns null if the connection was successfully closed, otherwise, it returns an error.

`connect`

Establish a database connection.

Returned type: `unknown` : Returns null if connection to the server was successfully established, otherwise, it returns an error.

Additional facets:

- **params** (`map`): Connection parameters

`executeUpdate`

- Make a connection to DBMS - Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.

Returned type: `int` : Returns the number of updated rows.

Additional facets:

- `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 the list used parameters to make a connection to DBMS (dbtype, url, port, database, user and passwd).

Returned type: `unknown` : Returns the list of used parameters to make a connection to DBMS.

`insert`

- Make a connection to DBMS - Executes the insert statement.

Returned type: `int` : Returns the number of updated rows.

Additional facets:

- `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`

To check if connection to the server was successfully established or not.

Returned type: `bool` : Returns true if connection to the server was successfully

established, otherwise, it returns false.

select

Make a connection to DBMS and execute the select statement.

Returned type: `list` : Returns the obtained result from executing the select statement.

Additional facets:

- `select` (string): select string
- `values` (list): List of values that are used to replace question marks

setParameter

Sets the parameters to use in order to make a connection to the DBMS (dbtype, url, port, database, user and passwd).

Returned type: `unknown` : null.

Additional facets:

- `params` (map): Connection parameters

testConnection

To test a database connection .

Returned type: `bool` : Returns true if connection to the server was successfully established, otherwise, it returns false.

Additional facets:

- `params` (map): Connection parameters

timeStamp

Get the current time of the system.

Returned type: float : Current time of the system in millisecondes

base_edge

A built-in species for agents representing the edges of a graph, from which one can inherit

Variables

- source (agent) : The source agent of this edge
- target (agent) : The target agent of this edge

Actions

experiment

An experiment is a declaration of the way to conduct simulations on a model. Any experiment attached to a model is a species (introduced by the keyword 'experiment' which directly or indirectly inherits from an abstract species called 'experiment' itself. This abstract species (sub-species of 'agent') defines several attributes and actions that can then be used in any experiment. 'experiment' defines several attributes, which, in addition to the attributes inherited from agent, form the minimal set of knowledge any experiment will have access to.

Variables

- **minimum_cycle_duration** (`float`): The minimum duration (in seconds) a simulation cycle should last. Default is 0. Units can be used to pass values smaller than a second (for instance '10 °msec')
- **model_path** (`string`): Contains the absolute path to the folder in which the current model is located
- **parameters** (`map`): A parameters set of this experiment agent
- **project_path** (`string`): Contains the absolute path to the project in which the current model is located
- **rng** (`string`): The random number generator to use. Four different ones are at the disposal of the modeler: 'mersenne' represents the default generator, based on the Mersenne-Twister algorithm. Very reliable, fast and deterministic (that is, using the same seed and the same sequence of calls, it will return the same stream of pseudo-random numbers). This algorithm is however not safe to use in simulations where agents can behave in parallel; 'threaded' is a very fast generator, based on the DotMix algorithm, that can be safely used in parallel simulations as it creates one instance per thread. However, determinism cannot be guaranteed and this algorithm does not accept a seed as each instance will compute its own; 'parallel' is a version of the Mersenne-Twister algorithm that can be safely used in parallel simulations by preventing a concurrent access to its internal state. Determinism is guaranteed (in terms of generation, but not in terms of execution, as the sequence in which the threads will access it cannot be determined) and it performs a bit slower than its base version. 'java' invokes the standard generator provided by the JDK, deterministic and thread-safe, albeit slower than all the other ones
- **rng_usage** (`int`): Returns the number of times the random number generator of the experiment has been drawn
- **seed** (`float`): The seed of the random number generator. Each time it is set, the

random number generator is reinitialized. WARNING: Setting it to zero actually means that you let GAMA choose a random seed

- **simulation** (`agent`): Contains a reference to the current simulation being run by this experiment
- **simulations** (`list`): Contains the list of currently running simulations
- **workspace_path** (`string`): Contains the absolute path to the workspace of GAMA

Actions

compact_memory

Forces a 'garbage collect' of the unused objects in GAMA

Returned type: `unknown`

update_outputs

Forces all outputs to refresh, optionally recomputing their values

Returned type: `unknown`

Additional facets:

- **recompute** (`boolean`): Whether or not to force the outputs to make a computation step

graph_edge

A species that represents an edge of a graph made of agents. The source and the target of the edge should be agents

Variables

- `source` (`agent`): The source agent of this edge
- `target` (`agent`): The target agent of this edge

Actions

`graph_node`

A base species to use as a parent for species representing agents that are nodes of a graph

Variables

- `my_graph` (`graph`): A reference to the graph containing the agent

Actions

`related_to`

This operator should never be called

Returned type: `bool`

Additional facets:

- `other` (`agent`): The other agent

physical_world

The base species for models that act as a 3D physical world. Can register and manage agents provided with either the 'static_body' or 'dynamic_body' skill. Inherits from 'static_body', so it can also act as a physical body itself (with a 'mass', 'friction', 'gravity'), of course without motion -- in this case, it needs to register itself as a physical agent using the 'register' action

Variables

- **accurate_collision_detection** (`boolean`): Enables or not a better (but slower) collision detection
- **automated_registration** (`boolean`): If set to true (the default), makes the world automatically register and unregister agents provided with either the 'static_body' or 'dynamic_body' skill. Otherwise, they must be registered using the 'register' action, which can be useful when only some agents need to be considered as 'physical agents'. Note that, in any case, the world needs to manually register itself if it is supposed to act as a physical body.
- **gravity** (`point`): Defines the value of gravity in this world. The default value is set to -9.80665 on the z-axis, that is 9.80665 m/s² towards the 'bottom' of the world. Can be set to any direction and intensity and applies to all the bodies present in the physical world
- **library** (`string`): This attribute allows to manually switch between two physics library, named 'bullet' and 'box2D'. The Bullet library, which comes in two flavors (see 'use_native') and the Box2D libray in its Java version (<https://github.com/jbox2d/jbox2d>). Bullet is the default library but models in 2D should better use Box2D
- **max_substeps** (`int`): If equal to 0 (the default), makes the simulation engine be stepped alongside the simulation (no substeps allowed). Otherwise, sets the

maximum number of physical simulation substeps that may occur within one GAMA simulation step

- **terrain** (field): This attribute is a matrix of float that can be used to represent a 3D terrain. The shape of the world, in that case, should be a box, where the dimension on the z-axis is used to scale the z-values of the DEM. The world needs to be register itself as a physical object
- **use_native** (boolean): This attribute allows to manually switch between the Java version of the Bullet library (JBullet, a modified version of <https://github.com/stephengold/jbullet>, which corresponds to version 2.72 of the original library) and the native Bullet library (Libbulletjme, <https://github.com/stephengold/Libbulletjme>, which is kept up-to-date with the 3.x branch of the original library). The native version is the default one unless the libraries cannot be loaded, making JBullet the default

Actions

register

An action that allows to register agents in this physical world. Unregistered agents will not be governed by the physical laws of this world. If the world is to play a role in the physical world, then it needs to register itself (i.e. do `register([self]);`)

Returned type: `unknown`

Additional facets:

- **bodies** (container): the list or container of agents to register in this physical world

Version: 1.9.1

Built-in Skills

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

Introduction

Skills are built-in modules, written in Java, that provide a set of related built-in variables and built-in actions (in addition to those already provided by GAMA) to the species that declare them. A declaration of skill is done by filling the skills attribute in the species definition:

```
species my_species skills: [skill1, skill2] {  
    ...  
}
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. An example of skill is the `moving` skill.

So, for instance, if a species is declared as:

```
species foo skills: [moving]{  
    ...  
}
```

Its agents will automatically be provided with the following variables : `speed`,

`heading`, `destination` and the following actions: `move`, `goto`, `wander`, `follow` in addition to those built-in in species and declared by the modeller. Most of these variables, except the ones marked read-only, can be customized and modified like normal variables by the modeller. For instance, one could want to set a maximum for the speed; this would be done by redeclaring it like this:

```
float speed max:100 min:0;
```

Or, to obtain a speed increasing at each simulation step:

```
float speed max:100 min:0 <- 1 update: speed * 1.01;
```

Or, to change the speed in a behavior:

```
if speed = 5 {  
    speed <- 10;  
}
```

Table of Contents

[advanced_driving](#), [driving](#), [dynamic_body](#), [fipa](#), [messaging](#), [moving](#), [moving3D](#), [network](#), [pedestrian](#), [pedestrian_road](#), [skill_road](#), [skill_road_node](#), [SQLSKILL](#), [static_body](#), [thread](#),

advanced_driving

Variables

- `acc_bias` (`float`): the bias term used for asymmetric lane changing, parameter 'a_bias' in MOBIL
- `acc_gain_threshold` (`float`): the minimum acceleration gain for the vehicle to switch to another lane, introduced to prevent frantic lane changing. Known as the parameter 'a_th' in the MOBIL lane changing model
- `acceleration` (`float`): the current acceleration of the vehicle (in m/s²)
- `allowed_lanes` (`list`): a list containing possible lane index values for the attribute lowest_lane
- `current_index` (`int`): the index of the current edge (road) in the path
- `current_lane` (`int`): the current lane on which the agent is
- `current_path` (`path`): the path which the agent is currently following
- `current_road` (`agent`): the road which the vehicle is currently on
- `current_target` (`agent`): the current target of the agent
- `delta_idm` (`float`): the exponent used in the computation of free-road acceleration in the Intelligent Driver Model
- `distance_to_current_target` (`float`): euclidean distance to the current target node
- `distance_to_goal` (`float`): euclidean distance to the endpoint of the current segment
- `final_target` (`agent`): the final target of the agent
- `follower` (`agent`): the vehicle following this vehicle
- `ignore_oneway` (`boolean`): if set to `true`, the vehicle will be able to violate one-way traffic rule

- **lane_change_coldown** (`float`): the duration that a vehicle must wait before changing lanes again
- **lane_change_limit** (`int`): the maximum number of lanes that the vehicle can change during a simulation step
- **leading_distance** (`float`): the distance to the leading vehicle
- **leading_speed** (`float`): the speed of the leading vehicle
- **leading_vehicle** (`agent`): the vehicle which is right ahead of the current vehicle. If this is set to nil, the leading vehicle does not exist or might be very far away.
- **linked_lane_limit** (`int`): the maximum number of linked lanes that the vehicle can use; the default value is -1, i.e. the vehicle can use all available linked lanes
- **lowest_lane** (`int`): the lane with the smallest index that the vehicle is in
- **max_acceleration** (`float`): the maximum acceleration of the vehicle. Known as the parameter 'a' in the Intelligent Driver Model
- **max_deceleration** (`float`): the maximum deceleration of the vehicle. Known as the parameter 'b' in the Intelligent Driver Model
- **max_safe_deceleration** (`float`): the maximum deceleration that the vehicle is willing to induce on its back vehicle when changing lanes. Known as the parameter 'b_save' in the MOBIL lane changing model
- **max_speed** (`float`): the maximum speed that the vehicle can achieve. Known as the parameter 'v0' in the Intelligent Driver Model
- **min_safety_distance** (`float`): the minimum distance of the vehicle's front bumper to the leading vehicle's rear bumper, known as the parameter s0 in the Intelligent Driver Model
- **min_security_distance** (`float`): the minimal distance to another vehicle
- **next_road** (`agent`): the road which the vehicle will enter next
- **num_lanes_occupied** (`int`): the number of lanes that the vehicle occupies
- **on_linked_road** (`boolean`): is the agent on the linked road?

- **politeness_factor** (`float`): determines the politeness level of the vehicle when changing lanes. Known as the parameter 'p' in the MOBIL lane changing model
- **proba_block_node** (`float`): probability to block a node (do not let other vehicle cross the crossroad), within one second
- **proba_lane_change_down** (`float`): probability to change to a lower lane (right lane if right side driving) to gain acceleration, within one second
- **proba_lane_change_up** (`float`): probability to change to a upper lane (left lane if right side driving) to gain acceleration, within one second
- **proba_respect_priorities** (`float`): probability to respect priority (right or left) laws, within one second
- **proba_respect_stops** (`list`): probability to respect stop laws - one value for each type of stop, within one second
- **proba_use_linked_road** (`float`): probability to change to a linked lane to gain acceleration, within one second
- **real_speed** (`float`): the actual speed of the agent (in meter/second)
- **right_side_driving** (`boolean`): are vehicles driving on the right size of the road?
- **safety_distance_coeff** (`float`): the coefficient for the computation of the min distance between two vehicles (according to the vehicle speed -
security_distance =max(min_security_distance, safety_distance_coeff *
min(self.real_speed, other.real_speed))
- **security_distance_coeff** (`float`): the coefficient for the computation of the min distance between two vehicles (according to the vehicle speed -
safety_distance =max(min_safety_distance, safety_distance_coeff *
min(self.real_speed, other.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 vehicle 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)
- `time_headway` (`float`): the time gap that to the leading vehicle that the driver must maintain. Known as the parameter 'T' in the Intelligent Driver Model
- `time_since_lane_change` (`float`): the elapsed time since the last lane change
- `using_linked_road` (`boolean`): indicates if the vehicle is occupying at least one lane on the linked road
- `vehicle_length` (`float`): the length of the vehicle (in meters)
- `violating_oneway` (`boolean`): indicates if the vehicle is moving in the wrong direction on an one-way (unlinked) road

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)

Returned type: `float` : the remaining time

Additional facets:

- `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

Examples:

```
do osm_follow path: the_path on: road_network;
```

choose_lane

Override this if you want to manually choose a lane when entering new road. By default, the vehicle tries to stay in the current lane. If the new road has fewer lanes than the current one and the current lane index is too big, it tries to enter the most uppermost lane.

Returned type: `int` : an integer representing the lane index

Additional facets:

- `new_road` (agent): the new road that's the vehicle is going to enter

compute_path

Action to compute the shortest path to the target node, or shortest path based on the provided list of nodes

Returned type: `path` : the computed path, or nil if no valid path is found

Additional facets:

- `graph` (graph): the graph representing the road network
- `target` (agent): the target node to reach
- `source` (agent): the source node (optional, if not defined, closest node to the agent location)
- `nodes` (list): the nodes forming the resulting path

Examples:

```
do compute_path graph: road_network target: target_node;
do compute_path graph: road_network nodes: [node1, node5, node10];
```

drive

action to drive toward the target

Returned type: `bool`

Examples:

```
do drive;
```

drive_random

action to drive by chosen randomly the next road

Returned type: `bool`

Additional facets:

- `graph` (graph): a graph representing the road network
- `proba_roads` (map): a map containing for each road (key), the probability to be selected as next road (value)

Examples:

```
do drive_random init_node: some_node;
```

`external_factor_impact`

action that allows to define how the remaining time is impacted by external factor

Returned type: `float` : the remaining time

Additional facets:

- `new_road` (agent): the road on which to the vehicle wants to go
- `remaining_time` (float): the remaining time

Examples:

```
do external_factor_impact new_road: a_road remaining_time: 0.5;
```

`force_move`

action to drive by chosen randomly the next road

Returned type: `float`

Additional facets:

- `lane` (int): the lane on which to make the agent move
- `acceleration` (float): acceleration of the vehicle
- `time` (float): time of move

Examples:

```
do drive_random init_node: some_node;
```

goto_drive

moves the agent towards the target passed in the arguments.

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- `target` (geometry): the entity towards which to move.
- `speed` (float): the speed to use for this move (replaces the current value of speed)
- `on` (any type): graph, topology, list of geometries or map of geometries that restrain 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)
- `following` (path): Path to follow.

Examples:

```
do goto_drive target: one_of road on: road_network;
```

lane_choice

action to choose a lane

Returned type: `int` : the chosen lane, return -1 if no lane can be taken

Additional facets:

- `new_road` (agent): the road on which to choose the lane

Examples:

```
do lane_choice new_road: a_road;
```

on_entering_new_road

**override this if you want to do something when the vehicle enters a new road
(e.g. adjust parameters)**

Returned type: `void`

path_from_nodes

action to compute a path from a list of nodes according to a given graph

Returned type: `path` : the computed path, return nil if no path can be taken

Additional facets:

- `graph` (graph): the graph representing the road network
- `nodes` (list): the list of nodes composing the path

Examples:

```
do compute_path_from_nodes graph: road_network nodes: [node1, node5,  
node10];
```

ready_to_cross

action to test if the vehicle cross a road node to move to a new road

Returned type: `bool` : true if the vehicle can cross the road node, false otherwise

Additional facets:

- **node** (agent): the road node to test
- **new_road** (agent): the road to test

Examples:

```
do is_ready_next_road new_road: a_road lane: 0;
```

speed_choice

action to choose a speed

Returned type: **float** : the chosen speed

Additional facets:

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

Examples:

```
do speed_choice new_road: the_road;
```

test_next_road

action to test if the vehicle can take the given road

Returned type: **bool** : true (the vehicle can take the road) or false (the vehicle cannot take the road)

Additional facets:

- **new_road** (agent): the road to test

Examples:

```
do test_next_road new_road: a_road;
```

unregister

remove the vehicle from its current roads

Returned type: `bool`

Examples:

```
do unregister
```

driving

Variables

- `lanes_attribute` (`string`): the name of the attribute 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.

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- `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)

Examples:

```
do follow speed: speed * 2 path: road_path;
```

goto_driving

moves the agent towards the target passed in the arguments while considering

the other agents in the network (only for graph topology)

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- `target` (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` (any type): 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 attribute of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

Examples:

```
do gotoTraffic target: one_of (list (species (self))) speed: speed *  
2 on: road_network living_space: 2.0;
```

dynamic_body

Variables

- **angular_damping** (`float`): Between 0 and 1. an angular deceleration coefficient that occurs even without contact
- **angular_velocity** (`point`): The angular velocity of the agent in the three directions, expressed as a point.
- **contact_damping** (`float`): Between 0 and 1. a deceleration coefficient that occurs in case of contact. Only available in the native Bullet library (no effect on the Java implementation)
- **damping** (`float`): Between 0 and 1. a linear deceleration coefficient that occurs even without contact
- **velocity** (`point`): The linear velocity of the agent in the three directions, expressed as a point.

Actions

apply

An action that allows to apply different effects to the object, like forces, impulses, etc.

Returned type: `unknown`

Additional facets:

- **clearance** (`boolean`): If true clears all forces applied to the agent and clears its velocity as well
- **impulse** (`point`): An idealised change of momentum. Adds to the velocity of the

object. This is the kind of push that you would use on a pool billiard ball.

- **force** (point): Move (push) the object once with a certain moment, expressed as a point (vector). Adds to the existing forces.
 - **torque** (point): Rotate (twist) the object once around its axes, expressed as a point (vector)
-

fipa

The fipa skill offers some primitives and built-in variables which enable agent to communicate with each other using the FIPA interaction protocol.

Variables

- **accept_proposals** (list): A list of 'accept_proposal' performative messages in the agent's mailbox
- **agrees** (list): A list of 'agree' performative messages.
- **cancels** (list): A list of 'cancel' performative messages.
- **cfps** (list): A list of 'cfp' (call for proposal) performative messages.
- **conversations** (list): A list containing the current conversations of agent. Ended conversations are automatically removed from this list.
- **failures** (list): A list of 'failure' performative messages.
- **informs** (list): A list of 'inform' performative messages.
- **proposes** (list): A list of 'propose' performative messages .
- **queries** (list): A list of 'query' performative messages.
- **refuses** (list): A list of 'reject_proposal' performative messages.
- **reject_proposals** (list): A list of 'reject_proposal' 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.

Returned type: unknown

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

agree

Replies a message with an 'agree' performative message.

Returned type: unknown

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

cancel

Replies a message with a 'cancel' peformative message.

Returned type: unknown

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

cfp

Replies a message with a 'cfp' performative message.

Returned type: `unknown`

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

end_conversation

Reply a message with an 'end_conversation' perfrormative 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. Please note that if the contents of the messages of the conversation are not read, then this command has no effect (i.e. it must be read by at least one of the agents in the conversation)

Returned type: `unknown`

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

failure

Replies a message with a 'failure' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The content of the replying message

inform

Replies a message with an 'inform' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The content of the replying message

propose

Replies a message with a 'propose' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The content of the replying message

`query`

Replies a message with a 'query' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The content of the replying message

`refuse`

Replies a message with a 'refuse' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The contents of the replying message

`reject_proposal`

Replies a message with a 'reject_proposal' performative message.

Returned type: `unknown`

Additional facets:

- `message` (message): The message to be replied
- `contents` (list): The content of the replying message

reply

Replies a message. This action should be only used to reply a message in a 'no-protocol' conversation and with a 'user defined performative'. For performatives supported by GAMA (i.e., standard FIPA performatives), please use the 'action' with the same name of 'performative'. For example, to reply a message with a 'request' performative message, the modeller should use the 'request' action.

Returned type: unknown

Additional facets:

- **message** (message): The message to be replied
- **performative** (string): The performative of the replying message
- **contents** (list): The content of the replying message

request

Replies a message with a 'request' performative message.

Returned type: unknown

Additional facets:

- **message** (message): The message to be replied
- **contents** (list): The content of the replying message

send

Starts a conversation/interaction protocol.

Returned type: message

Additional facets:

- **to** (list): A list of receiver agents
- **contents** (list): The content of the message. A list of any GAML type
- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

start_conversation

Starts a conversation/interaction protocol.

Returned type: `message`

Additional facets:

- **to** (list): A list of receiver agents
- **contents** (list): The content of the message. A list of any GAML type
- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

subscribe

Replies a message with a 'subscribe' performative message.

Returned type: `unknown`

Additional facets:

- **message** (message): The message to be replied
 - **contents** (list): The content of the replying message
-

messaging

A simple skill that provides agents with a mailbox than can be filled with messages

Variables

- `mailbox` (list): The list of messages that can be consulted by the agent

Actions

send

Action used to send a message (that can be of any kind of object) to an agent or a server.

Returned type: `message`

Additional facets:

- `to` (any type): The agent, or server, to which this message will be sent to
- `contents` (any type): The contents of the message, an arbitrary object

Examples:

```
do send to:dest contents:"This message is sent by " + name + " to " + dest;
```

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

- `current_edge` (`geometry`): Represents the agent/geometry on which the agent is located (only used with a graph)
- `current_path` (`path`): Represents the path on which the agent is moving on (goto action on a graph)
- `destination` (`point`): Represents the next location of the agent if it keeps its current speed and heading (read-only). **Only correct in continuous topologies and may return nil values if the destination is outside the environment**
- `heading` (`float`): Represents the absolute heading of the agent in degrees.
- `location` (`point`): Represents the current position of the agent
- `real_speed` (`float`): Represents the actual speed of the agent (in meter/second)
- `speed` (`float`): Represents the speed of the agent (in meter/second)

Actions

follow

moves the agent along a given path passed in the arguments.

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- **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)

Examples:

```
do follow speed: speed * 2 path: road_path;
```

goto

moves the agent towards the target passed in the arguments.

Returned type: **path** : optional: the path followed by the agent.

Additional facets:

- **target** (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** (any type): graph, topology, list of geometries or map of geometries that restrain this move
- **recompute_path** (boolean): if false, the path is not recomputed 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.

Examples:

```
do goto target: (one_of road).location speed: speed * 2 on:  
road_network;
```

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.

Returned type: **path**

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (float): the angle (in degree) of the target direction.
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

Examples:

```
do move speed: speed - 10 heading: heading + rnd (30) bounds: agentA;
```

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.

Returned type: **bool**

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **amplitude** (float): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)
- **on** (graph): the graph that restrains this move (the agent moves on the graph)
- **proba_edges** (map): When the agent moves on a graph, the probability to choose another edge. If not defined, each edge has the same probability to be chosen

Examples:

```
do wander speed: speed - 10 amplitude: 120 bounds: agentA;
```

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** (float): the absolute heading of the agent in degrees (in the range 0-359)

- **pitch** (float): the absolute pitch of the agent in degrees (in the range 0-359)
- **roll** (float): 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.

Returned type: **path**

Additional facets:

- **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)
- **roll** (int): int, optional, the direction to take for this move (replaces the current value of roll)
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

Examples:

```
do move speed: speed - 10 heading: heading + rnd (30) bounds: agentA;
```

network

The network skill provides new features to let agents exchange message through network. Sending and receiving data is done with the messaging skill's actions.

Variables

- **network_groups** (`list`): The set of groups the agent belongs to
- **network_name** (`string`): Net ID of the agent
- **network_server** (`list`): The list of all the servers to which the agent is connected

Actions

connect

Action used by a networking agent to connect to a server or to create a server.

Returned type: `bool`

Additional facets:

- **protocol** (`string`): protocol type (MQTT (by default), TCP, UDP, websocket, arduino): the possible value ares 'udp_server', 'udp_emitter', 'tcp_server', 'tcp_client', 'websocket_server', 'websocket_client', 'http', 'arduino', otherwise the MQTT protocol is used.
- **port** (`int`): Port number
- **raw** (`boolean`): message type raw or rich
- **with_name** (`string`): ID of the agent (its name) for the simulation
- **login** (`string`): login for the connection to the server

- **password** (string): password associated to the login
- **force_network_use** (boolean): force the use of the network even interaction between local agents
- **to** (string): server URL (localhost or a server URL)
- **size_packet** (int): For UDP connection, it sets the maximum size of received packets (default = 1024bits).

Examples:

```
do connect with_name:"any_name";
do connect to:"localhost" port:9876 with_name:"any_name";
do connect to:"localhost" protocol:"MQTT" port:9876
with_name:"any_name";
do connect to:"localhost" protocol:"udp_server" port:9876
with_name:"Server";
do connect to:"localhost" protocol:"udp_client" port:9876
with_name:"Client";
do connect to:"localhost" protocol:"udp_server" port:9877
size_packet: 4096;
do connect to:"localhost" protocol:"tcp_client" port:9876;
do connect to:"localhost" protocol:"tcp_server" port:9876 raw:true;
do connect to: "https://openlibrary.org" protocol: "http" port: 443
raw: true;
do connect protocol: "arduino";
```

execute

Action that executes a command in the OS, as if it is executed from a terminal.

Returned type: `string` : The error message if any

Additional facets:

- **command** (string): command to execute

fetch_message

Fetch the first message from the mailbox (and remove it from the mailing box). If the mailbox is empty, it returns a nil message.

Returned type: message

Examples:

```
message mess <- fetch_message();
loop while:has_more_message(){
    message mess <- fetch_message();
    write message.contents;
}
```

fetch_message_from_network

Fetch all messages from network to mailbox. Use this in specific case only, this action is done at the end of each step.

Returned type: bool

Examples:

```
do fetch_message_from_network;//forces gama to get all the new
messages since the begining of the cycle
loop while: has_more_message(){
    message mess <- fetch_message();
    write message.contents;
}
```

has_more_message

Check whether the mailbox contains any message.

Returned type: `bool`

Examples:

```
bool mailbox_contain_messages <- has_more_message();
loop while:has_more_message(){
    message mess <- fetch_message();
    write message.contents;
}
```

`join_group`

Allow an agent to join a group of agents in order to broadcast messages to other members or to receive messages sent by other members. Note that all members of the group called : "ALL".

Returned type: `bool`

Additional facets:

- `with_name` (string): name of the group

Examples:

```
do join_group with_name:"group name";
do join_group with_name:"group name";
do send to:"group name" contents:"I am new in this group";
```

`leave_group`

leave a group of agents. The leaving agent will not receive any message from the group. Otherwise, it can send messages to the left group

Returned type: `bool`

Additional facets:

- **with_name** (string): name of the group the agent wants to leave

Examples:

```
do leave_group with_name:"my_group";
```

pedestrian

Variables

- **A_obstacles_SFM** (float): Value of A in the SFM model for obstacles - the force of repulsive interactions (classic values : mean = 4.5, std = 0.3)
- **A_pedestrians_SFM** (float): Value of A in the SFM model for pedestrians - the force of repulsive interactions (classic values : mean = 4.5, std = 0.3)
- **avoid_other** (boolean): has the pedestrian to avoid other pedestrians?
- **B_obstacles_SFM** (float): Value of B in the SFM model for obstacles - the range (in meters) of repulsive interactions
- **B_pedestrians_SFM** (float): Value of B in the SFM model for pedestrians - the range (in meters) of repulsive interactions
- **current_index** (int): the current index of the agent waypoint (according to the waypoint list)
- **current_waypoint** (geometry): the current waypoint of the agent
- **final_waypoint** (geometry): the final waypoint of the agent
- **forces** (map): the map of forces
- **gama_SFM** (float): Value of gama in the SFM model the amount of normal social

force added in tangential direction. between 0.0 and 1.0 (classic values : mean = 0.35, std = 0.01)

- **k_SFM** (`float`): Value of k in the SFM model: force counteracting body compression
- **kappa_SFM** (`float`): Value of kappa in the SFM model: friction counteracting body compression
- **lambda_SFM** (`float`): Value of lambda in the SFM model - the (an-)isotropy (between 0.0 and 1.0)
- **minimal_distance** (`float`): Minimal distance between pedestrians
- **n_prime_SFM** (`float`): Value of n' in the SFM model (classic values : mean = 3.0, std = 0.7)
- **n_SFM** (`float`): Value of n in the SFM model (classic values : mean = 2.0, std = 0.1)
- **obstacle_consideration_distance** (`float`): Distance of consideration of obstacles (to compute the nearby obstacles, used as distance, the max between this value and (step * speed) - classic value: 3.5m
- **obstacle_species** (`list`): the list of species that are considered as obstacles
- **pedestrian_consideration_distance** (`float`): Distance of consideration of other pedestrians (to compute the nearby obstacles, used as distance, the max between this value and (step * speed) - classic value: 3.5m
- **pedestrian_model** (`string`): Model use for the movement of agents (Social Force Model). Can be either "simple" or "advanced" (default) for different versions of SFM Helbing model
- **pedestrian_species** (`list`): the list of species that are considered as pedestrians
- **proba_detour** (`float`): probability to accept to do a detour
- **relaxion_SFM** (`float`): Value of relaxion in the SFM model - the amount of delay time for an agent to adapt.(classic values : mean = 0.54, std = 0.05)
- **roads_waypoints** (`map`): for each waypoint, the associated road

- **shoulder_length** (float): The width of the pedestrian (in meters) - classic values: [0.39, 0.515]
- **tolerance_waypoint** (float): distance to a waypoint (in meters) to consider that an agent is arrived at the waypoint
- **use_geometry_waypoint** (boolean): use geometries as waypoint instead of points
- **velocity** (point): The velocity of the pedestrian (in meters)
- **waypoints** (list): the current list of points/shape that the agent has to reach (path)

Actions

compute_virtual_path

action to compute a path to a location according to a given graph

Returned type: **path** : the computed path, return nil if no path can be taken

Additional facets:

- **pedestrian_graph** (graph): the graph on which compute the path
- **target** (geometry): the target to reach, can be any agent

Examples:

```
do compute_virtual_path graph: pedestrian_network target: any_point;
```

release_path

clean all the internal state of the agent

Returned type: **bool**

Additional facets:

- `current_road` (agent): current road on which the agent is located (can be nil)

`walk`

action to walk toward the final target using the `current_path` (requires to use the `compute_virtual_path` action before)

Returned type: `bool`

Examples:

```
do walk;
```

`walk_to`

action to walk toward a target

Returned type: `bool`

Additional facets:

- `target` (geometry): Move toward the target using the SFM model
- `bounds` (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

Examples:

```
do walk_to {10,10};
```

pedestrian_road

Variables

- `agents_on` (`list`): for each people on the road
- `exit_nodes` (`map`): The exit hub (several exit connected to each road extremities) that makes it possible to reduce angular distance when travelling to connected pedestrian roads
- `free_space` (`geometry`): for each people on the road
- `intersection_areas` (`map`): map of geometries to connect segments linked to this road
- `linked_pedestrian_roads` (`list`): the close pedestrian roads
- `road_status` (`int`): When road status equals 1 it has 2D continuous space property for pedestrian; when equal to 2 is simply a 1D road

Actions

`build_exit_hub`

Add exit hub to pedestrian corridor to reduce angular distance between node of the network

Returned type: `bool`

Additional facets:

- `pedestrian_graph` (`graph`): The pedestrian network from which to find connected corridors
- `distance_between_targets` (`float`): min distances between 2 targets

Examples:

```
do build_exit_hub pedestrian_graph: pedestrian_network  
distance_between_targets: 10.0;
```

`build_intersection_areas`

Build intersection areas with connected roads

Returned type: `bool`

Additional facets:

- `pedestrian_graph` (graph): The pedestrian network from which to find connected corridors

Examples:

```
do build_intersection_areas pedestrian_graph: pedestrian_network;
```

`initialize`

action to initialize the free space of roads

Returned type: `bool`

Additional facets:

- `distance` (float): the maximal distance to the road
- `obstacles` (container): the list of species to consider as obstacles to remove from the free space
- `distance_extremity` (float): the distance added to the extremity to connect to

other road (in meters)

- **bounds** (container): the geometries (the localized entity geometries) that restrains the agent movement (the agent moves inside this geometry)
- **masked_by** (container): if defined, keep only the part of the geometry that is visible from the location of the road considering the given obstacles
- **masked_by_precision** (int): if masked_by is defined, number of triangles used to compute the visible geometries (default: 120)
- **status** (int): the status (int) of the road: 1 (default) for roads where agent move on a continuous 2D space and 0 for 1D roads with queu-in queu-out like movement

Examples:

```
do initialize distance: 10.0 obstacles: [building];
```

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
- **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
- **num_lanes** (int): the number of lanes
- **num_segments** (int): the number of road segments
- **segment_lengths** (list): stores the length of each road segment. The index of

each element corresponds to the segment index.

- **source_node** (agent): the source node of the road
- **target_node** (agent): the target node of the road
- **vehicle_ordering** (list): provides information about the ordering of vehicle on any given lane

Actions

register

register the agent on the road at the given lane

Returned type: bool

Additional facets:

- **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

Examples:

```
do register agent: the_driver lane: 0;
```

unregister

unregister the agent on the road

Returned type: bool

Additional facets:

- **agent** (agent): the agent to unregister on the road.

Examples:

```
do unregister agent: the_driver;
```

skill_road_node

Variables

- **block** (`map`): define the list of agents blocking the node, and for each agent, the list of concerned roads
- **priority_roads** (`list`): the list of priority roads
- **roads_in** (`list`): the list of input roads
- **roads_out** (`list`): the list of output roads
- **stop** (`list`): define for each type of stop, the list of concerned roads

Actions

SQLSKILL

This skill allows agents to be provided with actions and attributes in order to connect to SQL databases

Variables

Actions

`executeUpdate`

Action used to execute any update query (CREATE, DROP, INSERT...) to the database (query written in SQL).

Returned type: `int`

Additional facets:

- `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

Examples:

```
do executeUpdate params: PARAMS updateComm: "DROP TABLE IF EXISTS registration";
do executeUpdate params: PARAMS updateComm: "INSERT INTO registration " + "VALUES(100, 'Zara', 'Ali', 18);";
do executeUpdate params: PARAMS updateComm: "INSERT INTO registration " + "VALUES(?, ?, ?, ?);" values: [101, 'Mr', 'Mme', 45];
```

`insert`

Action used to insert new data in a database

Returned type: `int`

Additional facets:

- `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

Examples:

```
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'];
```

list2Matrix

Action that transforms the list of list of data and metadata (resulting from a query) into a matrix.

Returned type: `matrix`

Additional facets:

- `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

Examples:

```
list<list> t <- list<list> (select(PARAMS, "SELECT * FROM  
registration"));
```

select

Action used to retrieve data from a database

Returned type: list

Additional facets:

- params (map): Connection parameters
- select (string): select string with question marks
- values (list): List of values that are used to replace question marks

Examples:

```
list<list> t <- list<list> (select(PARAMS, "SELECT * FROM registration"));
```

testConnection

Action used to test the connection to a database

Returned type: bool

Additional facets:

- params (map): Connection parameters

Examples:

```
if (!first(DB_Accessor).testConnection(PARAMS)) {  
    write "Connection impossible";  
    do pause;  
}
```

static_body

Variables

- **aabb** (`geometry`): The axis-aligned bounding box. A box used to evaluate the probability of contacts between objects. Can be displayed as any other GAMA shapes/geometries in order to verify that the physical representation of the agent corresponds to its geometry in the model
- **friction** (`float`): Between 0 and 1. The coefficient of friction of the agent (how much it decelerates the agents in contact with him). Default is 0.5
- **mass** (`float`): The mass of the agent. Should be equal to 0.0 for static, motionless agents
- **restitution** (`float`): Between 0 and 1. The coefficient of restitution of the agent (defines the 'bounciness' of the agent). Default is 0
- **rotation** (`pair`): The rotation of the physical body, expressed as a pair which key is the angle in degrees and value the axis around which it is measured

Actions

contact_added_with

This action can be redefined in order for the agent to implement a specific behavior when it comes into contact (collision) with another agent. It is automatically called by the physics simulation engine on both colliding agents. The default built-in behavior does nothing.

Returned type: `unknown`

Additional facets:

- **other** (`agent`): represents the other agent with which a collision has been

detected

`contact_removed_with`

This action can be redefined in order for the agent to implement a specific behavior when a previous contact with another agent is removed. It is automatically called by the physics simulation engine on both colliding agents. The default built-in behavior does nothing.

Returned type: `unknown`

Additional facets:

- `other` (agent): represents the other agent with which a collision has been detected

`update_body`

This action must be called when the geometry of the agent changes in the simulation world and this change must be propagated to the physical world. The change of location (in either worlds) or the rotation due to physical forces do not count as changes, as they are already taken into account. However, a rotation in the simulation world need to be handled by calling this action. As it involves long operations (removing the agent from the physical world, then reinserting it with its new shape), this action should not be called too often.

Returned type: `unknown`

`thread`

The thread skill is intended to define the minimal set of behaviours required for agents that are able to run an action in a thread

Variables

Actions

`end_thread`

End the current thread.

Returned type: `bool` : true if the thread was well stopped, false otherwise

Examples:

```
do end_thread;
```

`run_thread`

Start a new thread that will run the 'thread_action' either once if no facets are defined, of at a fixed rate if 'every:' is defined or with a fixed delay if 'interval:' is defined.

Returned type: `bool` : true if the thread was well created and started, false otherwise

Additional facets:

- `every` (float): Rate in machine time at which this action is run. Default unit is in seconds, use explicit units to specify another, like 10 #ms. If no rate (and no interval) is specified, the action is run once. If the action takes longer than the interval to run, it runs immediately after the previous execution
- `interval` (float): Interval -- or delay -- between two executions of the action. Default unit is in seconds, use explicit units to specify another, like 10 #ms. If no interval (and no rate) is specified, the action is run once. An interval of 0 will make the action run continuously without delays

Examples:

```
do run_thread every: 10#ms;
```

thread_action

A virtual action, which contains what to execute in the thread. It needs to be redefined in the species that implement the `thread` skill

Returned type: `unknown`

Version: 1.9.1

Built-in Architectures

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

INTRODUCTION

Table of Contents

[fsm](#), [parallel_bdi](#), [probabilistic_tasks](#), [reflex](#), [rules](#), [simple_bdi](#), [sorted_tasks](#),
[user_first](#), [user_last](#), [user_only](#), [weighted_tasks](#),

fsm

Variables

- `state` (string): Returns the name of the current state of the agent
- `states` (list): Returns the list of all the states defined in the species

Actions

parallel_bdi

compute the bdi architecture in parallel. This skill inherit all actions and variables from SimpleBdiArchitecture

Variables

Actions

probabilistic_tasks

A control architecture, based on the concept of tasks, which are executed with a probability depending on their weight. This skill extends WeightedTasksArchitecture skills and have all his actions and variables

Variables

Actions

reflex

Represents the default behavioral architecture attached to species of agents if none is specified. This skills extends AbstractArchitecture and have all his actions and variables

Variables

Actions

rules

A control architecture based on the concept of rules

Variables

Actions

simple_bdi

this architecture enables to define a behaviour using BDI. It is an implementation of the BEN architecture (Behaviour with Emotions and Norms)

Variables

- `agreeableness` (float): an agreeableness value for the personality
- `belief_base` (list): the belief base of the agent
- `charisma` (float): a charisma value. By default, it is computed with personality
- `conscientiousness` (float): a conscientiousness value for the personality
- `current_norm` (any type): the current norm of the agent
- `current_plan` (any type): the current plan of the agent

- **desire_base** (list): the desire base of the agent
- **emotion_base** (list): the emotion base of the agent
- **extroversion** (float): an extraversion value for the personality
- **ideal_base** (list): the ideal base of the agent
- **intention_base** (list): the intention base of the agent
- **intention_persistence** (float): intention persistence
- **law_base** (list): the law base of the agent
- **neurotism** (float): a neurotism value for the personality
- **norm_base** (list): the norm base of the agent
- **obedience** (float): an obedience value. By default, it is computed with personality
- **obligation_base** (list): the obligation base of the agent
- **openness** (float): an openness value for the personality
- **plan_base** (list): the plan base of the agent
- **plan_persistence** (float): plan persistence
- **probabilistic_choice** (boolean): indicates if the choice is deterministic or probabilistic
- **receptivity** (float): a receptivity value. By default, it is computed with personality
- **sanction_base** (list): the sanction base of the agent
- **social_link_base** (list): the social link base of the agent
- **thinking** (list): the list of the last thoughts of the agent
- **uncertainty_base** (list): the uncertainty base of the agent
- **use_emotions_architecture** (boolean): indicates if emotions are automatically computed
- **use_norms** (boolean): indicates if the normative engine is used
- **use_persistence** (boolean): indicates if the persistence coefficient is computed

with personality (false) or with the value given by the modeler

- `use_personality` (boolean): indicates if the personnalit is used
- `use_social_architecture` (boolean): indicates if social relations are automaticaly computed

Actions

`add_belief`

add the predicate in the belief base.

- returns: bool
- `predicate` (predicate): predicate to add as a belief
- `strength` (float): the streghth of the belief
- `lifetime` (int): the lifetime of the belief

`add_belief_emotion`

add the belief about an emotion in the belief base.

- returns: bool
- `emotion` (emotion): emotion to add as a belief
- `strength` (float): the streghth of the belief
- `lifetime` (int): the lifetime of the belief

`add_belief_mental_state`

add the predicate in the belief base.

- returns: bool
- `mental_state` (mental_state): predicate to add as a belief
- `strength` (float): the streghth of the belief

- **lifetime** (int): the lifetime of the belief

add_desire

adds the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to add as a desire
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief
- **todo** (predicate): add the desire as a subintention of this parameter

add_desire_emotion

adds the emotion in the desire base.

- returns: bool
- **emotion** (emotion): emotion to add as a desire
- **strength** (float): the strength of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (predicate): add the desire as a subintention of this parameter

add_desire_mental_state

adds the mental state is in the desire base.

- returns: bool
- **mental_state** (mental_state): mental_state to add as a desire
- **strength** (float): the strength of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (predicate): add the desire as a subintention of this parameter

`add_directly_belief`

add the belief in the belief base.

- returns: bool
- `belief` (mental_state): belief to add in th belief base

`add_directly_desire`

add the desire in the desire base.

- returns: bool
- `desire` (mental_state): desire to add in th belief base

`add_directly_ideal`

add the ideal in the ideal base.

- returns: bool
- `ideal` (mental_state): ideal to add in the ideal base

`add_directly_uncertainty`

add the uncertainty in the uncertainty base.

- returns: bool
- `uncertainty` (mental_state): uncertainty to add in the uncertainty base

`add_emotion`

add the emotion to the emotion base.

- returns: bool

- `emotion` (emotion): emotion to add to the base

`add_ideal`

add a predicate in the ideal base.

- returns: bool
- `predicate` (predicate): predicate to add as an ideal
- `praiseworthiness` (float): the praiseworthiness value of the ideal
- `lifetime` (int): the lifetime of the ideal

`add_ideal_emotion`

add a predicate in the ideal base.

- returns: bool
- `emotion` (emotion): emotion to add as an ideal
- `praiseworthiness` (float): the praiseworthiness value of the ideal
- `lifetime` (int): the lifetime of the ideal

`add_ideal_mental_state`

add a predicate in the ideal base.

- returns: bool
- `mental_state` (mental_state): mental state to add as an ideal
- `praiseworthiness` (float): the praiseworthiness value of the ideal
- `lifetime` (int): the lifetime of the ideal

`add_intention`

check if the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to check
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_intention_emotion

check if the predicates is in the desire base.

- returns: bool
- **emotion** (emotion): emotion to add as an intention
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_intention_mental_state

check if the predicates is in the desire base.

- returns: bool
- **mental_state** (mental_state): predicate to add as an intention
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_obligation

add a predicate in the ideal base.

- returns: bool
- **predicate** (predicate): predicate to add as an obligation
- **strength** (float): the strength value of the obligation
- **lifetime** (int): the lifetime of the obligation

`add_social_link`

add the social link to the social link base.

- returns: bool
- `social_link` (social_link): social link to add to the base

`add_subintention`

adds the predicates is in the desire base.

- returns: bool
- `predicate` (mental_state): the intention that receives the sub_intention
- `subintentions` (predicate): the predicate to add as a subintention to the intention
- `add_as_desire` (boolean): add the subintention as a desire as well (by default, false)

`add_uncertainty`

add a predicate in the uncertainty base.

- returns: bool
- `predicate` (predicate): predicate to add
- `strength` (float): the strength of the belief
- `lifetime` (int): the lifetime of the belief

`add_uncertainty_emotion`

add a predicate in the uncertainty base.

- returns: bool

- **emotion** (emotion): emotion to add as an uncertainty
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_uncertainty_mental_state

add a predicate in the uncertainty base.

- returns: bool
- **mental_state** (mental_state): mental state to add as an uncertainty
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

change_dominance

changes the dominance value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with whom I get a social link
- **dominance** (float): a value to change the dominance value

change_familiarity

changes the familiarity value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with whom I get a social link
- **familiarity** (float): a value to change the familiarity value

change_liking

changes the liking value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with who I get a social link
- **liking** (float): a value to change the liking value

change_solidarity

changes the solidarity value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with who I get a social link
- **solidarity** (float): a value to change the solidarity value

change_trust

changes the trust value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with who I get a social link
- **trust** (float): a value to change the trust value

clear_beliefs

clear the belief base

- returns: bool

clear_desires

clear the desire base

- returns: bool

clear_emotions

clear the emotion base

- returns: bool

clear_ideals

clear the ideal base

- returns: bool

clear_intentions

clear the intention base

- returns: bool

clear_obligations

clear the obligation base

- returns: bool

clear_social_links

clear the intention base

- returns: bool

clear_uncertainties

clear the uncertainty base

- returns: bool

`current_intention_on_hold`

puts the current intention on hold until the specified condition is reached or all subintentions are reached (not in desire base anymore).

- returns: bool
- `until` (any type): the current intention is put on hold (fited plan are not considered) until specific condition is reached. Can be an expression (which will be tested), a list (of subintentions), or nil (by default the condition will be the current list of subintentions of the intention)

`get_belief`

return the belief about the predicate in the belief base (if several, returns the first one).

- returns: mental_state
- `predicate` (predicate): predicate to get

`get_belief_emotion`

return the belief about the emotion in the belief base (if several, returns the first one).

- returns: mental_state
- `emotion` (emotion): emotion about which the belief to get is

`get_belief_mental_state`

return the belief about the mental state in the belief base (if several, returns the first one).

- returns: mental_state

- **mental_state** (mental_state): mental state to get

get_belief_with_name

get the predicates is in the belief base (if several, returns the first one).

- returns: mental_state
- **name** (string): name of the predicate to check

get_beliefs

get the list of predicates in the belief base

- returns: list<mental_state>
- **predicate** (predicate): predicate to check

get_beliefs_mental_state

get the list of beliefs in the belief base containing the mental state

- returns: list<mental_state>
- **mental_state** (mental_state): mental state to check

get_beliefs_with_name

get the list of predicates is in the belief base with the given name.

- returns: list<mental_state>
- **name** (string): name of the predicates to check

get_current_intention

returns the current intention (last entry of intention base).

- returns: mental_state

`get_current_plan`

get the current plan.

- returns: BDIPlan

`get_desire`

get the predicates is in the desire base (if several, returns the first one).

- returns: mental_state
- `predicate` (predicate): predicate to check

`get_desire_mental_state`

get the mental state is in the desire base (if several, returns the first one).

- returns: mental_state
- `mental_state` (mental_state): mental state to check

`get_desire_with_name`

get the predicates is in the belief base (if several, returns the first one).

- returns: mental_state
- `name` (string): name of the predicate to check

`get_desires`

get the list of predicates is in the desire base

- returns: list<mental_state>

- **predicate** (predicate): name of the predicates to check

get_desires_mental_state

get the list of mental states is in the desire base

- returns: list<mental_state>
- **mental_state** (mental_state): name of the mental states to check

get_desires_with_name

get the list of predicates is in the belief base with the given name.

- returns: list<mental_state>
- **name** (string): name of the predicates to check

get_emotion

get the emotion in the emotion base (if several, returns the first one).

- returns: emotion
- **emotion** (emotion): emotion to get

get_emotion_with_name

get the emotion is in the emotion base (if several, returns the first one).

- returns: emotion
- **name** (string): name of the emotion to check

get_ideal

get the ideal about the predicate in the ideal base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to check ad an ideal

`get_ideal_mental_state`

get the mental state in the ideal base (if several, returns the first one).

- returns: mental_state
- **mental_state** (mental_state): mental state to return

`get_intention`

get the predicates in the intention base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to check

`get_intention_mental_state`

get the mental state is in the intention base (if several, returns the first one).

- returns: mental_state
- **mental_state** (mental_state): mental state to check

`get_intention_with_name`

get the predicates is in the belief base (if several, returns the first one).

- returns: mental_state
- **name** (string): name of the predicate to check

`get_intentions`

get the list of predicates is in the intention base

- returns: list<mental_state>
- **predicate** (predicate): name of the predicates to check

get_intentions_mental_state

get the list of mental state is in the intention base

- returns: list<mental_state>
- **mental_state** (mental_state): mental state to check

get_intentions_with_name

get the list of predicates is in the belief base with the given name.

- returns: list<mental_state>
- **name** (string): name of the predicates to check

get_obligation

get the predicates in the obligation base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to return

get_plan

get the first plan with the given name

- returns: BDIPlan
- **name** (string): the name of the planto get

get_plans

get the list of plans.

- returns: list<BDIPlan>

`get_social_link`

get the social link (if several, returns the first one).

- returns: social_link
- `social_link` (social_link): social link to check

`get_social_link_with_agent`

get the social link with the agent concerned (if several, returns the first one).

- returns: social_link
- `agent` (agent): an agent with who I get a social link

`get_uncertainty`

get the predicates is in the uncertainty base (if several, returns the first one).

- returns: mental_state
- `predicate` (predicate): predicate to return

`get_uncertainty_mental_state`

get the mental state is in the uncertainty base (if several, returns the first one).

- returns: mental_state
- `mental_state` (mental_state): mental state to return

`has_belief`

check if the predicates is in the belief base.

- returns: bool
- **predicate** (predicate): predicate to check

has_belief_mental_state

check if the mental state is in the belief base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_belief_with_name

check if the predicate is in the belief base.

- returns: bool
- **name** (string): name of the predicate to check

has_desire

check if the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to check

has_desire_mental_state

check if the mental state is in the desire base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_desire_with_name

check if the prediate is in the desire base.

- returns: bool
- **name** (string): name of the predicate to check

has_emotion

check if the emotion is in the belief base.

- returns: bool
- **emotion** (emotion): emotion to check

has_emotion_with_name

check if the emotion is in the emotion base.

- returns: bool
- **name** (string): name of the emotion to check

has_ideal

check if the predicates is in the ideal base.

- returns: bool
- **predicate** (predicate): predicate to check

has_ideal_mental_state

check if the mental state is in the ideal base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_ideal_with_name

check if the predicate is in the ideal base.

- returns: bool
- **name** (string): name of the predicate to check

has_obligation

check if the predicates is in the obligation base.

- returns: bool
- **predicate** (predicate): predicate to check

has_social_link

check if the social link base.

- returns: bool
- **social_link** (social_link): social link to check

has_social_link_with_agent

check if the social link base.

- returns: bool
- **agent** (agent): an agent with who I want to check if I have a social link

has_uncertainty

check if the predicates is in the uncertainty base.

- returns: bool
- **predicate** (predicate): predicate to check

has_uncertainty_mental_state

check if the mental state is in the uncertainty base.

- returns: bool
- `mental_state` (mental_state): mental state to check

`has_uncertainty_with_name`

check if the predicate is in the uncertainty base.

- returns: bool
- `name` (string): name of the uncertainty to check

`is_current_intention`

check if the predicates is the current intention (last entry of intention base).

- returns: bool
- `predicate` (predicate): predicate to check

`is_current_intention_mental_state`

check if the mental state is the current intention (last entry of intention base).

- returns: bool
- `mental_state` (mental_state): mental state to check

`is_current_plan`

tell if the current plan has the same name as tested

- returns: bool
- `name` (string): the name of the plan to test

`remove_all_beliefs`

removes the predicates from the belief base.

- returns: bool
- **predicate** (predicate): predicate to remove

`remove_belief`

removes the predicate from the belief base.

- returns: bool
- **predicate** (predicate): predicate to remove

`remove_belief_mental_state`

removes the mental state from the belief base.

- returns: bool
- **mental_state** (mental_state): mental state to remove

`remove_desire`

removes the predicates from the desire base.

- returns: bool
- **predicate** (predicate): predicate to remove from desire base

`remove_desire_mental_state`

removes the mental state from the desire base.

- returns: bool
- **mental_state** (mental_state): mental state to remove from desire base

`remove_emotion`

removes the emotion from the emotion base.

- returns: bool
- **emotion** (emotion): emotion to remove

remove_ideal

removes the predicates from the ideal base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_ideal_mental_state

removes the mental state from the ideal base.

- returns: bool
- **mental_state** (mental_state): metal state to remove

remove_intention

removes the predicates from the intention base.

- returns: bool
- **predicate** (predicate): intention's predicate to remove
- **desire_also** (boolean): removes also desire

remove_intention_mental_state

removes the mental state from the intention base.

- returns: bool
- **mental_state** (mental_state): intention's mental state to remove
- **desire_also** (boolean): removes also desire

`remove_obligation`

removes the predicates from the obligation base.

- returns: bool
- `predicate` (predicate): predicate to remove

`remove_social_link`

removes the social link from the social relation base.

- returns: bool
- `social_link` (social_link): social link to remove

`remove_social_link_with_agent`

removes the social link from the social relation base.

- returns: bool
- `agent` (agent): an agent with who I get the social link to remove

`remove_uncertainty`

removes the predicates from the uncertainty base.

- returns: bool
- `predicate` (predicate): predicate to remove

`remove_uncertainty_mental_state`

removes the mental state from the uncertainty base.

- returns: bool

- `mental_state` (mental_state): mental state to remove

`replace_belief`

replace the old predicate by the new one.

- returns: bool
 - `old_predicate` (predicate): predicate to remove
 - `predicate` (predicate): predicate to add
-

sorted_tasks

A control architecture, based on the concept of tasks, which are executed in an order defined by their weight. This skill extends the WeightedTasksArchitecture skill and take all his actions and variables

Variables

Actions

user_first

A control architecture, based on FSM, where the user is being given control before states / reflexes of the agent are executed. This skill extends the UserControlArchitecture skill and take all his actions and variables

Variables

Actions

user_last

A control architecture, based on FSM, where the user is being given control after states / reflexes of the agent are executed. This skill extends the UserControlArchitecture skill and take all his actions and variables

Variables

Actions

user_only

A control architecture, based on FSM, where the user is being given complete control of the agents. This skill extends the UserControlArchitecture skill and take all his actions and variables

Variables

Actions

weighted_tasks

The class WeightedTasksArchitecture. A simple architecture of competing tasks, where one can be active at a time. Weights of the tasks are computed every step and the chosen task is simply the one with the maximal weight

Variables

Actions

Version: 1.9.1

Statements

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

Table of Contents

=, abort, action, add, agents, annealing, ask, aspect, assert, benchmark, betad, break, browse, camera, capture, catch, category, chart, conscious_contagion, continue, coping, create, data, datalist, default, diffuse, diffusion, display, display_grid, do, draw, else, emotional_contagion, enforcement, enter, equation, error, event, exit, experiment, exploration, focus, focus_on, generate, genetic, global, graphics, grid, highlight, hill_climbing, if, image_layer, init, inspect, invoke, law, layout, let, light, loop, match, match_between, match_one, match_regex, mesh, migrate, monitor, morris, norm, output, output_file, overlay, parameter, perceive, permanent, plan, pso, put, reactive_tabu, reflex, release, remove, return, rotation, rule, rule, run, sanction, save, set, setup, sobol, socialize, solve, species, species_layer, start_simulation, state, status, stochanalyse, switch, tabu, task, test, text, trace, transition, try, unconscious_contagion, user_command, user_init, user_input, user_panel, using, Variable_container, Variable_number, Variable_regular, warn, write,

Statements by kinds

- **Batch method**
 - annealing, betad, exploration, genetic, hill_climbing, morris, pso, reactive_tabu, sobol, stochanalyse, tabu,
- **Behavior**
 - abort, aspect, coping, init, norm, plan, reflex, rule, sanction, state, task, test, user_init, user_panel,
- **Experiment**
 - experiment,

- **Layer**
 - agents, camera, chart, display_grid, event, graphics, image_layer, light, mesh, overlay, rotation, species_layer,
- **Output**
 - browse, display, inspect, layout, monitor, output, output_file, permanent,
- **Parameter**
 - parameter,
- **Sequence of statements or action**
 - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, generate, if, loop, match, match_between, match_one, match_regex, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,
- **Single statement**
 - =, add, assert, break, category, conscious_contagion, continue, data, datalist, diffuse, diffusion, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, invoke, law, let, put, remove, return, rule, save, set, socialize, solve, status, text, unconscious_contagion, user_input, warn, write,
- **Species**
 - global, grid, species,
- **Variable (container)**
 - Variable_container,
- **Variable (number)**
 - Variable_number,
- **Variable (regular)**
 - Variable_regular,

Statements by embedment

- **Behavior**
 - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, generate, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Environment**

- species,
- **Experiment**
 - action, annealing, betad, category, exploration, genetic, hill_climbing, morris, output, parameter, permanent, pso, reactive_tabu, reflex, rule, setup, sobol, state, stochanalyse, tabu, task, test, text, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Layer**
 - add, ask, benchmark, draw, error, focus_on, highlight, if, let, loop, put, remove, set, status, switch, trace, try, using, warn, write,
- **Model**
 - action, aspect, coping, equation, experiment, law, norm, output, perceive, plan, reflex, rule, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **Output**
 - ask, if,
- **Sequence of statements or action**
 - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, continue, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, generate, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, socialize, solve, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Single statement**
 - run, start_simulation,
- **Species**
 - action, aspect, coping, equation, law, norm, perceive, plan, reflex, rule, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,
- **action**
 - assert, return,
- **aspect**
 - draw,
- **chart**
 - add, ask, data, datalist, do, put, remove, set, using,
- **display**
 - agents, camera, chart, display_grid, event, graphics, image_layer, light, mesh, overlay,

- rotation, species_layer,
- **equation**
 - =,
- **fsm**
 - state, user_panel,
- **if**
 - else,
- **output**
 - display, inspect, layout, monitor, output_file,
- **parallel_bdi**
 - coping, rule,
- **permanent**
 - display, inspect, monitor, output_file,
- **probabilistic_tasks**
 - task,
- **rules**
 - rule,
- **simple_bdi**
 - coping, rule,
- **sorted_tasks**
 - task,
- **species_layer**
 - species_layer,
- **state**
 - enter, exit,
- **switch**
 - default, match,
- **test**
 - assert,
- **try**
 - catch,
- **user_command**
 - user_input,
- **user_first**

- user_panel,
- **user_init**
 - user_panel,
- **user_last**
 - user_panel,
- **user_only**
 - user_panel,
- **user_panel**
 - user_command,
- **weighted_tasks**
 - task,

General syntax

A statement represents either a declaration or an imperative command. It consists in a keyword, followed by specific facets, some of them mandatory (in bold), some of them optional. One of the facet names can be omitted (the one denoted as *omissible*). It has to be the first one.

```
statement_keyword expression1 facet2: expression2 ... ;
or
statement_keyword facet1: expression1 facet2: expression2 ...;
```

If the statement encloses other statements, it is called a **sequence statement**, and its sub-statements (either sequence statements or single statements) are declared between curly brackets, as in:

```
statement_keyword1 expression1 facet2: expression2... { // a sequence statement
    statement_keyword2 expression1 facet2: expression2...; // a single statement
    statement_keyword3 expression1 facet2: expression2...;
}
```

=

Facets

- **right** (float), (omissible) : the right part of the equation (it is mandatory that it can be evaluated as a float)
- **left** (any type): the left part of the equation (it should be a variable or a call to the diff() or diff2() operators)

Definition

Allows to implement an equation in the form function(n, t) = expression. The left function is only here as a placeholder for enabling a simpler syntax and grabbing the variable as its left member.

Usages

- The syntax of the = statement is a bit different from the other statements. It has to be used as follows (in an equation):

```
float t;
float S;
float I;
equation SI {
    diff(S,t) = (- 0.3 * S * I / 100);
    diff(I,t) = (0.3 * S * I / 100);
}
```

- See also: [equation](#), [solve](#),

Embedments

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

action

Facets

- `name` (an identifier), (omissible) : identifier of the action
- `index` (a datatype identifier): if the action returns a map, the type of its keys
- `of` (a datatype identifier): if the action returns a container, the type of its elements
- `type` (a datatype identifier): the action returned type
- `virtual` (boolean): whether the action is virtual (defined without a set of instructions) (false by default)

Definition

Allows to define in a species, model or experiment a new action that can be called elsewhere.

Usages

- The simplest syntax to define an action that does not take any parameter and does not return anything is:

```
action simple_action {  
    // [set of statements]  
}
```

- If the action needs some parameters, they can be specified 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]
```

- If virtual: is true, then the action is abstract, which means that the action is defined without body. A species containing at least one abstract action is abstract. Agents of this species cannot be created. The common use of an abstract action is to define an action that can be used by all its sub-species, which should redefine all abstract actions and implements its body.

```

species parent_species {
    int virtual_action(int i, string s);
}

species children parent: parent_species {
    int virtual_action(int i, string s) {
        return i + i;
    }
}

```

- See also: [do](#),

Embedments

- The `action` statement is of type: **Sequence of statements or action**
 - The `action` statement can be embedded into: Species, Experiment, Model,
 - The `action` statement embeds statements: [assert](#), [return](#),
-

add

Facets

- `to` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression to add in the container
- `all` (any type): Allows to either pass a container so as to add all its element, or 'true', if the item to add is already a container.
- `at` (any type): position in the container of added element

Definition

Allows to add, i.e. to insert, a new element in a container (a list, matrix, map, ...). Incorrect use: The

addition of a new element at a position out of the bounds of the container will produce a warning and let the container unmodified. If all: is specified, it has no effect if its argument is not a container, or if its argument is 'true' and the item to add is not a container. In that latter case

Usages

- The new element can be added either at the end of the container or at a particular position.

```
add expr to: expr_container;      // Add at the end
add expr at: expr to: expr_container; // Add at position expr
```

- Case of a list, the expression in the facet at: should be an integer.

```
list<int> workingList <- [];
add 0 at: 0 to: workingList ; // workingList equals [0]
add 10 at: 0 to: workingList ; // workingList equals [10,0]
add 20 at: 2 to: workingList ; // workingList equals [10,0,20]
add 50 to: workingList; // workingList equals [10,0,20,50]
add [60,70] all: true to: workingList; // workingList equals [10,0,20,50,60,70]
```

- Case of a map: As a map is basically a list of pairs key::value, we can also use the add statement on it. It is important to note that the behavior of the statement is slightly different, in particular in the use of the at facet, which denotes the key of the pair.

```
map<string,string> workingMap <- [];
add "val1" at: "x" to: workingMap; // workingMap equals ["x":"val1"]
```

- If the at facet is omitted, a pair (expr_item::expr_item) will be added to the map. An important exception is the case where the expr_item is a pair: in this case the pair is added.

```
add "val2" to: workingMap; // workingMap equals ["x":"val1", "val2":"val2"]
add "5":"val4" to: workingMap; // workingMap equals ["x":"val1",
"val2":"val2", "5":"val4"]
```

- Notice that, as the key should be unique, the addition of an item at an existing position (i.e. existing key) will only modify the value associated with the given key.

```
add "val3" at: "x" to: workingMap; // workingMap equals ["x":"val3",  
"val2":"val2", "5":"val4"]
```

- On a map, the all facet will add all value of a container in the map (so as pair val_cont::val_cont)

```
add ["val4","val5"] all: true at: "x" to: workingMap; // workingMap equals  
["x":"val3", "val2":"val2", "5":"val4","val4":"val4","val5":"val5"]
```

- In case of a graph, we can use the facets node, edge and weight to add a node, an edge or weights to the graph. However, these facets are now considered as deprecated, and it is advised to use the various edge(), node(), edges(), nodes() operators, which can build the correct objects to add to the graph

```
graph g <- as_edge_graph([{1,5}:{12,45}]);  
add edge: {1,5}:{2,3} to: g;  
list var <- g.vertices; // var equals [{1,5},{12,45},{2,3}]  
list var <- g.edges; // var equals  
[polyline({1.0,5.0}:{12.0,45.0}),polyline({1.0,5.0}:{2.0,3.0})]  
add node: {5,5} to: g;  
list var <- g.vertices; // var equals [{1.0,5.0},{12.0,45.0},{2.0,3.0},{5.0,5.0}]  
list var <- g.edges; // var equals  
[polyline({1.0,5.0}:{12.0,45.0}),polyline({1.0,5.0}:{2.0,3.0})]
```

- Case of a matrix: this statement can not be used on matrix. Please refer to the statement put.
- See also: [put](#), [remove](#),

Embedments

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

agents

Facets

- `value` (container): the set of agents to display
- `name` (a label), (omissible) : Human readable title of the layer
- `aspect` (an identifier): the name of the aspect that should be used to display the species
- `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false
- `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 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- `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)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `visible` (boolean): Defines whether this layer is visible or not

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_layer](#), [overlay](#), [species_layer](#),

Embedments

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

annealing

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the aggregation method
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `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. At each iteration, the current temperature is multiplied by this coefficient.

- `temp_end` (float): final temperature
- `temp_init` (float): initial temperature

Definition

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the annealing statement uses `method annealing` instead of the expected `annealing name: id`:

```
method annealing [facet: value];
```

- For example:

```
method annealing temp_init: 100  temp_end: 1 temp_decrease: 0.5 nb_iter_cst_temp:  
5 maximize: food_gathered;
```

Embedments

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

ask

Facets

- `target` (any type in [container, agent]), (omissible) : an expression that evaluates to an agent or a list of agents
- `as` (species): an expression that evaluates to a species
- `parallel` (any type in [boolean, int]): (experimental) setting this facet to 'true' will allow 'ask' to use concurrency when traversing the targets; setting it to an integer will set the threshold

under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is false by default.

Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. If the value of the target facet is nil or empty, the statement is ignored.

Usages

- Ask a set of receiver agents, stored in a container, to perform a block of statements. The block is evaluated in the context of the agents' species

```
ask ${receiver_agents} {  
    ${cursor}  
}
```

- Ask one agent to perform a block of statements. The block is evaluated in the context of the agent's species

```
ask ${one_agent} {  
    ${cursor}  
}
```

- If the species of the receiver agent(s) cannot be determined, it is possible to force it using the `as` facet. An error is thrown if an agent is not a direct or undirect instance of this species

```
ask ${receiver_agent(s)} as: ${a_species_expression} {  
    ${cursor}  
}
```

- To ask a set of agents to do something only if they belong to a given species, the `of_species` operator can be used. If none of the agents belong to the species, nothing happens

```
ask ${receiver_agents} of_species ${species_name} {  
    ${cursor}
```

- Any statement can be declared in the block statements. All the statements will be evaluated in the context of the receiver agent(s), as if they were defined in their species, which means that an expression like `self` will represent the receiver agent and not the sender. If the sender needs to refer to itself, some of its own attributes (or temporary variables) within the block statements, it has to use the keyword `myself`.

```

species animal {
    float energy <- rnd (1000) min: 0.0;
    reflex when: energy > 500 { // executed when the energy is above the given
threshold
        list<animal> others <- (animal at_distance 5); // find all the
neighboring animals in a radius of 5 meters
        float shared_energy <- (energy - 500) / length (others); // compute the
amount of energy to share with each of them
        ask others { // no need to cast, since others has already been filtered
to only include animals
            if (energy < 500) { // refers to the energy of each animal in others
                energy <- energy + myself.shared_energy; // increases the
energy of each animal
                myself.energy <- myself.energy - myself.shared_energy; // decreases the
energy of the sender
            }
        }
    }
}

```

- If the species of the receiver agent cannot be determined, it is possible to force it by casting the agent. Nothing happens if the agent cannot be casted to this species

Embedments

- The `ask` statement is of type: **Sequence of statements or action**
 - The `ask` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer, Output,
 - The `ask` statement embeds statements:
-

aspect

Facets

- `name` (an identifier), (omissible) : identifier of the aspect (it can be used in a display to identify which aspect should be used for the given species). Two special names can also be used: 'default' will allow this aspect to be used as a replacement for the default aspect defined in preferences; 'highlighted' will allow the aspect to be used when the agent is highlighted as a replacement for the default (application of a color)

Definition

Aspect statement is used to define a way to draw the current agent. Several aspects can be defined in one species. It can use attributes to customize each agent's aspect. The aspect is evaluated for each agent each time it has to be displayed.

Usages

- An example of use of the aspect statement:

```
species one_species {
    int a <- rnd(10);
    aspect aspect1 {
        if(a mod 2 = 0) { draw circle(a);}
        else {draw square(a);}
        draw text: "a= " + a color: #black size: 5;
    }
}
```

Embedments

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

assert

Facets

- `value` (boolean), (omissible) : a boolean expression. If its evaluation is true, the assertion is successful. Otherwise, an error (or a warning) is raised.
- `label` (string): a string displayed instead of the failed expression in order to customize the error or warning if the assertion is false
- `warning` (boolean): if set to true, makes the assertion emit a warning instead of an error

Definition

Allows to check if the evaluation of a given expression returns true. If not, an error (or a warning) is raised. If the statement is used inside a test, the error is not propagated but invalidates the test (in case of a warning, it partially invalidates it). Otherwise, it is normally propagated

Usages

- Any boolean expression can be used

```
assert (2+2) = 4;
assert self != nil;
int t <- 0; assert is_error(3/t);
(1 / 2) is float
```

- if the 'warn:' facet is set to true, the statement emits a warning (instead of an error) in case the expression is false

```
assert 'abc' is string warning: true
```

- See also: [test](#), [setup](#), [is_error](#), [is_warning](#),

Embedments

- The `assert` statement is of type: **Single statement**
- The `assert` statement can be embedded into: test, action, Sequence of statements or action, Behavior, Sequence of statements or action,

- The `assert` statement embeds statements:
-

benchmark

Facets

- `message` (any type), (omissible) : A message to display alongside the results. Should concisely describe the contents of the benchmark
- `repeat` (int): An int expression describing how many executions of the block must be handled. The output in this case will return the min, max and average durations

Definition

Displays in the console the duration in ms of the execution of the statements included in the block. It is possible to indicate, with the 'repeat' facet, how many times the sequence should be run

Usages

Embedments

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

betad

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `outputs` (list): The list of output variables to analyse
- `report` (string): The path to the file where the Betad report will be written
- `sampling` (an identifier): The sampling method to build parameters sets that must be factorial based to some extends - available are saltelli and default uniform
- `factorial` (list): The number of sample required.

- `results` (string): The path to the file where the automatic batch report will be written
- `sample` (int): The number of sample required.

Definition

This algorithm runs an exploration with a given sampling to compute BetadKu - see doi: 10.1007/s10588-021-09358-5

Usages

- For example:

```
method sobol sample_size:100 outputs:['my_var'] report:'.../path/to/report/file.txt';
```

Embedments

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

break

Facets

Definition

`break` allows to interrupt the current sequence of statements.

Usages

Embedments

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

camera

Facets

- `name` (string), (omissible) : The name of the camera. Will be used to populate a menu with the other camera presets. Can provide a value to the 'camera:' facet of the display, which specifies which camera to use. Using the special constant #default will make it the default of the surrounding display
- `distance` (float): If the 'location:' facet is not defined, defines the distance (in world units) that separates the camera from its target. If 'location:' is defined, especially if it is using a symbolic position, allows to specify the distance to keep from the target. If neither 'location:' or 'distance:' is defined, the default distance is the maximum between the width and the height of the world
- `dynamic` (boolean): If true, the location, distance and target are automatically recomputed every step. Default is false. When true, will also set 'locked' to true, to avoid interferences from users
- `lens` (any type in [float, int]): Allows to define the lens -- field of view in degrees -- of the camera. Between 0 and 360. Defaults to 45°
- `location` (any type in [point, string]): Allows to define the location of the camera in the world, i.e. from where it looks at its target. If 'distance:' is specified, the final location is translated on the target-camera axis to respect the distance. Can be a (possibly dynamically computed) point or a symbolic position (#from_above, #from_left, #from_right, #from_up_right, #from_up_left, #from_front, #from_up_front) that will be dynamically recomputed if the target moves. If 'location:' is not defined, it will be that of the default camera (#from_top, #from_left...) defined in the preferences.
- `locked` (boolean): If true, the user cannot modify the camera location and target by interacting with the display. It is automatically set when the camera is dynamic, so that the display can 'follow' the coordinates; but it can also be used with fixed coordinates to 'focus' the display on a specific scene
- `target` (any type in [point, agent, geometry]): Allows to define the target of the camera (what does it look at). It can be a point (in world coordinates), a geometry or an agent, in which case its (possibly dynamic) location it used as the target. This facet can be complemented by 'distance:' and/or 'location:' to specify from where the target is looked at. If 'target:' is not defined, the default target is the centroid of the world shape.

Definition

`camera` allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way. Several preset cameras are provided and accessible in the preferences (to choose the default) or in GAML using the keywords `#from_above`, `#from_left`, `#from_right`, `#from_up_right`, `#from_up_left`, `#from_front`, `#from_up_front`, `#isometric`. These cameras are unlocked (so that they can be manipulated by the user), look at the center of the world from a symbolic position, and the distance between this position and the target is equal to the maximum of the width and height of the world's shape. These preset cameras can be reused when defining new cameras, since their names can become symbolic positions for them. For instance: `camera 'my_camera' location: #from_top distance: 10;` will lower (or extend) the distance between the camera and the center of the world to 10. `camera 'my_camera' locked: true location: #from_up_front target: people(0);` will continuously follow the first agent of the people species from the up-front position.

Usages

- See also: [display](#), [agents](#), [chart](#), [event](#), [graphics](#), [display_grid](#), [image_layer](#), [species_layer](#),

Embedments

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

capture

Facets

- `target` (any type in [agent, container]), (omissible) : an expression that is evaluated as an agent or a list of the agent to be captured
- `as` (species): the species that the captured agent(s) will become, this is a micro-species of the calling agent's species
- `returns` (a new identifier): a list of the newly captured agent(s)

Definition

Allows an agent to capture other agent(s) as its micro-agent(s).

Usages

- The preliminary for an agent A to capture an agent B as its micro-agent is that the A's species must defined a micro-species which is a sub-species of B's species (cf. [Species161#Nesting_species Nesting species]).

```
species A {  
...  
}  
species B {  
...  
    species C parent: A {  
        ...  
    }  
...  
}
```

- To capture all "A" agents as "C" agents, we can ask an "B" agent to execute the following statement:

```
capture list(B) as: C;
```

- Deprecated writing:

```
capture target: list (B) as: C;
```

- See also: [release](#),

Embedments

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

catch

Facets

Definition

This statement cannot be used alone

Usages

- See also: [try](#),

Embedments

- The `catch` statement is of type: **Sequence of statements or action**
 - The `catch` statement can be embedded into: `try`,
 - The `catch` statement embeds statements:
-

category

Facets

- `name` (a label), (omissible) : The title of the category displayed in the UI
- `color` (rgb): The background color of the category in the UI
- `expanded` (boolean): Whether the category is initially expanded or not

Definition

Allows to define a category of parameters that will serve to group parameters in the UI. The category can be declared as initially expanded or closed (overriding the corresponding preference) and with a background color

Usages

Embedments

- The `category` statement is of type: **Single statement**
 - The `category` statement can be embedded into: Experiment,
 - The `category` statement embeds statements:
-

chart

Facets

- `name` (string), (omissible) : the identifier of the chart layer
- `axes` (rgb): the axis color
- `background` (rgb): the background color
- `color` (rgb): Text color
- `gap` (float): minimum gap between bars (in proportion)
- `label_background_color` (rgb): Color of the label background (for Pie chart)
- `label_font` (any type in [string, font]): Label font face. Either the name of a font face or a font
- `label_text_color` (rgb): Color of the label text (for Pie chart)
- `legend_font` (any type in [string, font]): Legend font face. Either the name of a font face or a font
- `memorize` (boolean): Whether or not to keep the values in memory (in order to produce a csv file, for instance). The default value, true, can also be changed in the preferences
- `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 for x and y. The z-coordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer.
- `reverse_axes` (boolean): reverse X and Y axis (for example to get horizontal bar charts)
- `series_label_position` (an identifier), takes values in: {default, none, legend, onchart, yaxis, xaxis}: Position of the Series names: default (best guess), none, legend, onchart, xaxis (for category plots) or yaxis (uses the first serie name).

- `size` (point): the layer resize factor: {1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer. In case of a 3D layer, a 3D point can be used (note that {1,1} is equivalent to {1,1,0}, so a resize of a layer containing 3D objects with a 2D points will remove the elevation)
- `style` (an identifier), takes values in: {line, area, bar, dot, step, spline, stack, 3d, ring, exploded, default}: The sub-style style, also default style for the series.
- `tick_font` (any type in [string, font]): Tick font face. Either the name of a font face or a font. When used for a series chart, it will set the font of values on the axes, but When used with a pie, it will modify the font of messages associated to each pie section.
- `tick_line_color` (rgb): the tick lines color
- `title_font` (any type in [string, font]): Title font face. Either the name of a font face or a font
- `title_visible` (boolean): chart title visible
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `type` (an identifier), takes values in: {xy, scatter, histogram, series, pie, radar, heatmap, box_whisker}: the type of chart. It could be histogram, series, xy, pie, radar, heatmap or box whisker. The difference between series and xy is that the former adds an implicit x-axis that refers to the numbers of cycles, while the latter considers the first declaration of data to be its x-axis.
- `visible` (boolean): Defines whether this layer is visible or not
- `x_label` (string): the title for the X axis
- `x_log_scale` (boolean): use Log Scale for X axis
- `x_range` (any type in [float, int, point, list]): range of the x-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `x_serie` (any type in [list, float, int]): for series charts, change the default common x serie (simulation cycle) for an other value (list or numerical).
- `x_serie_labels` (any type in [list, float, int, a label]): change the default common x series labels (replace x value or categories) for an other value (string or numerical).
- `x_tick_line_visible` (boolean): X tick line visible
- `x_tick_unit` (float): the tick unit for the y-axis (distance between horizontal lines and values on the left of the axis).
- `x_tick_values_visible` (boolean): X tick values visible
- `y_label` (string): the title for the Y axis

- `y_log_scale` (boolean): use Log Scale for Y axis
- `y_range` (any type in [float, int, point, list]): range of the y-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `y_serie_labels` (any type in [list, float, int, a label]): for heatmaps/3d charts, change the default y serie for an other value (string or numerical in a list or cumulative).
- `y_tick_line_visible` (boolean): Y tick line visible
- `y_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).
- `y_tick_values_visible` (boolean): Y tick values visible
- `y2_label` (string): the title for the second Y axis
- `y2_log_scale` (boolean): use Log Scale for second Y axis
- `y2_range` (any type in [float, int, point, list]): range of the second 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).
- `y2_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).

Definition

`chart` allows modeler to display a chart: this enables to display specific values of the model at each iteration. GAMA can display various chart types: time series (series), pie charts (pie) and histograms (histogram).

Usages

- The general syntax is:

```
display chart_display {
    chart "chart name" type: series [additional options] {
        [Set of data, datalists statements]
    }
}
```

- See also: `display`, `agents`, `event`, `graphics`, `display_grid`, `image_layer`, `overlay`, `quadtree`, `species_layer`, `text`,

Embedments

- The `chart` statement is of type: **Layer**
 - The `chart` statement can be embedded into: `display`,
 - The `chart` statement embeds statements: `add`, `ask`, `data`, `datalist`, `do`, `put`, `remove`, `set`, `using`,
-

conscious_contagion

Facets

- `emotion_created` (emotion): the emotion that will be created with the contagion
- `emotion_detected` (emotion): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `intensity` (float): The intensity value of the emotion added to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly add an emotion of a perceived species if the perceived agent gets a particular emotion.

Usages

- Other examples of use:

```
conscious_contagion emotion_detected:fear emotion_created:fearConfirmed;  
conscious_contagion emotion_detected:fear emotion_created:fearConfirmed charisma:  
0.5 receptivity: 0.5;
```

Embedments

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

continue

Facets

Definition

`continue` allows to skip the remaining statements inside a loop and an ask and directly move to the next element. Inside a switch, it has the same effect as break.

Usages

Embedments

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

coping

Facets

- `name` (an identifier), (omissible) : The name of the rule
- `belief` (predicate): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `desire` (predicate): The mandatory desire
- `desires` (list): The mandatory desires
- `emotion` (emotion): The mandatory emotion

- `emotions` (list): The mandatory emotions
- `ideal` (predicate): The mandatory ideal
- `ideals` (list): The mandatory ideals
- `lifetime` (int): the lifetime value of the mental state created
- `new_belief` (predicate): The belief that will be added
- `new_beliefs` (list): The belief that will be added
- `new_desire` (predicate): The desire that will be added
- `new_desires` (list): The desire that will be added
- `new_emotion` (emotion): The emotion that will be added
- `new_emotions` (list): The emotion that will be added
- `new_ideal` (predicate): The ideal that will be added
- `new_ideals` (list): The ideals that will be added
- `new_uncertainties` (list): The uncertainty that will be added
- `new_uncertainty` (predicate): The uncertainty that will be added
- `obligation` (predicate): The mandatory obligation
- `obligations` (list): The mandatory obligations
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `remove_belief` (predicate): The belief that will be removed
- `remove_beliefs` (list): The belief that will be removed
- `remove_desire` (predicate): The desire that will be removed
- `remove_desires` (list): The desire that will be removed
- `remove_emotion` (emotion): The emotion that will be removed
- `remove_emotions` (list): The emotion that will be removed
- `remove_ideal` (predicate): The ideal that will be removed
- `remove_ideals` (list): The ideals that will be removed
- `remove_intention` (predicate): The intention that will be removed
- `remove_obligation` (predicate): The obligation that will be removed
- `remove_obligations` (list): The obligation that will be removed
- `remove_uncertainties` (list): The uncertainty that will be removed

- `remove_uncertainty` (predicate): The uncertainty that will be removed
- `strength` (any type in [float, int]): The strength of the mental state created
- `threshold` (float): Threshold linked to the emotion.
- `uncertainties` (list): The mandatory uncertainties
- `uncertainty` (predicate): The mandatory uncertainty
- `when` (boolean):

Definition

enables to add or remove mental states depending on the emotions of the agent, after the emotional engine and before the cognitive or normative engine.

Usages

- Other examples of use:

```
coping emotion: new_emotion("fear") when: flip(0.5) new_desire:  
new_predicate("test");
```

Embedments

- The `coping` statement is of type: **Behavior**
- The `coping` statement can be embedded into: simple_bdi, parallel_bdi, Species, Model,
- The `coping` statement embeds statements:

create

Facets

- `species` (any type in [species, agent]), (omissible) : an expression that evaluates to a species, the species of the agents to be created. In the case of simulations, the name 'simulation', which represents the current instance of simulation, can also be used as a proxy to their species
- `as` (species): optionally indicates a species into which to cast the created agents.
- `from` (any type): an expression that evaluates to a localized entity, a list of localized entities, a

string (the path of a file), a file (shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a database

- `number` (int): an expression that evaluates to an int, the number of created agents
- `returns` (a new identifier): a new temporary variable name containing the list of created agents (a list, even if only one agent has been created)
- `with` (map): an expression that evaluates to a map, for each pair the key is a species attribute and the value the assigned value

Definition

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

Usages

- Its simple syntax to create `an_int` agents of species `a_species` is:

```
create a_species number: an_int;
create species_of(self) number: 5 returns: list5Agents;
```

- 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:: read('TYPE_OCC'),
nature::read('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
```

- 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)
}
```

- Deprecated uses:

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

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

Embedments

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

data

Facets

- `legend` (string), (omissible) : The legend of the chart
- `value` (any type in [float, point, list]): The value to output on the chart
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (any type in [rgb, list]): color of the serie, for heatmap can be a list to specify [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false)
- `line_visible` (boolean): Whether lines are visible or not
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker
- `marker_size` (float): Size in pixels of the marker
- `style` (an identifier), takes values in: {line, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one specified on chart statement)
- `thickness` (float): The thickness of the lines to draw
- `use_second_y_axis` (boolean): Use second y axis for this serie

- `x_err_values` (any type in [float, list]): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_err_values` (any type in [float, list]): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

Definition

This statement allows to describe the values that will be displayed on the chart.

Usages

Embedments

- The `data` statement is of type: **Single statement**
 - The `data` statement can be embedded into: chart, Sequence of statements or action,
 - The `data` statement embeds statements:
-

datalist

Facets

- `value` (list): the values to display. Has to be a matrix, a list or a List of List. Each element can be a number (series/histogram) or a list with two values (XY chart)
- `legend` (list), (omissible) : the name of the series: a list of strings (can be a variable with dynamic names)
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (list): list of colors, for heatmaps can be a list of [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false), same for all series.
- `line_visible` (boolean): Line visible or not (same for all series)
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle,

marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle};
Shape of the marker. Same one for all series.

- `marker_size` (list): the marker sizes to display. Can be a list of numbers (same size for each marker of the series) or a list of list (different sizes by point)
- `style` (an identifier), takes values in: {line, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one specified on chart statement)
- `thickness` (float): The thickness of the lines to draw
- `use_second_y_axis` (boolean): Use second y axis for this serie
- `x_err_values` (list): the X Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_err_values` (list): the Y Error bar values to display. Has to be a List. Each element can be a number or a list with two values (low and high value)
- `y_minmax_values` (list): the Y MinMax bar values to display (BW charts). Has to be a List. Each element can be a number or a list with two values (low and high value)

Definition

add a list of series to a chart. The number of series can be dynamic (the size of the list changes each step). See Ant Foraging (Charts) model in ChartTest for examples.

Usages

Embedments

- The `datalist` statement is of type: **Single statement**
 - The `datalist` statement can be embedded into: chart, Sequence of statements or action,
 - The `datalist` statement embeds statements:
-

default

Facets

- `value` (any type), (omissible) : The value or values this statement tries to match

Definition

Used in a switch match structure, the block prefixed by default is executed only if no other block has matched (otherwise it is not).

Usages

- See also: [switch](#), [match](#),

Embedments

- The `default` statement is of type: **Sequence of statements or action**
 - The `default` statement can be embedded into: `switch`,
 - The `default` statement embeds statements:
-

diffuse

Facets

- `var` (an identifier), (omissible) : the variable to be diffused. If diffused over a field, then this name will serve to identify the diffusion
- `on` (any type in [species, field, list]): the list of agents (in general cells of a grid), or a field on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the proportion value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix that masks the diffusion (created from an image for instance). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- `matrix` (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method. One of 'convolution' or 'dot_product'

- `min` (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- `propagation` (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : $S' = (S / N / \text{proportion}) - \text{variation}$. 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 neighbors receives the intensity : $S / \text{proportion} - \text{variation}$. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to 'diffusion'.
- `proportion` (float): a diffusion rate
- `radius` (int): a diffusion radius (in number of cells from the center)
- `variation` (float): an absolute value to decrease at each neighbors

Definition

This statements allows a value to diffuse among a species on agents (generally on a grid) depending on a given diffusion matrix.

Usages

- A basic example of diffusion of the variable phero defined in the species cells, given a diffusion matrix `math_diff` is:

```
matrix<float> math_diff <- matrix([[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]);  
diffuse var: phero on: cells matrix: math_diff;
```

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

```
diffuse var: phero on: cells matrix: math_diff mask: mymask;
```

- A convenient way to have an uniform diffusion in a given radius is (which is equivalent to the above diffusion):

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

Embedments

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

display

Facets

- `name` (a label), (omissible) : the identifier of the display
- `antialias` (boolean): Indicates whether to use advanced antialiasing for the display or not. The default value is the one indicated in the preferences of GAMA ('false' is its factory default). Antialiasing produces smoother outputs, but comes with a cost in terms of speed and memory used.
- `autosave` (any type in [boolean, point, string]): Allows to save this display on disk. This facet accepts bool, point or string values. If it is false or nil, nothing happens. 'true' will save it at a resolution of 500x500 with a standard name (containing the name of the model, display, resolution, cycle and time). A non-nil point will change that resolution. A non-nil string will keep 500x500 and change the filename (if it is not dynamically built, the previous file will be erased). Note that setting autosave to true in a display will synchronize all the displays defined in the experiment
- `axes` (boolean): Allows to enable/disable the drawing of the world shape and the ordinate axes. Default can be configured in Preferences
- `background` (rgb): Allows to fill the background of the display with a specific color
- `camera` (string): Allows to define the name of the camera to use. Default value is 'default'. Accepted values are (1) the name of one of the cameras defined using the 'camera' statement or (2) one of the preset cameras, accessible using constants: #from_above, #from_left, #from_right, #from_up_left, #from_up_right, #from_front, #from_up_front, #isometric
- `fullscreen` (any type in [boolean, int]): Indicates, when using a boolean value, whether or not the display should cover the whole screen (default is false). If an integer is passed, specifies also the screen to use: 0 for the primary monitor, 1 for the secondary one, and so on

and so forth. If the monitor is not available, the first one is used

- `keystone` (container): Set the position of the 4 corners of your screen ([topLeft,topRight,botLeft,botRight]), in (x,y) coordinate (the (0,0) position is the top left corner, while the (1,1) position is the bottom right corner). The default value is : [{0,0},{1,0},{0,1},{1,1}]
- `light` (boolean): Allows to enable/disable the light at once. Default is true
- `orthographic_projection` (boolean): Allows to enable/disable the orthographic projection. Default can be configured in Preferences
- `parent` (an identifier): Declares that this display inherits its layers and attributes from the parent display named as the argument. Expects the identifier of the parent display or a string if the name of the parent contains spaces
- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)
- `show_fps` (boolean): Allows to enable/disable the drawing of the number of frames per second
- `toolbar` (any type in [boolean, rgb]): Indicates whether the top toolbar of the display view should be initially visible or not. If a color is passed, then the background of the toolbar takes this color
- `type` (a label): Allows to use either Java2D (for planar models) or OpenGL (for 3D models) as the rendering subsystem
- `virtual` (boolean): Declaring a display as virtual makes it invisible on screen, and only usable for display inheritance
- `z_far` (float): Set the distances to the far depth clipping planes. Must be positive.
- `z_near` (float): Set the distances to the near depth clipping planes. Must be positive.

Definition

A display refers to an independent and mobile part of the interface that can display species, images, texts or charts.

Usages

- The general syntax is:

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

- Each display can include different layers (like in a GIS).

```
display gridWithElevationTriangulated type: opengl ambient_light: 100 {
    grid cell elevation: true triangulation: true;
    species people aspect: base;
}
```

Embedments

- The `display` statement is of type: **Output**
 - The `display` statement can be embedded into: `output`, `permanent`,
 - The `display` statement embeds statements: `agents`, `camera`, `chart`, `display_grid`, `event`, `graphics`, `image_layer`, `light`, `mesh`, `overlay`, `rotation`, `species_layer`,
-

display_grid

Facets

- `species` (species), (omissible) : the species of the agents in the grid
- `border` (rgb): the color to draw lines (borders of cells)
- `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)
- `hexagonal` (boolean):
- `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 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- `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)

- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `smooth` (boolean): Applies a simple convolution (box filter) to smooth out the terrain produced by this field. Does not change the values of course.
- `text` (boolean): specify whether the attribute used to compute the elevation is displayed on each cells (false by default)
- `texture` (file): Either file containing the texture image to be applied on the grid or, if not specified, the use of the image composed by the colors of the cells
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `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)
- `visible` (boolean): Defines whether this layer is visible or not
- `wireframe` (boolean): if true displays the grid in wireframe using the lines color

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](#), [species_layer](#),

Embedments

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

do

Facets

- `action` (an identifier), (omissible) : the name of an action or a primitive
- `internal_function` (any type):
- `with` (map): a map expression containing the parameters of the action

Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this [[BuiltIn161 page](#)]; for the list of primitives defined by the different skills, see this [[Skills161 page](#)]. Finally, see this [[Species161 page](#)] to know how to declare custom actions.

Usages

- The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

```
do name_of_action_or_primitive;
```

- In case the action expects one or more arguments to be passed, they are defined by using facets (enclosed tags or a map are now deprecated):

```
do name_of_action_or_primitive arg1: expression1 arg2: expression2;
```

- In case the result of the action needs to be made available to the agent, the action can be called with the agent calling the action (`self` when the agent itself calls the action) instead of `do`; the result should be assigned to a temporary variable:

```
type_returned_by_action result <- self name_of_action_or_primitive [];
```

- In case of an action expecting arguments and returning a value, the following syntax is used:

```
type_returned_by_action result <- self name_of_action_or_primitive  
[arg1::expression1, arg2::expression2];
```

- Deprecated uses: following uses of the `do` statement (still accepted) are now deprecated:

```
// Simple syntax:  
do action: name_of_action_or_primitive;  
  
// In case the result of the action needs to be made available to the agent, the  
`returns` keyword can be defined; the result will then be referred to by the  
temporary variable declared in this attribute:  
do name_of_action_or_primitive returns: result;  
do name_of_action_or_primitive arg1: expression1 arg2: expression2 returns:  
result;  
type_returned_by_action result <- name_of_action_or_primitive(self,  
[arg1::expression1, arg2::expression2]);  
  
// In case the result of the action needs to be made available to the agent  
let result <- name_of_action_or_primitive(self, []);  
  
// In case the action expects one or more arguments to be passed, they can also be  
defined by using enclosed `arg` statements, or the `with` facet with a map of  
parameters:  
do name_of_action_or_primitive with: [arg1::expression1, arg2::expression2];  
  
or  
  
do name_of_action_or_primitive {  
    arg arg1 value: expression1;
```

Embedments

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

draw

Facets

- `geometry` (any type), (omissible) : any type of data (it can be geometry, image, text)
- `anchor` (point): Only used when perspective: true in OpenGL. The anchor point of the location with respect to the envelope of the text to draw, can take one of the following values: #center, #top_left, #left_center, #bottom_left, #bottom_center, #bottom_right, #right_center, #top_right, #top_center; or any point between {0,0} (#bottom_left) and {1,1} (#top_right)
- `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
- `border` (any type in [rgb, boolean]): if used with a color, represents the color of the geometry border. If set to false, expresses that no border should be drawn. If not set, the borders will be drawn using the color of the geometry.
- `color` (any type in [rgb, container]): the color to use to display the object. In case of images, will try to colorize it. You can also pass a list of colors : in that case, each color will be matched to its corresponding vertex.
- `depth` (float): (only if the display type is opengl) Add an artificial depth to the geometry previously defined (a line becomes a plan, a circle becomes a cylinder, a square becomes a cube, a polygon becomes a polyhedron with height equal to the depth value). Note: This only works if the geometry is not a point
- `end_arrow` (any type in [int, float]): the size of the arrow, located at the end of the drawn geometry
- `font` (any type in [font, string]): the font used to draw the text, if any. Applying this facet to geometries or images has no effect. You can construct here your font with the operator "font".
`ex : font:font("Helvetica", 20, #plain)`
- `lighted` (boolean): Whether the object should be lighted or not (only applicable in the context

of opengl displays)

- `perspective` (boolean): Whether to render the text in perspective or facing the user. Default is in perspective.
- `precision` (float): (only if the display type is opengl and only for text drawing) controls the accuracy with which curves are rendered in glyphs. Between 0 and 1, the default is 0.1. Smaller values will output much more faithful curves but can be considerably slower, so it is better if they concern text that does not change and can be drawn inside layers marked as 'refresh: false'
- `rotate` (any type in [float, int, pair]): orientation of the shape/text/icon; can be either an int/float (angle) or a pair float::point (angle::rotation axis). The rotation axis, when expressed as an angle, is by default {0,0,1}
- `size` (any type in [float, point]): Size of the shape/icon/image to draw, expressed as a bounding box (width, height, depth; if expressed as a float, represents the box as a cube). Does not apply to texts: use a font with the required size instead
- `texture` (any type): the texture(s) that should be applied to the geometry. Either a path to a file or a list of paths
- `width` (float): The line width to use for drawing this object
- `wireframe` (boolean): a condition specifying whether to draw the geometry in wireframe or not

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 font:
font('Default', 12, #bold) ;
    draw file(ant_shape_full) rotate: heading at: location size: 5
}

```

- Arrows can be drawn with any kind of geometry, using begin_arrow and end_arrow facets, combined with the empty: facet to specify whether it is plain or empty

```

aspect arrowAspect {
    draw line([{20, 20}, {40, 40}]) color: #black begin_arrow:5;
    draw line([{10, 10},{20, 50}, {40, 70}]) color: #green end_arrow: 2
begin_arrow: 2 empty: true;
    draw square(10) at: {80,20} color: #purple begin_arrow: 2 empty: true;
}

```

Embedments

- The `draw` statement is of type: **Single statement**
 - The `draw` statement can be embedded into: aspect, Sequence of statements or action, Layer,
 - The `draw` statement embeds statements:
-

else

Facets

Definition

This statement cannot be used alone

Usages

- See also: [if](#),

Embedments

- The `else` statement is of type: **Sequence of statements or action**
- The `else` statement can be embedded into: if,

- The `else` statement embeds statements:
-

emotional_contagion

Facets

- `emotion_detected` (emotion): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the emotional contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `emotion_created` (emotion): the emotion that will be created with the contagion
- `intensity` (float): The intensity value of the emotion created to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to make conscious or unconscious emotional contagion

Usages

- Other examples of use:

```
emotional_contagion emotion_detected:fearConfirmed;
emotional_contagion emotion_detected:fear emotion_created:fearConfirmed;
emotional_contagion emotion_detected:fear emotion_created:fearConfirmed charisma:
0.5 receptivity: 0.5;
```

Embedments

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

enforcement

Facets

- `name` (an identifier), (omissible) : the identifier of the enforcement
- `law` (string): The law to enforce
- `norm` (string): The norm to enforce
- `obligation` (predicate): The obligation to enforce
- `reward` (string): The positive sanction to apply if the norm has been followed
- `sanction` (string): The sanction to apply if the norm is violated
- `when` (boolean): A boolean value to enforce only with a certain condition

Definition

apply a sanction if the norm specified is violated, or a reward if the norm is applied by the perceived agent

Usages

- Other examples of use:

```
focus var:speed; //where speed is a variable from a species that is being  
perceived
```

Embedments

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

enter

Facets

Definition

In an FSM architecture, `enter` introduces a sequence of statements to execute upon entering a state.

Usages

- In the following example, at the step it enters into the state `s_init`, the message 'Enter in `s_init`' is displayed followed by the display of the state name:

```
state s_init {  
    enter {  
        write "Enter in" + state;  
    }  
    write state;  
}
```

- See also: [state](#), [exit](#), [transition](#),

Embedments

- The `enter` statement is of type: **Sequence of statements or action**
- The `enter` statement can be embedded into: `state`,
- The `enter` statement embeds statements:

equation

Facets

- `name` (an identifier), (omissible) : the equation identifier
- `params` (list): the list of parameters used in predefined equation systems
- `simultaneously` (list): a list of species containing a system of equations (all systems will be

solved simultaneously)

- `vars` (list): the list of variables used in predefined equation systems

Definition

The equation statement is used to create an equation system from several single equations.

Usages

- The basic syntax to define an equation system is:

```
float t;
float S;
float I;
equation SI {
    diff(S,t) = (- 0.3 * S * I / 100);
    diff(I,t) = (0.3 * S * I / 100);
}
```

- If the type: facet is used, a predefined equation system is defined using variables `vars:` and parameters `params:` in the right order. All possible predefined equation systems are the following ones (see [EquationPresentation161 EquationPresentation161] for precise definition of each classical equation system):

```
equation eqSI type: SI vars: [S,I,t] params: [N,beta];
equation eqSIS type: SIS vars: [S,I,t] params: [N,beta,gamma];
equation eqSIR type:SIR vars:[S,I,R,t] params:[N,beta,gamma];
equation eqSIRS type: SIRS vars: [S,I,R,t] params: [N,beta,gamma,omega,mu];
equation eqSEIR type: SEIR vars: [S,E,I,R,t] params: [N,beta,gamma,sigma,mu];
equation eqLV type: LV vars: [x,y,t] params: [alpha,beta,delta,gamma];
```

- If the simultaneously: facet is used, system of all the agents will be solved simultaneously.
- See also: `=`, `solve`,

Embedments

- The `equation` statement is of type: **Sequence of statements or action**
- The `equation` statement can be embedded into: Species, Model,
- The `equation` statement embeds statements: `=`,

error

Facets

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

Definition

The statement makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

Usages

- Throwing an error

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

Embedments

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

event

Facets

- `name` (string), (omissible) : the type of event captured: basic events include #mouse_up, #mouse_down, #mouse_move, #mouse_exit, #mouse_enter, #mouse_menu, #arrow_down, #arrow_up, #arrow_left, #arrow_right, #escape, #tab, #enter, #page_up, #page_down or a character
- `action` (action): The identifier of the action to be executed in the context of the simulation. This action needs to be defined in 'global' or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the

```
pseudo-constant #user_location
```

- `type` (string): Type of device used to generate events. Defaults to 'default', which encompasses keyboard and mouse

Definition

`event` allows to interact with the simulation by capturing mouse or key events and doing an action. The name of this action can be defined with the 'action:' facet, in which case the action needs to be defined in 'global' or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the pseudo-constant `#user_location`. The statements to execute can also be defined in the block at the end of this statement, in which case they will be executed in the context of the experiment

Usages

- The general syntax is:

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

- For instance:

```
global {
    // ...
    action myAction () {
        point loc <- #user_location; // contains the location of the mouse in the
world
        list<agent> selected_agents <- agents inside (10#m around loc); // 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_layer`, `overlay`, `species_layer`,

Embedments

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

exit

Facets

Definition

In an FSM architecture, `exit` introduces a sequence of statements to execute right before exiting the state.

Usages

- In the following example, at the state it leaves the state `s_init`, he will display the message '`EXIT from s_init`':

```
state s_init initial: true {
    write state;
    transition to: s1 when: (cycle > 2) {
        write "transition s_init -> s1";
    }
    exit {
        write "EXIT from "+state;
    }
}
```

- See also: [enter](#), [state](#), [transition](#),

Embedments

- The `exit` statement is of type: **Sequence of statements or action**
- The `exit` statement can be embedded into: `state`,
- The `exit` statement embeds statements:

experiment

Facets

- `name` (a label), (omissible) : identifier of the experiment
- `title` (a label):
- `type` (a label): the type of the experiment (either 'gui' or 'batch')
- `autorun` (boolean): whether this experiment should be run automatically when launched (false by default)
- `benchmark` (boolean): If true, make GAMA record the number of invocations and running time of the statements and operators of the simulations launched in this experiment. The results are automatically saved in a csv file in a folder called 'benchmarks' when the experiment is closed
- `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): Allows to keep the same seed between simulations. Mainly useful for batch experiments
- `keep_simulations` (boolean): In the case of a batch experiment, specifies whether or not the simulations should be kept in memory for further analysis or immediately discarded with only their fitness kept in memory
- `parallel` (any type in [boolean, int]): When set to true, use multiple threads to run its simulations. Setting it to n will set the numbers of threads to use
- `parent` (an identifier): the parent experiment (in case of inheritance between experiments)
- `repeat` (int): In the case of a batch experiment, expresses how many times the simulations must be repeated
- `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):
- `until` (boolean): In the case of a batch experiment, an expression that will be evaluated to know when a simulation should be terminated
- `virtual` (boolean): whether the experiment is virtual (cannot be instantiated, but only used as a parent, false by default)

Definition

Declaration of a particular type of agent that can manage simulations. If the experiment directly imports a model using the 'model:' facet, this facet *must* be the first one after the name of the experiment

Usages

Embedments

- The `experiment` statement is of type: **Experiment**
 - The `experiment` statement can be embedded into: Model,
 - The `experiment` statement embeds statements:
-

exploration

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `factorial` (list): The number of sample required.
- `from` (string): a path to a file where each lines correspond to one parameter set and each colon a parameter
- `iterations` (int): The number of iteration for orthogonal sampling, 5 by default
- `levels` (int): The number of levels for morris sampling, 4 by default
- `sample` (int): The number of sample required, 132 by default
- `sampling` (string): The name of the method (among saltelli/morris/latinhypercube/orthogonal/uniform/factorial)

- `with` (list): the list of parameter sets to explore; a parameter set is defined by a map: key: name of the variable, value: expression for the value of the variable

Definition

This is the standard batch method. The exploration 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. You can also choose a sampling method for the exploration. See [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the exploration statement uses `method exploration` instead of the expected `exploration name: id`:

```
method exploration;
```

- Simplest example:

```
method exploration;
```

- Using sampling facet:

```
method exploration sampling:latinhypercube sample:100;
```

- Using from facet:

```
method exploration from:"./path/to/my/exploration/plan.csv";
```

- Using with facet:

```
method exploration with:[["a":0.5, "b":10],["a":0.1, "b":100]];
```

Embedments

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

focus

Facets

- `agent_cause` (agent): the agentCause value of the created belief (can be nil)
- `belief` (predicate): The predicate to focus on the beliefs of the other agent
- `desire` (predicate): The predicate to focus on the desires of the other agent
- `emotion` (emotion): The emotion to focus on the emotions of the other agent
- `expression` (any type): an expression that will be the value kept in the belief
- `id` (string): the identifier of the focus
- `ideal` (predicate): The predicate to focus on the ideals of the other agent
- `is_uncertain` (boolean): a boolean to indicate if the mental state created is an uncertainty
- `lifetime` (int): the lifetime value of the created belief
- `strength` (any type in [float, int]): The priority of the created predicate
- `truth` (boolean): the truth value of the created belief
- `uncertainty` (predicate): The predicate to focus on the uncertainties of the other agent
- `var` (any type in [any type, list, container]): the variable of the perceived agent you want to add to your beliefs
- `when` (boolean): A boolean value to focus only with a certain condition

Definition

enables to directly add a belief from the variable of a perceived species.

Usages

- Other examples of use:

```
focus var:speed /*where speed is a variable from a species that is being perceived*/
```

Embedments

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

focus_on

Facets

- `value` (any type), (omissible) : The agent, list of agents, geometry to focus on

Definition

Allows to focus on the passed parameter in all available displays. Passing 'nil' for the parameter will make all screens return to their normal zoom

Usages

- Focuses on an agent, a geometry, a set of agents, etc...

```
focus_on my_species();
```

Embedments

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

generate

Facets

- **attributes** (map): To specify the explicit link between agent attributes and file based attributes
- **from** (any type): To specify the input data used to inform the generation process. Various data input can be used:
 - list of csv_file: can be aggregated or micro data
 - matrix: describe the joint distribution of two attributes
 - genstar generator: a dedicated gaml type to enclose various genstar options all in one
 - **species** (any type in [species, agent]), (omissible) : The species of the agents to be created.
 - **generator** (string): To specify the type of generator you want to use: as of now there is only DS (or DirectSampling) available
 - **number** (int): To specify the number of created agents interpreted as an int value. If facet is ommited or value is 0 or less, generator will treat data used in the 'from' facet as contingencies (i.e. a count of entities) and infer a number to generate (if distribution is used, then only one entity will be created)

Definition

Allows to create a synthetic population of agent from a set of given rules

Usages

- The synthax to create a minimal synthetic population from aggregated file is:

```
generate species:people number: 10000
from:[csv_file("../includes/Age & Sexe-Tableau 1.csv",";")]
attributes:[ "Age"::[ "Moins de 5 ans", "5 à 9 ans", "10 à 14 ans", "15 à 19 ans",
"20 à 24 ans",
"25 à 29 ans", "30 à 34 ans", "35 à 39 ans", "40 à 44 ans", "45 à 49 ans",
"50 à 54 ans", "55 à 59 ans", "60 à 64 ans", "65 à 69 ans", "70 à 74 ans", "75 à
79 ans",
"80 à 84 ans", "85 à 89 ans", "90 à 94 ans", "95 à 99 ans", "100 ans ou plus"],

"Sexe"::[ "Hommes", "Femmes"]];
```

Embedments

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

genetic

Facets

- `name` (an identifier), (omissible) : The name of this method. For internal use only
- `aggregation` (a label), takes values in: {min, max, avr}: the aggregation method
- `crossover_prob` (float): crossover probability between two individual solutions
- `improve_sol` (boolean): if true, use a hill climbing algorithm to improve the solutions at each generation
- `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)
- `stochastic_sel` (boolean): if true, use a stochastic selection algorithm (roulette) rather a deterministic one (keep the best solutions)

Definition

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article and [batch161 the batch dedicated page]. The principle of the GA is to search an optimal solution by applying evolution operators on an initial population of solutions. There are three types of evolution operators: crossover, mutation and selection. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Usages

- As other batch methods, the basic syntax of the `genetic` statement uses `method genetic` instead of the expected `genetic name: id`:

```
method genetic [facet: value];
```

- For example:

```
method genetic maximize: food_gathered pop_dim: 5 crossover_prob: 0.7  
mutation_prob: 0.1 nb_prelim_gen: 1 max_gen: 20;
```

Embedments

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

graphics

Facets

- `name` (a label), (omissible) : the human readable title of the graphics
- `background` (rgb): the background color of the layer. Default is none
- `border` (rgb): Color to apply to the border of the rectangular shape of the layer. Default is none
- `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 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.

- `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)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `visible` (boolean): Defines whether this layer is visible or not

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_layer](#), [overlay](#), [species_layer](#),

Embedments

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

- The `graphics` statement embeds statements:
-

highlight

Facets

- `value` (agent), (omissible) : The agent to hightlight
- `color` (rgb): An optional color to highlight the agent. Note that this color will become the default color for further higlight operations

Definition

Allows to highlight the agent passed in parameter in all available displays, optionaly setting a color. Passing 'nil' for the agent will remove the current highlight

Usages

- Highlighting an agent

```
highlight my_species(0) color: #blue;
```

Embedments

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

hill_climbing

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max, avr}: the aggregation method
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable

- `iter_max` (int): number of iterations. this number corresponds to the number of "moves" in the parameter space. For each move, the algorithm will test the whole neighborhood of the current solution, each neighbor corresponding to a particular set of parameters and thus to a run. Thus, there can be several runs per iteration (maximum: $2^{(\text{number of parameters})}$).
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize

Definition

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the `hill_climbing` statement uses `method hill_climbing` instead of the expected `hill_climbing name: id :`

```
method hill_climbing [facet: value];
```

- For example:

```
method hill_climbing iter_max: 50 maximize : food_gathered;
```

Embedments

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

if

Facets

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

Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true.

Usages

- The generic syntax is:

```
if bool_expr {  
    [statements]  
}
```

- Optionally, the statements to execute when the condition evaluates to false can be defined in a following statement else. The syntax then becomes:

```
if bool_expr {  
    [statements]  
}  
else {  
    [statements]  
}  
string valTrue <- "";  
if true {  
    valTrue <- "true";  
}  
else {  
    valTrue <- "false";  
}  
// valTrue equals "true"  
string valFalse <- "";  
if false {  
    valFalse <- "true";  
}  
else {  
    valFalse <- "false";  
}  
// valFalse equals "false"
```

- ifs and elses can be imbricated as needed. For instance:

```

if bool_expr {
    [statements]
}
else if bool_expr2 {
    [statements]
}
else {
    [statements]
}

```

Embedments

- The `if` statement is of type: **Sequence of statements or action**
 - The `if` statement can be embedded into: Behavior, Sequence of statements or action, Layer, Output,
 - The `if` statement embeds statements: `else`,
-

image_layer

Facets

- `name` (any type), (omissible) : the name/path of the image (in the case of a raster image), a matrix of int, an image file
- `color` (rgb): in the case of a shapefile, this the color used to fill in geometries of the shapefile. In the case of an image, it is used to tint the 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 for x and y. The z-coordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- `refresh` (boolean): (openGL only) specify whether the image display is refreshed or not. (false by default, true should be used in cases of images that are modified over the simulation)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.

- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `visible` (boolean): Defines whether this layer is visible or not

Definition

`image_layer` allows modeler to display an image (e.g. as background of a simulation). Note that this image will not be dynamically changed or moved in OpenGL, unless the refresh: facet is set to true.

Usages

- The general syntax is:

```
display my_display {
    image image_file [additional options];
}
```

- For instance, in the case of a bitmap image

```
display my_display {
    image "../images/my_background.jpg";
}
```

- If you already have your image stored in a matrix

```
display my_display {
    image my_image_matrix;
}
```

- 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 "../images/image1.jpg";  
    image "../images/image2.jpg";  
    image "../images/image3.jpg" position: {0,0,0.5};  
}
```

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

Embedments

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

inspect

Facets

- `name` (any type), (omissible) : the identifier of the inspector
- `attributes` (list): the list of attributes to inspect. A list that can contain strings or `pair<string,type>`, or a mix of them. These can be variables of the species, but also attributes present in the attributes table of the agent. The type is necessary in that case
- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)
- `type` (an identifier), takes values in: {agent, table}: the way to inspect agents: in a table, or a set of inspectors
- `value` (any type): the set of agents to inspect, could be a species, a list of agents or an agent

Definition

`inspect` (and `browse`) statements allows modeler to inspect a set of agents, in a table with agents and all their attributes or an agent inspector per agent, depending on the type: chosen. Modeler can choose which attributes to display. When `browse` is used, type: default value is table, whereas when `inspect` is used, type: default value is agent.

Usages

- An example of syntax is:

```
inspect "my_inspector" value: ant attributes: ["name", "location"];
```

Embedments

- The `inspect` statement is of type: **Output**
- The `inspect` statement can be embedded into: output, permanent, Behavior, Sequence of statements or action,
- The `inspect` statement embeds statements:

law

Facets

- `name` (an identifier), (omissible) : The name of the law
- `all` (boolean): add an obligation for each belief
- `belief` (predicate): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `lifetime` (int): the lifetime value of the mental state created
- `new_obligation` (predicate): The predicate that will be added as an obligation
- `new_obligations` (list): The list of predicates that will be added as obligations
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the

preferences). This facet is true by default.

- `strength` (any type in [float, int]): The strength of the mental state created
- `threshold` (float): Threshold linked to the obedience value.
- `when` (boolean):

Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

Usages

- Other examples of use:

```
rule belief: new_predicate("test") when: flip(0.5) new_desire:  
new_predicate("test");
```

Embedments

- The `law` statement is of type: **Single statement**
- The `law` statement can be embedded into: Species, Model,
- The `law` statement embeds statements:

layout

Facets

- `value` (any type), (omissible) : Either `#none`, to indicate that no layout will be imposed, or one of the four possible predefined layouts: `#stack`, `#split`, `#horizontal` or `#vertical`. This layout will be applied to both experiment and simulation display views. In addition, it is possible to define a custom layout using the `horizontal()` and `vertical()` operators
- `background` (rgb): Whether the whole interface of GAMA should be colored or not (nil by default)
- `consoles` (boolean): Whether the consoles are visible or not (true by default)
- `controls` (boolean): Whether the experiment should show its control toolbar on top or not

- `editors` (boolean): Whether the editors should initially be visible or not
- `navigator` (boolean): Whether the navigator view is visible or not (true by default)
- `parameters` (boolean): Whether the parameters view is visible or not (true by default)
- `tabs` (boolean): Whether the displays should show their tab or not
- `toolbars` (boolean): Whether the displays should show their toolbar or not
- `tray` (boolean): Whether the bottom tray is visible or not (true by default)

Definition

Represents the layout of the display views of simulations and experiments

Usages

- For instance, this layout statement will allow to split the screen occupied by displays in four equal parts, with no tabs. Pairs of `display::weight` represent the number of the display in their order of definition and their respective weight within a horizontal and vertical section

```
layout
horizontal([vertical([0::5000,1::5000])::5000,vertical([2::5000,3::5000])::5000])
tabs: false;
```

Embedments

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

let

Facets

- `name` (a new identifier), (omissible) : The name of the variable declared
- `index` (a datatype identifier): The type of the index if this declaration concerns a container
- `of` (a datatype identifier): The type of the contents if this declaration concerns a container
- `type` (a datatype identifier): The type of the variable

- `value` (any type): The value assigned to this variable

Definition

Allows to declare a temporary variable of the specified type and to initialize it with a value

Usages

Embedments

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

light

Facets

- `name` (string), (omissible) : The name of the light source, must be unique (otherwise the last definition prevails). Will be used to populate a menu where light sources can be easily turned on and off. Special names can be used: Using the special constant `#ambient` will allow to redefine or control the ambient light intensity and presence. Using the special constant `#default` will replace the default directional light of the surrounding display
- `active` (boolean): a boolean expression telling if the light is on or off. (default value if not specified : true)
- `angle` (float): the angle of the spot light in degree (only for spot light). (default value : 45)
- `direction` (point): the direction of the light (only for direction and spot light). (default value : {0.5,0.5,-1})
- `dynamic` (boolean): specify if the parameters of the light need to be updated every cycle or treated as constants. (default value : true).
- `intensity` (any type in [int, rgb]): an int / rgb / rgba value to specify either the color+intensity of the light or simply its intensity. (default value if not specified can be set in the Preferences. If not, it is equal to: (160,160,160,255)).
- `linear_attenuation` (float): the linear attenuation of the positionnal light. (default value : 0)
- `location` (point): the location of the light (only for point and spot light) in model coordinates. Default is {0,0,20}

- `quadratic_attenuation` (float): the quadratic attenuation of the positionnal light. (default value : 0)
- `show` (boolean): If true, draws the light source. (default value if not specified : false).
- `type` (string): the type of light to create. A value among {#point, #direction, #spot}

Definition

`light` allows to define diffusion lights in your 3D display. They must be given a name, which will help track them in the UI. Two names have however special meanings: `#ambient`, which designates the ambient luminosity and color of the scene (with a default intensity of (160,160,160,255) or the value set in the Preferences) and `#default`, which designates the default directional light applied to a scene (with a default medium intensity of (160,160,160,255) or the value set in the Preferences in the direction given by (0.5,0.5,1)). Redefining a light named `#ambient` or `#regular` will then modify these default lights (for example changing their color or deactivating them). To be more precise, and given all the default values of the facets, the existence of these two lights is effectively equivalent to redefining:
`light #ambient intensity: gama.pref_display_light_intensity;`
`light #default type: #direction intensity: gama.pref_display_light_intensity direction: {0.5,0.5,-1};`

Usages

- The general syntax is:

```
light 1 type:point location:{20,20,20} color:255, linear_attenuation:0.01
quadratic_attenuation:0.0001 draw_light:true update:false;
light 'spot1' type: #spot location:{20,20,20} direction:{0,0,-1} color:255
angle:25 linear_attenuation:0.01 quadratic_attenuation:0.0001 draw:true dynamic:
false;
light 'point2' type: #point direction:{1,1,-1} color:255 draw:true dynamic: false;
```

- See also: [display](#),

Embedments

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

loop

Facets

- `name` (a new identifier), (omissible) : a temporary variable name
- `from` (int): an int expression
- `over` (any type in [container, point]): a list, point, matrix or map expression
- `step` (int): an int expression
- `times` (int): an int expression
- `to` (int): an int expression
- `while` (boolean): a boolean expression

Definition

Allows the agent to perform the same set of statements either a fixed number of times, or while a condition is true, or by progressing in a collection of elements or along an interval of integers. Be aware that there are no prevention of infinite loops. As a consequence, open loops should be used with caution, as one agent may block the execution of the whole model.

Usages

- The basic syntax for repeating a fixed number of times a set of statements is:

```
loop times: an_int_expression {
    // [statements]
}
```

- The basic syntax for repeating a set of statements while a condition holds is:

```
loop while: a_bool_expression {
    // [statements]
}
```

- The basic syntax for repeating a set of statements by progressing over a container 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 or -1, depending on the direction of the range. If it is defined, its sign will be respected, so that a positive step will never allow the loop to enter a loop from i to j where i is greater than j

```
list the_list <- list (species_of (self));  
loop i from: 0 to: length (the_list) - 1 {
```

Embedments

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

match

Facets

- `value` (any type), (omissible) : The value or values this statement tries to match

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. Four kinds of match can be used, equality, containment, betweenness and regex matching

Usages

- match block is executed if the switch value is equals to the value of the match:

```
switch 3 {  
    match 1 {write "Match 1"; }  
    match 3 {write "Match 2"; }  
}
```

- match_between block is executed if the switch value is in the interval given in value of the match_between:

```
switch 3 {  
    match_between [1,2] {write "Match OK between [1,2]"; }  
    match_between [2,5] {write "Match OK between [2,5]"; }  
}
```

- `match_one` block is executed if the switch value is equals to one of the values of the `match_one`:

```
switch 3 {
    match_one [0,1,2] {write "Match OK with one of [0,1,2]"; }
    match_between [2,3,4,5] {write "Match OK with one of [2,3,4,5]"; }
}
```

- See also: [switch](#), [default](#),

Embedments

- The `match` statement is of type: **Sequence of statements or action**
 - The `match` statement can be embedded into: `switch`,
 - The `match` statement embeds statements:
-

mesh

Facets

- `source` (any type in [file, matrix, species]), (omissible) : Allows to specify the elevation of each cell by passing a grid, a raster, image or csv file or directly a matrix of int/float. The dimensions of the field are those of the file or matrix.
- `above` (float): Can be used to specify a 'minimal' value under which the render will not render the cells with this value
- `border` (rgb): the color to draw lines (borders of cells)
- `color` (any type in [rgb, list, map]): if true, and if neither 'grayscale' or 'texture' are specified, displays the field using the given color or colors. List of colors, palettes (with interpolation), gradients and scales are supported
- `grayscale` (boolean): if true, gives a grey color to each polygon depending on its elevation (false by default). Supersedes 'color' if it is defined.
- `no_data` (float): Can be used to specify a 'no_data' value, forcing the renderer to not render the cells with this value. If not specified, that value will be searched in the field to display
- `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 for x and y. The z-coordinate can only be defined between 0 and 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, but should be deactivated if the field is static)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `scale` (float): Represents the z-scaling factor, which allows to scale all values of the field.
- `size` (any type in [point, float]): Represents the extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions. This latter possibility allows to limit the height of the field. If only a flat value is provided, it is considered implicitly as the z maximal amplitude (or z scaling factor if < 1)
- `smooth` (any type in [boolean, int]): Applies a simple convolution (box filter) to smooth out the terrain produced by this field. If true, one pass is done with a simple 3x3 kernel. Otherwise, the user can specify the number of successive passes (up to 4). Specifying 0 is equivalent to passing false
- `text` (boolean): specify whether the value that represents the elevation is displayed on each cell (false by default)
- `texture` (file): A file containing the texture image to be applied to the field. If not specified, the field will be displayed either in color or grayscale, depending on the other facets
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `triangulation` (boolean): specifies whether the cells of the field 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)
- `visible` (boolean): Defines whether this layer is visible or not
- `wireframe` (boolean): if true displays the field in wireframe using the lines color

Definition

Allows the modeler to display in an optimized way a field of values, optionally using elevation.

Useful for displaying DEMs, for instance, without having to load them into a grid. Can be fed with a

matrix of int/float, a grid, a csv/raster/image file and supports many visualisation options

Usages

- The general syntax is:

```
display my_display {  
    field a_filename lines: #black position: { 0.5, 0 } size: {0.5,0.5}  
    triangulated: true texture: anothe_file;  
}
```

- See also: [display](#), [agents](#), [grid](#), [event](#), [graphics](#), [image](#), [overlay](#), [species_layer](#),

Embedments

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

migrate

Facets

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

Definition

This command permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Species of source agents and target species respect the following constraints: (i) they are "peer" species (sharing the same direct macro-species), (ii) they have sub-species vs. parent-species relationship.

Usages

- It can be used in a 3-levels model, in case where individual agents can be captured into group meso agents and groups into clouds macro agents. `migrate` is used to allow agents captured by groups to migrate into clouds. See the model 'Balls, Groups and Clouds.gaml' in the library.

```
migrate ball_in_group target: ball_in_cloud;
```

- See also: [capture](#), [release](#),

Embedments

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

monitor

Facets

- `name` (a label), (omissible) : identifier of the monitor
- `value` (any type): expression that will be evaluated to be displayed in the monitor
- `color` (rgb): Indicates the (possibly dynamic) color of this output (default is a light gray)
- `refresh` (boolean): Indicates the condition under which this output should be refreshed (default is true)

Definition

A monitor allows to follow the value of an arbitrary expression in GAML.

Usages

- An example of use is:

```
monitor "nb preys" value: length(prey as list) refresh_every: 5;
```

Embedments

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

morris

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `levels` (an identifier): Number of level for the Morris method, can't be 1
- `outputs` (list): The list of output variables to analyze through morris method
- `report` (string): The path to the file where the Morris report will be written
- `sample` (an identifier): The size of the sample for Morris samples
- `csv` (string): The path of morris sample .csv file. If don't use, automatic morris sampling will be perform and saved in the corresponding file
- `results` (string): The path to the file where the automatic batch report will be written

Definition

This algorithm runs a Morris exploration - it has been built upon the SILAB librairy - disabled the repeat facet of the experiment

Usages

- For example:

```
method morris sample_size:100 nb_levels:4 outputs:['my_var']
report:'./path/to/report.txt';
```

Embedments

- The `morris` statement is of type: **Batch method**
- The `morris` statement can be embedded into: Experiment,

- The `morris` statement embeds statements:
-

norm

Facets

- `name` (an identifier), (omissible) : the name of the norm
- `finished_when` (boolean): the boolean condition when the norm is finished
- `instantaneous` (boolean): indicates if the norm is instantaneous
- `intention` (predicate): the intention triggering the norm
- `lifetime` (int): the lifetime of the norm
- `obligation` (predicate): the obligation triggering of the norm
- `priority` (float): the priority value of the norm
- `threshold` (float): the threshold to trigger the norm
- `when` (boolean): the boolean condition when the norm is active

Definition

a norm indicates what action the agent has to do in a certain context and with and obedience value higher than the threshold

Usages

Embedments

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

output

Facets

- `autosave` (any type in [boolean, string]): Allows to save the whole screen on disk. A value of true/false will save it with the resolution of the physical screen. Passing it a string allows to

define the filename Note that setting autosave to true (or to any other value than false) in a display will synchronize all the displays defined in the experiment

- `synchronized` (boolean): Indicates whether the displays that compose this output should be synchronized with the simulation cycles

Definition

`output` blocks define how to visualize a simulation (with one or more display blocks that define separate windows). It will include a set of displays, monitors and files statements. It will be taken into account only if the experiment type is `gui`.

Usages

- Its basic syntax is:

```
experiment exp_name type: gui {  
    // [inputs]  
    output {  
        // [display, file, inspect, layout or monitor statements]  
    }  
}
```

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

Embedments

- The `output` statement is of type: **Output**
- The `output` statement can be embedded into: Model, Experiment,
- The `output` statement embeds statements: [display](#), [inspect](#), [layout](#), [monitor](#), [output_file](#),

output_file

Facets

- `name` (an identifier), (omissible) : The name of the file where you want to export the data
- `data` (string): The data you want to export
- `footer` (string): Define a footer for your export file

- `header` (string): Define a header for your export file
- `refresh` (boolean): Indicates the condition under which this file should be saved (default is true)
- `rewrite` (boolean): Rewrite or not the existing file
- `type` (an identifier), takes values in: {csv, text, xml}: The type of your output data

Definition

Represents an output that writes the result of expressions into a file

Usages

Embedments

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

overlay

Facets

- `background` (rgb): the background color of the overlay displayed inside the view (the bottom overlay remains black)
- `border` (rgb): Color to apply to the border of the rectangular shape of the overlay. Nil by default
- `center` (any type): an expression that will be evaluated and displayed in the center section of the bottom overlay
- `color` (any type in [list, rgb]): the color(s) used to display the expressions given in the 'left', 'center' and 'right' facets
- `left` (any type): an expression that will be evaluated and displayed in the left section of the bottom overlay
- `position` (point): position of the upper-left corner of the 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 for x and y. The z-

ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.

- `right` (any type): an expression that will be evaluated and displayed in the right section of the bottom overlay
- `rounded` (boolean): Whether or not the rectangular shape of the overlay should be rounded. True by default
- `size` (point): extent of the layer in the view from its position. Coordinates in [0,1[are treated as percentages of the total surface of the view, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Unlike 'position', no elevation can be provided with the z coordinate
- `transparency` (float): the transparency rate of the overlay (between 0 -- opaque and 1 -- fully transparent) when it is displayed inside the view. The bottom overlay will remain at 0.75
- `visible` (boolean): Defines whether this layer is visible or not

Definition

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

Usages

- To display information in the bottom overlay, the syntax is:

```
overlay "Cycle: " + (cycle) center: "Duration: " + total_duration + "ms" right:  
"Model time: " + as_date(time,"") color: [#yellow, #orange, #yellow];
```

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

Embedments

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

parameter

Facets

- `var` (an identifier): the name of the variable (that should be declared in global)
- `name` (a label), (omissible) : The message displayed in the interface
- `among` (list): the list of possible values that this parameter can take
- `category` (a label): a category label, used to group parameters in the interface
- `colors` (list): The colors of the control in the UI. An empty list has no effects. Only used for sliders and switches so far. For sliders, 3 colors will allow to specify the color of the left section, the thumb and the right section (in this order); 2 colors will define the left and right sections only (thumb will be dark green); 1 color will define the left section and the thumb. For switches, 2 colors will define the background for respectively the left 'true' and right 'false' sections. 1 color will define both backgrounds
- `disables` (list): a list of global variables whose parameter editors will be disabled when this parameter value is set to true or to a value that casts to true (they are otherwise enabled)
- `enables` (list): a list of global variables whose parameter editors will be enabled when this parameter value is set to true or to a value that casts to true (they are otherwise disabled)
- `extensions` (list): Makes only sense for file parameters. A list of file extensions (like 'gaml', 'shp', etc.) that restricts the choice offered to the users to certain file types (folders not concerned). Default is empty, effectively accepting all files
- `in_workspace` (boolean): Makes only sense for file parameters. Whether the file selector will be restricted to the workspace or not
- `init` (any type): the init value
- `labels` (list): The labels that will be displayed for switches (instead of True/False)
- `max` (any type): the maximum value
- `min` (any type): the minimum value
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the parameter changes
- `read_only` (boolean): Whether this parameter is read-only or editable
- `slider` (boolean): Whether or not to display a slider for entering an int or float value. Default is true when max and min values are defined, false otherwise. If no max or min value is defined, setting this facet to true will have no effect

- `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
- `updates` (list): a list of global variables whose parameter editors will be updated when this parameter value is changed (their min, max, step and among values will be updated accordingly if they depend on this parameter. Note that it might lead to some inconsistencies, for instance a parameter value which becomes out of range, or which does not belong anymore to a list of possible values. In these cases, the value of the affected parameter will not change)

Definition

The parameter statement specifies which global attributes (i) will change through the successive simulations (in batch experiments), (ii) can be modified by user via the interface (in gui experiments). In GUI experiments, parameters are displayed depending on their type.

Usages

- In gui experiment, the general syntax is the following:

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

- In batch experiment, the two following syntaxes can be used to describe the possible values of a parameter:

```
parameter 'Value of toto:' var: toto among: [1, 3, 7, 15, 100];
parameter 'Value of titi:' var: titi min: 1 max: 100 step: 2;
```

Embedments

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

perceive

Facets

- `target` (any type in [container, agent]): the list of the agent you want to perceive
- `name` (an identifier), (omissible) : the name of the perception
- `as` (species): an expression that evaluates to a species
- `emotion` (emotion): The emotion needed to do the perception
- `in` (any type in [float, geometry]): a float or a geometry. If it is a float, it's a radius of a detection area. If it is a geometry, it is the area of detection of others species.
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `threshold` (float): Threshold linked to the emotion.
- `when` (boolean): a boolean to tell when does the perceive is active

Definition

Allow the agent, with a bdi architecture, to perceive others agents

Usages

- the basic syntax to perceive agents inside a circle of perception

```
perceive name_of_perception target: the_agents_you_want_to_perceive in: distance
when: condition {
    //Here you are in the context of the perceived agents. To refer to the agent
    who does the perception, use myself.
    //If you want to make an action (such as adding a belief for example), use ask
    myself{ do the_action}
}
```

Embedments

- The `perceive` statement is of type: **Sequence of statements or action**
- The `perceive` statement can be embedded into: Species, Model,

- The `perceive` statement embeds statements:
-

permanent

Facets

- `synchronized` (boolean): Indicates whether the displays that compose this output should be synchronized with the simulation cycles

Definition

Represents the outputs of the experiment itself. In a batch experiment, the permanent section allows to define an output block that will NOT be re-initialized at the beginning of each simulation but will be filled at the end of each simulation.

Usages

- For instance, this permanent section will allow to display for each simulation the end value of the food_gathered variable:

```
permanent {
    display Ants background: rgb('white') refresh_every: 1 {
        chart "Food Gathered" type: series {
            data "Food" value: food_gathered;
        }
    }
}
```

Embedments

- The `permanent` statement is of type: **Output**
 - The `permanent` statement can be embedded into: Experiment,
 - The `permanent` statement embeds statements: `display`, `inspect`, `monitor`, `output_file`,
-

plan

Facets

- `name` (an identifier), (omissible) :
- `emotion` (emotion):
- `finished_when` (boolean):
- `instantaneous` (boolean):
- `intention` (predicate):
- `priority` (float):
- `threshold` (float):
- `when` (boolean):

Definition

define an action plan performed by an agent using the BDI engine

Usages

Embedments

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

pso

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `iter_max` (int): number of iterations
- `aggregation` (a label), takes values in: {min, max, avr}: the aggregation method
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize

- `num_particles` (int): number of particles
- `weight_cognitive` (float): weight for the cognitive component
- `weight_inertia` (float): weight for the inertia component
- `weight_social` (float): weight for the social component

Definition

This algorithm is an implementation of the Particle Swarm Optimization algorithm. Only usable for numerical parameters and based on a continuous parameter space search. See the wikipedia article for more details.

Usages

- As other batch methods, the basic syntax of the `pso` statement uses `method pso` instead of the expected `pso name: id`:

```
method pso [facet: value];
```

- For example:

```
method pso iter_max: 50 num_particles: 10 weight_inertia:0.7 weight_cognitive:  
1.5 weight_social: 1.5 maximize: food_gathered;
```

Embedments

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

put

Facets

- `in` (any type in [container, species, agent, geometry]): an expression that evaluates to a container

- `item` (any type), (omissible) : any expression
- `all` (any type): any expression
- `at` (any type): any expression
- `key` (any type): any expression

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 be an integer in the bound of the list. The facet `all:` is used to replace all the elements of the list by the given value.

```
list<int>putList <- [1,2,3,4,5]; // putList equals [1,2,3,4,5]
put -10 at: 1 in: putList; // putList equals [1,-10,3,4,5]
put 10 all: true in: putList; // putList equals [10,10,10,10,10]
```

- In the case of a matrix, the position should be a point in the bound of the matrix. The facet `all:` is used to replace all the elements of the matrix by the given value.

```
matrix<int>putMatrix <- matrix([[0,1],[2,3]]); // putMatrix equals
matrix([[0,1],[2,3]])
put -10 at: {1,1} in: putMatrix; // putMatrix equals matrix([[0,1],[2,-10]])
put 10 all: true in: putMatrix; // putMatrix equals matrix([[10,10],[10,10]])
```

- In the case of a map, the position should be one of the key values of the map. Notice that if the given key value does not exist in the map, the given pair `key::value` will be added to the map. The facet `all` is used to replace the value of all the pairs of the map.

```

map<string,int>putMap <- ["x":4,"y":7]; // putMap equals ["x":4,"y":7]
put -10 key: "y" in: putMap; // putMap equals ["x":4,"y":-10]
put -20 key: "z" in: putMap; // putMap equals ["x":4,"y":-10, "z": -20]
put -30 all: true in: putMap; // putMap equals ["x": -30,"y": -30, "z": -30]

```

Embedments

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

reactive_tabu

Facets

- `name` (an identifier), (omissible) :
- `aggregation` (a label), takes values in: {min, max, avr}: the aggregation method
- `cycle_size_max` (int): minimal size of the considered cycles
- `cycle_size_min` (int): maximal size of the considered cycles
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `iter_max` (int): number of iterations. this number corresponds to the number of "moves" in the parameter space. For each move, the algorithm will test the whole neighborhood of the current solution, each neighbor corresponding to a particular set of parameters and thus to a run. Thus, there can be several runs per iteration (maximum: $2^{(\text{number of parameters})}$).
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `nb_tests_wthout_col_max` (int): number of movements without collision before shortening the tabu list
- `tabu_list_size_init` (int): initial size of the tabu list
- `tabu_list_size_max` (int): maximal size of the tabu list
- `tabu_list_size_min` (int): minimal size of the tabu list

Definition

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle. See the batch dedicated page.

Usages

- As other batch methods, the basic syntax of the reactive_tabu statement uses `method reactive_tabu` instead of the expected `reactive_tabu name: id`:

```
method reactive_tabu [facet: value];
```

- For example:

```
method reactive_tabu iter_max: 50 tabu_list_size_init: 5 tabu_list_size_min: 2  
tabu_list_size_max: 10 nb_tests_wthout_col_max: 20 cycle_size_min: 2  
cycle_size_max: 20 maximize: food_gathered;
```

Embedments

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

reflex

Facets

- `name` (an identifier), (omissible) : the identifier of the reflex
- `when` (boolean): an expression that evaluates a boolean, the condition to fulfill in order to execute the statements embedded in the reflex.

Definition

Reflexes are sequences of statements that can be executed by the agent. Reflexes prefixed by the 'reflex' keyword are executed continuously. Reflexes prefixed by 'init' are executed only immediately after the agent has been created. Reflexes prefixed by 'abort' just before the agent is killed. If a facet `when:` is defined, a reflex is executed only if the boolean expression evaluates to true.

Usages

- Example:

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

Embedments

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

release

Facets

- `target` (any type in [agent, list, attributes]), (omissible) : an expression that is evaluated as an agent/a list of the agents to be released or an agent saved as a map

- `as` (species): an expression that is evaluated as a species in which the micro-agent will be released
- `in` (agent): an expression that is evaluated as an agent that will be the macro-agent in which micro-agent will be released, i.e. their new host
- `returns` (a new identifier): a new variable containing a list of the newly released agent(s)

Definition

Allows an agent to release its micro-agent(s). The preliminary for an agent to release its micro-agents is that species of these micro-agents are sub-species of other species (cf. [Species161#Nesting_species Nesting species]). The released agents won't be micro-agents of the calling agent anymore. Being released from a macro-agent, the micro-agents will change their species and host (macro-agent).

Usages

- We consider the following species. Agents of "C" species can be released from a "B" agent to become agents of "A" species. Agents of "D" species cannot be released from the "A" agent because species "D" has no parent species.

```
species A {
...
}
species B {
...
  species C parent: A {
...
  }
  species D {
...
  }
...
}
```

- To release all "C" agents from a "B" agent, agent "C" has to execute the following statement. The "C" agent will change to "A" agent. 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](#),

Embedments

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

remove

Facets

- `from` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression to remove from the container
- `all` (any type): an expression that evaluates to a container. If it is true and if the value a list, it removes the first instance of each element of the list. If it is true and the value is not a container, it will remove all instances of this value.
- `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

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;

```

- 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();
        remove key:"a" from: sR; // sR.a now equals nil
    }
}

species speciesRemove {
    int a <- 100;
}
```

- This statement can not be used on *matrix*.
- See also: [add](#), [put](#),

Embedments

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

return

Facets

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

Definition

Allows to immediately stop and tell which value to return from the evaluation of the surrounding action or top-level statement (reflex, init, etc.). Usually used within the declaration of an action. For more details about actions, see the following [Section161 section].

Usages

- Example:

```
string foo {
    return "foo";
}

reflex {
    string foo_result <- foo();      // foos_result is now equals to "foo"
}
```

- In the specific case one wants an agent to ask another agent to execute a statement with a return, it can be done similarly to:

```
// In Species A:
string foo_different {
    return "foo_not_same";
}
/// ....
// In Species B:
reflex writing {
    string temp <- some_agent_A.foo_different [];    // temp is now equals to
"foo_not_same"
}
```

Embedments

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

rotation

Facets

- `angle` (any type in [float, int]), (omissible) : Defines the angle of rotation around the axis. No default defined.
- `axis` (point): The axis of rotation, defined by a vector. Default is {0,0,1} (rotation around the z axis) This facet can be complemented by 'distance:' and/or 'location:' to specify from where the target is looked at. If 'target:' is not defined, the default target is the centroid of the world shape.
- `dynamic` (boolean): If true, the rotation is applied every step. Default is false.
- `location` (point): Allows to define the center of the rotation. Default value is not specified is the center of mass of the world (i.e. {width/2, height/2, max(width, height) / 2})

Definition

`camera` allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way. Several preset cameras are provided and accessible in the preferences (to choose the default) or in GAML using the keywords #from_above, #from_left, #from_right, #from_up_right, #from_up_left, #from_front, #from_up_front. These cameras are unlocked (so that they can be manipulated by the user), look at the center of the world from a symbolic position, and the distance between this position and the target is equal to the maximum of the width and height of the world's shape. These preset cameras can be reused when defining new cameras, since their names can become symbolic positions for them. For instance: camera 'my_camera' location: #from_top distance: 10; will lower (or extend) the distance between the camera and the center of the world to 10. camera 'my_camera' locked: true location: #from_up_front target: people(0); will continuously follow the first agent of the people species from the up-front position.

Usages

- See also: [display](#), [agents](#), [chart](#), [event](#), [graphics](#), [display_grid](#), [image_layer](#), [species_layer](#),

Embedments

- The `rotation` statement is of type: **Layer**
- The `rotation` statement can be embedded into: `display`,

- The `rotation` statement embeds statements:
-

rule

Facets

- `name` (an identifier), (omissible) : the identifier of the rule
- `when` (boolean): The condition to fulfill in order to execute the statements embedded in the rule. when: true makes the rule always activable
- `priority` (float): An optional priority for the rule, which is used to sort activable rules and run them in that order

Definition

A simple definition of a rule (set of statements which execution depend on a condition and a priority).

Usages

Embedments

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

rule

Facets

- `name` (an identifier), (omissible) : The name of the rule
- `all` (boolean): add a desire for each belief
- `belief` (predicate): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `desire` (predicate): The mandatory desire
- `desires` (list): The mandatory desires

- `emotion` (emotion): The mandatory emotion
- `emotions` (list): The mandatory emotions
- `ideal` (predicate): The mandatory ideal
- `ideals` (list): The mandatory ideals
- `lifetime` (any type in [int, list]): the lifetime value of the mental state created
- `new_belief` (predicate): The belief that will be added
- `new_beliefs` (list): The belief that will be added
- `new_desire` (predicate): The desire that will be added
- `new_desires` (list): The desire that will be added
- `new_emotion` (emotion): The emotion that will be added
- `new_emotions` (list): The emotion that will be added
- `new_ideal` (predicate): The ideal that will be added
- `new_ideals` (list): The ideals that will be added
- `new_uncertainties` (list): The uncertainty that will be added
- `new_uncertainty` (predicate): The uncertainty that will be added
- `obligation` (predicate): The mandatory obligation
- `obligations` (list): The mandatory obligations
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `remove_belief` (predicate): The belief that will be removed
- `remove_beliefs` (list): The belief that will be removed
- `remove_desire` (predicate): The desire that will be removed
- `remove_desires` (list): The desire that will be removed
- `remove_emotion` (emotion): The emotion that will be removed
- `remove_emotions` (list): The emotion that will be removed
- `remove_ideal` (predicate): The ideal that will be removed
- `remove_ideals` (list): The ideals that will be removed
- `remove_intention` (predicate): The intention that will be removed
- `remove_obligation` (predicate): The obligation that will be removed
- `remove_obligations` (list): The obligation that will be removed

- `remove_uncertainties` (list): The uncertainty that will be removed
- `remove_uncertainty` (predicate): The uncertainty that will be removed
- `strength` (any type in [float, int, list]): The strength of the mental state created
- `threshold` (float): Threshold linked to the emotion.
- `uncertainties` (list): The mandatory uncertainties
- `uncertainty` (predicate): The mandatory uncertainty
- `when` (boolean):

Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

Usages

Embedments

- The `rule` statement is of type: **Single statement**
 - The `rule` statement can be embedded into: simple_bdi, parallel_bdi, Species, Model,
 - The `rule` statement embeds statements:
-

run

Facets

- `name` (string), (omissible) : Indicates the name of the experiment to run
- `of` (string): Indicates the model containing the experiment to run
- `core` (int): Indicates the number of cores to use to run the experiments
- `end_cycle` (int): Indicates the cycle at which the experiment should stop
- `seed` (int): Provides a predetermined seed instead of letting GAMA choose one
- `with_output` (map): *This needs to be documented*
- `with_param` (map): The parameters to pass to the new experiment

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:
-

sanction

Facets

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

Definition

declare the actions an agent execute when enforcing norms of others during a perception

Usages

Embedments

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

save

Facets

- `data` (any type), (omissible) : the data that will be saved to the file or the file itself to save when data is used in its simplest form
- `attributes` (any type in [map, list]): Allows to specify the attributes of a shape file or GeoJson file where agents are saved. Can be expressed as a list of string or as a literal map. When expressed as a list, each value should represent the name of an attribute of the shape or agent. The keys of the map are the names of the attributes that will be present in the file, the values are whatever expressions neeeded to define their value.

- `crs` (any type): the name of the projection, e.g. `crs:"EPSG:4326"` or its EPSG id, e.g. `crs:4326`.
Here a list of the CRS codes (and EPSG id): <http://spatialreference.org>
- `format` (an identifier): a string representing the format of the output file (e.g. "shp", "asc", "geotiff", "png", "text", "csv"). If the file extension is non ambiguous in facet 'to:', this format does not need to be specified. However, in many cases, it can be useful to do it (for instance, when saving a string to a .pgw file, it is always better to clearly indicate that the expected format is 'text').
- `header` (boolean): an expression that evaluates to a boolean, specifying whether the save will write a header if the file does not exist
- `rewrite` (boolean): a boolean expression specifying whether to erase the file if it exists or append data at the end of it. Only applicable to "text" or "csv" files. Default is true
- `to` (string): an expression that evaluates to an string, the path to the file, or directly to a file

Definition

Allows to save data in a file. The type of file can be "shp", "asc", "geotiff", "text" or "csv".

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->" + name + ":" + location) to: "save_data.txt" type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the values of all attributes of all the agents of a species into a csv (with optional attributes):

```
save species_of(self) to: "save_csvfile.csv" type: "csv" header: false;
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp" attributes: ['nameAgent'::name, 'locationAgent'::location] crs: "EPSG:4326";
```

- To save the grid_value attributes of all the cells of a grid into an ESRI ASCII Raster file:

```
save grid to: "save_grid.asc" type: "asc";
```

- To save the grid_value attributes of all the cells of a grid into geotiff:

```
save grid to: "save_grid.tif" type: "geotiff";
```

- To save the grid_value attributes of all the cells of a grid into png (with a worldfile):

```
save grid to: "save_grid.png" type: "image";
```

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

Embedments

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

set

Facets

- `name` (any type), (omissible) : the name of an existing variable or attribute to be modified

- **value** (any type): the value to affect to the variable or attribute

Definition

Allows to assign a value to the variable or attribute specified

Usages

Embedments

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

setup

Facets

Definition

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

Usages

- As every test should be independent from the others, the setup will mainly contain initialization of variables that will be used in each test.

```
species Tester {
    int val_to_test;

    setup {
        val_to_test <- 0;
    }

    test t1 {
        // [set of instructions, including asserts]
    }
}
```

- See also: [test](#), [assert](#),

Embedments

- The `setup` statement is of type: **Sequence of statements or action**
 - The `setup` statement can be embedded into: Species, Experiment, Model,
 - The `setup` statement embeds statements:
-

sobol

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `outputs` (list): The list of output variables to analyse through sobol indexes
- `report` (string): The path to the file where the Sobol report will be written
- `sample` (an identifier): The size of the sample for the sobol sequence
- `path` (string): The path to the saltelli sample csv file. If the file doesn't exist automatic Saltelli sampling will be performed and saved in the corresponding location
- `results` (string): The path to the file where the automatic batch report will be written

Definition

This algorithm runs a Sobol exploration - it has been built upon the moea framework at <https://github.com/MOEAFramework/MOEAFramework> - disabled the repeat facet of the experiment

Usages

- For example:

```
method sobol sample_size:100 outputs:['my_var'] report:'../path/to/report/file.txt';
```

Embedments

- The `sobol` statement is of type: **Batch method**

- The `sobel` statement can be embedded into: Experiment,
 - The `sobel` statement embeds statements:
-

socialize

Facets

- `name` (an identifier), (omissible) : the identifier of the socialize statement
- `agent` (agent): the agent value of the created social link
- `dominance` (float): the dominance value of the created social link
- `familiarity` (float): the familiarity value of the created social link
- `liking` (float): the appreciation value of the created social link
- `solidarity` (float): the solidarity value of the created social link
- `trust` (float): the trust value of the created social link
- `when` (boolean): A boolean value to socialize only with a certain condition

Definition

enables to directly add a social link from a perceived agent.

Usages

- Other examples of use:

```
do socialize;
```

Embedments

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

solve

Facets

- `equation` (an identifier), (omissible) : the equation system identifier to be numerically solved
- `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` (string): integration method (can be one of "Euler", "ThreeEighths", "Midpoint", "Gill", "Luther", "rk4" or "dp853", "AdamsBashforth", "AdamsMoulton", "DormandPrince54", "GraggBulirschStoer", "HighamHall54") (default value: "rk4") or the corresponding constant
- `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
- `nSteps` (float): Adams-Bashforth and Adams-Moulton methods only. The number of past steps used for computation excluding the one being computed (default value: 2)
- `scalAbsoluteTolerance` (float): allowed absolute error (used with dp853 method only)
- `scalRelativeTolerance` (float): allowed relative error (used with dp853 method only)
- `step` (float): (deprecated) integration step, use with fixed step integrator methods (default value: 0.005*step)
- `step_size` (float): integration step, use with fixed step integrator methods (default value: 0.005*step)
- `t0` (float): the first bound of the integration interval (default value: cycle*step, the time at the begining of the current cycle.)
- `tf` (float): the second bound of the integration interval. Can be smaller than t0 for a backward integration (default value: cycle*step, the time at the begining of the current cycle.)

Definition

Solves all equations which matched the given name, with all systems of agents that should solved simultaneously.

Usages

- Other examples of use:

```
solve SIR method: #rk4 step:0.001;
```

Embedments

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

species

Facets

- `name` (an identifier), (omissible) : the identifier of the species
- `cell_height` (float): (grid only), the height of the cells of the grid
- `cell_width` (float): (grid only), the width of the cells of the grid
- `compile` (boolean):
- `control` (skill): defines the architecture of the species (e.g. fsm...)
- `edge_species` (species): In the case of a species defining a graph topology for its instances (nodes of the graph), specifies the species to use for representing the edges
- `file` (file): (grid only), a bitmap file that will be loaded at runtime so that the value of each pixel can be assigned to the attribute 'grid_value'
- `files` (list): (grid only), a list of bitmap file that will be loaded at runtime so that the value of each pixel of each file can be assigned to the attribute 'bands'
- `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)
- `horizontal_orientation` (boolean): (hexagonal grid only),(true by default). Allows use a hexagonal grid with a horizontal or vertical orientation.
- `mirrors` (any type in [list, species]): The species this species is mirroring. The population of this current species will be dependent of that of the species mirrored (i.e. agents creation and death are entirely taken in charge by GAMA with respect to the demographics of the species mirrored). In addition, this species is provided with an attribute called 'target', which allows each agent to know which agent of the mirrored species it is representing.

- `neighbors` (int): (grid only), the chosen neighborhood (4, 6 or 8)
- `optimizer` (string): (grid only), ("A" by default). Allows to specify the algorithm for the shortest path computation ("BF", "Dijkstra", "A" or "JPS*")
- `parallel` (any type in [boolean, int]): (experimental) setting this facet to 'true' will allow this species to use concurrency when scheduling its agents; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet has a default set in the preferences (Under Performances > Concurrency)
- `parent` (species): the parent class (inheritance)
- `schedules` (container): A container of agents (a species, a dynamic list, or a combination of species and containers) , which represents which agents will be actually scheduled when the population is scheduled for execution. Note that the world (or the simulation) is *always* scheduled first, so there is no need to explicitly mention it. Doing so would result in a runtime error. 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;' allows to simulate a model where only the world and the agents of b are scheduled, 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_neighbors_cache` (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbors cache used for grids. Note that if a diffusion of variable occurs, GAMA will emit a warning and automatically switch to a caching version
- `use_regular_agents` (boolean): (grid only),(true by default). Allows to specify if the agents of the grid are regular agents (like those of any other species) or minimal ones (which can't have

sub-populations, can't inherit from a regular species, etc.)

- `virtual` (boolean): whether the species is virtual (cannot be instantiated, but only used as a parent) (false by default)
- `width` (int): (grid only), the width of the grid (in terms of agent number)

Definition

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

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

```
grid ant_grid file: grid_file neighbors: 8 use_regular_agents: false { }
```

Embedments

- The `species` statement is of type: **Species**

- The `species` statement can be embedded into: Model, Environment, Species,
 - The `species` statement embeds statements:
-

species_layer

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 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- `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)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- `size` (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- `trace` (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- `transparency` (float): the transparency level of the layer (between 0 -- opaque -- and 1 -- fully transparent)
- `visible` (boolean): Defines whether this layer is visible or not

Definition

The `species_layer` statement is used using the `species keyword`. It allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

Usages

- The general syntax is:

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

- Species can be superposed on the same plan (be careful with the order, the last one will be above all the others):

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

- Each species layer can be placed at a different z value using the opengl display. `position:{0,0,0}` means the layer will be placed on the ground and `position:{0,0,1}` means it will be placed at an height equal to the maximum size of the environment.

```
display my_display type: opengl{  
    species agent1 aspect: base ;  
    species agent2 aspect: base position:{0,0,0.5};  
    species agent3 aspect: base position:{0,0,1};  
}
```

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

Embedments

- The `species_layer` statement is of type: **Layer**

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

start_simulation

Facets

- `name` (string), (omissible) : The name of the experiment to run
- `of` (string): The path to the model containing the experiment
- `seed` (int): The seed to use for initializing the random number generator of the new experiment
- `with_param` (map): The parameters to pass to the new experiment

Embedments

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

state

Facets

- `name` (an identifier), (omissible) : the identifier of the state
- `final` (boolean): specifies whether the state is a final one (i.e. there is no transition from this state to another state) (default value= false)
- `initial` (boolean): specifies whether the state is the initial one (default value = false)

Definition

A state, like a reflex, can contains several statements that can be executed at each time step by the agent.

Usages

- Here is an exemple integrating 2 states and the statements in the FSM architecture:

```
state s_init initial: true {
    enter {
        write "Enter in" + state;
    }

    write state;

    transition to: s1 when: (cycle > 2) {
        write "transition s_init -> s1";
    }

    exit {
        write "EXIT from "+state;
    }
}
state s1 {

    enter {write 'Enter in '+state;}

    write state;

    exit {write 'EXIT from '+state;}
}
```

- See also: [enter](#), [exit](#), [transition](#),

Embedments

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

status

Facets

- **message** (any type), (omissible) : Allows to display a necessarily short message in the status box in the upper left corner. No formatting characters (carriage returns, tabs, or Unicode characters) should be used, but a background color can be specified. The message will remain in place until it is replaced by another one or by nil, in which case the standard status (number of cycles) will be displayed again
- **color** (rgb): The color used for displaying the background of the status message

Definition

The statement makes the agent output an arbitrary message in the status box.

Usages

- Outputting a message

```
status ("This is my status " + self) color: #yellow;
```

Embedments

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

stochanalyse

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **outputs** (list): The list of output variables to analyse
- **report** (string): The path to the file where the Sobol report will be written

- `results` (string): The path to the file where the automatic batch report will be written
- `sample` (int): The number of sample required , 10 by default
- `sampling` (an identifier): The sampling method to build parameters sets. Available methods are: latinhypercube, orthogonal, factorial, uniform, saltelli, morris

Definition

This algorithm runs an exploration with a given sampling to compute a Stochasticity Analysis

Usages

- For example:

```
method stochanalyse sampling:'latinhypercube' outputs:['my_var'] replicat:10
results:'./path/to/report/file.txt';
```

Embedments

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

switch

Facets

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

Definition

The "switch... match" statement is a powerful replacement for imbricated "if ... else ..." constructs. All the blocks that match are executed in the order they are defined, unless one invokes 'break', in which case the switch statement is exited. The block prefixed by default is executed only if none have matched (otherwise it is not).

Usages

- The prototypical syntax is as follows:

```
switch an_expression {  
    match value1 {...}  
    match_one [value1, value2, value3] {...}  
    match_between [value1, value2] {...}  
    default {...}  
}
```

- Example:

```
switch 3 {  
    match 1 {write "Match 1"; }  
    match 2 {write "Match 2"; }  
    match 3 {write "Match 3"; }  
    match_one [4,4,6,3,7] {write "Match one_of"; }  
    match_between [2, 4] {write "Match between"; }  
    default {write "Match Default"; }  
}
```

- See also: [match](#), [default](#), [if](#),

Embedments

- The `switch` statement is of type: **Sequence of statements or action**
- The `switch` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `switch` statement embeds statements: [default](#), [match](#),

tabu

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max, avr}: the aggregation method

- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `iter_max` (int): number of iterations. this number corresponds to the number of "moves" in the parameter space. For each move, the algorithm will test the whole neighborhood of the current solution, each neighbor corresponding to a particular set of parameters and thus to a run. Thus, there can be several runs per iteration (maximum: $2^{(\text{number of parameters})}$).
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `tabu_list_size` (int): size of the tabu list

Definition

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the tabu statement uses `method tabu` instead of the expected `tabu name: id`:

```
method tabu [facet: value];
```

- For example:

```
method tabu iter_max: 50 tabu_list_size: 5 maximize: food_gathered;
```

Embedments

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

task

Facets

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

Definition

As reflex, a task is a sequence of statements that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler chooses a task to execute based on its current priority weight value.

Usages

Embedments

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

test

Facets

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

Definition

The test statement allows modeler to define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue.

Usages

- An example of use:

```
species Tester {  
    // set of attributes that will be used in test  
  
    setup {  
        // [set of instructions... in particular initializations]  
    }  
  
    test t1 {  
        // [set of instructions, including asserts]  
    }  
}
```

- See also: [setup](#), [assert](#),

Embedments

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

text

Facets

- `message` (any type), (omissible) : the text to display.
- `background` (rgb): The color of the background of the text
- `category` (a label): a category label, used to group parameters in the interface
- `color` (rgb): The color with which the text will be displayed
- `font` (any type in [font, string]): the font used to draw the text, which can be built with the operator "font". ex : font:font("Helvetica", 20 , #bold)

Definition

The statement makes an experiment display text in the parameters view.

Usages

Embedments

- The `text` statement is of type: **Single statement**
 - The `text` statement can be embedded into: Experiment,
 - The `text` statement embeds statements:
-

trace

Facets

Definition

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

Usages

Embedments

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

transition

Facets

- `to` (an identifier): the identifier of the next state
- `when` (boolean), (omissible) : a condition to be fulfilled to have a transition to another given state

Definition

In an FSM architecture, `transition` specifies the next state of the life cycle. The transition occurs when the condition is fulfilled. The embedded statements are executed when the transition is triggered.

Usages

- In the following example, the transition is executed when after 2 steps:

```
state s_init initial: true {
    write state;
    transition to: s1 when: (cycle > 2) {
        write "transition s_init -> s1";
    }
}
```

- See also: [enter](#), [state](#), [exit](#),

Embedments

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

try

Facets

Definition

Allows the agent to execute a sequence of statements and to catch any runtime error that might happen in a subsequent `catch` block, either to ignore it (not a good idea, usually) or to safely stop the model

Usages

- The generic syntax is:

```
try {
    [statements]
}
```

- Optionally, the statements to execute when a runtime error happens in the block can be defined in a following statement 'catch'. The syntax then becomes:

```
try {
    [statements]
}
catch {
    [statements]
}
```

Embedments

- The `try` statement is of type: **Sequence of statements or action**
- The `try` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `try` statement embeds statements: `catch`,

unconscious_contagion

Facets

- `emotion` (emotion): the emotion that will be copied with the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly copy an emotion present in the perceived species.

Usages

- Other examples of use:

```
unconscious_contagion emotion:fearConfirmed;  
unconscious_contagion emotion:fearConfirmed charisma: 0.5 receptivity: 0.5;
```

Embedments

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

user_command

Facets

- `name` (a label), (omissible) : the identifier of the user_command
- `action` (action): the identifier of the action to be executed. This action should be accessible in the context in which the user_command is defined (an experiment, the global section or a species). A special case is allowed to maintain the compatibility with older versions of GAMA, when the user_command is declared in an experiment and the action is declared in 'global'. In that case, all the simulations managed by the experiment will run the action in response to the user executing the command
- `category` (a label): a category label, used to group parameters in the interface
- `color` (rgb): The color of the button to display
- `continue` (boolean): Whether or not the button, when clicked, should dismiss the user panel it is defined in. Has no effect in other contexts (menu, parameters, inspectors)
- `when` (boolean): the condition that should be fulfilled (in addition to the user clicking it) in order to execute this action
- `with` (map): the map of the parameters::values required by the action

Definition

Anywhere in the global block, in a species or in an (GUI) experiment, user_command statements

allows to either call directly an existing action (with or without arguments) or to be followed by a block that describes what to do when this command is run.

Usages

- The general syntax is for example:

```
user_command kill_myself action: some_action with: [arg1::val1, arg2::val2, ...];
```

- See also: [user_init](#), [user_panel](#), [user_input](#),

Embedments

- The `user_command` statement is of type: **Sequence of statements or action**
 - The `user_command` statement can be embedded into: `user_panel`, `Species`, `Experiment`, `Model`,
 - The `user_command` statement embeds statements: [user_input](#),
-

user_init

Facets

- `name` (an identifier), (omissible) : The name of the panel
- `initial` (boolean): Whether or not this panel will be the initial one

Definition

Used in the user control architecture, `user_init` is executed only once when the agent is created. It opens a special panel (if it contains `user_commands` statements). It is the equivalent to the `init` block in the basic agent architecture.

Usages

- See also: [user_command](#), [user_init](#), [user_input](#),

Embedments

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

user_input

Facets

- `init` (any type): the init value
- `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, only for string inputs
- `max` (float): the maximum value
- `min` (float): the minimum value
- `slider` (boolean): Whether to display a slider or not when applicable
- `type` (a datatype identifier): the variable type

Definition

It allows to let the user define the value of a variable.

Usages

- Other examples of use:

```
user_panel "Advanced Control" {  
    user_input "Location" returns: loc type: point <- {0,0};  
    create cells number: 10 with: [location::loc];  
}
```

- See also: [user_command](#), [user_init](#), [user_panel](#),

Embedments

- The `user_input` statement is of type: **Single statement**
 - The `user_input` statement can be embedded into: `user_command`,
 - The `user_input` statement embeds statements:
-

user_panel

Facets

- `name` (an identifier), (omissible) : The name of the panel
- `initial` (boolean): Whether or not this panel will be the initial one

Definition

It is the basic behavior of the user control architecture (it is similar to state for the FSM architecture). This `user_panel` translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each `user_panel`, like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of `user_commands` which describe the different action buttons present in the panel.

Usages

- The general syntax is for example:

```
user_panel default initial: true {
    user_input 'Number' returns: number type: int <- 10;
    ask (number among list(cells)){ do die; }
    transition to: "Advanced Control" when: every (10);
}

user_panel "Advanced Control" {
    user_input "Location" returns: loc type: point <- {0,0};
    create cells number: 10 with: [location::loc];
}
```

- See also: [user_command](#), [user_init](#), [user_input](#),

Embedments

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

using

Facets

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

Definition

`using` is a statement that allows to set the topology to use by its sub-statements. They can gather it by asking the scope to provide it.

Usages

- All the spatial operations are topology-dependent (e.g. neighbors are not the same in a continuous and in a grid topology). So `using` statement allows modelers to specify the topology in which the spatial operation will be computed.

```
float dist <- 0.0;
using topology(grid_ant) {
    d (self.location distance_to target.location);
}
```

Embedments

- The `using` statement is of type: **Sequence of statements or action**
- The `using` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer,
- The `using` statement embeds statements:

Variable_container

Facets

- **name** (a new identifier), (omissible) : The name of the attribute
- **<-** (any type): The initial value of the attribute. Same as init:
- **->** (any type in [int, float, point, date]): Used to specify an expression that will be evaluated each time the attribute is accessed. Equivalent to 'function:'. This facet is incompatible with both 'init:' and 'update:' and 'on_change:' (or the equivalent final block)

- `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. Equivalent to '`->`'. This facet is incompatible with both '`init:`', '`update:`' and '`on_change:`' (or the equivalent final block)
- `index` (a datatype identifier): The type of the key used to retrieve the contents of this attribute
- `init` (any type): The initial value of the attribute. Same as `<-`
- `of` (a datatype identifier): The type of the contents of this container attribute
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `type` (a datatype identifier): The type of the attribute
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute

Definition

Declaration of an attribute of a species or an experiment

Usages

Embedments

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

Variable_number

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `<-` (any type in [int, float, point, date]): The initial value of the attribute. Same as '`init:`'
- `->` (any type in [int, float, point, date]): Used to specify an expression that will be evaluated each time the attribute is accessed. Equivalent to '`function:`'. This facet is incompatible with both '`init:`' and '`update:`' and '`on_change:`' (or the equivalent final block)

- `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, point, date]): Used to specify an expression that will be evaluated each time the attribute is accessed. Equivalent to '`->`'. This facet is incompatible with both '`init:`' and '`update:`'
- `init` (any type in [int, float, point, date]): The initial value of the attribute. Same as '`<-`'
- `max` (any type in [int, float, point, date]): The maximum value this attribute can take. The value will be automatically clamped if it is higher.
- `min` (any type in [int, float, point, date]): The minimum value this attribute can take. The value will be automatically clamped if it is lower.
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `step` (any type in [int, float, point, date]): A discrete step (used in conjunction with `min` and `max`) that constrains the values this variable can take
- `type` (a datatype identifier): The type of the attribute, either '`int`', '`float`', '`point`' or '`date`'
- `update` (any type in [int, float, point, date]): An expression that will be evaluated each cycle to compute a new value for the attribute

Definition

Declaration of an attribute of a species or an experiment; this type of attributes accepts `min:`, `max:` and `step:` facets, automatically clamping the value if it is lower than `min` or higher than `max`.

Usages

Embedments

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

Variable_regular

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `<-` (any type): The initial value of the attribute. Same as `init`:
- `->` (any type in [int, float, point, date]): Used to specify an expression that will be evaluated each time the attribute is accessed. Equivalent to '`function`'. This facet is incompatible with both '`init`' and '`update`' and '`on_change`' (or the equivalent final block)
- `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`', '`update`' and '`on_change`' (or the equivalent final block)
- `index` (a datatype identifier): The type of the index used to retrieve elements if the type of the attribute is a container type
- `init` (any type): The initial value of the attribute. Same as `<-`
- `of` (a datatype identifier): The type of the elements contained in the type of this attribute if it is a container type
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `type` (a datatype identifier): The type of this attribute. Can be combined with facets '`of`' and '`index`' to describe container types
- `update` (any type): An expression that will be evaluated each cycle to compute a new value for the attribute

Definition

Declaration of an attribute of a species or an experiment

Usages

Embedments

- The `Variable_regular` statement is of type: **Variable (regular)**

- The `Variable_regular` statement can be embedded into: Species, Experiment, Model,
 - The `Variable_regular` statement embeds statements:
-

warn

Facets

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

Definition

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

Usages

- Emmitting a warning

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

Embedments

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

write

Facets

- `message` (any type), (omissible) : the message to display. Modelers can add some formatting characters to the message (carriage returns, tabs, or Unicode characters), which will be used accordingly in the console.
- `color` (rgb): The color with wich the message will be displayed. Note that different simulations will have different (default) colors to use for this purpose if this facet is not specified

Definition

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

Usages

- Outputting a message

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

Embedments

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



> GAML References

> **Types**

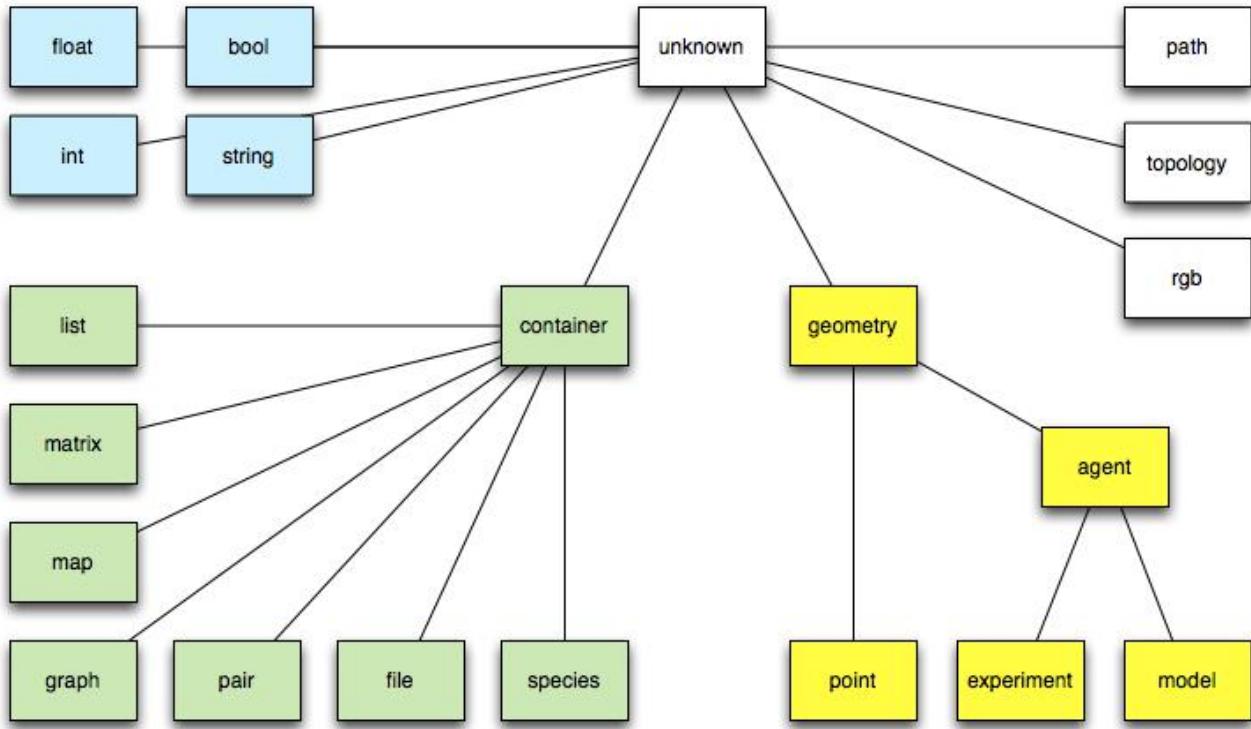
Version: 1.9.1

Types

A variable's or expression's *type* (or *data type*) determines the values it can take, plus the operations that can be performed on or with it. GAML is a statically-typed language, which means that the type of an expression is always known at compile time, and is even enforced with casting operations. There are 4 categories of types:

- primitive types, declared as keyword in the language,
- complex types, also declared as keyword in the language,
- parametric types, a refinement of complex types (mainly children of container) that is dynamically constructed using an enclosing type, a contents type and a key type,
- species types, dynamically constructed from the species declarations made by the modeler (and the built-in species present).

The hierarchy of types in GAML (only primitive and complex types are displayed here, of course, as the other ones are model-dependent) is the following:



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 () -> false
```

[Top of the page](#)

float

- **Definition:** primitive datatype holding floating point values, its absolute value is comprised between 4.9E-324 and 1.8E308.
- **Comments:** this datatype is internally backed up by the Java double datatype.
- **Literal declaration:** decimal notation 123.45 or exponential notation 123e45 are supported.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to float to their operand. Using it is a way to obtain a float constant.

```
float (12) -> 12.0
```

[Top of the page](#)

int

- **Definition:** primitive datatype holding integer values comprised between -2147483648 and 2147483647 (i.e. between -2^{31} and $2^{31} - 1$).
- **Comments:** this datatype is internally backed up by the Java int datatype.
- **Literal declaration:** decimal notation like 1, 256790 or hexadecimal notation like #1209FF are automatically interpreted.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to int to their operand. Using it is a way to obtain an integer constant.

```
int (234.5) -> 234.
```

[Top of the page](#)

string

- **Definition:** a datatype holding a sequence of characters.
- **Comments:** this datatype is internally backed up by the Java String class. However, contrary to Java, strings are considered as a primitive type, which means they do not contain character objects. This can be seen when casting a string to a list using the list operator: the result is a list of one-character strings, not a list of characters.
- **Litteral declaration:** a sequence of characters enclosed in quotes, like 'this is a string'. If one wants to literally declare strings that contain quotes, one has to double these quotes in the declaration. Strings accept escape characters like `\n` (newline), `\r` (carriage return), `\t` (tabulation), as well as any Unicode character (`\uXXXX`).
- **Other declarations:** see string
- **Example:** see [string operators](#).

[Top of the page](#)

Complex built-in types

Contrarily to primitive built-in types, complex types have often various attributes. They can be accessed in the same way as attributes of agents:

```
complex_type nom_var <- init_var;  
ltype_attr attr_var <- nom_var.attr_name;
```

For example:

```
file fileText <- file("../data/cell.Data");
bool fileTextReadable <- fileText.readable;
```

agent

- **Definition:** a generic datatype that represents an agent whatever its actual species.
- **Built-in attributes:** these attributes are common to any agent of the simulation
 - location (type = point): the location of the agent
 - shape (type = geometry): the shape of the agent
 - name (type = string): name of the agent (not necessarily unique in its population)
 - peers (type = list of agents of the same species): the population of agents of the same species, in the same host, minus the receiver agent
 - host (type = agent): the agent that hosts the population of the agent
- **Comments:** This datatype is barely used since species name can be directly used as datatypes themselves.
- **Declaration:** the agent casting operator can be applied to any unknown object to cast it as an agent.

[Top of the page](#)

container

- **Definition:** a generic datatype that represents a collection of data.
- **Comments:** a container variable can be a list, a matrix, a map... Conversely, each list, matrix, and map is a kind of container. In consequence, every container can be used in container-related operators.
- **See also:** [Container operators](#)

- **Declaration:**

```
container c <- [1,2,3];
container c <- matrix [[1,2,3],[4,5,6]];
container c <- map ["x":5, "y":12];
container c <- list species1;
```

[Top of the page](#)

conversation

- **Definition:** a datatype that represents a conversation between agents in a FIPA-ACL interaction. It contains in particular all the exchanged messages.
- **Built-in attributes:**
 - `messages` (type = list of messages): the list of messages that compose this conversation
 - `protocol` (type = string): the name of the protocol followed by the conversation
 - `initiator` (type = agent): the agent that has initiated this conversation
 - `participants` (type = list of agents): the list of agents that participate to this conversation
 - `ended` (type = bool): whether this conversation has ended or not

[Top of the page](#)

date

- **Definition:** a datatype that represents a date (day, month, year, and time). Any date variable can be created in the model. 2 built-in variables exist in a model: `starting_date` (containing the date at the start of the simulation), `current_date` (the date at the current step of the simulation, it is updated

automatically from `starting_date`, `step` and `time`). In addition, the constant `#now` contains the current (real) date. Many operators can be used on dates (such as `+`, `-`, `add_years`...).

- **Built-in attributes:**

- `year` (type = int): the year component of the date
- `month` (type = int): the month component of the date (1-12)
- `day` (type = int): the day component of the date (1-31)
- `hour` (type = int): the number of hours in the current day of this date (0-23)
- `minute` (type = int): the number of minutes in the current hour of this date (0-59)
- `second` (type = int): the number of seconds in the current minute of this date (0-59)
- `day_of_year` (type = int): the current day number in the year of this date (1-366)
- `day_of_week` (type = int): the index of the day in the current week (with Monday being 1)
- `second_of_day` (type = int): the index of seconds in the day of this date (0-86399)
- `minute_of_day` (type = int): the index of the minute in the day of this date (0-1439)
- `week_of_year` (type = int): the index of the week in the current year (1-52)
- `days_in_month` (type = int): the number of days in the current month of this date (28-31)
- `days_in_year` (type = int): the number of days in the current year of this date (365-366)
- `leap` (type = bool): returns true if the year is a leap year
- `date` (type = date): returns a new date object with only the year-month-day

components of this date

- **Declaration:** a date can be created using different sets of information.
 - The simplest one consists in using a list of int values: [year, month of the year, day of the month, hour of the day, minute of the hour, second of the minute] or simply [year, month of the year, day of the month] (time is set to 0 in this case).

```
date my_date <- date([2010,3,23,17,30,10]); // the 23th of March  
2010, at 17:30:10  
date my_date2 <- date([2010,3,23]); // the 23th of March 2010, at  
00:00:00
```

- Another way consists in using a string with the good format. The following one is perhaps the most complete, with year, month, day, hour, minute, second, and also the time zone.

```
date my_date <- date("2010-3-23T17:30:10+07:00");
```

- But the following ones can also be used:

```
// without time zone:  
my_date3 <- date("2010-03-23 17:30:10");  
//Dates (without time)  
my_date3 <- date("20100323");  
my_date3 <- date("2010-03-23");  
// Dates using some patterns:  
my_date3 <- date("03 23 2010", "MM dd yyyy");  
my_date3 <- date("01 23 20", "HH mm ss");
```

- **Note:** date creation format has been defined in an ISO norm. More examples can be found here: <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns>

- **See also:** [Date operators](#)

[Top of the page](#)

field

- **Definition:** Fields are two-dimensional matrices holding float values. They can be easily created from arbitrary sources (grid, raster or DEM files, matrices grids) and of course by hand. The values they hold are accessible by agents like grids are, using their current location. They can be the target of the 'diffuse' statement and can be displayed using the 'mesh' layer definition. As such, they represent a lightweight alternative to grids, as they hold spatialized discrete values without having to build agents, which can be particularly interesting for models with large raster data. Several fields can of course be defined, and it makes sense to define them in the global section as, for the moment, they cover by default the whole environment, exactly like grids, and are created alongside them.
- **Built-in attributes:** a field is a kind of matrix, it thus inherits from the matrix's attributes.
 - dimension (type = point): the dimension (columns x rows) of the receiver matrix
 - columns (type = int): the number of columns of the receiver matrix
 - rows (type = int): the number of rows of the receiver matrix
 - cell_size (type = point): the dimension of an individual cell as a point (width, height). Setting it will only change the interpretation made by the field of the values it contains, but not the values themselves.
 - bands (type = list of field): The list of bands that are optionally present in the field. The first band is the primary field itself, and each of these bands is a field w/o bands
 - no_data (type = float): the value that indicates the absence of data. Setting it

will only change the interpretation made by the field of the values it contains, but not the values themselves.

- **See also:** [Field operators](#)
- **Declaration:** a field can be created from a raster datafile (such as .asc or .tif files), a matrix or be specifying its dimensions.
 - a field can be created from a raster datafile

```
// Initialize a field from a asc simple raster file
field field_from_asc <- field(grid_file("includes/grid.asc"));

// initialize using a tiff raster file
field field_from_tiff <- field(grid_file("includes/
Lesponne.tif"));
```

- a field can be created manually:

```
// Init from a user defined matrix
field field_from_matrix <-
field(matrix([[1,2,3],[4,5,6],[7,8,9]]));

// init an empty field of a given size
field empty_field_from_size <- field(10,10);

// init a field for of a given value
field full_field_from_size<- field(10,10,1.0);

// init a field of given size, with a given value and no data
field full_field_from_size_with_nodata <- field (1,1,1.0,0.0);
```

- a field can be created from a grid of cells, the value stored will be the grid's grid_value attribute

```
global {
```

file

- **Definition:** a datatype that represents a file.
- **Built-in attributes:**
 - name (type = string): the name of the represented file (with its extension)
 - extension(type = string): the extension of the file
 - path (type = string): the absolute path of the file
 - readable (type = bool, read-only): a flag expressing whether the file is readable
 - writable (type = bool, read-only): a flag expressing whether the file is writable
 - exists (type = bool, read-only): a flag expressing whether the file exists
 - is_folder (type = bool, read-only): a flag expressing whether the file is folder
 - contents (type = container): a container storing the content of the file
- **Comments:** a variable with the `file` type can handle any kind of file (text, image or shape files...). The type of the `content` attribute will depend on the kind of file. Note that the allowed kinds of file are the followings:
 - text files: files with the extensions `.txt`, `.data`, `.csv`, `.text`, `.tsv`, `.asc`. The `content` is by default a list of string.
 - image files: files with the extensions `.pgm`, `.tif`, `.tiff`, `.jpg`, `.jpeg`, `.png`, `.gif`, `.pict`, `.bmp`. The `content` is by default a matrix of int.
 - shapefiles: files with the extension `.shp`. The `content` is by default a list of geometry.
 - properties files: files with the extension `.properties`. The `content` is by default a map of string::string.
 - folders. The `content` is by default a list of string.
- **Remark:** Files are also a particular kind of container and can thus be read,

written or iterated using the container operators and commands.

- **See also:** [File operators](#)
- **Declaration:** a file can be created using the generic `file` (that opens a file in read only mode and tries to determine its contents), `folder` or the `new_folder` (to open an existing folder or create a new one) unary operators. But things can be specialized with the combination of the `read/write` and `image/text/shapefile/properties` unary operators.

```
folder(a_string) // returns a file managing a existing folder  
file(a_string) // returns any kind of file in read-only mode  
read(text(a_string)) // returns a text file in read-only mode  
read(image(a_string)) // does the same with an image file.  
write(properties(a_string)) // returns a property file which is  
available for writing  
// (if it exists, contents will be  
appended unless it is cleared  
// using the standard container  
operations).
```

[Top of the page](#)

geometry

- **Definition:** a datatype that represents a vector geometry, i.e. a list of georeferenced points.
- **Built-in attributes:**
 - location (type = point): the centroid of the geometry
 - area (type = float): the area of the geometry
 - perimeter (type = float): the perimeter of the geometry
 - holes (type = list of geometry): the list of the hole inside the given geometry
 - contour (type = geometry): the exterior ring of the given geometry and of his

holes

- envelope (type = geometry): the geometry bounding box
- width (type = float): the width of the bounding box
- height (type = float): the height of the bounding box
- points (type = list of point): the set of the points composing the geometry
- **Comments:** a geometry can be either a point, a polyline or a polygon. Operators working on geometries handle transparently these three kinds of geometry. The envelope (a.k.a. the bounding box) of the geometry depends on the kind of geometry:
 - If this Geometry is the empty geometry, it is an empty point.
 - If the Geometry is a point, it is a non-empty point.
 - Otherwise, it is a Polygon whose points are (minx, miny), (maxx, miny), (maxx, maxy), (minx, maxy), (minx, miny).
- **See also:** [Spatial operators](#)
- **Declaration:** geometries can be built from a point, a list of points, or by using specific operators (circle, square, triangle...).

```
geometry varGeom <- circle(5);
geometry polygonGeom <- polygon([{3,5}, {5,6},{1,4}]);
```

[Top of the page](#)

graph

- **Definition:** a datatype that represents a graph composed of vertices linked by edges.
- **Built-in attributes:**
 - edges(type = list of agent/geometry): the list of all edges
 - vertices(type = list of agent/geometry): the list of all vertices

- circuit (type = path): an approximate minimal traveling salesman tour (hamiltonian cycle)
 - spanning_tree (type = list of agent/geometry): minimum spanning tree of the graph, i.e. a sub-graph such as every vertex lies in the tree, and as much edges lies in it but no cycles (or loops) are formed.
 - connected(type = bool): test whether the graph is connected
- **Remark:**
- graphs are also a particular kind of container and can thus be manipulated using the container operators and commands.
 - This algorithm used to compute the circuit requires that the graph be complete and the triangle inequality exists (if x,y,z are vertices then $d(x,y)+d(y,z) \geq d(x,z)$ for all x,y,z) then this algorithm will guarantee a hamiltonian cycle such that the total weight of the cycle is less than or equal to double the total weight of the optimal hamiltonian cycle.
 - The computation of the spanning tree uses an implementation of the Kruskal's minimum spanning tree algorithm. If the given graph is connected it computes the minimum spanning tree, otherwise it computes the minimum spanning forest.

• **See also:** [Graph operators](#)

- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create road from: shape_file_road;
graph the_graph <- as_edge_graph(road);

graph([1,9,5])      --: ([1: in[] + out[], 5: in[] + out[], 9: in[]
+ out[], []])
graph([node(0), node(1), node(2)]      // if node is a species
graph(['a':345, 'b':13])   --: ([b: in[] + out[b]:13], a: in[] +
out[a]:345, 13: in[b]:13 + out[], 345: in[a]:345 + out[],
```

[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 literally; 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 built literally from a point, or a string, or any other element by using the list casting operator.

```
list (1) -> [1]
```

```
list<int> myList <- [1,2,3,4];  
myList[2] => 3
```

[Top of the page](#)

map

- **Definition:** a composite datatype holding an ordered collection of pairs (a key, and its associated value).
- **Built-in attributes:**
 - keys (type = list): the list of all keys
 - values (type = list): the list of all values
 - pairs (type = list of pairs): the list of all pairs key::value
- **Comments:** maps are more or less equivalent to instances of Hashtable in Java (although they are backed up by a specific class).
- **Remark:** maps can contain values of any datatypes, including other maps or lists. Maps are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of pair expressions separated by commas, enclosed in square brackets; each pair is represented by a key and a value separated by ::. An example of map is [agentA::'big', agentB::'small', agentC::'big']. An empty map is noted [].
- **Other declarations:** lists can be built literally 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).

- **Built-in attributes:**
 - dimension (type = point): the dimension (columns x rows) of the receiver matrix
 - columns (type = int): the number of columns of the receiver matrix
 - rows (type = int): the number of rows of the receiver matrix
- **Comments:** Matrices are fixed-size structures that can be accessed by index (point for two-dimension matrices, integer for vectors).
- **Litteral declaration:** Matrices cannot be defined literally. One-dimension 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 rows, where
each cell is initialized to 0.0
matrix mat2 <- 0.0 as_matrix({10,5});
// builds a two-dimensions matrix with 2 columns and 3 rows, with
initialized cells
matrix mat3 <- matrix([[ "c11", "c12", "c13"], [ "c21", "c22", "c23"]]);
    -> c11;c21
        c12;c22
        c13;c23
```

[Top of the page](#)

message

- **Definition:** a datatype containing a message (sent during a communication, such as the one sent/received in a FIPA interaction).
- **Built-in attributes:**

- contents (type = unknown): the contents of this message, as a list of arbitrary objects
- sender (type = unknown): the sender that has sent this message
- unread (type = bool): whether this message is unread or not
- emission_timestamp (type = int): the emission time stamp of this message (I.e. at what cycle it has been emitted)
- reception_timestamp (type = int): the reception time stamp of this message (I.e. at what cycle it has been received)

pair

- **Definition:** a datatype holding a key and its associated value.
- **Built-in attributes:**
 - key (type = unknown, read-only): the key of the pair, i.e. the first element of the pair
 - value (type = unknown, read-only): 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 container operators and commands.
- **Litteral declaration:** a pair is defined by a key and a value separated by `::`. The type of the key and value can also be specified.

```
pair testPair <- "key)::56;
pair<string,int> pairWithType <- "tot)::23;
```

[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 depend 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 literally. 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 )
```

[Top of the page](#)

point

- **Definition:** a datatype normally holding two positive float values. Represents the absolute coordinates of agents in the model.
- **Built-in attributes:**
 - x (type = float): coordinate of the point on the x-axis
 - y (type = float): coordinate of the point on the y-axis
- **Comments:** point coordinates should be positive, if a negative value is used in its declaration, the point is built with the absolute value.
- **Remark:** points are particular cases of geometries and containers. Thus they have also all the built-in attributes of both the geometry and the container datatypes and can be used with every kind of operator or command admitting geometry and container.
- **Litteral declaration:** two numbers, separated by a comma, enclosed in braces, like {12.3, 14.5}
- **Other declarations:** points can be built literally 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:** rgb is also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** there exist a 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 takes 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 in 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.
- Literal declaration: the name of a declared species is already a literal declaration of species.
- Other declarations: the species casting operator, or its variant called species_of can be applied to an agent in order to get its species.

[Top of the page](#)

Species names as types

Once a species has been declared in a model, it automatically becomes a datatype.

This means that:

- It can be used to declare variables, parameters or constants,
- It can be used as an operand to commands or operators that require species parameters,
- It can be used as a casting operator (with the same capabilities as the built-in type agent)

In the simple following example, we create a set of "humans" and initialize a random "friendship network" among them. See how the name of the species, human, is used in the create command, as an argument to the list casting operator, and as the type

of the variable named friend.

```
global {
    init {
        create human number: 10;
        ask human {
            friend <- one_of (human - self);
        }
    }
}
entities {
    species human {
        human friend <- nil;
    }
}
```

[Top of the page](#)

topology

- **Definition:** a topology is basically on neighborhoods, distance,... structures in which agents evolves. It is the environment or the context in which all these values are computed. It also provides the access to the spatial index shared by all the agents. And it maintains a (eventually dynamic) link with the 'environment' which is a geometrical border.
- **Built-in attributes:**
 - places(type = container): the collection of places (geometry) defined by this topology.
 - environment(type = geometry): the environment of this topology (i.e. the geometry that defines its boundaries)
- **Comments:** the attributes `places` depends on the kind of the considered 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...)
- **Declaration:** To create a topology, we can use the `topology` unary casting operator applied to:
 - an agent: returns a continuous topology built from the agent's geometry
 - a species name: returns the topology defined for this species population
 - a geometry: returns a continuous topology built on this geometry
 - a geometry container (list, map, shapefile): returns an half-discrete (with corresponding places), half-continuous topology (to compute distances...)
 - a geometry matrix (i.e. a grid): returns a grid topology which computes specifically neighborhood and distances
 - a geometry graph: returns a graph topology which computes specifically neighborhood and distances More complex topologies can also be built using dedicated operators, e.g. to decompose a geometry...

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};
```

How to use this species to create 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;
}
}
```

Version: 1.9.1

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
```

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
Polygon
```

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile.

In order to know how to load a file, GAMA analyzes its extension. For instance for a file with a ".csv" extension, GAMA knows that the file is a **csv** one and will try to split each line with the , separator. However, if the modeler wants to split each line with a different separator (for instance ;) or load it as a text file, he/she will have to use a specific file operator.

Indeed, GAMA integrates specific operators corresponding to different types of files.

Table of contents

- File Types
 - Text File
 - Extensions
 - Content
 - Operators
 - CSV File
 - Extensions
 - Content
 - Operators
 - Shapefile
 - Extensions
 - Content
 - Operators
 - OSM File
 - Extensions
 - Content
 - Operators
 - Grid File
 - Extensions
 - Content
 - Operators
 - Image File
 - Extensions
 - Content
 - Operators

- SVG File
 - Extensions
 - Content
 - Operators
- Property File
 - Extensions
 - Content
 - Operators
- R File
 - Extensions
 - Content
 - Operators
- 3DS File
 - Extensions
 - Content
 - Operators
- OBJ File
 - Extensions
 - Content
 - Operators

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 objects: each row of the matrix is a line of the file; each column of the matrix is values delimited by the separator. By default, the delimiter is the "," and the datatype depends on the dataset. 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
...
```

To manipulate easily the data, we can consider the `contents` of the data file, that is a matrix. As an example, we can access the number of lines and columns of a data file named `my_file` with `my_file.contents.dimension`.

Operators

There are many operators available to load a csv_file.

- **csv_file(string path)**: load a file (with an authorized extension) as a csv file with default separator (","), and no assumption on the type of data.
- **csv_file(string path,bool header)"**: load a file as a CSV file with the default separator (coma), with specifying if the model has a header or not (boolean), and no assumption on the type of data.
- **csv_file(string path, string separator)**: load a file (with an authorized extension) as a csv file with the given separator, without making any assumption

on the type of data. Headers should be detected automatically if they exist.

```
file my_file <- csv_file("../includes/data.csv", ";");
```

- **csv_file(string path, string separator, bool header)**: load a file (with an authorized extension) as a csv file, specifying (1) the separator used; (2) if the model has a header or not, without making any assumption on the type of data.
- **csv_file(string path, string separator, string text_qualifier, bool header)**: load a file as a csv file specifying (1) the separator used; (2) the text qualifier used; (3) if the model has a header or not, without making any assumption on the type of data",
- **csv_file(string path, string separator, type datatype)**: load a file as a csv file specifying a given separator, no header, and the type of data. No text qualifier will be used.

```
file my_file <- csv_file("../includes/data.csv", ";", int);
```

- **csv_file(string path, string separator, string text_qualifier, type datatype)**: load a file as a csv file specifying the separator, text qualifier to use, and the type of data to read. Headers should be detected automatically if they exist.
- **csv_file(string path, string separator, type datatype, bool header)**: load a file as a csv file specifying the given separator, the type of data, with specifying if the model has a header or not (boolean). No text qualifier will be used".
- **csv_file(string path, string separator, type datatype, bool header, point dimensions)**: load a file as a csv file specifying a given separator, the type of data, with specifying the number of cols and rows taken into account. No text qualifier will be used.
- **csv_file(string path, matrix content)**: This file constructor allows to store a matrix in a CSV file (it does not save it - just store it in memory)

Finally, it is possible to check whether a file is a csv file:

- **is_csv(op)**: tests whether the operand is a csv file

Shapefile

Shapefiles are classical GIS data files. A shapefile is not simple file, but a set of several files (source: wikipedia):

- Mandatory files :
 - .shp - shape format; the feature geometry itself
 - .shx - shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly
 - .dbf - attribute format; columnar attributes for each shape, in dBase IV format
- Optional files :
 - .prj - projection format; the coordinate system and projection information, a plain text file describing the projection using well-known text format
 - .sbn and .sbx - a spatial index of the features
 - .fbn and .fbx - a spatial index of the features for shapefiles that are read-only
 - .ain and .aih - an attribute index of the active fields in a table
 - .ixs - a geocoding index for read-write shapefiles
 - .mxs - a geocoding index for read-write shapefiles (ODB format)
 - .atx - an attribute index for the .dbf file in the form of shapefile.columnname.atx (ArcGIS 8 and later)
 - .shp.xml - geospatial metadata in XML format, such as ISO 19115 or other XML schema
 - .cpg - used to specify the code page (only for .dbf) for identifying the character encoding to be used

More details about shapefiles can be found [here](#).

Extensions

Here the list of possible extension for shapefile:

- "shp"

Content

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile. For example:

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

will give:

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

Note that the attributes of each object of the shapefile are stored in their

corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of 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 an 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 corresponding attributes.

Operators

List of operators related to osm file:

- **osm_file(string path)**: load a file (with an authorized extension) as an 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 become a geometry.
- **osm_file(string path, string code)**: load a file (with an authorized extension) as an 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 become a geometry.
- **osm_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as an 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 become 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 an 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  
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 {
```

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:

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

Version: 1.9.1

Expressions

Expressions in GAML are the value part of the [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 [variable](#), an [attribute](#) or the application of one or several [operators](#) to compose a complex expression.

Version: 1.9.1

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.

Table of contents

- [Literals](#)
 - [Simple Types](#)
 - [Literal Constructors](#)
 - [Universal Literal](#)

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.

Version: 1.9.1

Units and constants

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

Introduction

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 (`°` 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 ...
```

3D

- `#ambient`, value= Ambient light, Comment: Represent the 'ambient' type of light
- `#direction`, value= Directional light, Comment: Represent the 'direction' type of light
- `#from_above`, value= From above, Comment: Represent the position of the camera, above the scene
- `#from_front`, value= From front, Comment: Represent the position of the camera, in front of the scene
- `#from_left`, value= From left, Comment: Represent the position of the camera, on the left of the scene

- `#from_right`, value= From right, Comment: Represent the position of the camera, on the right of the scene
 - `#from_up_front`, value= From up front, Comment: Represent the position of the camera, in front and slightly above the scene
 - `#from_up_left`, value= From up left, Comment: Represent the position of the camera, on the left, slightly above the scene
 - `#from_up_right`, value= From up right, Comment: Represent the position of the camera on the right, slightly above the scene
 - `#isometric`, value= Isometric, Comment: Represent the position of the camera, on the left of the scene
 - `#point`, value= Point light, Comment: Represent the 'point' type of light
 - `#spot`, value= Spot light, Comment: Represent the 'spot' type of light
-

Constants

- `#AdamsBashforth`, value= AdamsBashforth, Comment: AdamsBashforth solver
- `#AdamsMoulton`, value= AdamsMoulton, Comment: AdamsMoulton solver
- `#AStar`, value= AStar, Comment: AStar shortest path computation algorithm
- `#BellmannFord`, value= BellmannFord, Comment: BellmannFord shortest path computation algorithm
- `#Bhandari`, value= Bhandari, Comment: Bhandari K shortest paths computation algorithm
- `#BidirectionalDijkstra`, value= BidirectionalDijkstra, Comment: BidirectionalDijkstra shortest path computation algorithm
- `#CHBidirectionalDijkstra`, value= CHBidirectionalDijkstra, Comment: CHBidirectionalDijkstra shortest path computation algorithm
- `#current_error`, value= , Comment: The text of the last error thrown during the

current execution

- `#DeltaStepping`, value= DeltaStepping, Comment: DeltaStepping shortest path computation algorithm
- `#Dijkstra`, value= Dijkstra, Comment: Dijkstra shortest path computation algorithm
- `#DormandPrince54`, value= DormandPrince54, Comment: DormandPrince54 solver
- `#dp853`, value= dp853, Comment: dp853 solver
- `#e`, value= 2.718281828459045, Comment: The e constant
- `#Eppstein`, value= Eppstein, Comment: Eppstein K shortest paths computation algorithm
- `#Euler`, value= Euler, Comment: Euler solver
- `#FloydWarshall`, value= FloydWarshall, Comment: FloydWarshall shortest path computation algorithm
- `#Gill`, value= Gill, Comment: Gill solver
- `#GraggBulirschStoer`, value= GraggBulirschStoer, Comment: GraggBulirschStoer solver
- `#HighamHall54`, value= HighamHall54, Comment: HighamHall54 solver
- `#infinity`, value= Infinity, Comment: A constant holding the positive infinity of type float (Java Double.POSITIVE_INFINITY)
- `#Luther`, value= Luther, Comment: Luther solver
- `#max_float`, value= 1.7976931348623157E308, Comment: A constant holding the largest positive finite value of type float (Java Double.MAX_VALUE)
- `#max_int`, value= 2147483647, Comment: A constant holding the maximum value an int can have (Java Integer.MAX_VALUE)
- `#Midpoint`, value= Midpoint, Comment: Midpoint solver
- `#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= -2147483648, 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)
 - `#NBASTar`, value= NBASTar, Comment: NBASTar shortest path computation algorithm
 - `#NBASTarApprox`, value= NBASTarApprox, Comment: NBASTarApprox shortest path computation algorithm
 - `#pi`, value= 3.141592653589793, Comment: The PI constant
 - `#rk4`, value= rk4, Comment: rk4 solver
 - `#Suurballe`, value= Suurballe, Comment: Suurballe K shortest paths computation algorithm
 - `#ThreeEighths`, value= ThreeEighths, Comment: ThreeEighths solver
 - `#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
 - `#TransitNodeRouting`, value= TransitNodeRouting, Comment: TransitNodeRouting shortest path computation algorithm
 - `#Yen`, value= Yen, Comment: Yen K shortest paths computation algorithm
-

Constants

- `#AdamsBashforth`, value= AdamsBashforth, Comment: AdamsBashforth solver
- `#AdamsMoulton`, value= AdamsMoulton, Comment: AdamsMoulton solver
- `#AStar`, value= AStar, Comment: AStar shortest path computation algorithm
- `#BellmannFord`, value= BellmannFord, Comment: BellmannFord shortest path

computation algorithm

- `#Bhandari`, value= Bhandari, Comment: Bhandari K shortest paths computation algorithm
- `#BidirectionalDijkstra`, value= BidirectionalDijkstra, Comment: BidirectionalDijkstra shortest path computation algorithm
- `#CHBidirectionalDijkstra`, value= CHBidirectionalDijkstra, Comment: CHBidirectionalDijkstra shortest path computation algorithm
- `#current_error`, value= , Comment: The text of the last error thrown during the current execution
- `#DeltaStepping`, value= DeltaStepping, Comment: DeltaStepping shortest path computation algorithm
- `#Dijkstra`, value= Dijkstra, Comment: Dijkstra shortest path computation algorithm
- `#DormandPrince54`, value= DormandPrince54, Comment: DormandPrince54 solver
- `#dp853`, value= dp853, Comment: dp853 solver
- `#e`, value= 2.718281828459045, Comment: The e constant
- `#Eppstein`, value= Eppstein, Comment: Eppstein K shortest paths computation algorithm
- `#Euler`, value= Euler, Comment: Euler solver
- `#FloydWarshall`, value= FloydWarshall, Comment: FloydWarshall shortest path computation algorithm
- `#Gill`, value= Gill, Comment: Gill solver
- `#GraggBulirschStoer`, value= GraggBulirschStoer, Comment: GraggBulirschStoer solver
- `#HighamHall54`, value= HighamHall54, Comment: HighamHall54 solver
- `#infinity`, value= Infinity, Comment: A constant holding the positive infinity of type float (Java Double.POSITIVE_INFINITY)

- `#Luther`, value= Luther, Comment: Luther solver
- `#max_float`, value= 1.7976931348623157E308, Comment: A constant holding the largest positive finite value of type float (Java Double.MAX_VALUE)
- `#max_int`, value= 2147483647, Comment: A constant holding the maximum value an int can have (Java Integer.MAX_VALUE)
- `#Midpoint`, value= Midpoint, Comment: Midpoint solver
- `#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= -2147483648, 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)
- `#NBAStar`, value= NBAStar, Comment: NBAStar shortest path computation algorithm
- `#NBAStarApprox`, value= NBAStarApprox, Comment: NBAStarApprox shortest path computation algorithm
- `#pi`, value= 3.141592653589793, Comment: The PI constant
- `#rk4`, value= rk4, Comment: rk4 solver
- `#Suurballe`, value= Suurballe, Comment: Suurballe K shortest paths computation algorithm
- `#ThreeEighths`, value= ThreeEighths, Comment: ThreeEighths solver
- `#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
- `#TransitNodeRouting`, value= TransitNodeRouting, Comment: TransitNodeRouting shortest path computation algorithm
- `#Yen`, value= Yen, Comment: Yen K shortest paths computation algorithm

Graphics units

- `#bold`, value= 1, Comment: This constant allows to build a font with a bold face.
Can be combined with `#italic`
- `#bottom_center`, value= No Default Value, Comment: Represents an anchor situated at the center of the bottom side of the text to draw
- `#bottom_left`, value= No Default Value, Comment: Represents an anchor situated at the bottom left corner of the text to draw
- `#bottom_right`, value= No Default Value, Comment: Represents an anchor situated at the bottom right corner of the text to draw
- `#camera_location`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current position of the camera as a point
- `#camera_orientation`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current orientation of the camera as a point
- `#camera_target`, value= No Default Value, Comment: This unit, only available when running aspects or declaring displays, returns the current target of the camera as a point
- `#center`, value= No Default Value, Comment: Represents an anchor situated at the center of the text to draw
- `#display_height`, value= 1.0, Comment: This constant is only accessible in a graphical context: display, graphics...
- `#display_width`, value= 1.0, Comment: This constant is only accessible in a graphical context: display, graphics...
- `#flat`, value= 2, Comment: This constant represents a flat line buffer end cap style

- `#fullscreen`, value= false, Comment: This unit, only available when running aspects or declaring displays, returns whether the display is currently fullscreen or not
- `#hidpi`, value= false, Comment: This unit, only available when running aspects or declaring displays, returns whether the display is currently in HiDPI mode or not
- `#horizontal`, value= 3, Comment: This constant represents a layout where all display views are aligned horizontally
- `#italic`, value= 2, Comment: This constant allows to build a font with an italic face. Can be combined with `#bold`
- `#left_center`, value= No Default Value, Comment: Represents an anchor situated at the center of the left side of the text to draw
- `#none`, value= 0, Comment: This constant represents the absence of a predefined layout
- `#pixels` (#px), value= 1.0, Comment: This unit, only available when running aspects or declaring displays, returns a dynamic value instead of a fixed one. px (or pixels), returns the value of one pixel on the current view in terms of model units.
- `#plain`, value= 0, Comment: This constant allows to build a font with a plain face
- `#right_center`, value= No Default Value, Comment: Represents an anchor situated at the center of the right side of the text to draw
- `#round`, value= 1, Comment: This constant represents a round line buffer end cap style
- `#split`, value= 2, Comment: This constant represents a layout where all display views are split in a grid-like structure
- `#square`, value= 3, Comment: This constant represents a square line buffer end cap style
- `#stack`, value= 1, Comment: This constant represents a layout where all display views are stacked

- `#top_center`, value= No Default Value, Comment: Represents an anchor situated at the center of the top side of the text to draw
 - `#top_left`, value= No Default Value, Comment: Represents an anchor situated at the top left corner of the text to draw
 - `#top_right`, value= No Default Value, Comment: Represents an anchor situated at the top right corner of the text to draw
 - `#user_location`, value= No Default Value, Comment: This unit contains in permanence the location of the mouse on the display in which it is situated. The latest location is provided when it is out of a display
 - `#vertical`, value= 4, Comment: This constant represents a layout where all display views are aligned vertically
 - `#zoom`, value= 1.0, Comment: This unit, only available when running aspects or declaring displays, returns the current zoom level of the display as a positive float, where 1.0 represent the neutral zoom (100%)
-

Length units

- `#μm` (#micrometer,#micrometers), value= 1.0E-6, Comment: micrometer unit
- `#cm` (#centimeter,#centimeters), value= 0.01, Comment: centimeter unit
- `#dm` (#decimeter,#decimeters), value= 0.1, Comment: decimeter unit
- `#foot` (#feet,#ft), value= 0.3048, Comment: foot unit
- `#inch` (#inches), value= 0.02540000000000002, Comment: inch unit
- `#km` (#kilometer,#kilometers), value= 1000.0, Comment: kilometer unit
- `#m` (#meter,#meters), value= 1.0, Comment: meter: the length basic unit
- `#mile` (#miles), value= 1609.344, Comment: mile unit
- `#mm` (#milimeter,#milimeters), value= 0.001, Comment: millimeter unit
- `#nm` (#nanometer,#nanometers), value= 9.99999999999999E-10, Comment:

nanometer unit

- **#yard** (#yards), value= 0.9144, Comment: yard unit
-

Surface units

- **#m2**, value= 1.0, Comment: square meter: the basic unit for surfaces
 - **#sqft** (#square_foot,#square_feet), value= 0.09290304, Comment: square foot unit
 - **#sqin** (#square_inch,#square_inches), value= 6.451600000000001E-4, Comment: square inch unit
 - **#sqmi** (#square_mile,#square_miles), value= 2589988.110336, Comment: square mile unit
-

Time units

- **#custom**, value= CUSTOM, Comment: custom: a custom date/time pattern that can be defined in the preferences of GAMA and reused in models
- **#cycle** (#cycles), value= 1, Comment: cycle: the discrete measure of time in the simulation. Used to force a temporal expression to be expressed in terms of cycles rather than seconds
- **#day** (#d,#days), value= 86400.0, Comment: day time unit: defines an exact duration of 24 hours
- **#epoch**, value= No Default Value, Comment: The epoch default starting date as defined by the ISO format (1970-01-01T00:00Z)
- **#h** (#hour,#hours), value= 3600.0, Comment: hour time unit: defines an exact duration of 60 minutes

- **#iso_local**, value= ISO_LOCAL_DATE_TIME, Comment: iso_local: the standard ISO 8601 output / parsing format for local dates (i.e. with no time-zone information)
 - **#iso_offset**, value= ISO_OFFSET_DATE_TIME, Comment: iso_offset: the standard ISO 8601 output / parsing format for dates with a time offset
 - **#iso_zoned**, value= ISO_ZONED_DATE_TIME, Comment: iso_zoned: the standard ISO 8601 output / parsing format for dates with a time zone
 - **#minute** (#minutes,#mn), value= 60.0, Comment: minute time unit: defined an exact duration of 60 seconds
 - **#month** (#months), value= 2592000.0, Comment: month time unit: an approximate duration of 30 days. The number of days of each #month depend of course on the current_date of the model and cannot be constant
 - **#msec** (#millisecond,#milliseconds,#ms), value= 0.001, Comment: millisecond time unit: defines an exact duration of 0.001 second
 - **#now**, value= 1.0, Comment: This value represents the current date
 - **#sec** (#second,#seconds,#s), value= 1.0, Comment: second: the time basic unit, with a fixed value of 1. All other durations are expressed with respect to it
 - **#week** (#weeks), value= 604800.0, Comment: week time unit: defines an exact duration of 7 days
 - **#year** (#years,#y), value= 3.1536E7, Comment: year time unit: an approximate duration of 365 days. The value of #year in number of days varies depending on leap years, etc. and is dependend on the current_date of the model
-

User control operators

Volume units

- `#cl` (#centiliter,#centiliters), value= 1.0E-5, Comment: centiliter unit
 - `#dl` (#deciliter,#deciliters), value= 1.0E-4, Comment: deciliter unit
 - `#hl` (#hectoliter,#hectoliters), value= 0.1, Comment: hectoliter unit
 - `#l` (#liter,#liters,#dm3), value= 0.001, Comment: liter unit
 - `#m3`, value= 1.0, Comment: cube meter: the basic unit for volumes
-

Weight units

- `#gram` (#grams), value= 0.001, Comment: gram unit
 - `#kg` (#kilo,#kilogram,#kilos), value= 1.0, Comment: second: the basic unit for weights
 - `#longton` (#lton), value= 1016.0469088000001, Comment: short ton unit
 - `#ounce` (#oz,#ounces), value= 0.028349523125, Comment: ounce unit
 - `#pound` (#lb,#pounds,#lbf), value= 0.45359237, Comment: pound unit
 - `#shortton` (#ston), value= 907.18474, Comment: short ton unit
 - `#stone` (#st), value= 6.35029318, Comment: stone unit
 - `#ton` (#tons), value= 1000.0, Comment: ton unit
-

Colors

In addition to the previous units, GAML provides a direct access to the 147 named colors defined in CSS (see <http://www.cssportal.com/css3-color-names/>). E.g,

```
rgb my_color <- °teal;
```

Version: 1.9.1

Pseudo-variables

The expressions known as **pseudo-variables** are special read-only variables that are not declared anywhere (at least not in a species), and which represent a value that changes depending on the context of execution.

Table of contents

- [Pseudo-variables](#)
 - [self](#)
 - [myself](#)
 - [each](#)
 - [super](#)

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;
```

super

The pseudo-variable `super` behaves exactly in the same way as `self` except when calling an action, in which case it represents an indirection to the parent species. It is mainly used for allowing to call inherited actions within redefined ones. For instance:

```
species parent {  
  
    int add(int a, int b) {  
        return a + b;  
    }  
  
}  
  
species child parent: parent {  
  
    int add(int a, int b) {  
        // Calls the action defined in 'parent' with modified  
        // arguments  
        return super.add(a + 20, b + 20);  
    }  
  
}
```

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
```

Version: 1.9.1

Variables and Attributes

Variables and attributes represent named data that can be used in an expression. They can be accessed depending on their *scope*:

- the scope of attributes declared in a species is itself, its child species and its micro-species.
- the scope of temporary variables is the one in which they have been declared, and all its sub-scopes. Outside its *scope* of validity, an expression cannot use a variable or an attribute directly. However, attributes can be used in a remote fashion by using a dotted notation on a given agent (see [here](#)).

Table of contents

- [Variables and Attributes](#)
 - [Direct Access](#)
 - [Remote Access](#)

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 neighborhood, 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 neighbors
        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;
}
```

Version: 1.9.1

Operators (A to A)

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, ..., operand)`) or a special infix syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any side effects on their operands. For instance, the `shuffle` operator, which randomizes the positions of elements in a list, does not modify its list operand but returns a new shuffled list.

Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely:

- the constructor operators, like `::`, used to compose pairs of operands, have the

lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values.

- it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else).
 - next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test)
 - next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`)
 - next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators)
 - next the unary operators `-` and `!`
 - next the access operators `.` and `[]` (e.g. `{1, 2, 3}.x > 20 + {4, 5, 6}.y` will return the result of the comparison between the x and y ordinates of the two points)
 - and finally the functional operators, which have the highest priority of all.
-

Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
    int min(int x, int y) {
        return x > y ? x : y;
    }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or
min(my_agent, 10, 20);
    }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considered as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

box, cone3D, cube, cylinder, hexagon, pyramid, set_z, sphere, teapot,

Arithmetic operators

- , /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,

BDI

add_values, and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_likening, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or, set_about,

`set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity,
set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength,
set_trust, set_truth, with_values,`

Casting operators

`as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of,
to_gaml, to_geojson, to_list, with_size, with_style,`

Color-related operators

`-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean,
median, palette, rgb, rnd_color, scale, sum, to_hsb,`

Comparison operators

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

Containers-related operators

`-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect,
contains, contains_all, contains_any, contains_key, count, empty, every, first,
first_with, get, group_by, in, index_by, inter, interleave, internal_integrated_value,
last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul,
none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse,
shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where,
with_max_of, with_min_of,`

Date-related operators

- , != , + , < , <= , = , > , >= , after , before , between , every , milliseconds_between ,
minus_days , minus_hours , minus_minutes , minus_months , minus_ms , minus_weeks ,
minus_years , months_between , plus_days , plus_hours , plus_minutes , plus_months ,
plus_ms , plus_weeks , plus_years , since , to , until , years_between ,

Dates

Displays

horizontal , stack , vertical ,

edge

edge_between , strahler ,

EDP-related operators

diff , diff2 ,

Files-related operators

copy_file , crs , csv_file , delete_file , dxf_file , evaluate_sub_model , file_exists , folder ,

folder_exists, gaml_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphtsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphtsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,

GamaMetaType

type_of,

Gen*

add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,

Graphs-related operators

add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, as_spatial_graph, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_random_graph, generate_watts_strogatz, girvan_newman_clustering, grid_cells_to_graph, in_degree_of, in_edges_of, k_spanning_tree_clustering, label_propagation_clustering,

layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid,
load_shortest_paths, main_connected_component, max_flow_between,
maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of,
out_edges_of, path_between, paths_between, predecessors_of, remove_node_from,
rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of,
undirected, use_cache, weight_of, with_k_shortest_path_algorithm,
with_shortest_path_algorithm, with_weights,

Grid-related operators

as_4_grid, as_grid, as_hexagonal_grid, cell_at, cells_in, cells_overlapping, field,
grid_at, neighbors_of, path_between, points_in, values_in,

ImageOperators

*, antialiased, blend, blurred, brighter, clipped_with, darker, grayscale,
horizontal_flip, image, matrix, rotated_by, sharpened, snapshot, tinted_with,
vertical_flip, with_height, with_size, with_width,

Iterator operators

accumulate, all_match, as_map, collect, count, create_map, first_with, frequency_of,
group_by, index_by, last_with, max_of, mean_of, min_of, none_matches,
one_matches, product_of, sort_by, sum_of, variance_of, where, where, where,
with_max_of, with_min_of,

List-related operators

`all_indexes_of, copy_between, index_of, last_index_of,`

Logical operators

`:!, ?, add_3Dmodel, add_geometry, add_icon, and, or, xor,`

Map comparaison operators

`fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,`

Map-related operators

`as_map, create_map, index_of, last_index_of,`

Matrix-related operators

`-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list,
determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle,
trace, transpose,`

multicriteria operators

`electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM,
weighted_means_DM,`

Path-related operators

`agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths,
max_flow_between, path_between, path_to, paths_between, use_cache,`

Pedestrian

`generate_pedestrian_network,`

Points-related operators

`-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid,
closest_points_with, farthest_point_to, grid_at, norm, points_along, points_at,
points_on,`

Random operators

`binomial, flip, gamma_density, gamma_rnd, gamma_trunc_rnd, gauss,
generate_terrain, lognormal_density, lognormal_rnd, lognormal_trunc_rnd, poisson,
rnd, rnd_choice, sample, shuffle, skew_gauss, truncated_gauss, weibull_density,
weibull_rnd, weibull_trunc_rnd,`

ReverseOperators

`restore_simulation, restore_simulation_from_file, save_simulation, serialize,
serialize_agent,`

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, elliptical_arc, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

Spatial operators

- , *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, angle_between, any_location_in, arc, around, as_4_grid, as_driving_graph, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covering, covers, cross, crosses, crossing, crs, CRS_transform, cube, curve, cylinder, direction_between, disjoint_from, distance_between, distance_to, ellipse, elliptical_arc, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, inverse_rotation, k_nearest_neighbors, line, link, masked_by, moran, neighbors_at, neighbors_of, normalized_rotation, overlapping, overlaps, partially_overlapping, partially_overlaps, path_between, path_to, plan, points_along, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rotated_by, rotation_composition, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, touches, touching, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

Spatial properties operators

`covers`, `crosses`, `intersects`, `partially_overlaps`, `touches`,

Spatial queries operators

`agent_closest_to`, `agent_farthest_to`, `agents_at_distance`, `agents_covering`,
`agents_crossing`, `agents_inside`, `agents_overlapping`, `agents_partially_overlapping`,
`agents_touching`, `at_distance`, `closest_to`, `covering`, `crossing`, `farthest_to`, `inside`,
`neighbors_at`, `neighbors_of`, `overlapping`, `partially_overlapping`, `touching`,

Spatial relations operators

`direction_between`, `distance_between`, `distance_to`, `path_between`, `path_to`, `towards`,

Spatial statistical operators

`hierarchical_clustering`, `k_nearest_neighbors`, `simple_clustering_by_distance`,

Spatial transformations operators

`-`, `*`, `+`, `as_4_grid`, `as_grid`, `as_hexagonal_grid`, `at_location`, `clean`, `clean_network`,
`convex_hull`, `CRS_transform`, `inverse_rotation`, `normalized_rotation`, `rotated_by`,
`rotation_composition`, `scaled_to`, `simplification`, `skeletonize`, `smooth`, `split_geometry`,
`split_lines`, `to_GAMA_CRS`, `to_rectangles`, `to_segments`, `to_squares`,
`to_sub_geometries`, `transformed_by`, `translated_by`, `triangulate`, `voronoi`,
`with_precision`, `without_holes`,

Species-related operators

`index_of, last_index_of, of_generic_species, of_species,`

Statistical operators

`auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum,
build, chi_square, chi_square_complemented, correlation, covariance, dbscan,
distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma,
gamma_distribution, gamma_distribution_complemented, geometric_mean, gini,
harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma,
incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis,
log_gamma, max, mean, mean_deviation, median, min, moment, moran,
morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict,
pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated,
residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis,
split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum,
t_test, variance,`

Strings-related operators

`+, <, <=, >, >=, at, capitalize, char, contains, contains_all, contains_any, copy_between,
date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length,
lower_case, regex_matches, replace, replace_regex, reverse, sample, shuffle,
split_with, string, upper_case,`

SubModel

`load_sub_model,`

System

`., choose, command, copy, copy_from_clipboard, copy_to_clipboard,
copy_to_clipboard, dead, enter, eval_gaml, every, is_error, is_reachable, is_warning,
play_sound, user_confirm, user_input_dialog, wizard, wizard_page,`

Time-related operators

`date, string,`

Types-related operators

`action, agent, attributes, BDIPPlan, bool, container, conversation, directory, emotion,
file, float, gamm_type, gen_population_generator, gen_range, geometry, graph, int,
kml, list, map, matrix, mental_state, message, Norm, pair, path, point, predicate,
regression, rgb, Sanction, skill, social_link, species, topology, unknown,`

User control operators

`choose, enter, user_confirm, user_input_dialog, wizard, wizard_page,`

Operators

-

Possible uses:

- - (int) ---> int
- - (point) ---> point
- - (float) ---> float
- container - container ---> list
- - (container , container) ---> list
- matrix<unknown> - float ---> matrix
- - (matrix<unknown> , float) ---> matrix
- map - pair ---> map
- - (map , pair) ---> map
- int - int ---> int
- - (int , int) ---> int
- int - matrix ---> matrix
- - (int , matrix) ---> matrix
- date - int ---> date
- - (date , int) ---> date
- float - float ---> float
- - (float , float) ---> float
- geometry - float ---> geometry
- - (geometry , float) ---> geometry

- `point - int ---> point`
- `- (point , int)---> point`
- `field - int ---> field`
- `- (field , int)---> field`
- `matrix<unknown> - int ---> matrix`
- `- (matrix<unknown> , int)---> matrix`
- `int - float ---> float`
- `- (int , float)---> float`
- `geometry - geometry ---> geometry`
- `- (geometry , geometry)---> geometry`
- `rgb - rgb ---> rgb`
- `- (rgb , rgb)---> rgb`
- `field - matrix ---> field`
- `- (field , matrix)---> field`
- `geometry - container<unknown,geometry> ---> geometry`
- `- (geometry , container<unknown,geometry>)---> geometry`
- `date - date ---> float`
- `- (date , date)---> float`
- `map - map ---> map`
- `- (map , map)---> map`
- `date - float ---> date`
- `- (date , float)---> date`
- `float - matrix ---> matrix`
- `- (float , matrix)---> matrix`
- `matrix<unknown> - matrix ---> matrix`
- `- (matrix<unknown> , matrix)---> matrix`
- `float - int ---> float`

- $- (\text{float}, \text{int}) \rightarrow \text{float}$
- $\text{point} - \text{point} \rightarrow \text{point}$
- $- (\text{point}, \text{point}) \rightarrow \text{point}$
- $\text{list} - \text{unknown} \rightarrow \text{list}$
- $- (\text{list}, \text{unknown}) \rightarrow \text{list}$
- $\text{species} - \text{agent} \rightarrow \text{list}$
- $- (\text{species}, \text{agent}) \rightarrow \text{list}$
- $\text{point} - \text{float} \rightarrow \text{point}$
- $- (\text{point}, \text{float}) \rightarrow \text{point}$
- $\text{rgb} - \text{int} \rightarrow \text{rgb}$
- $- (\text{rgb}, \text{int}) \rightarrow \text{rgb}$
- $\text{field} - \text{float} \rightarrow \text{field}$
- $- (\text{field}, \text{float}) \rightarrow \text{field}$

Result:

Returns the difference of the two operands. If it is used as an unary operator, it returns the opposite of the operand.

Comment:

The behavior of the operator depends on the type of the operands.

Special cases:

- if both operands are containers and the right operand is empty, - returns the left operand
- if the left operand is a species and the right operand is an agent of the species, - returns a list containing all the agents of the species minus this agent
- if both operands are containers, returns a new list in which all the elements of the right operand have been removed from the left one

```
list<int> var18 <- [1,2,3,4,5,6] - [2,4,9]; // var18 equals [1,3,5,6]
list<int> var19 <- [1,2,3,4,5,6] - [0,8]; // var19 equals
[1,2,3,4,5,6]
```

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

```
int var20 <- 1 - 1; // var20 equals 0
```

- 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 var21 <- 3.5 - matrix([[2,5],[3,4]]); // var21 equals
matrix([[1.5,-1.5],[0.5,-0.5]])
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date minus the given number as duration (in seconds)

```
date var22 <- date('2000-01-01') - 86400; // var22 equals
date('1999-12-31')
```

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

```
geometry var23 <- shape - 5; // var23 equals a geometry corresponding
to the geometry of the agent applying the operator reduced by a
distance of 5
```

- if both operands are a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries

```
geometry var24 <- geom1 - geom2; // var24 equals a geometry corresponding to difference between geom1 and geom2
```

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

```
rgb var25 <- rgb([255, 128, 32]) - rgb('red'); // var25 equals  
rgb([0,128,32])
```

- if the right-operand is a list of points, geometries or agents, returns the geometry resulting from the difference between the left-geometry and all of the right-geometries

```
geometry var26 <- rectangle(10,10) - [circle(2), square(2)]; // var26 equals rectangle(10,10) - (circle(2) + square(2))
```

- if both operands are dates, returns the duration in seconds between date2 and date1. To obtain a more precise duration, in milliseconds, use milliseconds_between(date1, date2)

```
float var27 <- date('2000-01-02') - date('2000-01-01'); // var27 equals 86400
```

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

```
point var28 <- {1, 2} - {4, 5}; // var28 equals {-3.0, -3.0}
```

- if the left operand is a list and the right operand is an object of any type (except list), - returns a list containing the elements of the left operand minus the first occurrence of this object

```
list<int> var29 <- [1,2,3,4,5,6,2] - 2; // var29 equals [1,3,4,5,6,2]  
list<int> var30 <- [1,2,3,4,5,6] - 0; // var30 equals [1,2,3,4,5,6]
```

- if left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the difference of the operand coordinate with this number.

```
point var31 <- {1, 2} - 4.5; // var31 equals {-3.5, -2.5, -4.5}  
point var32 <- {1, 2} - 4; // var32 equals {-3.0, -2.0, -4.0}
```

- 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 var33 <- rgb([255, 128, 32]) - 3; // var33 equals  
rgb([252, 125, 29])
```

Examples:

```
map var0 <- ['a)::1,'b)::2] - ('b)::2); // var0 equals ['a)::1]  
map var1 <- ['a)::1,'b)::2] - ('c)::3); // var1 equals ['a)::1,'b)::2]  
float var2 <- 1.0 - 1.0; // var2 equals 0.0  
float var3 <- 3.7 - 1.2; // var3 equals 2.5  
float var4 <- 3.0 - 1.2; // var4 equals 1.8  
int var5 <- - (-56); // var5 equals 56
```

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

:

Possible uses:

- `unknown : unknown ---> unknown`
- `: (unknown , unknown) ---> unknown`

Result:

It is used in combination with the ? operator. If the left-hand of ? operand evaluates to true, returns the value of the left-hand operand of the :, otherwise that of the right-hand operand of the :

Examples:

```
list<string> var0 <- [10, 19, 43, 12, 7, 22] collect ((each > 20) ?  
'above' : 'below'); // var0 equals ['below', 'below', 'above',  
'below', 'below', 'above']
```

See also: [?](#),

::

Possible uses:

- `any expression :: any expression ---> pair`
- `:: (any expression , any expression) ---> pair`

Result:

produces a new pair combining the left and the right operands

Special cases:

- nil is not acceptable as a key (although it is as a value). If such a case happens, :: will throw an appropriate error
-

!

Possible uses:

- ! (bool) ---> bool

Result:

opposite boolean value.

Special cases:

- if the parameter is not boolean, it is casted to a boolean value.

Examples:

```
bool var0 <- ! (true); // var0 equals false
```

See also: [bool](#), [and](#), [or](#),

!=

Possible uses:

- `int != float --> bool`
- `!= (int, float) --> bool`
- `unknown != unknown --> bool`
- `!= (unknown, unknown) --> bool`
- `float != float --> bool`
- `!= (float, float) --> bool`
- `float != int --> bool`
- `!= (float, int) --> bool`
- `date != date --> bool`
- `!= (date, date) --> 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 <- [2,3] != [2,3]; // var2 equals false
bool var3 <- [2,4] != [2,3]; // var3 equals true
bool var4 <- 3.0 != 3.0; // var4 equals false
bool var5 <- 4.0 != 4.7; // var5 equals true
bool var6 <- 3.0 != 3; // var6 equals false
bool var7 <- 4.7 != 4; // var7 equals true
bool var8 <- #now != #now minus_hours 1; // var8 equals true
```

See also: [=](#), [>](#), [<](#), [>=](#), [<=](#),

?

Possible uses:

- `bool ? any expression ---> unknown`
- `? (bool , any expression)---> unknown`

Result:

It is used in combination with the `:` operator: if the left-hand operand evaluates to true, returns the value of the left-hand operand of the `:`; otherwise that of the right-hand operand of the `:`

Comment:

These functional tests can be combined together.

Examples:

```
list<string> var0 <- [10, 19, 43, 12, 7, 22] collect ((each > 20) ?  
'above' : 'below'); // var0 equals ['below', 'below', 'above',  
'below', 'below', 'above']  
rgb col <- (flip(0.3) ? #red : (flip(0.9) ? #blue : #green));
```

See also: [:](#),

/

Possible uses:

- `int / int ---> float`
- `/ (int , int) ---> float`
- `int / float ---> float`
- `/ (int , float) ---> float`
- `float / float ---> float`
- `/ (float , float) ---> float`
- `rgb / int ---> rgb`
- `/ (rgb , int) ---> rgb`
- `rgb / float ---> rgb`
- `/ (rgb , float) ---> rgb`
- `field / int ---> field`
- `/ (field , int) ---> field`
- `point / float ---> point`
- `/ (point , float) ---> point`
- `field / float ---> field`
- `/ (field , float) ---> field`
- `point / int ---> point`
- `/ (point , int) ---> point`
- `matrix<unknown> / float ---> matrix`
- `/ (matrix<unknown> , float) ---> matrix`
- `float / int ---> float`
- `/ (float , int) ---> float`
- `matrix<unknown> / int ---> matrix`

- `/ (matrix<unknown>, int) ---> matrix`
- `matrix<unknown> / matrix ---> matrix`
- `/ (matrix<unknown>, matrix) ---> matrix`

Result:

Returns the division of the two operands.

Special cases:

- if the right-hand operand is equal to zero, raises a "Division by zero" exception
- if both operands are numbers (float or int), performs a normal arithmetic division and returns a float.

```
float var0 <- 3 / 5.0; // var0 equals 0.6
```

- if one operand is a color and the other an integer, returns a new color resulting from the division of each component of the color by the right operand

```
rgb var1 <- rgb([255, 128, 32]) / 2; // var1 equals rgb([127, 64, 16])
```

- if 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: [+](#), [-](#), [*](#),



Possible uses:

- `agent . any expression` ---> `unknown`
- `. (agent , any expression)` ---> `unknown`

Result:

It has two different uses: it can be the dot product between 2 matrices or return an evaluation of the expression (right-hand operand) in the scope the given agent.

Special cases:

- if the agent is nil or dead, throws an exception
- if the left operand is an agent, it evaluates of the expression (right-hand operand) in the scope the given agent

```
unknown var0 <- agent1.location; // var0 equals the location of the
agent agent1
```

.

Possible uses:

- `matrix . matrix` ---> `matrix`
- `. (matrix , matrix)`---> `matrix`

Special cases:

- if both operands are matrix, returns the dot product of them

```
matrix var0 <- matrix([[1,1],[1,2]]) . matrix([[1,1],[1,2]]); // var0  
equals matrix([[2,3],[3,5]])
```

Λ

Possible uses:

- `float ^ int` ---> `float`
- `^ (float , int)`---> `float`
- `int ^ int` ---> `float`
- `^ (int , int)`---> `float`
- `float ^ float` ---> `float`
- `^ (float , float)`---> `float`
- `int ^ float` ---> `float`
- `^ (int , 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.
- Various examples of power

```
float var0 <- 2 ^ 3; // var0 equals 8.0
```

Examples:

```
float var1 <- 4.84 ^ 0.5; // var1 equals 2.2
```

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



Same signification as [at](#)



Possible uses:

- `int * float --> float`

- `* (int , float) ---> float`
- `field * float ---> field`
- `* (field , float) ---> field`
- `point * float ---> point`
- `* (point , float) ---> point`
- `int * int ---> int`
- `* (int , int) ---> int`
- `float * float ---> float`
- `* (float , float) ---> float`
- `geometry * float ---> geometry`
- `* (geometry , float) ---> geometry`
- `matrix<unknown> * matrix ---> matrix`
- `* (matrix<unknown> , matrix) ---> matrix`
- `matrix<unknown> * float ---> matrix`
- `* (matrix<unknown> , float) ---> matrix`
- `rgb * int ---> rgb`
- `* (rgb , int) ---> rgb`
- `matrix<unknown> * int ---> matrix`
- `* (matrix<unknown> , int) ---> matrix`
- `float * matrix ---> matrix`
- `* (float , matrix) ---> matrix`
- `point * int ---> point`
- `* (point , int) ---> point`
- `rgb * float ---> rgb`
- `* (rgb , float) ---> rgb`
- `point * point ---> float`
- `* (point , point) ---> float`

- `field * int` ---> `field`
- `* (field, int)` ---> `field`
- `int * matrix` ---> `matrix`
- `* (int, matrix)` ---> `matrix`
- `float * int` ---> `float`
- `* (float, int)` ---> `float`
- `geometry * point` ---> `geometry`
- `* (geometry, point)` ---> `geometry`

Result:

Returns the product of the two operands.

Special cases:

- if both operands are numbers (float or int), performs a normal arithmetic product and returns a float if one of them is a float.

```
int var1 <- 1 * 1; // var1 equals 1
```

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

```
geometry var2 <- circle(10) * 2; // var2 equals circle(20)
geometry var3 <- (circle(10) * 2).location with_precision 9; // var3
equals (circle(20)).location with_precision 9
float var4 <- (circle(10) * 2).height with_precision 9; // var4 equals
(circle(20)).height with_precision 9
```

- if one operand is a color and the other an integer, returns a new color resulting

from the product of each component of the color with the right operand (with a maximum value at 255)

```
rgb var5 <- rgb([255, 128, 32]) * 2; // var5 equals rgb([255, 255, 64])
```

- if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number

```
point var6 <- {2,5} * 4; // var6 equals {8.0, 20.0}  
point var7 <- {2, 4} * 2.5; // var7 equals {5.0, 10.0}
```

- if one operand is a color and the other a float, returns a new color resulting from the product of each component of the color with the right operand (with a maximum value at 255)

```
rgb var8 <- rgb([255, 128, 32]) * 2.0; // var8 equals  
rgb([255, 255, 64])
```

- if both operands are points, returns their scalar product

```
float var9 <- {2,5} * {4.5, 5}; // var9 equals 34.0
```

- 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 var10 <- 2 * matrix([[2,5],[3,4]]); // var10 equals  
matrix([[4,10],[6,8]])
```

- 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 var11 <- shape * {0.5,0.5,2}; // var11 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
```

Examples:

```
float var0 <- 2.5 * 2; // var0 equals 5.0
```

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

*

Possible uses:

- `image * float --> image`
- `* (image , float) --> image`

Result:

Applies a proportional scaling ratio to the image passed in parameter and returns a new scaled image. A ratio of 0 will return nil, a ratio of 1 will return the original image. Automatic scaling and resizing methods are used. The original image is left untouched

+

Possible uses:

- `string + string ---> string`
- `+ (string , string)---> string`
- `point + float ---> point`
- `+ (point , float)---> point`
- `field + int ---> field`
- `+ (field , int)---> field`
- `date + float ---> date`
- `+ (date , float)---> date`
- `matrix<unknown> + int ---> matrix`
- `+ (matrix<unknown> , int)---> matrix`
- `int + matrix ---> matrix`
- `+ (int , matrix)---> matrix`
- `container + container ---> container`
- `+ (container , container)---> container`
- `float + float ---> float`
- `+ (float , float)---> float`
- `int + float ---> float`
- `+ (int , float)---> float`
- `float + matrix ---> matrix`
- `+ (float , matrix)---> matrix`
- `point + int ---> point`
- `+ (point , int)---> point`
- `matrix<unknown> + float ---> matrix`

- $+ (\text{matrix<unknown>} , \text{float}) \rightarrow \text{matrix}$
- $\text{field} + \text{float} \rightarrow \text{field}$
- $+ (\text{field} , \text{float}) \rightarrow \text{field}$
- $\text{date} + \text{string} \rightarrow \text{string}$
- $+ (\text{date} , \text{string}) \rightarrow \text{string}$
- $\text{map} + \text{map} \rightarrow \text{map}$
- $+ (\text{map} , \text{map}) \rightarrow \text{map}$
- $\text{rgb} + \text{rgb} \rightarrow \text{rgb}$
- $+ (\text{rgb} , \text{rgb}) \rightarrow \text{rgb}$
- $\text{field} + \text{matrix} \rightarrow \text{field}$
- $+ (\text{field} , \text{matrix}) \rightarrow \text{field}$
- $\text{point} + \text{point} \rightarrow \text{point}$
- $+ (\text{point} , \text{point}) \rightarrow \text{point}$
- $\text{geometry} + \text{geometry} \rightarrow \text{geometry}$
- $+ (\text{geometry} , \text{geometry}) \rightarrow \text{geometry}$
- $\text{int} + \text{int} \rightarrow \text{int}$
- $+ (\text{int} , \text{int}) \rightarrow \text{int}$
- $\text{geometry} + \text{float} \rightarrow \text{geometry}$
- $+ (\text{geometry} , \text{float}) \rightarrow \text{geometry}$
- $\text{container} + \text{unknown} \rightarrow \text{list}$
- $+ (\text{container} , \text{unknown}) \rightarrow \text{list}$
- $\text{date} + \text{int} \rightarrow \text{date}$
- $+ (\text{date} , \text{int}) \rightarrow \text{date}$
- $\text{rgb} + \text{int} \rightarrow \text{rgb}$
- $+ (\text{rgb} , \text{int}) \rightarrow \text{rgb}$
- $\text{map} + \text{pair} \rightarrow \text{map}$
- $+ (\text{map} , \text{pair}) \rightarrow \text{map}$

- `float + int --> float`
- `+ (float, int) --> float`
- `matrix<unknown> + matrix --> matrix`
- `+ (matrix<unknown>, matrix) --> matrix`
- `string + unknown --> string`
- `+ (string, unknown) --> string`
- `+ (geometry, float, int) --> geometry`
- `+ (geometry, float, bool) --> geometry`
- `+ (geometry, float, int, int) --> geometry`
- `+ (geometry, float, int, int, bool) --> geometry`

Result:

Returns the sum, union or concatenation of the two operands.

Special cases:

- if one of the operands is nil, + throws an error
- if both operands are species, returns a special type of list called meta-population
- if the left-hand and right-hand operand are a string, returns the concatenation of the two operands

```
string var9 <- "hello " + "World"; // var9 equals "hello World"
```

- if the left-hand operand is a geometry and the right-hand operands a float and an integer, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand

```
geometry var10 <- circle(5) + (5,32); // var10 equals circle(10)
```

- if the left-hand operand is a point and the right-hand a number, returns a new point with each coordinate as the sum of the operand coordinate with this number.

```
point var11 <- {1, 2} + 4.5; // var11 equals {5.5, 6.5, 4.5}
```

- if the left-hand operand is a geometry and the right-hand operands a float, an integer, one of #round, #square or #flat and a boolean, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand and a flat, square or round end cap style and single sided is the boolean is true

```
geometry var12 <- line([{10,10}, {50,50}]) + (5,32,#round, true); //  
var12 equals A ploygon corresponding to the buffer generated
```

- if the left-hand operand is a geometry and the right-hand operands a float and a boolean, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), single sided is the boolean is true

```
geometry var13 <- line([{10,10}, {50,50}]) + (5, true); // var13  
equals A ploygon corresponding to the buffer generated
```

- 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 var14 <- 3.5 + matrix([[2,5],[3,4]]); // var14 equals  
matrix([[5.5,8.5],[6.5,7.5]])
```

- if both operands are list, + returns the concatenation of both lists.

```
list<int> var15 <- [1,2,3,4,5,6] + [2,4,9]; // var15 equals  
[1,2,3,4,5,6,2,4,9]  
list<int> var16 <- [1,2,3,4,5,6] + [0,8]; // var16 equals  
[1,2,3,4,5,6,0,8]
```

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

```
rgb var17 <- rgb([255, 128, 32]) + rgb('red'); // var17 equals  
rgb([255,128,32])
```

- if both operands are points, returns their sum.

```
point var18 <- {1, 2} + {4, 5}; // var18 equals {5.0, 7.0}
```

- if the left-hand operand is a geometry and the right-hand operand a float, an integer and one of #round, #square or #flat, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand and a flat, square or round end cap style

```
geometry var19 <- circle(5) + (5,32,#round); // var19 equals  
circle(10)
```

- if the right-operand is a point, a geometry or an agent, returns the geometry

resulting from the union between both geometries

```
geometry var20 <- geom1 + geom2; // var20 equals a geometry  
corresponding to union between geom1 and geom2
```

- 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 var21 <- 1 + 1; // var21 equals 2
```

- if the left-hand operand is a geometry and the right-hand operand a float, returns a geometry corresponding to the left-hand operand (geometry, agent, point) enlarged by the right-hand operand distance. The number of segments used by default is 8 and the end cap style is #round

```
geometry var22 <- circle(5) + 5; // var22 equals circle(10)
```

- if the right operand is an object of any type (except a container), + returns a list of the elements of the left operand, to which this object has been added

```
list<int> var23 <- [1,2,3,4,5,6] + 2; // var23 equals [1,2,3,4,5,6,2]  
list<int> var24 <- [1,2,3,4,5,6] + 0; // var24 equals [1,2,3,4,5,6,0]
```

- if one of the operands is a date and the other a number, returns a date corresponding to the date plus the given number as duration (in seconds)

```
date var25 <- date('2000-01-01') + 86400; // var25 equals  
date('2000-01-02')
```

- 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 var26 <- rgb([255, 128, 32]) + 3; // var26 equals  
rgb([255,131,35])
```

- if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one being casted into a string)

```
string var27 <- "hello " + 12; // var27 equals "hello 12"
```

Examples:

```
date var0 <- date('2016-01-01 00:00:01') + 86400; // var0 equals  
date('2016-01-02 00:00:01')  
point var1 <- {1, 2} + 4; // var1 equals {5.0, 6.0, 4.0}  
string var2 <- date('2000-01-01 00:00:00') + '_Test'; // var2 equals  
'2000-01-01 00:00:00_Test'  
map var3 <- ['a':1,'b':2] + ['c':3]; // var3 equals  
['a':1,'b':2,'c':3]  
map var4 <- ['a':1,'b':2] + [5::3.0]; // var4 equals  
['a':1,'b':2,5::3.0]  
map var5 <- ['a':1,'b':2] + ('c':3); // var5 equals  
['a':1,'b':2,'c':3]  
map var6 <- ['a':1,'b':2] + ('c':3); // var6 equals  
['a':1,'b':2,'c':3]  
float var7 <- 1.0 + 1; // var7 equals 2.0  
float var8 <- 1.0 + 2.5; // var8 equals 3.5
```

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

<

Possible uses:

- `int < float ---> bool`
- `< (int , float)---> bool`
- `point < point ---> bool`
- `< (point , point)---> bool`
- `float < float ---> bool`
- `< (float , float)---> bool`
- `float < int ---> bool`
- `< (float , int)---> bool`
- `string < string ---> bool`
- `< (string , string)---> bool`
- `int < int ---> bool`
- `< (int , int)---> bool`
- `date < date ---> bool`
- `< (date , date)---> 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 and only if the 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 var0 <- {5,7} < {4,6}; // var0 equals false  
bool var1 <- {5,7} < {4,8}; // var1 equals false
```

- if both operands are String, uses a lexicographic comparison of two strings

```
bool var2 <- 'abc' < 'aeb'; // var2 equals true
```

Examples:

```
bool var3 <- 3 < 2.5; // var3 equals false  
bool var4 <- 3.5 < 7.6; // var4 equals true  
bool var5 <- 3.5 < 7; // var5 equals true  
bool var6 <- 3 < 7; // var6 equals true  
bool var7 <- #now < #now minus_hours 1; // var7 equals false
```

See also: [>](#), [>=](#), [<=](#), [=](#), [!=](#),

<=

Possible uses:

- `int <= float` ---> `bool`
- `<= (int , float)`---> `bool`
- `string <= string` ---> `bool`
- `<= (string , string)`---> `bool`
- `float <= float` ---> `bool`
- `<= (float , float)`---> `bool`
- `float <= int` ---> `bool`
- `<= (float , int)`---> `bool`

- `date <= date` ---> `bool`
- `<= (date , date)`---> `bool`
- `point <= point` ---> `bool`
- `<= (point , point)`---> `bool`
- `int <= int` ---> `bool`
- `<= (int , int)`---> `bool`

Result:

true if the left-hand operand is less or equal than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' <= 'aeb'; // var0 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand 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 var1 <- {5,7} <= {4,6}; // var1 equals false
bool var2 <- {5,7} <= {4,8}; // var2 equals false
```

Examples:

```
bool var3 <- 3 <= 2.5; // var3 equals false
bool var4 <- 3.5 <= 3.5; // var4 equals true
```

See also: `>`, `<`, `>=`, `=`, `!=`,

=

Possible uses:

- `int = int ---> bool`
- `= (int , int) ---> bool`
- `date = date ---> bool`
- `= (date , date) ---> bool`
- `float = int ---> bool`
- `= (float , int) ---> bool`
- `int = float ---> bool`
- `= (int , float) ---> bool`
- `float = float ---> bool`
- `= (float , float) ---> bool`
- `unknown = unknown ---> bool`
- `= (unknown , unknown) ---> bool`

Result:

returns true if both operands are equal, false otherwise returns true if both operands are equal, false otherwise

Special cases:

- if both operands are any kind of objects, returns true if they are identical (i.e., the same object) or equal (comparisons between nil values are permitted)

```
bool var6 <- [2,3] = [2,3]; // var6 equals true
```

Examples:

```
bool var0 <- 4 = 5; // var0 equals false
bool var1 <- #now = #now minus_hours 1; // var1 equals false
bool var2 <- 4.7 = 4; // var2 equals false
bool var3 <- 3 = 3.0; // var3 equals true
bool var4 <- 4 = 4.7; // var4 equals false
bool var5 <- 4.5 = 4.7; // var5 equals false
```

See also: !=, >, <, >=, <=,



Possible uses:

- int > float ---> bool
- > (int , float)---> bool
- float > int ---> bool
- > (float , int)---> bool
- int > int ---> bool
- > (int , int)---> bool
- date > date ---> bool
- > (date , date)---> bool
- string > string ---> bool
- > (string , string)---> bool
- point > point ---> bool
- > (point , point)---> bool

- `float > float ---> bool`
- `> (float , float)---> 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 and only if the 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; // var4 equals false
bool var5 <- 13.0 > 7.0; // var5 equals true
bool var6 <- (#now > (#now minus_hours 1)); // var6 equals true
bool var7 <- 3.5 > 7.6; // var7 equals false
```

See also: `<`, `>=`, `<=`, `=`, `!=`,

`>=`

Possible uses:

- `string >= string ---> bool`
- `>= (string , string) ---> bool`
- `float >= float ---> bool`
- `>= (float , float) ---> bool`
- `date >= date ---> bool`
- `>= (date , date) ---> bool`
- `int >= float ---> bool`
- `>= (int , float) ---> bool`
- `float >= int ---> bool`
- `>= (float , int) ---> bool`
- `int >= int ---> bool`
- `>= (int , int) ---> bool`
- `point >= point ---> bool`
- `>= (point , 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 the two strings

```
bool var5 <- 'abc' >= 'aeb'; // var5 equals false  
bool var6 <- 'abc' >= 'abc'; // var6 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand is greater or equal than x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var7 <- {5,7} >= {4,6}; // var7 equals true  
bool var8 <- {5,7} >= {4,8}; // var8 equals false
```

Examples:

```
bool var0 <- 3.5 >= 3.5; // var0 equals true  
bool var1 <- #now >= #now minus_hours 1; // var1 equals true  
bool var2 <- 3 >= 2.5; // var2 equals true  
bool var3 <- 3.5 >= 7; // var3 equals false  
bool var4 <- 3 >= 7; // var4 equals false
```

See also: `>`, `<`, `<=`, `=`, `!=`,

abs

Possible uses:

- `abs` (`float`) ---> `float`
- `abs` (`int`) ---> `int`

Result:

Returns the absolute value of the operand (so a positive int or float depending on the type of the operand).

Examples:

```
float var0 <- abs (200 * -1 + 0.5); // var0 equals 199.5
int var1 <- abs (-10); // var1 equals 10
int var2 <- abs (10); // var2 equals 10
```

accumulate

Possible uses:

- container **accumulate** any expression ---> list
- **accumulate** (container , 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 left-hand operand elements.

Examples:

```
list var0 <- [a1,a2,a3] accumulate (each neighbors_at 10); // var0
equals a flat list of all the neighbors of these three agents
list<int> var1 <- [1,2,4] accumulate ([2,4]); // var1 equals
[2,4,2,4,2,4]
```

See also: [collect](#),

acos

Possible uses:

- `acos` (`int`) ---> `float`
- `acos` (`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](#), [cos](#),

action

Possible uses:

- `action` (`any`) ---> `action`

Result:

casts the operand in a action object.

add_3Dmodel

Possible uses:

- `add_3Dmodel (kml, point, float, float, string) ---> kml`
- `add_3Dmodel (kml, point, float, float, string, date, date) ---> kml`

Result:

the kml export manager with new 3D model: specify the 3D model (collada) to add to the kml

See also: [add_geometry](#), [add_icon](#), [add_label](#),

add_attribute

Possible uses:

- `add_attribute (gen_population_generator, string, any GAML type, list) ---> gen_population_generator`
- `add_attribute (gen_population_generator, string, any GAML type, list, bool) ---> gen_population_generator`
- `add_attribute (gen_population_generator, string, any GAML type, list, string, any GAML type) ---> gen_population_generator`
- `add_attribute (gen_population_generator, string, any GAML type, list,`

```
bool, string, any GAML type) --> gen_population_generator
```

Result:

add an attribute defined by its name (string), its datatype (type), its list of values (list) and attributeType name (type of the attribute among "range" and "unique") to a population_generator
add an attribute defined by its name (string), its datatype (type), its list of values (list) to a population_generator
add an attribute defined by its name (string), its datatype (type), its list of values (list) and record name (name of the attribute to record) to a population_generator
add an attribute defined by its name (string), its datatype (type), its list of values (list) to a population_generator

Examples:

```
add_attribute(pop_gen, "iris", string, liste_iris, "unique")
add_attribute(pop_gen, "Sex", string, ["Man", "Woman"])
add_attribute(pop_gen, "iris", string, liste_iris, "unique",
"P13_POP")
add_attribute(pop_gen, "Sex", string, ["Man", "Woman"])
```

add_census_file

Possible uses:

- `add_census_file` (gen_population_generator, string, string, string, int, int) --> gen_population_generator

Result:

add a census data file defined by its path (string), its type ("ContingencyTable", "GlobalFrequencyTable", "LocalFrequencyTable" or "Sample"), its separator (string), the index of the first row of data (int) and the index of the first column of data (int) to

a population_generator

Examples:

```
add_census_file(pop_gen, ".../data/Age_Couple.csv",
"ContingencyTable", ";", 1, 1)
```

add_days

Same signification as plus_days

add_edge

Possible uses:

- graph add_edge pair ---> graph
- add_edge (graph , pair)---> graph

Result:

add an edge between a source vertex and a target vertex (resp. the left and the right element of the pair operand)

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph. If the edge already exists, the graph is unchanged

Examples:

```
graph <- graph add_edge (source::target);
```

See also: [add_node](#), [graph](#),

add_geometry

Possible uses:

- `add_geometry (kml, geometry, float, rgb) ---> kml`
- `add_geometry (kml, geometry, rgb, rgb) ---> kml`
- `add_geometry (kml, geometry, float, rgb, rgb) ---> kml`
- `add_geometry (kml, geometry, float, rgb, rgb, date) ---> kml`
- `add_geometry (kml, geometry, float, rgb, rgb, date, date) ---> kml`

Result:

Define the kml export manager with new geometry

See also: [add_3Dmodel](#), [add_icon](#), [add_label](#),

add_hours

Same signification as [plus_hours](#)

add_icon

Possible uses:

- `add_icon (kml, point, float, float, string) ---> kml`
- `add_icon (kml, point, float, float, string, date, date) ---> kml`

Result:

Define the kml export manager with new icons

See also: [add_geometry](#), [add_icon](#),

add_mapper

Possible uses:

- `add_mapper (gen_population_generator, string, any GAML type, map) ---> gen_population_generator`
- `add_mapper (gen_population_generator, string, any GAML type, map, bool) ---> gen_population_generator`

Result:

add a mapper between source of data for a attribute to a population_generator. A mapper is defined by the name of the attribute, the datatype of attribute (type), the corresponding value (map<list,list>) and the type of attribute ("unique" or "range")
add a mapper between source of data for a attribute to a population_generator. A mapper is defined by the name of the attribute, the datatype of attribute (type), the corresponding value (map<list,list>) and the type of attribute ("unique" or "range")

Examples:

```
add_mapper(pop_gen, "Age", int, [{"0 to 18"}:: {"1 to 10", "11 to 18"}, {"18 to 100"}:: {"18 to 50", "51 to 100"}, "range"]);
add_mapper(pop_gen, "Age", int, [{"0 to 18"}:: {"1 to 10", "11 to 18"}, {"18 to 100"}:: {"18 to 50", "51 to 100"}, "range"]);
```

add_marginals

Possible uses:

- gen_population_generator **add_marginals** list --->
gen_population_generator
- **add_marginals** (gen_population_generator , list) --->
gen_population_generator

Result:

add a list of marginals (name of the attributes) to fit the population with, in any CO based algorithm

Examples:

```
add_marginals(pop_gen, ["gender", "age"]);
```

add_minutes

Same signification as [plus_minutes](#)

add_months

Same signification as [plus_months](#)

add_ms

Same signification as [plus_ms](#)

add_node

Possible uses:

- `graph add_node geometry --> graph`
- `add_node (graph, geometry) --> graph`

Result:

adds a node in a graph.

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph

Examples:

```
graph var0 <- graph add_node node(0); // var0 equals the graph, to  
which node(0) has been added
```

See also: [add_edge](#), [graph](#),

add_point

Possible uses:

- `geometry add_point point` ---> `geometry`
- `add_point (geometry , point)`---> `geometry`

Result:

A new geometry resulting from the addition of the right point (coordinate) to the left-hand geometry. Note that adding a point to a line or polyline will always return a closed contour. Also note that the position at which the added point will appear in the geometry is not necessarily the last one, as points are always ordered in a clockwise fashion in geometries

Examples:

```
geometry var0 <- polygon([{10,10},{10,20},{20,20}]) add_point  
{20,10}; // var0 equals polygon([{10,10},{10,20},{20,20},{20,10}])
```

add_range_attribute

Possible uses:

- `add_range_attribute (gen_population_generator, string, list, int, int)`
---> `gen_population_generator`

Result:

add a rangee attribute defined by its name (string), the list of ranges (list) to a population_generator

Examples:

```
add_attribute(pop_gen, "Sex", string, ["Man", "Woman"])
```

add_seconds

Same signification as +

add_values

Possible uses:

- `predicate add_values map` ---> `predicate`
- `add_values (predicate , map)` ---> `predicate`

Result:

add a new value to the map of the given predicate

Examples:

```
predicate add_values ["time":10];
```

add_weeks

Same signification as [plus_weeks](#)

add_years

Same signification as [plus_years](#)

adjacency

Possible uses:

- `adjacency` (`graph`) ---> `matrix<float>`

Result:

adjacency matrix of the given graph.

after

Possible uses:

- `after` (`date`) ---> `bool`
- `any expression after date` ---> `bool`
- `after (any expression , date)` ---> `bool`

Result:

Returns true if the `current_date` of the model is strictly after the date passed in

argument. Synonym of 'current_date > argument'. Can be used in its composed form with 2 arguments to express the lower boundary for the computation of a frequency. Note that only dates strictly after this one will be tested against the frequency

Examples:

```
reflex when: after(starting_date) {}      // this reflex will always be
run after the first step
reflex when: false after(starting date + #10days) {}    // This reflex
will not be run after this date. Better to use 'until' or 'before' in
that case
every(2#days) after (starting_date + 1#day)      // the computation
will return true every two days (using the starting_date of the model
as the starting point) only for the dates strictly after this
starting_date + 1#day
```

agent

Possible uses:

- `agent` (`any`) ---> `agent`

Result:

casts the operand in a agent object.

agent_closest_to

Possible uses:

- `agent_closest_to` (`unknown`) ---> `agent`

Result:

An agent, the closest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
agent var0 <- agent_closest_to(self); // var0 equals the closest agent  
to the agent applying the operator.
```

See also: [neighbors_at](#), [neighbors_of](#), [agents_inside](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

agent_farthest_to

Possible uses:

- `agent_farthest_to` (`unknown`) ---> `agent`

Result:

An agent, the farthest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
agent var0 <- agent_farthest_to(self); // var0 equals the farthest  
agent to the agent applying the operator.
```

See also: [neighbors_at](#), [neighbors_of](#), [agents_inside](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#), [agent_closest_to](#), [farthest_to](#),

agent_from_geometry

Possible uses:

- `path agent_from_geometry geometry --> agent`
- `agent_from_geometry (path, geometry) --> agent`

Result:

returns the agent corresponding to given geometry (right-hand operand) in the given path (left-hand operand).

Special cases:

- if the left-hand operand is nil, returns nil

Examples:

```
geometry line <- one_of(path_followed.segments);  
road ag <- road(path_followed agent_from_geometry line);
```

See also: [path](#),

agent_intersecting

Same signification as [agents_overlapping](#)

agents_at_distance

Possible uses:

- `agents_at_distance` (`float`) ---> `list`

Result:

A list of agents situated at a distance lower than the right argument.

Examples:

```
list var0 <- agents_at_distance(20); // var0 equals all the agents  
(excluding the caller) which distance to the caller is lower than 20
```

See also: [neighbors_at](#), [neighbors_of](#), [agent_closest_to](#), [agents_inside](#), [closest_to](#), [inside](#), [overlapping](#), [at_distance](#),

agents_covering

Possible uses:

- `agents_covering` (`unknown`) ---> `list<agent>`

Result:

A list of agents covered by the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_covering(self); // var0 equals the agents  
that cover the shape of the agent applying the operator.
```

See also: [agent_closest_to](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

agents_crossing

Possible uses:

- **agents_crossing** (unknown) ---> `list<agent>`

Result:

A list of agents cross the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_crossing(self); // var0 equals the agents  
that crossing the shape of the agent applying the operator.
```

See also: [agent_closest_to](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

agents_inside

Possible uses:

- `agents_inside` (`unknown`) ---> `list<agent>`

Result:

A list of agents covered by the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_inside(self); // var0 equals the agents  
that are covered by the shape of the agent applying the operator.
```

See also: [agent_closest_to](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

agents_overlapping

Possible uses:

- `agents_overlapping` (`unknown`) ---> `list<agent>`

Result:

A list of agents overlapping the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_overlapping(self); // var0 equals the  
agents that overlap the shape of the agent applying the operator.
```

See also: [neighbors_at](#), [neighbors_of](#), [agent_closest_to](#), [agents_inside](#), [closest_to](#), [inside](#), [overlapping](#), [at_distance](#),

agents_partially_overlapping

Possible uses:

- **agents_partially_overlapping** (`unknown`) --> `list<agent>`

Result:

A list of agents that partially overlap the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_partially_overlapping(self); // var0 equals  
the agents that partially overlap the shape of the agent applying the  
operator.
```

See also: [agent_closest_to](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

agents_touching

Possible uses:

- **agents_touching** (`unknown`) --> `list<agent>`

Result:

A list of agents touching the operand (casted as a geometry).

Examples:

```
list<agent> var0 <- agents_touching(self); // var0 equals the agents  
that touch the shape of the agent applying the operator.
```

See also: [agent_closest_to](#), [agents_overlapping](#), [closest_to](#), [inside](#), [overlapping](#),

all_indexes_of

Possible uses:

- `list all_indexes_of unknown ---> list`
- `all_indexes_of (list , unknown) ---> list`

Result:

all the index of all the occurrences of the right operand in the left operand container

Comment:

The definition of `all_indexes_of` and the type of the index depend on the container

Special cases:

- if the left operand is a list, `all_indexes_of` returns a list of all the indexes as integers

```
list var0 <- [1,2,3,1,2,3] all_indexes_of 1; // var0 equals [0,3]  
list var1 <- [1,2,3,1,2,3] all_indexes_of 4; // var1 equals []
```

See also: [index_of](#), [last_index_of](#),

all_match

Possible uses:

- `container all_match any expression ---> bool`
- `all_match (container, any expression)---> bool`

Result:

Returns true if all the elements of the left-hand operand make the right-hand operand evaluate to true. Returns true if the left-hand operand is empty. 'c all_match each.property' is strictly equivalent to '(c count each.property) = length(c)' but faster in most cases (as it is a shortcircuited operator)

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, all_match throws an error

Examples:

```
bool var0 <- [1,2,3,4,5,6,7,8] all_match (each > 3); // var0 equals  
false  
bool var1 <- [1::2, 3::4, 5::6] all_match (each > 4); // var1 equals  
false
```

See also: [none_matches](#), [one_matches](#), [count](#),

all_pairs_shortest_path

Possible uses:

- `all_pairs_shortest_path` (`graph`) ---> `matrix<int>`

Result:

returns the successor matrix of shortest paths between all node pairs (rows: source, columns: target): a cell (i,j) will thus contains the next node in the shortest path between i and j.

Examples:

```
matrix<int> var0 <- all_pairs_shortest_paths(my_graph); // var0 equals  
shortest_paths_matrix will contain all pairs of shortest paths
```

all_verify

Same signification as `all_match`

alpha_index

Possible uses:

- `alpha_index` (`graph`) ---> `float`

Result:

returns the alpha index of the graph (measure of connectivity which evaluates the number of cycles in a graph in comparison with the maximum number of cycles. The higher the alpha index, the more a network is connected: $\text{alpha} = \text{nb_cycles} / (2 * S - 5)$ - 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](#),

among

Possible uses:

- `int among container --> list`
- `among (int , 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 (converted as a list). If it is smaller or equal to zero, it returns an empty list

Examples:

```
list<int> var0 <- 3 among [1,2,4,3,5,7,6,8]; // var0 equals [1,2,8]
(for example)
list var1 <- 3 among g2; // var1 equals [node6,node11,node7]
list var2 <- 3 among list(node); // var2 equals [node1,node11,node4]
list<int> var3 <- 1 among [1::2,3::4]; // var3 equals 2 or 4
```

and

Possible uses:

- `bool and any expression --> bool`
- `and (bool , any expression) --> bool`

Result:

a bool value, equal to the logical and between the left-hand operand and the right-hand operand.

Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like (1 and 0) is accepted and returns false.

Examples:

```
bool var0 <- true and false; // var0 equals false
bool var1 <- false and false; // var1 equals false
bool var2 <- false and true; // var2 equals false
bool var3 <- true and true; // var3 equals true
int a <- 3 ; int b <- 4; int c <- 7;
bool var5 <- ((a+b) = c ) and ((a+b) > c ); // var5 equals false
```

See also: [bool](#), [or](#), [!](#),

and

Possible uses:

- `predicate and predicate` --> `predicate`
- `and (predicate , predicate)`--> `predicate`

Result:

create a new predicate from two others by including them as subintentions

Examples:

```
predicate1 and predicate2
```

angle_between

Possible uses:

- `angle_between` (`point`, `point`, `point`) ---> `float`

Result:

the angle between vectors P0P1 and P0P2 (P0, P1, P2 being the three point operands)

Examples:

```
float var0 <- angle_between({5,5},{10,5},{5,10}); // var0 equals 90
```

antialiased

Possible uses:

- `antialiased` (`image`) ---> `image`

Result:

Application of a very light blur kernel that acts like an anti-aliasing filter when applied to an image. This operation can be applied multiple times in a row if greater.

any

Same signification as `one_of`

any_location_in

Possible uses:

- `any_location_in` (`geometry`) --> `point`

Result:

A point inside (or touching) the operand-geometry.

Examples:

```
point var0 <- any_location_in(square(5)); // var0 equals a point in  
the square, for example : {3,4.6}.
```

See also: [closest_points_with](#), [farthest_point_to](#), [points_at](#),

any_point_in

Same signification as [any_location_in](#)

append_horizontally

Possible uses:

- `matrix append_horizontally matrix` --> `matrix`
- `append_horizontally (matrix, matrix)` --> `matrix`

Result:

A matrix resulting from the concatenation of the rows of the two given matrices.

append_vertically

Possible uses:

- `matrix append_vertically matrix` ---> `matrix`
- `append_vertically (matrix, matrix)` ---> `matrix`

Result:

A matrix resulting from the concatenation of the columns of the two given matrices.

Examples:

```
matrix var0 <- matrix([[1,2],[3,4]]) append_vertically  
matrix([[1,2],[3,4]]); // var0 equals matrix([[1,2,1,2],[3,4,3,4]])
```

arc

Possible uses:

- `arc (float, float, float)` ---> `geometry`
- `arc (float, float, float, bool)` ---> `geometry`

Result:

An arc, which radius is equal to the first operand, heading to the second, amplitude

to the third and a boolean indicating whether to return a linestring or a polygon to the fourth

Comment:

the center of the arc is by default the location of the current agent in which has been called this operator.the center of the arc is by default the location of the current agent in which has been called this operator. This operator returns a polygon by default.

Special cases:

- returns a point if the radius operand is lower or equal to 0.
- returns a point if the radius operand is lower or equal to 0.

Examples:

```
geometry var0 <- arc(4,45,90, false); // var0 equals a geometry as an
arc of radius 4, in a direction of 45° and an amplitude of 90°, which
only contains the points on the arc
geometry var1 <- arc(4,45,90); // var1 equals a geometry as an arc of
radius 4, in a direction of 45° and an amplitude of 90°
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [super_ellipse](#), [rectangle](#), [square](#), [circle](#), [ellipse](#), [triangle](#),

around

Possible uses:

- `float around unknown --> geometry`
- `around (float , unknown) --> geometry`

Result:

A geometry resulting from the difference between a buffer around the right-operand casted in geometry at a distance left-operand (right-operand buffer left-operand) and the right-operand casted as geometry.

Special cases:

- returns a circle geometry of radius right-operand if the left-operand is nil

Examples:

```
geometry var0 <- 10 around circle(5); // var0 equals the ring geometry  
between 5 and 10.
```

See also: [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

as

Possible uses:

- `unknown as any GAML type` ---> `unknown`
- `as (unknown, any GAML type)` ---> `unknown`

Result:

Casting of the first argument into a given type

Comment:

It is equivalent to the application of the type operator on the left operand.

Examples:

```
int var0 <- 3.5 as int; // var0 equals int(3.5)
```

as_4_grid

Possible uses:

- `geometry as_4_grid point --> matrix`
- `as_4_grid (geometry , point) --> matrix`

Result:

A matrix of square geometries (grid with 4-neighborhood) with dimension given by the right-hand operand (`{nb_cols, nb_lines}`) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

Examples:

```
matrix var0 <- self as_4_grid {10, 5}; // var0 equals the matrix of
square geometries (grid with 4-neighborhood) with 10 columns and 5
lines corresponding to the square tessellation of the geometry of the
agent applying the operator.
```

See also: [as_grid](#), [as_hexagonal_grid](#),

as_distance_graph

Possible uses:

- `container as_distance_graph float --> graph`
- `as_distance_graph (container, float) --> graph`
- `as_distance_graph (container, float, species) --> graph`

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices close enough (less than a distance, right-hand operand).

Comment:

`as_distance_graph` is more efficient for a list of points than `as_intersection_graph`.

Examples:

```
list(ant) as_distance_graph 3.0
```

See also: [as_intersection_graph](#), [as_edge_graph](#),

as_driving_graph

Possible uses:

- `container as_driving_graph container --> graph`
- `as_driving_graph (container, container) --> graph`

Result:

creates a graph from the list/map of edges given as operand and connect the node to the edge

Examples:

```
as_driving_graph(road, node) --: build a graph while using the road  
agents as edges and the node agents as nodes
```

See also: [as_intersection_graph](#), [as_distance_graph](#), [as_edge_graph](#),

as_edge_graph

Possible uses:

- `as_edge_graph` (`container`) -> `graph`
- `as_edge_graph` (`map`) -> `graph`
- `container as_edge_graph float` -> `graph`
- `as_edge_graph` (`container`, `float`) -> `graph`
- `container as_edge_graph container` -> `graph`
- `as_edge_graph` (`container`, `container`) -> `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 edges

```
graph var0 <-
as_edge_graph([line([{1,5},{12,45}]),line([{12,45},{34,56}])]); //  
var0 equals a graph with two edges and three vertices
```

- if the operand is a list and a tolerance (max distance in meters to consider that 2 points are the same node) is given, the graph will be built with elements of the list as edges and two edges will be connected by a node if the distance between their extremity (first or last points) are at distance lower or equal to the tolerance

```
graph var1 <-
as_edge_graph([line([{1,5},{12,45}]),line([{13,45},{34,56}])],1); //  
var1 equals a graph with two edges and three vertices
```

- if the operand is a map, the graph will be built by creating edges from pairs of the map

```
graph var2 <- as_edge_graph([{1,5}:{12,45},{12,45}:{34,56}]); //  
var2 equals a graph with these three vertices and two edges
```

See also: [as_intersection_graph](#), [as_distance_graph](#),

as_grid

Possible uses:

- `geometry as_grid point --> matrix`
- `as_grid (geometry , point) --> matrix`

Result:

A matrix of square geometries (grid with 8-neighborhood) with dimension given by the right-hand operand (`{nb_cols, nb_lines}`) corresponding to the square tessellation of the left-hand operand geometry (geometry, agent)

Examples:

```
matrix var0 <- self as_grid {10, 5}; // var0 equals a matrix of square geometries (grid with 8-neighborhood) with 10 columns and 5 lines corresponding to the square tessellation of the geometry of the agent applying the operator.
```

See also: [as_4_grid](#), [as_hexagonal_grid](#),

as_hexagonal_grid

Possible uses:

- `geometry as_hexagonal_grid point` ---> `list<geometry>`
- `as_hexagonal_grid (geometry , point)` ---> `list<geometry>`

Result:

A list of geometries (hexagonal) corresponding to the hexagonal tessellation of the first operand geometry

Examples:

```
list<geometry> var0 <- self as_hexagonal_grid {10, 5}; // var0 equals list of geometries (hexagonal) corresponding to the hexagonal
```

See also: [as_4_grid](#), [as_grid](#),

as_int

Possible uses:

- `string as_int int ---> int`
- `as_int (string, int) ---> int`

Result:

parses the string argument as a signed integer in the radix specified by the second argument.

Special cases:

- if the left operand is nil or empty, as_int returns 0
- if the left operand does not represent an integer in the specified radix, as_int throws an exception

Examples:

```
int var0 <- '20' as_int 10; // var0 equals 20
int var1 <- '20' as_int 8; // var1 equals 16
int var2 <- '20' as_int 16; // var2 equals 32
int var3 <- '1F' as_int 16; // var3 equals 31
int var4 <- 'hello' as_int 32; // var4 equals 18306744
```

See also: [int](#),

as_intersection_graph

Possible uses:

- container `as_intersection_graph` float ---> graph
- `as_intersection_graph` (container , float) ---> graph
- `as_intersection_graph` (container , float , species) ---> graph

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices with an intersection (with a given tolerance). creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices with an intersection (with a given tolerance).

Comment:

`as_intersection_graph` is more efficient for a list of geometries (but less accurate) than `as_distance_graph`.

Examples:

```
list(ant) as_intersection_graph 0.5
```

See also: [as_distance_graph](#), [as_edge_graph](#),

as_json_string

Possible uses:

- `as_json_string` (`container`) ---> `string`

Result:

Tries to convert the container into a json-formatted string

Special cases:

- With a map:

```
string var0 <- as_json_string(map('int_value':1,  
'string_value':'some words', 'tab':[1, 2, 3])); // var0 equals  
{"int_value":1,"string_value":"some words","tab":[1,2,3]}
```

- With an array:

```
string var1 <- as_json_string([1, 2, 3, 'some words']); // var1 equals  
[1,2,3,"some words"]
```

as_map

Possible uses:

- `container as_map any expression` ---> `map`
- `as_map (container , any expression)` ---> `map`

Result:

produces a new map from the evaluation of the right-hand operand for each element

of the left-hand operand

Comment:

the right-hand operand should be a pair

Special cases:

- if the left-hand operand is nil, as_map throws an error.

Examples:

```
map<int,int> var0 <- [1,2,3,4,5,6,7,8] as_map (each::(each * 2)); //  
var0 equals [1::2, 2::4, 3::6, 4::8, 5::10, 6::12, 7::14, 8::16]  
map<int,int> var1 <- [1::2,3::4,5::6] as_map (each::(each * 2)); //  
var1 equals [2::4, 4::8, 6::12]
```

as_matrix

Possible uses:

- `unknown as_matrix point` --> `matrix`
- `as_matrix (unknown , point)` --> `matrix`

Result:

casts the left operand into a matrix with right operand as preferred size

Comment:

This operator is very useful to cast a file containing raster data into a matrix. Note that both components of the right operand point should be positive, otherwise an exception is raised. The operator as_matrix creates a matrix of preferred size. It fills in it with elements of the left operand until the matrix is full. If the size is 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](#),

as_path

Possible uses:

- `list<geometry> as_path graph ---> path`
- `as_path (list<geometry>, graph) ---> path`

Result:

create a graph path from the list of shape

Examples:

```
path var0 <- [road1,road2,road3] as_path my_graph; // var0 equals a  
path road1->road2->road3 of my_graph
```

as_spatial_graph

Possible uses:

- `as_spatial_graph (graph) --->`
`msi.gama.metamodel.topology.graph.ISpatialGraph`

Result:

Creates a spatial graph out of an arbitrary graph. If the argument is already a spatial

graph, returns it unchanged. If it contains geometrical nodes or edges, they are kept unchanged

asin

Possible uses:

- `asin (int) ---> float`
- `asin (float) ---> float`

Result:

the arcsin of the operand

Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

Examples:

```
float var0 <- asin (90); // var0 equals #nan
float var1 <- asin (0); // var1 equals 0.0
```

See also: [acos](#), [atan](#), [sin](#),

at

Possible uses:

- `species at int ---> agent`
- `at (species , int) ---> agent`
- `list at int ---> unknown`

- `at (list , int) ---> unknown`
- `matrix at point ---> unknown`
- `at (matrix , point) ---> unknown`
- `string at int ---> string`
- `at (string , int) ---> string`
- `container at unknown ---> unknown`
- `at (container , unknown) ---> unknown`

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 var1 <- [1, 2, 3] at 2; // var1 equals 3
point var2 <- [{1,2}, {3,4}, {5,6}] at 0; // var2 equals {1.0,2.0}
```

Examples:

```
string var0 <- 'abcdef' at 0; // var0 equals 'a'
```

See also: [contains_all](#), [contains_any](#),

at_distance

Possible uses:

- `container<unknown, geometry>` `at_distance` `float` ---> `list<geometry>`
- `at_distance` (`container<unknown, geometry>` , `float`)---> `list<geometry>`

Result:

A list of agents or geometries among the left-operand list that are located at a distance \leq the right operand from the caller agent (in its topology)

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] at_distance 20; // var0 equals
the agents of the list located at a distance  $\leq$  20 from the caller
agent (in the same order).
```

See also: [neighbors_at](#), [neighbors_of](#), [agent_closest_to](#), [agents_inside](#), [closest_to](#), [inside](#), [overlapping](#),

at_location

Possible uses:

- `geometry at_location point` ---> `geometry`
- `at_location (geometry , point)`---> `geometry`

Result:

A geometry resulting from the tran of a translation to the right-hand operand point of the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self at_location {10, 20}; // var0 equals the  
geometry resulting from a translation to the location {10, 20} of the  
left-hand geometry (or agent).  
float var1 <- (box({10, 10 , 5}) at_location  
point(50,50,0)).location.x; // var1 equals 50.0
```

atan

Possible uses:

- `atan (int)`---> `float`
- `atan (float)`---> `float`

Result:

Returns the value (in the interval [-90,90], in decimal degrees) of the arctan of the operand (which can be any real number).

Examples:

```
float var0 <- atan (1); // var0 equals 45.0
```

See also: [acos](#), [asin](#), [tan](#),

atan2

Possible uses:

- float atan2 float -> float
- atan2 (float, float) -> float

Result:

the atan2 value of the two operands.

Comment:

The function atan2 is the arctangent function with two arguments. The purpose of using two arguments instead of one is to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle, which is not possible for the single-argument arctangent function.

Examples:

```
float var0 <- atan2 (0,0); // var0 equals 0.0
```

See also: [atan](#), [acos](#), [asin](#),

attributes

Possible uses:

- `attributes` (`any`) ---> `attributes`

Result:

casts the operand in a attributes object.

auto_correlation

Possible uses:

- `container auto_correlation int` ---> `float`
- `auto_correlation (container, int)` ---> `float`

Result:

Returns the auto-correlation of a data sequence given some lag

Examples:

```
float var0 <- auto_correlation([1,0,1,0,1,0],2); // var0 equals 1
float var1 <- auto_correlation([1,0,1,0,1,0],1); // var1 equals -1
```

Version: 1.9.1

Operators (B to C)

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 notation 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, ..., operand)`) or a special infix syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any side effects on their operands. For instance, the `shuffle` operator, which randomizes the positions of elements in a list, does not modify its list

operand but returns a new shuffled list.

Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely:

- the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values).
- it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else).
- next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test)
- next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`)
- next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators)
- next the unary operators `-` and `!`
- next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points)
- and finally the functional operators, which have the highest priority of all.

Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
    int min(int x, int y) {
        return x > y ? x : y;
    }
}
```

Any agent instance of spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
    }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considered as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

`box`, `cone3D`, `cube`, `cylinder`, `hexagon`, `pyramid`, `set_z`, `sphere`, `teapot`,

Arithmetic operators

`-`, `/`, `^`, `*`, `+`, `abs`, `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cos_rad`, `div`, `even`, `exp`, `fact`, `floor`, `hypot`, `is_finite`, `is_number`, `ln`, `log`, `mod`, `round`, `signum`, `sin`, `sin_rad`, `sqrt`, `tan`, `tan_rad`, `tanh`, `with_precision`,

BDI

`add_values`, `and`, `eval_when`, `get_about`, `get_agent`, `get_agent_cause`, `get_belief_op`, `get_belief_with_name_op`, `get_beliefs_op`, `get_beliefs_with_name_op`, `get_current_intention_op`, `get_decay`, `get_desire_op`, `get_desire_with_name_op`, `get_desires_op`, `get_desires_with_name_op`, `get_dominance`, `get_familiarity`,

get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or, set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_values,

Casting operators

as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of, to_gaml, to_geojson, to_list, with_size, with_style,

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean, median, palette, rgb, rnd_color, scale, sum, to_hsb,

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect, contains, contains_all, contains_any, contains_key, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

Date-related operators

-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours,

minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours, plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,

Dates

Displays

horizontal, stack, vertical,

edge

edge_between, strahler,

EDP-related operators

diff, diff2,

Files-related operators

copy_file, crs, csv_file, delete_file, dxf_file, evaluate_sub_model, file_exists, folder, folder_exists, gaml_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphtsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphtsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,

GamaMetaType

type_of,

Gen*

add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,

Graphs-related operators

add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, as_spatial_graph, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_random_graph, generate_watts_strogatz, girvan_newman_clustering, grid_cells_to_graph, in_degree_of, in_edges_of, k_spanning_tree_clustering, label_propagation_clustering, layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid, load_shortest_paths, main_connected_component, max_flow_between, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_from, rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of, undirected, use_cache, weight_of, with_k_shortest_path_algorithm, with_shortest_path_algorithm, with_weights,

Grid-related operators

as_4_grid, as_grid, as_hexagonal_grid, cell_at, cells_in, cells_overlapping, field, grid_at, neighbors_of, path_between, points_in, values_in,

ImageOperators

*, antialiased, blend, blurred, brighter, clipped_with, darker, grayscale, horizontal_flip, image, matrix, rotated_by, sharpened, snapshot, tinted_with, vertical_flip, with_height, with_size, with_width,

Iterator operators

accumulate, all_match, as_map, collect, count, create_map, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, none_matches, one_matches, product_of, sort_by, sum_of, variance_of, where, where, where, with_max_of, with_min_of,

List-related operators

all_indexes_of, copy_between, index_of, last_index_of,

Logical operators

`: ! ? add_3Dmodel add_geometry add_icon and or xor,`

Map comparaison operators

`fuzzy_kappa fuzzy_kappa_sim kappa kappa_sim percent_absolute_deviation,`

Map-related operators

`as_map create_map index_of last_index_of,`

Matrix-related operators

`- / . * + append_horizontally append_vertically column_at columns_list determinant eigenvalues index_of inverse last_index_of row_at rows_list shuffle trace transpose,`

multicriteria operators

`electre_DM evidence_theory_DM fuzzy_choquet_DM promethee_DM weighted_means_DM,`

Path-related operators

`agent_from_geometry all_pairs_shortest_path as_path load_shortest_paths max_flow_between, path_between path_to paths_between use_cache,`

Pedestrian

`generate_pedestrian_network,`

Points-related operators

`- / * + < <= > >= add_point angle_between any_location_in centroid closest_points_with farthest_point_to, grid_at norm points_along points_at points_on,`

Random operators

`binomial`, `flip`, `gamma_density`, `gamma_rnd`, `gamma_trunc_rnd`, `gauss`, `generate_terrain`, `lognormal_density`,
`lognormal_rnd`, `lognormal_trunc_rnd`, `poisson`, `rnd`, `rnd_choice`, `sample`, `shuffle`, `skew_gauss`, `truncated_gauss`,
`weibull_density`, `weibull_rnd`, `weibull_trunc_rnd`,

ReverseOperators

`restore_simulation`, `restore_simulation_from_file`, `save_simulation`, `serialize`, `serialize_agent`,

Shape

`arc`, `box`, `circle`, `cone`, `cone3D`, `cross`, `cube`, `curve`, `cylinder`, `ellipse`, `elliptical_arc`, `envelope`, `geometry_collection`,
`hexagon`, `line`, `link`, `plan`, `polygon`, `polyhedron`, `pyramid`, `rectangle`, `sphere`, `square`, `squircle`, `teapot`, `triangle`,

Spatial operators

`-`, `*`, `+`, `add_point`, `agent_closest_to`, `agent_farthest_to`, `agents_at_distance`, `agents_covering`, `agents_crossing`,
`agents_inside`, `agents_overlapping`, `agents_partially_overlapping`, `agents_touching`, `angle_between`,
`any_location_in`, `arc`, `around`, `as_4_grid`, `as_driving_graph`, `as_grid`, `as_hexagonal_grid`, `at_distance`, `at_location`,
`box`, `centroid`, `circle`, `clean`, `clean_network`, `closest_points_with`, `closest_to`, `cone`, `cone3D`, `convex_hull`, `covering`,
`covers`, `cross`, `crosses`, `crossing`, `crs`, `CRS_transform`, `cube`, `curve`, `cylinder`, `direction_between`, `disjoint_from`,
`distance_between`, `distance_to`, `ellipse`, `elliptical_arc`, `envelope`, `farthest_point_to`, `farthest_to`,
`geometry_collection`, `gini`, `hexagon`, `hierarchical_clustering`, `IDW`, `inside`, `inter`, `intersects`, `inverse_rotation`,
`k_nearest_neighbors`, `line`, `link`, `masked_by`, `moran`, `neighbors_at`, `neighbors_of`, `normalized_rotation`,
`overlapping`, `overlaps`, `partially_overlapping`, `partially_overlaps`, `path_between`, `path_to`, `plan`, `points_along`,
`points_at`, `points_on`, `polygon`, `polyhedron`, `pyramid`, `rectangle`, `rotated_by`, `rotation_composition`, `round`,
`scaled_to`, `set_z`, `simple_clustering_by_distance`, `simplification`, `skeletonize`, `smooth`, `sphere`, `split_at`,
`split_geometry`, `split_lines`, `square`, `squircle`, `teapot`, `to_GAMA_CRS`, `to_rectangles`, `to_segments`, `to_squares`,
`to_sub_geometries`, `touches`, `touching`, `towards`, `transformed_by`, `translated_by`, `triangle`, `triangulate`, `union`,
`using`, `voronoi`, `with_precision`, `without_holes`,

Spatial properties operators

`covers`, `crosses`, `intersects`, `partially_overlaps`, `touches`,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, at_distance, closest_to, covering, crossing, farthest_to, inside, neighbors_at, neighbors_of, overlapping, partially_overlapping, touching,

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

Spatial statistical operators

hierarchical_clustering, k_nearest_neighbors, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, inverse_rotation, normalized_rotation, rotated_by, rotation_composition, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum, build, chi_square, chi_square_complemented, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis, log_gamma, max, mean, mean_deviation, median, min, moment, moran, morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, t_test, variance,

Strings-related operators

+,<,<=,>,>=, at, capitalize, char, contains, contains_all, contains_any, copy_between, date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, regex_matches, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

SubModel

load_sub_model,

System

., choose, command, copy, copy_from_clipboard, copy_to_clipboard, copy_to_clipboard, dead, enter, eval_gaml, every, is_error, is_reachable, is_warning, play_sound, user_confirm, user_input_dialog, wizard, wizard_page,

Time-related operators

date, string,

Types-related operators

action, agent, attributes, BDIPlan, bool, container, conversation, directory, emotion, file, float, gaml_type, gen_population_generator, gen_range, geometry, graph, int, kml, list, map, matrix, mental_state, message, Norm, pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, species, topology, unknown,

User control operators

choose, enter, user_confirm, user_input_dialog, wizard, wizard_page,

Operators

BDIPlan

Possible uses:

- **BDIPlan** (any) ---> **BDIPlan**

Result:

casts the operand in a BDIPlan object.

before

Possible uses:

- **before** (date) ---> bool
- any expression **before** date ---> bool
- **before** (any expression , date) ---> bool

Result:

Returns true if the current_date of the model is strictly before the date passed in argument. Synonym of 'current_date < argument'

Examples:

```
reflex when: before(starting_date) {} // this reflex will never be run
```

beta

Possible uses:

- float **beta** float ---> float
- **beta** (float , float)---> float

Result:

Returns the beta function with arguments a, b.

Comment:

Checked on R. beta(4,5)

Examples:

```
float var0 <- beta(4,5) with_precision(4); // var0 equals 0.0036
```

beta_index

Possible uses:

- `beta_index` (`graph`) ---> `float`

Result:

returns the beta index of the graph (Measures the level of connectivity in a graph and is expressed by the relationship between the number of links (e) over the number of nodes (v) : $\text{beta} = e/v$.

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](#),

between

Possible uses:

- `date between date` ---> `bool`
- `between (date, date)` ---> `bool`
- `between (date, date, date)` ---> `bool`
- `between (int, int, int)` ---> `bool`
- `between (any expression, date, date)` ---> `bool`
- `between (float, float, float)` ---> `bool`

Result:

returns true if the first operand is bigger than the second operand and smaller than the third operand

Special cases:

- returns true if the first operand is between the two dates passed in arguments (both exclusive). Can be

combined with 'every' to express a frequency between two dates

```
bool var0 <- (date('2016-01-01') between(date('2000-01-01'), date('2020-02-02'))); // var0  
equals true  
// will return true every new day between these two dates, taking the first one as the  
starting point  
every(#day between(date('2000-01-01'), date('2020-02-02')))
```

- With only 2 date operands, it returns true if the current_date is between the 2 date operands.

```
bool var3 <- between(date('2000-01-01'), date('2020-02-02')); // var3 equals false
```

Examples:

```
bool var4 <- between(5, 1, 10); // var4 equals true  
bool var5 <- between(5.0, 1.0, 10.0); // var5 equals true
```

betweenness_centrality

Possible uses:

- `betweenness_centrality` (graph) --> map

Result:

returns a map containing for each vertex (key), its betweenness centrality (value): number of shortest paths passing through each vertex

Examples:

```
graph graphEpidemio <- graph([]);  
map var1 <- betweenness_centrality(graphEpidemio); // var1 equals the betweenness centrality  
index of the graph
```

biggest_cliques_of

Possible uses:

- `biggest_cliques_of` (graph) --> list<list>

Result:

returns the biggest cliques of a graph using the Bron-Kerbosch clique detection algorithm

Examples:

```
graph my_graph <- graph([]);  
list<list> var1 <- biggest_cliques_of (my_graph); // var1 equals the list of the biggest  
cliques as list
```

See also: [maximal_cliques_of](#),

binomial

Possible uses:

- `int binomial float --> int`
- `binomial (int , float) --> int`

Result:

A value from a random variable following a binomial distribution. The operands represent the number of experiments n and the success probability p.

Comment:

The binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p, cf. Binomial distribution on Wikipedia.

Examples:

```
int var0 <- binomial(15,0.6); // var0 equals a random positive integer
```

See also: [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#),

binomial_coeff

Possible uses:

- `int binomial_coeff int --> float`

- `binomial_coeff` (`int`, `int`) \rightarrow `float`

Result:

Returns n choose k as a double. Note the integerization of the double return value.

Examples:

```
float var0 <- binomial_coeff(10,2); // var0 equals 45
```

binomial_complemented

Possible uses:

- `binomial_complemented` (`int`, `int`, `float`) \rightarrow `float`

Result:

Returns the sum of the terms $k+1$ through n of the Binomial probability density, where n is the number of trials and P is the probability of success in the range 0 to 1.

Examples:

```
float var0 <- binomial_complemented(10,5,0.5) with_precision(2); // var0 equals 0.38
```

binomial_sum

Possible uses:

- `binomial_sum` (`int`, `int`, `float`) \rightarrow `float`

Result:

Returns the sum of the terms 0 through k of the Binomial probability density, where n is the number of trials and p is the probability of success in the range 0 to 1.

Examples:

```
float var0 <- binomial_sum(5,10,0.5) with_precision(2); // var0 equals 0.62
```

blend

Possible uses:

- `rgb blend rgb --> rgb`
- `blend (rgb, rgb) --> rgb`
- `blend (rgb, rgb, float) --> rgb`

Result:

Blend two colors with an optional ratio ($c1 * r + c2 * (1 - r)$) between 0 and 1

Special cases:

- If the ratio is omitted, an even blend is done

```
rgb var0 <- blend(#red, #blue); // var0 equals to a color very close to the purple
```

Examples:

```
rgb var1 <- blend(#red, #blue, 0.3); // var1 equals to a color between the purple and the blue
```

See also: [rgb](#), [hsb](#),

blend

Possible uses:

- `blend (image, image, float) --> image`

Result:

Blend two images with an optional ratio between 0 and 1 (determines the transparency of the second image, applied as an overlay to the first). The size of the resulting image is that of the first parameter. The original image is left untouched

Examples:

```
image var0 <- blend(img1, img2, 0.3); // var0 equals to a composed image with the two
```

blurred

Possible uses:

- `blurred` (`image`) ---> `image`

Result:

Application of a blurring filter to the image passed in parameter. This operation can be applied multiple times.
The original image is left untouched

bool

Possible uses:

- `bool` (`any`) ---> `bool`

Result:

casts the operand in a bool object.

box

Possible uses:

- `box` (`point`) ---> `geometry`
- `box` (`float`, `float`, `float`) ---> `geometry`

Result:

A box geometry which side sizes are given by the operands.

Comment:

the center of the box is by default the location of the current agent in which has been called this operator.the center of the box is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- box({10, 5, 5}); // var0 equals a geometry as a rectangle with width = 10,  
height = 5 depth= 5.  
float var1 <- (box({10, 10, 5}) at_location point(50,50,0)).location.y; // var1 equals 50.0  
geometry var2 <- box(10, 5, 5); // var2 equals a geometry as a rectangle with width = 10,  
height = 5 depth= 5.
```

See also: [around](#), [circle](#), [sphere](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [square](#), [cube](#), [triangle](#),

brewer_colors

Possible uses:

- `brewer_colors (string) --> list<rgb>`
- `string brewer_colors int --> list<rgb>`
- `brewer_colors (string, int) --> list<rgb>`

Result:

Build a list of colors of a given type (see website <http://colorbrewer2.org/>) with a given number of classes Build a list of colors of a given type (see website <http://colorbrewer2.org/>). The list of palettes can be obtained by calling brewer_palettes. This list can be safely modified afterwards (adding or removing colors)

Examples:

```
list<rgb> var0 <- list<rgb> colors <- brewer_colors("Pastel1", 5); // var0 equals a list of 5  
sequential colors in the palette named 'Pastel1'. The list of palettes can be obtained by  
calling brewer_palettes  
list<rgb> var1 <- list<rgb> colors <- brewer_colors("OrRd"); // var1 equals a list of 6 blue  
colors
```

See also: [brewer_palettes](#),

brewer_palettes

Possible uses:

- `brewer_palettes (int) --> list<string>`
- `int brewer_palettes int --> list<string>`
- `brewer_palettes (int, int) --> list<string>`

Result:

returns the list a palette with a given min number of classes and max number of classes) returns the list a palette with a given min number of classes)

Examples:

```
list<string> var0 <- list<string> palettes <- brewer_palettes(5,10);; // var0 equals a list of palettes that are composed of a min of 5 colors and a max of 10 colors  
list<string> var1 <- list<string> palettes <- brewer_palettes(3);; // var1 equals a list of palettes that are composed of a min of 3 colors
```

See also: [brewer_colors](#),

brighter

Possible uses:

- `brighter (image) ---> image`

Result:

Used to return an image 10% brigther. This operation can be applied multiple times in a row if greater than 10% changes in brightness are desired.

buffer

Same signification as [+](#)

build

Possible uses:

- `build (matrix) ---> regression`

Result:

returns the regression build from the matrix data (a row = an instance, the first value of each line is the y value) while using the given ordinary least squares method. Usage: build(data)

Examples:

```
build(matrix([[1.0, 2.0, 3.0, 4.0], [2.0, 3.0, 4.0, 2.0]]))
```

capitalize

Possible uses:

- `capitalize` (`string`) ---> `string`

Result:

Returns a string where the first letter is capitalized

Examples:

```
string var0 <- capitalize("abc"); // var0 equals 'Abc'
```

See also: [lower_case](#), [upper_case](#),

cartesian_product

Possible uses:

- `cartesian_product` (`list`) ---> `unknown`

ceil

Possible uses:

- `ceil` (`float`) ---> `float`

Result:

Maps the operand to the smallest following integer, i.e. the smallest integer not less than x.

Examples:

```
float var0 <- ceil(3); // var0 equals 3.0
```

See also: [floor](#), [round](#),

cell_at

Possible uses:

- `field cell_at point --> geometry`
 - `cell_at (field , point) --> geometry`
 - `cell_at (field , int , int) --> geometry`
-

cells_in

Possible uses:

- `field cells_in geometry --> list<geometry>`
 - `cells_in (field , geometry) --> list<geometry>`
-

cells_overlapping

Possible uses:

- `field cells_overlapping geometry --> list<geometry>`
 - `cells_overlapping (field , geometry) --> list<geometry>`
-

centroid

Possible uses:

- `centroid (geometry) --> point`

Result:

Centroid (weighted sum of the centroids of a decomposition of the area into triangles) of the operand-geometry.
Can be different to the location of the geometry

Examples:

```
point var0 <- centroid(world); // var0 equals the centroid of the square, for example :  
{50.0,50.0}.
```

See also: [any_location_in](#), [closest_points_with](#), [farthest_point_to](#), [points_at](#),

char

Possible uses:

- `char (int) ---> string`

Special cases:

- converts ACSII integer value to character

```
string var0 <- char (34); // var0 equals '''
```

chi_square

Possible uses:

- `float chi_square float ---> float`
- `chi_square (float , float)---> float`

Result:

Returns the area under the left hand tail (from 0 to x) of the Chi square probability density function with df degrees of freedom.

Examples:

```
float var0 <- chi_square(20.0,10) with_precision(3); // var0 equals 0.971
```

chi_square_complemented

Possible uses:

- `float chi_square_complemented float ---> float`
- `chi_square_complemented (float , float)---> float`

Result:

Returns the area under the right hand tail (from x to infinity) of the Chi square probability density function with

df degrees of freedom.

Examples:

```
float var0 <- chi_square_complemented(2,10) with_precision(3); // var0 equals 0.996
```

choose

Possible uses:

- `choose` (`string`, `any GAML type`, `unknown`, `list`) ---> `unknown`

Result:

Allows the user to choose a value by specifying a title, a type, and a list of possible values

circle

Possible uses:

- `circle` (`float`) ---> `geometry`
- `float` `circle` `point` ---> `geometry`
- `circle` (`float`, `point`) ---> `geometry`

Result:

A circle geometry which radius is equal to the operand.

Comment:

the center of the circle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the radius operand is lower or equal to 0.
- When circle is used with 2 operands, the second one is the center of the created circle.

```
geometry var1 <- circle(10,{80,30}); // var1 equals a geometry as a circle of radius 10, the center will be in the location {80,30}.
```

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](#),

clean

Possible uses:

- `clean` (`geometry`) ---> `geometry`

Result:

A geometry corresponding to the cleaning of the operand (geometry, agent, point)

Comment:

The cleaning corresponds to a buffer with a distance of 0.0

Examples:

```
geometry var0 <- clean(self); // var0 equals returns the geometry resulting from the cleaning  
of the geometry of the agent applying the operator.
```

clean_network

Possible uses:

- `clean_network` (`list<geometry>`, `float`, `bool`, `bool`) ---> `list<geometry>`

Result:

A list of polylines corresponding to the cleaning of the first operand (list of polyline geometry or agents), considering the tolerance distance given by the second operand; the third operator is used to define if the operator should as well split the lines at their intersections(true to split the lines); the last operand is used to specify if the operator should as well keep only the main connected component of the network. Usage:
`clean_network(lines:list of geometries or agents, tolerance: float, split_lines: bool, keepMainConnectedComponent: bool)`

Comment:

The cleaned set of polylines

Examples:

```
list<geometry> var0 <- clean_network(my_road_shapefile.contents, 1.0, true, false); // var0  
equals returns the list of polylines resulting from the cleaning of the geometry of the agent  
applying the operator with a tolerance of 1m, and splitting the lines at their intersections.  
list<geometry> var1 <- clean_network([line({10,10}, {20,20}),  
line({10,20},{20,10})],3.0,true,false); // var1 equals  
[line({10.0,20.0,0.0},{15.0,15.0,0.0}),line({15.0,15.0,0.0},{20.0,10.0,0.0}),  
line({10.0,10.0,0.0},{15.0,15.0,0.0}), line({15.0,15.0,0.0},{20.0,20.0,0.0})]
```

clipped_with**Possible uses:**

- `clipped_with` (`image`, `int`, `int`, `int`, `int`) ---> `image`

Result:

Used to crop the given image using a rectangle starting at the top-left x, y coordinates and expanding using the width and height. If one of the dimensions of the resulting image is 0, or if they are equal to that of the given image, returns it. The original image is left untouched

closest_points_with**Possible uses:**

- `geometry closest_points_with geometry` ---> `list<point>`
- `closest_points_with (geometry , geometry)` ---> `list<point>`

Result:

A list of two closest points between the two geometries.

Examples:

```
list<point> var0 <- geom1 closest_points_with(geom2); // var0 equals [pt1, pt2] with pt1 the  
closest point of geom1 to geom2 and pt1 the closest point of geom2 to geom1
```

See also: [any_location_in](#), [any_point_in](#), [farthest_point_to](#), [points_at](#),

closest_to

Possible uses:

- `container<unknown,geometry> closest_to geometry ---> geometry`
- `closest_to (container<unknown,geometry>, geometry) ---> geometry`
- `closest_to (container<unknown,geometry>, geometry, int) ---> list<geometry>`

Result:

The N agents or geometries among the left-operand list of agents, species or meta-population (addition of species), that are the closest to the operand (casted as a geometry). An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the closest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.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:

```
list<geometry> var0 <- [ag1, ag2, ag3] closest_to(self, 2); // var0 equals return the 2  
closest agents among ag1, ag2 and ag3 to the agent applying the operator.  
(species1 + species2) closest_to (self, 5)  
geometry var2 <- [ag1, ag2, ag3] closest_to(self); // var2 equals return the closest agent  
among ag1, ag2 and ag3 to the agent applying the operator.  
(species1 + species2) closest_to self
```

See also: [neighbors_at](#), [neighbors_of](#), [inside](#), [overlapping](#), [agents_overlapping](#), [agents_inside](#), [agent_closest_to](#),

collect

Possible uses:

- `container collect any expression ---> list`
- `collect (container, any expression) ---> list`

Result:

returns a new list, in which each element is the evaluation of the right-hand operand.

Comment:

collect is similar to accumulate except that accumulate always produces flat lists if the right-hand operand returns a list. In addition, collect can be applied to any container.

Special cases:

- if the left-hand operand is nil, collect throws an error

Examples:

```
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](#),

column_at

Possible uses:

- `matrix<unknown> column_at int ---> list<unknown>`
- `column_at (matrix<unknown>, int) ---> list<unknown>`

Result:

returns the column at a num_col (right-hand operand)

Examples:

```
list<unknown> var0 <-
matrix([["el11","el12","el13"],["el21","el22","el23"],["el31","el32","el33"]]) column_at 2; // 
var0 equals ["el31","el32","el33"]
```

See also: [row_at](#), [rows_list](#),

columns_list

Possible uses:

- `columns_list` (`matrix<unknown>`) ---> `list<list<unknown>>`

Result:

returns a list of the columns of the matrix, with each column as a list of elements

Examples:

```
list<list<unknown>> var0 <-  
columns_list(matrix([["el11", "el12", "el13"], ["el21", "el22", "el23"], ["el31", "el32", "el33"]]));  
// var0 equals [[{"el11", "el12", "el13"}, {"el21", "el22", "el23"}, {"el31", "el32", "el33"}]]
```

See also: [rows_list](#),

command

Possible uses:

- `command` (`string`) ---> `string`
- `string` `command` `string` ---> `string`
- `command` (`string`, `string`) ---> `string`
- `command` (`string`, `string`, `map<string, string>`) ---> `string`

Result:

command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence ' &' is used at the end. In this case, the result of the operator is an empty string. The basic form with only one string in argument uses the directory of the model and does not set any environment variables. Two other forms (with a directory and a map<string, string> of environment variables) are available.

cone

Possible uses:

- `cone` (`point`) ---> `geometry`
- `int` `cone` `int` ---> `geometry`
- `cone` (`int`, `int`) ---> `geometry`

Result:

A cone geometry which min and max angles are given by the operands.

Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- cone(0, 45); // var0 equals a geometry as a cone with min angle is 0 and max angle is 45.  
geometry var1 <- cone({0, 45}); // var1 equals a geometry as a cone with min angle is 0 and max angle is 45.
```

See also: [around](#), [circle](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

cone3D

Possible uses:

- `float` `cone3D` `float` ---> `geometry`
- `cone3D` (`float`, `float`) ---> `geometry`

Result:

A cone geometry which base radius size is equal to the first operand, and which the height is equal to the second operand.

Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- cone3D(10.0,5.0); // var0 equals a geometry as a cone with a base circle of radius 10 and a height of 5.
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

connected_components_of

Possible uses:

- `connected_components_of (graph) ---> list<list>`
- `graph connected_components_of bool ---> list<list>`
- `connected_components_of (graph , bool) ---> list<list>`

Result:

returns the connected components of a graph, i.e. the list of all vertices that are in the maximally connected component together with the specified vertex. returns the connected components of a graph, i.e. the list of all edges (if the boolean is true) or vertices (if the boolean is false) that are in the connected components.

Examples:

```
graph my_graph <- graph([]);  
list<list> var1 <- connected_components_of (my_graph); // var1 equals the list of all the components as list  
graph my_graph2 <- graph([]);  
list<list> var3 <- connected_components_of (my_graph2, true); // var3 equals the list of all the components as list
```

See also: [alpha_index](#), [connectivity_index](#), [nb_cycles](#),

connectivity_index

Possible uses:

- `connectivity_index` (`graph`) ---> `float`

Result:

returns a simple connectivity index. This number is estimated through the number of nodes (v) and of sub-graphs (p) : IC = (v - p) / (v - 1).

Examples:

```
graph graphEpidemio <- graph([]);  
float var1 <- connectivity_index(graphEpidemio); // var1 equals the connectivity index of the  
graph
```

See also: [alpha_index](#), [beta_index](#), [gamma_index](#), [nb_cycles](#),

container

Possible uses:

- `container` (`any`) ---> `container`

Result:

casts the operand in a container object.

contains

Possible uses:

- `string` `contains` `string` ---> `bool`
- `contains` (`string` , `string`)---> `bool`
- `container<KeyType,ValueType>` `contains` `unknown` ---> `bool`
- `contains` (`container<KeyType,ValueType>` , `unknown`)---> `bool`

Result:

true, if the container contains the right operand, false otherwise. 'contains' can also be written 'contains_value'.

On graphs, it is equivalent to calling 'contains_edge'

Comment:

the contains operator behavior depends on the nature of the operand

Special cases:

- if both operands are strings, returns true if the right-hand operand contains the right-hand pattern;
- if it is a map, contains, which can also be written 'contains_value', returns true if the operand is a value of the map
- if it is a pair, contains_key returns true if the operand is equal to the value of the pair
- 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 can be written 'contains_edge' and returns true if the operand is an edge of the graph, false otherwise (use 'contains_node' for testing the presence of a node)
- if it is a list or a matrix, contains returns true if the list or matrix contains the right operand

```
bool var1 <- [1, 2, 3] contains 2; // var1 equals true
bool var2 <- [{1,2}, {3,4}, {5,6}] contains {3,4}; // var2 equals true
```

Examples:

```
bool var0 <- 'abcded' contains 'bc'; // var0 equals true
```

See also: [contains_all](#), [contains_any](#), [contains_key](#),

contains_all

Possible uses:

- `string contains_all list --> bool`
- `contains_all (string , list)---> bool`
- `container contains_all container --> bool`
- `contains_all (container , container)---> 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 var0 <- "abcababc" contains_all ["ca", "xy"]; // var0 equals false
```

Examples:

```
bool var1 <- [1,2,3,4,5,6] contains_all [2,4]; // var1 equals true
bool var2 <- [1,2,3,4,5,6] contains_all [2,8]; // var2 equals false
bool var3 <- [1::2, 3::4, 5::6] contains_all [1,3]; // var3 equals false
bool var4 <- [1::2, 3::4, 5::6] contains_all [2,4]; // var4 equals true
```

See also: [contains](#), [contains_any](#),

contains_any

Possible uses:

- `string contains_any list` ---> `bool`
- `contains_any (string , list)` ---> `bool`
- `container contains_any container` ---> `bool`
- `contains_any (container , container)` ---> `bool`

Result:

true if the left operand contains one of the elements of the right operand, false otherwise

Comment:

the definition of contains depends on the container

Special cases:

- if the right operand is nil or empty, contains_any returns false

Examples:

```
bool var0 <- "abcababc" contains_any ["ca","xy"]; // var0 equals true
bool var1 <- [1,2,3,4,5,6] contains_any [2,4]; // var1 equals true
bool var2 <- [1,2,3,4,5,6] contains_any [2,8]; // var2 equals true
bool var3 <- [1::2, 3::4, 5::6] contains_any [1,3]; // var3 equals false
bool var4 <- [1::2, 3::4, 5::6] contains_any [2,4]; // var4 equals true
```

See also: [contains](#), [contains_all](#),

contains_edge

Possible uses:

- `graph contains_edge pair --> bool`
- `contains_edge (graph , pair) --> bool`
- `graph contains_edge unknown --> bool`
- `contains_edge (graph , unknown) --> 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 var0 <- graphEpidemio contains_edge (node(0)::node(3)); // var0 equals true
```

Examples:

```
graph graphFromMap <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
bool var2 <- graphFromMap contains_edge link({1,5},{12,45}); // var2 equals true
```

See also: [contains_vertex](#),

contains_key

Possible uses:

- `container<KeyType,ValueType> contains_key unknown ---> bool`
- `contains_key (container<KeyType,ValueType>, unknown) ---> bool`

Result:

true, if the left-hand operand -- the container -- contains a key -- or an index -- equal to the right-hand operand, false otherwise. On graphs, 'contains_key' is equivalent to calling 'contains_vertex'

Comment:

the behavior of contains_key depends on the nature of the container

Special cases:

- if it is a map, contains_key returns true if the operand is a key of the map
- if it is a pair, contains_key returns true if the operand is equal to the key of the pair
- if it is a matrix, contains_key returns true if the point operand is a valid index of the matrix (i.e. $\geq \{0,0\}$ and $< \{\text{rows}, \text{col}\}$)
- if it is a file, contains_key is applied to the file contents -- a container
- if it is a graph, contains_key returns true if the graph contains the corresponding vertex
- if it is a list, contains_key returns true if the right-hand operand is an integer and if it is a valid index (i.e. ≥ 0 and $< \text{length}$)

```
bool var0 <- [1, 2, 3] contains_key 3; // var0 equals false
bool var1 <- [{1,2}, {3,4}, {5,6}] contains_key 0; // var1 equals true
```

See also: [contains_all](#), [contains](#), [contains_any](#),

contains_node

Same signification as [contains_key](#)

contains_value

Same signification as [contains](#)

contains_vertex

Possible uses:

- `graph contains_vertex unknown --> bool`
- `contains_vertex (graph , 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](#),

conversation

Possible uses:

- `conversation (any) --> conversation`

Result:

casts the operand in a conversation object.

convex_hull

Possible uses:

- `convex_hull (geometry) --> geometry`

Result:

A geometry corresponding to the convex hull of the operand.

Examples:

```
geometry var0 <- convex_hull(self); // var0 equals the convex hull of the geometry of the agent applying the operator
```

copy

Possible uses:

- `copy` (`unknown`) ---> `unknown`

Result:

returns a copy of the operand.

copy_between

Possible uses:

- `copy_between` (`list`, `int`, `int`) ---> `list`
- `copy_between` (`string`, `int`, `int`) ---> `string`

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:

```
list var0 <- copy_between ([4, 1, 6, 9 ,7], 1, 3); // var0 equals [1, 6]
string var1 <- copy_between("abcabcabc", 2,6); // var1 equals "cabc"
```

copy_file

Possible uses:

- `string copy_file string --> bool`
- `copy_file (string, string) --> bool`
- `copy_file (string, string, bool) --> bool`

Result:

copy a file or a folder copy a file or a folder

Examples:

```
bool copy_file_ok <- copy_file("../includes/my_folder","../includes/my_new_folder");
bool copy_file_ok <- copy_file("../includes/my_folder","../includes/my_new_folder",true);
```

copy_from_clipboard

Possible uses:

- `copy_from_clipboard (any GAML type) --> unknown`

Result:

Tries to copy data from the clipboard by passing its expected type. Returns nil if it has not been correctly retrieved, or not retrievable using the given type or if GAMA is in a headless environment

Examples:

```
string copied <- copy_from_clipboard(string);
```

copy_to_clipboard

Possible uses:

- `copy_to_clipboard (string) --> bool`

Result:

Tries to copy the text in parameter to the clipboard and returns whether it has been correctly copied or not (for instance it might be impossible in a headless environment)

Examples:

```
bool copied <- copy_to_clipboard('text to copy');
```

copy_to_clipboard

Possible uses:

- `copy_to_clipboard` (`image`) ---> `bool`

Result:

Tries to copy the given image to the clipboard and returns whether it has been correctly copied or not (for instance it might be impossible in a headless environment)

Examples:

```
bool copied <- copy_to_clipboard(img);
```

correlation

Possible uses:

- `container correlation container` ---> `float`
- `correlation` (`container`, `container`) ---> `float`

Result:

Returns the correlation of two data sequences (having the same size)

Examples:

```
float var0 <- correlation([1,2,1,3,1,2], [1,2,1,3,1,2]) with_precision(4); // var0 equals 1.2
float var1 <- correlation([13,2,1,4,1,2], [1,2,1,3,1,2]) with_precision(2); // var1 equals -0.21
```

cos

Possible uses:

- `cos (float) -> float`
- `cos (int) -> float`

Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- cos (0.0); // var0 equals 1.0
float var1 <- cos(360.0); // var1 equals 1.0
float var2 <- cos(-720.0); // var2 equals 1.0
float var3 <- cos (0); // var3 equals 1.0
float var4 <- cos(360); // var4 equals 1.0
float var5 <- cos(-720); // var5 equals 1.0
```

See also: [sin](#), [tan](#),

cos_rad

Possible uses:

- `cos_rad (float) -> float`

Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in radians).

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- cos_rad(0.0); // var0 equals 1.0
float var1 <- cos_rad(#pi); // var1 equals -1.0
```

See also: [sin](#), [tan](#),

count

Possible uses:

- `container count any expression ---> int`
- `count (container, any expression)---> int`

Result:

returns an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

Special cases:

- if the left-hand operand is nil, count throws an error

Examples:

```
int var0 <- [1,2,3,4,5,6,7,8] count (each > 3); // var0 equals 5
// Number of nodes of graph g2 without any out edge
graph g2 <- graph([]);
int var3 <- g2 count (length(g2 out_edges_of each) = 0 ) ; // var3 equals the total number of
out edges
// Number of agents node with x > 32
int n <- (list(node)) count (round(node(each).location.x) > 32);
int var6 <- [1::2, 3::4, 5::6] count (each > 4); // var6 equals 1
```

See also: [group_by](#),

covariance

Possible uses:

- `container covariance container ---> float`
- `covariance (container , container) ---> float`

Result:

Returns the covariance of two data sequences

Examples:

```
float var0 <- covariance([13,2,1,4,1,2], [1,2,1,3,1,2]) with_precision(2); // var0 equals -0.67
```

covering

Possible uses:

- `container<unknown,geometry> covering geometry ---> list<geometry>`
- `covering (container<unknown,geometry> , geometry) ---> list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), covering the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] covering(self); // var0 equals the agents among ag1, ag2 and ag3 that cover the shape of the right-hand argument.  
list<geometry> var1 <- (species1 + species2) covering (self); // var1 equals the agents among species species1 and species2 that covers the shape of the right-hand argument.
```

See also: [neighbors_at](#), [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [inside](#), [agents_inside](#), [agent_closest_to](#),

covers

Possible uses:

- `geometry covers geometry --> bool`
- `covers (geometry, geometry) --> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) covers the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- square(5) covers square(2); // var0 equals true
```

See also: [disjoint_from](#), [crosses](#), [overlaps](#), [partially_overlaps](#), [touches](#),

create_map

Possible uses:

- `list create_map list --> map`
- `create_map (list, list) --> map`

Result:

returns a new map using the left operand as keys for the right operand

Special cases:

- if the left operand contains duplicates, `create_map` throws an error.
- if both operands have different lengths, choose the minimum length between the two operands for the size of the map

Examples:

```
map<int, string> var0 <- create_map([0,1,2], ['a', 'b', 'c']); // var0 equals  
[0:'a', 1:'b', 2:'c']
```

cropped_to

Same signification as [clipped_with](#)

cross

Possible uses:

- `cross (float) ---> geometry`
- `float cross float ---> geometry`
- `cross (float , float) ---> geometry`

Result:

A cross, which radius is equal to the first operand (and eventually the width of the lines for the second)

Examples:

```
geometry var0 <- cross(10); // var0 equals a geometry as a cross of radius 10
geometry var1 <- cross(10,2); // var1 equals a geometry as a cross of radius 10, and with a
width of 2 for the lines
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [super_ellipse](#), [rectangle](#), [square](#), [circle](#), [ellipse](#), [triangle](#),

crosses

Possible uses:

- `geometry crosses geometry ---> bool`
- `crosses (geometry , geometry) ---> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) crosses the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.
- if one operand is a point, returns false.

Examples:

```
bool var0 <- polyline([{"10,10"}, {"20,20}]) crosses polyline([{"10,20"}, {"20,10}]); // var0 equals true  
bool var1 <- polyline([{"10,10"}, {"20,20}]) crosses {15,15}; // var1 equals true  
bool var2 <- polyline([{"0,0"}, {"25,25}]) crosses polygon([{"10,10"}, {"10,20"}, {"20,20"}, {"20,10}]); // var2 equals true
```

See also: [disjoint_from](#), [intersects](#), [overlaps](#), [partially_overlaps](#), [touches](#),

crossing

Possible uses:

- `container<unknown,geometry> crossing geometry` ---> `list<geometry>`
- `crossing (container<unknown,geometry>, geometry)` ---> `list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), crossing the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] crossing(self); // var0 equals the agents among ag1, ag2 and ag3 that cross the shape of the right-hand argument.  
list<geometry> var1 <- (species1 + species2) crossing (self); // var1 equals the agents among species species1 and species2 that cross the shape of the right-hand argument.
```

See also: [neighbors_at](#), [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [inside](#), [agents_inside](#), [agent_closest_to](#),

crs

Possible uses:

- `crs (file)` ---> `string`

Result:

the Coordinate Reference System (CRS) of the GIS file

Examples:

```
string var0 <- crs(my_shapefile); // var0 equals the crs of the shapefile
```

CRS_transform

Possible uses:

- `CRS_transform` (`geometry`) ---> `geometry`
- `geometry` `CRS_transform` `string` ---> `geometry`
- `CRS_transform` (`geometry`, `string`) ---> `geometry`
- `CRS_transform` (`geometry`, `string`, `string`) ---> `geometry`

Special cases:

- returns the geometry corresponding to the transformation of the given geometry from the first CRS to the second CRS (Coordinate Reference System)

```
geometry var0 <- {8.35, 47.22} CRS_transform("EPSG:4326", "EPSG:4326"); // var0 equals  
{929517.7481238344, 5978057.894895313, 0.0}
```

- returns the geometry corresponding to the transformation of the given geometry by the current CRS (Coordinate Reference System), the one corresponding to the world's agent one

```
geometry var1 <- CRS_transform(shape); // var1 equals a geometry corresponding to the agent  
geometry transformed into the current CRS
```

- returns the geometry corresponding to the transformation of the given geometry by the left operand CRS (Coordinate Reference System)

```
geometry var2 <- shape CRS_transform("EPSG:4326"); // var2 equals a geometry corresponding to  
the agent geometry transformed into the EPSG:4326 CRS
```

csv_file

Possible uses:

- `csv_file` (`string`) ---> `file`

- `string csv_file bool --> file`
- `csv_file (string , bool) --> file`
- `string csv_file string --> file`
- `csv_file (string , string) --> file`
- `string csv_file matrix<unknown> --> file`
- `csv_file (string , matrix<unknown>) --> file`
- `csv_file (string, string, bool) --> file`
- `csv_file (string, string, any GAML type) --> file`
- `csv_file (string, string, string, bool) --> file`
- `csv_file (string, string, string, any GAML type) --> file`
- `csv_file (string, string, any GAML type, bool) --> file`
- `csv_file (string, string, any GAML type, point) --> file`

Result:

Constructs a file of type csv. Allowed extensions are limited to csv, tsv

Special cases:

- `csv_file(string)`: This file constructor allows to read a CSV file with the default separator (coma), no header, and no assumption on the type of data. No text qualifier will be used

```
csv_file f <- csv_file("file.csv");
```

- `csv_file(string,bool)`: This file constructor allows to read a CSV file with the default separator (coma), with specifying if the model has a header or not (boolean), and no assumption on the type of data. No text qualifier will be used

```
csv_file f <- csv_file("file.csv",true);
```

- `csv_file(string,string)`: This file constructor allows to read a CSV file and specify the separator used, without making any assumption on the type of data. Headers should be detected automatically if they exist. No text qualifier will be used

```
csv_file f <- csv_file("file.csv", ";");
```

- `csv_file(string,string,bool)`: This file constructor allows to read a CSV file and specify (1) the separator used; (2) if the model has a header or not, without making any assumption on the type of data. No text qualifier will be used

```
csv_file f <- csv_file("file.csv", ";", true);
```

- `csv_file(string,string,string,bool)`: This file constructor allows to read a CSV file and specify (1) the separator used; (2) the text qualifier used; (3) if the model has a header or not, without making any assumption on the type of data

```
csv_file f <- csv_file("file.csv", ';', '"', true);
```

- `csv_file(string,string,any GAML type)`: This file constructor allows to read a CSV file with a given separator, no header, and the type of data. No text qualifier will be used

```
csv_file f <- csv_file("file.csv", ";", int);
```

- `csv_file(string,string,string,any GAML type)`: This file constructor allows to read a CSV file and specify the separator, text qualifier to use, and the type of data to read. Headers should be detected automatically if they exist.

```
csv_file f <- csv_file("file.csv", ';', '"', int);
```

- `csv_file(string,string,any GAML type,bool)`: This file constructor allows to read a CSV file with a given separator, the type of data, with specifying if the model has a header or not (boolean). No text qualifier will be used

```
csv_file f <- csv_file("file.csv", ";", int, true);
```

- `csv_file(string,string,any GAML type,point)`: This file constructor allows to read a CSV file with a given separator, the type of data, with specifying the number of cols and rows taken into account. No text qualifier will be used

```
csv_file f <- csv_file("file.csv", ";", int, true, {5, 100});
```

- `csv_file(string,matrix<unknown>)`: This file constructor allows to store a matrix in a CSV file (it does not save it - just store it in memory),

```
csv_file f <- csv_file("file.csv", matrix([10,10],[10,10]));
```

See also: [is_csv](#),

cube

Possible uses:

- `cube (float) ---> geometry`

Result:

A cube geometry which side size is equal to the operand.

Comment:

the center of the cube is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- cube(10); // var0 equals a geometry as a square of side size 10.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [triangle](#),

curve

Possible uses:

- `curve (point, point, point) ---> geometry`
- `curve (point, point, float) ---> geometry`
- `curve (point, point, float, float) ---> geometry`
- `curve (point, point, point, int) ---> geometry`
- `curve (point, point, point, point) ---> geometry`
- `curve (point, point, float, bool) ---> geometry`
- `curve (point, point, point, point, int) ---> geometry`
- `curve (point, point, float, bool, int) ---> geometry`
- `curve (point, point, float, int, float) ---> geometry`
- `curve (point, point, float, bool, int, float) ---> geometry`
- `curve (point, point, float, int, float, float) ---> geometry`

Result:

The operator computes a Bezier curve geometry between the given operators, with 10 or a given number of points, and from left to right or right to left.

Special cases:

- if one of the operand is nil, returns nil
- When used with 2 points, a float coefficient, a boolean, an integer number of points, and a float proportion, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points - the boolean is used to specified if it is the right side and the last value to indicate where is the inflection point (between 0.0 and 1.0 - default 0.5).

```
geometry var0 <- curve({0,0},{10,10}, 0.5, false, 100, 0.8); // var0 equals a cubic Bezier curve geometry composed of 100 points from p0 to p1 at the right side.
```

- When used with 2 points, a float coefficient, and a float angle, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius considering the given rotation angle (90 = along the z axis).

```
geometry var1 <- curve({0,0},{10,10}, 0.5, 90); // var1 equals a cubic Bezier curve geometry composed of 100 points from p0 to p1 at the right side.
```

- When used with 4 points and an integer number of points, it computes a cubic Bezier curve geometry built from the four given points composed of a given number of points. If the number of points is lower than 2, it returns nil.

```
geometry var2 <- curve({0,0}, {0,10}, {10,10}); // var2 equals a cubic Bezier curve geometry composed of 10 points from p0 to p3.
```

- When used with 3 points, it computes a quadratic Bezier curve geometry built from the three given points and composed of 10 points.

```
geometry var3 <- curve({0,0}, {0,10}, {10,10}); // var3 equals a quadratic Bezier curve geometry composed of 10 points from p0 to p2.
```

- When used with 2 points, a float coefficient, a boolean, and an integer number of points, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points - the boolean is used to specified if it is the right side.

```
geometry var4 <- curve({0,0},{10,10}, 0.5, false, 100); // var4 equals a cubic Bezier curve geometry composed of 100 points from p0 to p1 at the right side.
```

- When used with 2 points, a float coefficient, a boolean, an integer number of points, a float proportion, and a float angle, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points, considering the given inflection point (between 0.0 and 1.0 - default 0.5), and the given rotation angle (90 = along the z axis).

```
geometry var5 <- curve({0,0},{10,10}, 0.5, 100, 0.8, 90); // var5 equals a cubic Bezier curve
geometry composed of 100 points from p0 to p1 at the right side.
```

- When used with 3 points and an integer, it computes a quadratic Bezier curve geometry built from the three given points. If the last operand (number of points) is inferior to 2, returns nil

```
geometry var6 <- curve({0,0}, {0,10}, {10,10}, 20); // var6 equals a quadratic Bezier curve
geometry composed of 20 points from p0 to p2.
```

- When used with 4 points, it computes, it computes a cubic Bezier curve geometry built from the four given points and composed of 10 points.

```
geometry var7 <- curve({0,0}, {0,10}, {10,10}); // var7 equals a cubic Bezier curve geometry
composed of 10 points from p0 to p3.
```

- When used with 2 points, a float coefficient, a boolean, an integer number of points, and a float angle, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of the given number of points, considering the given rotation angle (90 = along the z axis).

```
geometry var8 <- curve({0,0},{10,10}, 0.5, 100, 90); // var8 equals a cubic Bezier curve
geometry composed of 100 points from p0 to p1 at the right side.
```

- When used with 2 points and a float coefficient, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of 10 points.

```
geometry var9 <- curve({0,0},{10,10}, 0.5); // var9 equals a cubic Bezier curve geometry
composed of 10 points from p0 to p1.
```

- When used with 2 points, a float coefficient and a boolean, it computes a cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of 10 points. The last boolean is used to specified if it is the right side.

```
geometry var10 <- curve({0,0},{10,10}, 0.5, false); // var10 equals a cubic Bezier curve
geometry composed of 10 points from p0 to p1 at the left side.
```

See also: [around](#), [circle](#), [cone](#), [link](#), [norm](#), [point](#), [polygone](#), [rectangle](#), [square](#), [triangle](#), [line](#),

cylinder

Possible uses:

- float cylinder float --> geometry
- cylinder (float , float) --> geometry

Result:

A cylinder geometry which radius is equal to the operand.

Comment:

the center of the cylinder is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- cylinder(10,10); // var0 equals a geometry as a circle of radius 10.
```

See also: around, cone, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

Version: 1.9.1

Operators (D to H)

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 spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
    }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considered as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

`box, cone3D, cube, cylinder, hexagon, pyramid, set_z, sphere, teapot,`

Arithmetic operators

`-, /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,`

BDI

add_values, and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or, set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_values,

Casting operators

as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of, to_gaml, to_geojson, to_list, with_size, with_style,

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean, median, palette, rgb, rnd_color, scale, sum, to_hsb,

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect, contains, contains_all, contains_any, contains_key, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

Date-related operators

-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours, minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours, plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,

Dates

Displays

horizontal, stack, vertical,

edge

`edge_between, strahler,`

EDP-related operators

`diff, diff2,`

Files-related operators

`copy_file, crs, csv_file, delete_file, dxf_file, evaluate_sub_model, file_exists, folder, folder_exists, gaml_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,`

GamaMetaType

`type_of,`

Gen*

`add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,`

Graphs-related operators

`add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, as_spatial_graph, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_random_graph, generate_watts_strogatz, girvan_newman_clustering, grid_cells_to_graph, in_degree_of, in_edges_of, k_spanning_tree_clustering, label_propagation_clustering, layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid, load_shortest_paths, main_connected_component, max_flow_between, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_from, rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of, undirected, use_cache, weight_of, with_k_shortest_path_algorithm, with_shortest_path_algorithm, with_weights,`

Grid-related operators

`as_4_grid, as_grid, as_hexagonal_grid, cell_at, cells_in, cells_overlapping, field, grid_at, neighbors_of, path_between, points_in, values_in,`

ImageOperators

`*, antialiased, blend, blurred, brighter, clipped_with, darker, grayscale, horizontal_flip, image, matrix, rotated_by, sharpened, snapshot, tinted_with, vertical_flip, with_height, with_size, with_width,`

Iterator operators

accumulate, all_match, as_map, collect, count, create_map, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, none_matches, one_matches, product_of, sort_by, sum_of, variance_of, where, where, where, with_max_of, with_min_of,

List-related operators

all_indexes_of, copy_between, index_of, last_index_of,

Logical operators

; !, ?, add_3Dmodel, add_geometry, add_icon, and, or, xor,

Map comparaison operators

fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,

Map-related operators

as_map, create_map, index_of, last_index_of,

Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

multicriteria operators

electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM, weighted_means_DM,

Path-related operators

agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, max_flow_between, path_between, path_to, paths_between, use_cache,

Pedestrian

generate_pedestrian_network,

Points-related operators

-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid, closest_points_with, farthest_point_to, grid_at, norm, points_along, points_at, points_on,

Random operators

binomial, flip, gamma_density, gamma_rnd, gamma_trunc_rnd, gauss, generate_terrain, lognormal_density, lognormal_rnd, lognormal_trunc_rnd, poisson, rnd, rnd_choice, sample, shuffle, skew_gauss, truncated_gauss, weibull_density, weibull_rnd, weibull_trunc_rnd,

ReverseOperators

restore_simulation, restore_simulation_from_file, save_simulation, serialize, serialize_agent,

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, elliptical_arc, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

Spatial operators

- , *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, angle_between, any_location_in, arc, around, as_4_grid, as_driving_graph, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covering, covers, cross, crosses, crossing, crs, CRS_transform, cube, curve, cylinder, direction_between, disjoint_from, distance_between, distance_to, ellipse, elliptical_arc, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, inverse_rotation, k_nearest_neighbors, line, link, masked_by, moran, neighbors_at, neighbors_of, normalized_rotation, overlapping, overlaps, partially_overlapping, partially_overlaps, path_between, path_to, plan, points_along, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rotated_by, rotation_composition, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, touches, touching, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, at_distance, closest_to, covering, crossing, farthest_to, inside, neighbors_at, neighbors_of, overlapping, partially_overlapping, touching,

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

Spatial statistical operators

hierarchical_clustering, k_nearest_neighbors, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, inverse_rotation, normalized_rotation, rotated_by, rotation_composition, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum, build, chi_square, chi_square_complemented, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis, log_gamma, max, mean, mean_deviation, median, min, moment, moran, morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, t_test, variance,

Strings-related operators

+, <, <=, >, >=, at, capitalize, char, contains, contains_all, contains_any, copy_between, date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, regex_matches, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

SubModel

load_sub_model,

System

., choose, command, copy, copy_from_clipboard, copy_to_clipboard, copy_to_clipboard, dead, enter, eval_gaml, every, is_error, is_reachable, is_warning, play_sound, user_confirm, user_input_dialog, wizard, wizard_page,

Time-related operators

date, string,

Types-related operators

action, agent, attributes, BDIPPlan, bool, container, conversation, directory, emotion, file, float, gamm_type, gen_population_generator, gen_range, geometry, graph, int, kml, list, map, matrix, mental_state, message, Norm, pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, species, topology, unknown,

User control operators

choose, enter, user_confirm, user_input_dialog, wizard, wizard_page,

Operators

darker

Possible uses:

- `darker` (`image`) --> `image`

Result:

Used to return an image 10% darker. This operation can be applied multiple times in a row if greater than 10% changes in brightness are desired.

date

Possible uses:

- `string date string` --> `date`
- `date (string, string)` --> `date`
- `date (string, string, string)` --> `date`

Result:

converts a string to a date following a custom pattern. The pattern can use "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will parse the date using one of the ISO date & time formats (similar to `date(...)` in that case). The pattern can also follow the pattern definition found here, which gives much more control over what will be parsed: <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns>. Different patterns are available by default as constant: `#iso_local`, `#iso_simple`, `#iso_offset`, `#iso_zoned` and `#custom`, which can be changed in the preferences

Special cases:

- In addition to the date and pattern string operands, a specific locale (e.g. 'fr', 'en'...) can be added.

```
date d <- date("1999-january-30", 'yyyy-MMM-d', 'en');
```

Examples:

```
date den <- date("1999-12-30", 'yyyy-MM-dd');
```

dbSCAN

Possible uses:

- `dbSCAN` (`list`, `float`, `int`) --> `list<list>`

Result:

returns the list of clusters (list of instance indices) computed with the dbSCAN (density-based spatial clustering of applications with noise) algorithm from the first operand data according to the maximum radius of the neighborhood to be considered (eps) and the minimum number of points needed for a cluster (minPts). Usage: `dbSCAN(data,eps,minPoints)`

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
list<list> var0 <- dbscan ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],10,2); // var0 equals [[0,1,2,3]]
```

dead

Possible uses:

- `dead` (`agent`) ---> `bool`

Result:

true if the agent is dead (or null), false otherwise.

Examples:

```
bool var0 <- dead(agent_A); // var0 equals true or false
```

degree_of

Possible uses:

- `graph` `degree_of` `unknown` ---> `int`
- `degree_of` (`graph` , `unknown`) ---> `int`

Result:

returns the degree (in+out) of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap degree_of (node(3)); // var1 equals 3
```

See also: [in_degree_of](#), [out_degree_of](#),

delete_file

Possible uses:

- `delete_file` (`string`) ---> `bool`

Result:

delete a file or a folder

Examples:

```
bool delete_file_ok <- delete_file(["..../includes/my_folder"]);
```

det

Same signification as [determinant](#)

determinant

Possible uses:

- `float determinant(matrix) -> float`

Result:

The determinant of the given matrix

Examples:

```
float var0 <- determinant(matrix([[1,2],[3,4]])); // var0 equals -2
```

diff

Possible uses:

- `float diff float -> float`
- `diff (float, float) -> float`

Result:

A placeholder function for expressing equations

diff2

Possible uses:

- `float diff2 float -> float`
- `diff2 (float, float) -> float`

Result:

A placeholder function for expressing equations

directed

Possible uses:

- `directed (graph) -> graph`

Result:

the operand graph becomes a directed graph.

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph.

See also: [undirected](#),

direction_between

Possible uses:

- `topology direction_between container<unknown,geometry> --> float`
- `direction_between (topology , container<unknown,geometry>) --> float`

Result:

A direction (in degree) between a list of two geometries (geometries, agents, points) considering a topology.

Examples:

```
float var0 <- my_topology direction_between [ag1, ag2]; // var0 equals the direction between ag1 and ag2 considering the topology my_topology
```

See also: [towards](#), [direction_to](#), [distance_to](#), [distance_between](#), [path_between](#), [path_to](#),

direction_to

Same signification as [towards](#)

directory

Possible uses:

- `directory (any) --> directory`

Result:

casts the operand in a directory object.

disjoint_from

Possible uses:

- `geometry disjoint_from geometry --> bool`
- `disjoint_from (geometry , 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
```

See also: [intersects](#), [crosses](#), [overlaps](#), [partially_overlaps](#), [touches](#),

distance_between

Possible uses:

- `topology distance_between container<unknown, geometry> --> float`
- `distance_between (topology , container<unknown, geometry>) --> float`

Result:

A distance between a list of geometries (geometries, agents, points) considering a topology.

Examples:

```
float var0 <- my_topology distance_between [ag1, ag2, ag3]; // var0 equals the distance between ag1, ag2 and ag3 considering the topology my_topology
```

See also: [towards](#), [direction_to](#), [distance_to](#), [direction_between](#), [path_between](#), [path_to](#),

distance_to

Possible uses:

- `geometry distance_to geometry --> float`
- `distance_to (geometry , geometry)--> float`
- `point distance_to point --> float`
- `distance_to (point , 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](#),

distinct

Same signification as [remove_duplicates](#)

distribution_of

Possible uses:

- `distribution_of (container) --> map`
- `container distribution_of int --> map`
- `distribution_of (container , int) --> map`
- `distribution_of (container , int , float , float) --> map`

Result:

Discretize a list of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nbins) or (list,nbbins,valmin,valmax)

Examples:

```
map var0 <- distribution_of([1,1,2,12.5],10); // var0 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'::[['0.0:2.0'],['2.0:4.0'],['4.0:6.0'],['6.0:8.0'],['8.0:10.0'],['10.0:12.0'],['12.0:14.0'],['14.0:16.0']]}  
map var1 <- distribution_of([1,1,2,12.5]); // var1 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'::[['0.0:2.0'],['2.0:4.0'],['4.0:6.0'],['6.0:8.0'],['8.0:10.0'],['10.0:12.0'],['12.0:14.0'],['14.0:16.0']]}  
map var2 <- distribution_of([1,1,2,12.5]); // var2 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0],'legend'::[['0.0:2.0'],['2.0:4.0'],['4.0:6.0'],['6.0:8.0'],['8.0:10.0'],['10.0:12.0'],['12.0:14.0'],['14.0:16.0']]})
```

See also: [as_map](#),

distribution2d_of

Possible uses:

- `container distribution2d_of container` --> `map`
 - `distribution2d_of (container, container)` --> `map`
 - `distribution2d_of (container, container, int, int)` --> `map`
 - `distribution2d_of (container, container, int, float, float, int, float, float)` --> `map`

Result:

Discretize two lists of values into n bins (computes the bins from a numerical variable into n (default 10) bins. Returns a distribution map with the values (values key), the interval legends (legend key), the distribution parameters (params keys, for cumulative charts). Parameters can be (list), (list, nb bins) or (list, nb bins, val min, val max)

Examples:

```
map var0 <- distribution2d_of([1,1,2,12.5],10); // var0 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0], 'legend'::[['0.0:2.0'], ['2.0:4.0'], ['4.0:6.0'], ['6.0:8.0'], ['8.0:10.0'], ['10.0:12.0'], ['12.0:14.0'], ['14.0:16.0'], ['16.0:18.0'], ['18.0:20.0']]}  
map var1 <- distribution2d_of([1,1,2,12.5]); // var1 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0], 'legend'::[['0.0:2.0'], ['2.0:4.0'], ['4.0:6.0'], ['6.0:8.0'], ['8.0:10.0'], ['10.0:12.0'], ['12.0:14.0'], ['14.0:16.0'], ['16.0:18.0'], ['18.0:20.0']]}  
map var2 <- distribution2d_of([1,1,2,12.5],10); // var2 equals  
map(['values'::[2,1,0,0,0,0,1,0,0,0], 'legend'::[['0.0:2.0'], ['2.0:4.0'], ['4.0:6.0'], ['6.0:8.0'], ['8.0:10.0'], ['10.0:12.0'], ['12.0:14.0'], ['14.0:16.0'], ['16.0:18.0'], ['18.0:20.0']]}
```

See also: [as_map](#),

div

Possible uses:

- `int div int -> int`
 - `div (int , int) -> int`
 - `float div float -> int`
 - `div (float , float) -> int`
 - `int div float -> int`
 - `div (int , float) -> int`
 - `float div int -> int`

- `div` (`float`, `int`) --> `int`

Result:

Returns the truncation of the division of the left-hand operand by the right-hand operand.

Special cases:

- if the right-hand operand is equal to zero, raises an exception.

Examples:

```
int var0 <- 40 div 3; // var0 equals 13
int var1 <- 40.1 div 4.5; // var1 equals 8
int var2 <- 40 div 4.1; // var2 equals 9
int var3 <- 40.5 div 3; // var3 equals 13
```

See also: [mod](#),

dnorm

Same signification as [normal_density](#)

dtw

Possible uses:

- `list dtw list` --> `float`
- `dtw(list, list)` --> `float`
- `dtw(list, list, int)` --> `float`

Result:

returns the dynamic time warping between the two series of values (step pattern used: symmetric1) returns the dynamic time warping between the two series of values (step pattern used: symmetric1) with Sakoe-Chiba band (radius: the window width of Sakoe-Chiba band)

Examples:

```
float var0 <- dtw([32.0,5.0,1.0,3.0],[1.0,10.0,5.0,1.0]); // var0 equals 38.0
float var1 <- dtw([10.0,5.0,1.0, 3.0],[1.0,10.0,5.0,1.0], 2); // var1 equals 11.0
```

durbin_watson

Possible uses:

- `durbin_watson(container)` --> `float`

Result:

Durbin-Watson computation

Examples:

```
float var0 <- durbin_watson([13,2,1,4,1,2]) with_precision(4); // var0 equals 0.7231
```

dx_f_file

Possible uses:

- `dxf_file` (string) ---> file
- string `dxf_file` float ---> file
- `dxf_file` (string, float) ---> file

Result:

Constructs a file of type dxf. Allowed extensions are limited to dxf

Special cases:

- `dxf_file`(string): This file constructor allows to read a dxf (.dxf) file

```
file f <- dxf_file("file.dxf");
```

- `dxf_file`(string,float): This file constructor allows to read a dxf (.dxf) file and specify the unit (meter by default)

```
file f <- dxf_file("file.dxf",#m);
```

See also: [is_dxf](#),

edge

Possible uses:

- `edge` (pair)---> unknown
- `edge` (unknown)---> unknown
- pair `edge` float ---> unknown
- `edge` (pair, float)---> unknown
- unknown `edge` float ---> unknown
- `edge` (unknown, float)---> unknown
- unknown `edge` unknown ---> unknown
- `edge` (unknown, unknown)---> unknown
- pair `edge` int ---> unknown
- `edge` (pair, int)---> unknown
- unknown `edge` int ---> unknown
- `edge` (unknown, int)---> unknown
- `edge` (pair, unknown, float)---> unknown
- `edge` (unknown, unknown, int)---> unknown
- `edge` (unknown, unknown, unknown)---> unknown
- `edge` (pair, unknown, int)---> unknown
- `edge` (unknown, unknown, float)---> unknown
- `edge` (unknown, unknown, unknown, float)---> unknown
- `edge` (unknown, unknown, unknown, int)---> unknown

Result:

Allows to create a wrapper (of type unknown) that wraps two objects and indicates they should be considered as the source and the target of a new edge of a graph. The third (omissible) parameter indicates which weight this edge should have in the graph

Comment:

Useful only in graph-related operations (addition, removal of edges, creation of graphs)

edge_between**Possible uses:**

- `graph edge_between pair --> unknown`
- `edge_between (graph , pair) --> unknown`

Result:

returns the edge linking two nodes

Examples:

```
unknown var0 <- graphFromMap edge_between node1::node2; // var0 equals edge1
```

See also: [out_edges_of](#), [in_edges_of](#),

edge_betweenness**Possible uses:**

- `edge_betweenness (graph) --> map`

Result:

returns a map containing for each edge (key), its betweenness centrality (value): number of shortest paths passing through each edge

Examples:

```
graph graphEpidemio <- graph([]);  
map var1 <- edge_betweenness(graphEpidemio); // var1 equals the edge betweenness index of the graph
```

edges**Possible uses:**

- `edges (container) --> container`

Result:

Allows to create a wrapper (of type list) that wraps a list of objects and indicates they should be considered as edges of a graph

eigenvalues

Possible uses:

- `eigenvalues` (`matrix`) ---> `list<float>`

Result:

The list of the eigen values of the given matrix

Examples:

```
list<float> var0 <- eigenvalues(matrix([[5,-3],[6,-4]])); // var0 equals [2.000000000000004, -0.9999999999999998]
```

electre_DM

Possible uses:

- `electre_DM` (`list<list>`, `list<map<string, unknown>>`, `float`) ---> `int`

Result:

The index of the best candidate according to a method based on the ELECTRE methods. The principle of the ELECTRE methods is to compare the possible candidates by pair. These methods analyses the possible outranking relation existing between two candidates. A candidate outranks another if this one is at least as good as the other one. The ELECTRE methods are based on two concepts: the concordance and the discordance. The concordance characterizes the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterizes the fact that, for an outranking relation to be validated, none of the criteria in the minority should oppose too strongly this assertion. These two conditions must be true for validating the outranking assertion. More information about the ELECTRE methods can be found in Figueira, J., Mousseau, V., Roy, B.: ELECTRE Methods. In: Figueira, J., Greco, S., and Ehrgott, M., (Eds.), *Multiple Criteria Decision Analysis: State of the Art Surveys*, Springer, New York, 133–162 (2005). The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fives elements: a name, a weight, a preference value (p), an indifference value (q) and a veto value (v). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant. The veto value represents the threshold from which the difference between two criterion values disqualifies the candidate that obtained the smaller value; the last operand is the fuzzy cut.

Special cases:

- returns -1 if the list of candidates is nil or empty

Examples:

```
int var0 <- electre_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [{"name": "utility", "weight" :: 2.0, "p"::0.5, "q":0.0, "s":1.0, "maximize" :: true}, {"name": "price", "weight" :: 1.0, "p":0.5, "q":0.0, "s":1.0, "maximize" :: false}], 0.7); // var0 equals 0
```

See also: [weighted_means_DM](#), [promethee_DM](#), [evidence_theory_DM](#),

ellipse

Possible uses:

- `float ellipse float` ---> `geometry`
- `ellipse (float , float)` ---> `geometry`

Result:

An ellipse geometry which x-radius is equal to the first operand and y-radius is equal to the second operand

Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if both operands are lower or equal to 0, a line if only one is.

Examples:

```
geometry var0 <- ellipse(10, 10); // var0 equals a geometry as an ellipse of width 10 and height 10.
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [circle](#), [squircle](#), [triangle](#),

elliptical_arc**Possible uses:**

- `elliptical_arc` (`point`, `point`, `float`, `int`) --> `geometry`

Result:

An elliptical arc from the first operand (point) to the second operand (point), which radius is equal to the third operand, and a int giving the number of points to use as a last operand

Examples:

```
geometry var0 <- elliptical_arc({0,0},{10,10},5.0, 20); // var0 equals a geometry from {0,0} to {10,10} considering a radius of 5.0 built using 20 points
```

See also: [arc](#), [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [super_ellipse](#), [rectangle](#), [square](#), [circle](#), [ellipse](#), [triangle](#),

emotion**Possible uses:**

- `emotion` (`any`) --> `emotion`

Result:

casts the operand in a emotion object.

empty**Possible uses:**

- `empty` (`string`) --> `bool`
- `empty` (`container<KeyType,ValueType>`) --> `bool`

Result:

true if the operand is empty, false otherwise.

Comment:

the empty operator behavior depends on the nature of the operand

Special cases:

- if it is a map, empty returns true if the map contains no key-value mappings, and false otherwise
- if it is a file, empty returns true if the content of the file (that is also a container) is empty, and false otherwise
- if it is a population, empty returns true if there is no agent in the population, and false otherwise
- if it is a graph, empty returns true if it contains no vertex and no edge, and false otherwise
- if it is a matrix of int, float or object, it will return true if all elements are respectively 0, 0.0 or null, and false otherwise
- if it is a matrix of geometry, it will return true if the matrix contains no cell, and false otherwise
- if it is a string, empty returns true if the string does not contain any character, and false otherwise

```
bool var0 <- empty ('abcd'); // var0 equals false
```

- if it is a list, empty returns true if there is no element in the list, and false otherwise

```
bool var1 <- empty([]); // var1 equals true
```

enlarged_by

Same signification as [+](#)

enter**Possible uses:**

- string **enter** unknown ---> unknown
- **enter** (string , unknown)---> unknown
- string **enter** int ---> unknown
- **enter** (string , int)---> unknown
- string **enter** any GAML type ---> unknown
- **enter** (string , any GAML type)---> unknown
- string **enter** bool ---> unknown
- **enter** (string , bool)---> unknown
- string **enter** string ---> unknown
- **enter** (string , string)---> unknown
- string **enter** float ---> unknown
- **enter** (string , float)---> unknown
- **enter** (string, any GAML type, unknown)---> unknown
- **enter** (string, float, float, float)---> unknown
- **enter** (string, int, int, int)---> unknown
- **enter** (string, float, float, float, float)---> unknown
- **enter** (string, int, int, int, int)---> unknown

Result:

Allows the user to enter a string by specifying a title and an initial value

Special cases:

- When the second operand is the boolean type or a boolean value, the GUI is then a switch

```
map<string, unknown> m <- user_input(enter("Title", true));
map<string, unknown> m2 <- user_input(enter("Title", bool));
```

- The GUI is then a slider when an init value, a min (int or float), a max (int or float) (and eventually a step (int or float)) operands.

```
map resMinMax <- user_input([enter("Title", 5, 0)]);
map resMinMax <- user_input([enter("Title", 5, 0, 10)]);
map resMMStepFF <- user_input([enter("Title", 5, 0.1, 10.1, 0.5)]);
```

envelope

Possible uses:

- `envelope` (unknown) ---> geometry

Result:

A 3D geometry that represents the box that surrounds the geometries or the surface described by the arguments. More general than `geometry(arguments).envelope`, as it allows to pass int, double, point, image files, shape files, asc files, or any list combining these arguments, in which case the envelope will be correctly expanded. If an envelope cannot be determined from the arguments, a default one of dimensions (0,100, 0, 100, 0, 100) is returned

Special cases:

- This operator is often used to define the environment of simulation

Examples:

```
file road_shapefile <- file("../includes/roads.shp");
geometry shape <- envelope(road_shapefile);
// shape is the system variable of the environment
geometry var3 <- polygon([{0,0}, {20,0}, {10,10}, {10,0}]); // var3 equals create a polygon to get the envelope
float var4 <- envelope(polygon([{0,0}, {20,0}, {10,10}, {10,0}])).area; // var4 equals 200.0
```

eval_gaml

Possible uses:

- `eval_gaml` (string) ---> unknown

Result:

evaluates the given GAML string.

Examples:

```
unknown var0 <- eval_gaml("2+3"); // var0 equals 5
```

eval_when

Possible uses:

- `eval_when (BDIPlan) ---> bool`

Result:

evaluate the facet when of a given plan

Examples:

```
eval_when(plan1)
```

evaluate_sub_model

Possible uses:

- `agent evaluate_sub_model string ---> unknown`
- `evaluate_sub_model (agent , string)---> unknown`

Result:

Load a submodel

Comment:

loaded submodel

even

Possible uses:

- `even (int) ---> bool`

Result:

Returns true if the operand is even and false if it is odd.

Special cases:

- if the operand is equal to 0, it returns true.
- if the operand is a float, it is truncated before

Examples:

```
bool var0 <- even (3); // var0 equals false
bool var1 <- even(-12); // var1 equals true
```

every

Possible uses:

- `every (int) --> bool`
- `every (any expression) --> bool`
- `float every int --> float`
- `every (float , int) --> float`
- `list every int --> list`
- `every (list , int) --> list`
- `unknown every int --> unknown`
- `every (unknown , int) --> unknown`
- `int every int --> int`
- `every (int , int) --> int`
- `bool every int --> bool`
- `every (bool , int) --> bool`
- `list every any expression --> list<date>`
- `every (list , any expression) --> list<date>`

Result:

true every operand *cycle*, *false* otherwise returns the first float operand every 2nd operand cycle, 0.0 otherwise expects a frequency (expressed in seconds of simulated time) as argument. Will return true every time the current_date matches with this frequency. Retrieves elements from the first argument every *step* (second argument) elements. Raises an error if the step is negative or equal to zero returns the first operand every 2nd operand *cycle*, *nil* otherwise returns the first integer operand every 2nd operand cycle, 0 otherwise returns the first bool operand every 2nd operand * cycle, false otherwise applies a step to an interval of dates defined by 'date1' to 'date2'. Beware that using every with #month or #year will produce odd results, as these pseudo-constants are not constant; only the first value will be used to compute the intervals, so, for instance, if current_date is set to February#month will only represent 28 or 29 days.

Comment:

the value of the every operator depends on the cycle. It can be used to do something every x cycle. the value of the every operator depends on the cycle. It can be used to return a value every x cycle. `1000.0 every(10#cycle)` is strictly equivalent to `every(10#cycle) ? 1000.0 : 0.0`. Used to do something at regular intervals of time. Can be used in conjunction with 'since', 'after', 'before', 'until' or 'between', so that this computation only takes place in the temporal segment defined by these operators. In all cases, the starting_date of the model is used as a reference starting point. the value of the every operator depends on the cycle. It can be used to return a value every x cycle. `object every(10#cycle)` is strictly equivalent to `every(10#cycle) ? object : nil`. the value of the every operator depends on the cycle. It can be used to return a value every x cycle. `1000 every(10#cycle)` is strictly equivalent to `every(10#cycle) ? 1000 : 0`. the value of the every operator depends on the cycle. It can be used to return a value every x cycle. `object every(10#cycle)` is strictly equivalent to `every(10#cycle) ? object : false`.

Examples:

```
if every(2#cycle) {write "the cycle number is even";}
    else {write "the cycle number is odd";}
if (1000.0 every(2#cycle) != 0) {write "this is a value";}
    else {write "this is 0.0";}
reflex when: every(2#days) since date('2000-01-01') { .. }
state a { transition to: b when: every(2#mn);} state b { transition to: a when: every(30#s);} // This oscillatory behavior
will use the starting_date of the model as its starting point in time
if ({2000,2000} every(2#cycle) != nil) {write "this is a point";}
    else {write "this is nil";}
if (1000 every(2#cycle) != 0) {write "this is a value";}
    else {write "this is 0";}
if (true every(2#cycle) != false) {write "this is true";}
    else {write "this is false";}
(date('2000-01-01') to date('2010-01-01')) every (#day) // builds an interval between these two dates which contains all the
```

See also: [since](#), [after](#), [to](#),

every_cycle

Same signification as [every](#)

evidence_theory_DM

Possible uses:

- `list<list> evidence_theory_DM list<map<string, unknown>> --> int`
- `evidence_theory_DM (list<list>, list<map<string, unknown>>) --> int`
- `evidence_theory_DM (list<list>, list<map<string, unknown>>, bool) --> int`

Result:

The index of the best candidate according to a method based on the Evidence theory. This theory, which was proposed by Shafer ([Shafer G \(1976\) A mathematical theory of evidence, Princeton University Press](#)), is based on the work of Dempster ([Dempster A \(1967\) Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, vol. 38, pp. 325–339](#)) on lower and upper probability distributions. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains seven elements: a name, a first threshold s1, a second threshold s2, a value for the assertion "this candidate is the best" at threshold s1 (v1p), a value for the assertion "this candidate is the best" at threshold s2 (v2p), a value for the assertion "this candidate is not the best" at threshold s1 (v1c), a value for the assertion "this candidate is not the best" at threshold s2 (v2c). v1p, v2p, v1c and v2c have to be defined in order that: v1p + v1c <= 1.0; v2p + v2c <= 1.0.; the last operand allows to use a simple version of this multi-criteria decision making method (simple if true)

Special cases:

- if the operator is used with only 2 operands (the candidates and the criteria), the last parameter (use simple method) is set to true
- returns -1 if the list of candidates is nil or empty

Examples:

```
int var0 <- evidence_theory_DM([[1.0, 7.0], [4.0, 2.0], [3.0, 3.0]], [{"name": "utility", "s1": 0.0, "s2": 1.0, "v1p": 0.0, "v2p": 1.0, "v1c": 0.0, "v2c": 0.0, "maximize": true}, {"name": "price", "s1": 0.0, "s2": 1.0, "v1p": 0.0, "v2p": 1.0, "v1c": 0.0, "v2c": 0.0, "maximize": true}]); // var0 equals 0
int var1 <- evidence_theory_DM([[1.0, 7.0], [4.0, 2.0], [3.0, 3.0]], [{"name": "utility", "s1": 0.0, "s2": 1.0, "v1p": 0.0, "v2p": 1.0, "v1c": 0.0, "v2c": 0.0, "maximize": true}, {"name": "price", "s1": 0.0, "s2": 1.0, "v1p": 0.0, "v2p": 1.0, "v1c": 0.0, "v2c": 0.0, "maximize": true}], false); // var1 equals 0
```

See also: [weighted_means_DM](#), [electre_DM](#),

exp

Possible uses:

- `exp (float) --> float`
- `exp (int) --> float`

Result:

Returns Euler's number e raised to the power of the operand.

Special cases:

- the operand is casted to a float before being evaluated.

Examples:

```
float var0 <- exp (0.0); // var0 equals 1.0
```

See also: [ln](#),

fact

Possible uses:

- `fact` (`int`) ---> `float`

Result:

Returns the factorial of the operand.

Special cases:

- if the operand is less than 0, fact returns 0.

Examples:

```
float var0 <- fact(4); // var0 equals 24
```

farthest_point_to

Possible uses:

- `geometry farthest_point_to point` ---> `point`
- `farthest_point_to (geometry, point)` ---> `point`

Result:

the farthest point of the left-operand to the left-point.

Examples:

```
point var0 <- geom farthest_point_to(pt); // var0 equals the farthest point of geom to pt
```

See also: [any_location_in](#), [any_point_in](#), [closest_points_with](#), [points_at](#),

farthest_to

Possible uses:

- `container<unknown, geometry> farthest_to geometry` ---> `geometry`
- `farthest_to (container<unknown, geometry>, geometry)` ---> `geometry`

Result:

An agent or a geometry among the left-operand list of agents, species or meta-population (addition of species), the farthest to the operand (casted as a geometry).

Comment:

the distance is computed in the topology of the calling agent (the agent in which this operator is used), with the distance algorithm specific to the topology.

Examples:

```
geometry var0 <- [ag1, ag2, ag3] closest_to(self); // var0 equals return the farthest agent among ag1, ag2 and ag3 to the agent applying the operator.  
(species1 + species2) closest_to self
```

See also: [neighbors_at](#), [neighbors_of](#), [inside](#), [overlapping](#), [agents_overlapping](#), [agents_inside](#), [agent_closest_to](#), [closest_to](#), [agent_farthest_to](#),

field

Possible uses:

- `int field int --> field`
- `field (int, int) --> field`
- `unknown field float --> field`
- `field (unknown, float) --> field`
- `field (int, int, float) --> field`
- `field (int, int, float, float) --> field`

field_with

Possible uses:

- `point field_with any expression --> field`
- `field_with (point, any expression) --> field`

Result:

creates a field with a size provided by the first operand, and filled by the evaluation of the second operand for each cell

Comment:

Note that both components of the right operand point should be positive, otherwise an exception is raised.

See also: [matrix](#), [as_matrix](#),

file

Possible uses:

- `file (any) --> file`

Result:

casts the operand in a file object.

file_exists

Possible uses:

- `file_exists (string) --> bool`

Result:

Test whether the parameter is the path to an existing file. False if it does not exist or if it is a folder

Examples:

```
string file_name <- ".../includes/buildings.shp";
if file_exists(file_name){
    write "File exists in the computer";
}
```

first

Possible uses:

- `first (string) --> string`
- `first (container<KeyType,ValueType>) --> ValueType`
- `int first container --> list`
- `first (int, 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 string, first returns a string composed of its first character

```
string var0 <- first ('abce'); // var0 equals 'a'
```

- if it is a list, first returns the first element of the list, or nil if the list is empty

```
int var1 <- first ([1, 2, 3]); // var1 equals 1
```

See also: [last](#),

first_of

Same signification as [first](#)

first_with

Possible uses:

- `container first_with any expression ---> unknown`
- `first_with (container, any expression) ---> unknown`

Result:

the first element of the left-hand operand that makes the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, first_with throws an error. If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
int var4 <- [1::2, 3::4, 5::6] first_with (each >= 4); // var4 equals 4
pair var5 <- [1::2, 3::4, 5::6].pairs first_with (each.value >= 4); // var5 equals (3::4)
```

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] first_with (each > 3); // var0 equals 4
unknown var2 <- g2 first_with (length(g2 out_edges_of each) = 0); // var2 equals node9
unknown var3 <- (list(node) first_with (round(node(each).location.x) > 32); // var3 equals node2
```

See also: [group_by](#), [last_with](#), [where](#),

flip

Possible uses:

- `flip (float) ---> bool`

Result:

true or false given the probability represented by the operand

Special cases:

- flip 0 always returns false, flip 1 true

Examples:

```
bool var0 <- flip (0.66666); // var0 equals 2/3 chances to return true.
```

See also: [rnd](#),

float

Possible uses:

- `float` (`any`) ---> `float`

Result:

casts the operand in a float object.

floor

Possible uses:

- `floor` (`float`) ---> `int`

Result:

Maps the operand to the largest previous following integer, i.e. the largest integer not greater than x.

Examples:

```
int var0 <- floor(3); // var0 equals 3
int var1 <- floor(3.5); // var1 equals 3
int var2 <- floor(-4.7); // var2 equals -5
```

See also: [ceil](#), [round](#),

folder

Possible uses:

- `folder` (`string`) ---> `file`

Result:

opens an existing repository

Special cases:

- If the specified string does not refer to an existing repository, an exception is risen.

Examples:

```
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](#),

folder_exists

Possible uses:

- `folder_exists` (`string`) ---> `bool`

- `string folder_exists (list<string>) -> bool`
- `folder_exists (string , list<string>) -> bool`

Result:

Test whether the parameter is the path to an existing folder. False if it doesn't exist or if it is a file Test whether the parameter is the path to an existing folder. False if it doesn't exist or if it is a file

Examples:

```
string file_name <- "../includes/";
if folder_exists(file_name){
    write "Folder exists in the computer";
}
string file_name <- "../includes/";
if folder_exists(file_name){
    write "Folder exists in the computer";
}
```

font

Possible uses:

- `string font int -> font`
- `font (string , int) -> font`
- `font (string , int , int) -> font`

Result:

Creates a new font, by specifying its name (either a font face name like 'Lucida Grande Bold' or 'Helvetica', or a logical name like 'Dialog', 'SansSerif', 'Serif', etc.), a size in points and a style, either #bold, #italic or #plain or a combination (addition) of them.

Examples:

```
font var0 <- font ('Helvetica Neue',12, #bold + #italic); // var0 equals a bold and italic face of the Helvetica Neue family
```

frequency_of

Possible uses:

- `container frequency_of any expression -> map`
- `frequency_of (container , any expression) -> map`

Result:

Returns a map with keys equal to the application of the right-hand argument (like collect) and values equal to the frequency of this key (i.e. how many times it has been obtained)

Examples:

```
map var0 <- [1, 2, 3, 3, 4, 4, 5, 3, 3, 4] frequency_of each; // var0 equals map([1::1,2::1,3::4,4::3,5::1])
```

from

Same signification as [since](#)

fuzzy_choquet_DM

Possible uses:

- `fuzzy_choquet_DM (list<list>, list<string>, map) --> int`

Result:

The index of the candidate that maximizes the Fuzzy Choquet Integral value. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion (list of string); the third operand the weights of each sub-set of criteria (map with list for key and float for value)

Special cases:

- returns -1 if the list of candidates is nil or empty

Examples:

```
int var0 <- fuzzy_choquet_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], ["utility", "price",  
"size"],[[["utility"]]:=0.5,[["size"]]:=0.1,[["price"]]:=0.4,[["utility", "price"]]:=0.55]); // var0 equals 0
```

See also: [promethee_DM](#), [electre_DM](#), [evidence_theory_DM](#),

fuzzy_kappa

Possible uses:

- `fuzzy_kappa (list<agent>, list<unknown>, list<unknown>, list<float>, list<unknown>, matrix<float>, float) --> float`
- `fuzzy_kappa (list<agent>, list<unknown>, list<unknown>, list<float>, list<unknown>, matrix<float>, float, list<unknown>) --> float`

Result:

fuzzy kappa indicator for 2 map comparisons: `fuzzy_kappa(agents_list,list_vals1,list_vals2, output_similarity_per_agents, categories, fuzzy_categories_matrix, fuzzy_distance, weights)`. Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21 fuzzy kappa indicator for 2 map comparisons: `fuzzy_kappa(agents_list,list_vals1,list_vals2, output_similarity_per_agents, categories, fuzzy_categories_matrix, fuzzy_distance)`. Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21

Examples:

```
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
similarity_per_agents,[cat1,cat2,cat3],[[1,0,0],[0,1,0],[0,0,1]], 2, [1.0,3.0,2.0,2.0,4.0])  
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
similarity_per_agents,[cat1,cat2,cat3],[[1,0,0],[0,1,0],[0,0,1]], 2)
```

fuzzy_kappa_sim

Possible uses:

- **fuzzy_kappa_sim** (`list<agent>`, `list<unknown>`, `list<unknown>`, `list<unknown>`, `list<float>`, `list<unknown>`, `matrix<float>`, `float`) ...
 `> float`
 - **fuzzy_kappa_sim** (`list<agent>`, `list<unknown>`, `list<unknown>`, `list<unknown>`, `list<float>`, `list<unknown>`, `matrix<float>`, `float`,
`list<unknown>`) ...
 `>> float`

Result:

fuzzy kappa simulation indicator for 2 map comparisons: `fuzzy_kappa_sim(agents_list,list_vals1,list_vals2,`
`output_similarity_per_agents,fuzzy_transitions_matrix,fuzzy_distance,weights)`. Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800, fuzzy kappa simulation indicator for 2 map comparisons: `fuzzy_kappa_sim(agents_list,list_vals1,list_vals2,`
`output_similarity_per_agents,fuzzy_transitions_matrix,fuzzy_distance)`. Reference: Jasper van Vliet, Alex Hagen-Zanker, Jelle Hurkens, Hedwig van Delden, A fuzzy set approach to assess the predictive accuracy of land use simulations, Ecological Modelling, 24 July 2013, Pages 32-42, ISSN 0304-3800,

Examples:

```
fuzzy_kappa_sim([ag1, ag2, ag3, ag4, ag5], [cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
similarity_per_agents,[cat1,cat2,cat3],[[1,0,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0,0],[0,0,1,0,0,0,0,0,0,0],[0,0,0,1,0,0,0,0,0,0],[0,0,0,0,1,0,0,0,0,0],[0,  
2,[1.0,3.0,2.0,2.0,4.0]])  
fuzzy_kappa_sim([ag1, ag2, ag3, ag4, ag5], [cat1,cat1,cat2,cat3,cat2],[cat2,cat1,cat2,cat1,cat2],  
similarity_per_agents,[cat1,cat2,cat3],[[1,0,0,0,0,0,0,0,0,0],[0,1,0,0,0,0,0,0,0,0],[0,0,1,0,0,0,0,0,0,0],[0,0,0,1,0,0,0,0,0,0],[0,0,0,0,1,0,0,0,0,0],[0,  
2])
```

gaml_file

Possible uses:

- `gaml_file` (`string`) -> `file`
 - `gaml_file` (`string, string, string`) -> `file`

Result:

Constructs a file of type `gaml`. Allowed extensions are limited to `gaml`, `experiment`

Special cases:

- `qaml_file(string)`: This file constructor allows to read a qaml file (.qaml).

```
file f <- gaml_file("file.gaml");
```

- `qaml file(string,string,string)`: This file constructor allows to compile a qaml file and run an experiment.

```
file f <- qaml file("file.qaml", "my experiment", "my model")
```

See also: [is_gaml](#),

gaml_type

Possible uses:

- `gaml_type` (`any`) ---> `gaml_type`

Result:

casts the operand in a `gaml_type` object.

gamma

Possible uses:

- `gamma` (`float`) ---> `float`

Result:

Returns the value of the Gamma function at x.

Examples:

```
float var0 <- gamma(5); // var0 equals 24.0
```

gamma_density

Possible uses:

- `gamma_density` (`float`, `float`, `float`) ---> `float`

Result:

`gamma_density(x,shape,scale)` returns the probability density function (PDF) at the specified point x of the Gamma distribution with the given shape and scale.

Examples:

```
float var0 <- gamma_density(1,9,0.5); // var0 equals 0.731
```

See also: [binomial](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#), [weibull_density](#), [lognormal_density](#),

gamma_distribution

Possible uses:

- `gamma_distribution` (`float`, `float`, `float`) ---> `float`

Result:

Returns the integral from zero to x of the gamma probability density function.

Comment:

`incomplete_gamma(a,x)` is equal to `pgamma(a,1,x)`.

Examples:

```
float var0 <- gamma_distribution(2,3,0.9) with_precision(3); // var0 equals 0.269
```

gamma_distribution_complemented

Possible uses:

- `gamma_distribution_complemented` (`float`, `float`, `float`) \rightarrow `float`

Result:

Returns the integral from x to infinity of the gamma probability density function.

Examples:

```
float var0 <- gamma_distribution_complemented(2,3,0.9) with_precision(3); // var0 equals 0.731
```

gamma_index

Possible uses:

- `gamma_index` (`graph`) \rightarrow `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: $\text{gamma} = e/(3 \cdot (v - 2))$ - 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](#),

gamma_rnd

Possible uses:

- `float gamma_rnd float` \rightarrow `float`
- `gamma_rnd` (`float`, `float`) \rightarrow `float`

Result:

returns a random value from a gamma distribution with specified values of the shape and scale parameters

Examples:

```
float var0 <- gamma_rnd(9,0.5); // var0 equals 0.731
```

See also: [binomial](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#), [gamma_trunc_rnd](#),

gamma_trunc_rnd

Possible uses:

- `gamma_trunc_rnd` (`float`, `float`, `float`, `float`) \rightarrow `float`
- `gamma_trunc_rnd` (`float`, `float`, `float`, `bool`) \rightarrow `float`

Result:

returns a random value from a truncated gamma distribution (in a range or given only one boundary) with specified values of the shape and scale parameters.

Special cases:

- when 2 float operands are specified, they are taken as minimum and maximum values for the result

```
gamma_trunc_rnd(2,3,0,5)
```

- when 1 float and a boolean (isMax) operands are specified, the float value represents the single boundary (max if the boolean is true, min otherwise),

```
gamma_trunc_rnd(2,3,5,true)
```

See also: [gamma_rnd](#), [weibull_trunc_rnd](#), [lognormal_trunc_rnd](#), [truncated_gauss](#),

gauss

Possible uses:

- `gauss` (`point`) \rightarrow `float`
- `float` `gauss` `float` \rightarrow `float`
- `gauss` (`float`, `float`) \rightarrow `float`

Result:

A value from a normally distributed random variable with expected value (mean as first operand) and variance (standardDeviation as second operand). The probability density function of such a variable is a Gaussian. The operator can be used with an operand of type `point` {mean,standardDeviation}.

Special cases:

- when standardDeviation value is 0.0, it always returns the mean value
- when the operand is a point, it is read as {mean, standardDeviation}

Examples:

```
float var0 <- gauss(0,0.3); // var0 equals 0.22354
float var1 <- gauss({0,0.3}); // var1 equals 0.22354
```

See also: [binomial](#), [gamma_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#),

gauss_rnd

Same signification as [gauss](#)

gen_population_generator

Possible uses:

- `gen_population_generator` (any) ---> `gen_population_generator`

Result:

casts the operand in a `gen_population_generator` object.

gen_range

Possible uses:

- `gen_range` (any) ---> `gen_range`

Result:

casts the operand in a `gen_range` object.

generate_barabasi_albert

Possible uses:

- `generate_barabasi_albert` (`container`, `int`, `int`, `bool`) ---> `graph`
- `generate_barabasi_albert` (`int`, `int`, `int`, `bool`) ---> `graph`
- `generate_barabasi_albert` (`int`, `int`, `int`, `bool`, `species`) ---> `graph`
- `generate_barabasi_albert` (`int`, `int`, `int`, `bool`, `species`, `species`) ---> `graph`

Result:

returns a random scale-free network (following Barabasi-Albert (BA) model). returns a random scale-free network (following Barabasi-Albert (BA) model). returns a random scale-free network (following Barabasi-Albert (BA) model).

Comment:

The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:

Special cases:

- "nbInitNodes": number of initial nodes; "nbEdgesAdded": number of edges of each new node added during the network growth; "nbNodes": final number of nodes; "directed": is the graph directed or not; "node_species": the species of vertices; "edges_species": the species of edges

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
    60,  
    1,  
    100,  
    true,  
    myVertexSpecies);
```

- "nbInitNodes": number of initial nodes; "nbEdgesAdded": number of edges of each new node added during the network growth; "nbNodes": final number of nodes; "directed": is the graph directed or not; "node_species": the species of vertices; "edges_species": the species of edges

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
    60,  
    1,  
    100,  
    true,  
    myVertexSpecies,  
    myEdgeSpecies);
```

- "nbInitNodes": number of initial nodes; "nodes": list of existing nodes to connect (agents or geometries); "nbEdgesAdded": number of edges of each new node added during the network growth; "directed": is the graph directed or not;

```
graph myGraph <- generate_watts_strogatz(people, 10,1,false);
```

- "nbInitNodes": number of initial nodes; "nbEdgesAdded": number of edges of each new node added during the network growth; "nbNodes": final number of nodes; "directed": is the graph directed or not;

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
    60,  
    1,  
    100,  
    true);
```

See also: [generate_watts_strogatz](#),

generate_complete_graph

Possible uses:

- `int generate_complete_graph bool --> graph`
- `generate_complete_graph (int, bool) --> graph`
- `bool generate_complete_graph list --> graph`
- `generate_complete_graph (bool, list) --> graph`
- `generate_complete_graph (int, bool, species) --> graph`
- `generate_complete_graph (bool, list, species) --> graph`
- `generate_complete_graph (int, bool, species, species) --> graph`

Result:

returns a fully connected graph. returns a fully connected graph.

Special cases:

- `nbNodes`: number of nodes to create; `directed`: is the graph directed or not; `node_species`: the species of nodes

```
graph myGraph <- generate_complete_graph(  
    100,  
    true,  
    node_species);
```

- "directed": is the graph has to be directed or not;"nodes": the list of existing nodes; "edges_species": the species of edges

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(  
true,  
nodes,  
edge_species);
```

- `nbNodes`: number of nodes to create; `directed`: is the graph directed or not; `node_species`: the species of nodes; `edges_species`: the species of edges

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(  
100,  
true,  
node_species,  
edge_species);
```

- `nbNodes`: number of nodes to create; `directed`: is the graph directed or not

```
graph myGraph <- generate_complete_graph(  
    100,  
    true);
```

- "directed": is the graph has to be directed or not;"nodes": the list of existing nodes

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(  
    true,  
    nodes);
```

See also: [generate_barabasi_albert](#), [generate_watts_strogatz](#),

generate_pedestrian_network

Possible uses:

- `generate_pedestrian_network` (`list<container<unknown,geometry>>`, `container<unknown,geometry>`, `bool`, `bool`, `float`, `float`, `bool`,
`float`, `float`, `float`, `float`) \rightarrow `list<geometry>`
- `generate_pedestrian_network` (`list<container<unknown,geometry>>`, `container<unknown,geometry>`, `container<unknown,geometry>`,
`bool`, `bool`, `float`, `float`, `bool`, `float`, `float`, `float`, `float`) \rightarrow `list<geometry>`
- `generate_pedestrian_network` (`list<container<unknown,geometry>>`, `container<unknown,geometry>`, `bool`, `bool`, `float`, `float`, `bool`,
`float`, `float`, `float`, `float`, `float`) \rightarrow `list<geometry>`
- `generate_pedestrian_network` (`list<container<unknown,geometry>>`, `container<unknown,geometry>`, `container<unknown,geometry>`,
`bool`, `bool`, `float`, `float`, `bool`, `float`, `float`, `float`, `float`) \rightarrow `list<geometry>`

Result:

The method allows to build a network of corridors to be used by pedestrian while traveling around a space made of obstacles and other users. It makes it possible to avoid collision with other agents (e.g. buildings) including other pedestrians and in the same time managing a path to a destination in a complex environment (e.g. a city). The method is highly customizable, with many parameters listed as below: <p>

1. obstacles : a list containing the lists of geometries or agents that are obstacles for pedestrians (e.g. walls, cars).
2. bounds : a list of geometries that represent the spatial boundary of the network (i.e. the enclosing space of the network).
3. open : a boolean expression that will add nodes in the network within open areas. More precisely, new invisible points are added to improve triangulation in areas with very few obstacles.
4. randomDist : a boolean expression, related to the previous 'open' parameter, that allows to switch between a random (true) spatial distribution or a distribution (false) that build upon a equidistant repartition of points all around the area.
5. open area : a float in meters representing the minimum distance for an area to be considered as an open area (i.e. euclidian distance between centroid and farest obstacle)
6. density point : a float representing the density of points per meter within open areas.
7. clean network : a boolean expression that allows to enhance the network (true) or living as it is generated (false). Enhancement includes filling very small gaps between edges and nodes.
8. cliping : tolerance for the cliping in triangulation (float; distance) - see skeletonize operator
9. tolerance : tolerance for the triangulation (float)
10. min dist obstacle : minimal distance to obstacles to keep a path (float; if 0.0, no filtering)
11. simplification : simplification distance for the final geometries
12. square size : size of squares for decomposition (optimization)

Special cases:

- The method allows to build a network of corridors to be used by pedestrian while traveling around a space made of obstacles and other users. It makes it possible to avoid collision with other agents (e.g. buildings) including other pedestrians and in the same time managing a path to a destination in a complex environment (e.g. a city). The method is highly customizable, with many parameters listed as below: <p>
 - i. obstacles : a list containing the lists of geometries or agents that are obstacles for pedestrians (e.g. walls, cars).
 - ii. bounds : a list of geometries that represent the spatial boundary of the network (i.e. the enclosing space of the network).
 - iii. regular network : allows to combine the generated network with a simplified car user oriented network. More specifically, the network generated will combine enhance pedestrian oriented generated network with the given network: The property of the latter does not allow pedestrian to avoid collision (1D) when using its edges (while moving in 2D space and avoiding collision in the former).
 - iv. open : a boolean expression that will add nodes in the network within open areas. More precisely, new invisible points are added to improve triangulation in areas with very few obstacles.
 - v. randomDist : a boolean expression, related to the previous 'open' parameter, that allows to switch between a random (true) spatial distribution or a distribution (false) that build upon a equidistant repartition of points all around the area.
 - vi. open area : a float in meters representing the minimum distance for an area to be considered as an open area (i.e. euclidian distance between centroid and farest obstacle)
 - vii. density point : a float representing the density of points per meter within open areas.
 - viii. clean network : a boolean expression that allows to enhance the network (true) or living as it is generated (false). Enhancement includes filling very small gaps between edges and nodes.
 - ix. cliping : tolerance for the cliping in triangulation (float; distance) - see skeletonize operator
 - x. tolerance : tolerance for the triangulation (float)
 - xi. min dist obstacle : minimal distance to obstacles to keep a path (float; if 0.0, no filtering)
- The method allows to build a network of corridors to be used by pedestrian while traveling around a space made of obstacles and other users. It makes it possible to avoid collision with other agents (e.g. buildings) including other pedestrians and in the same time managing a path to a destination in a complex environment (e.g. a city). The method is highly customizable, with many parameters listed as below: <p>
 - i. obstacles : a list containing the lists of geometries or agents that are obstacles for pedestrians (e.g. walls, cars).
 - ii. bounds : a list of geometries that represent the spatial boundary of the network (i.e. the enclosing space of the network).
 - iii. regular network : allows to combine the generated network with a simplified car user oriented network. More specifically, the network generated will combine enhance pedestrian oriented generated network with the given network: The property of the latter does not allow

pedestrian to avoid collision (1D) when using its edges (while moving in 2D space and avoiding collision in the former).

- iv. open : a boolean expression that will add nodes in the network within open areas. More precisely, new invisible points are added to improve triangulation in areas with very few obstacles.
 - v. randomDist : a boolean expression, related to the previous 'open' parameter, that allows to switch between a random (true) spatial distribution or a distribution (false) that build upon a equidistant repartition of points all around the area.
 - vi. open area : a float in meters representing the minimum distance for an area to be considered as an open area (i.e. euclidian distance between centroid and farest obstacle)
 - vii. density point : a float representing the density of points per meter within open areas.
 - viii. clean network : a boolean expression that allows to enhance the network (true) or living as it is generated (false). Enhancement includes filling very small gaps between edges and nodes.
 - ix. cliping : tolerance for the cliping in triangulation (float; distance) - see skeletonize operator
 - x. tolerance : tolerance for the triangulation (float)
 - xi. min dist obstacle : minimal distance to obstacles to keep a path (float; if 0.0, no filtering)
 - xii. simplification : simplification distance for the final geometries
- The method allows to build a network of corridors to be used by pedestrian while traveling around a space made of obstacles and other users. It makes it possible to avoide collision with other agents (e.g. buildings) including other pedestrians and in the same time managing a path to a destination in a complex environment (e.g. a city). The method is highly customizable, with many parameters listed as below: <p>
 - i. obstacles : a list containing the lists of geometries or agents that are obstacles for pedestrians (e.g. walls, cars).
 - ii. bounds : a list of geometries that represent the spatial boundary of the network (i.e. the enclosing space of the network).
 - iii. open : a boolean expression that will add nodes in the network within open areas. More precisely, new invisible points are added to improve triangulation in areas with very few obstacles.
 - iv. randomDist : a boolean expression, related to the previous 'open' parameter, that allows to switch between a random (true) spatial distribution or a distribution (false) that build upon a equidistant repartition of points all around the area.
 - v. open area : a float in meters representing the minimum distance for an area to be considered as an open area (i.e. euclidian distance between centroid and farest obstacle)
 - vi. density point : a float representing the density of points per meter within open areas.
 - vii. clean network : a boolean expression that allows to enhance the network (true) or living as it is generated (false). Enhancement includes filling very small gaps between edges and nodes.
 - viii. cliping : tolerance for the cliping in triangulation (float; distance) - see skeletonize operator
 - ix. tolerance : tolerance for the triangulation (float)
 - x. min dist obstacle : minimal distance to obstacles to keep a path (float; if 0.0, no filtering)
 - xi. simplification : simplification distance for the final geometries

Examples:

```
list<geometry> var0 <- generate_pedestrian_network([wall], [world], [road], true, false, 3.0, 0.1, true, 0.1, 0.0, 0.0, 0.0); //  
var0 equals a list of polylines corresponding to the pedestrian paths  
list<geometry> var1 <- generate_pedestrian_network([wall], [world], true, false, 3.0, 0.1, true, 0.1, 0.0, 0.0, 0.0, 50.0); // var1  
equals a list of polylines corresponding to the pedestrian paths  
list<geometry> var2 <- generate_pedestrian_network([wall], [world], [road], true, false, 3.0, 0.1, true, 0.1, 0.0, 0.0, 0.0, 50.0);  
// var2 equals a list of polylines corresponding to the pedestrian paths  
list<geometry> var3 <- generate_pedestrian_network([wall], [world], true, false, 3.0, 0.1, true, 0.1, 0.0, 0.0, 0.0, 0.0); // var3  
equals a list of polylines corresponding to the pedestrian paths
```

generate_random_graph

Possible uses:

- `generate_random_graph (int, int, bool) -> graph`
- `generate_random_graph (int, int, bool, species) -> graph`
- `generate_random_graph (int, int, bool, species, species) -> graph`

Result:

returns a random graph. returns a random graph. returns a random graph.

Special cases:

- `nbNodes`: number of nodes to be created; `nbEdges`: number of edges to be created; `directed`: is the graph has to be directed or not; `node_species`: the species of nodes; `edges_species`: the species of edges

```
graph<node_species,edge_species> myGraph <- generate_random_graph(  
50,  
100,  
true,  
node_species,  
edge_species);
```

- `nbNodes`: number of nodes to create; `nbEdges`: number of edges to create; `directed`: is the graph directed or not; `node_species`: the species of nodes

```
graph myGraph <- generate_random_graph(  
50,  
100,  
true,  
node_species);
```

- `nbNodes`: number of nodes to create; `nbEdges`: number of edges to create; `directed`: is the graph directed or not

```
graph myGraph <- generate_random_graph(  
50,  
100,  
true);
```

See also: [generate_barabasi_albert](#), [generate_watts_strogatz](#),

generate_terrain

Possible uses:

- `generate_terrain (int, int, int, float, float, float) --> field`

Result:

This operator allows to generate a pseudo-terrain using a simplex noise generator. Its usage is kept simple: it takes first a seed (random or not), then the dimensions (width and height) of the field to generate, then a level (between 0 and 1) of details (which actually determines the number of passes to make), then the value (between 0 and 1) of smoothness, with 0 being completely rough and 1 super smooth, and finally the value (between 0 and 1) of scattering, with 0 building maps in 'one piece' and 1 completely scattered ones.

generate_watts_strogatz

Possible uses:

- `generate_watts_strogatz (int, float, int, bool) --> graph`
- `generate_watts_strogatz (container, float, int, bool) --> graph`
- `generate_watts_strogatz (int, float, int, bool, species) --> graph`
- `generate_watts_strogatz (int, float, int, bool, species, species) --> graph`

Result:

returns a random small-world network (following Watts-Strogatz model). returns a random small-world network (following Watts-Strogatz model).
returns a random small-world network (following Watts-Strogatz model). returns a random small-world network (following Watts-Strogatz model).

Comment:

The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements: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:

- "nbNodes": 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); "directed": is the graph directed or not

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
  100,  
  0.3,  
  5,  
  true);
```

- "nbNodes": 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); "directed": is the graph directed or not; "node_species": the species of vertices; "edges_species": the species of edges

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
  100,  
  0.3,  
  5,  
  true,  
  myVertexSpecies,  
  myEdgeSpecies);
```

- "nbNodes": 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); "directed": is the graph directed or not; "node_species": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
  100,  
  0.3,  
  5,  
  true,  
  myVertexSpecies);
```

- "nodes": the list of nodes to connect; "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); "directed": is the graph directed or not

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(  
    people,  
    0.3,  
    5,  
    true);
```

See also: [generate_barabasi_albert](#),

geojson_file

Possible uses:

- `geojson_file (string) --> file`
- `string geojson_file int --> file`
- `geojson_file (string , int) --> file`
- `string geojson_file string --> file`
- `geojson_file (string , string) --> file`
- `string geojson_file bool --> file`
- `geojson_file (string , bool) --> file`
- `geojson_file (string, int, bool)--> file`
- `geojson_file (string, string, bool)--> file`

Result:

Constructs a file of type geojson. Allowed extensions are limited to json, geojson, geo.json

Special cases:

- `geojson_file(string)`: This file constructor allows to read a geojson file (<https://geojson.org/>)

```
file f <- geojson_file("file.json");
```

- `geojson_file(string,int)`: This file constructor allows to read a geojson file and specifying the coordinates system code, as an int

```
file f <- geojson_file("file.json", 32648);
```

- `geojson_file(string,string)`: This file constructor allows to read a geojson file and specifying the coordinates system code (epg,...,), as a string

```
file f <- geojson_file("file.json", "EPSG:32648");
```

- `geojson_file(string,bool)`: This file constructor allows to read a geojson file and take a potential z value (not taken in account by default)

```
file f <- geojson_file("file.json", true);
```

- `geojson_file(string,int,bool)`: This file constructor allows to read a geojson file, specifying the coordinates system code, as an int and take a potential z value (not taken in account by default)

```
file f <- geojson_file("file.json",32648, true);
```

- `geojson_file(string,string,bool)`: This file constructor allows to read a geojson file, specifying the coordinates system code (epg,...,), as a string and take a potential z value (not taken in account by default)

```
file f <- geojson_file("file.json", "EPSG:32648", true);
```

See also: [is_geojson](#),

geometric_mean

Possible uses:

- `geometric_mean` (container) ---> `float`

Result:

the geometric mean of the elements of the operand. See [Geometric_mean](#) for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- geometric_mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 4.962326343467649
```

See also: [mean](#), [median](#), [harmonic_mean](#),

geometry

Possible uses:

- `geometry` (any) ---> `geometry`

Result:

casts the operand in a geometry object.

geometry_collection

Possible uses:

- `geometry_collection` (container<unknown,geometry>) ---> `geometry`

Result:

A geometry collection (multi-geometry) composed of the given list of geometries.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single geometry, returns a copy of the geometry.

Examples:

```
geometry var0 <- geometry_collection([{0,0}, {0,10}, {10,10}, {10,0}]); // var0 equals a geometry composed of the 4 points (multi-point).
```

See also: [around](#), [circle](#), [cone](#), [link](#), [norm](#), [point](#), [polygone](#), [rectangle](#), [square](#), [triangle](#), [line](#),

get

Possible uses:

- `geometry get string` ---> `unknown`
- `get (geometry, string)`---> `unknown`
- `agent get string` ---> `unknown`
- `get (agent, string)`---> `unknown`

Result:

Reads an attribute of the specified agent (or geometry) (left operand). The attribute name is specified by the right operand.

Special cases:

- Reading the attribute of a geometry

```
string geom_area <- a_geometry get('area');      // reads then 'area' attribute of 'a_geometry' variable then assigns the returned value to the geom_area variable
```

- Reading the attribute of another agent

```
string agent_name <- an_agent get('name');      // reads then 'name' attribute of an_agent then assigns the returned value to the agent_name variable
```

get_about

Possible uses:

- `get_about (emotion)`---> `predicate`

Result:

get the about value of the given emotion

Examples:

```
get_about(emotion)
```

get_agent

Possible uses:

- `get_agent (social_link)`---> `agent`

Result:

get the agent value of the given social link

Examples:

```
get_agent(social_link1)
```

get_agent_cause

Possible uses:

- `get_agent_cause` (`predicate`) ---> `agent`
- `get_agent_cause` (`emotion`) ---> `agent`

Result:

evaluate the agent_cause value of a predicate get the agent cause value of the given emotion

Examples:

```
get_agent_cause(pred1)
get_agent_cause(emotion)
```

get_belief_op

Possible uses:

- `agent` `get_belief_op` `predicate` ---> `mental_state`
- `get_belief_op` (`agent`, `predicate`) ---> `mental_state`

Result:

get the belief in the belief base with the given predicate.

Examples:

```
mental_state var0 <- get_belief_op(self, predicate("has_water")); // var0 equals nil
```

get_belief_with_name_op

Possible uses:

- `agent` `get_belief_with_name_op` `string` ---> `mental_state`
- `get_belief_with_name_op` (`agent`, `string`) ---> `mental_state`

Result:

get the belief in the belief base with the given name.

Examples:

```
mental_state var0 <- get_belief_with_name_op(self, "has_water"); // var0 equals nil
```

get_beliefs_op

Possible uses:

- `agent` `get_beliefs_op` `predicate` ---> `list<mental_state>`

- `get_beliefs_op` (`agent`, `predicate`) \rightarrow `list<mental_state>`

Result:

get the beliefs in the belief base with the given predicate.

Examples:

```
get_beliefs_op(self, predicate("has_water"))
```

get_beliefs_with_name_op

Possible uses:

- `agent` `get_beliefs_with_name_op` `string` \rightarrow `list<mental_state>`
- `get_beliefs_with_name_op` (`agent`, `string`) \rightarrow `list<mental_state>`

Result:

get the list of beliefs in the belief base which predicate has the given name.

Examples:

```
get_beliefs_with_name_op(self, "has_water")
```

get_current_intention_op

Possible uses:

- `get_current_intention_op` (`agent`) \rightarrow `mental_state`

Result:

get the current intention.

Examples:

```
mental_state var0 <- get_current_intention_op(self); // var0 equals nil
```

get_decay

Possible uses:

- `get_decay` (`emotion`) \rightarrow `float`

Result:

get the decay value of the given emotion

Examples:

```
get_decay(emotion)
```

get_desire_op

Possible uses:

- `agent get_desire_op predicate --> mental_state`
- `get_desire_op (agent, predicate) --> mental_state`

Result:

get the desire in the desire base with the given predicate.

Examples:

```
mental_state var0 <- get_belief_op(self, predicate("has_water")); // var0 equals nil
```

get_desire_with_name_op

Possible uses:

- `agent get_desire_with_name_op string --> mental_state`
- `get_desire_with_name_op (agent, string) --> mental_state`

Result:

get the desire in the desire base with the given name.

Examples:

```
mental_state var0 <- get_desire_with_name_op(self, "has_water"); // var0 equals nil
```

get_desires_op

Possible uses:

- `agent get_desires_op predicate --> list<mental_state>`
- `get_desires_op (agent, predicate) --> list<mental_state>`

Result:

get the desires in the desire base with the given predicate.

Examples:

```
get_desires_op(self, predicate("has_water"))
```

get_desires_with_name_op

Possible uses:

- `agent get_desires_with_name_op string --> list<mental_state>`
- `get_desires_with_name_op (agent, string) --> list<mental_state>`

Result:

get the list of desires in the desire base which predicate has the given name.

Examples:

```
get_desires_with_name_op(self, "has_water")
```

get_dominance**Possible uses:**

- `get_dominance(social_link) -> float`

Result:

get the dominance value of the given social link

Examples:

```
get_dominance(social_link1)
```

get_familiarity**Possible uses:**

- `get_familiarity(social_link) -> float`

Result:

get the familiarity value of the given social link

Examples:

```
get_familiarity(social_link1)
```

get_ideal_op**Possible uses:**

- `agent get_ideal_op predicate -> mental_state`
- `get_ideal_op(agent, predicate) -> mental_state`

Result:

get the ideal in the ideal base with the given name.

Examples:

```
mental_state var0 <- get_ideal_op(self, predicate("has_water")); // var0 equals nil
```

get_ideal_with_name_op

Possible uses:

- `agent get_ideal_with_name_op string --> mental_state`
- `get_ideal_with_name_op (agent, string) --> mental_state`

Result:

get the ideal in the ideal base with the given name.

Examples:

```
mental_state var0 <- get_ideal_with_name_op(self, "has_water"); // var0 equals nil
```

get_ideals_op

Possible uses:

- `agent get_ideals_op predicate --> list<mental_state>`
- `get_ideals_op (agent, predicate) --> list<mental_state>`

Result:

get the ideal in the ideal base with the given name.

Examples:

```
get_ideals_op(self, predicate("has_water"))
```

get_ideals_with_name_op

Possible uses:

- `agent get_ideals_with_name_op string --> list<mental_state>`
- `get_ideals_with_name_op (agent, string) --> list<mental_state>`

Result:

get the list of ideals in the ideal base which predicate has the given name.

Examples:

```
get_ideals_with_name_op(self, "has_water")
```

get_intensity

Possible uses:

- `get_intensity (emotion) --> float`

Result:

get the intensity value of the given emotion

Examples:

```
get_intensity(emo1)
```

get_intention_op

Possible uses:

- `agent get_intention_op predicate --> mental_state`
- `get_intention_op (agent, predicate) --> mental_state`

Result:

get the intention in the intention base with the given predicate.

Examples:

```
get_intention_op(self, predicate("has_water"))
```

get_intention_with_name_op

Possible uses:

- `agent get_intention_with_name_op string --> mental_state`
- `get_intention_with_name_op (agent, string) --> mental_state`

Result:

get the intention in the intention base with the given name.

Examples:

```
get_intention_with_name_op(self, "has_water")
```

get_intentions_op

Possible uses:

- `agent get_intentions_op predicate --> list<mental_state>`
- `get_intentions_op (agent, predicate) --> list<mental_state>`

Result:

get the intentions in the intention base with the given predicate.

Examples:

```
get_intentions_op(self, predicate("has_water"))
```

get_intentions_with_name_op

Possible uses:

- `agent get_intentions_with_name_op string --> list<mental_state>`
- `get_intentions_with_name_op(agent, string) --> list<mental_state>`

Result:

get the list of intentions in the intention base which predicate has the given name.

Examples:

```
get_intentions_with_name_op(self, "has_water")
```

get_lifetime

Possible uses:

- `get_lifetime(mental_state) --> int`

Result:

get the lifetime value of the given mental state

Examples:

```
get_lifetime(mental_state1)
```

get_liking

Possible uses:

- `get_liking(social_link) --> float`

Result:

get the liking value of the given social link

Examples:

```
get_liking(social_link1)
```

get_modality

Possible uses:

- `get_modality(mental_state) --> string`

Result:

get the modality value of the given mental state

Examples:

```
get_modality(mental_state1)
```

get_obligation_op

Possible uses:

- `agent get_obligation_op predicate --> mental_state`
- `get_obligation_op (agent, predicate) --> mental_state`

Result:

get the obligation in the obligation base with the given predicate.

Examples:

```
mental_state var0 <- get_obligation_op(self, predicate("has_water")); // var0 equals nil
```

get_obligation_with_name_op

Possible uses:

- `agent get_obligation_with_name_op string --> mental_state`
- `get_obligation_with_name_op (agent, string) --> mental_state`

Result:

get the obligation in the obligation base with the given name.

Examples:

```
mental_state var0 <- get_obligation_with_name_op(self, "has_water"); // var0 equals nil
```

get_obligations_op

Possible uses:

- `agent get_obligations_op predicate --> list<mental_state>`
- `get_obligations_op (agent, predicate) --> list<mental_state>`

Result:

get the obligations in the obligation base with the given predicate.

Examples:

```
get_obligations_op(self, predicate("has_water"))
```

get_obligations_with_name_op

Possible uses:

- `agent get_obligations_with_name_op string --> list<mental_state>`
- `get_obligations_with_name_op (agent, string) --> list<mental_state>`

Result:

get the list of obligations in the obligation base which predicate has the given name.

Examples:

```
get_obligations_with_name_op(self, "has_water")
```

get_plan_name

Possible uses:

- `get_plan_name (BDIPlan) --> string`

Result:

get the name of a given plan

Examples:

```
get_plan_name(agent.current_plan)
```

get_predicate

Possible uses:

- `get_predicate (mental_state) --> predicate`

Result:

get the predicate value of the given mental state

Examples:

```
get_predicate(mental_state1)
```

get_solidarity

Possible uses:

- `get_solidarity (social_link) --> float`

Result:

get the solidarity value of the given social link

Examples:

```
get_solidarity(social_link1)
```

get_strength

Possible uses:

- `get_strength` (`mental_state`) --> `float`

Result:

get the strength value of the given mental state

Examples:

```
get_strength(mental_state1)
```

get_super_intention

Possible uses:

- `get_super_intention` (`predicate`) --> `mental_state`

Result:

get the super intention linked to a mental state

Examples:

```
get_super_intention(get_belief(pred1))
```

get_trust

Possible uses:

- `get_trust` (`social_link`) --> `float`

Result:

get the familiarity value of the given social link

Examples:

```
get_familiarity(social_link1)
```

get_truth

Possible uses:

- `get_truth` (`predicate`) --> `bool`

Result:

evaluate the truth value of a predicate

Examples:

```
get_truth(pred1)
```

get_uncertainties_op

Possible uses:

- `agent get_uncertainties_op predicate --> list<mental_state>`
- `get_uncertainties_op (agent, predicate) --> list<mental_state>`

Result:

get the uncertainties in the uncertainty base with the given predicate.

Examples:

```
get_uncertainties_op(self, predicate("has_water"))
```

get_uncertainties_with_name_op

Possible uses:

- `agent get_uncertainties_with_name_op string --> list<mental_state>`
- `get_uncertainties_with_name_op (agent, string) --> list<mental_state>`

Result:

get the list of uncertainties in the uncertainty base which predicate has the given name.

Examples:

```
get_uncertainties_with_name_op(self, "has_water")
```

get_uncertainty_op

Possible uses:

- `agent get_uncertainty_op predicate --> mental_state`
- `get_uncertainty_op (agent, predicate) --> mental_state`

Result:

get the uncertainty in the uncertainty base with the given predicate.

Examples:

```
mental_state var0 <- get_uncertainty_op(self, predicate("has_water")); // var0 equals nil
```

get_uncertainty_with_name_op

Possible uses:

- `agent get_uncertainty_with_name_op string --> mental_state`
- `get_uncertainty_with_name_op (agent, string) --> mental_state`

Result:

get the uncertainty in the uncertainty base with the given name.

Examples:

```
mental_state var0 <- get_uncertainty_with_name_op(self, "has_water"); // var0 equals nil
```

get_values

Possible uses:

- `get_values (predicate) --> map<string, unknown>`

Result:

return the map values of a predicate

Examples:

```
get_values(pred1)
```

gif_file

Possible uses:

- `gif_file (string) --> file`
- `string gif_file matrix<int> --> file`
- `gif_file (string, matrix<int>) --> file`

Result:

Constructs a file of type gif. Allowed extensions are limited to gif

Special cases:

- `gif_file(string):` This file constructor allows to read a gif file

```
gif_file f <- gif_file("file.gif");
```

- `gif_file(string,matrix<int>):` This file constructor allows to store a matrix in a gif file (it does not save it - just store it in memory)

```
gif_file f <- gif_file("file.gif",matrix([10,10],[10,10]));
```

See also: [is_gif](#),

gini

Possible uses:

- `gini (list<float>) ---> float`

Special cases:

- return the Gini Index of the given list of values (list of floats)

```
float var0 <- gini([1.0, 0.5, 2.0]); // var0 equals the gini index computed i.e. 0.2857143
```

girvan_newman_clustering

Possible uses:

- `graph girvan_newman_clustering int ---> list`
- `girvan_newman_clustering (graph, int)---> list`

Result:

The Girvan-Newman algorithm is a hierarchical method used to detect communities. It detects communities by progressively removing edges from the original network. It returns a list of lists of vertices and takes as operand the graph and the number of clusters

gml_file

Possible uses:

- `gml_file (string) ---> file`
- `string gml_file int ---> file`
- `gml_file (string, int)---> file`
- `string gml_file string ---> file`
- `gml_file (string, string)---> file`
- `string gml_file bool ---> file`
- `gml_file (string, bool)---> file`
- `gml_file (string, int, bool)---> file`
- `gml_file (string, string, bool)---> file`

Result:

Constructs a file of type gml. Allowed extensions are limited to gml

Special cases:

- `gml_file(string):` This file constructor allows to read a gml file

```
file f <- gml_file("file.gml");
```

- `gml_file(string,int):` This file constructor allows to read a gml file and specifying the coordinates system code, as an int (epsg code)

```
file f <- gml_file("file.gml", 32648);
```

- `gml_file(string,string)`: This file constructor allows to read a gml file and specifying the coordinates system code (epg,...,), as a string

```
file f <- gml_file("file.gml", "EPSG:32648");
```

- `gml_file(string,bool)`: This file constructor allows to read a gml file and take a potential z value (not taken in account by default)

```
file f <- gml_file("file.gml", true);
```

- `gml_file(string,int,bool)`: This file constructor allows to read a gml file, specifying the coordinates system code, as an int (epsg code) and take a potential z value (not taken in account by default)

```
file f <- gml_file("file.gml", 32648, true);
```

- `gml_file(string,string,bool)`: This file constructor allows to read a gml file, specifying the coordinates system code (epg,...,), as a string and take a potential z value (not taken in account by default)

```
file f <- gml_file("file.gml", "EPSG:32648",true);
```

See also: [is_gml](#),

gradient

Possible uses:

- `gradient (map<rgb, float>) ---> map<rgb, float>`
- `gradient (list<rgb>) ---> map<rgb, float>`
- `rgb gradient rgb ---> map<rgb, float>`
- `gradient (rgb , rgb) ---> map<rgb, float>`
- `gradient (rgb, rgb, float) ---> map<rgb, float>`

Result:

returns the definition of a linear gradient between n colors provided with their positions on a scale between 0 and 1. A similar color map is returned, in the same color order, with all the positions normalized (so that they are shifted and scaled to fit between 0 and 1). Throws an error if the number of colors is less than 2 or if the positions are not strictly ordered returns the definition of a linear gradient between two colors, represented internally as a color map [start::0.0,stop::1.0] returns the definition of a linear gradient between two colors, with a ratio (between 0 and 1, otherwise clamped) represented internally as a color map [start::0.0,(startr+stop(1-r)::r, stop::1.0] returns the definition of a linear gradient between n colors, represented internally as a color map [c1::1/n,c2::1/n, ... cn::1/n]

graph

Possible uses:

- `graph (any) ---> graph`

Result:

casts the operand in a graph object.

graph6_file

Possible uses:

- `graph6_file` (`string`) ---> `file`
- `string` `graph6_file` (`species`) ---> `file`
- `graph6_file` (`string`, `species`) ---> `file`
- `graph6_file` (`string`, `species`, `species`) ---> `file`

Result:

Constructs a file of type graph6. Allowed extensions are limited to graph6

Special cases:

- `graph6_file(string)`: References a graph6 file by its filename
- `graph6_file(string,species)`: References a graph6 file by its filename and the species to use to instantiate the nodes
- `graph6_file(string,species,species)`: References a graph6 file by its filename and the species to use to instantiate the nodes and the edges

See also: [is_graph6](#),

graphdimacs_file

Possible uses:

- `graphdimacs_file` (`string`) ---> `file`
- `string` `graphdimacs_file` (`species`) ---> `file`
- `graphdimacs_file` (`string`, `species`) ---> `file`
- `graphdimacs_file` (`string`, `species`, `species`) ---> `file`

Result:

Constructs a file of type graphdimacs. Allowed extensions are limited to dimacs

Special cases:

- `graphdimacs_file(string)`: References a dimacs file by its filename
- `graphdimacs_file(string,species)`: References a dimacs file by its filename and the species to use to instantiate the nodes
- `graphdimacs_file(string,species,species)`: References a dimacs file by its filename and the species to use to instantiate the nodes and the edges

See also: [is_graphdimacs](#),

graphdot_file

Possible uses:

- `graphdot_file` (`string`) ---> `file`
- `string` `graphdot_file` (`species`) ---> `file`
- `graphdot_file` (`string`, `species`) ---> `file`
- `graphdot_file` (`string`, `species`, `species`) ---> `file`

Result:

Constructs a file of type graphdot. Allowed extensions are limited to dot

Special cases:

- `graphdot_file(string)`: References a dot graph file by its filename
- `graphdot_file(string,species)`: References a dot graph file by its filename and the species to use to instantiate the nodes
- `graphdot_file(string,species,species)`: References a dot graph file by its filename and the 2 species to use to instantiate the nodes and the edges

See also: [is_graphdot](#),

`graphgexf_file`

Possible uses:

- `graphgexf_file (string) --> file`
- `string graphgexf_file species --> file`
- `graphgexf_file (string , species) --> file`
- `graphgexf_file (string, species, species) --> file`

Result:

Constructs a file of type graphgexf. Allowed extensions are limited to gexf

Special cases:

- `graphgexf_file(string)`: References a gexf graph file by its filename
- `graphgexf_file(string,species)`: References a gexf graph file by its filename and the species to use to instantiate the nodes
- `graphgexf_file(string,species,species)`: References a gexf graph file by its filename and the 2 species to use to instantiate the nodes and the edges

See also: [is_graphgexf](#),

`graphgml_file`

Possible uses:

- `graphgml_file (string) --> file`
- `string graphgml_file species --> file`
- `graphgml_file (string , species) --> file`
- `graphgml_file (string, species, species) --> file`

Result:

Constructs a file of type graphgml. Allowed extensions are limited to gml

Special cases:

- `graphgml_file(string)`: References a gml graph file by its filename
- `graphgml_file(string,species)`: References a gml graph file by its filename and the species to use to instantiate the nodes
- `graphgml_file(string,species,species)`: References a gml graph file by its filename and the 2 species to use to instantiate the nodes and the edges

See also: [is_graphgml](#),

graphml_file

Possible uses:

- `graphml_file (string) ---> file`
- `string graphml_file species ---> file`
- `graphml_file (string, species) ---> file`
- `graphml_file (string, species, species) ---> file`
- `graphml_file (string, species, species, string, string) ---> file`

Result:

Constructs a file of type graphml. Allowed extensions are limited to graphml

Special cases:

- `graphml_file(string)`: References a graphml graph file by its filename
- `graphml_file(string,species)`: References a graphml graph file by its filename and the species to use to instantiate the nodes
- `graphml_file(string,species,species)`: References a graphml graph file by its filename and the 2 species to use to instantiate the nodes and the edges
- `graphml_file(string,species,species,string,string)`: References a graphml graph file by its filename and the 2 species to use to instantiate the nodes and the edges

See also: [is_graphml](#),

graphtsplib_file

Possible uses:

- `graphtsplib_file (string) ---> file`
- `string graphtsplib_file species ---> file`
- `graphtsplib_file (string, species) ---> file`
- `graphtsplib_file (string, species, species) ---> file`

Result:

Constructs a file of type graphtsplib. Allowed extensions are limited to tsplib

Special cases:

- `graphtsplib_file(string)`: References a tsplib graph file by its filename
- `graphtsplib_file(string,species)`: References a tsplib graph file by its filename and the species to use to instantiate the nodes
- `graphtsplib_file(string,species,species)`: References a tsplib graph file by its filename and the 2 species to use to instantiate the nodes and the edges

See also: [is_graphtsplib](#),

grayscale

Possible uses:

- `grayscale (rgb) ---> rgb`

Result:

Converts rgb color to grayscale value

Comment:

r=red, g=green, b=blue. Between 0 and 255 and gray = $0.299 \cdot \text{red} + 0.587 \cdot \text{green} + 0.114 \cdot \text{blue}$ (Photoshop value)

Examples:

```
rgb var0 <- grayscale (rgb(255,0,0)); // var0 equals to a dark grey
```

See also: [rgb](#), [hsb](#),

grayscale

Possible uses:

- `grayscale (image) ---> image`

Result:

Used to convert any image to a grayscale color palette and return it. The original image is left untouched

grid_at

Possible uses:

- `species grid_at point ---> agent`
- `grid_at (species , point) ---> agent`

Result:

returns the cell of the grid (right-hand operand) at the position given by the right-hand operand

Comment:

If the left-hand operand is a point of floats, it is used as a point of ints.

Special cases:

- if the left-hand operand is not a grid cell species, returns nil

Examples:

```
agent var0 <- grid_cell grid_at {1,2}; // var0 equals the agent grid_cell with grid_x=1 and grid_y = 2
```

grid_cells_to_graph

Possible uses:

- `grid_cells_to_graph (container) ---> graph`
- `container grid_cells_to_graph species ---> graph`
- `grid_cells_to_graph (container , species) ---> graph`

Result:

creates a graph from a list of cells (operand). An edge is created between neighbors.

Examples:

```
my_cell_graph <- grid_cells_to_graph(cells_list);
```

See also: [as_intersection_graph](#), [as_edge_graph](#),

grid_file

Possible uses:

- `grid_file(string) -> file`
- `string grid_file bool -> file`
- `grid_file(string, bool) -> file`
- `string grid_file int -> file`
- `grid_file(string, int) -> file`
- `string grid_file string -> file`
- `grid_file(string, string) -> file`
- `string grid_file field -> file`
- `grid_file(string, field) -> file`

Result:

Constructs a file of type grid. Allowed extensions are limited to asc, tif

Special cases:

- `grid_file(string)`: This file constructor allows to read a asc file or a tif (geotif) file

```
file f <- grid_file("file.asc");
```

- `grid_file(string,bool)`: This file constructor allows to read a asc file or a tif (geotif) file, but without converting it into shapes. Only a matrix of float values is created

```
file f <- grid_file("file.asc", false);
```

- `grid_file(string,int)`: This file constructor allows to read a asc file or a tif (geotif) file specifying the coordinates system code, as an int (epsg code)

```
file f <- grid_file("file.asc", 32648);
```

- `grid_file(string,string)`: This file constructor allows to read a asc file or a tif (geotif) file specifying the coordinates system code (epg,...), as a string

```
file f <- grid_file("file.asc", "EPSG:32648");
```

- `grid_file(string,field)`: This allows to build a writable grid file from the values of a field

```
file f <- grid_file("file.tif", my_field); save f;
```

See also: [is_grid](#),

group_by

Possible uses:

- `container group_by any expression ---> map`
- `group_by (container , any expression) ---> map`

Result:

Returns a map, where the keys take the possible values of the right-hand operand and the map values are the list of elements of the left-hand operand associated to the key value

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, group_by throws an error

Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] group_by (each > 3); // var0 equals [false:[1, 2, 3], true:[4, 5, 6, 7, 8]]  
map var1 <- g2 group_by (length(g2 out_edges_of each) ); // var1 equals [ 0:[node9, node7, node10, node8, node11],  
1:[node6], 2:[node5], 3:[node4]]  
map var2 <- (list(node) group_by (round(node(each).location.x)); // var2 equals [32:[node5], 21:[node1], 4:[node0],  
66:[node2], 96:[node3]]  
map<bool,list> var3 <- [1::2, 3::4, 5::6] group_by (each > 4); // var3 equals [false:[2, 4], true:[6]]
```

See also: [first_with](#), [last_with](#), [where](#),

harmonic_mean

Possible uses:

- `harmonic_mean (container)---> float`

Result:

the harmonic mean of the elements of the operand. See [Harmonic_mean](#) for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- harmonic_mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 4.804159445407279
```

See also: [mean](#), [median](#), [geometric_mean](#),

has_belief_op

Possible uses:

- `agent has_belief_op predicate --> bool`
- `has_belief_op (agent, predicate) --> bool`

Result:

indicates if there already is a belief about the given predicate.

Examples:

```
bool var0 <- has_belief_op(self,predicate("has_water")); // var0 equals false
```

has_belief_with_name_op

Possible uses:

- `agent has_belief_with_name_op string --> bool`
- `has_belief_with_name_op (agent, string) --> bool`

Result:

indicates if there already is a belief about the given name.

Examples:

```
bool var0 <- has_belief_with_name_op(self,"has_water"); // var0 equals false
```

has_desire_op

Possible uses:

- `agent has_desire_op predicate --> bool`
- `has_desire_op (agent, predicate) --> bool`

Result:

indicates if there already is a desire about the given predicate.

Examples:

```
bool var0 <- has_desire_op(self,predicate("has_water")); // var0 equals false
```

has_desire_with_name_op

Possible uses:

- `agent has_desire_with_name_op string --> bool`
- `has_desire_with_name_op (agent, string) --> bool`

Result:

indicates if there already is a desire about the given name.

Examples:

```
bool var0 <- has_desire_with_name_op(self, "has_water"); // var0 equals false
```

has_ideal_op**Possible uses:**

- `agent has_ideal_op predicate ---> bool`
- `has_ideal_op (agent , predicate)---> bool`

Result:

indicates if there already is an ideal about the given predicate.

Examples:

```
bool var0 <- has_ideal_op(self,predicate("has_water")); // var0 equals false
```

has_ideal_with_name_op**Possible uses:**

- `agent has_ideal_with_name_op string ---> bool`
- `has_ideal_with_name_op (agent , string)---> bool`

Result:

indicates if there already is an ideal about the given name.

Examples:

```
bool var0 <- has_ideal_with_name_op(self, "has_water"); // var0 equals false
```

has_intention_op**Possible uses:**

- `agent has_intention_op predicate ---> bool`
- `has_intention_op (agent , predicate)---> bool`

Result:

indicates if there already is an intention about the given predicate.

Examples:

```
bool var0 <- has_intention_op(self,predicate("has_water")); // var0 equals false
```

has_intention_with_name_op

Possible uses:

- `agent has_intention_with_name_op string --> bool`
- `has_intention_with_name_op (agent, string) --> bool`

Result:

indicates if there already is an intention about the given name.

Examples:

```
bool var0 <- has_intention_with_name_op(self, "has_water"); // var0 equals false
```

has_obligation_op

Possible uses:

- `agent has_obligation_op predicate --> bool`
- `has_obligation_op (agent, predicate) --> bool`

Result:

indicates if there already is an obligation about the given predicate.

Examples:

```
bool var0 <- has_obligation_op(self, predicate("has_water")); // var0 equals false
```

has_obligation_with_name_op

Possible uses:

- `agent has_obligation_with_name_op string --> bool`
- `has_obligation_with_name_op (agent, string) --> bool`

Result:

indicates if there already is an obligation about the given name.

Examples:

```
bool var0 <- has_obligation_with_name_op(self, "has_water"); // var0 equals false
```

has_uncertainty_op

Possible uses:

- `agent has_uncertainty_op predicate --> bool`
- `has_uncertainty_op (agent, predicate) --> bool`

Result:

indicates if there already is an uncertainty about the given predicate.

Examples:

```
bool var0 <- has_uncertainty_op(self,predicate("has_water")); // var0 equals false
```

has_uncertainty_with_name_op**Possible uses:**

- `agent has_uncertainty_with_name_op string --> bool`
- `has_uncertainty_with_name_op (agent , string)---> bool`

Result:

indicates if there already is an uncertainty about the given name.

Examples:

```
bool var0 <- has_uncertainty_with_name_op(self,"has_water"); // var0 equals false
```

hexagon**Possible uses:**

- `hexagon (point) --> geometry`
- `hexagon (float) --> geometry`
- `float hexagon float --> geometry`
- `hexagon (float , float) --> geometry`

Result:

A hexagon geometry which the given width and height

Comment:

the center of the hexagon is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- hexagon(10,5); // var0 equals a geometry as a hexagon of width of 10 and height of 5.  
geometry var1 <- hexagon({10,5}); // var1 equals a geometry as a hexagon of width of 10 and height of 5.  
geometry var2 <- hexagon(10); // var2 equals a geometry as a hexagon of width of 10 and height of 10.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [triangle](#),

hierarchical_clustering

Possible uses:

- `container<unknown,agent>` `hierarchical_clustering` `float` ---> `list`
- `hierarchical_clustering` (`container<unknown,agent>`, `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](#),

horizontal

Possible uses:

- `horizontal` (`map<unknown,int>`) ---> `unknown<string>`

Result:

Creates a horizontal layout node (a sash). Sashes can contain any number (> 1) of other elements: stacks, horizontal or vertical sashes, or display indices. Each element is represented by a pair in the map, where the key is the element and the value its weight within the sash

horizontal_flip

Possible uses:

- `horizontal_flip` (`image`) ---> `image`

Result:

Returns an image flipped horizontally by reflecting the original image around the y axis. The original image is left untouched

hsb

Possible uses:

- `hsb` (`float`, `float`, `float`) ---> `rgb`
- `hsb` (`float`, `float`, `float`, `int`) ---> `rgb`
- `hsb` (`float`, `float`, `float`, `float`) ---> `rgb`

Result:

Converts hsb (h=hue, s=saturation, b=brightness) value to Gama color

Comment:

h,s and b components should be floating-point values between 0.0 and 1.0 and when used alpha should be an integer (between 0 and 255) or a float (between 0 and 1). Examples: Red=(0.0,1.0,1.0), Yellow=(0.16,1.0,1.0), Green=(0.33,1.0,1.0), Cyan=(0.5,1.0,1.0), Blue=(0.66,1.0,1.0), Magenta=(0.83,1.0,1.0)

Examples:

```
rgb var0 <- hsb (0.5,1.0,1.0,0.0); // var0 equals rgb("cyan",0)
rgb var1 <- hsb (0.0,1.0,1.0); // var1 equals rgb("red")
```

See also: [rgb](#),

hypot**Possible uses:**

- `hypot (float, float, float, 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)
```

Version: 1.9.1

Operators (I to M)

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 spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
    }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considered as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

`box, cone3D, cube, cylinder, hexagon, pyramid, set_z, sphere, teapot,`

Arithmetic operators

`-, /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,`

BDI

add_values, and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or, set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_values,

Casting operators

as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of, to_gaml, to_geojson, to_list, with_size, with_style,

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean, median, palette, rgb, rnd_color, scale, sum, to_hsb,

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect, contains, contains_all, contains_any, contains_key, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

Date-related operators

-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours, minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours, plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,

Dates

Displays

horizontal, stack, vertical,

edge

`edge_between, strahler,`

EDP-related operators

`diff, diff2,`

Files-related operators

`copy_file, crs, csv_file, delete_file, dxf_file, evaluate_sub_model, file_exists, folder, folder_exists, gaml_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,`

GamaMetaType

`type_of,`

Gen*

`add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,`

Graphs-related operators

`add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, as_spatial_graph, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_random_graph, generate_watts_strogatz, girvan_newman_clustering, grid_cells_to_graph, in_degree_of, in_edges_of, k_spanning_tree_clustering, label_propagation_clustering, layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid, load_shortest_paths, main_connected_component, max_flow_between, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_from, rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of, undirected, use_cache, weight_of, with_k_shortest_path_algorithm, with_shortest_path_algorithm, with_weights,`

Grid-related operators

`as_4_grid, as_grid, as_hexagonal_grid, cell_at, cells_in, cells_overlapping, field, grid_at, neighbors_of, path_between, points_in, values_in,`

ImageOperators

`*, antialiased, blend, blurred, brighter, clipped_with, darker, grayscale, horizontal_flip, image, matrix, rotated_by, sharpened, snapshot, tinted_with, vertical_flip, with_height, with_size, with_width,`

Iterator operators

accumulate, all_match, as_map, collect, count, create_map, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, none_matches, one_matches, product_of, sort_by, sum_of, variance_of, where, where, where, with_max_of, with_min_of,

List-related operators

all_indexes_of, copy_between, index_of, last_index_of,

Logical operators

; !, ?, add_3Dmodel, add_geometry, add_icon, and, or, xor,

Map comparaison operators

fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,

Map-related operators

as_map, create_map, index_of, last_index_of,

Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

multicriteria operators

electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM, weighted_means_DM,

Path-related operators

agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, max_flow_between, path_between, path_to, paths_between, use_cache,

Pedestrian

generate_pedestrian_network,

Points-related operators

-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid, closest_points_with, farthest_point_to, grid_at, norm, points_along, points_at, points_on,

Random operators

binomial, flip, gamma_density, gamma_rnd, gamma_trunc_rnd, gauss, generate_terrain, lognormal_density, lognormal_rnd, lognormal_trunc_rnd, poisson, rnd, rnd_choice, sample, shuffle, skew_gauss, truncated_gauss, weibull_density, weibull_rnd, weibull_trunc_rnd,

ReverseOperators

restore_simulation, restore_simulation_from_file, save_simulation, serialize, serialize_agent,

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, elliptical_arc, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

Spatial operators

- , *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, angle_between, any_location_in, arc, around, as_4_grid, as_driving_graph, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covering, covers, cross, crosses, crossing, crs, CRS_transform, cube, curve, cylinder, direction_between, disjoint_from, distance_between, distance_to, ellipse, elliptical_arc, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, inverse_rotation, k_nearest_neighbors, line, link, masked_by, moran, neighbors_at, neighbors_of, normalized_rotation, overlapping, overlaps, partially_overlapping, partially_overlaps, path_between, path_to, plan, points_along, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rotated_by, rotation_composition, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, touches, touching, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, at_distance, closest_to, covering, crossing, farthest_to, inside, neighbors_at, neighbors_of, overlapping, partially_overlapping, touching,

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

Spatial statistical operators

hierarchical_clustering, k_nearest_neighbors, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, inverse_rotation, normalized_rotation, rotated_by, rotation_composition, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum, build, chi_square, chi_square_complemented, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis, log_gamma, max, mean, mean_deviation, median, min, moment, moran, morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, t_test, variance,

Strings-related operators

+, <, <=, >, >=, at, capitalize, char, contains, contains_all, contains_any, copy_between, date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, regex_matches, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

SubModel

load_sub_model,

System

., choose, command, copy, copy_from_clipboard, copy_to_clipboard, copy_to_clipboard, dead, enter, eval_gaml, every, is_error, is_reachable, is_warning, play_sound, user_confirm, user_input_dialog, wizard, wizard_page,

Time-related operators

date, string,

Types-related operators

action, agent, attributes, BDIPPlan, bool, container, conversation, directory, emotion, file, float, gamm_type, gen_population_generator, gen_range, geometry, graph, int, kml, list, map, matrix, mental_state, message, Norm, pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, species, topology, unknown,

User control operators

choose, enter, user_confirm, user_input_dialog, wizard, wizard_page,

Operators

IDW

Possible uses:

- `IDW (container<unknown, geometry>, map, int) --> map<geometry, float>`

Result:

Inverse Distance Weighting (IDW) is a type of deterministic method for multivariate interpolation with a known scattered set of points. The assigned values to each geometry are calculated with a weighted average of the values available at the known points. See: http://en.wikipedia.org/wiki/Inverse_distance_weighting Usage: IDW (list of geometries, map of points (key: point, value: value), power parameter)

Examples:

```
map<geometry, float> var0 <- IDW([ag1, ag2, ag3, ag4, ag5], [{10,10}::25.0, {10,80}::10.0, {100,10}::15.0], 2); // var0 equals for example, can return [ag1::12.0, ag2::23.0, ag3::12.0, ag4::14.0, ag5::17.0]
```

image

Possible uses:

- `int image int --> image`
- `image (int, int) --> image`
- `image (int, int, rgb) --> image`
- `image (int, int, bool) --> image`

Result:

Builds a new image with the specified dimensions and already filled with the given rgb color
Builds a new blank image with the specified dimensions and indicates if it will support transparency or not
Builds a new blank image of the specified dimensions, which does not accept transparency

image_file

Possible uses:

- `image_file (string) --> file`
- `string image_file string --> file`
- `image_file (string, string) --> file`
- `string image_file matrix<int> --> file`
- `image_file (string, matrix<int>) --> file`
- `image_file (string, java.awt.image.BufferedImage, bool) --> file`

Result:

Constructs a file of type image. Allowed extensions are limited to tiff, jpg, jpeg, png, pict, bmp

Special cases:

- `image_file(string,java.awt.image.BufferedImage,bool):`
- `image_file(string):` This file constructor allows to read an image file (tiff, jpg, jpeg, png, pict, bmp)

```
file f <-image_file("file.png");
```

- `image_file(string,string):` This file constructor allows to read an image file (tiff, jpg, jpeg, png, pict, bmp) and to force the extension of the file (can be useful for images coming from URL)

```
file f <-image_file("http://my_url", "png");
```

- `image_file(string,matrix<int>):` This file constructor allows to store a matrix in a image file (it does not save it - just store it in memory)

```
file f <-image_file("file.png");
```

See also: [is_image](#),

in

Possible uses:

- `string in string --> bool`
- `in (string , string) --> bool`
- `unknown in container --> bool`
- `in (unknown , container) --> bool`

Result:

true if the right operand contains the left operand, false otherwise

Comment:

the definition of in depends on the container

Special cases:

- if both operands are strings, returns true if the left-hand operand patterns is included in to the right-hand string;
- if the right operand is nil or empty, in returns false

Examples:

```
bool var0 <- 'bc' in 'abcd'; // var0 equals true
bool var1 <- 2 in [1,2,3,4,5,6]; // var1 equals true
bool var2 <- 7 in [1,2,3,4,5,6]; // var2 equals false
bool var3 <- 3 in [1::2, 3::4, 5::6]; // var3 equals false
bool var4 <- 6 in [1::2, 3::4, 5::6]; // var4 equals true
```

See also: [contains](#),

in_degree_of

Possible uses:

- graph in_degree_of unknown --> int
- in_degree_of(graph, unknown) --> int

Result:

returns the in degree of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap in_degree_of (node(3)); // var1 equals 2
```

See also: [out_degree_of](#), [degree_of](#),

in_edges_of

Possible uses:

- graph in_edges_of unknown --> list
- in_edges_of(graph, 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](#),

incomplete_beta

Possible uses:

- incomplete_beta(float, float, float) --> float

Result:

Returns the regularized integral of the beta function with arguments a and b, from zero to x.

Examples:

```
float var0 <- incomplete_beta(2,3,0.9) with_precision(3); // var0 equals 0.996
```

incomplete_gamma

Possible uses:

- float incomplete_gamma(float) --> float

- `incomplete_gamma`(`float`, `float`) \rightarrow `float`

Result:

Returns the regularized integral of the Gamma function with argument a to the integration end point x.

Examples:

```
float var0 <- incomplete_gamma(1,5.3) with_precision(3); // var0 equals 0.995
```

incomplete_gamma_complement

Possible uses:

- `float incomplete_gamma_complement float` \rightarrow `float`
- `incomplete_gamma_complement (float, float)` \rightarrow `float`

Result:

Returns the complemented regularized incomplete Gamma function of the argument a and integration start point x.

Comment:

Is the complement to 1 of incomplete_gamma.

Examples:

```
float var0 <- incomplete_gamma_complement(1,5.3) with_precision(3); // var0 equals 0.005
```

indented_by

Possible uses:

- `string indented_by int` \rightarrow `string`
- `indented_by (string, int)` \rightarrow `string`

Result:

Converts a (possibly multiline) string by indenting it by a number -- specified by the second operand -- of tabulations to the right

Examples:

```
string var0 <- "my" + indented_by("text", 1); // var0 equals "my      text"
```

index_by

Possible uses:

- `container index_by any expression` \rightarrow `map`
- `index_by (container, any expression)` \rightarrow `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. If the operation results in duplicate keys, only the first value corresponding to the key is kept

Examples:

```
map var0 <- [1,2,3,4,5,6,7,8] index_by (each - 1); // var0 equals [0::1, 1::2, 2::3, 3::4, 4::5, 5::6, 6::7, 7::8]
```

index_of

Possible uses:

- matrix index_of unknown ---> point
- index_of (matrix, unknown) ---> point
- species index_of unknown ---> int
- index_of (species, unknown) ---> int
- string index_of string ---> int
- index_of (string, string) ---> int
- list index_of unknown ---> int
- index_of (list, unknown) ---> int
- map<unknown, unknown> index_of unknown ---> unknown
- index_of (map<unknown, unknown>, unknown) ---> unknown

Result:

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 matrix, index_of returns the index as a point

```
point var0 <- matrix([[1,2,3],[4,5,6]]) index_of 4; // var0 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 var1 <- "abcababc" index_of "ca"; // var1 equals 2
```

- if the left operand is a list, index_of returns the index as an integer

```
int var2 <- [1,2,3,4,5,6] index_of 4; // var2 equals 3  
int var3 <- [4,2,3,4,5,4] index_of 4; // var3 equals 0
```

Examples:

```
unknown var4 <- [1::2, 3::4, 5::6] index_of 4; // var4 equals 3
```

See also: [at](#), [last_index_of](#),

inside

Possible uses:

- `container<unknown, geometry> inside geometry --> list<geometry>`
- `inside (container<unknown, geometry>, geometry) --> list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), covered by the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] inside(self); // var0 equals the agents among ag1, ag2 and ag3 that are covered by the shape of the right-hand argument.  
list<geometry> var1 <- (species1 + species2) inside (self); // var1 equals the agents among species species1 and species2 that are covered by the shape of the right-hand argument.
```

See also: [neighbors_at](#), [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [agents_inside](#), [agent_closest_to](#),

int

Possible uses:

- `int (any) --> int`

Result:

casts the operand in a int object.

inter

Possible uses:

- `geometry inter geometry --> geometry`
- `inter(geometry, geometry) --> geometry`
- `container inter container --> list`
- `inter(container, container) --> list`

Result:

A geometry resulting from the intersection between the two geometries the intersection of the two operands

Comment:

both containers are transformed into sets (so without duplicated element, cf. `remove_duplicates` operator) before the set intersection is computed.

Special cases:

- returns nil if one of the operands is nil
- if an operand is a graph, it will be transformed into the set of its nodes
- if an operand is a map, it will be transformed into the set of its values

```
list var3 <- [1::2, 3::4, 5::6] inter [2,4]; // var3 equals [2,4]
list var4 <- [1::2, 3::4, 5::6] inter [1,3]; // var4 equals []
```

- if an operand is a matrix, it will be transformed into the set of the lines

```
list var5 <- matrix([[3,2,1],[4,5,4]]) inter [3,4]; // var5 equals [3,4]
```

Examples:

```
geometry var0 <- square(10) inter circle(5); // var0 equals circle(5)
list var1 <- [1,2,3,4,5,6] inter [2,4]; // var1 equals [2,4]
list var2 <- [1,2,3,4,5,6] inter [0,8]; // var2 equals []
```

See also: [union](#), [+](#), [-](#), [remove_duplicates](#),

interleave

Possible uses:

- `interleave` (`container`) --> `list`

Result:

Returns 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": "e11", "e12": "e12", "e13": "e13"}, {"e21": "e21", "e22": "e22", "e23": "e23"}, {"e31": "e31", "e32": "e32", "e33": "e33"}]]; // var1 equals [{"e11": "e11", "e12": "e12", "e13": "e13"}, {"e21": "e21", "e22": "e22", "e23": "e23"}, {"e31": "e31", "e32": "e32", "e33": "e33"}]]
```

internal_integrated_value

Possible uses:

- `any expression` `internal_integrated_value` `any expression` --> `list`
- `internal_integrated_value` (`any expression`, `any expression`) --> `list`

Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with [index]

intersecting

Same signification as [overlapping](#)

intersection

Same signification as [inter](#)

intersects

Possible uses:

- `geometry intersects geometry --> bool`
- `intersects (geometry, geometry) --> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) intersects the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- square(5) intersects {10,10}; // var0 equals false
```

See also: [disjoint_from](#), [crosses](#), [overlaps](#), [partially_overlaps](#), [touches](#),

inverse

Possible uses:

- `inverse (matrix) --> matrix<float>`

Result:

The inverse matrix of the given matrix. If no inverse exists, returns a matrix that has properties that resemble that of an inverse.

Examples:

```
matrix<float> var0 <- inverse(matrix([[4,3],[3,2]])); // var0 equals matrix([[-2.0,3.0],[3.0,-4.0]])
```

inverse_distance_weighting

Same signification as [IDW](#)

inverse_rotation

Possible uses:

- `inverse_rotation` (`pair<float, point>`) \rightarrow `pair<float, point>`

Result:

The inverse rotation. It is a rotation around the same axis with the opposite angle.

Examples:

```
pair<float, point> var0 <- inverse_rotation(38.0:{1,1,1}); // var0 equals -38.0:{1,1,1}
```

See also: [rotation_composition, normalized_rotation](OperatorsSZ#rotation_composition, normalized_rotation),

is

Possible uses:

- `unknown is any expression` \rightarrow `bool`
- `is (unknown, any expression)` \rightarrow `bool`

Result:

returns true if the left operand is of the right operand type, false otherwise

Examples:

```
bool var0 <- 0 is int; // var0 equals true
bool var1 <- an_agent is node; // var1 equals true
bool var2 <- 1 is float; // var2 equals false
```

is_csv

Possible uses:

- `is_csv (any)` \rightarrow `bool`

Result:

Tests whether the operand is a csv file.

See also: [csv_file](#),

is_dxf

Possible uses:

- `is_dxf (any)` \rightarrow `bool`

Result:

Tests whether the operand is a dxf file.

See also: [dxf_file](#),

is_error

Possible uses:

- `is_error` (any expression) ---> `bool`

Result:

Returns whether or not the argument raises an error when evaluated

is_finite

Possible uses:

- `is_finite` (float) ---> `bool`

Result:

Returns whether the argument is a finite number or not

Examples:

```
bool var0 <- is_finite(4.66); // var0 equals true  
bool var1 <- is_finite(#infinity); // var1 equals false
```

is_gaml

Possible uses:

- `is_gaml` (any) ---> `bool`

Result:

Tests whether the operand is a gaml file.

See also: [gaml_file](#),

is_geojson

Possible uses:

- `is_geojson` (any) ---> `bool`

Result:

Tests whether the operand is a geojson file.

See also: [geojson_file](#),

is_gif

Possible uses:

- `is_gif` (any) ---> bool

Result:

Tests whether the operand is a gif file.

See also: [gif_file](#),

is_gml

Possible uses:

- `is_gml` (any) ---> bool

Result:

Tests whether the operand is a gml file.

See also: [gml_file](#),

is_graph6

Possible uses:

- `is_graph6` (any) ---> bool

Result:

Tests whether the operand is a graph6 file.

See also: [graph6_file](#),

is_graphdimacs

Possible uses:

- `is_graphdimacs` (any) ---> bool

Result:

Tests whether the operand is a graphdimacs file.

See also: [graphdimacs_file](#),

is_graphdot

Possible uses:

- `is_graphdot` (any) ---> bool

Result:

Tests whether the operand is a graphdot file.

See also: [graphdot_file](#),

is_graphgexf

Possible uses:

- `is_graphgexf`(any) ---> bool

Result:

Tests whether the operand is a graphgexf file.

See also: [graphgexf_file](#),

is_graphgml

Possible uses:

- `is_graphgml`(any) ---> bool

Result:

Tests whether the operand is a graphgml file.

See also: [graphgml_file](#),

is_graphml

Possible uses:

- `is_graphml`(any) ---> bool

Result:

Tests whether the operand is a graphml file.

See also: [graphml_file](#),

is_graphtsplib

Possible uses:

- `is_graphtsplib`(any) ---> bool

Result:

Tests whether the operand is a graphtsplib file.

See also: [graphtsplib_file](#),

is_grid

Possible uses:

- `is_grid` (any) --> bool

Result:

Tests whether the operand is a grid file.

See also: [grid_file](#),

is_image

Possible uses:

- `is_image` (any) --> bool

Result:

Tests whether the operand is a image file.

See also: [image_file](#),

is_json

Possible uses:

- `is_json` (any) --> bool

Result:

Tests whether the operand is a json file.

See also: [json_file](#),

is_number

Possible uses:

- `is_number` (float) --> bool
- `is_number` (string) --> bool

Result:

Returns whether the argument is a real number or not tests whether the operand represents a numerical value

Comment:

Note that the symbol . should be used for a float value (a string with , will not be considered as a numeric value). Symbols e and E are also accepted. A hexadecimal value should begin with #.

Examples:

```
bool var0 <- is_number(4.66); // var0 equals true
```

is_obj

Possible uses:

- `is_obj` (any) ---> bool

Result:

Tests whether the operand is a obj file.

See also: [obj_file](#),

is_osm

Possible uses:

- `is_osm` (any) ---> bool

Result:

Tests whether the operand is a osm file.

See also: [osm_file](#),

is_pgm

Possible uses:

- `is_pgm` (any) ---> bool

Result:

Tests whether the operand is a pgm file.

See also: [pgm_file](#),

is_property

Possible uses:

- `is_property` (any) ---> bool

Result:

Tests whether the operand is a property file.

See also: [property_file](#),

is_reachable

Possible uses:

- `string is_reachable int` ---> bool
- `is_reachable(string, int)` ---> bool

- `is_reachable` (`string`, `int`, `int`) \rightarrow `bool`

Result:

Returns whether or not the given web address is reachable or not before a time_out time in milliseconds Returns whether or not the given web address is reachable or not before a time_out time in milliseconds

Examples:

```
write sample(is_reachable("www.google.com", 200));
write sample(is_reachable("www.google.com", 200));
```

is_saved_simulation

Possible uses:

- `is_saved_simulation` (`any`) \rightarrow `bool`

Result:

Tests whether the operand is a saved_simulation file.

See also: [saved_simulation_file](#),

is_shape

Possible uses:

- `is_shape` (`any`) \rightarrow `bool`

Result:

Tests whether the operand is a shape file.

See also: [shape_file](#),

is_skill

Possible uses:

- `unknown is_skill string` \rightarrow `bool`
- `is_skill (unknown, string)` \rightarrow `bool`

Result:

returns true if the left operand is an agent whose species implements the right-hand skill name

Examples:

```
bool var0 <- agentA is_skill 'moving'; // var0 equals true
```

is_svg

Possible uses:

- `is_svg` (`any`) --> `bool`

Result:

Tests whether the operand is a svg file.

See also: [svg_file](#),

is_text

Possible uses:

- `is_text` (`any`) --> `bool`

Result:

Tests whether the operand is a text file.

See also: [text_file](#),

is_threeds

Possible uses:

- `is_threeds` (`any`) --> `bool`

Result:

Tests whether the operand is a threeds file.

See also: [threeds_file](#),

is_warning

Possible uses:

- `is_warning` (`any expression`) --> `bool`

Result:

Returns whether or not the argument raises a warning when evaluated

is_xml

Possible uses:

- `is_xml` (`any`) --> `bool`

Result:

Tests whether the operand is a xml file.

See also: [xml_file](#),

json_file

Possible uses:

- `json_file (string) ---> file`
- `string json_file map<string,unknown> ---> file`
- `json_file (string , map<string,unknown>) ---> file`

Result:

Constructs a file of type json. Allowed extensions are limited to json

Special cases:

- `json_file(string)`: This file constructor allows to read a json file

```
file f <- json_file("file.json");
```

- `json_file(string,map<string,unknown>)`: This constructor allows to store a map in a json file (it does not save it). The file can then be saved later using the `save` statement

```
file f <- json_file("file.json", map(["var1":1.0, "var2":3.0]));
```

See also: [is_json](#),

k_nearest_neighbors

Possible uses:

- `k_nearest_neighbors (agent, map<agent,unknown>, int) ---> unknown`

Result:

This operator allows user to find the attribute of an agent basing on its k-nearest agents

Comment:

In order to use this operator, users have to create a map which map the agents with one of their attributes (for example color or size,...). In the example below, 'map' is the map that I mention above, 'k' is the number of the nearest agents that we are considering

Examples:

```
unknown var0 <- self k_nearest_neighbors (map,k); // var0 equals this will return the attribute which has highest frequency  
in the k-nearest neighbors of our agent
```

k_spanning_tree_clustering

Possible uses:

- `graph k_spanning_tree_clustering int ---> list`
- `k_spanning_tree_clustering (graph , int)---> list`

Result:

The algorithm finds a minimum spanning tree T using Prim's algorithm, then executes Kruskal's algorithm only on the edges of T until k trees are formed. The resulting trees are the final clusters. It returns a list of list of vertices and takes as operand the graph and the number of clusters

kappa

Possible uses:

- `kappa ([list<unknown>, list<unknown>, list<unknown>] ---> float)`
- `kappa ([list<unknown>, list<unknown>, list<unknown>, list<unknown>] ---> 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 0.3333333333333334  
float var2 <- kappa([1,1,1,1,5],[1,1,1,1,5],[1,3,5]); // var2 equals 1.0  
float var3 <- kappa(["cat1", "cat3", "cat2", "cat1", "cat3"], ["cat1", "cat3", "cat2", "cat3", "cat1"], ["cat1", "cat2", "cat3"], [1.0, 2.0, 3.0, 1.0, 5.0]); // var3 equals 0.29411764705882354
```

kappa_sim

Possible uses:

- `kappa_sim ([list<unknown>, list<unknown>, list<unknown>, list<unknown>] ---> float)`
- `kappa_sim ([list<unknown>, list<unknown>, list<unknown>, list<unknown>, list<unknown>] ---> float)`

Result:

kappa simulation indicator for 2 map comparisons: `kappa(list_valsInits, list_valsObs, list_valsSim, categories)`. Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8).

Special cases:

- `kappa_sim` can be used with an additional weights operand

```
float var1 <-  
kappa_sim(["cat1", "cat1", "cat2", "cat2", "cat2"], ["cat1", "cat3", "cat2", "cat1", "cat3"], ["cat1", "cat3", "cat2", "cat3", "cat1"], ["cat1", "cat2", "cat3"], [1.0, 2.0, 3.0, 1.0, 5.0]); // var1 equals 0.2702702702702703
```

Examples:

```
float var0 <-  
kappa_sim(["cat1", "cat1", "cat2", "cat2", "cat2"], ["cat1", "cat3", "cat2", "cat1", "cat3"], ["cat1", "cat3", "cat2", "cat3", "cat1"], ["cat1", "cat2", "cat3"], // var0 equals 0.3333333333333335
```

kmeans

Possible uses:

- `list kmeans int` ---> `list<list>`
- `kmeans (list , int)`---> `list<list>`
- `kmeans (list, int, int)`---> `list<list>`

Result:

returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k) and the maximum number of iterations to run the algorithm.(If negative, no maximum will be used) (maxIt). Usage: kmeans(data,k,maxit)

Special cases:

- The maximum number of (third operand) can be omitted.

```
list<list> var0 <- kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2); // var0 equals [[0,2,3],[1]]
```

Examples:

```
list<list> var1 <- kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2,10); // var1 equals [[0,2,3],[1]]
```

kml

Possible uses:

- `kml (any)`---> `kml`

Result:

casts the operand in a kml object.

kurtosis

Possible uses:

- `kurtosis (list)`---> `float`
- `float kurtosis float`---> `float`
- `kurtosis (float , float)`---> `float`

Result:

Returns the kurtosis from a moment and a standard deviation Returns the kurtosis (aka excess) of a list of values (kurtosis = { $[n(n+1) / (n - 1)(n - 2)(n - 3)] \sum[(x_i - \text{mean})^4] / \text{std}^4$ } - $[3(n-1)^2 / (n-2)(n-3)]$)

Special cases:

- if the length of the list is lower than 3, returns NaN

Examples:

```
float var0 <- kurtosis(3,12) with_precision(4); // var0 equals -2.9999
```

label_propagation_clustering

Possible uses:

- `graph label_propagation_clustering int --> list`
- `label_propagation_clustering (graph , int) --> list`

Result:

The algorithm is a near linear time algorithm capable of discovering communities in large graphs. It is described in detail in the following:
Raghavan, U. N., Albert, R., and Kumara, S. (2007). Near linear time algorithm to detect

- community structures in large-scale networks. Physical review E, 76(3), 036106. It returns a list of list of vertices and takes as operand the graph and maximal number of iteration

last

Possible uses:

- `last (container<KeyType,ValueType>) --> ValueType`
- `last (string) --> string`
- `int last container --> list`
- `last (int , 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 list, last returns the last element of the list, or nil if the list is empty

```
int var0 <- last ([1, 2, 3]); // var0 equals 3
```

- if it is a string, last returns a string composed of its last character, or an empty string if the operand is empty

```
string var1 <- last ('abce'); // var1 equals 'e'
```

See also: [first](#),

last_index_of

Possible uses:

- `species last_index_of unknown ---> int`
- `last_index_of (species , unknown)---> int`
- `list last_index_of unknown ---> int`
- `last_index_of (list , unknown)---> int`
- `matrix last_index_of unknown ---> point`
- `last_index_of (matrix , unknown)---> point`
- `map<unknown, unknown> last_index_of unknown ---> unknown`
- `last_index_of (map<unknown, unknown> , unknown)---> unknown`
- `string last_index_of string ---> int`
- `last_index_of (string , string)---> int`

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 list, `last_index_of` returns the index as an integer

```
int var0 <- [1,2,3,4,5,6] last_index_of 4; // var0 equals 3  
int var1 <- [4,2,3,4,5,4] last_index_of 4; // var1 equals 5
```

- if the left operand is a matrix, `last_index_of` returns the index as a point

```
point var2 <- matrix([[1,2,3],[4,5,4]]) last_index_of 4; // var2 equals {1.0,2.0}
```

- if the left operand is a map, `last_index_of` returns the index as an int (the key of the pair)

```
unknown var3 <- [1::2, 3::4, 5::4] last_index_of 4; // var3 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 var4 <- "ababcabc" last_index_of "ca"; // var4 equals 5
```

See also: [at](#), [index_of](#), [last_index_of](#),

last_of

Same signification as [last](#)

last_with

Possible uses:

- `container last_with any expression --> unknown`
- `last_with(container, any expression) --> unknown`

Result:

the last element of the left-hand operand that makes the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, `last_with` throws an error.
- If there is no element that satisfies the condition, it returns nil
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] last_with (each >= 4); // var4 equals 6
unknown var5 <- [1::2, 3::4, 5::6].pairs last_with (each.value >= 4); // var5 equals (5::6)
```

Examples:

```
int 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 a node
unknown var3 <- (list(node)) last_with (round(node(each).location.x) > 32); // var3 equals node3
```

See also: [group_by](#), [first_with](#), [where](#),

layout_circle

Possible uses:

- `layout_circle(graph, geometry, bool) --> graph`

Result:

layouts a Gama graph on a circle with equidistance between nodes. For now there is no optimization on node ordering.

Special cases:

- Usage: `layoutCircle(graph, bound, shuffle) => graph` : the graph to layout, `bound` : the geometry to display the graph within, `shuffle` : if true shuffle the nodes, then render same ordering

Examples:

```
layout_circle(graph, world.shape, false);
```

layout_force

Possible uses:

- `layout_force` (`graph`, `geometry`, `float`, `float`, `int`) --> `graph`
- `layout_force` (`graph`, `geometry`, `float`, `float`, `int`, `float`) --> `graph`

Result:

layouts a GAMA graph using Force model (in a given spatial bound and given coeff_force, cooling_rate, max_iteration, and equilibirum criterion parameters).

Special cases:

- usage: `layoutForce(graph, bounds, coeff_force, cooling_rate, max_iteration)`. graph is the graph to which applied the layout; bounds is the shape (geometry) in which the graph should be located; coeff_force is the coefficient used to compute the force, typical value is 0.4; cooling rate is the decreasing coefficient of the temperature, typical value is 0.01; max_iteration is the maximal number of iterationsdistance of displacement for a vertice to be considered as in equilibrium
- usage: `layoutForce(graph, bounds, coeff_force, cooling_rate, max_iteration, equilibrium_criterion)`. graph is the graph to which applied the layout; bounds is the shape (geometry) in which the graph should be located; coeff_force is the coefficient used to compute the force, typical value is 0.4; cooling rate is the decreasing coefficient of the temperature, typical value is 0.01; max_iteration is the maximal number of iterations; equilibrium criterion is the maximaldistance of displacement for a vertice to be considered as in equilibrium

layout_force_FR

Possible uses:

- `layout_force_FR` (`graph`, `geometry`, `float`, `int`) --> `graph`

Result:

layouts a GAMA graph using Fruchterman and Reingold Force-Directed Placement Algorithm (in a given spatial bound, normalization factor and max_iteration parameters).

Special cases:

- usage: `layoutForce(graph, bounds, normalization_factor, max_iteration, equilibrium_criterion)`. graph is the graph to which applied the layout; bounds is the shape (geometry) in which the graph should be located; normalization_factor is the normalization factor for the optimal distance, typical value is 1.0; max_iteration is the maximal number of iterations

layout_force_FR_indexed

Possible uses:

- `layout_force_FR_indexed` (`graph`, `geometry`, `float`, `float`, `int`) --> `graph`

Result:

layouts a GAMA graph using Fruchterman and Reingold Force-Directed Placement Algorithm with The Barnes-Hut indexing technique(in a given spatial bound, theta, normalization factor and max_iteration parameters).

Special cases:

- usage: `layoutForce(graph, bounds, normalization_factor, max_iteration, equilibrium_criterion)`. graph is the graph to which applied the layout; bounds is the shape (geometry) in which the graph should be located; theta value for approximation using the Barnes-Hut technique, typical value is 0.5; normalization_factor is the normalization factor for the optimal distance, typical value is 1.0; max_iteration is the maximal number

of iterations

layout_grid

Possible uses:

- `layout_grid (graph, geometry, float) ---> graph`

Result:

layouts a Gama graph based on a grid lattice. usage: `layoutForce(graph, bounds, coeff_nb_cells)`. graph is the graph to which the layout is applied; bounds is the shape (geometry) in which the graph should be located; coeff_nb_cellsthe coefficient for the number of cells to locate the vertices (nb of places = coeff_nb_cells * nb of vertices).

Examples:

```
layout_grid(graph, world.shape);
```

length

Possible uses:

- `length (string) ---> int`
- `length (container<KeyType,ValueType>) ---> 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 string, length returns the number of characters

```
int var0 <- length ("I am an agent"); // var0 equals 13
```

- if it is a list or a map, length returns the number of elements in the list or map

```
int var1 <- length([12,13]); // var1 equals 2
int var2 <- length([]); // var2 equals 0
```

- if it is a matrix, length returns the number of cells

```
int var3 <- length(matrix([[ "c11", "c12", "c13"], [ "c21", "c22", "c23"]])); // var3 equals 6
```

lgamma

Same signification as `log_gamma`

line

Possible uses:

- `line` (`container<unknown,geometry>`) \rightarrow `geometry`
- `container<unknown,geometry>` `line` `float` \rightarrow `geometry`
- `line` (`container<unknown,geometry>`, `float`) \rightarrow `geometry`

Result:

A polyline geometry from the given list of points.

Special cases:

- if the points list operand is nil, returns the point geometry {0,0}
- if the points list operand is composed of a single point, returns a point geometry.
- if a radius is added, the given list of points represented as a cylinder of radius r

```
geometry var0 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}],0.2); // var0 equals a polyline geometry composed of the 4 points.
```

Examples:

```
geometry var1 <- polyline([{0,0}, {0,10}, {10,10}]); // var1 equals a polyline geometry composed of the 3 points.  
geometry var2 <- line([{10,10}, {10,0}]); // var2 equals a line from 2 points.  
string var3 <- string(polyline([{0,0}, {0,10}, {10,10}])+line([{10,10}, {10,0}])); // var3 equals "MULTILINESTRING ((0 0, 0 10, 10 10), (10 10, 10 0))"
```

See also: [around](#), [circle](#), [cone](#), [link](#), [norm](#), [point](#), [polygone](#), [rectangle](#), [square](#), [triangle](#),

link

Possible uses:

- `geometry link geometry` \rightarrow `geometry`
- `link` (`geometry`, `geometry`) \rightarrow `geometry`

Result:

A dynamic line geometry between the location of the two operands

Comment:

The geometry of the link is a line between the locations of the two operands, which is built and maintained dynamically

Special cases:

- if one of the operands is nil, link returns a point geometry at the location of the other. If both are null, it returns a point geometry at {0,0}

Examples:

```
geometry var0 <- link (geom1,geom2); // var0 equals a link geometry between geom1 and geom2.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

list

Possible uses:

- `list` (`any`) ---> `list`

Result:

casts the operand in a list object.

list_with

Possible uses:

- `int` `list_with` (`any expression`) ---> `list`
- `list_with` (`int` , `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 first operand should be positive, and that the second one is evaluated for each position in the list.

Examples:

```
list var0 <- list_with(5,2); // var0 equals [2,2,2,2,2]
```

See also: [list](#),

ln

Possible uses:

- `ln` (`int`) ---> `float`
- `ln` (`float`) ---> `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(1); // var0 equals 0.0
float var1 <- ln(exp(1)); // var1 equals 1.0
```

See also: [exp](#),

load_shortest_paths

Possible uses:

- graph load_shortest_paths matrix --> graph
- load_shortest_paths (graph , matrix) --> graph

Result:

put in the graph cache the computed shortest paths contained in the matrix (rows: source, columns: target)

Examples:

```
graph var0 <- load_shortest_paths(shortest_paths_matrix); // var0 equals return my_graph with all the shortest paths computed
```

load_sub_model

Possible uses:

- string load_sub_model string --> agent
- load_sub_model (string , string) --> agent

Result:

Load a submodel

Comment:

loaded submodel

log

Possible uses:

- log (float) --> float
- log (int) --> float

Result:

Returns the logarithm (base 10) of the operand.

Special cases:

- an exception is raised if the operand is equals or less than zero.

Examples:

```
float var0 <- log(10); // var0 equals 1.0
float var1 <- log(1); // var1 equals 0.0
```

See also: [ln](#),

log_gamma

Possible uses:

- `log_gamma` (`float`) ---> `float`

Result:

Returns the log of the value of the Gamma function at x.

Examples:

```
float var0 <- log_gamma(0.6) with_precision(4); // var0 equals 0.3982
```

lognormal_density

Possible uses:

- `lognormal_density` (`float`, `float`, `float`) ---> `float`

Result:

`lognormal_density(x,shape,scale)` returns the probability density function (PDF) at the specified point x of the logNormal distribution with the given shape and scale.

Examples:

```
float var0 <- lognormal_density(1,2,3) ; // var0 equals 0.731
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [poisson_rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#), [weibull_density](#), [gamma_density](#),

lognormal_rnd

Possible uses:

- `float lognormal_rnd float` ---> `float`
- `lognormal_rnd` (`float`, `float`) ---> `float`

Result:

returns a random value from a Log-Normal distribution with specified values of the shape (alpha) and scale (beta) parameters. See https://en.wikipedia.org/wiki/Log-normal_distribution for more details.

Examples:

```
float var0 <- lognormal_rnd(2,3); // var0 equals 0.731
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [poisson_rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#), [lognormal_trunc_rnd](#),

lognormal_trunc_rnd

Possible uses:

- `lognormal_trunc_rnd` (`float`, `float`, `float`, `float`) ---> `float`
- `lognormal_trunc_rnd` (`float`, `float`, `float`, `bool`) ---> `float`

Result:

returns a random value from a truncated Log-Normal distribution (in a range or given only one boundary) with specified values of the shape (alpha) and scale (beta) parameters. See https://en.wikipedia.org/wiki/Log-normal_distribution for more details.

Special cases:

- when 2 float operands are specified, they are taken as minimum and maximum values for the result

```
lognormal_trunc_rnd(2,3,0,5)
```

- when 1 float and a boolean (isMax) operands are specified, the float value represents the single boundary (max if the boolean is true, min otherwise),

```
lognormal_trunc_rnd(2,3,5,true)
```

See also: [lognormal_rnd](#), [gamma_trunc_rnd](#), [weibull_trunc_rnd](#), [truncated_gauss](#),

lower_case

Possible uses:

- `lower_case` (`string`) ---> `string`

Result:

Converts all of the characters in the string operand to lower case

Examples:

```
string var0 <- lower_case("Abc"); // var0 equals 'abc'
```

See also: [upper_case](#),

main_connected_component

Possible uses:

- `main_connected_component` (`graph`) ---> `graph`

Result:

returns the sub-graph corresponding to the main connected components of the graph

Examples:

```
graph var0 <- main_connected_component(my_graph); // var0 equals the sub-graph corresponding to the main connected components of the graph
```

See also: [connected_components_of](#),

map

Possible uses:

- `map` (`any`) ---> `map`

Result:

casts the operand in a map object.

masked_by

Possible uses:

- `geometry` `masked_by` `container<unknown,geometry>` ---> `geometry`
- `masked_by` (`geometry`, `container<unknown,geometry>`) ---> `geometry`
- `masked_by` (`geometry`, `container<unknown,geometry>`, `int`) ---> `geometry`

Examples:

```
geometry var0 <- perception_geom masked_by obstacle_list; // var0 equals the geometry representing the part of perception_geom visible from the agent position considering the list of obstacles obstacle_list.  
geometry var1 <- perception_geom masked_by obstacle_list; // var1 equals the geometry representing the part of perception_geom visible from the agent position considering the list of obstacles obstacle_list.
```

matrix

Possible uses:

- `matrix` (`any`) ---> `matrix`

Result:

casts the operand in a matrix object.

matrix

Possible uses:

- `matrix` (`image`) ---> `matrix`

Result:

Returns the `matrix<int>` value of the image passed in parameter, where each pixel is represented by the RGB int value. The dimensions of the matrix are those of the image.

matrix_with

Possible uses:

- point matrix_with any expression --> matrix
- matrix_with (point, any expression) --> matrix

Result:

creates a matrix with a size provided by the first operand, and filled with the second operand. The given expression, unless constant, is evaluated for each cell

Comment:

Note that both components of the right operand point should be positive, otherwise an exception is raised.

See also: [matrix](#), [as_matrix](#),

max

Possible uses:

- max (container) --> unknown

Result:

the maximum element found in the operand

Comment:

the max operator behavior depends on the nature of the operand

Special cases:

- if it is a population of a list of other type: max transforms all elements into integer and returns the maximum of them
- if it is a map, max returns the maximum among the list of all elements value
- if it is a file, max returns the maximum of the content of the file (that is also a container)
- if it is a graph, max returns the maximum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, max returns the maximum of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, max returns the maximum of the list of the geometries
- if it is a matrix of another type, max returns the maximum of the elements transformed into float
- if it is a list of int 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](#),

max_flow_between

Possible uses:

- `max_flow_between` (`graph`, `unknown`, `unknown`) ---> `map<unknown, float>`

Result:

The max flow (`map<edge,flow>` in a graph between the source and the sink using Edmonds-Karp algorithm

Examples:

```
max_flow_between(my_graph, vertice1, vertice2)
```

max_of

Possible uses:

- `container` `max_of` `any expression` ---> `unknown`
- `max_of` (`container`, `any expression`)---> `unknown`

Result:

the maximum value of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword `each` can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- As of GAMA 1.6, if the left-hand operand is nil or empty, `max_of` throws an error
- if the left-operand is a map, the keyword `each` will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] max_of (each + 3); // var4 equals 9
```

Examples:

```
unknown var0 <- [1,2,4,3,5,7,6,8] max_of (each * 100); // var0 equals 800
graph g2 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
unknown var2 <- g2.vertices max_of (g2.degree_of( each )); // var2 equals 2
unknown var3 <- (list(node)) max_of (round(node(each).location.x)); // var3 equals 96
```

See also: [min_of](#),

maximal_cliques_of

Possible uses:

- `maximal_cliques_of` (`graph`) ---> `list<list>`

Result:

returns the maximal cliques of a graph using the Bron-Kerbosch clique detection algorithm: A clique is maximal if it is impossible to enlarge it by adding another vertex from the graph. Note that a maximal clique is not necessarily the biggest clique in the graph.

Examples:

```
graph my_graph <- graph([]);  
list<list> var1 <- maximal_cliques_of (my_graph); // var1 equals the list of all the maximal cliques as list
```

See also: [biggest_cliques_of](#),

mean

Possible uses:

- `mean` (`container`) --> `unknown`

Result:

the mean of all the elements of the operand

Comment:

the elements of the operand are summed (see `sum` for more details about the sum of container elements) and then the sum value is divided by the number of elements.

Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

Examples:

```
unknown var0 <- mean ([4.5, 3.5, 5.5, 7.0]); // var0 equals 5.125
```

See also: [sum](#),

mean_deviation

Possible uses:

- `mean_deviation` (`container`) --> `float`

Result:

the deviation from the mean of all the elements of the operand. See [Mean_deviation](#) for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- mean_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.125
```

See also: [mean](#), [standard_deviation](#),

mean_of

Possible uses:

- `container mean_of any_expression ---> unknown`
- `mean_of (container, any_expression) ---> unknown`

Result:

the mean of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] mean_of (each); // var1 equals 4
```

Examples:

```
unknown var0 <- [1,2] mean_of (each * 10 ); // var0 equals 15
```

See also: [min_of](#), [max_of](#), [sum_of](#), [product_of](#),

median

Possible uses:

- `median (container) ---> unknown`

Result:

the median of all the elements of the operand.

Special cases:

- if the container contains points, the result will be a point. If the container contains rgb values, the result will be a rgb color

Examples:

```
unknown var0 <- median ([4.5, 3.5, 5.5, 3.4, 7.0]); // var0 equals 4.5
```

See also: [mean](#),

mental_state

Possible uses:

- `mental_state (any) ---> mental_state`

Result:

casts the operand in a mental_state object.

message**Possible uses:**

- `message` (`any`) --> `message`

Result:

casts the operand in a message object.

milliseconds_between**Possible uses:**

- `date milliseconds_between date` --> `float`
- `milliseconds_between (date, date)` --> `float`

Result:

Provide the exact number of milliseconds between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

Examples:

```
float var0 <- milliseconds_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 2.6784E9
```

min**Possible uses:**

- `min (container)` --> `unknown`

Result:

the minimum element found in the operand.

Comment:

the min operator behavior depends on the nature of the operand

Special cases:

- if it is a list of points: min returns the minimum of all points as a point (i.e. the point with the smallest coordinate on the x-axis, in case of equality the point with the smallest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)
- if it is a population of a list of other types: min transforms all elements into integer and returns the minimum of them
- if it is a map, min returns the minimum among the list of all elements value
- if it is a file, min returns the minimum of the content of the file (that is also a container)
- if it is a graph, min returns the minimum of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, min returns the minimum of all the numerical elements (thus all elements for integer and float matrices)

- if it is a matrix of geometry, min returns the minimum of the list of the geometries
- if it is a matrix of another type, min returns the minimum of the elements transformed into float
- if it is a list of int or float: min returns the minimum of all the elements

```
unknown var0 <- min ([100, 23.2, 34.5]); // var0 equals 23.2
```

See also: [max](#),

min_of

Possible uses:

- `container min_of any_expression` --> `unknown`
- `min_of (container, any_expression)` --> `unknown`

Result:

the minimum value of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil or empty, min_of throws an error
- if the left-operand is a map, the keyword each will contain each value

```
unknown var4 <- [1::2, 3::4, 5::6] min_of (each + 3); // var4 equals 5
```

Examples:

```
unknown var0 <- [1,2,4,3,5,7,6,8] min_of (each * 100 ); // var0 equals 100
graph g2 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
unknown var2 <- g2 min_of (length(g2 out_edges_of each) ); // var2 equals 0
unknown var3 <- (list(node) min_of (round(node(each).location.x))); // var3 equals 4
```

See also: [max_of](#),

minus_days

Possible uses:

- `date minus_days int` --> `date`
- `minus_days (date, int)` --> `date`

Result:

Subtract a given number of days from a date

Examples:

```
date var0 <- date('2000-01-01') minus_days 20; // var0 equals date('1999-12-12')
```

minus_hours

Possible uses:

- `date minus_hours int` ---> `date`
- `minus_hours (date, int)` ---> `date`

Result:

Remove a given number of hours from a date

Examples:

```
// equivalent to date1 - 15 #h
date var1 <- date('2000-01-01') minus_hours 15 ; // var1 equals date('1999-12-31 09:00:00')
```

minus_minutes

Possible uses:

- `date minus_minutes int` ---> `date`
- `minus_minutes (date, int)` ---> `date`

Result:

Subtract a given number of minutes from a date

Examples:

```
// date('2000-01-01') to date1 - 5#mn
date var1 <- date('2000-01-01') minus_minutes 5 ; // var1 equals date('1999-12-31 23:55:00')
```

minus_months

Possible uses:

- `date minus_months int` ---> `date`
- `minus_months (date, int)` ---> `date`

Result:

Subtract a given number of months from a date

Examples:

```
date var0 <- date('2000-01-01') minus_months 5; // var0 equals date('1999-08-01')
```

minus_ms

Possible uses:

- `date minus_ms int` ---> `date`

- `minus_ms` (`date`, `int`) \rightarrow `date`

Result:

Remove a given number of milliseconds from a date

Examples:

```
// equivalent to date1 - 15 #ms
date var1 <- date('2000-01-01') minus_ms 1000; // var1 equals date('1999-12-31 23:59:59')
```

minus_seconds

Same signification as -

minus_weeks

Possible uses:

- `date minus_weeks int` \rightarrow `date`
- `minus_weeks` (`date`, `int`) \rightarrow `date`

Result:

Subtract a given number of weeks from a date

Examples:

```
date var0 <- date('2000-01-01') minus_weeks 15; // var0 equals date('1999-09-18')
```

minus_years

Possible uses:

- `date minus_years int` \rightarrow `date`
- `minus_years` (`date`, `int`) \rightarrow `date`

Result:

Subtract a given number of year from a date

Examples:

```
date var0 <- date('2000-01-01') minus_years 3; // var0 equals date('1997-01-01')
```

mod

Possible uses:

- `int mod int` \rightarrow `int`
- `mod` (`int`, `int`) \rightarrow `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](#),

moment**Possible uses:**

- `moment(container, int, float) -> float`

Result:

Returns the moment of k-th order with constant c of a data sequence

Examples:

```
float var0 <- moment([13,2,1,4,1,2], 2, 1.2) with_precision(4); // var0 equals 24.74
```

months_between**Possible uses:**

- `date months_between date -> int`
- `months_between(date, date) -> int`

Result:

Provide the exact number of months between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

Examples:

```
int var0 <- months_between(date('2000-01-01'), date('2000-02-01')); // var0 equals 1
```

moran**Possible uses:**

- `list<float> moran matrix<float> -> float`
- `moran(list<float>, matrix<float>) -> float`

Special cases:

- return the Moran Index of the given list of interest points (list of floats) and the weight matrix (matrix of float)

```
float var0 <- moran([1.0, 0.5, 2.0], weight_matrix); // var0 equals the Moran index is computed
```

morrisAnalysis

Possible uses:

- `morrisAnalysis (string, int, int) ---> string`

Result:

Return a string containing the Report of the morris analysis for the corresponding CSV file

mul

Possible uses:

- `mul (container) ---> unknown`

Result:

the product of all the elements of the operand

Comment:

the mul operator behavior depends on the nature of the operand

Special cases:

- if it is a list of points: mul returns the product of all points as a point (each coordinate is the product of the corresponding coordinate of each element)
- if it is a list of other types: mul transforms all elements into integer and multiplies them
- if it is a map, mul returns the product of the value of all elements
- if it is a file, mul returns the product of the content of the file (that is also a container)
- if it is a graph, mul returns the product of the list of the elements of the graph (that can be the list of edges or vertexes depending on the graph)
- if it is a matrix of int, float or object, mul returns the product of all the numerical elements (thus all elements for integer and float matrices)
- if it is a matrix of geometry, mul returns the product of the list of the geometries
- if it is a matrix of other types: mul transforms all elements into float and multiplies them
- if it is a list of int or float: mul returns the product of all the elements

```
unknown var0 <- mul ([100, 23.2, 34.5]); // var0 equals 80040.0
```

See also: [sum](#),

Version: 1.9.1

Operators (N to R)

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, ..., operand)`) or a special infix syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any side effects on their operands. For instance, the `shuffle` operator, which randomizes the positions of elements in a list, does not modify its list operand but returns a new shuffled list.

Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely:

- the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values).
 - it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else).
 - next are the logical operators, `and` and `or` (e.g. `a > b or b > c` will return the value of the test)
 - next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`)
 - next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators)
 - next the unary operators `-` and `!`
 - next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points)
 - and finally the functional operators, which have the highest priority of all.
-

Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
    int min(int x, int y) {
        return x > y ? x : y;
    }
}
```

Any agent instance of `spec1` can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```

global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
    }
}

```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

`box, cone3D, cube, cylinder, hexagon, pyramid, set_z, sphere, teapot,`

Arithmetic operators

`-, /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,`

BDI

`add_values, and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get Obligations_op,`

```
get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention,  
get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op,  
get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op,  
has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op,  
has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op,  
has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or,  
set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime,  
set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_values,
```

Casting operators

```
as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of, to_gaml, to_geojson, to_list,  
with_size, with_style,
```

Color-related operators

```
-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean, median, palette, rgb, rnd_color,  
scale, sum, to_hsb,
```

Comparison operators

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

Containers-related operators

```
-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect, contains, contains_all,  
contains_any, contains_key, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave,  
internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul,  
none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, split,  
split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,
```

Date-related operators

```
-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours,  
minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours,  
plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,
```

Dates

Displays

horizontal, stack, vertical,

edge

edge_between, strahler,

EDP-related operators

diff, diff2,

Files-related operators

copy_file, crs, csv_file, delete_file, dxf_file, evaluate_sub_model, file_exists, folder, folder_exists, gamm_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphtsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphtsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,

GamaMetaType

type_of,

Gen*

add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,

Graphs-related operators

[add_edge](#), [add_node](#), [adjacency](#), [agent_from_geometry](#), [all_pairs_shortest_path](#), [alpha_index](#), [as_distance_graph](#), [as_edge_graph](#), [as_intersection_graph](#), [as_path](#), [as_spatial_graph](#), [beta_index](#), [betweenness_centrality](#), [biggest_cliques_of](#), [connected_components_of](#), [connectivity_index](#), [contains_edge](#), [contains_vertex](#), [degree_of](#), [directed](#), [edge](#), [edge_between](#), [edge_betweenness](#), [edges](#), [gamma_index](#), [generate_barabasi_albert](#), [generate_complete_graph](#), [generate_random_graph](#), [generate_watts_strogatz](#), [girvan_newman_clustering](#), [grid_cells_to_graph](#), [in_degree_of](#), [in_edges_of](#), [k_spanning_tree_clustering](#), [label_propagation_clustering](#), [layout_circle](#), [layout_force](#), [layout_force_FR](#), [layout_force_FR_indexed](#), [layout_grid](#), [load_shortest_paths](#), [main_connected_component](#), [max_flow_between](#), [maximal_cliques_of](#), [nb_cycles](#), [neighbors_of](#), [node](#), [nodes](#), [out_degree_of](#), [out_edges_of](#), [path_between](#), [paths_between](#), [predecessors_of](#), [remove_node_from](#), [rewire_n](#), [source_of](#), [spatial_graph](#), [strahler](#), [successors_of](#), [sum](#), [target_of](#), [undirected](#), [use_cache](#), [weight_of](#), [with_k_shortest_path_algorithm](#), [with_shortest_path_algorithm](#), [with_weights](#),

Grid-related operators

[as_4_grid](#), [as_grid](#), [as_hexagonal_grid](#), [cell_at](#), [cells_in](#), [cells_overlapping](#), [field](#), [grid_at](#), [neighbors_of](#), [path_between](#), [points_in](#), [values_in](#),

ImageOperators

[*](#), [antialiased](#), [blend](#), [blurred](#), [brighter](#), [clipped_with](#), [darker](#), [grayscale](#), [horizontal_flip](#), [image](#), [matrix](#), [rotated_by](#), [sharpened](#), [snapshot](#), [tinted_with](#), [vertical_flip](#), [with_height](#), [with_size](#), [with_width](#),

Iterator operators

[accumulate](#), [all_match](#), [as_map](#), [collect](#), [count](#), [create_map](#), [first_with](#), [frequency_of](#), [group_by](#), [index_by](#), [last_with](#), [max_of](#), [mean_of](#), [min_of](#), [none_matches](#), [one_matches](#), [product_of](#), [sort_by](#), [sum_of](#), [variance_of](#), [where](#), [where](#), [where](#), [with_max_of](#), [with_min_of](#),

List-related operators

[all_indexes_of](#), [copy_between](#), [index_of](#), [last_index_of](#),

Logical operators

`:!, ?, add_3Dmodel, add_geometry, add_icon, and, or, xor,`

Map comparaison operators

`fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,`

Map-related operators

`as_map, create_map, index_of, last_index_of,`

Matrix-related operators

`-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,`

multicriteria operators

`electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM, weighted_means_DM,`

Path-related operators

`agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, max_flow_between, path_between, path_to, paths_between, use_cache,`

Pedestrian

`generate_pedestrian_network,`

Points-related operators

`-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid, closest_points_with, farthest_point_to, grid_at, norm, points_along, points_at, points_on,`

Random operators

`binomial`, `flip`, `gamma_density`, `gamma_rnd`, `gamma_trunc_rnd`, `gauss`, `generate_terrain`, `lognormal_density`, `lognormal_rnd`, `lognormal_trunc_rnd`, `poisson`, `rnd`, `rnd_choice`, `sample`, `shuffle`, `skew_gauss`, `truncated_gauss`, `weibull_density`, `weibull_rnd`, `weibull_trunc_rnd`,

ReverseOperators

`restore_simulation`, `restore_simulation_from_file`, `save_simulation`, `serialize`, `serialize_agent`,

Shape

`arc`, `box`, `circle`, `cone`, `cone3D`, `cross`, `cube`, `curve`, `cylinder`, `ellipse`, `elliptical_arc`, `envelope`, `geometry_collection`, `hexagon`, `line`, `link`, `plan`, `polygon`, `polyhedron`, `pyramid`, `rectangle`, `sphere`, `square`, `squircle`, `teapot`, `triangle`,

Spatial operators

`-`, `*`, `+`, `add_point`, `agent_closest_to`, `agent_farthest_to`, `agents_at_distance`, `agents_covering`, `agents_crossing`, `agents_inside`, `agents_overlapping`, `agents_partially_overlapping`, `agents_touching`, `angle_between`, `any_location_in`, `arc`, `around`, `as_4_grid`, `as_driving_graph`, `as_grid`, `as_hexagonal_grid`, `at_distance`, `at_location`, `box`, `centroid`, `circle`, `clean`, `clean_network`, `closest_points_with`, `closest_to`, `cone`, `cone3D`, `convex_hull`, `covering`, `covers`, `cross`, `crosses`, `crossing`, `crs`, `CRS_transform`, `cube`, `curve`, `cylinder`, `direction_between`, `disjoint_from`, `distance_between`, `distance_to`, `ellipse`, `elliptical_arc`, `envelope`, `farthest_point_to`, `farthest_to`, `geometry_collection`, `gini`, `hexagon`, `hierarchical_clustering`, `IDW`, `inside`, `inter`, `intersects`, `inverse_rotation`, `k_nearest_neighbors`, `line`, `link`, `masked_by`, `moran`, `neighbors_at`, `neighbors_of`, `normalized_rotation`, `overlapping`, `overlaps`, `partially_overlapping`, `partially_overlaps`, `path_between`, `path_to`, `plan`, `points_along`, `points_at`, `points_on`, `polygon`, `polyhedron`, `pyramid`, `rectangle`, `rotated_by`, `rotation_composition`, `round`, `scaled_to`, `set_z`, `simple_clustering_by_distance`, `simplification`, `skeletonize`, `smooth`, `sphere`, `split_at`, `split_geometry`, `split_lines`, `square`, `squircle`, `teapot`, `to_GAMA_CRS`, `to_rectangles`, `to_segments`, `to_squares`, `to_sub_geometries`, `touches`, `touching`, `towards`, `transformed_by`, `translated_by`, `triangle`, `triangulate`, `union`, `using`, `voronoi`, `with_precision`, `without_holes`,

Spatial properties operators

`covers`, `crosses`, `intersects`, `partially_overlaps`, `touches`,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, at_distance, closest_to, covering, crossing, farthest_to, inside, neighbors_at, neighbors_of, overlapping, partially_overlapping, touching,

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

Spatial statistical operators

hierarchical_clustering, k_nearest_neighbors, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, inverse_rotation, normalized_rotation, rotated_by, rotation_composition, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum, build, chi_square, chi_square_complemented, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis, log_gamma, max, mean, mean_deviation, median, min, moment, moran, morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, t_test, variance,

Strings-related operators

+,<,<=,>,>=,at,capitalize,char,contains,contains_all,contains_any,copy_between,date,empty,first,in,indented_by,index_of,is_number,last,last_index_of,length,lower_case,regex_matches,replace,replace_regex,reverse,sample,shuffle,split_with,string,upper_case,

SubModel

load_sub_model,

System

.,choose,command,copy,copy_from_clipboard,copy_to_clipboard,copy_to_clipboard,dead,enter,eval_gaml,every,is_error,is_reachable,is_warning,play_sound,user_confirm,user_input_dialog,wizard,wizard_page,

Time-related operators

date,string,

Types-related operators

action,agent,attributes,BDIPlan,bool,container,conversation,directory,emotion,file,float,gaml_type,gen_population_generator,gen_range,geometry,graph,int,kml,list,map,matrix,mental_state,message,Norm,pair,path,point,predicate,regression,rgb,Sanction,skill,social_link,species,topology,unknown,

User control operators

choose,enter,user_confirm,user_input_dialog,wizard,wizard_page,

Operators

nb_cycles

Possible uses:

- `nb_cycles` (`graph`) ---> `int`

Result:

returns the maximum number of independent cycles in a graph. This number (u) is estimated through the number of nodes (v), links (e) and of sub-graphs (p): $u = e - v + p$.

Examples:

```
graph graphEpidemio <- graph([]);  
int var1 <- nb_cycles(graphEpidemio); // var1 equals the number of cycles in the graph
```

See also: [alpha_index](#), [beta_index](#), [gamma_index](#), [connectivity_index](#),

neighbors_at

Possible uses:

- `geometry neighbors_at float` ---> `list`
- `neighbors_at` (`geometry` , `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 neighborhood 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 neighbors_at (10)); // var0 equals all the agents located at a distance  
lower or equal to 10 to the agent applying the operator.
```

See also: [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [agents_inside](#), [agent_closest_to](#), [at_distance](#),

neighbors_of

Possible uses:

- topology **neighbors_of** agent ---> list
- **neighbors_of** (topology , agent)---> list
- field **neighbors_of** point ---> list<point>
- **neighbors_of** (field , point)---> list<point>
- graph **neighbors_of** unknown ---> list
- **neighbors_of** (graph , unknown)---> list
- **neighbors_of** (topology , geometry , float)---> list

Result:

a list, containing all the agents of the same species than the argument (if it is an agent) located at a distance inferior or equal to 1 to the right-hand operand agent considering the left-hand operand topology.

Special cases:

- a list, containing all the agents of the same species than the left argument (if it is an agent) located at a distance inferior or equal to the third argument to the second argument (agent, geometry or point) considering the first operand topology.

```
list var0 <- neighbors_of (topology(self), self,10); // var0 equals all the agents located at a  
distance lower or equal to 10 to the agent applying the operator considering its topology.
```

Examples:

```
list var1 <- topology(self) neighbors_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 neighbors_of (node(3)); // var2 equals [node0,node2]  
list var3 <- graphFromMap neighbors_of node({12,45}); // var3 equals [{1.0,5.0},{34.0,56.0}]
```

See also: [neighbors_at](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [agents_inside](#), [agent_closest_to](#), [predecessors_of](#), [successors_of](#),

new_emotion

Possible uses:

- **new_emotion** (string)---> emotion

- `string new_emotion predicate ---> emotion`
- `new_emotion (string , predicate)---> emotion`
- `string new_emotion agent ---> emotion`
- `new_emotion (string , agent)---> emotion`
- `string new_emotion float ---> emotion`
- `new_emotion (string , float)---> emotion`
- `new_emotion (string , predicate , agent)---> emotion`
- `new_emotion (string , float , agent)---> emotion`
- `new_emotion (string , float , float)---> emotion`
- `new_emotion (string , float , predicate)---> emotion`
- `new_emotion (string , float , float , agent)---> emotion`
- `new_emotion (string , float , predicate , agent)---> emotion`
- `new_emotion (string , float , predicate , float)---> emotion`
- `new_emotion (string , float , predicate , float , agent)---> emotion`

Result:

a new emotion with the given properties (at least its name, and eventually intensity, parameters...)

Special cases:

- a new emotion with a given name and the predicate it is about

```
new_emotion("joy",estFood)
new_emotion("joy",agent1)
```

- a new emotion with a given name and the agent which has caused this emotion

```
new_emotion("joy",agent1)
```

- A decay value value can be added to define a new emotion.

```
new_emotion("joy",12.3,4.0)
```

- a new emotion with a name and an initial intensity:

```
new_emotion("joy",12.3)
```

- Various combinations are possible to create the emotion: (name,intensity,about), (name,about,cause),

(name,intensity,cause)...

```
new_emotion("joy",12.3, eatFood)
new_emotion("joy", eatFood, agent1)
new_emotion("joy", 12.3, agent1)
```

Examples:

```
emotion("joy",12.3, eatFood, 4, agent1)
emotion("joy", 12.3, 4, agent1)
new_emotion("joy",12.3, eatFood, agent1)
new_emotion("joy")
new_emotion("joy",12.3, eatFood, 4.0)
```

new_folder

Possible uses:

- `new_folder` (string) ---> file

Result:

opens an existing repository or create a new folder if it does not exist.

Special cases:

- If the specified string does not refer to an existing repository, the repository is created.
- If the string refers to an existing file, an exception is risen.

Examples:

```
file dirNewT <- new_folder("incl/"); // dirNewT represents the repository "../incl/"
                                         // eventually creates the directory
.. /incl
```

See also: [folder](#), [file](#), [folder_exists](#),

new_mental_state

Possible uses:

- `new_mental_state` (string) ---> mental_state

- `string new_mental_state mental_state ---> mental_state`
- `new_mental_state(string , mental_state)---> mental_state`
- `string new_mental_state emotion ---> mental_state`
- `new_mental_state(string , emotion)---> mental_state`
- `string new_mental_state predicate ---> mental_state`
- `new_mental_state(string , predicate)---> mental_state`
- `new_mental_state(string , mental_state , agent)---> mental_state`
- `new_mental_state(string , predicate , float)---> mental_state`
- `new_mental_state(string , emotion , float)---> mental_state`
- `new_mental_state(string , predicate , int)---> mental_state`
- `new_mental_state(string , mental_state , float)---> mental_state`
- `new_mental_state(string , predicate , agent)---> mental_state`
- `new_mental_state(string , mental_state , int)---> mental_state`
- `new_mental_state(string , emotion , agent)---> mental_state`
- `new_mental_state(string , emotion , int)---> mental_state`
- `new_mental_state(string , emotion , int , agent)---> mental_state`
- `new_mental_state(string , emotion , float , agent)---> mental_state`
- `new_mental_state(string , mental_state , float , int)---> mental_state`
- `new_mental_state(string , predicate , float , agent)---> mental_state`
- `new_mental_state(string , predicate , float , int)---> mental_state`
- `new_mental_state(string , mental_state , int , agent)---> mental_state`
- `new_mental_state(string , predicate , int , agent)---> mental_state`
- `new_mental_state(string , emotion , float , int)---> mental_state`
- `new_mental_state(string , mental_state , float , int , agent)---> mental_state`
- `new_mental_state(string , emotion , float , int , agent)---> mental_state`
- `new_mental_state(string , predicate , float , int , agent)---> mental_state`

Result:

creates a new mental state with a given modality (e.g. belief or desire) and various properties (a predicate it is about, a strength, a lifetime, an owner agent and an emotion it is about

Examples:

```
new_mental_state("belief", my_joy, 10, agent1)
new_mental_state("belief", mental_state1, agent1)
new_mental_state("belief", mental_state1, 12.3, 10, agent1)
```

new_predicate

Possible uses:

- `new_predicate (string) -> predicate`
- `string new_predicate map -> predicate`
- `new_predicate (string , map) -> predicate`
- `string new_predicate agent -> predicate`
- `new_predicate (string , agent) -> predicate`
- `string new_predicate bool -> predicate`
- `new_predicate (string , bool) -> predicate`
- `new_predicate (string , map , bool) -> predicate`
- `new_predicate (string , map , agent) -> predicate`
- `new_predicate (string , map , bool , agent) -> predicate`

Result:

creates a new predicate with a given name and additional properties (values, agent causing the predicate, whether it is true...)

Examples:

```
new_predicate("people to meet")
new_predicate("people to meet", ["time":10], true, agentA)
new_predicate("people to meet", map(["val1":23]) )
new_predicate("people to meet", agent1)
new_predicate("people to meet", ["time":10], true)
new_predicate("haswater", true)
new_predicate("people to meet", ["time":10], agentA)
```

new_social_link

Possible uses:

- `new_social_link (agent) -> social_link`
- `new_social_link (agent, float, float, float, float) -> social_link`

Result:

creates a new social link with another agent (eventually given additional parameters such as the appreciation,

dominance, solidarity, and familiarity values).

Examples:

```
new_social_link(agentA)
new_social_link(agentA, 0.0, -0.1, 0.2, 0.1)
```

node

Possible uses:

- `node` (`unknown`) ---> `unknown`
- `unknown` `node` `float` ---> `unknown`
- `node` (`unknown`, `float`) ---> `unknown`

Result:

Allows to create a wrapper (of type unknown) that wraps an actual object and indicates it should be considered as a node of a graph. The second (optional) parameter indicates which weight the node should have in the graph

Comment:

Useful only in graph-related operations (addition, removal of nodes, creation of graphs)

nodes

Possible uses:

- `nodes` (`container`) ---> `container`

Result:

Allows to create a wrapper (of type list) that wraps a list of objects and indicates they should be considered as nodes of a graph

none_matches

Possible uses:

- `container` `none_matches` `any expression` ---> `bool`
- `none_matches` (`container`, `any expression`) ---> `bool`

Result:

Returns true if none of the elements of the left-hand operand make the right-hand operand evaluate to true. 'c none_matches each.property' is strictly equivalent to '(c count each.property) = 0'

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, none_matches throws an error.
- If the left-hand operand is empty, none_matches returns true.

Examples:

```
bool var0 <- [1,2,3,4,5,6,7,8] none_matches (each > 3); // var0 equals false
bool var1 <- [1::2, 3::4, 5::6] none_matches (each > 4); // var1 equals false
```

See also: [one_matches](#), [all_match](#), [count](#),

none_verifies

Same signification as [none_matches](#)

norm

Possible uses:

- `norm` (`point`) ---> `float`

Result:

the norm of the vector with the coordinates of the point operand.

Examples:

```
float var0 <- norm({3,4}); // var0 equals 5.0
```

Norm

Possible uses:

- `Norm` (`any`) ---> `Norm`

Result:

casts the operand in a Norm object.

normal_area

Possible uses:

- `normal_area` (`float`, `float`, `float`) ---> `float`

Result:

Returns the area to the left of x in the normal distribution with the given mean and standard deviation.

Examples:

```
float var0 <- normal_area(0.9,0,1) with_precision(3); // var0 equals 0.816
```

normal_density

Possible uses:

- `normal_density` (`float`, `float`, `float`) ---> `float`

Result:

Returns the probability of x in the normal distribution with the given mean and standard deviation.

Examples:

```
float var0 <- (normal_density(2,1,1)*100) with_precision 2; // var0 equals 24.2
```

normal_inverse

Possible uses:

- `normal_inverse` (`float`, `float`, `float`) ---> `float`

Result:

Returns the x in the normal distribution with the given mean and standard deviation, to the left of which lies the given area. `normal`.

Examples:

```
float var0 <- normal_inverse(0.98,0,1) with_precision(2); // var0 equals 2.05
```

normalized_rotation

Possible uses:

- `normalized_rotation` (`pair`) ---> `pair<float, point>`

Result:

The rotation normalized according to Euler formalism with a positive angle, such that each rotation has a unique set of parameters (positive angle, normalize axis rotation).

Examples:

```
pair<float, point> var0 <- normalized_rotation(-38.0:{1,1,1}); // var0 equals  
38.0:{-0.5773502691896258, -0.5773502691896258, -0.5773502691896258}
```

See also: [rotation_composition, inverse_rotation](OperatorsSZ#rotation_composition, inverse_rotation),

not

Same signification as !

not

Possible uses:

- `not (predicate) ---> predicate`

Result:

create a new predicate with the inverse truth value

Examples:

```
not predicate1
```

obj_file

Possible uses:

- `obj_file (string) ---> file`
- `string obj_file pair<float,point> ---> file`
- `obj_file (string , pair<float,point>) ---> file`
- `string obj_file string ---> file`
- `obj_file (string , string) ---> file`
- `obj_file (string , string , pair<float,point>) ---> file`

Result:

Constructs a file of type obj. Allowed extensions are limited to obj, OBJ

Special cases:

- `obj_file(string)`: This file constructor allows to read an obj file. The associated mlt file have to have the same name as the file to be read.

```
file f <- obj_file("file.obj");
```

- `obj_file(string,pair<float,point>)`: This file constructor allows to read an obj file and apply an init rotation to it. The rotation is a pair angle::rotation vector. The associated mlt file have to have the same name as the file to be read.

```
file f <- obj_file("file.obj", 90.0::{-1,0,0});
```

- `obj_file(string,string)`: This file constructor allows to read an obj file, using a specific mlt file

```
file f <- obj_file("file.obj","file.mlt");
```

- `obj_file(string,string,pair<float,point>)`: This file constructor allows to read an obj file, using a specific mlt file, and apply an init rotation to it. The rotation is a pair angle::rotation vector

```
file f <- obj_file("file.obj","file.mlt", 90.0::{-1,0,0});
```

See also: [is_obj](#),

of

Same signification as .

of_generic_species

Possible uses:

- `container of_generic_species species --> list`
- `of_generic_species (container , 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 speciesA {}
// species sub_speciesA parent: speciesA {}
list var2 <- [sub_speciesA(0),sub_speciesA(1),speciesA(2),speciesA(3)] of_generic_species
speciesA; // var2 equals [sub_speciesA0,sub_speciesA1,speciesA0,speciesA1]
list var3 <- [sub_speciesA(0),sub_speciesA(1),speciesA(2),speciesA(3)] of_generic_species
sous_test; // var3 equals [sub_speciesA0,sub_speciesA1]
list var4 <- [sub_speciesA(0),sub_speciesA(1),speciesA(2),speciesA(3)] of_species speciesA; //
var4 equals [speciesA0,speciesA1]
list var5 <- [sub_speciesA(0),sub_speciesA(1),speciesA(2),speciesA(3)] of_species sous_test; //
var5 equals [sub_speciesA0,sub_speciesA1]
```

See also: [of_species](#),

of_species

Possible uses:

- `container of_species species --> list`
- `of_species (container, 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 neighbors_at 10) of_species (species (self)); // var0 equals all the
neighboring 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](#),

one_matches

Possible uses:

- `container one_matches any_expression --> bool`
- `one_matches (container, any_expression) --> bool`

Result:

Returns true if at least one of the elements of the left-hand operand make the right-hand operand evaluate to true. Returns false if the left-hand operand is empty. 'c one_matches each.property' is strictly equivalent to '(c count each.property) > 0' but faster in most cases (as it is a shortcircuited operator)

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, one_matches throws an error

Examples:

```
bool var0 <- [1,2,3,4,5,6,7,8] one_matches (each > 3); // var0 equals true  
bool var1 <- [1::2, 3::4, 5::6] one_matches (each > 4); // var1 equals true
```

See also: [none_matches](#), [all_match](#), [count](#),

one_of

Possible uses:

- `one_of` (`container<KeyType, ValueType>`) ---> `ValueType`

Result:

one of the values stored in this container at a random key

Comment:

the one_of operator behavior depends on the nature of the operand

Special cases:

- if the operand is empty, one_of returns nil
- 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 it is a list or a matrix, one_of returns one of the values of the list or of the matrix

```
inti <- 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 var3 <- [2::3, 4::5, 6::7].values contains im; // var3 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](#),

one_verifies

Same signification as [one_matches](#)

or

Possible uses:

- `bool or any expression --> bool`
- `or (bool , any expression)--> bool`

Result:

a bool value, equal to the logical or between the left-hand operand and the right-hand operand.

Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like 1 or 0 is accepted and returns true.

Examples:

```
bool var0 <- true or false; // var0 equals true
int a <-3 ; int b <- 4; int c <- 7;
bool var2 <- ((a+b) = c ) or ((a+b) > c ); // var2 equals true
```

See also: [bool](#), [and](#), [!](#)

or

Possible uses:

- `predicate or predicate` ---> `predicate`
- `or (predicate , predicate)` ---> `predicate`

Result:

create a new predicate from two others by including them as subintentions. It's an exclusive "or"

Examples:

```
predicate1 or predicate2
```

osm_file

Possible uses:

- `osm_file (string)` ---> `file`
- `string osm_file map<string,list>` ---> `file`
- `osm_file (string , map<string,list>)` ---> `file`

Result:

Constructs a file of type osm. Allowed extensions are limited to osm, pbf, bz2, gz

Special cases:

- `osm_file(string)`: This file constructor allows to read a osm (.osm, .pbf, .bz2, .gz) file (using WGS84 coordinate system for the data)

```
file f <- osm_file("file");
```

- `osm_file(string,map<string,list>)`: This file constructor allows to read an osm (.osm, .pbf, .bz2, .gz) file (using WGS84 coordinate system for the data)The map is used to filter the objects in the file according their attributes: for each key (string) of the map, only the objects that have a value for the attribute contained in the value set are kept. For an exhaustive list of the attribute of OSM data, see: http://wiki.openstreetmap.org/wiki/Map_Features

```
void var1 <- file f <- osm_file("file", map(["highway": ["primary", "secondary"],  
"building": ["yes"], "amenity": []])); // var1 equals f will contain all the objects of file  
that have the attribute 'highway' with the value 'primary' or 'secondary', and the objects that  
have the attribute 'building' with the value 'yes', and all the objects that have the attribute  
'amenity' (whatever the value).
```

See also: [is_osm](#),

out_degree_of

Possible uses:

- `graph out_degree_of unknown --> int`
- `out_degree_of (graph, unknown) --> int`

Result:

returns the out degree of a vertex (right-hand operand) in the graph given as left-hand operand.

Examples:

```
int var1 <- graphFromMap out_degree_of (node(3)); // var1 equals 4
```

See also: [in_degree_of](#), [degree_of](#),

out_edges_of

Possible uses:

- `graph out_edges_of unknown --> list`
- `out_edges_of (graph, 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](#),

overlapping

Possible uses:

- `container<unknown, geometry> overlapping geometry ---> list<geometry>`
- `overlapping (container<unknown, geometry>, geometry) ---> list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), overlapping the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] overlapping(self); // var0 equals return the agents  
among ag1, ag2 and ag3 that overlap the shape of the agent applying the operator.  
(species1 + species2) overlapping self
```

See also: [neighbors_at](#), [neighbors_of](#), [agent_closest_to](#), [agents_inside](#), [closest_to](#), [inside](#), [agents_overlapping](#),

overlaps

Possible uses:

- `geometry overlaps geometry ---> bool`
- `overlaps (geometry, geometry) ---> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) overlaps the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns false.
- if one operand is a point, returns true if the point is included in the geometry

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) overlaps polyline([{15,15},{25,25}]); // var0 equals  
true  
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
```

See also: [disjoint_from](#), [crosses](#), [intersects](#), [partially_overlaps](#), [touches](#),

pair

Possible uses:

- `pair` (`any`) ---> `pair`

Result:

casts the operand in a pair object.

palette

Possible uses:

- `palette` (`list<rgb>`) ---> `list<rgb>`
- `map<rgb, float>` `palette` `int` ---> `list<rgb>`
- `palette` (`map<rgb, float>` , `int`) ---> `list<rgb>`

Result:

returns a list of n colors chosen in the gradient provided. Colors are chosen by interpolating the stops of the gradient (the colors) using their weight, in the order described in the gradient. In case the `map<rgb, float>` passed in argument is not a gradient but a scale, the colors will be chosen in the set of colors and might appear duplicated in the palette transforms a list of n colors into a palette (necessary for some layers)

partially_overlapping

Possible uses:

- `container<unknown, geometry>` `partially_overlapping` `geometry` ---> `list<geometry>`
- `partially_overlapping` (`container<unknown, geometry>` , `geometry`) ---> `list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), partially_overlapping the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] partially_overlapping(self); // var0 equals the agents among ag1, ag2 and ag3 that partially_overlap the shape of the right-hand argument.  
list<geometry> var1 <- (species1 + species2) partially_overlapping (self); // var1 equals the agents among species species1 and species2 that partially_overlap the shape of the right-hand argument.
```

See also: [neighbors_at](#), [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [inside](#), [agents_inside](#), [agent_closest_to](#),

partially_overlaps

Possible uses:

- `geometry partially_overlaps geometry --> bool`
- `partially_overlaps (geometry , geometry) --> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) partially overlaps the right-geometry (or agent/point).

Comment:

if one geometry operand fully covers the other geometry operand, returns false (contrarily to the overlaps operator).

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) partially_overlaps polyline([{15,15},{25,25}]); // var0 equals true  
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps  
polygon([{15,15},{15,25},{25,25},{25,15}]); // var1 equals true  
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps {25,25}; // var2  
equals false  
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) partially_overlaps  
polyline([{10,10},{20,20}]); // var3 equals false
```

See also: [disjoint_from](#), [crosses](#), [overlaps](#), [intersects](#), [touches](#),

path

Possible uses:

- `path` (any) ---> `path`

Result:

casts the operand in a path object.

Special cases:

- if the operand is a path, returns this path
- if the operand is a geometry of an agent, returns a path from the list of points of the geometry
- if the operand is a list, cast each element of the list as a point and create a path from these points

```
path p <- path([{12,12},{30,30},{50,50}]);
```

path_between

Possible uses:

- `topology` `path_between` `container<unknown,geometry>` ---> `path`
- `path_between` (`topology` , `container<unknown,geometry>`)---> `path`
- `map<agent,unknown>` `path_between` `container<unknown,geometry>` ---> `path`
- `path_between` (`map<agent,unknown>` , `container<unknown,geometry>`)---> `path`
- `list<agent>` `path_between` `container<unknown,geometry>` ---> `path`
- `path_between` (`list<agent>` , `container<unknown,geometry>`)---> `path`
- `path_between` (`graph`, `unknown`, `unknown`)---> `path`
- `path_between` (`list<agent>`, `geometry`, `geometry`)---> `path`
- `path_between` (`map<agent,unknown>`, `geometry`, `geometry`)---> `path`
- `path_between` (`topology`, `geometry`, `geometry`)---> `path`

Result:

The shortest path between a list of two objects in a graph
The shortest path between two objects according to set of cells
The shortest path between several objects according to set of cells with corresponding weights
The shortest path between two objects according to set of cells with corresponding weights
The shortest path between several objects according to set of cells

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
path var2 <- path_between (cell_grid where each.is_free, ag1, ag2); // var2 equals A path
between ag1 and ag2 passing through the given cell_grid agents
path var3 <- path_between (cell_grid as_map (each::each.is_obstacle ? 9999.0 : 1.0), [ag1,
ag2, ag3]); // var3 equals A path between ag1 and ag2 and ag3 passing through the given
cell_grid agents with minimal cost
path var4 <- path_between (cell_grid as_map (each::each.is_obstacle ? 9999.0 : 1.0), ag1,
ag2); // var4 equals A path between ag1 and ag2 passing through the given cell_grid agents with
a minimal cost
path var5 <- my_topology path_between (ag1, ag2); // var5 equals A path between ag1 and ag2
path var6 <- path_between (cell_grid where each.is_free, [ag1, ag2, ag3]); // var6 equals A
path between ag1 and ag2 and ag3 passing through the given cell_grid agents
```

See also: [towards](#), [direction_to](#), [distance_between](#), [direction_between](#), [path_to](#), [distance_to](#),

path_to

Possible uses:

- point **path_to** point ---> path
- **path_to** (point, point)---> path
- geometry **path_to** geometry ---> path
- **path_to** (geometry, 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](#),

paths_between

Possible uses:

- `paths_between` (`graph`, `pair`, `int`) ---> `list<path>`

Result:

The K shortest paths between a list of two objects in a graph

Examples:

```
list<path> var0 <- paths_between(my_graph, ag1:: ag2, 2); // var0 equals the 2 shortest paths  
(ordered by length) between ag1 and ag2
```

pbinom

Same signification as [binomial_sum](#)

pchisq

Same signification as [chi_square](#)

percent_absolute_deviation

Possible uses:

- `list<float> percent_absolute_deviation` (`list<float>`) ---> `float`
- `percent_absolute_deviation` (`list<float>`, `list<float>`) ---> `float`

Result:

percent absolute deviation indicator for 2 series of values:

```
percent_absolute_deviation(list_vals_observe,list_vals_sim)
```

Examples:

```
float var0 <- percent_absolute_deviation([200,300,150,150,200],[250,250,100,200,200]); // var0  
equals 20.0
```

percentile

Same signification as [quantile_inverse](#)

pgamma

Same signification as [gamma_distribution](#)

pgm_file

Possible uses:

- `pgm_file (string) ---> file`

Result:

Constructs a file of type pgm. Allowed extensions are limited to pgm

Special cases:

- `pgm_file(string):` This file constructor allows to read a pgm file

```
file f <-pgm_file("file.pgm");
```

See also: [is_pgm](#),

plan

Possible uses:

- `container<unknown, geometry> plan float ---> geometry`
- `plan (container<unknown, geometry> , float) ---> geometry`

Result:

A polyline geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.

Examples:

```
geometry var0 <- polyplan([{0,0}, {0,10}, {10,10}, {10,0}],10); // var0 equals a polyline  
geometry composed of the 4 points with a depth of 10.
```

See also: [around](#), [circle](#), [cone](#), [link](#), [norm](#), [point](#), [polygone](#), [rectangle](#), [square](#), [triangle](#),

play_sound

Possible uses:

- `play_sound (string) -> bool`

Result:

Play a wave file

Examples:

```
bool sound_ok <- play_sound('beep.wav');
```

plus_days

Possible uses:

- `date plus_days int -> date`
- `plus_days (date, int) -> date`

Result:

Add a given number of days to a date

Examples:

```
date var0 <- date('2000-01-01') plus_days 12; // var0 equals date('2000-01-13')
```

plus_hours

Possible uses:

- `date plus_hours int --> date`
- `plus_hours (date, int) --> date`

Result:

Add a given number of hours to a date

Examples:

```
// equivalent to date1 + 15 #h
date var1 <- date('2000-01-01') plus_hours 24; // var1 equals date('2000-01-02')
```

plus_minutes

Possible uses:

- `date plus_minutes int --> date`
- `plus_minutes (date, int) --> date`

Result:

Add a given number of minutes to a date

Examples:

```
// equivalent to date1 + 5 #mn
date var1 <- date('2000-01-01') plus_minutes 5; // var1 equals date('2000-01-01 00:05:00')
```

plus_months

Possible uses:

- `date plus_months int --> date`
- `plus_months (date, int) --> date`

Result:

Add a given number of months to a date

Examples:

```
date var0 <- date('2000-01-01') plus_months 5; // var0 equals date('2000-06-01')
```

plus_ms

Possible uses:

- `date plus_ms int` --> `date`
- `plus_ms (date, int)` --> `date`

Result:

Add a given number of milliseconds to a date

Examples:

```
// equivalent to date('2000-01-01') + 15 #ms
date var1 <- date('2000-01-01') plus_ms 1000; // var1 equals date('2000-01-01 00:00:01')
```

plus_seconds

Same signification as `+`

plus_weeks

Possible uses:

- `date plus_weeks int` --> `date`
- `plus_weeks (date, int)` --> `date`

Result:

Add a given number of weeks to a date

Examples:

```
date var0 <- date('2000-01-01') plus_weeks 15; // var0 equals date('2000-04-15')
```

plus_years

Possible uses:

- `date plus_years int` ---> `date`
- `plus_years (date, int)` ---> `date`

Result:

Add a given number of years to a date

Examples:

```
date var0 <- date('2000-01-01') plus_years 15; // var0 equals date('2015-01-01')
```

pnorm

Same signification as `normal_area`

point

Possible uses:

- `point (any)` ---> `point`

Result:

casts the operand in a point object.

points_along

Possible uses:

- `geometry points_along list<float>` ---> `list`

- `points_along` (`geometry` , `list<float>`)---> `list`

Result:

A list of points along the operand-geometry given its location in terms of rate of distance from the starting points of the geometry.

Examples:

```
list var0 <- line([{10,10},{80,80}]) points_along ([0.3, 0.5, 0.9]); // var0 equals the list  
of following points: [{31.0,31.0,0.0},{45.0,45.0,0.0},{73.0,73.0,0.0}]
```

See also: [closest_points_with](#), [farthest_point_to](#), [points_at](#), [points_on](#),

points_at

Possible uses:

- `int points_at float` ---> `list<point>`
- `points_at (int , float)`---> `list<point>`

Result:

A list of left-operand number of points located at a the right-operand distance to the agent location.

Examples:

```
list<point> var0 <- 3 points_at(20.0); // var0 equals returns [pt1, pt2, pt3] with pt1, pt2 and  
pt3 located at a distance of 20.0 to the agent location
```

See also: [any_location_in](#), [any_point_in](#), [closest_points_with](#), [farthest_point_to](#),

points_in

Possible uses:

- `field points_in geometry` ---> `list<point>`
- `points_in (field , geometry)`---> `list<point>`

points_on

Possible uses:

- `geometry points_on float` ---> `list`
- `points_on (geometry , 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](#),

poisson

Possible uses:

- `poisson (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](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#),

polygon

Possible uses:

- `polygon` (`container<unknown,geometry>`) ---> `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.  
float var1 <- polygon([{0,0}, {0,10}, {10,10}, {10,0}]).area; // var1 equals 100.0  
point var2 <- polygon([{0,0}, {0,10}, {10,10}, {10,0}]).location; // var2 equals  
point(5.0,5.0,0.0)
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

polyhedron

Possible uses:

- `container<unknown,geometry>` `polyhedron` `float` ---> `geometry`
- `polyhedron` (`container<unknown,geometry>` , `float`) ---> `geometry`

Result:

A polyhedron geometry from the given list of points.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry
- if the operand is composed of 2 points, returns a polyline geometry.

Examples:

```
geometry var0 <- polyhedron([{0,0}, {0,10}, {10,10}, {10,0}],10); // var0 equals a polygon  
geometry composed of the 4 points and of depth 10.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

polyline

Same signification as [line](#)

polyplan

Same signification as [plan](#)

predecessors_of

Possible uses:

- `graph predecessors_of unknown --> list`
- `predecessors_of (graph , 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: [neighbors_of](#), [successors_of](#),

predicate

Possible uses:

- `predicate` (`any`) ---> `predicate`

Result:

casts the operand in a predicate object.

predict

Possible uses:

- `regression predict list` ---> `float`
- `predict (regression , list)` ---> `float`

Result:

returns the value predicted by the regression parameters for a given instance. Usage: `predict(regression, instance)`

Examples:

```
predict(my_regression, [1,2,3])
```

product

Same signification as `mul`

product_of

Possible uses:

- `container product_of any expression` ---> `unknown`
- `product_of (container , any expression)` ---> `unknown`

Result:

the product of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] product_of (each); // var1 equals 48
```

Examples:

```
unknown var0 <- [1,2] product_of (each * 10 ); // var0 equals 200
```

See also: [min_of](#), [max_of](#), [sum_of](#), [mean_of](#),

promethee_DM

Possible uses:

- `list<list> promethee_DM list<map<string,unknown>> ---> int`
- `promethee_DM (list<list>, list<map<string,unknown>>) ---> int`

Result:

The index of the best candidate according to the Promethee II method. This method is based on a comparison per pair of possible candidates along each criterion: all candidates are compared to each other by pair and ranked. More information about this method can be found in [Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research\(2010\)](#). The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fours elements: a name, a weight, a preference value (p) and an indifference value (q). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant.

Special cases:

- returns -1 if the list of candidates is nil or empty

Examples:

```
int var0 <- promethee_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [{"name":"utility", "weight" :: 2.0, "p":0.5, "q":0.0, "s":1.0, "maximize" :: true}, {"name":"price", "weight" :: 1.0, "p":0.5, "q":0.0, "s":1.0, "maximize" :: false}]); // var0 equals 1
```

See also: [weighted_means_DM](#), [electre_DM](#), [evidence_theory_DM](#),

property_file

Possible uses:

- `property_file (string) ---> file`
- `string property_file (map<string,string>) ---> file`
- `property_file (string , map<string,string>) ---> file`

Result:

Constructs a file of type property. Allowed extensions are limited to properties

Special cases:

- `property_file(string):` This file constructor allows to read a property file (.properties)

```
file f <-property_file("file.properties");
```

- `property_file(string,map<string,string>):` This file constructor allows to store a map in a property file (it does not save it - just store it in memory)

```
file f <-property_file("file.properties", map(["param1":1.0,"param3":10.0 ]));
```

See also: [is_property](#),

pValue_for_fStat

Possible uses:

- `pValue_for_fStat (float, int, int) ---> float`

Result:

Returns the P value of F statistic fstat with numerator degrees of freedom dfn and denominator degrees of freedom dfd. Uses the incomplete Beta function.

Examples:

```
float var0 <- pValue_for_fStat(1.9,10,12) with_precision(3); // var0 equals 0.145
```

pValue_for_tStat**Possible uses:**

- float pValue_for_tStat int ---> float
- pValue_for_tStat (float , int) ---> float

Result:

Returns the P value of the T statistic tstat with df degrees of freedom. This is a two-tailed test so we just double the right tail which is given by studentT of -|tstat|.

Examples:

```
float var0 <- pValue_for_tStat(0.9,10) with_precision(3); // var0 equals 0.389
```

pyramid**Possible uses:**

- pyramid (float) ---> geometry

Result:

A square geometry which side size is given by the operand.

Comment:

the center of the pyramid is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- pyramid(5); // var0 equals a geometry as a square with side_size = 5.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#),

quantile

Possible uses:

- `container quantile float --> float`
- `quantile (container , float) --> float`

Result:

Returns the phi-quantile; that is, an element elem for which holds that phi percent of data elements are less than elem. The quantile does not need necessarily to be contained in the data sequence, it can be a linear interpolation. Note that the container holding the values must be sorted first

Examples:

```
float var0 <-
quantile([1,3,5,6,9,11,12,13,19,21,22,32,35,36,45,44,55,68,79,80,81,88,90,91,92,100], 0.5); // var0 equals 35.5
```

quantile_inverse

Possible uses:

- `container quantile_inverse float --> float`
- `quantile_inverse (container , float) --> float`

Result:

Returns how many percent of the elements contained in the receiver are \leq element. Does linear interpolation if the element is not contained but lies in between two contained elements. Note that the container holding the values must be sorted first

Examples:

```
float var0 <-
quantile_inverse([1,3,5,6,9,11,12,13,19,21,22,32,35,36,45,44,55,68,79,80,81,88,90,91,92,100],
35.5) with_precision(2); // var0 equals 0.52
```

range

Possible uses:

- `range (int) --> list`
- `int range int --> list`
- `range (int , int) --> list`
- `range (int , int , int) --> list`

Result:

builds a list of int representing all contiguous values from zero to the argument. The range can be increasing or decreasing.

Special cases:

- Passing 0 will return a singleton list with 0.
- When used with 2 operands, it returns the list of int representing all contiguous values from the first to the second argument. Passing the same value for both will return a singleton list with this value

```
list var0 <- range(0,2); // var0 equals [0,1,2]
```

- When used with 3 operands, it returns a list of int representing all contiguous values from the first to the second argument, using the step represented by the third argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value. Passing a step of 0 will result in an exception. Attempting to build infinite ranges (e.g. end > start with a negative step) will similarly not be accepted and yield an exception

```
list var1 <- range(0,6,2); // var1 equals [0,2,4,6]
```

rank_interpolated

Possible uses:

- container rank_interpolated float --> float
- rank_interpolated (container , float)---> float

Result:

Returns the linearly interpolated number of elements in a list less or equal to a given element. The rank is the number of elements <= element. Ranks are of the form {0, 1, 2,..., sortedList.size()}. If no element is <= element, then the rank is zero. If the element lies in between two contained elements, then linear interpolation is used and a non integer value is returned. Note that the container holding the values must be sorted first

Examples:

```
float var0 <-  
rank_interpolated([1,3,5,6,9,11,12,13,19,21,22,32,35,36,45,44,55,68,79,80,81,88,90,91,92,100],  
35); // var0 equals 13.0
```

read

Possible uses:

- read (string)---> unknown

Result:

Reads an attribute of the agent. The attribute's name is specified by the operand.

Examples:

```
unknownagent_name <- read ('name'); // agent_name equals reads the 'name' variable of agent  
then assigns the returned value to the 'agent_name' variable.
```

rectangle

Possible uses:

- rectangle (point)---> geometry
- float rectangle float ---> geometry

- `rectangle` (`float`, `float`) -> `geometry`
- `point` `rectangle` `point` -> `geometry`
- `rectangle` (`point`, `point`) -> `geometry`

Result:

A rectangle geometry, computed from the operands values (e.g. the 2 side sizes).

Comment:

the center of the rectangle is by default the location of the current agent in which has been called this operator.the center of the rectangle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- rectangle(10, 5); // var0 equals a geometry as a rectangle with width = 10 and
height = 5.
geometry var1 <- rectangle({0.0,0.0}, {10.0,10.0}); // var1 equals a geometry as a rectangle
with {1.0,1.0} as the upper-left corner, {10.0,10.0} as the lower-right corner.
geometry var2 <- rectangle({10, 5}); // var2 equals a geometry as a rectangle with width = 10
and height = 5.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [square](#), [triangle](#),

reduced_by

Same signification as -

regex_matches

Possible uses:

- `string` `regex_matches` `string` -> `list<string>`
- `regex_matches` (`string`, `string`) -> `list<string>`

Result:

Returns the list of sub-strings of the first operand that match the regular expression provided in the second

operand

Examples:

```
list<string> var0 <- regex_matches("colour, color", "colou?r"); // var0 equals  
['colour','color']
```

See also: [replace_regex](#),

regression

Possible uses:

- `regression` (any) ---> `regression`

Result:

casts the operand in a regression object.

remove_duplicates

Possible uses:

- `remove_duplicates` (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 a graph, remove_duplicates returns the set of nodes
- if the operand is empty, remove_duplicates returns an empty list

```
list var1 <- remove_duplicates([]); // var1 equals []
```

- if the operand is a map, remove_duplicates returns the set of values without duplicate

```
list var2 <- remove_duplicates([1::3,2::4,3::3,5::7]); // var2 equals [3,4,7]
```

- if the operand is a matrix, remove_duplicates returns a list containing all the elements with duplicates.

```
list var3 <- remove_duplicates([[ "c11", "c12", "c13", "c13"], [ "c21", "c22", "c23", "c23"]]); // var3  
equals [[ "c11", "c12", "c13", "c21", "c22", "c23"]]
```

Examples:

```
list var0 <- remove_duplicates([3,2,5,1,2,3,5,5,5]); // var0 equals [3,2,5,1]
```

remove_node_from

Possible uses:

- `geometry remove_node_from graph` --> `graph`
- `remove_node_from (geometry , graph)`--> `graph`

Result:

removes a node from a graph.

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph. 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)
```

rename_file

Possible uses:

- `string rename_file string`-->`bool`
- `rename_file (string , string)`-->`bool`

Result:

rename/move a file or a folder

Examples:

```
bool rename_file_ok <- rename_file("../includes/my_folder", "../includes/my_new_folder");
```

replace

Possible uses:

- `replace (string, string, string) -> string`

Result:

Returns the string obtained by replacing by the third operand, in the first operand, all the sub-strings equal to the second operand

Examples:

```
string var0 <- replace('to be or not to be,that is the question','to', 'do'); // var0 equals  
'do be or not do be,that is the question'
```

See also: [replace_regex](#),

replace_regex

Possible uses:

- `replace_regex (string, string, string) -> string`

Result:

Returns the string obtained by replacing by the third operand, in the first operand, all the sub-strings that match the regular expression of the second operand

Examples:

```
string var0 <- replace_regex("colour, color", "colou?r", "col"); // var0 equals 'col, col'
```

See also: [replace](#),

residuals

Possible uses:

- `residuals` (`regression`) ---> `list<float>`

Result:

Return the list of residuals for a given regression model

Examples:

```
residuals(my_regression)
```

restore_simulation

Possible uses:

- `restore_simulation` (`string`) ---> `int`

Result:

restores a simulation from a string value containing a serialized simulation.

Comment:

This operator should be used in a reflex of an experiment and it will remove the current simulation and replace it by the new restored simulation

See also: [restore_simulation_from_file](#),

restore_simulation_from_file

Possible uses:

- `restore_simulation_from_file` (`file`) ---> `int`

Result:

Restores a simulation from a saved simulation file.

Comment:

This operator should be used in a reflex of an experiment and it will remove the current simulation and replace it by the new restored simulation

See also: [restore_simulation](#),

reverse

Possible uses:

- `reverse (map<K, V>)` ---> `map`
- `reverse (string)` ---> `string`
- `reverse (container<KeyType, ValueType>)` ---> `container<unknown, unknown>`

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 var2 <- reverse ('abcd'); // var2 equals 'dcba'
```

- if it is a list, reverse returns a copy of the operand list with elements in the reversed order

```
list<int> var3 <- reverse ([10,12,14]); // var3 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)

```
map<int, string> var4 <- reverse ([ 'k1'::44, 'k2'::32, 'k3'::12]); // var4 equals [44::'k1',  
32::'k2', 12::'k3']
```

- if it is a matrix, reverse returns a new matrix containing the transpose of the operand.

```
matrix<string> var5 <- reverse(matrix([["c11", "c12", "c13"], ["c21", "c22", "c23"]])); // var5  
equals matrix([["c11", "c21"], ["c12", "c22"], ["c13", "c23"]])
```

Examples:

```
map<int, int> m <- [1::111, 2::222, 3::333, 4::444];  
map var1 <- reverse(m); // var1 equals map([111::1, 222::2, 333::3, 444::4])
```

rewire_n

Possible uses:

- graph rewire_n int --> graph
- rewire_n (graph, int) --> graph

Result:

rewires the given count of edges.

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph. If there are too many edges, all the edges will be rewired.

Examples:

```
graph var1 <- graphEpidemio rewire_n 10; // var1 equals the graph with 3 edges rewired
```

rgb

Possible uses:

- rgb rgb float --> rgb
- rgb (rgb, float) --> rgb
- rgb rgb int --> rgb

- `rgb (rgb , int) ---> rgb`
- `string rgb int ---> rgb`
- `rgb (string , int) ---> rgb`
- `rgb (int , int , int) ---> rgb`
- `rgb (int , int , int , int) ---> rgb`
- `rgb (int , int , int , float) ---> rgb`

Result:

Returns a color defined by red, green, blue components and an alpha blending value.

Special cases:

- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0 and 255)
- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0.0 and 1.0)
- It can be used with r=red, g=green, b=blue, each between 0 and 255
- It can be used with a color and an alpha between 0 and 1
- 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 (255,0,0,125); // 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(rgb(255,0,0),0.5); // var3 equals a light red color
rgb var4 <- rgb(rgb(255,0,0),125); // var4 equals a light red color
rgb var5 <- rgb ("red"); // var5 equals rgb(255,0,0)
```

See also: [hsb](#),

`rgb`

Possible uses:

- `rgb (any) ---> rgb`

Result:

casts the operand in a `rgb` object.

rms

Possible uses:

- `int rms float --> float`
- `rms (int , float) --> float`

Result:

Returns the RMS (Root-Mean-Square) of a data sequence. The RMS of data sequence is the square-root of the mean of the squares of the elements in the data sequence. It is a measure of the average size of the elements of a data sequence.

Examples:

```
list<float> data_sequence <- [6.0, 7.0, 8.0, 9.0];
list<float> squares <- data_sequence collect (each*each);
float var2 <- rms(length(data_sequence),sum(squares)) with_precision(4) ; // var2 equals 7.5829
```

rnd

Possible uses:

- `rnd (int) --> int`
- `rnd (point) --> point`
- `rnd (float) --> float`
- `point rnd point --> point`
- `rnd (point , point) --> point`
- `int rnd int --> int`
- `rnd (int , int) --> int`
- `float rnd float --> float`
- `rnd (float , float) --> float`
- `rnd (point, point, float) --> point`
- `rnd (int, int, int) --> int`
- `rnd (float, float, float) --> float`

Result:

returns a random value in a range (the type value depends on the operand type): when called with an integer, it

returns a random integer in the interval [0, operand]

Comment:

to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision

Special cases:

- if the operand is a point, returns a point with three random float ordinates, each in the interval [0, ordinate of argument]
- if the operand is a float, returns an uniformly distributed float random number in [0.0, to]

Examples:

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

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [skew_gauss](#), [truncated_gauss](#), [weibull_rnd](#),

rnd_choice

Possible uses:

- `rnd_choice (list) -> int`
- `rnd_choice (map<unknown, unknown>) -> unknown`

Result:

returns an index of the given list with a probability following the (normalized) distribution described in the list (a form of lottery) returns a key from the map with a probability following the (normalized) distribution described in map values (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  
unknown var1 <- rnd_choice(["toto":0.2,\\"tata\\":0.5,\\"tonton\\":0.3]); // var1 equals 2/10
```

See also: [rnd](#),

rnd_color

Possible uses:

- `rnd_color (int) --> rgb`
- `int rnd_color int --> rgb`
- `rnd_color (int , int) --> rgb`

Result:

`rgb` color Return a random color equivalent to `rgb(rnd(first_op, last_op),rnd(first_op, last_op),rnd(first_op, last_op))`

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))  
rgb var1 <- rnd_color(100, 200); // var1 equals a random color, equivalent to rgb(rnd(100, 200),rnd(100, 200),rnd(100, 200))
```

See also: [rgb](#), [hsb](#),

rotated_by

Possible uses:

- `point rotated_by pair --> point`
- `rotated_by (point , pair) --> point`
- `geometry rotated_by int --> geometry`
- `rotated_by (geometry , int) --> geometry`
- `geometry rotated_by float --> geometry`
- `rotated_by (geometry , float) --> geometry`
- `geometry rotated_by pair --> geometry`
- `rotated_by (geometry , pair) --> geometry`
- `rotated_by (geometry , float , point) --> 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) A geometry resulting from the application of a rotation by the operand angles (degree) along the operand axis (last operand) to the left-hand operand (geometry, agent, point)

Special cases:

- When used with a point and a pair angle::point, it returns a point resulting from the application of the right-hand rotation operand (angles in degree) to the left-hand operand point
- the right-hand operand representing the angle can be a float or an integer

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.
geometry var1 <- rotated_by(pyramid(10),45.0:{1,0,0}); // var1 equals the geometry resulting
from a 45 degrees rotation along the {1,0,0} vector to the geometry of the agent applying the
operator.
geometry var2 <- rotated_by(pyramid(10),45.0, {1,0,0}); // var2 equals the geometry resulting
from a 45 degrees rotation along the {1,0,0} vector to the geometry of the agent applying the
operator.
```

See also: [transformed_by](#), [translated_by](#),

rotated_by

Possible uses:

- `image rotated_by float` ---> `image`
- `rotated_by (image , float)` ---> `image`

Result:

Returns the image rotated using the angle in degrees passed in parameter. A positive angle means a clockwise rotation, and a negative one a counter-clockwise. The original image is left untouched

rotation_composition

Possible uses:

- `rotation_composition (list<pair>)` ---> `pair<float, point>`

Result:

The rotation resulting from the composition of the rotations in the list, from left to right. Angles are in degrees.

Examples:

```
pair<float,point> var0 <- rotation_composition([38.0::{1,1,1},90.0::{1,0,0}]); // var0 equals  
115.22128507898108:{0.9491582126366207,0.31479943993669307,-0.0}
```

See also: [inverse_rotation](#),

round

Possible uses:

- `round (point) --> point`
- `round (int) --> int`
- `round (float) --> int`

Result:

Returns the rounded value of the operand.

Special cases:

- if the operand is an int, round returns it

Examples:

```
point var0 <- {12345.78943, 12345.78943, 12345.78943} with_precision 2; // var0 equals  
{12345.79,12345.79,12345.79}  
int var1 <- round (0.51); // var1 equals 1  
int var2 <- round (100.2); // var2 equals 100  
int var3 <- round(-0.51); // var3 equals -1
```

See also: [round](#), [int](#), [with_precision](#),

row_at

Possible uses:

- `matrix<unknown> row_at int --> list<unknown>`
- `row_at (matrix<unknown>, int) --> list<unknown>`

Result:

returns the row at a num_line (right-hand operand)

Examples:

```
list<unknown> var0 <-  
matrix([["el11", "el12", "el13"], ["el21", "el22", "el23"], ["el31", "el32", "el33"]]) row_at 2; //  
var0 equals ["el13", "el23", "el33"]
```

See also: [column_at](#), [columns_list](#),

rows_list

Possible uses:

- `rows_list` (`matrix<unknown>`) --> `list<list<unknown>>`

Result:

returns a list of the rows of the matrix, with each row as a list of elements

Examples:

```
list<list<unknown>> 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](#),

rSquare

Possible uses:

- `rSquare` (`regression`) --> `float`

Result:

Return the value of the adjusted R square for a given regression model

Examples:

```
rSquare(my_regression)
```

Version: 1.9.1

Operators (S 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 spec1 can use `min` as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
    init {
        create spec1;
        spec1 my_agent <- spec1[0];
        int the_min <- my_agent min(10,20); // or min(my_agent, 10, 20);
    }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considered as returning a value of type `unknown` (e.g. `unknown result <- my_agent the_action(op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

`box, cone3D, cube, cylinder, hexagon, pyramid, set_z, sphere, teapot,`

Arithmetic operators

`-, /, ^, *, +, abs, acos, asin, atan, atan2, ceil, cos, cos_rad, div, even, exp, fact, floor, hypot, is_finite, is_number, ln, log, mod, round, signum, sin, sin_rad, sqrt, tan, tan_rad, tanh, with_precision,`

BDI

add_values, and, eval_when, get_about, get_agent, get_agent_cause, get_belief_op, get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op, get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op, get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity, get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op, get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op, get_intentions_with_name_op, get_lifetime, get_liking, get_modality, get_obligation_op, get_obligation_with_name_op, get_obligations_op, get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity, get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op, get_uncertainties_with_name_op, get_uncertainty_op, get_uncertainty_with_name_op, get_values, has_belief_op, has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op, has_ideal_with_name_op, has_intention_op, has_intention_with_name_op, has_obligation_op, has_obligation_with_name_op, has_uncertainty_op, has_uncertainty_with_name_op, new_emotion, new_mental_state, new_predicate, new_social_link, not, or, set_about, set_agent, set_agent_cause, set_decay, set_dominance, set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate, set_solidarity, set_strength, set_trust, set_truth, with_values,

Casting operators

as, as_int, as_matrix, field_with, font, is, is_skill, list_with, matrix_with, species_of, to_gaml, to_geojson, to_list, with_size, with_style,

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, gradient, grayscale, hsb, mean, median, palette, rgb, rnd_color, scale, sum, to_hsb,

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, all_match, among, as_json_string, at, cartesian_product, collect, contains, contains_all, contains_any, contains_key, count, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, none_matches, one_matches, one_of, product_of, range, remove_duplicates, reverse, shuffle, sort_by, split, split_in, split_using, sum, sum_of, union, variance_of, where, with_max_of, with_min_of,

Date-related operators

-, !=, +, <, <=, =, >, >=, after, before, between, every, milliseconds_between, minus_days, minus_hours, minus_minutes, minus_months, minus_ms, minus_weeks, minus_years, months_between, plus_days, plus_hours, plus_minutes, plus_months, plus_ms, plus_weeks, plus_years, since, to, until, years_between,

Dates

Displays

horizontal, stack, vertical,

edge

`edge_between, strahler,`

EDP-related operators

`diff, diff2,`

Files-related operators

`copy_file, crs, csv_file, delete_file, dxf_file, evaluate_sub_model, file_exists, folder, folder_exists, gaml_file, geojson_file, get, gif_file, gml_file, graph6_file, graphdimacs_file, graphdot_file, graphgexf_file, graphgml_file, graphml_file, graphsplib_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot, is_graphgexf, is_graphgml, is_graphml, is_graphsplib, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_saved_simulation, is_shape, is_svg, is_text, is_threeds, is_xml, json_file, new_folder, obj_file, osm_file, pgm_file, property_file, read, rename_file, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, unzip, writable, xml_file, zip,`

GamaMetaType

`type_of,`

Gen*

`add_attribute, add_census_file, add_mapper, add_marginals, add_range_attribute, with_generation_algo,`

Graphs-related operators

`add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, as_spatial_graph, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_random_graph, generate_watts_strogatz, girvan_newman_clustering, grid_cells_to_graph, in_degree_of, in_edges_of, k_spanning_tree_clustering, label_propagation_clustering, layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid, load_shortest_paths, main_connected_component, max_flow_between, maximal_cliques_of, nb_cycles, neighbors_of, node, nodes, out_degree_of, out_edges_of, path_between, paths_between, predecessors_of, remove_node_from, rewire_n, source_of, spatial_graph, strahler, successors_of, sum, target_of, undirected, use_cache, weight_of, with_k_shortest_path_algorithm, with_shortest_path_algorithm, with_weights,`

Grid-related operators

`as_4_grid, as_grid, as_hexagonal_grid, cell_at, cells_in, cells_overlapping, field, grid_at, neighbors_of, path_between, points_in, values_in,`

ImageOperators

`*, antialiased, blend, blurred, brighter, clipped_with, darker, grayscale, horizontal_flip, image, matrix, rotated_by, sharpened, snapshot, tinted_with, vertical_flip, with_height, with_size, with_width,`

Iterator operators

accumulate, all_match, as_map, collect, count, create_map, first_with, frequency_of, group_by, index_by, last_with, max_of, mean_of, min_of, none_matches, one_matches, product_of, sort_by, sum_of, variance_of, where, where, where, with_max_of, with_min_of,

List-related operators

all_indexes_of, copy_between, index_of, last_index_of,

Logical operators

; !, ?, add_3Dmodel, add_geometry, add_icon, and, or, xor,

Map comparaison operators

fuzzy_kappa, fuzzy_kappa_sim, kappa, kappa_sim, percent_absolute_deviation,

Map-related operators

as_map, create_map, index_of, last_index_of,

Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

multicriteria operators

electre_DM, evidence_theory_DM, fuzzy_choquet_DM, promethee_DM, weighted_means_DM,

Path-related operators

agent_from_geometry, all_pairs_shortest_path, as_path, load_shortest_paths, max_flow_between, path_between, path_to, paths_between, use_cache,

Pedestrian

generate_pedestrian_network,

Points-related operators

-, /, *, +, <, <=, >, >=, add_point, angle_between, any_location_in, centroid, closest_points_with, farthest_point_to, grid_at, norm, points_along, points_at, points_on,

Random operators

binomial, flip, gamma_density, gamma_rnd, gamma_trunc_rnd, gauss, generate_terrain, lognormal_density, lognormal_rnd, lognormal_trunc_rnd, poisson, rnd, rnd_choice, sample, shuffle, skew_gauss, truncated_gauss, weibull_density, weibull_rnd, weibull_trunc_rnd,

ReverseOperators

restore_simulation, restore_simulation_from_file, save_simulation, serialize, serialize_agent,

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, elliptical_arc, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

Spatial operators

- , *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, angle_between, any_location_in, arc, around, as_4_grid, as_driving_graph, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covering, covers, cross, crosses, crossing, crs, CRS_transform, cube, curve, cylinder, direction_between, disjoint_from, distance_between, distance_to, ellipse, elliptical_arc, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, inverse_rotation, k_nearest_neighbors, line, link, masked_by, moran, neighbors_at, neighbors_of, normalized_rotation, overlapping, overlaps, partially_overlapping, partially_overlaps, path_between, path_to, plan, points_along, points_at, points_on, polygon, polyhedron, pyramid, rectangle, rotated_by, rotation_composition, round, scaled_to, set_z, simple_clustering_by_distance, simplification, skeletonize, smooth, sphere, split_at, split_geometry, split_lines, square, squircle, teapot, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, touches, touching, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_covering, agents_crossing, agents_inside, agents_overlapping, agents_partially_overlapping, agents_touching, at_distance, closest_to, covering, crossing, farthest_to, inside, neighbors_at, neighbors_of, overlapping, partially_overlapping, touching,

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

Spatial statistical operators

hierarchical_clustering, k_nearest_neighbors, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, clean_network, convex_hull, CRS_transform, inverse_rotation, normalized_rotation, rotated_by, rotation_composition, scaled_to, simplification, skeletonize, smooth, split_geometry, split_lines, to_GAMA_CRS, to_rectangles, to_segments, to_squares, to_sub_geometries, transformed_by, translated_by, triangulate, voronoi, with_precision, without_holes,

Species-related operators

index_of, last_index_of, of_generic_species, of_species,

Statistical operators

auto_correlation, beta, binomial_coeff, binomial_complemented, binomial_sum, build, chi_square, chi_square_complemented, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, k_nearest_neighbors, kmeans, kurtosis, log_gamma, max, mean, mean_deviation, median, min, moment, moran, morrisAnalysis, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, residuals, rms, rSquare, simple_clustering_by_distance, skewness, sobolAnalysis, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, t_test, variance,

Strings-related operators

+, <, <=, >, >=, at, capitalize, char, contains, contains_all, contains_any, copy_between, date, empty, first, in, indented_by, index_of, is_number, last, last_index_of, length, lower_case, regex_matches, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

SubModel

load_sub_model,

System

., choose, command, copy, copy_from_clipboard, copy_to_clipboard, copy_to_clipboard, dead, enter, eval_gaml, every, is_error, is_reachable, is_warning, play_sound, user_confirm, user_input_dialog, wizard, wizard_page,

Time-related operators

date, string,

Types-related operators

action, agent, attributes, BDIPPlan, bool, container, conversation, directory, emotion, file, float, gamm_type, gen_population_generator, gen_range, geometry, graph, int, kml, list, map, matrix, mental_state, message, Norm, pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, species, topology, unknown,

User control operators

choose, enter, user_confirm, user_input_dialog, wizard, wizard_page,

Operators

sample

Possible uses:

- `sample` (any expression) --> string
- string `sample` any expression --> string
- `sample` (string , any expression) --> string
- `sample` (list, int, bool) --> list
- `sample` (list, int, bool, list) --> list

Result:

takes a sample of the specified size from the elements of x using either with or without replacement takes a sample of the specified size from the elements of x using either with or without replacement with given weights

Examples:

```
list var0 <- sample([2,10,1],2,false); // var0 equals [10,1]
list var1 <- sample([2,10,1],2,false,[0.1,0.7,0.2]); // var1 equals [10,2]
```

Sanction

Possible uses:

- `Sanction` (any) --> Sanction

Result:

casts the operand in a Sanction object.

save_simulation

Possible uses:

- `save_simulation` (string) --> int

Result:

saves the current simulation in a given file

Comment:

About to be deprecated, the save statement should be used instead.

saved_simulation_file

Possible uses:

- `saved_simulation_file` (string) --> file

- `string saved_simulation_file` `list<agent>` ---> `file`
- `saved_simulation_file (string , list<agent>)` ---> `file`
- `string saved_simulation_file` `bool` ---> `file`
- `saved_simulation_file (string , bool)` ---> `file`

Result:

Constructs a file of type saved_simulation. Allowed extensions are limited to gsim, gasim

Special cases:

- `saved_simulation_file(string)`: Constructor for saved simulation files: read the metadata and content.
- `saved_simulation_file(string,list<agent>)`: Constructor for saved simulation files from a list of agents: it is used with aim of saving a simulation agent.
- `saved_simulation_file(string,bool)`: Constructor for saved simulation files: read the metadata. If and only if the boolean operand is true, the content of the file is read.

See also: [is_saved_simulation](#),

scale

Possible uses:

- `scale (map<rgb,unknown>)` ---> `map<float,rgb>`
- `scale (map<rgb,unknown>, float, float)` ---> `map<float,rgb>`

Result:

Similar to gradient(`map<rgb, float>`) but reorders the colors based on their weight and does not normalize them, so as to effectively represent a color scale (i.e. a correspondance between a range of value and a color that implicitly begins with the lowest value) For instance `scale([#red::10, #green::0, #blue::30])` would produce the reverse map and associate #green to the interval 0-10, #red to 10-30, and #blue above 30. The main difference in usages is that, for instance in the definition of a mesh to display, a gradient will produce interpolated colors to accomodate for the intermediary values, while a scale will stick to the colors defined. Expects a gradient, i.e. a `map<rgb,float>`, where values represent the different stops of the colors. First normalizes the passed gradient, and then applies the resulting weights to the interval represented by min and max, so as to return a scale (i.e. absolute values instead of the stops)

See also: [gradient](#),

scaled_by

Same signification as *

scaled_to

Possible uses:

- `geometry scaled_to point` ---> `geometry`
- `scaled_to (geometry , point)` ---> `geometry`

Result:

allows to restrict the size of a geometry so that it fits in the envelope {width, height, depth} defined by the second operand

Examples:

```
geometry var0 <- shape scaled_to {10,10}; // var0 equals a geometry corresponding to the geometry of the agent applying the operator scaled so that it fits a square of 10x10
```

select

Same signification as [where](#)

serialize

Possible uses:

- `serialize` (`unknown`) ---> `string`

Result:

It serializes any object, i.e. transforms it into a string.

See also: [serialize_agent](#),

serialize_agent

Possible uses:

- `serialize_agent` (`agent`) ---> `string`

Result:

serializes an agent (i.e. transforms into a string value).

Comment:

As a simulation is a particular agent, it can be used to serialize a simulation and save it.

See also: [serialize](#),

set_about

Possible uses:

- `emotion` `set_about` `predicate` ---> `emotion`
- `set_about` (`emotion`, `predicate`) ---> `emotion`

Result:

change the about value of the given emotion

Examples:

```
emotion set_about predicate1
```

set_agent

Possible uses:

- `social_link set_agent agent` ---> `social_link`
- `set_agent(social_link, agent)` ---> `social_link`

Result:

change the agent value of the given social link

Examples:

```
social_link set_agent agentA
```

set_agent_cause

Possible uses:

- `predicate set_agent_cause agent` ---> `predicate`
- `set_agent_cause(predicate, agent)` ---> `predicate`
- `emotion set_agent_cause agent` ---> `emotion`
- `set_agent_cause(emotion, agent)` ---> `emotion`

Result:

change the agentCause value of the given predicate change the agentCause value of the given emotion

Examples:

```
predicate set_agent_cause agentA  
new_emotion set_agent_cause agentA
```

set_decay

Possible uses:

- `emotion set_decay float` ---> `emotion`
- `set_decay(emotion, float)` ---> `emotion`

Result:

change the decay value of the given emotion

Examples:

```
emotion set_decay 12
```

set_dominance

Possible uses:

- `social_link set_dominance float --> social_link`
- `set_dominance (social_link, float) --> social_link`

Result:

change the dominance value of the given social link

Examples:

```
social_link set_dominance 0.4
```

set_familiarity

Possible uses:

- `social_link set_familiarity float --> social_link`
- `set_familiarity (social_link, float) --> social_link`

Result:

change the familiarity value of the given social link

Examples:

```
social_link set_familiarity 0.4
```

set_intensity

Possible uses:

- `emotion set_intensity float --> emotion`
- `set_intensity (emotion, float) --> emotion`

Result:

change the intensity value of the given emotion

Examples:

```
emotion set_intensity 12
```

set_lifetime

Possible uses:

- `mental_state set_lifetime int --> mental_state`
- `set_lifetime (mental_state, int) --> mental_state`

Result:

change the lifetime value of the given mental state

Examples:

```
mental state set_lifetime 1
```

set_liking

Possible uses:

- `social_link set_liking float` ---> `social_link`
- `set_liking (social_link, float)` ---> `social_link`

Result:

change the liking value of the given social link

Examples:

```
social_link set_liking 0.4
```

set_modality

Possible uses:

- `mental_state set_modality string` ---> `mental_state`
- `set_modality (mental_state, string)` ---> `mental_state`

Result:

change the modality value of the given mental state

Examples:

```
mental state set_modality belief
```

set_predicate

Possible uses:

- `mental_state set_predicate predicate` ---> `mental_state`
- `set_predicate (mental_state, predicate)` ---> `mental_state`

Result:

change the predicate value of the given mental state

Examples:

```
mental state set_predicate pred1
```

set_solidarity

Possible uses:

- `social_link set_solidarity float` ---> `social_link`
- `set_solidarity (social_link, float)`---> `social_link`

Result:

change the solidarity value of the given social link

Examples:

```
social_link set_solidarity 0.4
```

set_strength

Possible uses:

- `mental_state set_strength float` ---> `mental_state`
- `set_strength (mental_state, float)`---> `mental_state`

Result:

change the strength value of the given mental state

Examples:

```
mental_state set_strength 1.0
```

set_trust

Possible uses:

- `social_link set_trust float` ---> `social_link`
- `set_trust (social_link, float)`---> `social_link`

Result:

change the trust value of the given social link

Examples:

```
social_link set_familiarity 0.4
```

set_truth

Possible uses:

- `predicate set_truth bool` ---> `predicate`
- `set_truth (predicate, bool)`---> `predicate`

Result:

change the is_true value of the given predicate

Examples:

```
predicate set_truth false
```

set_z**Possible uses:**

- `geometry set_z [container<unknown, float> ---> geometry]`
- `set_z (geometry , container<unknown, float>) ---> geometry`
- `set_z (geometry , int , float) ---> geometry`

Result:

Sets the z ordinate of the n-th point of a geometry to the value provided by the third argument

Examples:

```
triangle(3) set_z [5,10,14]  
set_z (triangle(3), 1, 3.0)
```

shape_file**Possible uses:**

- `shape_file (string) ---> file`
- `string shape_file int ---> file`
- `shape_file (string , int) ---> file`
- `string shape_file string ---> file`
- `shape_file (string , string) ---> file`
- `string shape_file bool ---> file`
- `shape_file (string , bool) ---> file`
- `shape_file (string , int , bool) ---> file`
- `shape_file (string , string , bool) ---> file`

Result:

Constructs a file of type shape. Allowed extensions are limited to shp, SHP

Special cases:

- `shape_file(string):` This file constructor allows to read a shapefile (.shp) file

```
file f <- shape_file("file.shp");
```

- `shape_file(string,int):` This file constructor allows to read a shapefile (.shp) file and specifying the coordinates system code, as an int (epsg code)

```
file f <- shape_file("file.shp", "32648");
```

- `shape_file(string,string)`: This file constructor allows to read a shapefile (.shp) file and specifying the coordinates system code (epg,...,), as a string

```
file f <- shape_file("file.shp", "EPSG:32648");
```

- `shape_file(string,bool)`: This file constructor allows to read a shapefile (.shp) file and take a potential z value (not taken in account by default)

```
file f <- shape_file("file.shp", true);
```

- `shape_file(string,int,bool)`: This file constructor allows to read a shapefile (.shp) file and specifying the coordinates system code, as an int (epsg code) and take a potential z value (not taken in account by default)

```
file f <- shape_file("file.shp", "32648", true);
```

- `shape_file(string,string,bool)`: This file constructor allows to read a shapefile (.shp) file and specifying the coordinates system code (epg,...,), as a string and take a potential z value (not taken in account by default)

```
file f <- shape_file("file.shp", "EPSG:32648",true);
```

See also: [is_shape](#),

sharpened

Possible uses:

- `sharpened (image) ---> image`

Result:

Application of a sharpening filter to the image passed in parameter. This operation can be applied multiple times. The original image is left untouched

shuffle

Possible uses:

- `shuffle (string)---> string`
- `shuffle (matrix)---> matrix`
- `shuffle (container)---> list`

Result:

The elements of the operand in random order.

Special cases:

- if the operand is empty, returns an empty list (or string, matrix)

Examples:

```
string var0 <- shuffle ('abc'); // var0 equals 'bac' (for example)
matrix var1 <- shuffle (matrix([["c11","c12","c13"], ["c21","c22","c23"]])); // var1 equals
matrix([["c12","c21","c11"], ["c13","c22","c23"]]) (for example)
list var2 <- shuffle ([12, 13, 14]); // var2 equals [14,12,13] (for example)
```

See also: [reverse](#),

signum

Possible uses:

- `signum (float) -> int`
- `signum (int) -> 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 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(-12.8); // var0 equals -1
int var1 <- signum(14.5); // var1 equals 1
int var2 <- signum(0.0); // var2 equals 0
int var3 <- signum(-12); // var3 equals -1
int var4 <- signum(14); // var4 equals 1
int var5 <- signum(0); // var5 equals 0
```

simple_clustering_by_distance

Possible uses:

- `container<unknown,agent> simple_clustering_by_distance float -> list<list<agent>>`
- `simple_clustering_by_distance (container<unknown,agent>, float) -> list<list<agent>>`

Result:

A list of agent groups clustered by distance considering a distance min between two groups.

Examples:

```
list<list<agent>> var0 <- [ag1, ag2, ag3, ag4, ag5] simpleClusteringByDistance 20.0; // var0 equals for example, can return
[[ag1, ag3], [ag2], [ag4, ag5]]
```

See also: [hierarchical_clustering](#),

simple_clustering_by_envelope_distance

Same signification as `simple_clustering_by_distance`

simplification

Possible uses:

- `geometry simplification float` ---> `geometry`
- `simplification (geometry, float)` ---> `geometry`

Result:

A geometry corresponding to the simplification of the operand (geometry, agent, point) considering a tolerance distance.

Comment:

The algorithm used for the simplification is Douglas-Peucker

Examples:

```
geometry var0 <- self simplification 0.1; // var0 equals the geometry resulting from the application of the Douglas-Peucker algorithm on the geometry of the agent applying the operator with a tolerance distance of 0.1.
```

sin

Possible uses:

- `sin (float)` ---> `float`
- `sin (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) with_precision 10 with_precision 10; // var0 equals 0.0
float var1 <- sin (0); // var1 equals 0.0
```

See also: [cos](#), [tan](#),

sin_rad

Possible uses:

- `sin_rad (float)` ---> `float`

Result:

Returns the value (in [-1,1]) of the sinus of the operand (in radians).

Examples:

```
float var0 <- sin_rad(0); // var0 equals 0.0
```

See also: [cos_rad](#), [tan_rad](#),

since

Possible uses:

- `since (date) --> bool`
- `any expression since date --> bool`
- `since (any expression , date) --> bool`

Result:

Returns true if the current_date of the model is after (or equal to) the date passed in argument. Synonym of 'current_date >= argument'. Can be used, like 'after', in its composed form with 2 arguments to express the lowest boundary of the computation of a frequency. However, contrary to 'after', there is a subtle difference: the lowest boundary will be tested against the frequency as well

Examples:

```
reflex when: since(starting_date) {}      // this reflex will always be run
every(2#days) since (starting_date + 1#day) // the computation will return true 1 day after the starting date and every two
days after this reference date
```

skeletonize

Possible uses:

- `skeletonize (geometry) --> list<geometry>`
- `geometry skeletonize float --> list<geometry>`
- `skeletonize (geometry , float) --> list<geometry>`
- `skeletonize (geometry , float , float) --> list<geometry>`
- `skeletonize (geometry , float , float , bool) --> list<geometry>`

Result:

A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent)

Special cases:

- It can be used with 2 additional float operands: the tolerances for the clipping and for the triangulation
- It can be used with 3 additional float operands: the tolerance for the clipping, the tolerance for the triangulation, and the approximation for the clipping.
- It can be used with 1 additional float operand: the tolerance for the clipping.

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.
```

skew

Same signification as [skewness](#)

skew_gauss

Possible uses:

- `skew_gauss (float, float, float, float) --> float`

Result:

A value from a skew normally distributed random variable with min value (the minimum skewed value possible), max value (the maximum skewed value possible), skew (the degree to which the values cluster around the mode of the distribution; higher values mean tighter clustering) and bias (the tendency of the mode to approach the min, max or midpoint value; positive values bias toward max, negative values toward min). The algorithm was taken from <http://stackoverflow.com/questions/5853187/skewing-java-random-number-generation-toward-a-certain-number>

Examples:

```
float var0 <- skew_gauss(0.0, 1.0, 0.7, 0.1); // var0 equals 0.1729218460343077
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [truncated_gauss](#), [weibull_rnd](#),

skewness

Possible uses:

- `skewness (list) --> float`

Result:

returns skewness value computed from the operand list of values

Special cases:

- if the length of the list is lower than 3, returns NaN

Examples:

```
float var0 <- skewness ([1,2,3,4,5]); // var0 equals 0.0
```

skill

Possible uses:

- `skill (any) --> skill`

Result:

casts the operand in a skill object.

smooth

Possible uses:

- `geometry smooth float --> geometry`
- `smooth (geometry, float) --> geometry`

Result:

Returns a 'smoothed' geometry, where straight lines are replaced by polynomial (bicubic) curves. The first parameter is the original geometry, the second is the 'fit' parameter which can be in the range 0 (loose fit) to 1 (tightest fit).

Examples:

```
geometry var0 <- smooth(square(10), 0.0); // var0 equals a 'rounded' square
```

snapshot**Possible uses:**

- `snapshot (string) --> image`
- `agent snapshot string --> image`
- `snapshot (agent, string) --> image`
- `snapshot (agent, string, point) --> image`

Result:

Takes a snapshot of the display whose name is passed in parameter and returns the image. The search for the display begins in the current agent's simulation and, if not found, its experiment. Returns nil if no display can be found or the snapshot cannot be taken. Takes a snapshot of the display whose name is passed in parameter and returns the image. The search for the display begins in the agent passed in parameter and, if not found, its experiment. A custom size (a point representing width x height) can be given Returns nil if no display can be found or the snapshot cannot be taken. Takes a snapshot of the display whose name is passed in parameter and returns the image. The search for the display begins in the agent passed in parameter and, if not found, its experiment. The size of the snapshot will be that of the viewReturns nil if no display can be found or the snapshot cannot be taken.

sobolAnalysis**Possible uses:**

- `sobolAnalysis (string, [string], int) --> string`

Result:

Return a string containing the Report of the sobol analysis for the corresponding .csv file and save this report in a txt file.

social_link**Possible uses:**

- `social_link (any) --> social_link`

Result:

casts the operand in a social_link object.

solid

Same signification as [without_holes](#)

sort

Same signification as [sort_by](#)

sort_by

Possible uses:

- `container sort_by any expression --> list`
- `sort_by (container, 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. If the sorting function returns values that cannot be compared, an error will be thrown as well

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](#),

source_of

Possible uses:

- `graph source_of unknown --> unknown`
- `source_of (graph, unknown) --> unknown`

Result:

returns the source of the edge (right-hand operand) contained in the graph given in left-hand operand.

Special cases:

- if the left-hand operand (the graph) is nil, throws an Exception

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species":edge, "vertices_specy":node, "size":3, "m":5] );
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1
graph graphFromMap <- as_edge_graph([{1,5}:{12,45}, {12,45}:{34,56}]);
```

See also: [target_of](#),

spatial_graph

Possible uses:

- `spatial_graph` (`container`) --> `graph`

Result:

allows to create a spatial graph from a container of vertices, without trying to wire them. The container can be empty. Emits an error if the contents of the container are not geometries, points or agents

See also: [graph](#),

species

Possible uses:

- `species` (`any`) --> `species`

Result:

casts the operand in a species object.

Special cases:

- if the operand is nil, returns nil;
- if the operand is an agent, returns its species;
- if the operand is a string, returns the species with this name (nil if not found);
- otherwise, returns nil

Examples:

```
species var0 <- species(self); // var0 equals the species of the current agent
species var1 <- species('node'); // var1 equals node
species var2 <- species([1,5,9,3]); // var2 equals nil
species var3 <- species(node1); // var3 equals node
```

species_of

Possible uses:

- `species_of` (`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
```

sphere

Possible uses:

- `sphere (float) --> geometry`

Result:

A sphere geometry which radius is equal to the operand.

Comment:

the centre of the sphere is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- sphere(10); // var0 equals a geometry as a circle of radius 10 but displays a sphere.
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

split

Possible uses:

- `split (list<unknown>) --> list<list<unknown>>`

Result:

Splits a list of numbers into $n=(1+3.3\log_{10}(\text{elements}))$ bins. The splitting is strict (i.e. elements are in the i th bin if they are strictly smaller than the i th bound)

Examples:

```
list<list<unknown>> var0 <- split([1.0,2.0,1.0,3.0,1.0,2.0]); // var0 equals [[1.0,1.0,1.0],[2.0,2.0],[3.0]]
```

See also: [split_in](#), [split_using](#),

split_at

Possible uses:

- `geometry split_at point --> list<geometry>`
- `split_at (geometry , point) --> list<geometry>`

Result:

The two part of the left-operand lines split at the given right-operand point

Special cases:

- if the left-operand is a point or a polygon, returns an empty list

Examples:

```
list<geometry> var0 <- polyline([{{1,2},{4,6}}]) split_at {7,6}; // var0 equals [polyline([{{1.0,2.0},{7.0,6.0}}]), polyline([{{7.0,6.0},{4.0,6.0}}])]
```

split_geometry

Possible uses:

- `geometry split_geometry float --> list<geometry>`
- `split_geometry (geometry, float) --> list<geometry>`
- `geometry split_geometry point --> list<geometry>`
- `split_geometry (geometry, point) --> list<geometry>`
- `split_geometry (geometry, int, int) --> list<geometry>`

Result:

A list of geometries that result from the decomposition of the geometry by square cells of the given side size (geometry, size). It can be used to split in rectangles by giving a point or 2 integer values as operand.

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
```

split_in

Possible uses:

- `list<unknown> split_in int --> list<list<unknown>>`
- `split_in (list<unknown>, int) --> list<list<unknown>>`
- `split_in (list<unknown>, int, bool) --> list<list<unknown>>`

Result:

Splits a list of numbers into n bins defined by n-1 bounds between the minimum and maximum values found in the first argument. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith bound) Splits a list of numbers into n bins defined by n-1 bounds between the minimum and maximum values found in the first argument. The boolean argument controls whether or not the splitting is strict (if true, elements are in the ith bin if they are strictly smaller than the ith bound)

Examples:

```
list<float> li <- [1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0];
list<list<unknown>> var1 <- split_in(li,3); // var1 equals [[1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0],[19.9],[35.9,40.0]]
list<float> l <- [1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0];
list<list<unknown>> var3 <- split_in(l,3, true); // var3 equals [[1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0],[19.9],[35.9,40.0]]
```

See also: [split](#), [split_using](#),

split_lines

Possible uses:

- `split_lines (container<unknown,geometry>) -> list<geometry>`
- `container<unknown, geometry> split_lines bool -> list<geometry>`
- `split_lines (container<unknown,geometry>, bool) -> list<geometry>`

Result:

A list of geometries resulting after cutting the lines at their intersections. A list of geometries resulting after cutting the lines at their intersections. if the last boolean operand is set to true, the split lines will import the attributes of the initial lines

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}}])
list<geometry> var1 <- split_lines([line([{{0,10}, {20,10}}]), line([{{0,10}, {20,10}}])]); // var1 equals a list of four
polylines: line([{{0,10}, {10,10}}], line([{{10,10}, {20,10}}]), line([{{10,0}, {10,10}}]) and line([{{10,10}, {10,20}}])
```

split_using

Possible uses:

- `list<unknown> split_using list<unknown> -> list<list<unknown>>`
- `split_using (list<unknown>, list<unknown>) -> list<list<unknown>>`
- `split_using (list<unknown>, list<unknown>, bool) -> list<list<unknown>>`

Result:

Splits a list of numbers into n+1 bins using a set of n bounds passed as the second argument. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith bound), when no boolean attribute is specified.

Examples:

```
list<float> li <- [1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0];
list<list<unknown>> var1 <- split_using(li,[1.0,3.0,4.2]); // var1 equals
[[[],[1.0],[3.1],[5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0]]
list<float> l <- [1.0,3.1,5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0];
list<list<unknown>> var3 <- split_using(l,[1.0,3.0,4.2], true); // var3 equals
[[[],[1.0],[3.1],[5.2,6.0,9.2,11.1,12.0,13.0,19.9,35.9,40.0]]
```

See also: [split](#), [split_in](#),

split_with

Possible uses:

- `string split_with string --> list`
- `split_with (string, string) --> list`
- `split_with (string, string, bool) --> 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.

Special cases:

- when used with an additional boolean operand, it returns a list containing the sub-strings (tokens) of the left-hand operand delimited either by each of the characters of the right-hand operand (false) or by the whole right-hand operand (true).

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']
list var1 <- 'aa:bb:cc' split_with ('::', true); // var1 equals ['aa', 'bb:cc']
list var2 <- 'aa:bb:cc' split_with ('::', false); // var2 equals ['aa', 'bb', 'cc']
```

sqrt

Possible uses:

- `sqrt (float) --> float`
- `sqrt (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
```

square

Possible uses:

- `square (float) --> geometry`

Result:

A square geometry which side size is equal to the operand.

Comment:

the centre of the square is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.

Examples:

```
geometry var0 <- square(10); // var0 equals a geometry as a square of side size 10.  
float var1 <- var0.area; // var1 equals 100.0
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [triangle](#),

squircle

Possible uses:

- `float squircle(float float ---> geometry)`
- `squircle (float , float)---> geometry`

Result:

A mix of square and circle geometry (see : <http://en.wikipedia.org/wiki/Squircle>), which side size is equal to the first operand and power is equal to the second operand

Comment:

the center of the ellipse is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the side operand is lower or equal to 0.

Examples:

```
geometry var0 <- squircle(4,4); // var0 equals a geometry as a squircle of side 4 with a power of 4.
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [super_ellipse](#), [rectangle](#), [square](#), [circle](#), [ellipse](#), [triangle](#),

stack

Possible uses:

- `stack ([list<int>)---> unknown<string>`
- `stack (map<unknown,int>)---> [unknown<string>]`

Result:

Creates a stack layout node. Stacks can only contain one or several indices of displays (without weight) Creates a stack layout node. Accepts the same argument as [horizontal](#) or [vertical](#) (a map of display indices and weights) but the weights are not taken into account

standard_deviation

Possible uses:

- `standard_deviation` (`container`) --> `float`

Result:

the standard deviation on the elements of the operand. See [Standard_deviation](#) for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- standard_deviation ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.2930100540985752
```

See also: [mean](#), [mean_deviation](#),

step_sub_model

Possible uses:

- `step_sub_model` (`agent`) --> `int`

Result:

Load a submodel

Comment:

loaded submodel

strahler

Possible uses:

- `strahler` (`graph`) --> `map`

Result:

return for each edge, its strahler number

string

Possible uses:

- `date string string` --> `string`
- `string (date, string)` --> `string`
- `string (date, string, string)` --> `string`

Result:

converts a date to a string following a custom pattern. The pattern can use "%Y %M %N %D %E %h %m %s %z" for outputting years, months, name of month, days, name of days, hours, minutes, seconds and the time-zone. A null or empty pattern will return the complete date as defined by the ISO date & time format. The pattern can also follow the pattern definition found here, which gives much more control over the format of the date: <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns>. Different patterns are available by default as constants: #iso_local, #iso_simple, #iso_Offset, #iso_zoned and #custom, which can be changed in the preferences

Examples:

```
string(#now, 'yyyy-MM-dd', 'en')
string(#now, 'yyyy-MM-dd')
```

student_area**Possible uses:**

- `float student_area int --> float`
- `student_area (float, int) --> float`

Result:

Returns the area to the left of x in the Student T distribution with the given degrees of freedom.

Examples:

```
float var0 <- student_area(1.64,3) with_precision(2); // var0 equals 0.9
```

student_t_inverse**Possible uses:**

- `float student_t_inverse int --> float`
- `student_t_inverse (float, int) --> float`

Result:

Returns the value, t, for which the area under the Student-t probability density function (integrated from minus infinity to t) is equal to x.

Examples:

```
float var0 <- student_t_inverse(0.9,3) with_precision(2); // var0 equals 1.64
```

subtract_days

Same signification as [minus_days](#)

subtract_hours

Same signification as [minus_hours](#)

subtract_minutes

Same signification as [minus_minutes](#)

subtract_months

Same signification as [minus_months](#)

subtract_ms

Same signification as [minus_ms](#)

subtract_seconds

Same signification as -

subtract_weeks

Same signification as [minus_weeks](#)

subtract_years

Same signification as [minus_years](#)

successors_of

Possible uses:

- `graph successors_of unknown ---> list`
- `successors_of (graph, 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](#), [neighbors_of](#),

sum

Possible uses:

- `sum (graph) ---> float`
- `sum (container) ---> unknown`

Result:

the sum of all the elements of the operand

Comment:

the behavior depends on the nature of the operand

Special cases:

- if it is a population or a list of other types: sum transforms all elements into float and sums them
- if it is a map, sum returns the sum of the value of all elements
- if it is a file, sum returns the sum of the content of the file (that is also a container)
- if it is a graph, sum returns the total weight of 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 other types: sum transforms all elements into float and sums them
- if it is a list of colors: sum will sum them and return the blended resulting color
- if it is a list of int or float: sum returns the sum of all the elements

```
int var0 <- sum ([12,10,3]); // var0 equals 25
```

- if it is a list of points: sum returns the sum of all points as a point (each coordinate is the sum of the corresponding coordinate of each element)

```
unknown var1 <- sum([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]); // var1 equals {20.0,17.0}
```

See also: [mul](#),

sum_of**Possible uses:**

- `container sum_of any_expression --> unknown`
- `sum_of (container, any_expression) --> unknown`

Result:

the sum of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-operand is a map, the keyword each will contain each value

```
unknown var1 <- [1::2, 3::4, 5::6] sum_of (each + 3); // var1 equals 21
```

Examples:

```
unknown var0 <- [1,2] sum_of (each * 100 ); // var0 equals 300
```

See also: [min_of](#), [max_of](#), [product_of](#), [mean_of](#),

svg_file

Possible uses:

- `svg_file (string) --> file`
- `string svg_file (point) --> file`
- `svg_file (string, point) --> file`

Result:

Constructs a file of type svg. Allowed extensions are limited to svg

Special cases:

- `svg_file(string)`: This file constructor allows to read a svg file

```
file f <-svg_file("file.svg");
```

- `svg_file(string,point)`: This file constructor allows to read a svg file, specifying the size of the bounding box

```
file f <-svg_file("file.svg", {10,10});
```

See also: [is_svg](#),

t_test

Possible uses:

- `list t_test list --> float`
- `t_test (list, list) --> float`

Result:

Returns the observed significance level, or p-value, associated with a two-sample, two-tailed t-test comparing the means of the two input lists. The number returned is the smallest significance level at which one can reject the null hypothesis

Examples:

```
float var0 <- t_test([10.0,5.0,1.0, 3.0],[1.0,10.0,5.0,1.0]); // var0 equals 0.01
```

tan

Possible uses:

- `tan (int) --> float`
- `tan (float) --> float`

Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in decimal degrees).

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 $90 + k \cdot 180$ (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](#),

tan_rad

Possible uses:

- `tan_rad (float) ---> float`

Result:

Returns the value (in [-1,1]) of the trigonometric tangent of the operand (in radians).

Examples:

```
float var0 <- tan_rad(0); // var0 equals 0.0
```

See also: [cos_rad](#), [sin_rad](#),

tanh

Possible uses:

- `tanh (int) ---> float`
- `tanh (float) ---> float`

Result:

Returns the value (in the interval [-1,1]) of the hyperbolic tangent of the operand (which can be any real number, expressed in decimal degrees).

Examples:

```
float var0 <- tanh(0); // var0 equals 0.0
float var1 <- tanh(100); // var1 equals 1.0
```

target_of

Possible uses:

- `graph target_of unknown ---> unknown`
- `target_of (graph , unknown) ---> unknown`

Result:

returns the target of the edge (right-hand operand) contained in the graph given in left-hand operand.

Special cases:

- if the left-hand operand (the graph) is nil, returns nil

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"]::edge, "vertices_specy"]::node, "size"]::3, "m"]::5 );
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1
graph graphFromMap <- as_edge_graph([{1,5}:{12,45},{12,45}:{34,56}]);
unknown var3 <- graphFromMap target_of(link({1,5},{12,45})); // var3 equals {12,45}
```

See also: [source_of](#),

teapot

Possible uses:

- `teapot` (float) --> geometry

Result:

A teapot geometry which radius is equal to the operand.

Comment:

the centre of the teapot is by default the location of the current agent in which has been called this operator.

Special cases:

- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- teapot(10); // var0 equals a geometry as a circle of radius 10 but displays a teapot.
```

See also: [around](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#), [triangle](#),

text_file

Possible uses:

- `text_file` (string) --> file
- string `text_file` list<string> --> file
- `text_file` (string, list<string>) --> file

Result:

Constructs a file of type text. Allowed extensions are limited to txt, data, text

Special cases:

- `text_file(string)`: This file constructor allows to read a text file (.txt, .data, .text)

```
file f <-text_file("file.txt");
```

- `text_file(string,list<string>)`: This file constructor allows to store a list of string in a text file (it does not save it - just store it in memory)

```
file f <- text_file("file.txt", ["item1","item2","item3"]);
```

See also: [is_text](#),

TGauss

Same signification as [truncated_gauss](#)

threeds_file

Possible uses:

- `threeds_file(string) --> file`

Result:

Constructs a file of type threeds. Allowed extensions are limited to 3ds, max

Special cases:

- `threeds_file(string)`: This file constructor allows to read a 3DS Max file. Only loads vertices and faces

```
threeds_file f <- threeds_file("file");
```

See also: [is_threeds](#),

tinted_with

Possible uses:

- `image tinted_with rgb --> image`
- `tinted_with (image, rgb) --> image`
- `tinted_with (image, rgb, float) --> image`

Result:

Returns the image tinted using the color passed in parameter and a factor between 0 and 1, determining the transparency of the dyeing to apply.
The original image is left untouched Returns the image tinted using the color passed in parameter. This effectively multiplies the colors of the image by it. The original image is left untouched

to

Same signification as [until](#)

Possible uses:

- `date to date --> list<date>`
- `to (date, date) --> list<date>`

Result:

builds an interval between two dates (the first inclusive and the second exclusive, which behaves like a read-only list of dates. The default step between two dates is the step of the model

Comment:

The default step can be overruled by using the every operator applied to this interval

Examples:

```
date('2000-01-01') to date('2010-01-01') // builds an interval between these two dates
(date('2000-01-01') to date('2010-01-01')) every (#day) // builds an interval between these two dates which contains all the days starting from the beginning of the interval. Beware that using every with #month or #year will produce odd results, as these pseudo-constants are not constant; only the first value will be used to compute the intervals (if current_date is set to a month of February, #month will only represent 28 or 29 days depending on whether it is a leap year or not !). If such intervals need to be built, it is recommended to use a generative way, for instance a loop using the 'plus_years' or 'plus_months' operators to build a list of dates
```

See also: [every](#),

to_GAMA_CRS

Possible uses:

- `to_GAMA_CRS (geometry) --> geometry`
- `geometry to_GAMA_CRS string --> geometry`
- `to_GAMA_CRS (geometry , string) --> geometry`

Special cases:

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by the current CRS, the one corresponding to the world's agent one

```
geometry var0 <- to_GAMA_CRS({121,14}); // var0 equals a geometry corresponding to the agent geometry transformed into the GAMA CRS
```

- returns the geometry corresponding to the transformation of the given geometry to the GAMA CRS (Coordinate Reference System) assuming the given geometry is referenced by given CRS

```
geometry var1 <- to_GAMA_CRS({121,14}, "EPSG:4326"); // var1 equals a geometry corresponding to the agent geometry transformed into the GAMA CRS
```

to_gaml

Possible uses:

- `to_gaml (unknown) --> string`

Result:

returns the literal description of an expression or description -- action, behavior, species, aspect, even model -- in gaml

Examples:

```
string var0 <- to_gaml(0); // var0 equals '0'
string var1 <- to_gaml(3.78); // var1 equals '3.78'
string var2 <- to_gaml({23, 4.0}); // var2 equals '{23.0,4.0,0.0}'
string var3 <- to_gaml(rgb(255,0,125)); // var3 equals 'rgb (255, 0, 125,255)'
string var4 <- to_gaml('hello'); // var4 equals "hello"
```

to_geojson

Possible uses:

- `to_geojson` (`any expression`, `string`, `any expression`) \rightarrow `string`

Result:

returns geojson of species with crs

Examples:

```
string var0 <- to_geojson(boat, "EPSG:4326", ["color"]); // var0 equals
{"type": "FeatureCollection", "features": [{"type": "Feature", "geometry": {"type": "Point", "coordinates": [100.51155642068785, 3.514781609095577E-4, 0.0]}]}
```

to_hsb

Possible uses:

- `to_hsb` (`rgb`) \rightarrow `list<float>`

Result:

Converts a Gama color to hsb (h=hue, s=saturation, b=brightness) value

Examples:

```
list<float> var0 <- to_hsb (#cyan); // var0 equals [0.5, 1.0, 1.0]
```

to_list

Possible uses:

- `to_list` (`unknown`) \rightarrow `list`

Result:

casts the operand to a list, making an explicit copy if it is already a list or a subtype of list (interval, population, etc.)

See also: [list](#),

to_rectangles

Same signification as [split_geometry](#)

Possible uses:

- `to_rectangles` (`geometry`, `point`, `bool`) \rightarrow `list<geometry>`
- `to_rectangles` (`geometry`, `int`, `int`, `bool`) \rightarrow `list<geometry>`

Result:

A list of rectangles of the size corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry,

dimension, overlaps), if overlaps = true, add the rectangles that overlap the border of the geometry

Examples:

```
list<geometry> var0 <- to_rectangles(self, 5, 20, true); // var0 equals the list of rectangles corresponding to the  
discretization by a grid of 5 columns and 20 rows into rectangles of the geometry of the agent applying the operator. The  
rectangles overlapping the border of the geometry are kept  
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 discretization into rectangles of the geometry of the agent applying the operator. The rectangles  
overlapping the border of the geometry are kept
```

to_segments

Possible uses:

- `to_segments(geometry) --> list<geometry>`

Result:

A list of segments resulting from the decomposition of the geometry (or its contours for polygons) into segments

Examples:

```
list<geometry> var0 <- to_segments(line([{10,10},{80,10},{80,80}])); // var0 equals [line([{10,10},{80,10}]),  
line([{80,10},{80,80}])]
```

to_squares

Possible uses:

- `to_squares(geometry, int, bool) --> list<geometry>`
- `to_squares(geometry, float, bool) --> list<geometry>`
- `to_squares(geometry, int, bool, float) --> list<geometry>`

Result:

A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps), if overlaps = true, add the squares that overlap the border of the geometry A list of a given number of squares from the decomposition of the geometry into squares (geometry, nb_square, overlaps, precision_coefficient), if overlaps = true, add the squares that overlap the border of the geometry, coefficient_precision should be close to 1.0 A list of squares of the size corresponding to the given size that result from the decomposition of the geometry into squares (geometry, size, overlaps), if overlaps = true, add the squares that overlap the border of the geometry

Examples:

```
list<geometry> var0 <- to_squares(self, 10, true); // var0 equals the list of 10 squares corresponding to the discretization  
into squares of the geometry of the agent applying the operator. The squares overlapping the border of the geometry are kept  
list<geometry> var1 <- to_squares(self, 10, true, 0.99); // var1 equals the list of 10 squares corresponding to the  
discretization into squares of the geometry of the agent applying the operator. The squares overlapping the border of the  
geometry are kept  
list<geometry> var2 <- to_squares(self, 10.0, true); // var2 equals the list of squares of side size 10.0 corresponding to  
the discretization into squares of the geometry of the agent applying the operator. The squares overlapping the border of  
the geometry are kept
```

to_sub_geometries

Possible uses:

- `geometry to_sub_geometries [list<float> ---> list<geometry>`
- `to_sub_geometries (geometry, list<float>) ---> list<geometry>`
- `to_sub_geometries (geometry, list<float>, float) ---> list<geometry>`

Result:

A list of geometries resulting after splitting the geometry into sub-geometries.

Examples:

```
list<geometry> var0 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4], 1.0); // var0 equals a list of three geometries corresponding to 3 sub-geometries using cubes of 1m size
list<geometry> var1 <- to_sub_geometries(rectangle(10, 50), [0.1, 0.5, 0.4]); // var1 equals a list of three geometries corresponding to 3 sub-geometries
```

to_triangles

Same signification as [triangulate](#)

tokenize

Same signification as [split_with](#)

topology

Possible uses:

- `topology (any) ---> topology`

Result:

casts the operand in a topology object.

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(); // var0 equals nil
topology(a_graph)  --: Multiple topology in POLYGON ((24.712119771887785 7.867357373616512, 24.712119771887785
61.283226839310565, 82.4013676510046 7.867357373616512)) at location[53.556743711446195;34.57529210646354]
```

See also: [geometry](#),

touches

Possible uses:

- `geometry touches geometry --> bool`
- `touches (geometry , geometry) --> bool`

Result:

A boolean, equal to true if the left-geometry (or agent/point) touches the right-geometry (or agent/point).

Comment:

returns true when the left-operand only touches the right-operand. When one geometry covers partially (or fully) the other one, it returns false.

Special cases:

- if one of the operand is null, returns false.

Examples:

```
bool var0 <- {15,15} touches {15,15}; // var0 equals false
bool var1 <- polyline([{10,10},{20,20}]) touches {10,10}; // var1 equals true
bool var2 <- polyline([{10,10},{20,20}]) touches polyline([{10,10},{5,5}]); // var2 equals true
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon([{15,15},{15,25},{25,25},{25,15}]); // var3 equals false
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches {10,15}; // var4 equals true
```

See also: [disjoint_from](#), [crosses](#), [overlaps](#), [partially_overlaps](#), [intersects](#),

touching

Possible uses:

- `container<unknown,geometry> touching geometry --> list<geometry>`
- `touching (container<unknown,geometry> , geometry) --> list<geometry>`

Result:

A list of agents or geometries among the left-operand list, species or meta-population (addition of species), touching the operand (casted as a geometry).

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] toucing(self); // var0 equals the agents among ag1, ag2 and ag3 that touch the shape of the right-hand argument.
list<geometry> var1 <- (species1 + species2) touching (self); // var1 equals the agents among species species1 and species2 that touch the shape of the right-hand argument.
```

See also: [neighbors_at](#), [neighbors_of](#), [closest_to](#), [overlapping](#), [agents_overlapping](#), [inside](#), [agents_inside](#), [agent_closest_to](#),

towards

Possible uses:

- `geometry towards geometry --> float`
- `towards (geometry, geometry) --> float`

Result:

The direction (in degree) between the two geometries (geometries, agents, points) considering the topology of the agent applying the operator.

Examples:

```
float var0 <- ag1 towards ag2; // var0 equals the direction between ag1 and ag2 and ag3 considering the topology of the agent applying the operator
```

See also: [distance_between](#), [distance_to](#), [direction_between](#), [path_between](#), [path_to](#),

trace

Possible uses:

- `trace (matrix) --> float`

Result:

The trace of the given matrix (the sum of the elements on the main diagonal).

Examples:

```
float var0 <- trace(matrix([[1,2],[3,4]])); // var0 equals 5
```

transformed_by

Possible uses:

- `geometry transformed_by point --> geometry`
- `transformed_by (geometry, point) --> geometry`

Result:

A geometry resulting from the application of a rotation and a scaling (right-operand : point {angle(degree), scale factor} of the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self transformed_by {45, 0.5}; // var0 equals the geometry resulting from 45 degrees rotation and 50% scaling of the geometry of the agent applying the operator.
```

See also: [rotated_by](#), [translated_by](#),

translated_by

Possible uses:

- `geometry translated_by point --> geometry`
- `translated_by(geometry, point) --> geometry`

Result:

A geometry resulting from the application of a translation by the right-hand operand distance to the left-hand operand (geometry, agent, point)

Examples:

```
geometry var0 <- self translated_by {10,10,10}; // var0 equals the geometry resulting from applying the translation to the left-hand geometry (or agent).
```

See also: [rotated_by](#), [transformed_by](#),

translated_to

Same signification as [at_location](#)

transpose

Possible uses:

- `transpose(matrix) --> matrix`

Result:

The transposition of the given matrix

Examples:

```
matrix var0 <- transpose(matrix([[5,-3],[6,-4]])); // var0 equals matrix([[5,6],[-3,-4]])
```

triangle

Possible uses:

- `triangle(float) --> geometry`
- `float triangle float --> geometry`
- `triangle(float, float) --> geometry`

Result:

A triangle geometry which side size is given by the operand. A triangle geometry which the base and height size are given by the operand.

Comment:

the center of the triangle is by default the location of the current agent in which has been called this operator.the center of the triangle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if one of the operand is nil.

Examples:

```
geometry var0 <- triangle(5); // var0 equals a geometry as a triangle with side_size = 5.  
geometry var1 <- triangle(5, 10); // var1 equals a geometry as a triangle with a base of 5m and a height of 10m.
```

See also: [around](#), [circle](#), [cone](#), [line](#), [link](#), [norm](#), [point](#), [polygon](#), [polyline](#), [rectangle](#), [square](#),

triangulate

Possible uses:

- `triangulate (list<geometry>) ---> list<geometry>`
- `triangulate (geometry) ---> list<geometry>`
- `geometry triangulate float ---> list<geometry>`
- `triangulate (geometry, float) ---> list<geometry>`
- `triangulate (geometry, float, float) ---> list<geometry>`
- `triangulate (geometry, float, float, bool) ---> list<geometry>`

Result:

A list of geometries (triangles) corresponding to the Delaunay triangulation computed from the list of polylines A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point) with the given tolerance for the clipping A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point) with the given tolerance for the clipping and for the triangulation A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point, use_approx_clipping) with the given tolerance for the clipping and for the triangulation with using an approximate clipping is the last operand is true A list of geometries (triangles) corresponding to the Delaunay triangulation of the operand geometry (geometry, agent, point)

Examples:

```
list<geometry> var0 <- triangulate([line([{{0,50},{100,50}}]), line([{{50,0},{50,100}}])); // 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, 0.1); // var1 equals the list of geometries (triangles) corresponding to the Delaunay triangulation of the geometry of the agent applying the operator.  
list<geometry> var2 <- triangulate(self, 0.1, 1.0); // var2 equals the list of geometries (triangles) corresponding to the Delaunay triangulation of the geometry of the agent applying the operator.  
list<geometry> var3 <- triangulate(self, 0.1, 1.0, true); // var3 equals the list of geometries (triangles) corresponding to the Delaunay triangulation of the geometry of the agent applying the operator.  
list<geometry> var4 <- triangulate(self); // var4 equals the list of geometries (triangles) corresponding to the Delaunay triangulation of the geometry of the agent applying the operator.
```

truncated_gauss

Possible uses:

- `truncated_gauss (point) ---> float`
- `truncated_gauss (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 a float between -0.3 and 0.3
float var1 <- truncated_gauss ([0.5, 0.0]); // var1 equals 0.5
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [weibull_rnd](#), [gamma_trunc_rnd](#), [weibull_trunc_rnd](#), [lognormal_trunc_rnd](#),

type_of

Possible uses:

- `type_of` (`unknown`) ---> `any GAML type<unknown>`

Result:

Returns the GAML type of the operand

Examples:

```
string var0 <- string(type_of("a string")); // var0 equals "string"
string var1 <- string(type_of([1,2,3,4,5])); // var1 equals "list<int>"
geometry g0 <- to_GAMA_CRS({121,14}, "EPSG:4326");
string var3 <- string(type_of(g0)); // var3 equals "point"
```

undirected

Possible uses:

- `undirected` (`graph`) ---> `graph`

Result:

the operand graph becomes an undirected graph.

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph.

See also: [directed](#),

union

Possible uses:

- `union` (`container<unknown, geometry>`) ---> `geometry`
- `container` `union` `container` ---> `list`
- `union` (`container`, `container`) ---> `list`

Result:

returns a new list containing all the elements of both containers without duplicated elements.

Special cases:

- if the left or right operand is nil, union throws an error
- if the right-operand is a container of points, geometries or agents, returns the geometry resulting from the union all the geometries

Examples:

```
list var0 <- [1,2,3,4,5,6] union [2,4,9]; // var0 equals [1,2,3,4,5,6,9]
list var1 <- [1,2,3,4,5,6] union [0,8]; // var1 equals [1,2,3,4,5,6,0,8]
list var2 <- [1,3,2,4,5,6,8,5,6] union [0,8]; // var2 equals [1,3,2,4,5,6,8,0]
geometry var3 <- union([geom1, geom2, geom3]); // var3 equals a geometry corresponding to union between geom1, geom2 and
geom3
```

See also: [inter](#), [+](#),

unknown

Possible uses:

- `unknown` (`any`) ---> `unknown`

Result:

casts the operand in a unknown object.

until

Possible uses:

- `until` (`date`) ---> `bool`
- `any expression` `until` `date` ---> `bool`
- `until` (`any expression`, `date`) ---> `bool`

Result:

Returns true if the `current_date` of the model is before (or equal to) the date passed in argument. Synonym of '`current_date <= argument`'

Examples:

```
reflex when: until(starting_date) {} // This reflex will be run only once at the beginning of the simulation
```

unzip

Possible uses:

- `string unzip string --> bool`
- `unzip(string, string) --> bool`

Result:

Unzip a given zip file into a given folder. Returns true if the file is well unzipped

Examples:

```
bool unzip_ok <- unzip(["..../includes/my_folder"], "folder.zip";
```

upper_case

Possible uses:

- `upper_case(string) --> string`

Result:

Converts all of the characters in the string operand to upper case

Examples:

```
string var0 <- upper_case("Abc"); // var0 equals 'ABC'
```

See also: [lower_case](#),

use_cache

Possible uses:

- `graph use_cache bool --> graph`
- `use_cache(graph, bool) --> graph`

Result:

if the second operand is true, the operand graph will store in a cache all the previously computed shortest path (the cache be cleared if the graph is modified).

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph.

See also: [path_between](#),

user_confirm

Possible uses:

- `string user_confirm string --> bool`

- `user_confirm(string, string) -> bool`

Result:

Asks the user to confirm a choice. The two string are used to specify the title and the message of the dialog box.

Examples:

```
bool confirm <- user_confirm("Confirm", "Please confirm");
```

user_input_dialog

Possible uses:

- `user_input_dialog(list) -> map<string, unknown>`
- `string|user_input_dialog list -> map<string, unknown>`
- `user_input_dialog(string, list) -> map<string, unknown>`
- `user_input_dialog(string, list, font) -> map<string, unknown>`
- `user_input_dialog(string, list, font, rgb) -> map<string, unknown>`
- `user_input_dialog(string, list, font, rgb, bool) -> map<string, unknown>`

Result:

Asks the user for some values and returns a map containing these values. Takes a string and a list of calls to the `enter()` or `choose()` operators as arguments. The string is used to specify the message of the dialog box. The list is used to specify the parameters the user can enter. Finally, the font of the title can be specified
Asks the user for some values and returns a map containing these values. Takes a string and a list of calls to the `enter()` or `choose()` operators as arguments. The string is used to specify the message of the dialog box. The list is used to specify the parameters the user can enter. Finally, the font of the title can be specified, as well as the background color and whether the title and close button of the dialog should be displayed or not
Asks the user for some values and returns a map containing these values. Takes a string and a list of calls to the `enter()` or `choose()` operators as arguments. The string is used to specify the message of the dialog box. The list is used to specify the parameters the user can enter. Finally, the font of the title can be specified as well as the background color
Asks the user for some values and returns a map containing these values. Takes a string and a list of calls to the `enter()` or `choose()` operators as arguments. The string is used to specify the message of the dialog box. The list is to specify the parameters the user can enter

Examples:

```
map<string, unknown> values_no_title <- user_input_dialog([enter('Number', 100), enter('Location', point, {10, 10})]);
create bug number: int(values2 at "Number") with: [location:: (point(values2 at "Location"))];
map<string, unknown> values2 <- user_input_dialog('Enter number of agents and locations', [enter('Number', 100),
enter('Location', point, {10, 10})], font('Helvetica', 18));
create bug number: int(values2 at "Number") with: [location:: (point(values2 at "Location"))];
map<string, unknown> values2 <- user_input_dialog('Enter number of agents and locations', [enter('Number', 100),
enter('Location', point, {10, 10})], font('Helvetica', 18));
create bug number: int(values2 at "Number") with: [location:: (point(values2 at "Location"))];
map<string, unknown> values2 <- user_input_dialog('Enter number of agents and locations', [enter('Number', 100),
enter('Location', point, {10, 10})], font('Helvetica', 18));
create bug number: int(values2 at "Number") with: [location:: (point(values2 at "Location"))];
map<string, unknown> values2 <- user_input_dialog('Enter number of agents and locations', [enter('Number', 100),
enter('Location', point, {10, 10})], font('Helvetica', 18));
create bug number: int(values2 at "Number") with: [location:: (point(values2 at "Location"))];
```

using

Possible uses:

- `any expression using topology -> unknown`

- `using` (`any expression`, `topology`) \rightarrow `unknown`

Result:

Allows to specify in which topology a spatial computation should take place.

Special cases:

- has no effect if the topology passed as a parameter is nil

Examples:

```
unknown var0 <- (agents closest_to self) using topology(world); // var0 equals the closest agent to self (the caller) in the continuous topology of the world
```

values_in

Possible uses:

- `field values_in geometry` \rightarrow `list<float>`
- `values_in (field, geometry)` \rightarrow `list<float>`

variance

Possible uses:

- `variance (float)` \rightarrow `float`
- `variance (container)` \rightarrow `float`
- `variance (int, float, float)` \rightarrow `float`

Result:

Returns the variance of a data sequence. That is $(\text{sumOfSquares} - \text{mean}^2 * \text{size}) / \text{size}$ with $\text{mean} = \text{sum}/\text{size}$. Returns the variance from a standard deviation. the variance of the elements of the operand. See [Variance](#) for more details.

Comment:

In the example we consider variance of [1,3,5,7]. The size is 4, the sum is $1+3+5+7=16$ and the sum of squares is 84.The variance is $(84- 16^2/4)/4$. CQFD.The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
int var0 <- int(variance(4,16,84)); // var0 equals 5
int var1 <- int(variance([1,3,5,6,9,11,12,13])); // var1 equals 17
float var2 <- variance ([4.5, 3.5, 5.5, 7.0]); // var2 equals 1.671875
```

See also: [mean](#), [median](#),

variance_of

Possible uses:

- `container variance_of any expression` \rightarrow `unknown`
- `variance_of (container, any expression)` \rightarrow `unknown`

Result:

the variance of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Examples:

```
float var0 <- [1,2,3,4,5,6] variance_of each with_precision 2; // var0 equals 2.92
```

See also: [min_of](#), [max_of](#), [sum_of](#), [product_of](#),

vertical

Possible uses:

- `vertical` (`map<unknown,int>`) ---> `unknown<string>`

Result:

Creates a vertical layout node (a sash). Sashes can contain any number (> 1) of other elements: stacks, horizontal or vertical sashes, or display indices. Each element is represented by a pair in the map, where the key is the element and the value its weight within the sash

vertical_flip

Possible uses:

- `vertical_flip` (`image`) ---> `image`

Result:

Returns an image flipped vertically by reflecting the original image around the x axis. The original image is left untouched

voronoi

Possible uses:

- `voronoi` (`list<point>`) ---> `list<geometry>`
- `list<point>` `voronoi` `geometry` ---> `list<geometry>`
- `voronoi` (`list<point>`, `geometry`) ---> `list<geometry>`

Result:

A list of geometries corresponding to the Voronoi diagram built from the list of points (with eventually a given clip).

Examples:

```
list<geometry> var0 <- voronoi([{10,10},{50,50},{90,90},{10,90},{90,10}], square(300)); // var0 equals the list of
geometries corresponding to the Voronoi Diagram built from the list of points with a square of 300m side size as clip.
list<geometry> var1 <- voronoi([{10,10},{50,50},{90,90},{10,90},{90,10}]); // vari equals the list of geometries
corresponding to the Voronoi Diagram built from the list of points.
```

weibull_density

Possible uses:

- `weibull_density` (`float`, `float`, `float`) \rightarrow `float`

Result:

`weibull_density(x,shape,scale)` returns the probability density function (PDF) at the specified point `x` of the Weibull distribution with the given shape and scale.

Examples:

```
float var0 <- weibull_rnd(1,2,3) ; // var0 equals 0.731
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [lognormal_density](#), [gamma_density](#),

weibull_rnd

Possible uses:

- `float weibull_rnd float` \rightarrow `float`
- `weibull_rnd` (`float`, `float`) \rightarrow `float`

Result:

returns a random value from a Weibull distribution with specified values of the shape (alpha) and scale (beta) parameters. See <https://mathworld.wolfram.com/WeibullDistribution.html> for more details (equations 1 and 2).

Examples:

```
float var0 <- weibull_rnd(2,3) ; // var0 equals 0.731
```

See also: [binomial](#), [gamma_rnd](#), [gauss_rnd](#), [lognormal_rnd](#), [poisson](#), [rnd](#), [skew_gauss](#), [truncated_gauss](#), [weibull_trunc_rnd](#),

weibull_trunc_rnd

Possible uses:

- `weibull_trunc_rnd` (`float`, `float`, `float`, `bool`) \rightarrow `float`
- `weibull_trunc_rnd` (`float`, `float`, `float`, `float`) \rightarrow `float`

Result:

returns a random value from a truncated Weibull distribution (in a range or given only one boundary) with specified values of the shape (alpha) and scale (beta) parameters. See <https://mathworld.wolfram.com/WeibullDistribution.html> for more details (equations 1 and 2).

Special cases:

- when 1 float and a boolean (isMax) operands are specified, the float value represents the single boundary (max if the boolean is true, min otherwise),

```
weibull_trunc_rnd(2,3,5,true)
```

- when 2 float operands are specified, they are taken as minimum and maximum values for the result

```
weibull_trunc_rnd(2,3,0.0,5.0)
```

See also: [weibull_rnd](#), [gamma_trunc_rnd](#), [lognormal_trunc_rnd](#), [truncated_gauss](#),

weight_of

Possible uses:

- `graph weight_of unknown --> float`
- `weight_of (graph , unknown) --> float`

Result:

returns the weight of the given edge (right-hand operand) contained in the graph given in right-hand operand.

Comment:

In a localized graph, an edge has a weight by default (the distance between both vertices).

Special cases:

- if the left-operand (the graph) is nil, returns nil
- if the right-hand operand is not an edge of the given graph, weight_of checks whether it is a node of the graph and tries to return its weight
- if the right-hand operand is neither a node, nor an edge, returns 1.

Examples:

```
graph graphFromMap <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
float var1 <- graphFromMap weight_of(link({1,5},{12,45})); // var1 equals 1.0
```

weighted_means_DM

Possible uses:

- `list<list> weighted_means_DM list<map<string,unknown>> --> int`
- `weighted_means_DM (list<list> , list<map<string,unknown>>) --> int`

Result:

The index of the candidate that maximizes the weighted mean of its criterion values. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion (list of map)

Special cases:

- returns -1 if the list of candidates is nil or empty

Examples:

```
int var0 <- weighted_means_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [{"name)::"utility", "weight" :: 2.0}, {"name)::"price", "weight" :: 1.0]}); // var0 equals 1
```

See also: [promethee_DM](#), [electre_DM](#), [evidence_theory_DM](#),

where

Possible uses:

- `species where any expression --> list`
- `where(species, any expression) --> list`
- `container where any expression --> list`
- `where(container, any expression) --> list`
- `list where any expression --> list`
- `where(list, 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 nil, where throws an error
- 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](#),

with_generation_algo

Possible uses:

- `gen_population_generator with_generation_algo string --> gen_population_generator`
- `with_generation_algo(gen_population_generator, string) --> gen_population_generator`

Result:

define the algorithm used for the population generation among: IS (independant hypothesis Algorothm) and simple_draw (simple draw of entities in a sample)

Examples:

```
my_pop_generator with_generation_algo "simple_draw"
```

`with_height`

Possible uses:

- `image with_height int --> image`
- `with_height (image, int) --> image`

Result:

Applies a proportional scaling to the image passed in parameter to return a new scaled image with the corresponding height. A height of 0 will return nil, a height equal to the height of the image will return the original image. Automatic scaling and resizing methods are used. The original image is left untouched

`with_k_shortest_path_algorithm`

Possible uses:

- `graph with_k_shortest_path_algorithm string --> graph`
- `with_k_shortest_path_algorithm (graph, string) --> graph`

Result:

changes the K shortest paths computation algorithm of the given graph

Comment:

the right-hand operand can be #Yen and #Bhandari to use the associated algorithm.

Examples:

```
the_graph <- the_graph with_k_shortest_path_algorithm #Yen;
```

`with_max_of`

Possible uses:

- `container with_max_of any_expression --> unknown`
- `with_max_of (container, any_expression) --> unknown`

Result:

one of elements of the left-hand operand that maximizes the value of the right-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_max_of (each); // var0 equals 8
unknown var2 <- g2 with_max_of (length(g2 out_edges_of each)) ; // var2 equals node4
```

See also: [where](#), [with_min_of](#),

with_min_of

Possible uses:

- `container with_min_of any expression ---> unknown`
- `with_min_of (container , any expression)---> unknown`

Result:

one of elements of the left-hand operand that minimizes the value of the right-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is nil, with_max_of returns the default value of the right-hand operand

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] with_min_of (each ); // var0 equals 1
unknown var2 <- g2 with_min_of (length(g2 out_edges_of each) ) ; // var2 equals node11
unknown var3 <- (list(node) with_min_of (round(node(each).location.x))); // var3 equals node0
unknown var4 <- [1::2, 3::4, 5::6] with_min_of (each); // var4 equals 2
```

See also: [where](#), [with_max_of](#),

with_precision

Possible uses:

- `float with_precision int ---> float`
- `with_precision (float , int)---> float`
- `point with_precision int ---> point`
- `with_precision (point , int)---> point`
- `geometry with_precision int ---> geometry`
- `with_precision (geometry , int)---> geometry`

Result:

Rounds off the value of left-hand operand to the precision given by the value of right-hand operand Rounds off the ordinates of the left-hand point to the precision given by the value of right-hand operand A geometry corresponding to the rounding of points of the operand considering a given precision.

Examples:

```
float var0 <- 12345.78943 with_precision 2; // var0 equals 12345.79
float var1 <- 123 with_precision 2; // var1 equals 123.00
point var2 <- {12345.78943, 12345.78943, 12345.78943} with_precision 2 ; // var2 equals {12345.79, 12345.79, 12345.79}
geometry var3 <- self with_precision 2; // var3 equals the geometry resulting from the rounding of points of the geometry
with a precision of 0.1.
```

See also: [round](#),

with_shortest_path_algorithm

Possible uses:

- `graph with_shortest_path_algorithm string --> graph`
- `with_shortest_path_algorithm(graph, string) --> graph`

Result:

changes the shortest path computation algorithm of the given graph

Comment:

the right-hand operand can be #Dijkstra, #BidirectionalDijkstra, #BellmannFord, #FloydWarshall, #Astar, #NBAStar, #NBAStarApprox, #DeltaStepping, #CHBidirectionalDijkstra, #TransitNodeRouting to use the associated algorithm.

Examples:

```
road_network <- road_network with_shortestpath_algorithm #TransitNodeRouting;
```

with_size

Possible uses:

- `font with_size int --> font`
- `with_size(font, int) --> font`

Result:

Creates a new font from an existing font, with a new size in points

Examples:

```
font var0 <- font ('Helvetica Neue',12, #bold + #italic) with_size 24; // var0 equals a bold and italic face of the Helvetica Neue family with a size of 24 points
```

with_size

Possible uses:

- `with_size(image, int, int) --> image`

Result:

Applies a non-proportional scaling to the image passed in parameter to return a new scaled image with the corresponding width and height. A height of 0 or a width of 0 will return nil. If the width and height parameters are respectively equal to the width and height of the original image, it is returned. Automatic scaling and resizing methods are used. The original image is left untouched

with_style

Possible uses:

- `font with_style int --> font`

- `with_style` (`font` , `int`)---> `font`

Result:

Creates a new font from an existing font, with a new style: either #bold, #italic or #plain or a combination (addition) of them.

Examples:

```
font var0 <- font ('Helvetica Neue',12, #bold + #italic) with_style #plain; // var0 equals a plain face of the Helvetica  
Neue family with a size of 12 points
```

with_values

Possible uses:

- `predicate` `with_values` (`map`)---> `predicate`
- `with_values` (`predicate` , `map`)---> `predicate`

Result:

change the parameters of the given predicate

Examples:

```
predicate with_values [ "time"::10]
```

with_weights

Possible uses:

- `graph` `with_weights` (`list`)---> `graph`
- `with_weights` (`graph` , `list`)---> `graph`
- `graph` `with_weights` (`map`)---> `graph`
- `with_weights` (`graph` , `map`)---> `graph`

Result:

returns the graph (left-hand operand) with weight given in the map (right-hand operand).

Comment:

WARNING / side effect: this operator modifies the operand and does not create a new graph. It also re-initializes the path finder

Special cases:

- if the right-hand operand is a list, assigns 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)
```

with_width

Possible uses:

- `image with_width int --> image`
- `with_width (image , int)--> image`

Result:

Applies a proportional scaling to the image passed in parameter to return a new scaled image with the corresponding width. A width of 0 will return nil, a width equal to the width of the image will return the original image. Automatic scaling and resizing methods are used. The original image is left untouched

without_holes

Possible uses:

- `without_holes (geometry) --> geometry`

Result:

A geometry corresponding to the operand geometry (geometry, agent, point) without its holes

Examples:

```
geometry var0 <- solid(self); // var0 equals the geometry corresponding to the geometry of the agent applying the operator without its holes.  
float var1 <- without_holes(polygon([{0,50}, {0,0}, {50,0}, {50,50}, {0,50}]) - square(10) at_location {10,10}).area; // var1 equals 2500.0
```

wizard

Possible uses:

- `string wizard (list<map<string, unknown>> --> map<string, map<string, unknown>>`
- `wizard (string , list<map<string, unknown>>) --> map<string, map<string, unknown>>`
- `wizard (string, action, list<map<string, unknown>>) --> map<string, map<string, unknown>>`

Result:

Build a wizard and return the values enter by the user as a map of map ["title page 1":["var1":1,"var2":2]]. Takes a string, an action and a list of calls to the `wizard_page()` operator. The first string is used to specify the title. The action to describe when the wizard is supposed to be finished. A classic way of defining the action is `bool eval_finish(map<string,map> input_map) {return input_map["page1"]["file"] != nil;}`. The list is to specify the wizard pages. Build a wizard and return the values enter by the user as a map of map ["title page 1":["var1":1,"var2":2]]. Takes a string, a list of calls to the `wizard_page()` operator. The first string is used to specify the title. The list is to specify the wizard pages.

Examples:

```
map results <- wizard("My wizard",eval_finish, [wizard_page("page1","enter info" ,[enter("var1",string)], font("Arial", 10))]);  
map results <- wizard("My wizard",[wizard_page("page1","enter info" ,[enter("var1",string)], font("Arial", 10))]);
```

wizard_page

Possible uses:

- `wizard_page (string, string, list) --> map<string, unknown>`
- `wizard_page (string, string, list, font) --> map<string, unknown>`

Result:

Build a wizard page. Takes two strings and a list of calls to the `enter()` or `choose()` operators. The first string is used to specify the title, the second the description of the dialog box. The list is to specify the parameters the user can enter. Build a wizard page. Takes two strings, a list of calls to the `enter()` or `choose()` operators and a font as arguments. The first string is used to specify the title, the second the description of the dialog box. The list is to specify the parameters the user can enter. The font is used to specify the font

Examples:

```
map results <- wizard("My wizard", [wizard_page("page1", "enter info", [enter("var1", string)]));  
map results <- wizard("My wizard", [wizard_page("page1", "enter info", [enter("var1", string)], font("Arial", 10))]);
```

writable

Possible uses:

- `file writable bool --> file`
- `writable (file, bool) --> file`

Result:

Marks the file as read-only or not, depending on the second boolean argument, and returns the first argument

Comment:

A file is created using its native flags. This operator can change them. Beware that this change is system-wide (and not only restrained to GAMA): changing a file to read-only mode (e.g. `"writable(f, false)"`)

Examples:

```
file var0 <- shape_file("../images/point_eau.shp") writable false; // var0 equals returns a file in read-only mode
```

See also: [file](#),

xml_file

Possible uses:

- `xml_file (string) --> file`

Result:

Constructs a file of type xml. Allowed extensions are limited to xml

Special cases:

- `xml_file(string)`: This file constructor allows to read a xml file

```
file f <- xml_file("file.xml");
```

See also: [is_xml](#),

xor

Possible uses:

- `bool xor bool --> bool`
- `xor (bool, bool) --> bool`

Result:

a bool value, equal to the logical xor between the left-hand operand and the right-hand operand. False when they are equal

Comment:

both operands are always casted to bool before applying the operator. Thus, an expression like 1 xor 0 is accepted and returns true.

Examples:

```
bool var0 <- xor(true, false); // var0 equals true
bool var1 <- xor(false, false); // var1 equals false
bool var2 <- xor(false, true); // var2 equals true
bool var3 <- xor(true, true); // var3 equals false
bool var4 <- true xor true; // var4 equals false
```

See also: [or](#), [and](#), [!](#),

years_between

Possible uses:

- `date years_between date --> int`
- `years_between (date, date) --> int`

Result:

Provide the exact number of years between two dates. This number can be positive or negative (if the second operand is smaller than the first one)

Examples:

```
int var0 <- years_between(date('2000-01-01'), date('2010-01-01')); // var0 equals 10
```

zip

Possible uses:

- `list<string> zip string --> bool`
- `zip (list<string>, string) --> bool`

Result:

Zip a given list of files or folders. Returns true if the files are well zipped

Examples:

```
bool zip_ok <- zip(["..../includes/my_folder"], "folder.zip";
```



Version: 1.9.1

Exhaustive list of GAMA Keywords

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

Operators

-,:,:,!,!=,?,/,.,.^,@,*,*+,<,<=,=,>,>=,abs,accumulate,acos,action,
add_3Dmodel,add_attribute,add_census_file,add_days,add_edge,add_geometry,
add_hours,add_icon,add_mapper,add_marginals,add_minutes,add_months,
add_ms,add_node,add_point,add_range_attribute,add_seconds,add_values,
add_weeks,add_years,adjacency,after,agent,agent_closest_to,agent_farthest_to,
agent_from_geometry,agent_intersecting,agents_at_distance,agents_covering,
agents_crossing,agents_inside,agents_overlapping,agents_partially_overlapping,
agents_touching,all_indexes_of,all_match,all_pairs_shortest_path,all_verify,
alpha_index,among,any,any_location_in,
any_point_in,append_horizontally,append_vertically,arc,around,as,as_4_grid,
as_distance_graph,as_driving_graph,as_edge_graph,as_grid,as_hexagonal_grid,
as_int,as_intersection_graph,as_json_string,as_map,as_matrix,as_path,
as_spatial_graph,asin,at,at_distance,at_location,atan,atan2,attributes,
auto_correlation,BDIPlan,before,beta,beta_index,between,
betweenness_centrality,biggest_cliques_of,binomial,binomial_coeff,
binomial_complemented,binomial_sum,blend,blend,blurred,bool,box,
brewer_colors,brewer_palettes,brighter,buffer,build,capitalize,cartesian_product,

ceil, cell_at, cells_in, cells_overlapping, centroid, char, chi_square,
chi_square_complemented, choose, circle, clean, clean_network, clipped_with,
closest_points_with, closest_to, collect, column_at, columns_list, command, cone,
cone3D, connected_components_of, connectivity_index, container, contains,
contains_all, contains_any, contains_edge, contains_key, contains_node,
contains_value, contains_vertex, conversation, convex_hull, copy, copy_between,
copy_file, copy_from_clipboard, copy_to_clipboard, copy_to_clipboard, correlation,
cos, cos_rad, count, covariance, covering, covers, create_map, cropped_to, cross,
crosses, crossing, crs, CRS_transform, csv_file, cube, curve, cylinder, darker, date,
dbSCAN, dead, degree_of, delete_file, det, determinant, diff, diff2, directed,
direction_between, direction_to, directory, disjoint_from, distance_between,
distance_to, distinct, distribution_of, distribution2d_of, div, dnorm, dtw,
durbin_watson, dxf_file, edge, edge_between, edge_betweenness, edges,
eigenvalues, electre_DM, ellipse, elliptical_arc, emotion, empty, enlarged_by, enter,
envelope, eval_gaml, eval_when, evaluate_sub_model, even, every, every_cycle,
evidence_theory_DM, exp, fact, farthest_point_to, farthest_to, field, field_with, file,
file_exists, first, first_of, first_with, flip, float, floor, folder, folder_exists, font,
frequency_of, from, fuzzy_choquet_DM, fuzzy_kappa, fuzzy_kappa_sim, gaml_file,
gaml_type, gamma, gamma_density, gamma_distribution,
gamma_distribution_complemented, gamma_index, gamma_rnd, gamma_trunc_rnd,
gauss, gauss_rnd, gen_population_generator, gen_range, generate_barabasi_albert,
generate_complete_graph, generate_pedestrian_network, generate_random_graph,
generate_terrain, generate_watts_strogatz, geojson_file, geometric_mean, geometry,
geometry_collection, get, get_about, get_agent, get_agent_cause, get_belief_op,
get_belief_with_name_op, get_beliefs_op, get_beliefs_with_name_op,
get_current_intention_op, get_decay, get_desire_op, get_desire_with_name_op,
get_desires_op, get_desires_with_name_op, get_dominance, get_familiarity,
get_ideal_op, get_ideal_with_name_op, get_ideals_op, get_ideals_with_name_op,
get_intensity, get_intention_op, get_intention_with_name_op, get_intentions_op,
get_intentions_with_name_op, get_lifetime, get_liking, get_modality,
get_obligation_op, get_obligation_with_name_op, get Obligations_op,

get_obligations_with_name_op, get_plan_name, get_predicate, get_solidarity,
get_strength, get_super_intention, get_trust, get_truth, get_uncertainties_op,
get_uncertainties_with_name_op, get_uncertainty_op,
get_uncertainty_with_name_op, get_values, gif_file, gini, girvan_newman_clustering,
gml_file, gradient, graph, graph6_file, graphdimacs_file, graphdot_file,
graphgexf_file, graphgml_file, graphml_file, graphtsplib_file, grayscale, grayscale,
grid_at, grid_cells_to_graph, grid_file, group_by, harmonic_mean, has_belief_op,
has_belief_with_name_op, has_desire_op, has_desire_with_name_op, has_ideal_op,
has_ideal_with_name_op, has_intention_op, has_intention_with_name_op,
has Obligation_op, has Obligation_with_name_op, has Uncertainty_op,
has Uncertainty_with_name_op, hexagon, hierarchical_clustering, horizontal,
horizontal_flip, hsb, hypot, IDW, image, image_file, in, in_degree_of, in_edges_of,
incomplete_beta, incomplete_gamma, incomplete_gamma_complement,
indented_by, index_by, index_of, inside, int, inter, interleave,
internal_integrated_value, intersecting, intersection, intersects, inverse,
inverse_distance_weighting, inverse_rotation, is, is_csv, is_dxf, is_error, is_finite,
is_gaml, is_geojson, is_gif, is_gml, is_graph6, is_graphdimacs, is_graphdot,
is_graphgexf, is_graphgml, is_graphml, is_graphtsplib, is_grid, is_image, is_json,
is_number, is_obj, is_osm, is_pgm, is_property, is_reachable, is_saved_simulation,
is_shape, is_skill, is_svg, is_text, is_threeds, is_warning, is_xml, json_file,
k_nearest_neighbors, k_spanning_tree_clustering, kappa, kappa_sim, kmeans, kml,
kurtosis, label_propagation_clustering, last, last_index_of, last_of, last_with,
layout_circle, layout_force, layout_force_FR, layout_force_FR_indexed, layout_grid,
length, lgamma, line, link, list, list_with, ln, load_shortest_paths, load_sub_model, log,
log_gamma, lognormal_density, lognormal_rnd, lognormal_trunc_rnd, lower_case,
main_connected_component, map, masked_by, matrix, matrix, matrix_with, max,
max_flow_between, max_of, maximal cliques_of, mean, mean_deviation, mean_of,
median, mental_state, message, milliseconds_between, min, min_of, minus_days,
minus_hours, minus_minutes, minus_months, minus_ms, minus_seconds,
minus_weeks, minus_years, mod, moment, months_between, moran, morrisAnalysis,
mul, nb_cycles, neighbors_at, neighbors_of, new_emotion, new_folder,

new_mental_state, new_predicate, new_social_link, node, nodes, none_matches,
none_verifies, norm, Norm, normal_area, normal_density, normal_inverse,
normalized_rotation, not, not, obj_file, of, of_generic_species, of_species,
one_matches, one_of, one_verifies, or, or, osm_file, out_degree_of, out_edges_of,
overlapping, overlaps, pair, palette, partially_overlapping, partially_overlaps, path,
path_between, path_to, paths_between, pbinom, pchisq, percent_absolute_deviation,
percentile, pgamma, pgm_file, plan, play_sound, plus_days, plus_hours,
plus_minutes, plus_months, plus_ms, plus_seconds, plus_weeks, plus_years, pnorm,
point, points_along, points_at, points_in, points_on, poisson, polygon, polyhedron,
polyline, polyplan, predecessors_of, predicate, predict, product, product_of,
promethee_DM, property_file, pValue_for_fStat, pValue_for_tStat, pyramid, quantile,
quantile_inverse, range, rank_interpolated, read, rectangle, reduced_by,
regex_matches, regression, remove_duplicates, remove_node_from, rename_file,
replace, replace_regex, residuals, restore_simulation, restore_simulation_from_file,
reverse, rewire_n, rgb, rgb, rms, rnd, rnd_choice, rnd_color, rotated_by, rotated_by,
rotation_composition, round, row_at, rows_list, rSquare, sample, Sanction,
save_simulation, saved_simulation_file, scale, scaled_by, scaled_to, select, serialize,
serialize_agent, set_about, set_agent, set_agent_cause, set_decay, set_dominance,
set_familiarity, set_intensity, set_lifetime, set_liking, set_modality, set_predicate,
set_solidarity, set_strength, set_trust, set_truth, set_z, shape_file, sharpened, shuffle,
signum, simple_clustering_by_distance, simple_clustering_by_envelope_distance,
simplification, sin, sin_rad, since, skeletonize, skew, skew_gauss, skewness, skill,
smooth, snapshot, sobolAnalysis, social_link, solid, sort, sort_by, source_of,
spatial_graph, species, species_of, sphere, split, split_at, split_geometry, split_in,
split_lines, split_using, split_with, sqrt, square, squircle, stack, standard_deviation,
step_sub_model, strahler, string, student_area, student_t_inverse, subtract_days,
subtract_hours, subtract_minutes, subtract_months, subtract_ms, subtract_seconds,
subtract_weeks, subtract_years, successors_of, sum, sum_of, svg_file, t_test, tan,
tan_rad, tanh, target_of, teapot, text_file, TGauss, threeds_file, tinted_with, to,
to_GAMA_CRS, to_gaml, to_geojson, to_hsb, to_list, to_rectangles, to_segments,
to_squares, to_sub_geometries, to_triangles, tokenize, topology, touches, touching,

towards, trace, transformed_by, translated_by, translated_to, transpose, triangle, triangulate, truncated_gauss, type_of, undirected, union, unknown, until, unzip, upper_case, use_cache, user_confirm, user_input_dialog, using, values_in, variance, variance_of, vertical, vertical_flip, voronoi, weibull_density, weibull_rnd, weibull_trunc_rnd, weight_of, weighted_means_DM, where, with_generation_algo, with_height, with_k_shortest_path_algorithm, with_max_of, with_min_of, with_precision, with_shortest_path_algorithm, with_size, with_size, with_style, with_values, with_weights, with_width, without_holes, wizard, wizard_page, writable, xml_file, xor, years_between, zip,

Statements

=, abort, action, add, agents, annealing, ask, aspect, assert, benchmark, betad, break, browse, camera, capture, catch, category, chart, conscious_contagion, continue, coping, create, data, datalist, default, diffuse, diffusion, display, display_grid, do, draw, else, emotional_contagion, enforcement, enter, equation, error, event, exit, experiment, exploration, focus, focus_on, generate, genetic, global, graphics, grid, highlight, hill_climbing, if, image_layer, init, inspect, invoke, law, layout, let, light, loop, match, match_between, match_one, match_regex, mesh, migrate, monitor, morris, norm, output, output_file, overlay, parameter, perceive, permanent, plan, pso, put, reactive_tabu, reflex, release, remove, return, rotation, rule, rule, run, sanction, save, set, setup, sobol, socialize, solve, species, species_layer, start_simulation, state, status, stochanalyse, switch, tabu, task, test, text, trace, transition, try, unconscious_contagion, user_command, user_init, user_input, user_panel, using, Variable_container, Variable_number, Variable_regular, warn, write,

Architectures

fsm, parallel_bdi, probabilistic_tasks, reflex, rules, simple_bdi, sorted_tasks,
user_first, user_last, user_only, weighted_tasks,

Constants and colors

#μm (#micrometer,#micrometers), #AdamsBashforth, #AdamsMoulton, #ambient,
#AStar, #BellmannFord, #Bhandari, #BidirectionalDijkstra, #bold, #bottom_center,
#bottom_left, #bottom_right, #camera_location, #camera_orientation,
#camera_target, #center, #CHBidirectionalDijkstra, #cl (#centiliter,#centiliters), #cm
(#centimeter,#centimeters), #current_error, #custom, #cycle (#cycles), #day
(#d,#days), #DeltaStepping, #Dijkstra, #direction, #display_height, #display_width,
#dl (#deciliter,#deciliters), #dm (#decimeter,#decimeters), #DormandPrince54,
#dp853, #e, #epoch, #Eppstein, #Euler, #flat, #FloydWarshall, #foot (#feet,#ft),
#from_above, #from_front, #from_left, #from_right, #from_up_front, #from_up_left,
#from_up_right, #fullscreen, #Gill, #GraggBulirschStoer, #gram (#grams), #h
(#hour,#hours), #hidpi, #HighamHall54, #hl (#hectoliter,#hectoliters), #horizontal,
#inch (#inches), #infinity, #iso_local, #iso_offset, #iso_zoned, #isometric, #italic, #kg
(#kilo,#kilogram,#kilos), #km (#kilometer,#kilometers), #l (#liter,#liters,#dm3),
#left_center, #longton (#lton), #Luther, #m (#meter,#meters), #m2, #m3,
#max_float, #max_int, #Midpoint, #mile (#miles), #min_float, #min_int, #minute
(#minutes,#mn), #mm (#milimeter,#milimeters), #month (#months), #msec
(#millisecond,#milliseconds,#ms), #nan, #NBAStar, #NBAStarApprox, #nm
(#nanometer,#nanometers), #none, #now, #ounce (#oz,#ounces), #pi, #pixels (#px),
#plain, #point, #pound (#lb,#pounds,#lbf), #right_center, #rk4, #round, #sec
(#second,#seconds,#s), #shortton (#ston), #split, #spot, #sqft
(#square_foot,#square_feet), #sqin (#square_inch,#square_inches), #sqmi

(#square_mile,#square_miles), #square, #stack, #stone (#st), #Suurballe,
#ThreeEighths, #to_deg, #to_rad, #ton (#tons), #top_center, #top_left, #top_right,
#TransitNodeRouting, #user_location, #vertical, #week (#weeks), #yard (#yards),
#year (#years,#y), #Yen, #zoom,

Skills

advanced_driving, driving, dynamic_body, fipa, messaging, moving, moving3D,
network, pedestrian, pedestrian_road, skill_road, skill_road_node, SQLSKILL,
static_body, thread,

Species

agent, AgentDB, base_edge, experiment, graph_edge, graph_node, physical_world,
world

Actions

*init, step, isConnected, close, timeStamp, connect, testConnection, select,
executeUpdate, getParameter, setParameter, insert, update_outputs,
compact_memory, related_to, register, advanced_follow_driving, ready_to_cross,
test_next_road, compute_path, path_from_nodes, drive_random, drive,
on_entering_new_road, external_factor_impact, unregister, speed_choice,
lane_choice, choose_lane, force_move, goto_drive, follow_driving, goto_driving,
apply, start_conversation, send, reply, accept_proposal, agree, cancel, cfp,
end_conversation, failure, inform, propose, query, refuse, reject_proposal, request,*

subscribe, send, wander, move, follow, goto, move, execute, connect, fetch_message, has_more_message, join_group, leave_group, fetch_message_from_network, walk_to, compute_virtual_path, release_path, walk, initialize, build_intersection_areas, build_exit_hub, register, unregister, testConnection, executeUpdate, insert, select, list2Matrix, update_body, contact_added_with, contact_removed_with, run_thread, end_thread, thread_action,

Variables

speed, real_speed, acceleration, current_path, final_target, current_target, current_index, targets, security_distance_coeff, safety_distance_coeff, min_security_distance, min_safety_distance, current_lane, lowest_lane, num_lanes_occupied, vehicle_length, speed_coeff, max_speed, time_headway, max_acceleration, max_deceleration, delta_idm, politeness_factor, max_safe_deceleration, acc_gain_threshold, acc_bias, lane_change_coldown, time_since_lane_change, ignore_oneway, violating_oneway, current_road, next_road, on_linked_road, using_linked_road, allowed_lanes, linked_lane_limit, lane_change_limit, proba_lane_change_up, proba_lane_change_down, proba_use_linked_road, proba_respect_priorities, proba_respect_stops, proba_block_node, right_side_driving, distance_to_goal, distance_to_current_target, segment_index_on_road, leading_vehicle, leading_distance, leading_speed, follower, living_space, lanes_attribute, tolerance, obstacle_species, speed, damping, angular_damping, contact_damping, angular_velocity, velocity, conversations, accept_proposals, agrees, cancels, cfps, failures, informs, proposes, queries, refuses, reject_proposals, requests, requestWhens, subscribes, mailbox, location, speed, heading, current_path, current_edge, real_speed, destination, speed, heading, pitch, roll, destination, network_name, network_groups, network_server, shoulder_length, minimal_distance, pedestrian_consideration_distance, obstacle_consideration_distance, avoid_other, obstacle_species, pedestrian_species, proba_detour, A_pedestrians_SFM, A_obstacles_SFM, B_pedestrians_SFM,

B_obstacles_SFM, k_SFM, kappa_SFM, relaxion_SFM, gama_SFM, lambda_SFM, n_SFM,
n_prime_SFM, pedestrian_model, velocity, forces, final_waypoint, current_waypoint,
current_index, waypoints, roads_waypoints, use_geometry_waypoint,
tolerance_waypoint, agents_on, free_space, road_status, intersection_areas,
linked_pedestrian_roads, exit_nodes, agents_on, all_agents, source_node,
target_node, num_lanes, num_segments, linked_road, maxspeed, segment_lengths,
vehicle_ordering, roads_in, priority_roads, roads_out, stop, block, mass, rotation,
friction, restitution, aabb,

Pseudo-Variables

self, myself, each

Types

action, agent, attributes, BDIPPlan, bool, container, conversation, date, directory,
emotion, field, file, float, font, gaml_type, gen_population_generator, gen_range,
geometry, graph, image, int, kml, list, map, matrix, mental_state, message, Norm,
pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, species, string,
topology, unknown,

the world

torus, Environment Size, world, time cycle, step, time, duration, total_duration
average_duration, machine_time, agents, stop, halt, pause, scheduling

Grid

[grid_x](#), [grid_y](#), [agents](#), [color](#), [grid_value](#)

Other concepts

[scheduling](#), [step](#), [Key concepts](#), [Object-oriented paradigm to GAML](#), [Correspondence GAML and Netlogo](#)



>

Developing GAMA

Version: 1.9.1

Get into the GAMA Java API

GAMA is written in Java and made of tens of Eclipse plugins and projects, thousand of classes, methods and annotations. This section of the wiki should help you have a general idea on how to manipulate GAMA Java API and where to find the proper classes and methods. A general introduction to the [GAMA architecture](#) gives a general overview of the organization of Java packages and Eclipse plugins, and should be read first. In the following sub-sections we give a more practical introduction.

1. [Introduction to GAMA Java API](#)
 - i. [Installing the GIT version](#)
 - ii. [Architecture of GAMA](#)
 - iii. [IScope](#)
2. [Developing Extensions](#)
 - i. [Developing Plugins](#)
 - ii. [Developing Skills](#)
 - iii. [Developing Statements](#)
 - iv. [Developing Operators](#)
 - v. [Developing Types](#)
 - vi. [Developing Species](#)
 - vii. [Developing Control Architectures](#)
 - viii. [Index of annotations](#)
3. [Create a release of Gama](#)

4. Generation of the documentation

Version: 1.9.1

Installing the GIT version

Important note: the current Git version contains 1 main branch:

- **GAMA_1.9.2**: that contains the code of the GAMA alpha(GAMA 1.9.2) (it works with **JDK 17 LTS** and **Eclipse 2022-12**).

Changes made to other branches won't be added to the next gama release

The following tutorial describes the installation for this version.

Install Eclipse 2022-12

Download the "[Installer of 2022-12](#)" and choose to install the **Eclipse IDE for Java and DSL Developers** version. This is the latest version under which GAMA is certified to work.

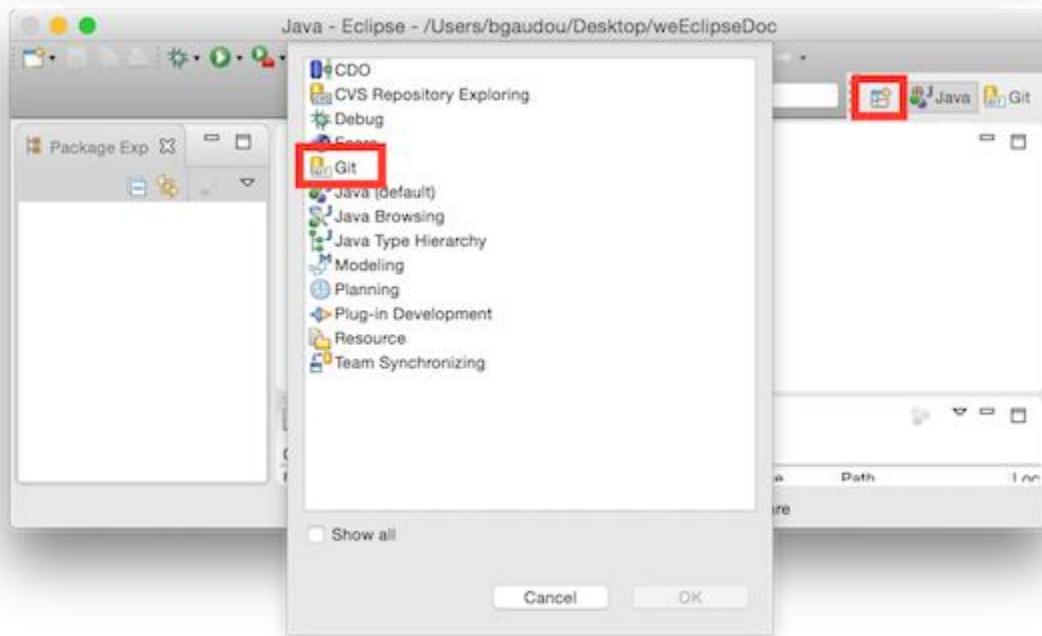
Note: Regarding Java, Eclipse embeds the [Adoptium \(ex Adopt-OpenJDK\) 17 LTS \(HotSpot\)](#), which is the recommended version for GAMA, you may be able to use another one, but we won't fix any related issue.

Install GAMA source code

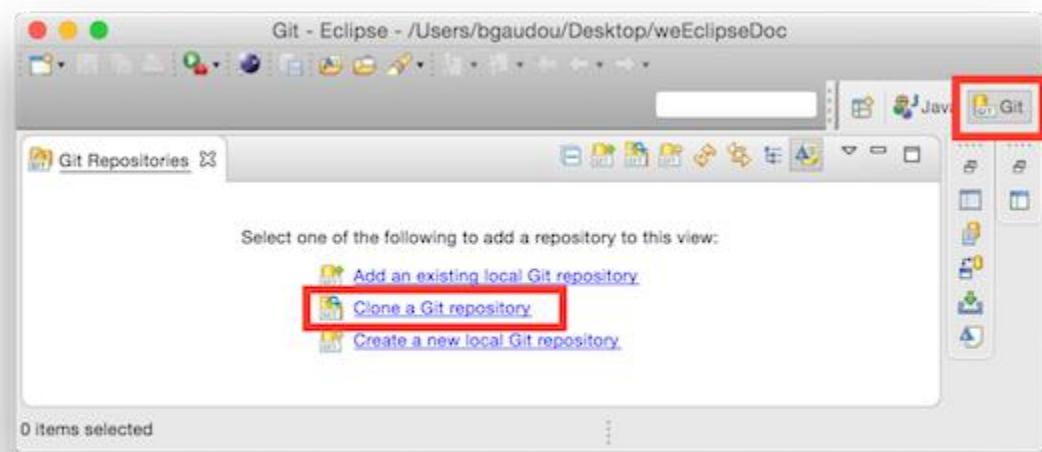
The source is to be downloaded from GitHub in two steps: by creating a local clone of the GitHub repository and then importing the different projects that constitute GAMA into the Eclipse workspace.

1. Open the Git perspective:

- Windows > Perspective > Open Perspective > Other...
- Choose **Git**

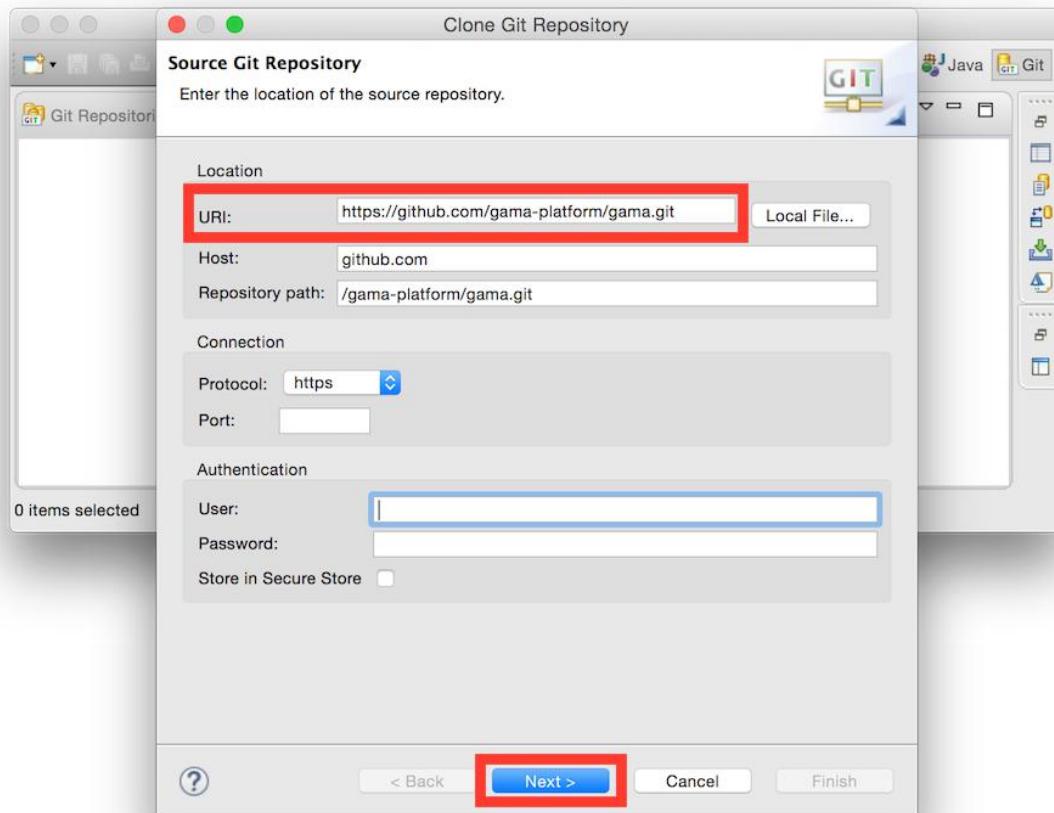


2. Click on "Clone a Git repository"



- In **Source Git repository** window:

- * Fill in the URI label with:
`<https://github.com/gama-platform/gama.git>`
- * Other fields will be automatically filled in.



- In **Branch Selection** windows,
 - check the **GAMA_1.9.2** branch
 - Next

Import Projects from Git



Branch Selection

Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository.

Branches of <https://github.com/gama-platform/gama.git>:

type filter text

-  GAMA_1.8.1
-  GAMA_1.8.2
-  GAMA_1.8.2_pedestrian
-  GAMA_2.0
-  pdfGeneration
-  public-transport-skill

Tag fetching strategy

- When fetching a commit, also fetch its tags
- Fetch all tags and their commits
- Don't fetch any tags

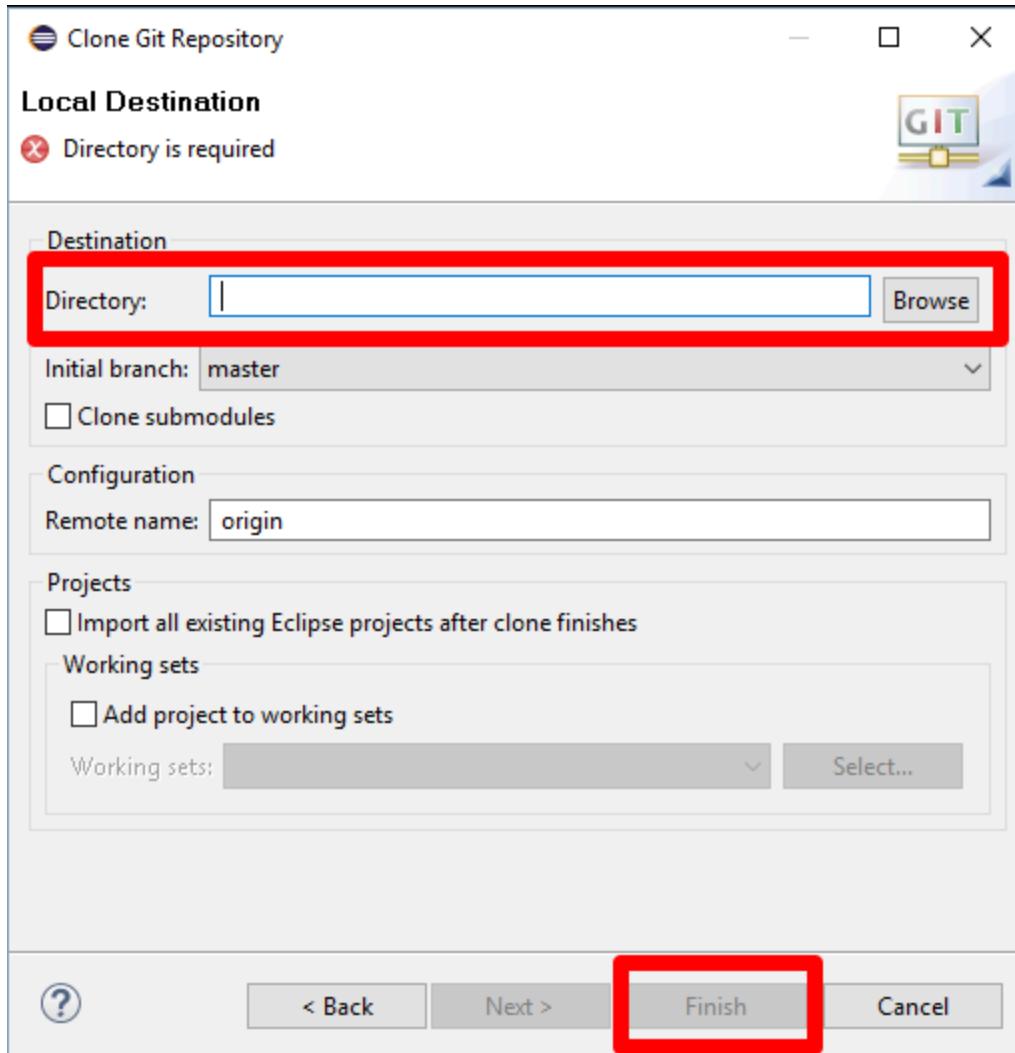


< Back

Next >

- In **Local Destination** windows,

- Choose a Directory (where the source files will be downloaded).
 - Everything else should be unchecked
 - Finish



This can take a while...

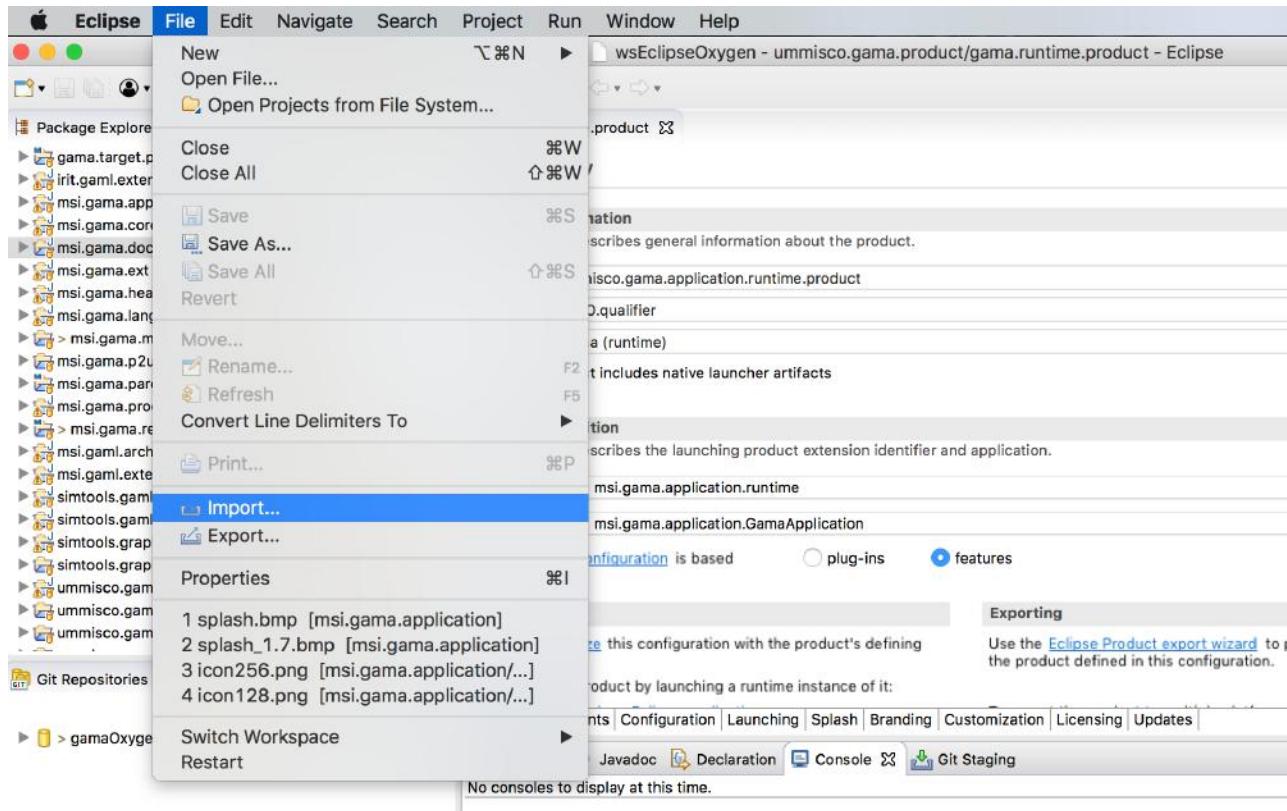
Import projects into workspace

You have now to import projects into the workspace (notice that the folders downloaded during the clone will neither be copied nor moved).

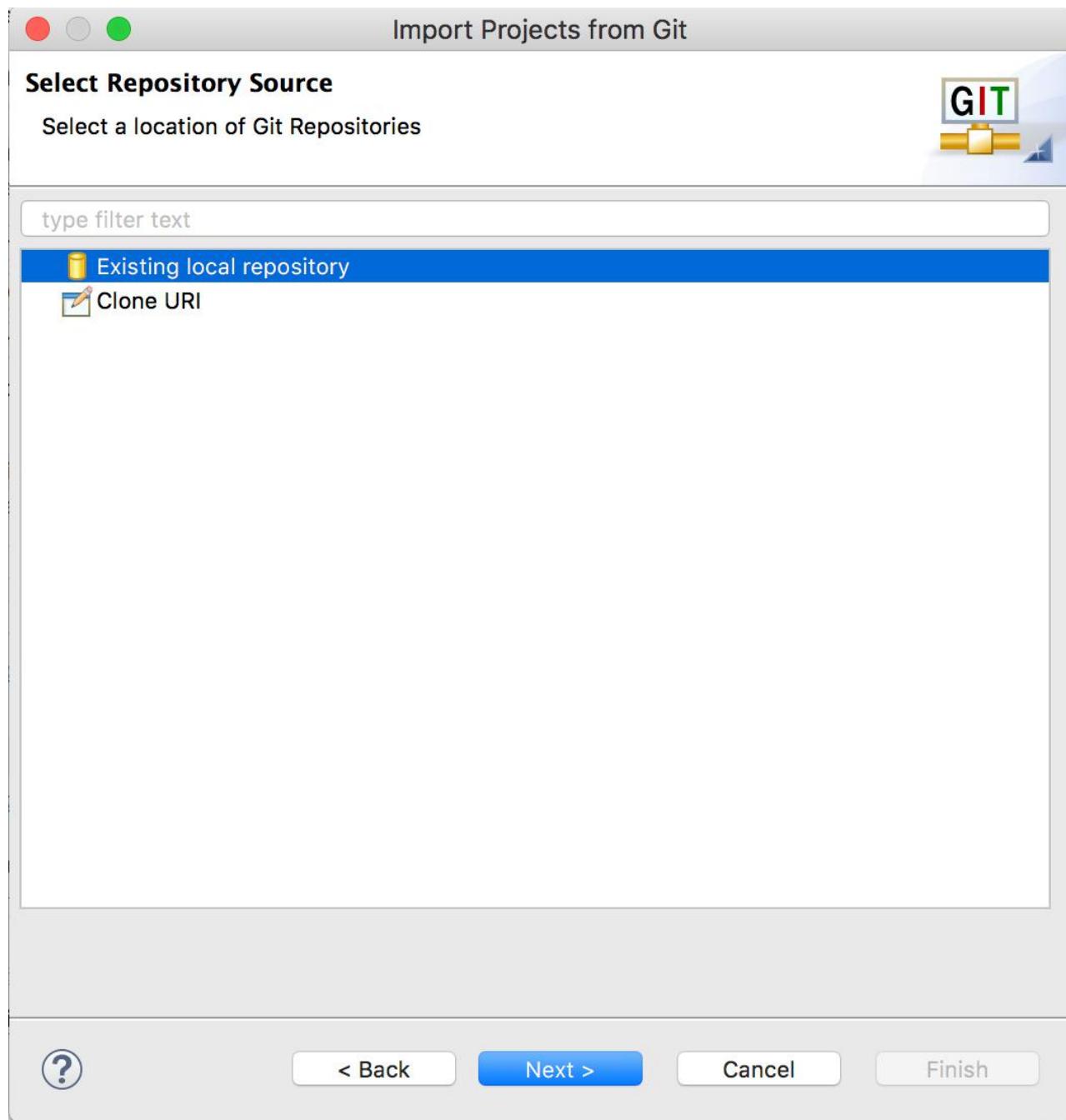
Note: contrarily to previous Eclipse versions, import project from the Git perspective does not work properly for GAMA.

1. In the **Java perspective**, choose:

- **File / Import...**,

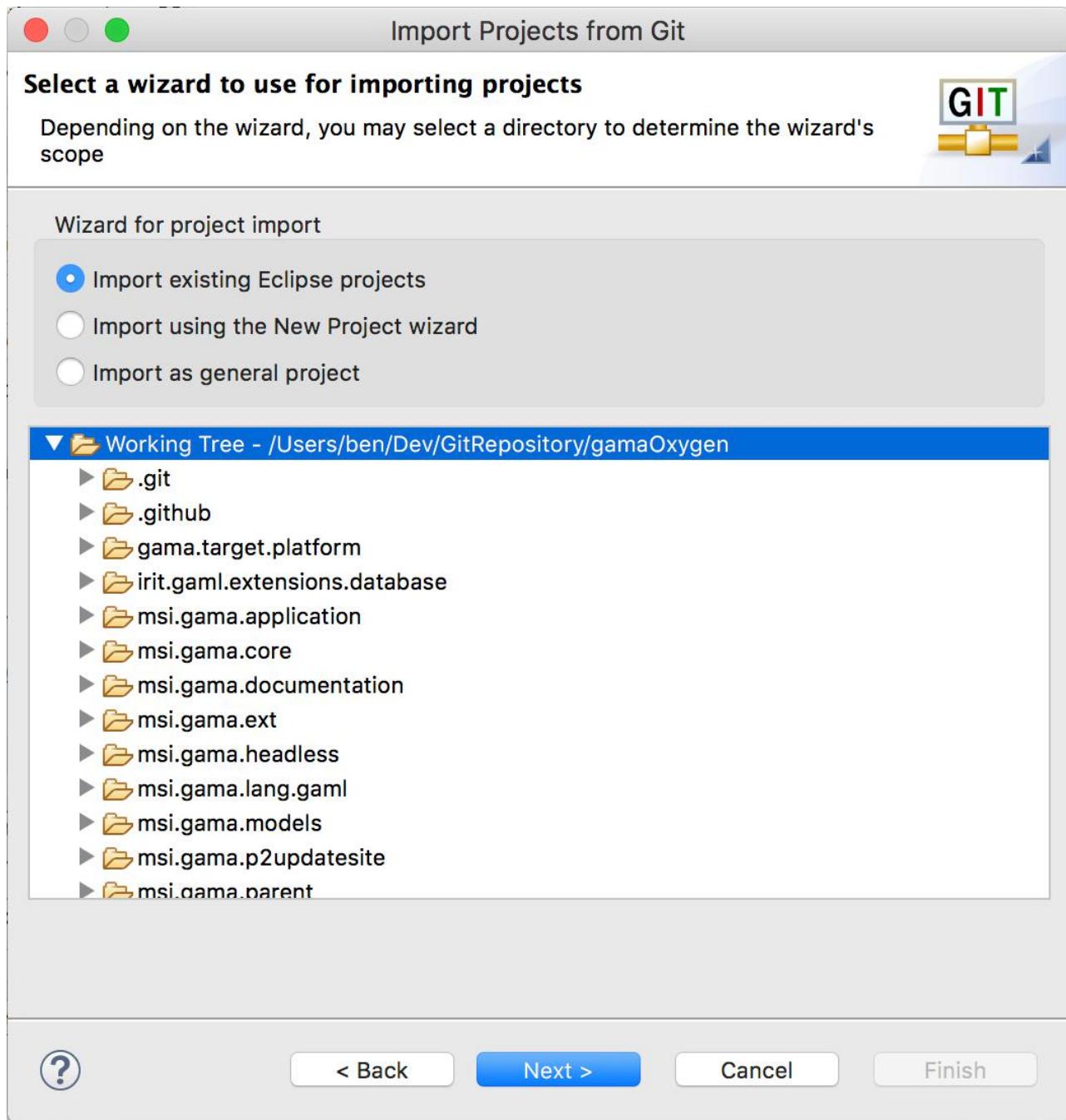


- In the install window, select **Git / Projects from Git**,
- Click on Next,
- In the **Project from Git** window, select **Existing local repository**.



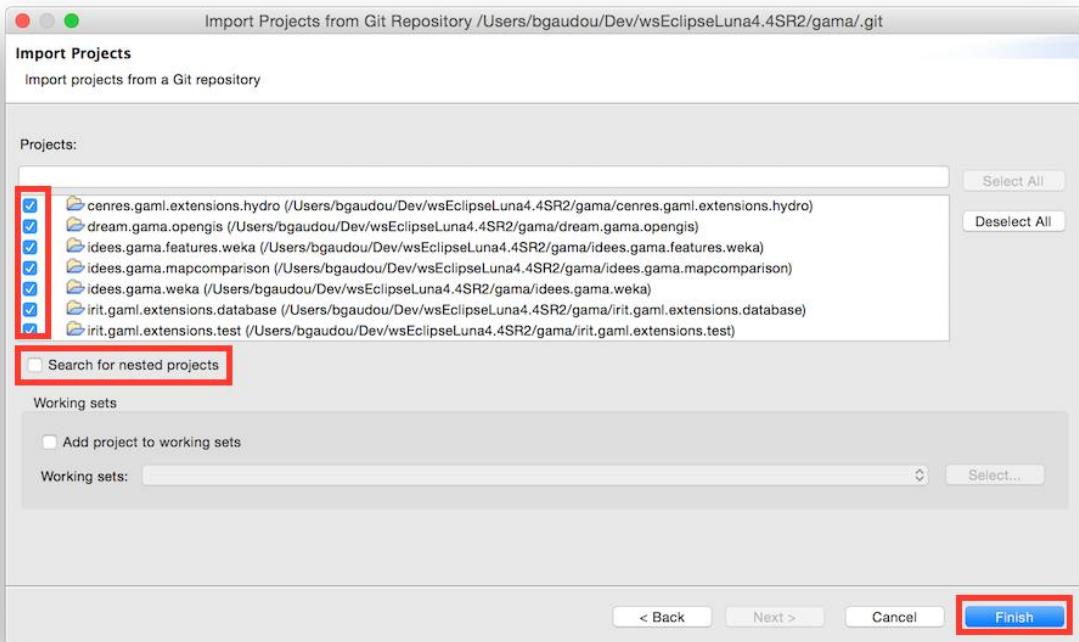
- Click on Next,
- In the new window, select your Git repository,
- Click on Next,
- In the **Select a wizard to used to import projects**, check that
 - Import existing Eclipse projects is selected

- Working Tree is selected



- Click on Next,
- In the **Import project** window,

- * **Uncheck Search for nested projects**
- * Select all the projects



- Finish
3. Clean project (Project menu > Clean ...)

If you have errors...

If errors continue to show on in the different projects, be sure to correctly set the JDK used in the Eclipse preferences. GAMA (version 1.9.2) is targeting JDK 17, and Eclipse could produce errors if it did not find in your environment. So, either you set the compatibility to 17 by default (in Preferences > Java > Compiler > Compiler Compliance Level) or you change the error produced by Eclipse to a warning only (in Preferences > Java > Compiler > Building > "No strictly compatible JRE for execution

environment available).

On Windows : if the project still don't compile, try to add the vm argument in eclipse.ini files (inside the directory where your eclipse is installed) before the -startup line Example :

```
-vm  
C:\Program Files\Java\JDK17\bin  
-startup  
.....
```

Run GAMA

0. Be sure to be in the Java Perspective (top right button)
1. In the `ummisco.gama.product` plugin, open the `gama.product` file (`gama.headless.product` is used to produce the headless).
2. Go to "Overview" tab and click on Synchronize
3. Click on "Launch an Eclipse Application"

GIT Tutorials

For those who want to learn more about Git and Egit, please consult the following tutorials/papers

1. EGIT/User Guide http://wiki.eclipse.org/EGit/User_Guide
2. Git version control with Eclipse (EGIT) - Tutorial <http://www.vogella.com/tutorials/EclipseGit/article.html>
3. 10 things I hate about Git <http://stevebennett.me/2012/02/24/10-things-i-hate-about-git/>
4. Learn Git and GitHub Tutorial <https://www.youtube.com/playlist?list=PL1F56EA413018EEE1>

Version: 1.9.1

Developing Extensions

GAMA accepts *extensions* to the GAML language, defined by external programmers and dynamically loaded by the platform each time it is run. Extensions can represent new built-in species, types, file-types, skills, operators, statements, new control architectures or even types of displays. Other internal structures of GAML will be progressively "opened" to this mechanism in the future: display layers (hardwired for the moment), new types of outputs (hardwired for the moment), scheduling policies (hardwired for the moment), random number generators (hardwired for the moment). The extension mechanism relies on two complementary techniques:

- the first one consists in defining the GAML extensions [in a plug-in](#) (in the OSGI sense, see [here](#)) that will be loaded by GAMA at runtime and must "declare" that it is contributing to the platform.
- the second one is to indicate to GAMA where to look for extensions, using Java annotations that are gathered at compile time (some being also used at runtime) and directly compiled into GAML structures.

The following sections describe this extension process.

- i. [Developing Plugins](#)
- ii. [Developing Skills](#)
- iii. [Developing Statements](#)
- iv. [Developing Operators](#)
- v. [Developing Types](#)
- vi. [Developing Species](#)
- vii. [Developing Control Architectures](#)
- viii. [IScope](#)

- ix. Index of annotations



Version: 1.9.1

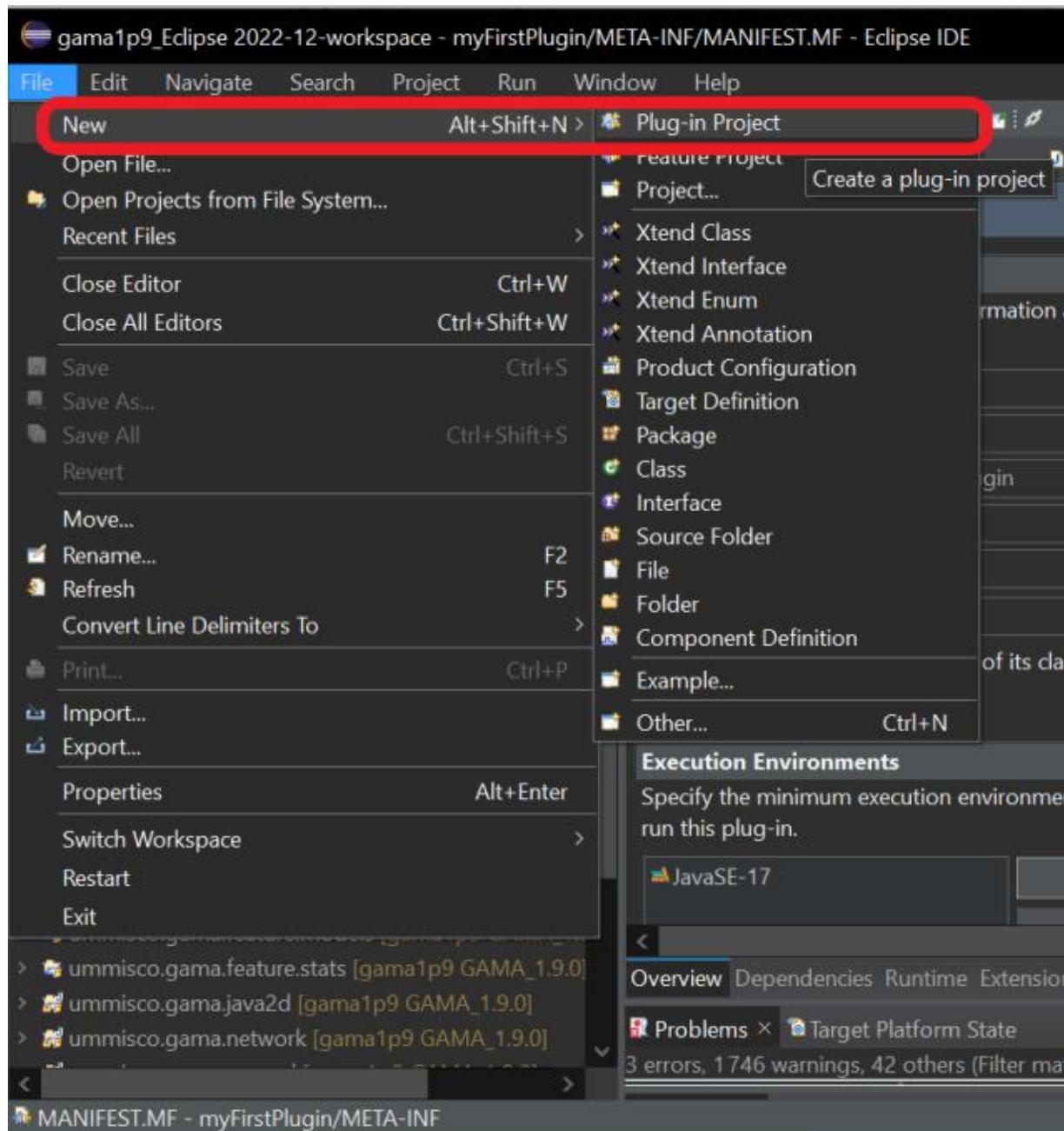
Developing Plugins

This page details how to create a new plug-in in order to extend the GAML language with new skills, species, displays or operators. It also details how to create a plug-in that can be uploaded on an update site and can be installed into the GAMA release. We consider here that the developer version of GAMA has been installed (as detailed in [this page](#)).

I. Creation of a plug-in

Here are detailed steps to create and configure a new GAMA plug-in.

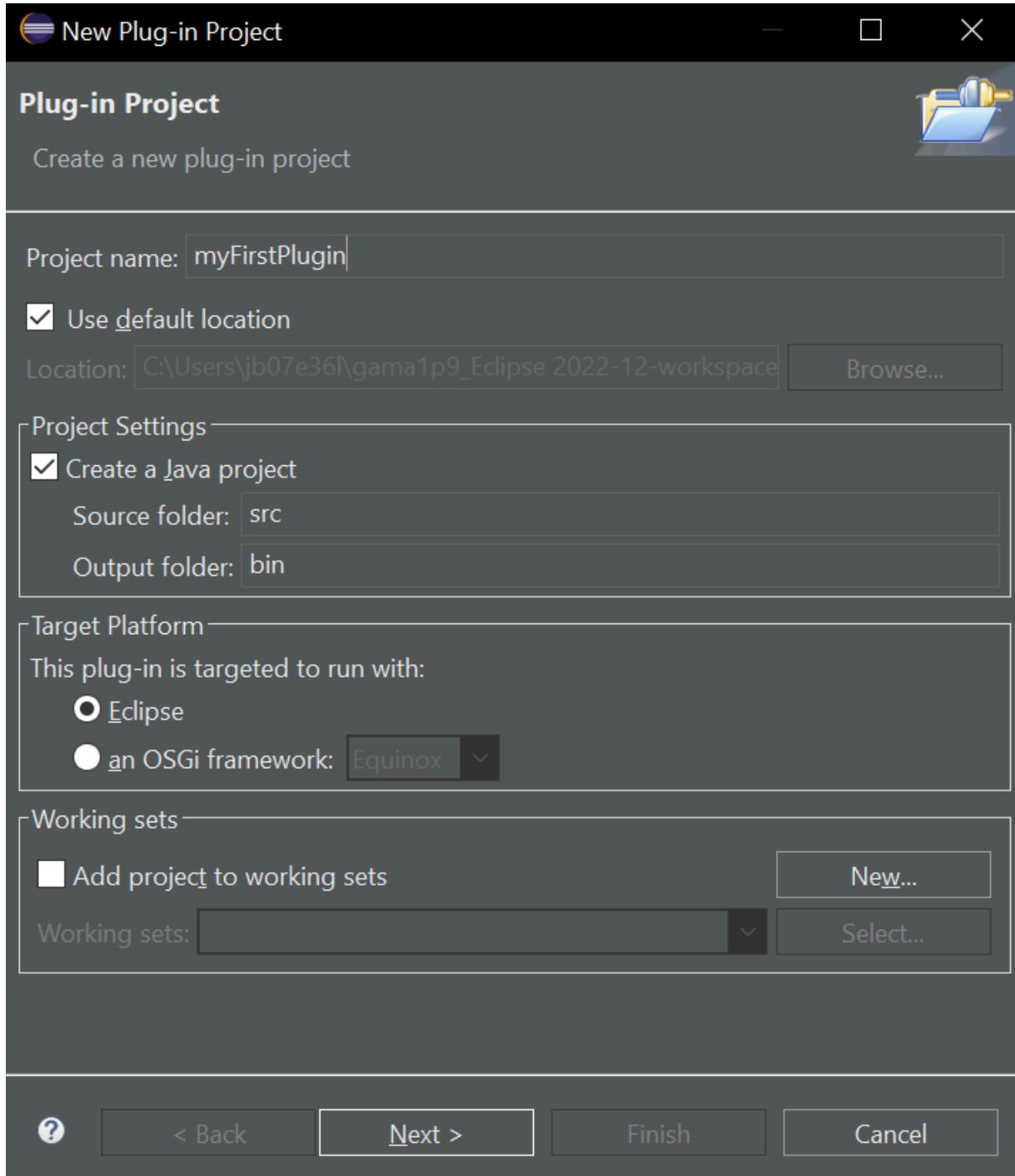
1. From the Eclipse main menu tab, click on File, then New, then Project, then finally select plug-in project.



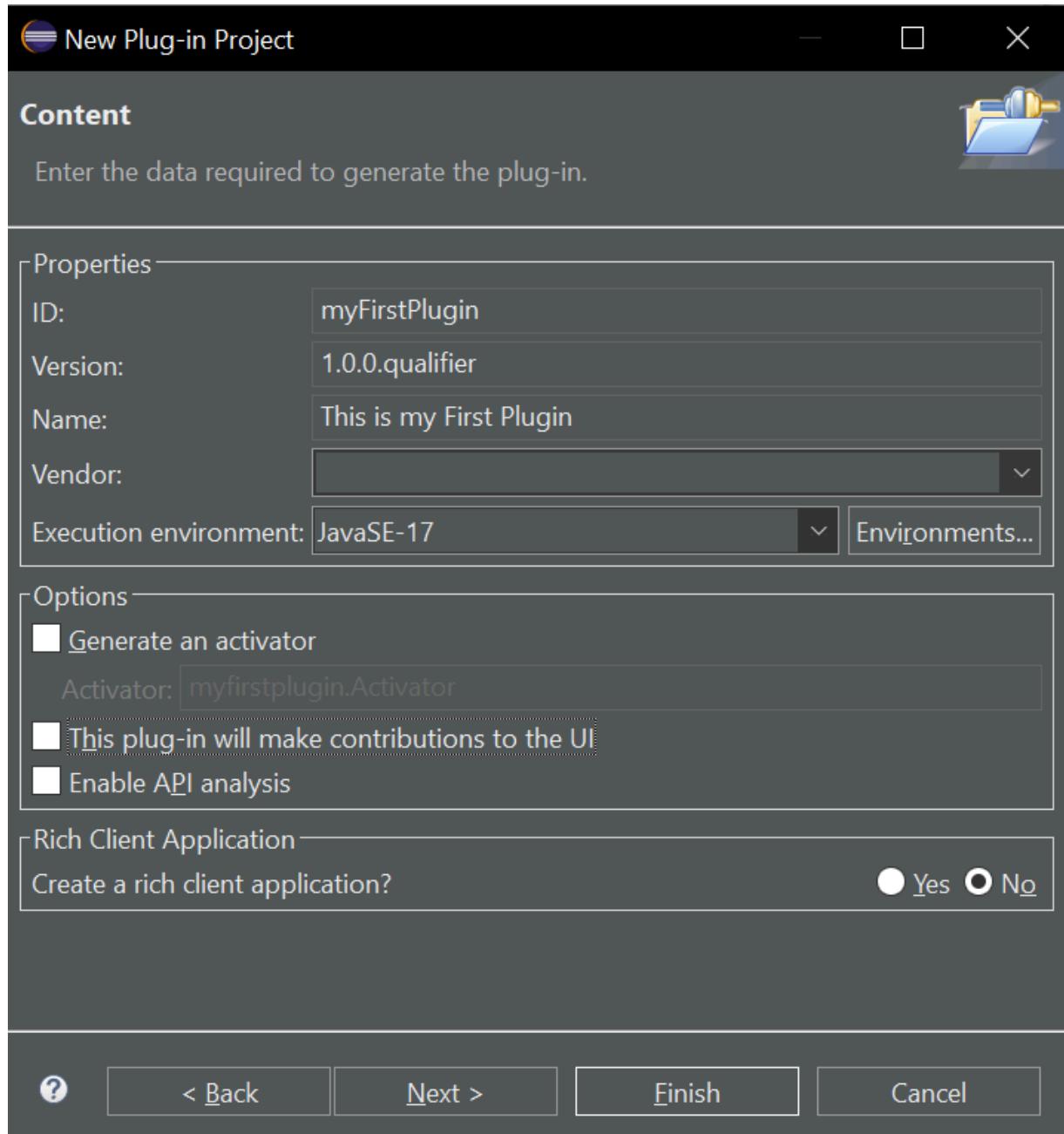
2. In the "New plug-in Project" / "Plug-in project" window:

- i. Choose as **name** « name_of_the_plugin » (or anything else)*
- ii. Check "Use default location"
- iii. Check "Create a Java Project"

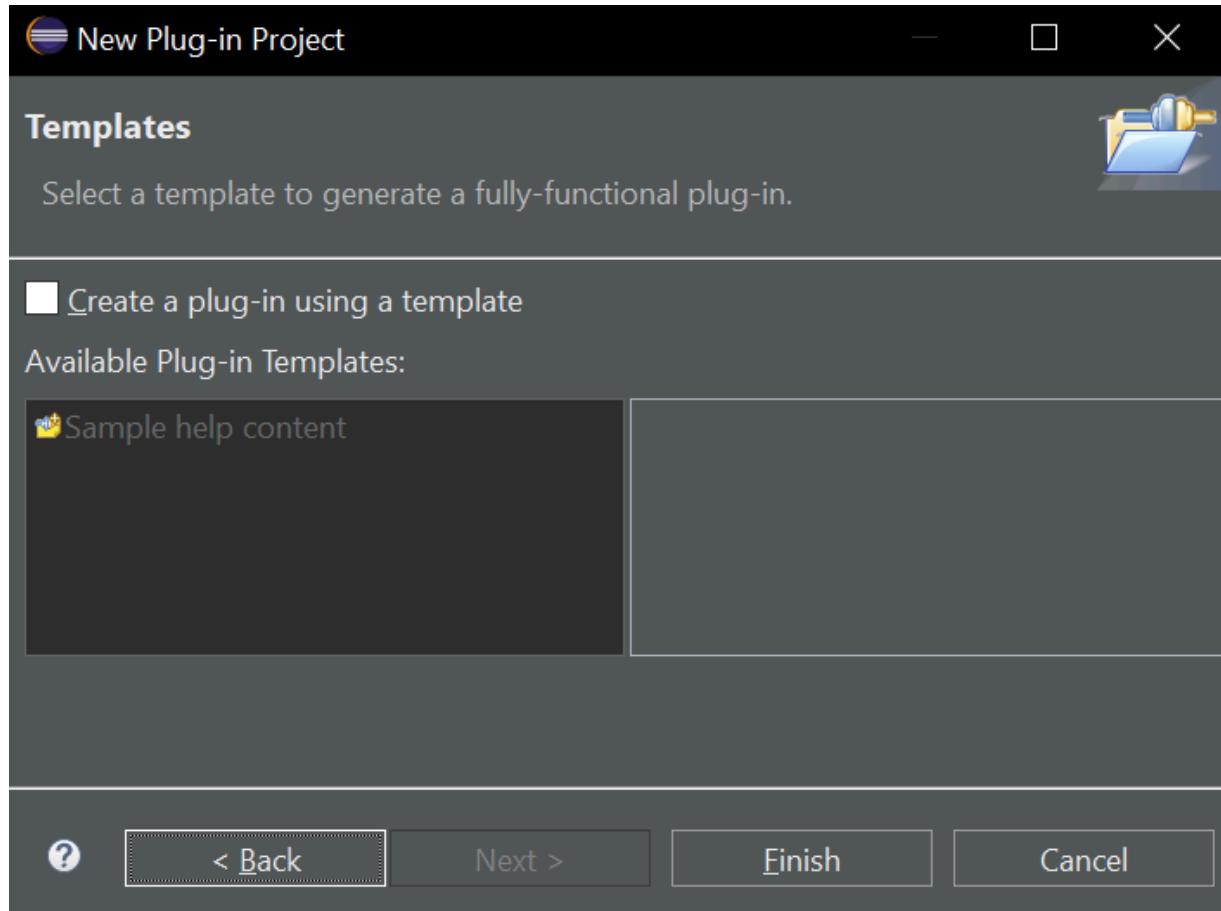
- iv. The project should be targeted to run with Eclipse
- v. working set is unchecked
- vi. Click on "Next"



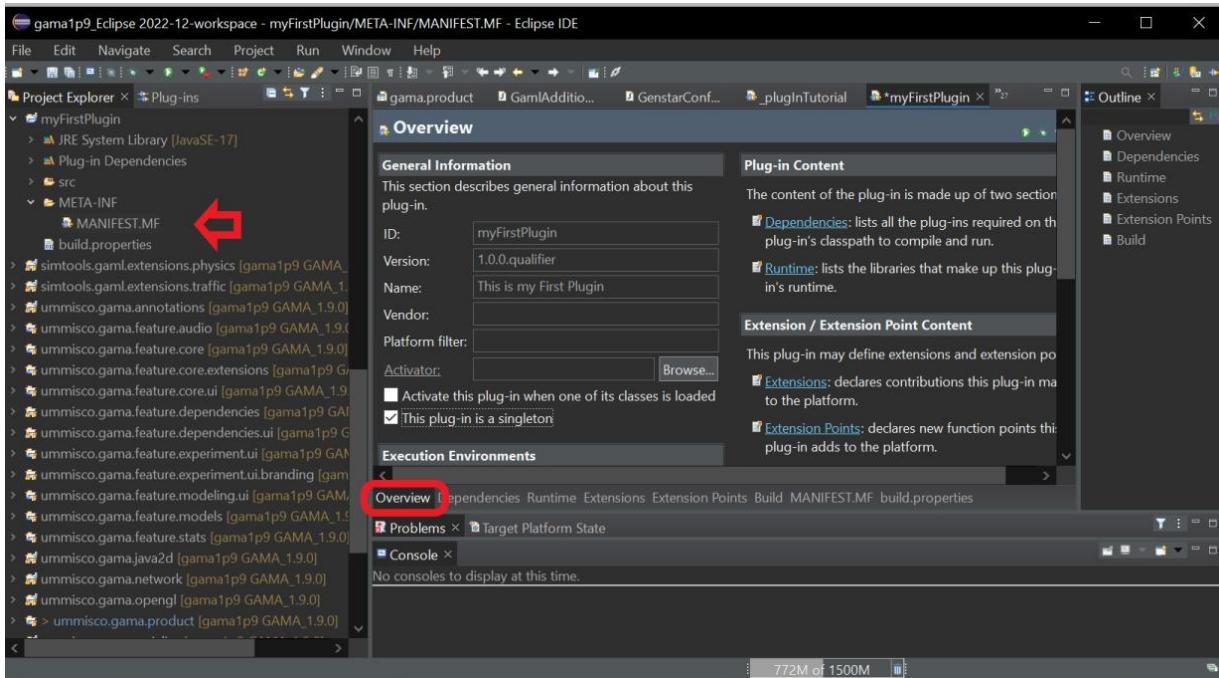
3. In the "New plug-in Project" / "Content" window:
 - i. Id : could contain the name of your institution and/or your project, e.g. « irit.maelia.gaml.additions »
 - ii. version 1.0.0.qualifier (this latter mention is important if you plan on distributing the plugin on GAMA update site)
 - iii. Name «This is my First Plugin.»
 - iv. Uncheck "Generate an activator, a Java class that controls the plug-in's life cycle" ,
 - v. Uncheck "This plug-in will make contributions to the UI"
 - vi. Check "No" when it asks "Would you like to create a rich client application ?"
 - vii. Click on "Next"



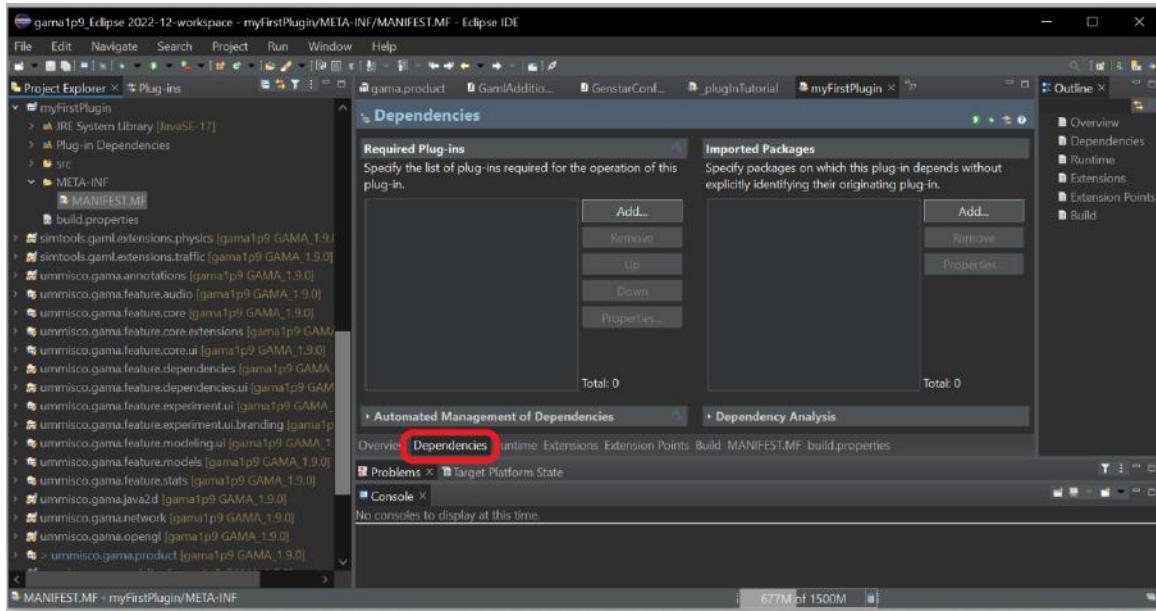
4. In the "New plug-in Project" / "Templates" window:
 - i. Uncheck "Create a plug-in using one of the templates"
 - ii. Click on "Finish"



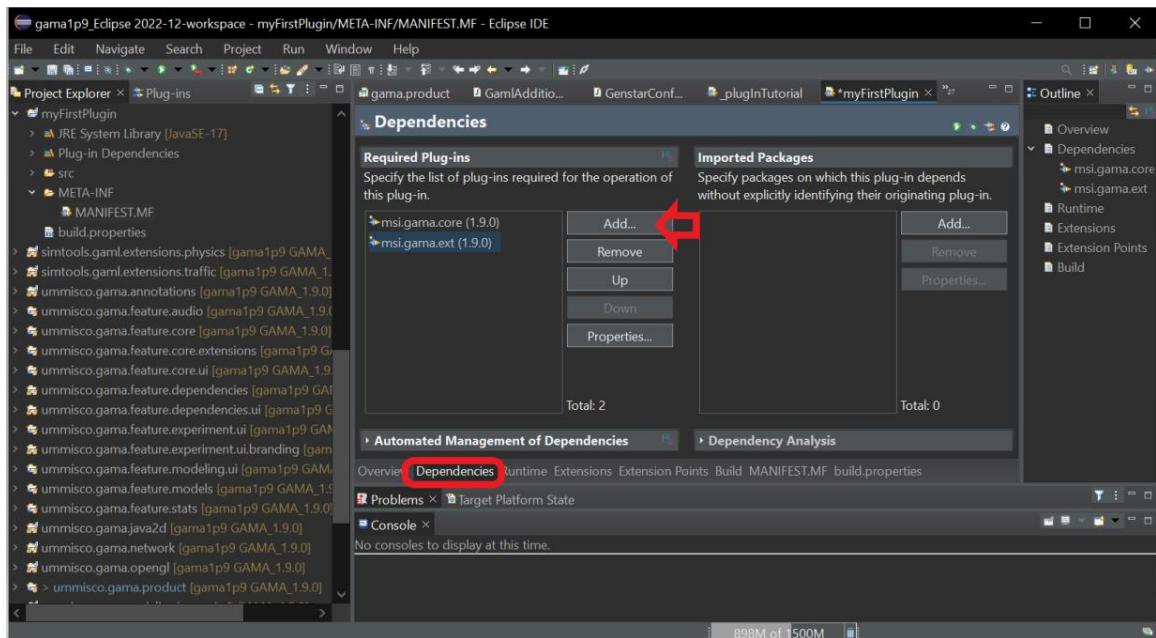
- iii. Your plug-in has been created.
5. Edit the file "Manifest.MF":
 - i. From the Project Explorer pane, expand your plugin folder.
 - ii. Click on the META-INF folder.
 - iii. Click on the MANIFEST.MF file.



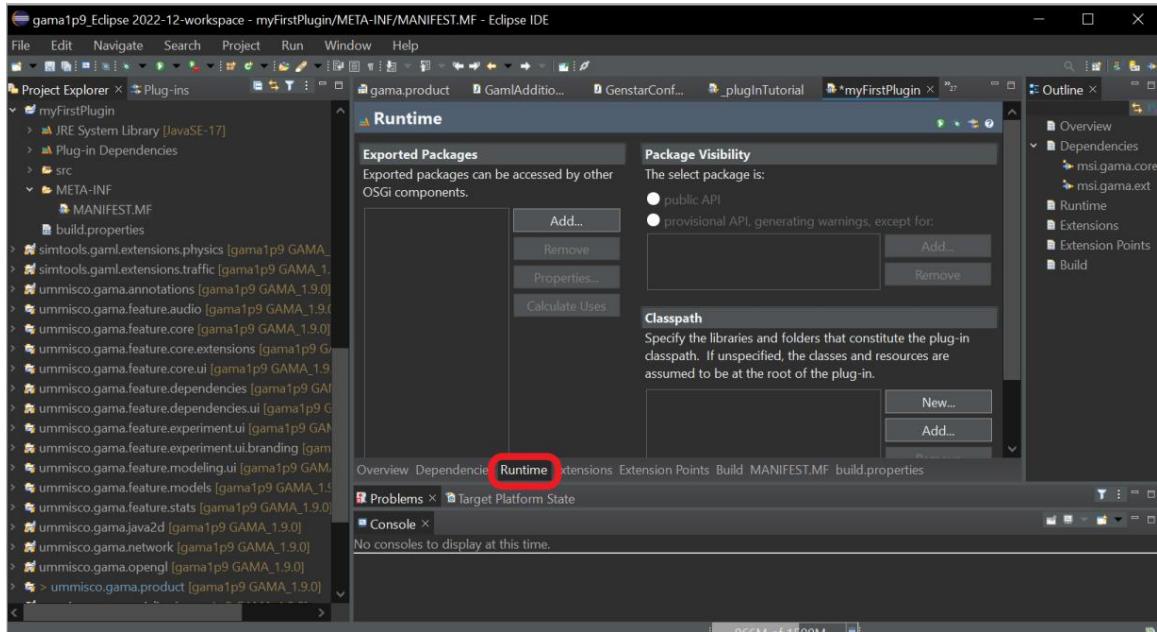
- iv. Click on the Overview tab to open the **Overview pane**:
- v. On the Overview pane, check as shown « This plug-in is a singleton »
- vi. Dependencies pane:
 - a. Click on the Dependencies tab to open the Dependencies Pane.



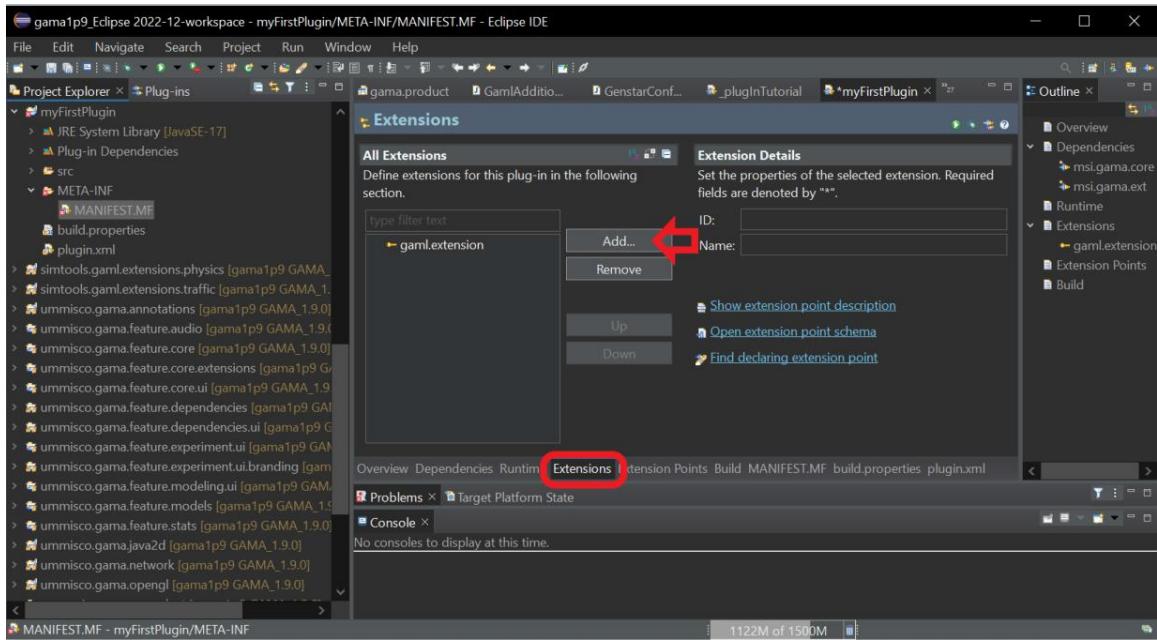
- b. add (at least minimum) the two plug-ins "msi.gama.core" and "msi.gama.ext" in the "Required Plug-ins". When you click on "Add", a new window will appear without any plug-in. Just write the beginning of the plug-in name in the text field under "Select a plug-in"



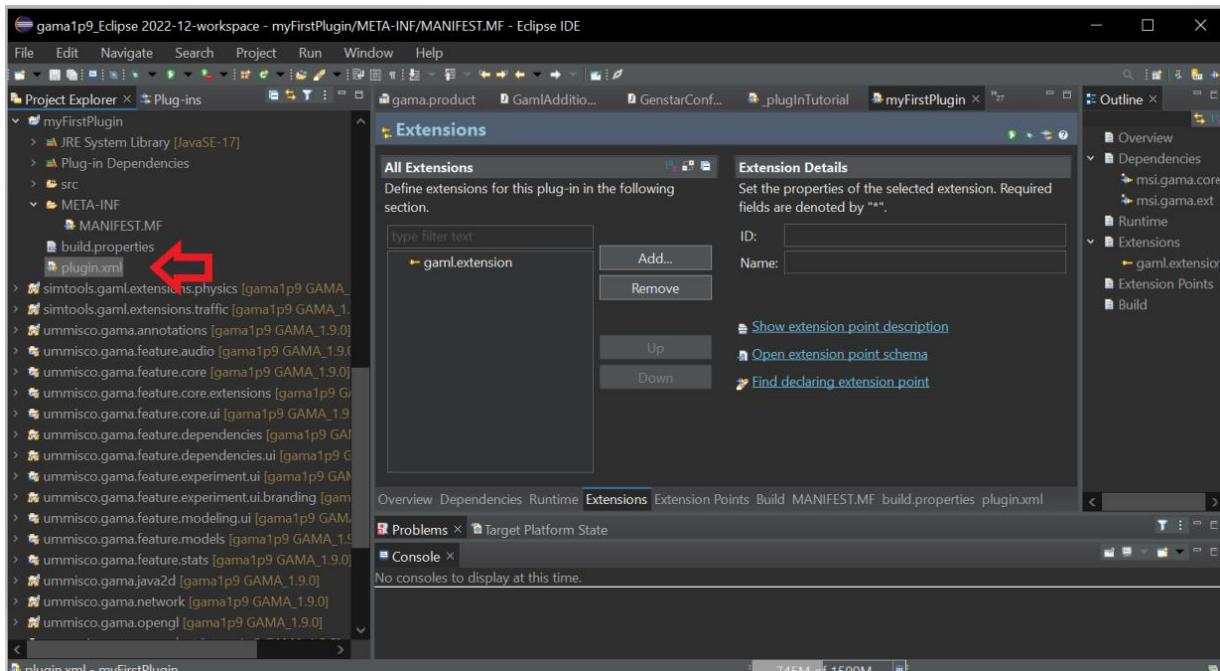
- vii. Click on the Runtime tab to open the Runtime pane:



- a. In exported Packages: nothing (but when you will have implemented new packages in the plug-in you should add them there)
 - b. Add in the classpath all the additional libraries (.jar files) used in the project.
- viii. Click on the Extension tab to open the Extensions pane:
- Click the Add button and add "gaml.extension"

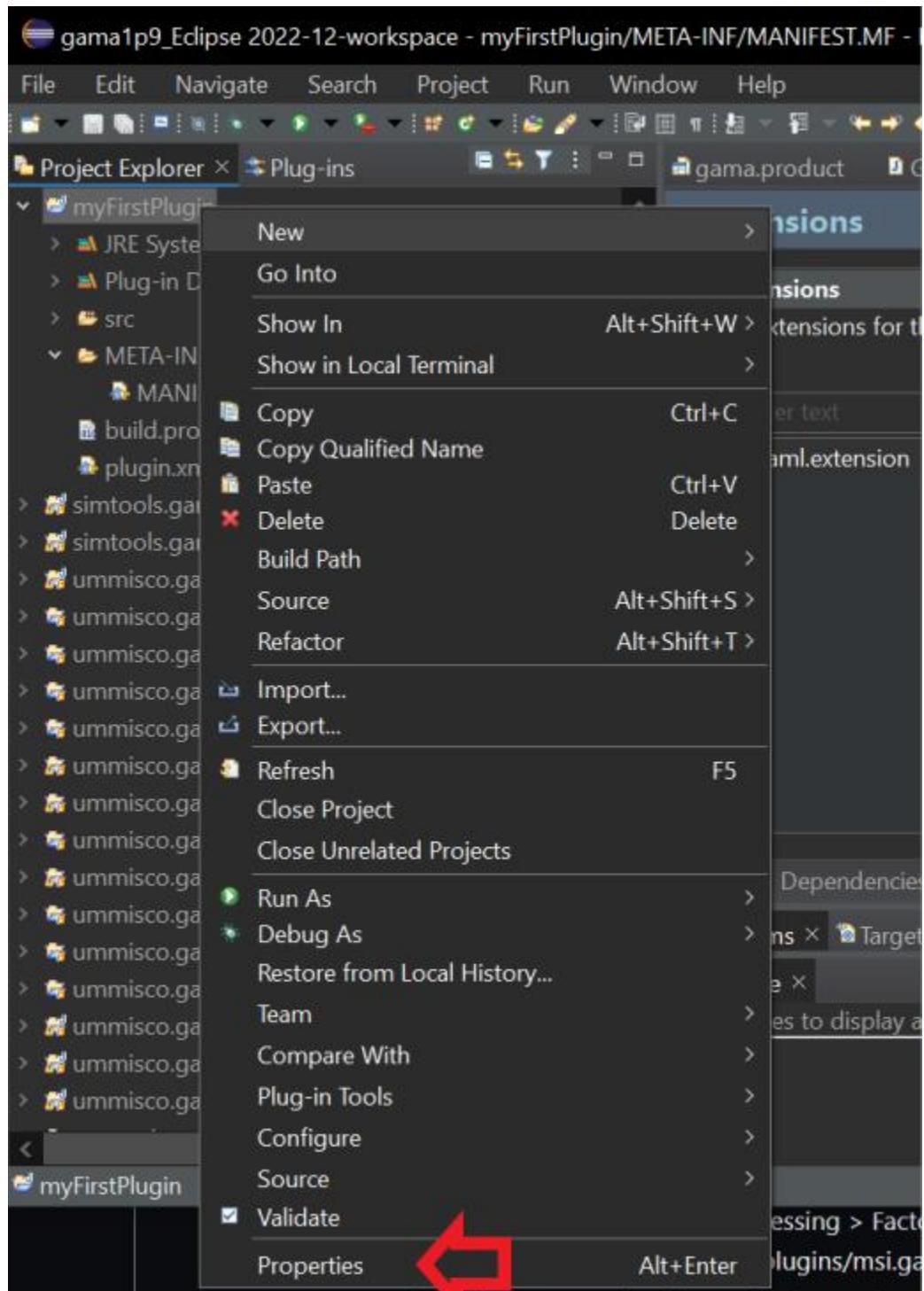


6. On the main menu, click on File, then select, Save, to save the file. This should create a "plugin.xml" file.

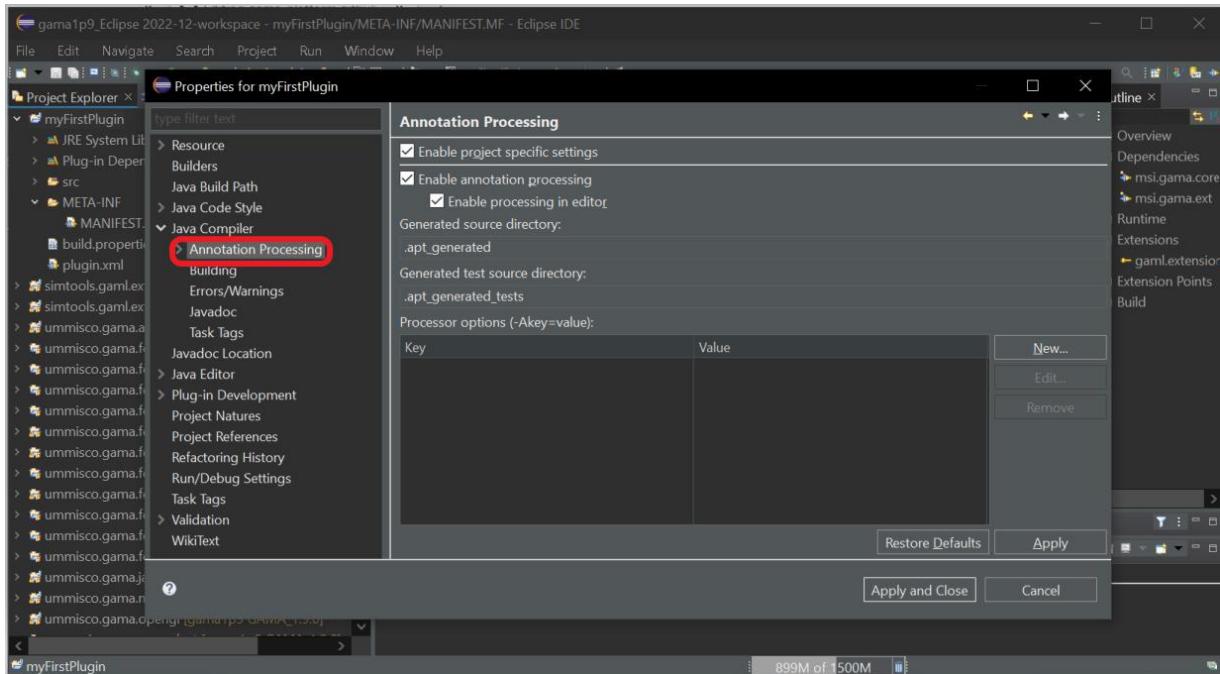


7. In the Project Explorer pane, Select the plugin, right_click, and in the dropdown

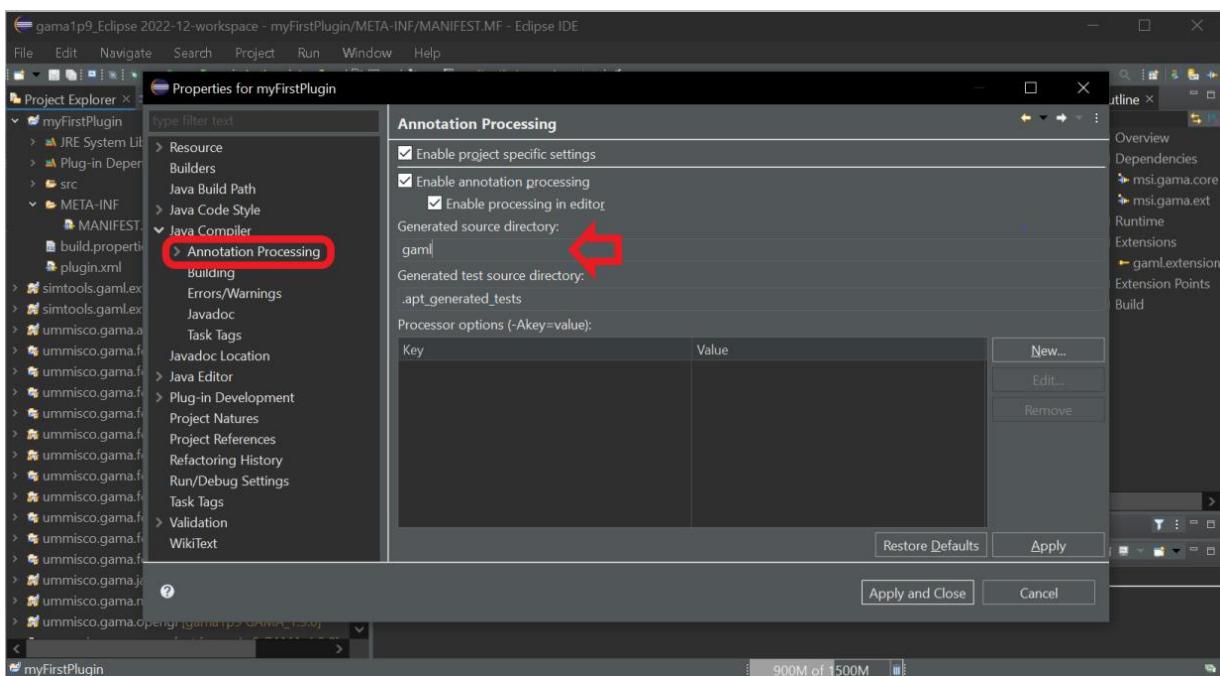
menu select Properties:



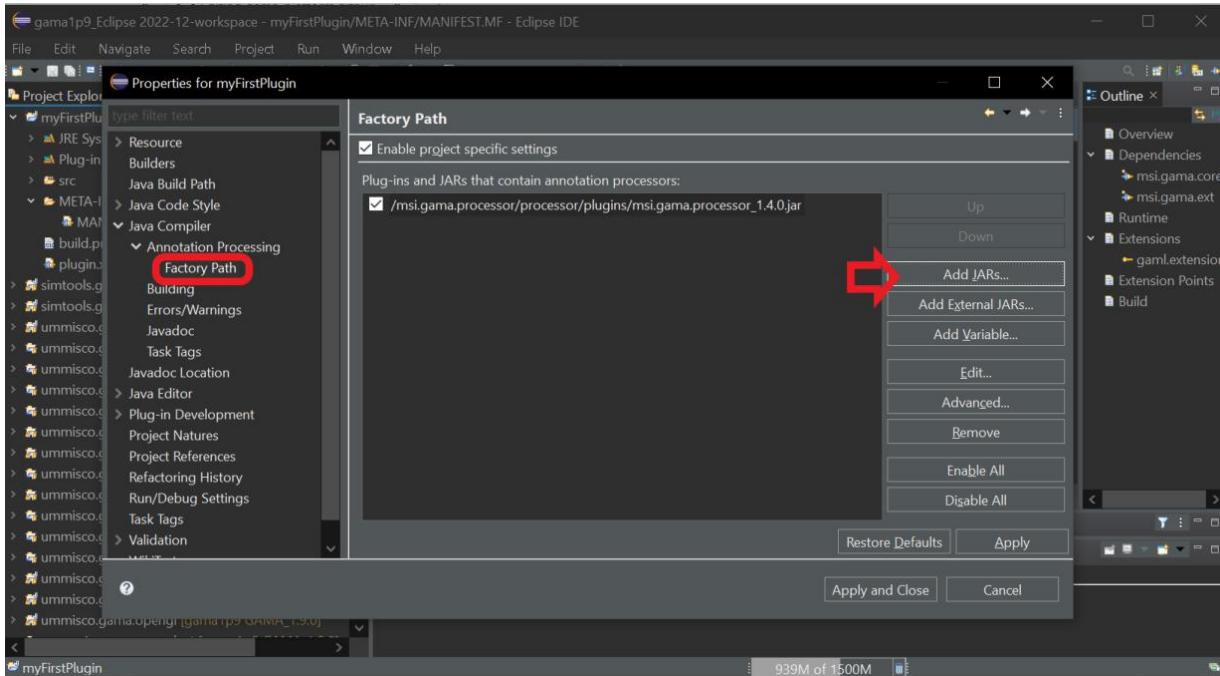
- i. The Properties for myFirstPlugin dialog opens as shown.



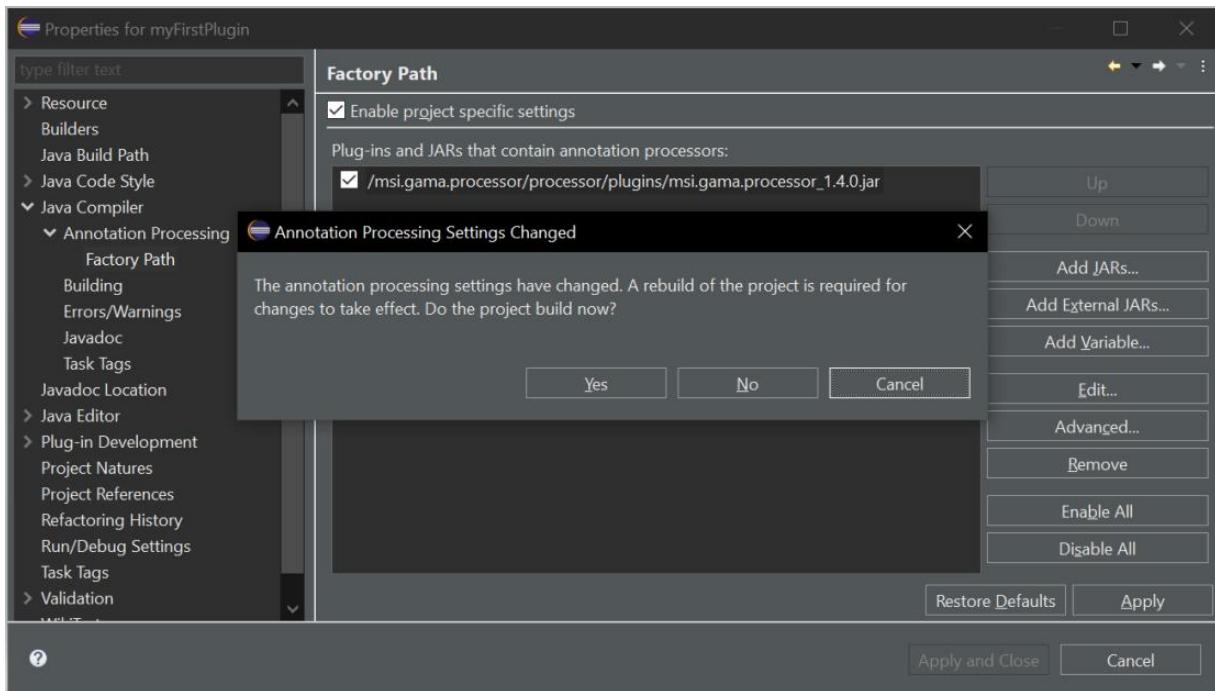
- ii. In the Properties dialog .. Go to Java Compiler, then Annotation Processing:
check "Enable project specific settings", then in "Generated Source
Directory", change ".apt_generated" to "gaml",



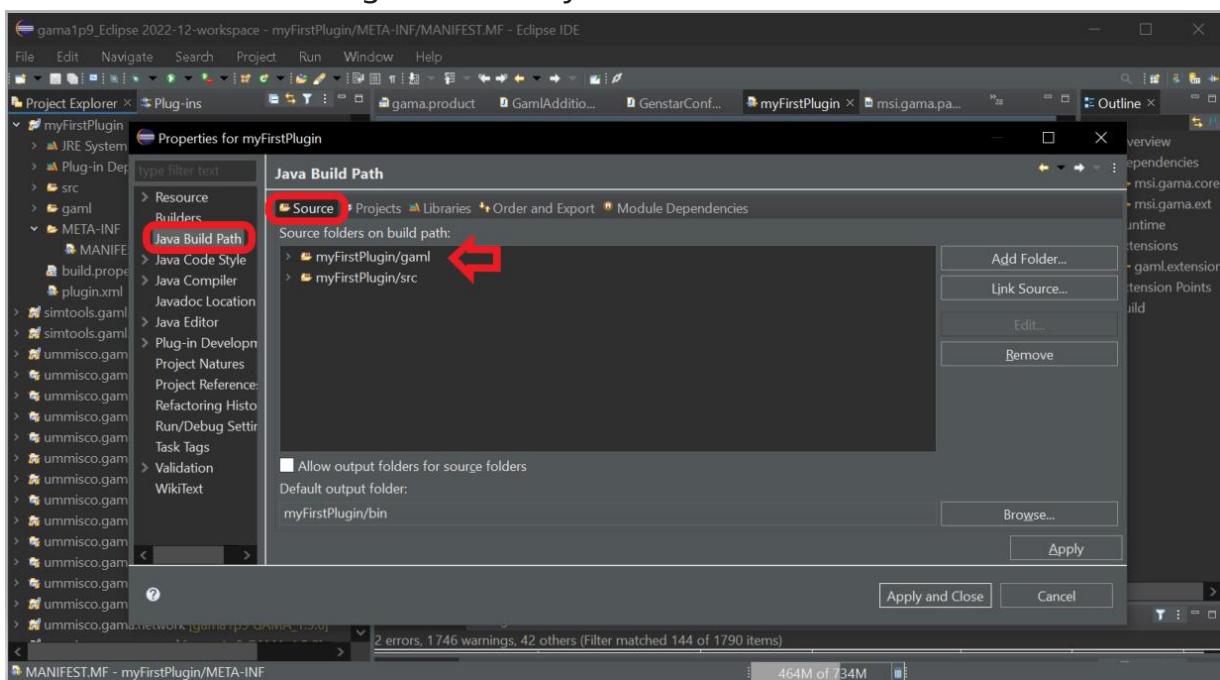
- iii. Go again to Java Compiler, then Annotation Processing, then Factory path: check "Enable project specific settings", then "Add Jars" and choose "msi.gama.processor/processor/plugins/msi.gama.processor.1.4.0.jar"



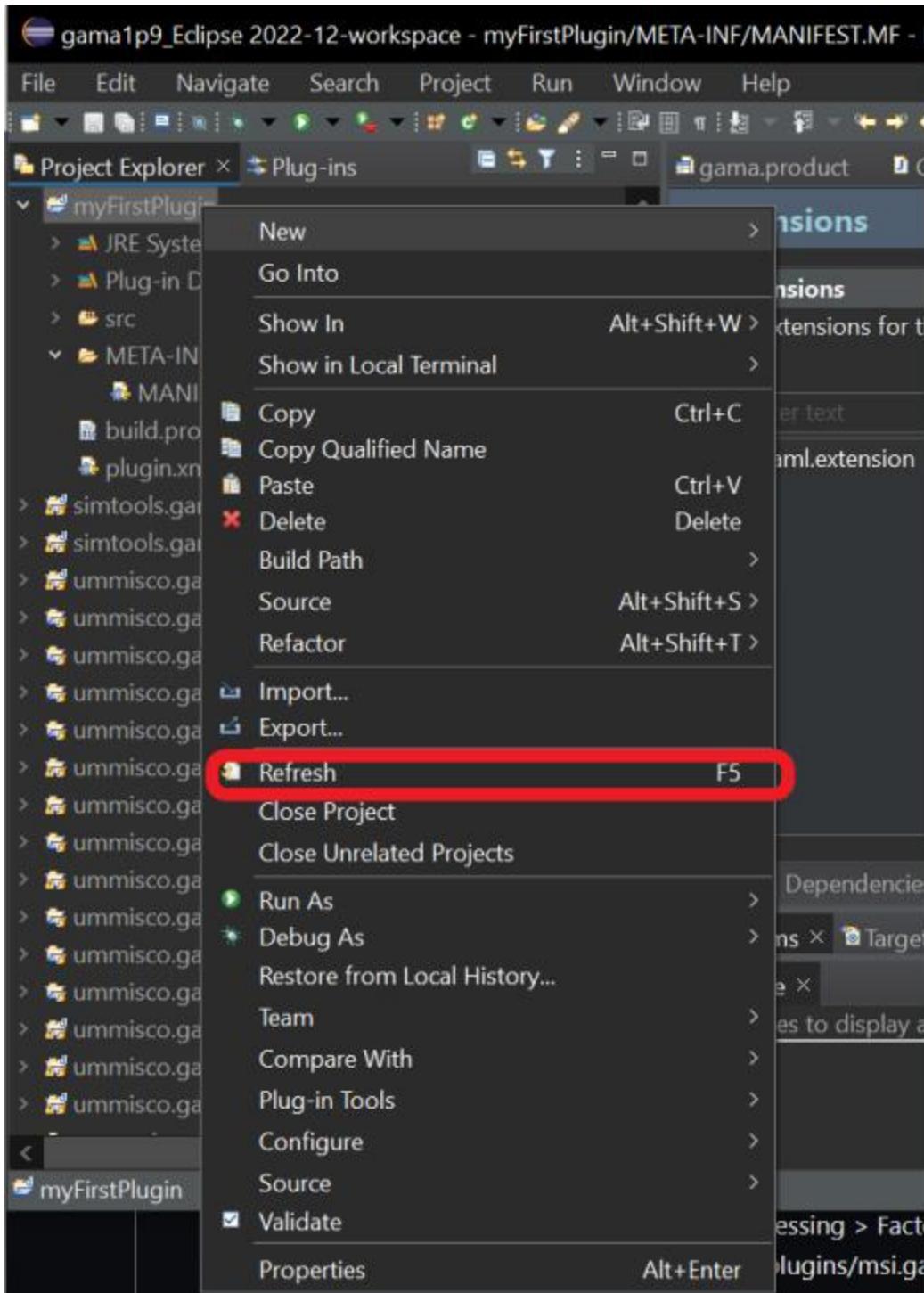
- iv. Close the menu. Click on Yes in the succeeding dialogs (Annotation settings changed ..). After, this should compile the project and create the `gaml` directory.



8. Return to the Properties dialog of your plugin by clicking from the main menu bar, Project, then click on Properties. Go to Java Build Path, click on the Source Tab, and check that the `gaml` directory has been added.



9. If the gaml folder is not present, click on Add Folder and select the gaml directory. Right click on the project, then refresh it (F5 or from the File menu -> Refresh)



10. Now, there should be a `gaml` directory. This `gaml` directory will later contain the package containing `GamlAdditions.java`, and other related files created after

creating classes. If there is no package in the folder, try creating a class, then try to refresh or close the project and reopen it, or clean the projects by going into Project tabs, and clicking on clean.

The plug-in is ready to accept any addition to the GAML language, e.g. skills, actions, operators. To proceed to creating a skill click on this [link](#).

Do not forget to export the created packages that could be used by "clients", especially the packages containing the code of the additions (in the plugin.xml of the new project, tab "Runtime").

To test the plug-in and use it into GAMA, developers have to define a new feature project containing your plugin and its dependencies, and adds this feature to the existing product (or a new .product file of your own).

The use of feature is also mandatory to define a plug-in that can be uploaded on the update site and can be installed in the release of GAMA.

Creation of a feature

A feature is an Eclipse project dedicated to gather one or several plug-ins to integrate them into a product or to deploy them on the update site and install them from the GAMA release (a feature is mandatory in this case).

Here are detailed steps to create and configure a new feature.

- File > New > Feature project (or File > New > Project... then Plug-in Development > Feature Project)
- In Feature properties
 - Choose a project name (e.g. "institution.gama.feature.pluginsName")
 - Click on "Next"

- In Referenced Plug-ins and fragments
 - Check "Initialize from the plug-ins list:"
 - Choose the plug-ins that have to be gathered in the feature
 - Click on "Finish"
- A new project has been created. The "feature.xml" file will configure the feature.
 - In "Information pane":
 - You can add description of the various plug-ins of the feature, define the copyright notice and the licence.
 - In "Plug-ins and Fragments"
 - In the Plug-ins and Fragments, additional plug-ins can be added.

Addition of a feature to the product

To load the plugin into GAMA, go into the project `ummisco.gama.product` and open `gama.product` and go into the overview tab, under the section Testing, click Synchronize, go into the contents tab, click on Add, and add the features related to your plugin. Click the Run tab in the main menu bar, click on Run Configuration, then you should have the gama runtime product window open, click on the plugins tab, and check your plugin in the list. Click on Apply. Now your plugin is accessible in GAMA, now we can run the application. Click on Run.

Remark: To check whether the new plug-in has been taken into account by GAMA, after GAMA launch, it should appear in the Eclipse console in a line beginning by ">> GAMA bundle loaded in".

If you plan to deploy your plugin to be used by other users from the GAMA community, proceed with the succeeding steps. If not, we can proceed with the creation of skills and types.

In the product, e.g. `gama.product` in the `ummisco.gama.product` project:

- Contents pane
 - Click on Add button
 - In the window select the feature
 - Click on OK.

Create examples model

In order to make your plugin usable by everyone, it is very important to bring potential users model examples to introduce new gaml primitives, statements and operators. This way, modelers can easily get into the plugin you developed in a practical way.

The process is twofold:

- Mount your plugin into your GAMA (see below or use the Git version)
- Create a new project in the user model folder. Put your GAMA model examples there.
- Move your project into a folder called "models" at the root of the plugin

Hence this is done, you can update your Plugin models library folder and have access to the plugin models

How to make a plug-in available at GAMA update site for the GAMA release

Considering a working GAMA plugin named institution.gama.pluginsName

Configure plugin to be available for Maven

a/ Add `pom.xml` for plugin `institution.gama.pluginsName`:

- Right click -> Configure -> Convert to maven project to add pom.xml:
- Set:
 - Group id: institution.gama.pluginsName
 - Artifact id: institution.gama.pluginsName
 - Version: 1.0.0-SNAPSHOT // must have -SNAPSHOT if the plugin version is x.x.x.qualifier
 - Packaging: eclipse-plugin // this element is not in the list (jar/pom/war) because of the incompatible of tycho, maven and eclipse, so just type it in although it will be a warning
- Finish

b/ Configure pom.xml to recognize the parent pom.xml for Maven builds

- Open pom.xml in `institution.gama.pluginsName`
- Tab overview, Parent section, type in:
 - Group id: msi.gama
 - Artifact id: msi.gama.experimental.parent
 - Version: 1.7.0-SNAPSHOT
 - Relative path: ..`/msi.gama.experimental.parent`
- Save

c/ Update maven cache in eclipse (optional) It will fix this compilation error "Project configuration is not up-to-date with pom.xml. Select: Maven->Update Project... from the project context menu or use Quick Fix."

- Right click -> Maven -> Update project

Create a feature for the plugin

a/ Create new feature

- New -> Project -> type in : feature -> Select "Feature Project"
- Set:
 - Project name: institution.gama.feature.pluginsName
 - Uncheck use default location, type in: {current git repository}\aaa.bbb.feature.ccc
 - Feature Version: 1.0.0.qualifier
 - Update Site URL: <http://updates.gama-platform.org/experimental>
 - Update Site Name: GAMA 1.7.x Experimental Plugins Update Site
- Click Next
 - Initialize from the plugin list -> check all plugins needed:
institution.gama.pluginsName (1.0.0.qualifier)
- Finish

b/ Add pom.xml for feature institution.gama.feature.pluginsName:

- Right click -> Configure -> Convert to maven project (to add pom.xml)
- Set:
 - Group id: institution.gama.feature.pluginsName
 - Artifact id: institution.gama.feature.pluginsName
 - Version: 1.0.0-SNAPSHOT
 - Packaging: eclipse-feature
- Finish

c/ Configure pom.xml to recognize the parent pom.xml for Maven builds

- Open pom.xml in institution.gama.pluginsName
- Tab overview, Parent section, type in:

- Group id: msi.gama
 - Artifact id: msi.gama.experimental.parent
 - Version: 1.7.0-SNAPSHOT
 - Relative path: ../msi.gama.experimental.parent
- Save

d/ Update maven cache in eclipse (optional) It will fix this compilation error "Project configuration is not up-to-date with pom.xml. Select: Maven->Update Project... from the project context menu or use Quick Fix."

- Right click -> Maven -> Update project

Update p2updatesite category.xml (this step will be done automatically by travis, soon)

Open msi.gama.experimental.p2updatesite

- Tab Managing the Categories -> Add feature ->
institution.gama.feature.pluginsName

How to make a plug-in available as an extension for the GAMA release (obsolete)

Once the plug-in has been tested in the GAMA SVN version, it can be made available for GAMA release users.

First, the `update_site` should be checked out from the SVN repository:

- File > New > Other... > SVN > Project from SVN
- In Checkout Project from SVN repository
 - Use existing repository location (it is the same location as for the GAMA

- code)
- Next
- In Select resource:
 - Browse
 - choose svn > update_site
 - Finish
- Finish

Now the update_site project is available in the project list (in Package Explorer). The sequel describes how to add a new feature to the update site.

- Open the `site.xml` file
- In update site Map:
 - Click on Extensions
 - click on the Add Feature... button
 - Choose the feature to be added
 - It should appear in Extensions
 - Select the added feature and click on the Synchronize... button
 - Check Synchronize selected features only
 - Finish
 - Select the added feature and click on the Build button
- All the files and folder of the update_site project have been modified.
- Commit all the modifications on the SVN repository
 - Right-click on the project, Team > Update
 - Right-click on the project, Team > Commit...

The plug-in is now available as an extension from the GAMA release. More details about the update of the GAMA release are available [on the dedicated page](#).

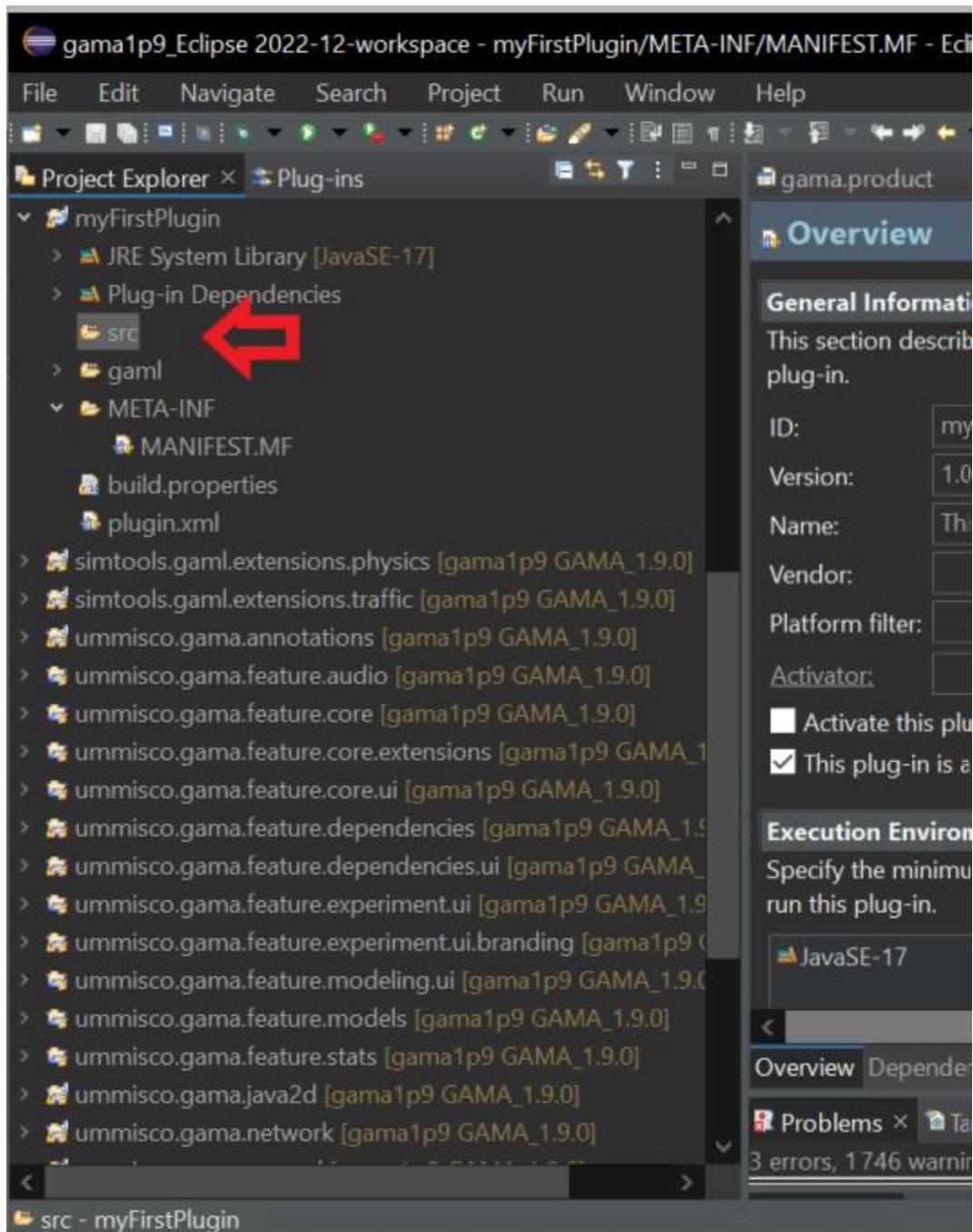
Version: 1.9.1

DevelopingSkills

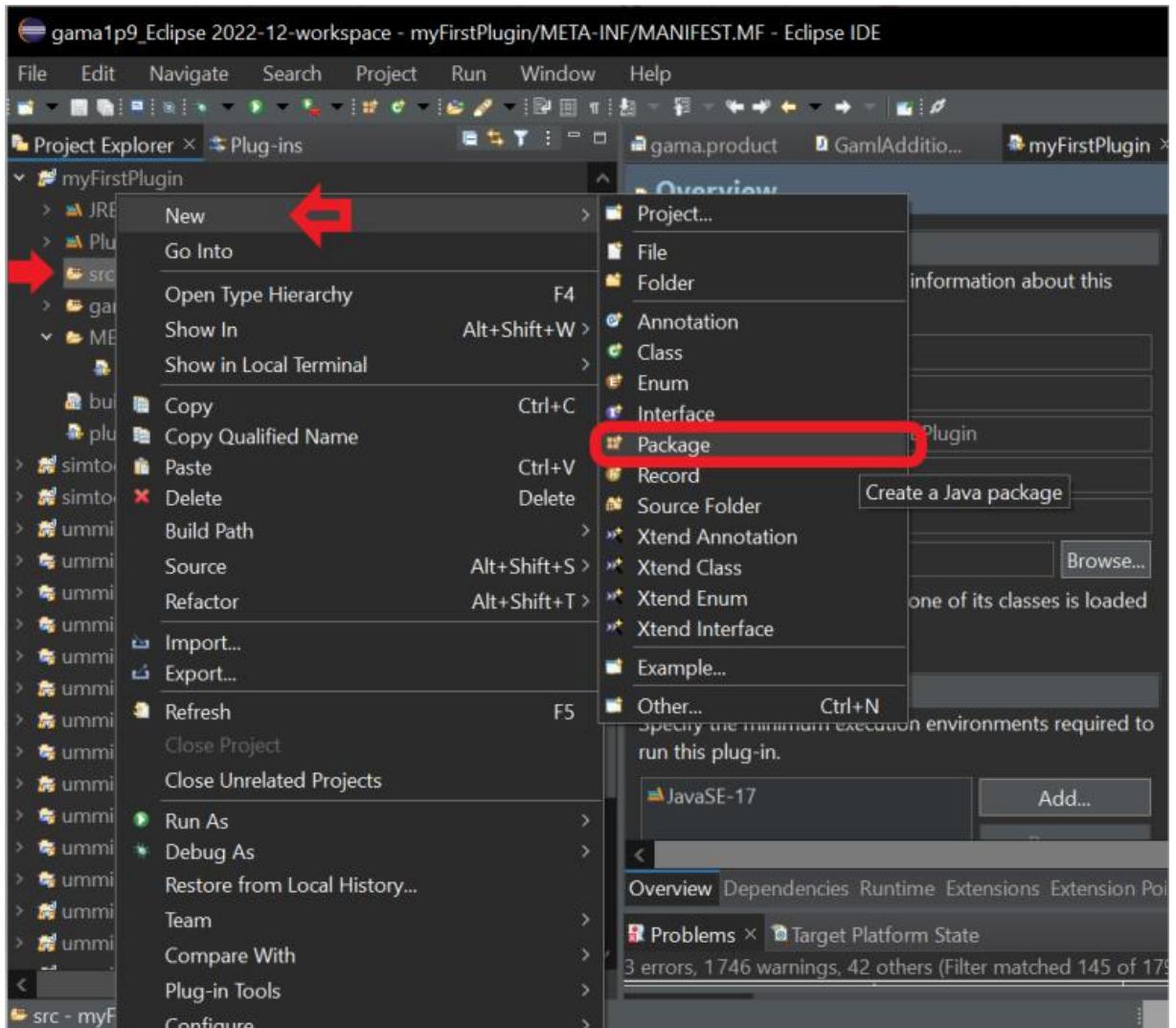
Defining the class as a skill

In this section we will be creating a class "FirstClass", that is included in a package named "skill".

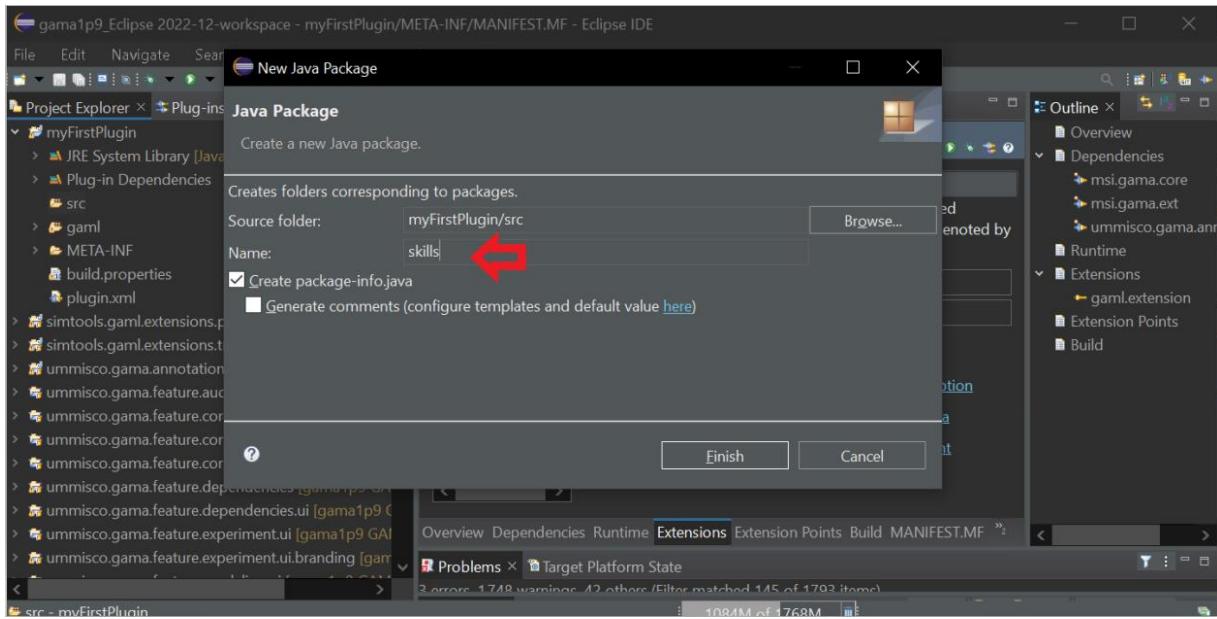
1. Create Package
 - i. From the Project Explorer pane, go to your plugin's folder, and then go to the src folder.



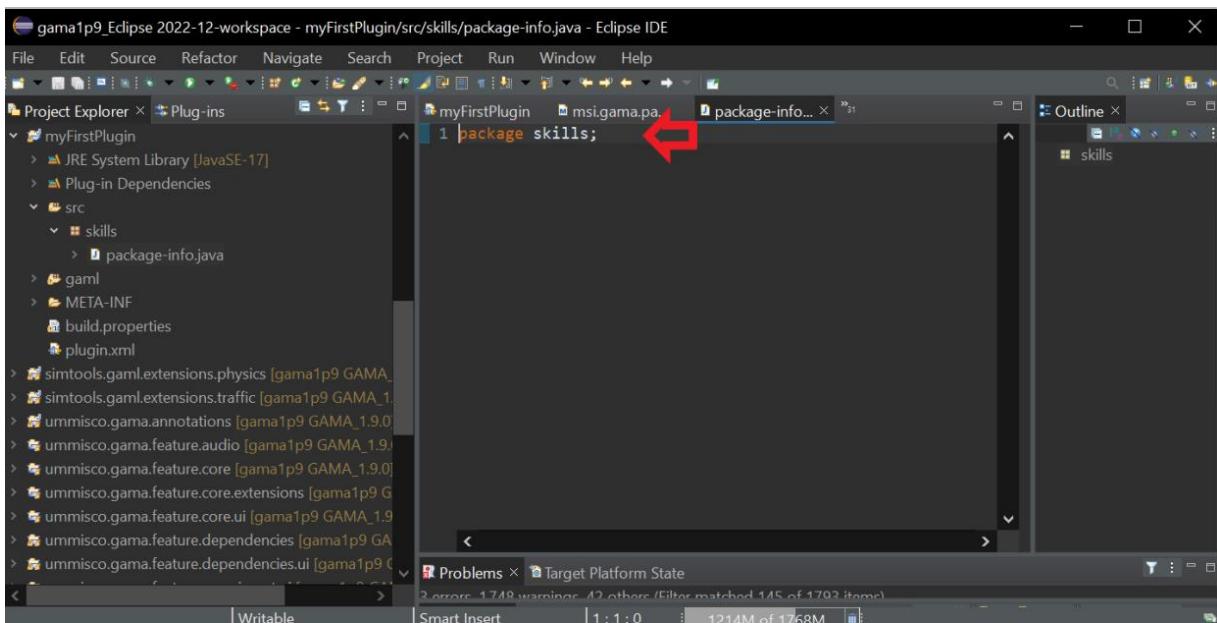
- ii. Right click on the src folder. Click on New > Package.



iii. In the New Java Package dialog, set the Name to "skills".



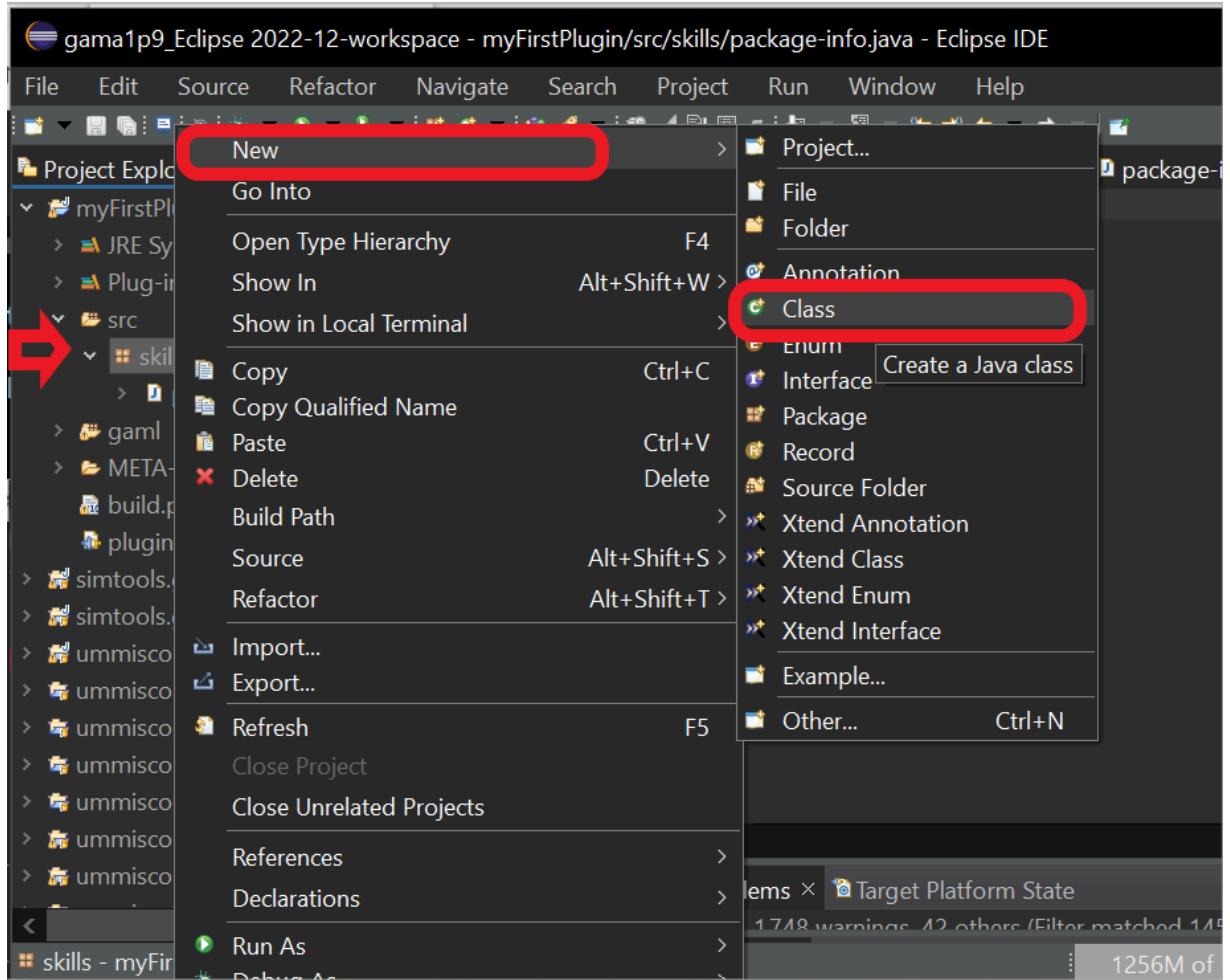
iv. Click on Finish. This will create the package "skills".



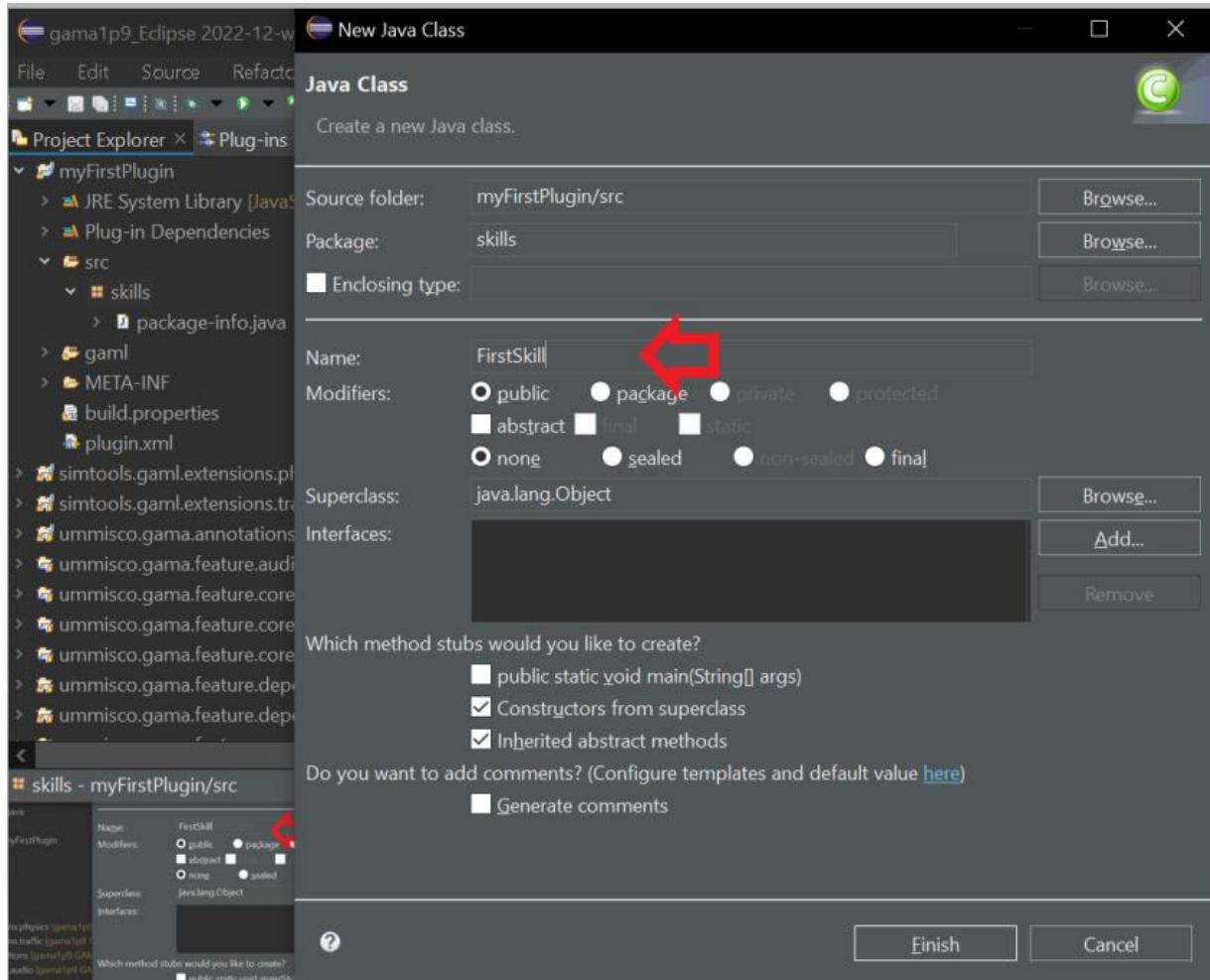
2. Create class in the Package

- In the Project Explorer, go to the folder of the Plugin. We create the Java class in this package. To do this, right click on the package, then click on

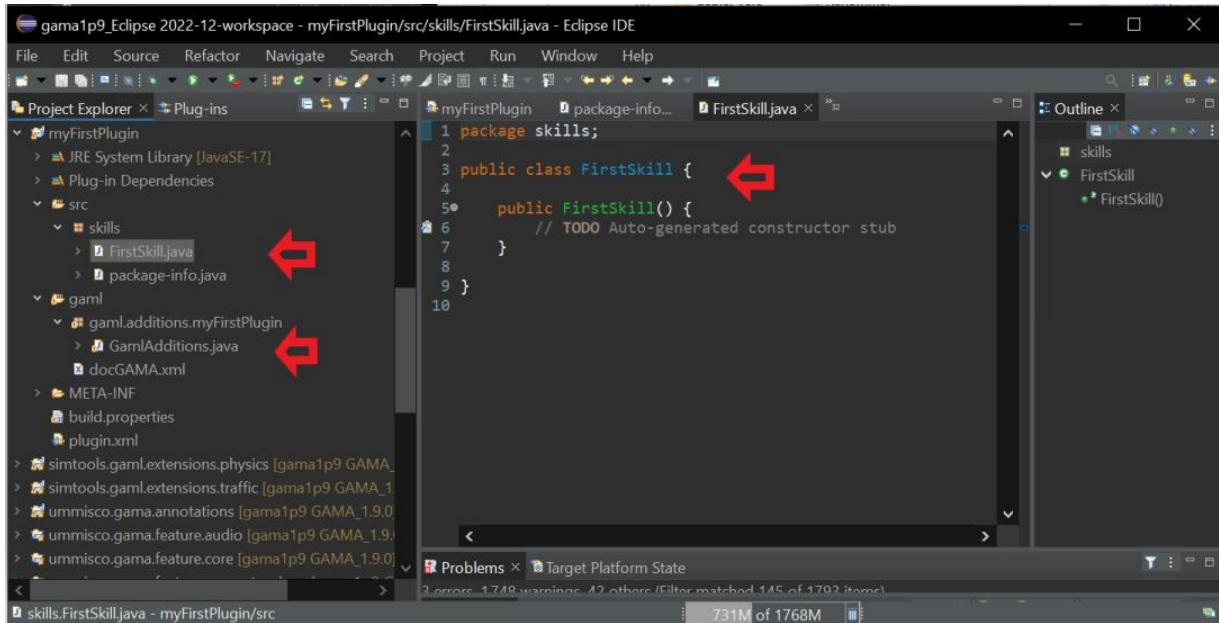
New, select Class.



- ii. On the New Java class dialog, set the Name to 'FirstSkill'. Click on Finish.



- iii. Now we have an empty Java class named FirstSkill. Note as well that the GamlAdditions.java file was also automatically added into the gaml/gaml.additions.myFirstPlugin folder.



Using Annotations to define class as a skill

Annotations are used to link Java methods and classes to GAMA language. Note that GAMA annotations are classes defined into the `msi.gama.precompiler.GamlAnnotations` class.

We need to tell GAMA that our class "FirstSkill" will be used as skill. To do that we will use Annotation in writing the code of the class. The annotations for skill is described as follows:

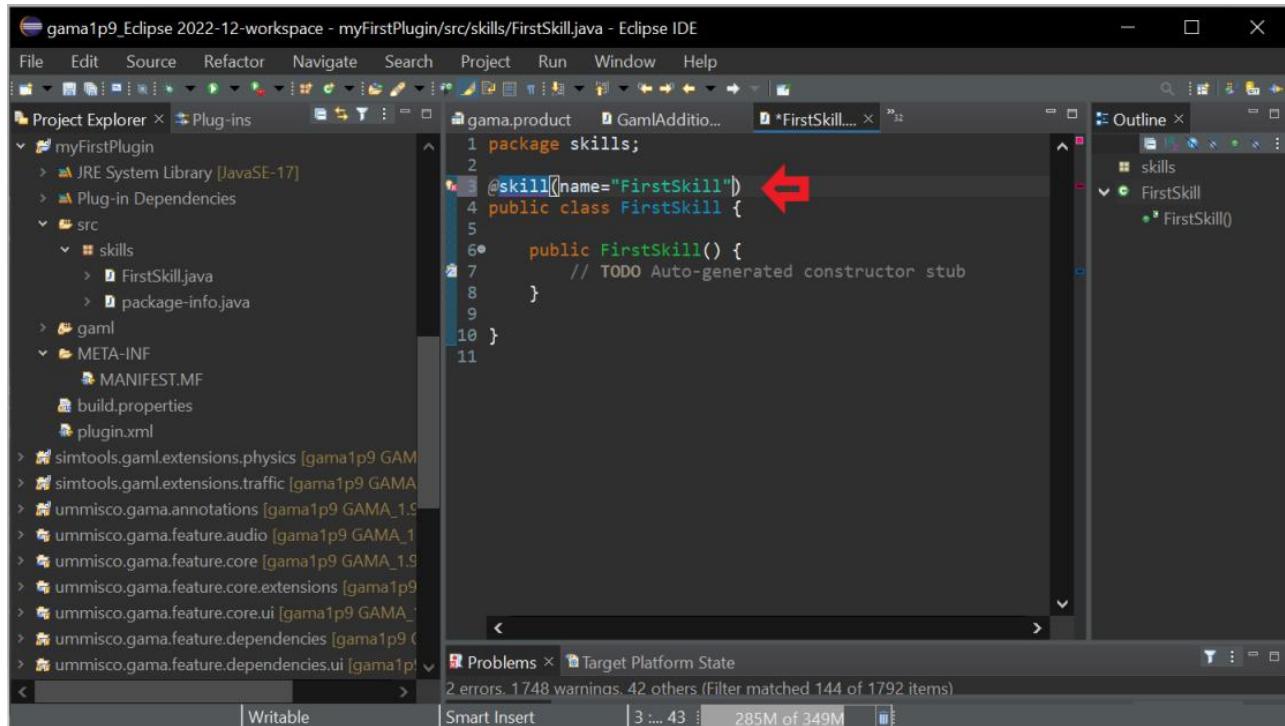
@skill

This annotations Allows to define a new skill (class grouping variables and actions that can be used by agents).

This annotation contains:

- **name** (String): a String representing the skill name in GAML (must be unique throughout GAML).
- **attach_to** (set of strings): an array of species names to which the skill will be automatically added (complements the "skills" parameter of species).
- **internal** (boolean, false by default): return whether this skill is for internal use only.
- **doc** (set of @doc, empty by default): the documentation associated to the skill.
 - Just before the class declaration, add this line to indicate that this class is a skill named "FirstSkill"

```
@skill(name = "FirstSkill")
```



In line 3, a red broken line under the skill word can be seen. This is an error since at this point in the code. Hover your mouse over the error, and the message box would indicate that the skill cannot be resolved to a type.

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the project structure with a red arrow pointing to the "src" folder.
- Code Editor:** Displays the file `FirstSkill.java` containing the following code:

```
1 package skills;
2
3 @skill(name="FirstSkill")
4
5 public class FirstSkill {
6     ...
7 }
```

A red arrow points to the annotation `@skill(name="FirstSkill")`, which is underlined and has a tooltip: "skill cannot be resolved to a type". A context menu is open with the following options:
 - Import 'skill' (msi.gama.precompiler.GamaAnnotations)
 - Create annotation 'skill'
 - Fix project setup...
- Outline View:** Shows the class `FirstSkill` and its method `FirstSkill()`.
- Problems View:** Shows 2 errors, 1748 warnings, and 42 others.

To correct this error, we take the suggestion of importing 'skill' (msi.gama.precompiler.GamaAnnotations). Click on this option.

The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the project structure with a red arrow pointing to the "src" folder.
- Code Editor:** Displays the file `FirstSkill.java` containing the following code:

```
1 package skills;
2
3 import msi.gama.precompiler.GamaAnnotations.skill;
4
5 @skill(name="FirstSkill")
6 public class FirstSkill {
7     ...
8 }
```

A red arrow points to the `import` statement `import msi.gama.precompiler.GamaAnnotations.skill;`.
- Outline View:** Shows the class `FirstSkill` and its method `FirstSkill()`.
- Problems View:** Shows 2 errors, 1748 warnings, and 42 others.

It is a good practice to define all the names of plugins/actions/variables in the IKeyword class, which is located in the plugin msi.gama.common.interfaces.IKeyword.java.

In this class you can define a string variable which contain the name of your plugin.

```
String FIRST_SKILL = "FirstSkill";
```

Now that we have defined a global string containing the name of the plugin, we can use it in the annotation and in the code of the class.

```
@skill(name = IKeyword.FIRST_SKILL)
```

We have now defined that our current class is a skill that can be used in GAMA.

To use our plugin in gaml model, we have to create a species using the skill:

```
species tutorialSpecies skills:[FirstSkill]  
{  
}
```

Defining new attributes for the skill

Now we have a skill that is empty. So we need to add some variable for the plugin to have a purpose.

To add new attributes to the species that declares this skill, we have to define them before the class using annotation like we did before.

@variable

This annotations is used to describe a single variable or field.

This annotation contains:

- **name** (String): *the name of the variable as it can be used in GAML.*
- **type** (int): *The textual representation of the type of the variable (see IType).*
- **of** (int, 0 by default): *The textual representation of the content type of the variable (see IType#defaultContentType()).*
- **index** (int, 0 by default): *The textual representation of the index type of the variable (see IType#defaultKeyType()).*
- **constant** (int, false by default): *returns whether or not this variable should be considered as non modifiable.*
- **init** (String, "" by default): *the initial value of this variable as a String that will be interpreted by GAML.*
- **depend_on** (set of Strings, empty by default): *an array of String representing the names of the variables on which this variable depends (so that they are computed before).*
- **internal** (boolean, false by default): *return whether this var is for internal use only.*
- **doc** (set of @doc, empty by default): *the documentation associated to the variable.*

The `@vars` annotation contains a set of `@variable` elements.

Just like we did before with the declaration of the name of our skill, we can declare globally the name of our incoming new variables, in the IKeyword class.

```
String FIRST_VARIABLE = "FirstVariable"; String SECOND_VARIABLE =  
"SecondVariable";
```

Now that we have defined the names of our variables, we need to declare them in our skill class.

Here is how to declare the variables:

```
@vars({
    @variable(name = IKeyword.FIRST_VARIABLE, type = IType.INT, init =
"1"),
    @variable(name = IKeyword.SECOND_VARIABLE, type = IType.FLOAT, init
= "1.0")
})
```

In order to access these new attributes, in the GAMA application, you need to define `@getter` and `@setter` methods:

@getter

This annotations is used to indicate that a method is to be used as a getter for a variable defined in the class. The variable must be defined on its own (in vars).

This annotation contains:

- **value** (String): the name of the variable for which the annotated method is to be considered as a getter.
- **initializer** (boolean, false by default): returns whether or not this getter should also be used as an initializer

@setter

This annotations is used to indicate that a method is to be used as a setter for a variable defined in the class. The variable must be defined on its own (in vars).

This annotation contains:

- **value** (String): the name of the variable for which the annotated method is to be considered as a setter.

```

@getter(IKeyword.FIRST_VARIABLE)
public int getFirstVariable(final IAgent agent) {
    return (int) agent.getAttribute(IKeyword.FIRST_VARIABLE);
}

@Setter(IKeyword.FIRST_VARIABLE)
public void setFirstVariable(final IAgent agent, final int value) {
    agent.setAttribute(IKeyword.FIRST_VARIABLE, value);
}

@Getter(IKeyword.SECOND_VARIABLE)
public double getFirstVariable(final IAgent agent) {
    return (double) agent.getAttribute(IKeyword.SECOND_VARIABLE);
}

@Setter(IKeyword.SECOND_VARIABLE)
public void setFirstVariable(final IAgent agent, final double value) {
    agent.setAttribute(IKeyword.SECOND_VARIABLE, value);
}

```

At this point we can use the variable defined directly in our agent :

```

species tutorialSpecies skills:[FirstSkill]
{
    init
    {
        write(firstVariable);
        write(SecondVariable);
    }
}

```

Defining new actions

An action (also called `primitive`) is basically a Java method that can be called from

the GAML language using the same syntax as the one used for calling actions defined in a model. The method should be annotated with `@action`, supplying the name of the action as it will be available in GAML.

@action

This annotations is used to tag a method that will be considered as an action (or primitive) in GAML. The method must have the following signature: `Object methodName(IScope) throws GamaRuntimeException` and be contained in a class annotated with `@species` or `@skill` (or a related class, like a subclass or an interface).

This annotation contains:

- **name** (String): *the name of the variable as it can be used in GAML.*
- **virtual** (boolean, false by default): *if true the action is virtual, i.e. equivalent to abstract method in java.*
- **args** (set of arg, empty by default): *the list of arguments passed to this action. Each argument is an instance of arg.*
- **doc** (set of @doc, empty by default): *the documentation associated to the action.*

We can also define parameters for this action using the annotation `@arg` will a set of parameters names.

@arg

This annotations describes an argument passed to an action.

This annotation contains:

- **name** (String, "" by default): *the name of the argument as it can be used in GAML.*
- **type** (set of ints, empty by default): *An array containing the textual representation of the types that can be taken by the argument (see IType).*

- **optional** (boolean, true by default): *whether this argument is optional or not.*
- **doc** (set of @doc, empty by default): *the documentation associated to the argument.*

Here is an example of an empty action for our skill (dont forget to define every keywords like `IKeyword.NUMBER_TO_ADD` in the `IKeyword` class):

```
@action(name = "add",
        args = {
            @arg(name = IKeyword.NUMBER_TO_ADD, type = IType.INT,
optional = false)})
@doc("Function to add a number to FirstVariable")
public int add(final IScope scope){
    return 0;
}
```

Now that we have defined the action we can access the parameter `IKeyword.NUMBER_TO_ADD` and use it as we want.

Access to parameters in actions

To get the value of the arguments passed in GAML to the Java code, two methods can be useful:

- `scope.hasArg("name_of_argument")` returns a boolean value testing whether the argument "name_of_argument" has been defined by the modeler, since all the arguments to actions should be considered as optional.
- `getArg(name_arg, IType)`, `getFloatArg(name_param_of_float)`, `getIntArg(name_param_of_int)` and their variants return the value of the given parameter using a given (or predefined) type to cast it.

Warnings

Developers should notice that:

- the method should have only one parameter: the scope (type IScope).
- the method can only throw `GamaRuntimeExceptions`. Other exceptions should be caught in the method and wrapped in a `GamaRuntimeException` before being thrown.

Here is the complete action code :

```
@action(name = "add",
        args = {
            @arg(name = IKeyword.NUMBER_TO_ADD, type = IType.INT,
optional = false)})
@doc("Function to add a number to FirstVariable")
public int add(final IScope scope)
{
    int firstVariable = getVariable(scope.getAgent());
    int numberToAdd = (Integer) scope.getArg(IKeyword.NUMBER_TO_ADD);

    setVariable(scope.getAgent(), firstVariable + numberToAdd);

    return firstVariable + numberToAdd;
}
```

It is called in GAMA models with:

```
species tutorialSpecies skills:[FirstSkill]
{
    init
    {
        do add(5);
```

For our action we used the annotation `@doc` to give a description of what the purpose of the action.

@doc

It provides a unified way of attaching documentation to the various GAML elements tagged by the other annotations. The documentation is automatically assembled at compile time and also used at runtime in GAML editors.

- **value** (String, "" by default): *a String representing the documentation of a GAML element.*
- **deprecated** (String, "" by default): *a String indicating (if it is not empty) that the element is deprecated and defining, if possible, what to use instead.*
- **returns** (String, "" by default): *the documentation concerning the value(s) returned by this element (if any)..*
- **comment** (String, "" by default): *an optional comment that will appear differently from the documentation itself.*
- **special_cases** (set of Strings, empty by default): *an array of String representing the documentation of the "special cases" in which the documented element takes part.*
- **examples** (set of Strings, empty by default): *an array of String representing some examples or use-cases about how to use this element.*
- **see** (set of Strings, empty by default): *an array of String representing cross-references to other elements in GAML.*

All these annotations are defined in the `GamlAnnotations.java` file of the `msi.gama.processor` plug-in.

Complete java class

```
package skills;

import msi.gama.common.interfaces.IKeyword;
import msi.gama.metamodel.agent.IAgent;
import msi.gama.precompiler.GamlAnnotations.action;
import msi.gama.precompiler.GamlAnnotations.arg;
import msi.gama.precompiler.GamlAnnotations.doc;
import msi.gama.precompiler.GamlAnnotations.getter;
import msi.gama.precompiler.GamlAnnotations.setter;
import msi.gama.precompiler.GamlAnnotations.skill;
import msi.gama.precompiler.GamlAnnotations.variable;
import msi.gama.precompiler.GamlAnnotations.vars;
import msi.gama.runtime.IScope;
import msi.gaml.skills.Skill;
import msi.gaml.types.IType;

@vars({
    @variable(name = IKeyword.FIRST_VARIABLE, type = IType.INT, init
= "1"),
    @variable(name = IKeyword.SECOND_VARIABLE, type = IType.FLOAT,
init = "1.0")
})
@skill(name = "FirstSkill")
public class FirstSkill extends Skill
{

    @getter(IKeyword.FIRST_VARIABLE)
    public int getFirstVariable(final IAgent agent) {
        return (Integer) agent.getAttribute(IKeyword.FIRST_VARIABLE);
    }

    @setter(IKeyword.FIRST_VARIABLE)
    public void setFirstVariable(final IAgent agent, final int value)
{
        agent.setAttribute(IKeyword.FIRST_VARIABLE, value);
    }
}
```

Complete gaml model

```
/**  
 * Name: FirstSkill  
 * First skill tutorial.  
 * Author: Lucas Grosjean, Julius Bangate  
 * Tags: tutorial, skill  
 */  
  
model FirstSkill  
  
global{  
    init {  
        create tutorial;  
    }  
}  
  
species tutorial skills:[FirstSkill] {  
    init {  
        do add(5);  
        int result <- add(10);  
  
        write(firstVariable);  
        write(secondVariable);  
    }  
}  
  
experiment main{}
```

Version: 1.9.1

Developing Statements

Statements are a fundamental part of GAML, as they represent both commands (imperative programming style) or declarations (declarative programming style). Developing a new statement allows, then, to add a new instruction to GAML.

Statements can be used in any context unlike action that need to be used from an agent context.

Defining the class

A new statement must be a Java class that:

- either implements the interface `IStatement` or extends an existing implementation of this interface (like `AbstractStatement` or `AbstractSequenceStatement`).
- begins by the 2 following mandatory annotations:
 - `@symbol`: `@symbol(name = "name_of_the_statement_gaml", kind = "kind_of_statement", with_sequence = true/false),`
 - `@inside`: `@symbol(kinds = {"kind_of_statement_1", "kind_of_statement_2", "..."})`

In addition the 4 following optional annotations can be added:

- `@facets`: to describe the set of `@facet` annotations,
- `@doc`: to document the statement.
- `@serializer`: in addition, statements can benefit from a custom serializer, by

declaring `@serializer(CustomSerializer.class)`, with a class extending `SymbolSerializer`.

- `@validator`: in addition, statements can benefit from a custom validation during the validation process, by declaring `@validator(CustomValidator.class)` with a class implementing `IDescriptionValidator` as value. This class will receive the `IDescription` of the statement and be able to execute further validations on the type of expressions, etc. or even to change the `IDescription` (by adding new information, changing the value of facets, etc.).

Note: GAMA annotations are classes defined into the `msi.gama.precompiler.GamlAnnotations` class.

Examples

The `write` statement

The `write` statement is an example of a `SINGLE_STATEMENT` (i.e. statement that does not embed a sequence of statements). It can be used inside a `BEHAVIOR` statement (i.e. `reflex`, `init`...), a `SEQUENCE_STATEMENT` (e.g. `loop`, `ask`, `if`...) or a `LAYER` statement. It defines a single facet ("message") mandatory and ommissible.

```
@symbol(name = IKeyword.WRITE, kind = ISymbolKind.SINGLE_STATEMENT,
with_sequence = false)
@inside(kinds = { ISymbolKind.BEHAVIOR,
ISymbolKind.SEQUENCE_STATEMENT, ISymbolKind.LAYER })
@facets(value = {
    @facet(name = IKeyword.MESSAGE, type = IType.NONE, optional =
false),
    }, ommissible = IKeyword.MESSAGE)
public class WriteStatement extends AbstractStatement {
```

The aspect statement

The `aspect` statement defines an example of BEHAVIOR statement (i.e. a statement that can be written at the same level as `init`, `reflex`...), containing a sequence of embedded statements. It can only be used inside a `species` statement (i.e. the definition of a new species) and the `global` block. It defines a single facet `name` mandatory and omissible.

```
@symbol(name = { IKeyword.ASPECT }, kind = ISymbolKind.BEHAVIOR,  
with_sequence = true, unique_name = true)  
@inside(kinds = { ISymbolKind.SPECIES, ISymbolKind.MODEL })  
@facets(value = { @facet(name = IKeyword.NAME, type = ITYPE.ID,  
optional = true)  
, omissible = IKeyword.NAME})  
public class AspectStatement extends AbstractStatementSequence {
```

The action statement

The `action` statement defines an example of ACTION statement containing a sequence of embedded statements and that can have arguments. It can be used (to define an action) in any species, experiment or global statement. It defines several facets and uses a custom validator and a custom serializer.

```
@symbol(name = IKeyword.ACTION, kind = ISymbolKind.ACTION,  
with_sequence = true, with_args = true, unique_name = true)  
@inside(kinds = { ISymbolKind.SPECIES, ISymbolKind.EXPERIMENT,  
ISymbolKind.MODEL })  
@facets(value = {  
    @facet(name = IKeyword.NAME, type = ITYPE.ID, optional = false),  
    @facet(name = IKeyword.TYPE, type = ITYPE.TYPE_ID, optional =  
true, internal = true),  
    @facet(name = IKeyword.OF, type = ITYPE.TYPE_ID, optional = true),
```

Implementation

All the statements inherit from the abstract class `AbstractStatement`. Statements with a sequence of embedded statements inherit from the class `AbstractStatementSequence` (which extends `AbstractStatement`).

The main methods of a statement class are:

- its constructor, that is executed at the compilation of the model.
- `executeOn(final IScope scope)`, it executes the statement on a given scope.
This method is executed at each call of the statement in the model,
- `privateExecuteIn(IScope scope)`: the `executeOn(final IScope scope)` method implemented in `AbstractStatement` does some verification and call the `privateExecuteIn(IScope scope)` method to perform the statement. **The execution of any statement should be redefined in this method.**

Define a SINGLE_STATEMENT statement

To define a SINGLE_STATEMENT statement that can be executed in any behavior and sequence of statements and with 2 facets, we first define a new Java class that extends `AbstractStatement` such as:

```
@symbol(name = "testStatement", kind = ISymbolKind.SINGLE_STATEMENT,  
with_sequence = false)  
@inside(kinds = { ISymbolKind.BEHAVIOR,  
ISymbolKind.SEQUENCE_STATEMENT})  
@facets(value = {  
    @facet(name = IKeyword.NAME, type = IType.NONE, optional =  
false),  
    @facet(name = "test_facet", type = IType.NONE, optional =  
true)}
```

The class should at least implement:

- a **constructor**: the constructor is called at the compilation. It is usually used to get the expressions given to the facets (using the `getFacet(String)` method) and to store it into an attribute of the class.

```
final IExpression name;

public SingleStatementExample(final IDescription desc) {
    super(desc);
    name = getFacet(IKeyword.NAME);
}
```

- the **method `privateExecuteIn`**: this method is executed each time the statement is called in the model.

```
protected Object privateExecuteIn(IScope scope) throws
GamaRuntimeException {
    IAgent agent = scope.getAgent();
    String nameStr = null;
    if (agent != null && !agent.dead()) {
        nameStr = Cast.asString(scope, name.value(scope));
        if (nameStr == null) {
            nameStr = "nil";
        }
        scope.getGui().getConsole().informConsole(nameStr,
scope.getRoot());
    }
    return nameStr;
}
```

The variable `scope` of type `IScope` can be used to:

- get the current agent with: `scope.getAgent()`

- evaluate an expression in the current scope: `Cast.asString(scope, message.value(scope))`

Define a statement with sequence

This kind of statements includes SEQUENCE_STATEMENT (e.g. `if`, `loop`,...), BEHAVIOR (e.g. `reflex`,...)...

Such a statement is defined in a class extending the `AbstractStatementSequence` class, e.g.:

```
@symbol(name = { IKeyword.REFLEX, IKeyword.INIT }, kind =
ISymbolKind.BEHAVIOR, with_sequence = true, unique_name = true)
@inside(kinds = { ISymbolKind.SPECIES, ISymbolKind.EXPERIMENT,
ISymbolKind.MODEL })
@facets(value = { @facet(name = IKeyword.WHEN, type = IType.BOOL,
optional = true),
      @facet(name = IKeyword.NAME, type = IType.ID, optional = true) },
omissible = IKeyword.NAME)
@validator(ValidNameValidator.class)

public class ReflexStatement extends AbstractStatementSequence {
```

This class should only implement a constructor. The class `AbstractStatementSequence` provides a generic implementation for:

- `privateExecuteIn(IScope scope)`: it executes each embedded statement with the scope.
- `executeOn(final IScope scope)`: it executes the statement with a given scope.

Additional methods that can be implemented

The following methods have a default implementation, but can be overridden if

necessary:

- the `String getTrace(final IScope scope)` method is called to trace the execution of statements using `trace` statement.

```
public String getTrace(final IScope scope) {  
    // We dont trace write statements  
    return "";  
}
```

- the `setChildren(final List<? extends ISymbol> commands)` is used to define which are the statement children to the sequence statement. By default, all the embedded statements are taken as children

Annotations

@symbol

This annotation represents a "statement" in GAML, and is used to define its name(s) as well as some meta-data that will be used during the validation process.

This annotation contains:

- **name** (set of string, empty by default): *names of the statement*.
- **kind** (int): *the kind of the annotated symbol (see `ISymbolKind.java` for more details)*.
- **with_scope** (boolean, true by default): *indicates if the statement (usually a sequence) defines its own scope. Otherwise, all the temporary variables defined in it are actually defined in the super-scope*.
- **with_sequence** (boolean): *indicates whether or not a sequence can or should follow the symbol denoted by this class*.
- **with_args** (boolean, false by default): *indicates whether or not the symbol denoted*

by this class will accept arguments.

- **remote_context** (boolean, false by default): *indicates that the context of this statement is actually an hybrid context: although it will be executed in a remote context, any temporary variables declared in the enclosing scopes should be passed on as if the statement was executed in the current context.*
- **doc** (set of @doc, empty by default): *the documentation attached to this symbol.*

@inside

This annotation is used in conjunction with symbol. Provides a way to tell where this symbol should be located in a model (i.e. what its parents should be). Either direct symbol names (in symbols) or generic symbol kinds can be used.

This annotation contains:

- **symbols** (set of Strings, empty by default): *symbol names of the parents.*
- **kinds** (set of int, empty by default): *generic symbol kinds of the parents (see [ISymbolKind.java](#) for more details).*

@facets

This annotation describes a list of facets used by a statement in GAML.

This annotation contains:

- **value** (set of @facet): array of @facet, each representing a facet name, type..
- **ommissible** (string): *the facet that can be safely omitted by the modeler (provided its value is the first following the keyword of the statement).*

@facet

This facet describes a facet in a list of facets.

This annotation contains:

- **name** (String): *the name of the facet. Must be unique within a symbol.*
- **type** (set of Strings): *the string values of the different types that can be taken by this facet.*
- **values** (set of Strings): *the values that can be taken by this facet. The value of the facet expression will be chosen among the values described here.*
- **optional** (boolean, false by default): *whether or not this facet is optional or mandatory.*
- **doc** (set of @doc, empty by default): *the documentation associated to the facet.*

@doc

It provides a unified way of attaching documentation to the various GAML elements tagged by the other annotations. The documentation is automatically assembled at compile time and also used at runtime in GAML editors.

- **value** (String, "" by default): *a String representing the documentation of a GAML element.*
- **deprecated** (String, "" by default): *a String indicating (if it is not empty) that the element is deprecated and defining, if possible, what to use instead.*
- **returns** (String, "" by default): *the documentation concerning the value(s) returned by this element (if any)..*
- **comment** (String, "" by default): *an optional comment that will appear differently from the documentation itself.*
- **special_cases** (set of Strings, empty by default): *an array of String representing the documentation of the "special cases" in which the documented element takes part.*
- **examples** (set of Strings, empty by default): *an array of String representing some examples or use-cases about how to use this element.*
- **see** (set of Strings, empty by default): *an array of String representing cross-*

references to other elements in GAML.

@serializer

It allows to declare a custom serializer for Symbols (statements, var declarations, species, experiments, etc.). This serializer will be called instead of the standard serializer, superseding this last one. Serializers must be subclasses of the SymbolSerializer class.

- **value** (Class): *the serializer class.*

@validator

It allows to declare a custom validator for Symbols (statements, var declarations, species, experiments, etc.). This validator, if declared on subclasses of Symbol, will be called after the standard validation is done. The validator must be a subclass of IDescriptionValidator.

- **value** (Class): *the validator class.*

All these annotations are defined in the `GamlAnnotations.java` file of the `msi.gama.processor` plug-in.

Version: 1.9.1

Developing Operators

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. Developing a new operator allows, then, to add a new function to GAML.

Implementation

A new operator can be **any Java method** that:

- begins by the `@operator` (other fields can be added to the annotation):

```
@operator(value = "name_of_the_operator_gaml")
```

```
@operator(value = "rgb")
public static GamaColor rgb(final int r, final int g, final int b,
final double alpha) {
```

The method:

- must return a value (that has to be one of the GAMA Type: Integer, Double, Boolean, String, IShape, IList, IGraph, IAgent...),
- can define any number of parameters, defined using Java type,
- can be either static or non-static:
 - in the case it is static, the number of parameters (except an IScope attribute) of the method is equal to the number of operands of the GAML operator.

- in the case it is not static, a first operand is added to the operator with the type of the current class.
- can have a IScope parameter, that will be taken into account as operand of the operator.

Annotations

@operator

This annotation represents an "operator" in GAML, and is used to define its name(s) as well as some meta-data that will be used during the validation process.

This annotation contains:

- **value** (set of string, empty by default): *names of the operator*.
- **content_type** (integer) : *if the operator returns a container, type of elements contained in the container*
- **can_be_const** (boolean, false by default): *if true: if the operands are constant, returns a constant value.*
- **category** (set of string, empty by default): *categories to which the operator belong (for documentation purpose).*
- **doc** (set of @doc, empty by default): *the documentation attached to this operator.*

@doc

It provides a unified way of attaching documentation to the various GAML elements tagged by the other annotations. The documentation is automatically assembled at compile time and also used at runtime in GAML editors.

- **value** (String, "" by default): *a String representing the documentation of a GAML*

element.

- **deprecated** (String, "" by default): *a String indicating (if it is not empty) that the element is deprecated and defining, if possible, what to use instead.*
- **returns** (String, "" by default): *the documentation concerning the value(s) returned by this element (if any)..*
- **comment** (String, "" by default): *an optional comment that will appear differently from the documentation itself.*
- **special_cases** (set of Strings, empty by default): *an array of String representing the documentation of the "special cases" in which the documented element takes part.*
- **examples** (set of Strings, empty by default): *an array of String representing some examples or use-cases about how to use this element.*
- **see** (set of Strings, empty by default): *an array of String representing cross-references to other elements in GAML.*

All these annotations are defined in the `GamlAnnotations.java` file of the `msi.gama.processor` plug-in.

Version: 1.9.1

Developing Types

GAML provides a given number of built-in simple types (int, bool...) and more complex ones (path, graph...). Developing a new type allows, then, to add a new data structure to GAML.

Implementation

Developing a new type requires the implementation of 2 Java files:

- the first one that describes the data structure (e.g.: `GamaColor.java` to define a type color)
- the second one that implements the type itself, wrapping the data structure file (e.g.: `GamaColorType.java`), and providing accessors to data structure attributes.

The data structure file

The class representing the data structure is a Java class annotated by:

- a `@vars` annotation to describe the attributes of a complex type. The `@vars` annotation contains a set of `@variable` elements.

```
@vars {  
    @variable (  
        name = "red",
```

It is recommended that this class implements the `IValue` interface. It provides a clean way to give a string representation of the type and thus eases good serialization of the object. You will need to implement the `stringValue` method:

```
public class GamaColor implements IValue {  
    @Override  
    public String stringValue(IScope scope) throws  
GamaRuntimeException {  
    ...  
}  
}
```

You should also have some class attributes that correspond to your custom data type's attributes

```
public class GamaColor implements IValue {  
    private int red;  
    private int green;  
    private int blue;  
    ...  
}
```

and then you can create setters and/or getters for each of the attributes. Setters and getters are methods annotated by the `@getter` or `@setter` annotations.

```
@getter("red")  
public Integer getRed() {  
    return this.red;  
}  
  
@Setter("red")  
public void setRed(int red) {  
    this.red = red;  
}
```

The type file

The class representing the type is a Java class such that:

- the class should be annotated by the `@type` annotation,
- the class should extend the class `GamaType<DataStructureFile>` (and thus implement its 3 methods),

Example (from `GamaFloatType.java`):

```
@type(
    name = IKeyword.FLOAT,
    id = IType.FLOAT, wraps = { Double.class, double.class },
    kind = ISymbolKind.Variable.NUMBER,
    doc = {
        @doc("Represents floating point numbers (equivalent to Double
in Java)") },
    concept = { IConcept.TYPE })
public class GamaFloatType extends GamaType<Double> {
```

Inheritance from the `GamaType<T>` class

Each java class aiming at implementing a type should inherit from the `GamaType` abstract class. Example (from `GamaColorType.java`):

```
public class GamaColorType extends GamaType<GamaColor>
```

This class imposes to implement the three following methods (with the example of the `GamaColorType`):

- `public boolean canCastToConst()`

- `public GamaColor cast(IScope scope, Object obj, Object param)`: the way to cast any object in the type,
- `public GamaColor getDefault()`: to define the default value of a variable of the current type.

Remark: for each type, a unary operator is created with the exact name of the type. It can be used to cast any expression in the given type. This operator calls the previous `cast` method.

Annotations

`@type`

It provides information necessary to the processor to identify a type.

This annotation contains:

- **name** (String, "" by default): *a String representing the type name in GAML.*
- **id** (int, 0 by default): *the unique identifier for this type. User-added types can be chosen between `IType.AVAILABLE_TYPE` and `IType.SPECIES_TYPE` (exclusive) (cf. [IType.java](#)).*
- **wraps** (tab of Class, null by default): *the list of Java Classes this type is "wrapping" (i.e. representing). The first one is the one that will be used preferentially throughout GAMA. The other ones are to ensure compatibility, in operators, with compatible Java classes (for instance, `List` and `GamaList`).*
- **kind** (int, ISymbolKind.Variable.REGULAR by default): *the kind of Variable used to store this type. See [ISymbolKind.Variable](#).*
- **internal** (boolean, false by default): *whether this type is for internal use only.*
- **doc** (set of @doc, empty by default): *the documentation associated to the facet.*

All these annotations are defined in the file [GamlAnnotations.java](#).

Version: 1.9.1

Developing Species

Additional [built-in species](#) can be defined in Java in order to be used in GAML models. Additional attributes and actions can be defined. It could be very useful in order to define its behavior thanks to external libraries (e.g. [database connection](#)...).

A new built-in species extends the `GamlAgent` class, which defines the basic GAML agents. As a consequence, new built-in species have all the attributes (`name`, `shape`, ...) and actions (`die`...) of [regular species](#).

Implementation

A new species can be **any Java class** that:

- extends the `GamlAgent` class,
- begins by the `@species: @species(name = "name_of_the_species_gaml")`,

```
@species(name = "multicriteria_analyzer")
public class MulticriteriaAnalyzer extends GamlAgent {
```

Similarly to [skills](#), a species can define additional attributes and actions.

Additional attributes

Defining new attributes needs:

- to add `@vars` (and one embedded `@var` per additional attribute) annotation on

top of the class,

- to define `@setter` and `@getter` annotations to the accessors methods.

For example, regular species are defined with the following annotation:

```
@vars({ @var(name = IKeyword.NAME, type = IType.STRING), @var(name = IKeyword.PEERS, type = IType.LIST),
    @var(name = IKeyword.HOST, type = IType.AGENT),
    @var(name = IKeyword.LOCATION, type = IType.POINT, depends_on =
IKeyword.SHAPE),
    @var(name = IKeyword.SHAPE, type = IType.GEOMETRY) })
```

And accessors are defined using:

```
@getter(IKeyword.NAME)
public abstract String getName();

@Setter(IKeyword.NAME)
public abstract void setName(String name);
```

Additional actions

An additional action is a method annotated by the `@action` annotation.

```
@action(name = ISpecies.stepActionName)
public Object _step_(final IScope scope) {
```

Annotations

@species

This annotation represents a "species" in GAML. The class annotated with this annotation will be the support of a species of agents.

This annotation contains:

- **name** (string): *the name of the species that will be created with this class as base. Must be unique throughout GAML.*
- **skills** (set of strings, empty by default): *An array of skill names that will be automatically attached to this species.* Example: `@species(value="animal" skills={"moving"})`
- **internal** (boolean, false by default): *whether this species is for internal use only.*
- **doc** (set of @doc, empty by default): *the documentation attached to this operator.*

All these annotations are defined in the `GamlAnnotations.java` file of the `msi.gama.processor` plug-in.

Version: 1.9.1

Developing architecture

In addition to existing [control architectures](#), developers can add new ones.

Defining a new control architecture needs to [create new statements of type behavior](#) and included in species statements and to define how to manage their execution.

Implementation

A control architecture is a Java class, that:

- is annotated by the `@skill` annotation,
- extends the `AbstractArchitecture` class (to get benefits of everything from the `reflex`-based control architecture, the `ReflexArchitecture` class can be extended instead).

The `AbstractArchitecture` extends the `ISkill` and `IStatement` interfaces and add the 2 following methods:

- `public abstract boolean init(IScope scope) throws GamaRuntimeException;`
- `public abstract void verifyBehaviors(ISpecies context);`

The three main methods to implement are thus:

- `public void setChildren(final List<? extends ISymbol> children)`: this method will be called at the compilation of the model. It allows to manage all the embeded statements (in `children`) and for example separate the statements

that should be executed at the initialization only from the ones that should be executed at each simulation step. Following example allows to test the name of the all the embedded statements:

```
for ( final ISymbol c : children ) {  
    if(  
        IKeyword.INIT.equals(c.getFacet(IKeyword.KEYWORD).literalValue()) ) {
```

- `public abstract boolean init(IScope scope) throws GamaRuntimeException`: this method is called only once, at the initialization of the agent.
- `public Object executeOn(final IScope scope) throws GamaRuntimeException`: this method is executed at each simulation step. It should manage the execution of the various embedded behaviors (e.g. their order or choose which one will be executed...).

Version: 1.9.1

Index of annotations

Annotations are used to link Java methods and classes to GAML language.

@action

This annotation is used to tag a method that will be considered as an action (or primitive) in GAML. The method must have the following signature: `Object methodName(IScope) throws GamaRuntimeException` and be contained in a class annotated with `@species` or `@skill` (or a related class, like a subclass or an interface).

This annotation contains:

- **name** (String): *the name of the variable as it can be used in GAML.*
- **virtual** (boolean, false by default): *if true the action is virtual, i.e. equivalent to abstract method in java.*
- **args** (set of `@arg`, empty by default): *the list of arguments passed to this action. Each argument is an instance of arg.*
- **doc** (set of `@doc`, empty by default): *the documentation associated to the action.*

@arg

This annotation describes an argument passed to an action.

This annotation contains:

- **name** (String, "" by default): *the name of the argument as it can be used in GAML.*
- **type** (set of ints, empty by default): *An array containing the textual representation of the types that can be taken by the argument (see [IType](#)).*
- **optional** (boolean, true by default): *whether this argument is optional or not.*
- **doc** (set of [@doc](#), empty by default): *the documentation associated to the argument.*

@constant

This annotation is used to annotate fields that are used as constants in GAML.

This annotation contains:

- **category** (set of Strings, empty by default): *an array of strings, each representing a category in which this constant can be classified (for documentation indexes).*
- **value** (String): *a string representing the basic keyword for the constant. Does not need to be unique throughout GAML.*
- **altNames** (set of Strings, empty by default): *an Array of strings, each representing a possible alternative name for the constant. Does not need to be unique throughout GAML.*
- **doc** (set of [@doc](#), empty by default): *the documentation attached to this constant.*

@doc

It provides a unified way of attaching documentation to the various GAML elements tagged by the other annotations. The documentation is automatically assembled at compile time and also used at runtime in GAML editors.

This annotation contains:

- **value** (String, "" by default): *a String representing the documentation of a GAML element.*
- **masterDoc** (boolean, false by default): *a boolean representing the fact that this instance of the operator is the master one, that is whether its value will subsume the value of all other instances of it.*
- **deprecated** (String, "" by default): *a String indicating (if it is not empty) that the element is deprecated and defining, if possible, what to use instead.*
- **returns** (String, "" by default): *the documentation concerning the value(s) returned by this element (if any)..*
- **comment** (String, "" by default): *an optional comment that will appear differently from the documentation itself.*
- **special_cases** (set of Strings, empty by default): *an array of String representing the documentation of the "special cases" in which the documented element takes part.*
- **examples** (set of [@example](#), empty by default): *an array of String representing some examples or use-cases about how to use this element.*
- **usages** (set of [@usage](#), empty by default): *An array of usages representing possible usage of the element in GAML.*
- **see** (set of Strings, empty by default): *an array of String representing cross-references to other elements in GAML.*

@example

This facet describes an example, that can be used either in the documentation, as unit test or as pattern.

This annotation contains:

- **value** (String, "" by default): *a String representing the expression as example.*
- **var** (String, "" by default): *The variable that will be tested in the equals, if it is omitted a default variable will be used.*

- **equals** (String, "" by default): *The value to which the value will be compared.*
- **returnType** (String, "" by default): *The type of the value that should be tested.*
- **isNot** (String, "" by default): *The value to which the value will be compared.*
- **raises** (String, "" by default): *The exception or warning that the expression could raise.*
- **isTestOnly** (boolean, false by default): *specifies that the example should not be included in the documentation.*
- **isExecutable** (boolean, true by default): *specifies that the example is correct GAML code that can be executed.*
- **test** (boolean, true by default): *specifies that the example is will be tested with the equals.*
- **isPattern** (boolean, false by default): *whether or not this example should be treated as part of a pattern (see @usage). If true, the developers might want to consider writing the example line (and its associated lines) using template variables (e.g. \${my_agent}).*

@facet

This facet describes a facet in a list of facets.

This annotation contains:

- **name** (String): *the name of the facet. Must be unique within a symbol.*
- **type** (set of int): *the string values of the different types that can be taken by this facet.*
- **values** (set of Strings, empty by default): *the values that can be taken by this facet. The value of the facet expression will be chosen among the values described here.*
- **optional** (boolean, false by default): *whether or not this facet is optional or mandatory.*

- **doc** (set of [@doc](#), empty by default): *the documentation associated to the facet.*

@facets

This annotation describes a list of facets used by a statement in GAML.

This annotation contains:

- **value** (set of [@facet](#)): array of @facet, each representing a facet name, type..
- **ommissible** (string): *the facet that can be safely omitted by the modeler (provided its value is the first following the keyword of the statement).*

@file

This annotation is used to define a type of file.

This annotation contains:

- **name** (String): *a (human-understandable) string describing this type of files, suitable for use in composed operator names (e.g. "shape", "image"...). This name will be used to generate two operators: name+_file and "is"+name. The first operator may have variants taking one or several arguments, depending on the @builder annotations present on the class_.*
- **extensions** (set of Strings): *an array of extensions (without the '.' delimiter) or an empty array if no specific extensions are associated to this type of files (e.g. ["png","jpg","jpeg"...]). The list of file extensions allowed for this type of file. These extensions will be used to check the validity of the file path, but also to generate the correct type of file when a path is passed to the generic "file" operator.*
- **buffer_content** (int, ITypeProvider.NONE by default): *the type of the content of the buffer. Can be directly a type in IType or one of the constants declared in*

ITypeProvider (in which case, the content type is searched using this provider).

- **buffer_index** (int, ITypeProvider.NONE by default): *the type of the index of the buffer. Can be directly a type in IType or one of the constants declared in ITypeProvider (in which case, the index type is searched using this provider).*
- **buffer_type** (int, ITypeProvider.NONE by default): *the type of the buffer. Can be directly a type in IType or one of the constants declared in ITypeProvider (in which case, the type is searched using this provider).*
- **doc** (set of [@doc](#), empty by default): *the documentation attached to this operator.*

@getter

This annotation is used to indicate that a method is to be used as a getter for a variable defined in the class. The variable must be defined on its own (in vars).

This annotation contains:

- **value** (String): the name of the variable for which the annotated method is to be considered as a getter.
- **initializer** (boolean, false by default): returns whether or not this getter should also be used as an initializer

@inside

This annotation is used in conjunction with [@symbol](#). It provides a way to tell where this symbol should be located in a model (i.e. what its parents should be). Either direct symbol names (in symbols) or generic symbol kinds can be used.

This annotation contains:

- **symbols** (set of Strings, empty by default): *symbol names of the parents.*

- **kinds** (set of int, empty by default): *generic symbol kinds of the parents (see [ISymbolKind.java](#) for more details).*

@operator

This annotation represents an "operator" in GAML and is used to define its name(s) as well as some meta-data that will be used during the validation process.

This annotation contains:

- **value** (set of Strings, empty by default): *names of the operator.*
- **category** (set of string, empty by default): *categories to which the operator belongs (for documentation purpose).*
- **iterator** (boolean, false by default): *true if this operator should be treated as an iterator (i.e.requires initializing the special variable "each" of WorldSkill within the method).*
- **can_be_const** (boolean, false by default): *if true: if the operands are constant, returns a constant value.*
- **content_type** (int, ITypeProvider.NONE by default): *the type of the content if the returned value is a container. Can be directly a type in IType or one of the constants declared in ITypeProvider (in which case, the content type is searched using this provider).*
- **index_type** (int, ITypeProvider.NONE by default): *the type of the index if the returned value is a container. Can be directly a type in IType or one of the constants declared in ITypeProvider (in which case, the index type is searched using this provider).*
- **expected_content_type** (set of int, empty by default): *if the argument is a container, returns the types expected for its contents. Should be an array of IType.XXX.*
- **type** (int, ITypeProvider.NONE by default): *the type of the expression if it cannot be determined at compile time (i.e. when the return type is "Object"). Can be directly a*

type in IType or one of the constants declared in ITypeProvider (in which case, the type is searched using this provider)..

- **internal** (boolean, false by default): *returns whether this operator is for internal use only.*
- **doc** (set of [@doc](#), empty by default): *the documentation attached to this operator.*

@serializer

It allows to declare a custom serializer for Symbols (statements, var declarations, species, experiments, etc.). This serializer will be called instead of the standard serializer, superseding this last one. Serializers must be subclasses of the SymbolSerializer class.

- **value** (Class): *the serializer class.*

@setter

This annotation is used to indicate that a method is to be used as a setter for a variable defined in the class. The variable must be defined on its own (in vars).

This annotation contains:

- **value** (String): *the name of the variable for which the annotated method is to be considered as a setter.*

@skill

This annotation allows to define a new skill (class grouping variables and actions that can be used by agents).

This annotation contains:

- **name** (String): *a String representing the skill name in GAML (must be unique throughout GAML).*
- **attach_to** (set of strings): *an array of species names to which the skill will be automatically added (complements the "skills" parameter of species).*
- **internal** (boolean, false by default): *return whether this skill is for internal use only.*
- **doc** (set of [@doc](#), empty by default): *the documentation associated to the skill.*

@species

This annotation represents a "species" in GAML. The class annotated with this annotation will be the support of a species of agents.

This annotation contains:

- **name** (string): *the name of the species that will be created with this class as base. Must be unique throughout GAML.*
- **skills** (set of strings, empty by default): *An array of skill names that will be automatically attached to this species. Example: `@species(value="animal" skills={"moving"})`*
- **internal** (boolean, false by default): *whether this species is for internal use only.*
- **doc** (set of [@doc](#), empty by default): *the documentation attached to this operator.*

@symbol

This annotation represents a "statement" in GAML and is used to define its name(s) as well as some meta-data that will be used during the validation process.

This annotation contains:

- **name** (set of string, empty by default): *names of the statement.*
- **kind** (int): *the kind of the annotated symbol (see [ISymbolKind.java](#) for more details).*
- **with_scope** (boolean, true by default): *indicates if the statement (usually a sequence) defines its own scope. Otherwise, all the temporary variables defined in it are actually defined in the super-scope.*
- **with_sequence** (boolean): *indicates whether or not a sequence can or should follow the symbol denoted by this class.*
- **with_args** (boolean, false by default): *indicates whether or not the symbol denoted by this class will accept arguments.*
- **remote_context** (boolean, false by default): *indicates that the context of this statement is actually a hybrid context: although it will be executed in a remote context, any temporary variables declared in the enclosing scopes should be passed on as if the statement was executed in the current context.*
- **doc** (set of [@doc](#), empty by default): *the documentation attached to this symbol.*
- **internal** (boolean, false by default): *returns whether this symbol is for internal use only.*
- **unique_in_context** (boolean, false by default): *Indicates that this statement must be unique in its super context (for example, only one return is allowed in the body of an action)..*
- **unique_name** (boolean, false by default): *Indicates that only one statement with the same name should be allowed in the same super context.*

@type

It provides information necessary to the processor to identify a type.

This annotation contains:

- **name** (String, "" by default): *a String representing the type name in GAML.*
- **id** (int, 0 by default): *the unique identifier for this type. User-added types can be*

chosen between `IType.AVAILABLE_TYPES` and `IType.SPECIES_TYPES` (exclusive) (cf. [IType.java](#)).

- **wraps** (tab of Class, null by default): *the list of Java Classes this type is "wrapping" (i.e. representing). The first one is the one that will be used preferentially throughout GAMA. The other ones are to ensure compatibility, in operators, with compatible Java classes (for instance, `List` and `GamaList`).*
- **kind** (int, `ISymbolKind.Variable.REGULAR` by default): *the kind of Variable used to store this type. See [ISymbolKind.Variable](#).*
- **internal** (boolean, false by default): *whether this type is for internal use only.*
- **doc** (set of [@doc](#), empty by default): *the documentation associated to the facet.*

@usage

This replaces `@special_cases` and `@examples`, and unifies the doc for operators, statements, and others. An `@usage` can also be used for defining a template for a GAML structure, and in that case, requires the following to be defined:

- A name (attribute "name"), optional, but better
- A description (attribute "value"), optional
- A menu name (attribute "menu"), optional
- A hierarchical path within this menu (attribute "path"), optional
- A pattern (attribute "pattern" or concatenation of the `@example` present in "examples" that define "isPattern" as true)

This annotation contains:

- **value** (String): *a String representing one usage of the keyword. Note that for usages aiming at defining templates, the description is displayed on a tooltip in the editor. The use of the path allows to remove unnecessary explanations. For instance, instead of writing: `description="This template illustrates the use of a complex form of the`*

"create" statement, which reads agents from a shape file and uses the tabular data of the file to initialize their attributes", choose: name="Create agents from shapefile" menu=STATEMENT; path={"Create", "Complex forms"} description="Read agents from a shape file and initialize their attributes". If no description is provided, GAMA will try to grab it from the context where the template is defined (in the documentation, for example).

- **menu** (String, "" by default): *Define the top-level menu where this template should appear. Users are free to use other names than the provided constants if necessary (i.e. "My templates"). When no menu is defined, GAMA tries to guess it from the context where the template is defined.*
- **path** (set of Strings, empty by default): The path indicates where to put this template in the menu. For instance, the following annotation: " menu = STATEMENT; path = {"Control", "If"} will put the template in a menu called "If", within "Control", within the top menu "Statement". When no path is defined, GAMA will try to guess it from the context where the template is defined (i.e. keyword of the statement, etc.)
- **name** (String, "" by default): *The name of the template should be both concise (as it will appear in a menu) and precise (to remove ambiguities between templates).*
- **examples** (set of [@example](#), empty by default): *An array of String representing some examples or use-cases about how to use this element, related to the particular usage above.*
- **pattern** (String, "" by default): *Alternatively, the contents of the usage can be described using a @pattern (rather than an array of @example). The formatting of this string depends entirely on the user (e.g. including \n and \t for indentation, for instance).*

@validator

It allows to declare a custom validator for Symbols (statements, var declarations, species, experiments, etc.). This validator, if declared on subclasses of Symbol, will be

called after the standard validation is done. The validator must be a subclass of `IDescriptionValidator`.

- **value** (Class): *the validator class.*

@variable

This annotation is used to describe a single variable or field.

This annotation contains:

- **name** (String): *the name of the variable as it can be used in GAML.*
- **type** (int): *The textual representation of the type of the variable (see `IType`).*
- **of** (int, 0 by default): *The textual representation of the content type of the variable (see `IType#defaultContentType()`).*
- **index** (int, 0 by default): *The textual representation of the index type of the variable (see `IType#defaultKeyType()`).*
- **constant** (boolean, false by default): *returns whether or not this variable should be considered as non modifiable.*
- **init** (String, "" by default): *the initial value of this variable as a String that will be interpreted by GAML.*
- **depend_on** (set of Strings, empty by default): *an array of String representing the names of the variables on which this variable depends (so that they are computed before).*
- **internal** (boolean, false by default): *return whether this var is for internal use only.*
- **doc** (set of [@doc](#), empty by default): *the documentation associated to the variable.*

@vars

This annotation is used to describe a set of variables or fields.

This annotation contains:

- **value** (set of @var): *an Array of var instances, each representing a variable.*

Version: 1.9.1

Introduction to GAMA Java API

This introduction to the Java API is dedicated to programmers that want to participate in the java code of GAMA. The main purpose is to describe the main packages and classes of the API to makes it simple to find such crucial information such as: how GAMA create containers, agent and geometries, how exceptions and log are managed, how java code maintain Type safety, etc.

Table of content

Concepts

1. [Factories](#)
2. [Spatial](#)
3. [Type](#)
4. [IScope](#)
5. [Exception](#)
6. [Debug](#)
7. [Test](#)

Packages

- 1.[Core](#)
-

Factories

Container factories

GAMA provides 2 factories for containers: `GamaListFactory` and `GamaMapFactory`. Each of them has `create` methods to create objects of type `IList` and `IMap`. The types of elements in the container can be specified at creation using one of the elements defined in `Types`.

Warning: the `create` method is used to create the container, with elements of a given type, **but it also converts elements added in this type**. To avoid conversion (not recommended), use the method `createWithoutCasting`.

1. `GamaListFactory` : factory to create list of different type (see [Java class](#))

As an example:

```
IList<Double> distribution = GamaListFactory.create(Types.FLOAT);
```

To create `List` object without specifying the type, use `Types.NO_TYPE`:

```
IList<Object> result = GamaListFactory.create(Types.NO_TYPE);
```

or only:

```
IList<Object> result = GamaListFactory.create();
```

2. `GamaMapFactory` : factory to create map of different type (see [Java class](#))

As an example:

```
final IMap<String, IList<?>> nCDATA =  
GamaMapFactory.create(Types.STRING, Types.LIST);
```

To create `Map` object without specifying the type, use `Types.NO_TYPE`:

```
IMap<Object, Object> result = GamaMapFactory.create(Types.NO_TYPE,  
Types.NO_TYPE);
```

or only:

```
IMap<Object, Object> result = GamaMapFactory.create();
```

If you want to use map or set, try to the best to rely on collection that ensure order, so to avoid inconsistency in container access. Try the most to avoid returning high order hash based collection, e.g. Set or Map; in this case, rely on standard definition in Gama:

3. TOrderedHashMap : see [trove api](#).
4. TLinkedHashSet : see [trove api](#)
5. Stream : you can use java build-in streams but there is a special version in Gama taken from [StreamEx](#) that should be preferred.

```
my_container.stream(my_scope)
```

If you want to get a stream back to a Gama container, you can use the collector in Factories:

```
my_container.stream(my_scope).collect(GamaListFactory.toGamaList())
```

Geometry factory

Gama geometry is based on the well established Jstор geometric library, while geographic aspect are handle using GeoTools library

1. Spatial.Creation : provide several static method to initialize geometries
- 2.

Spatial

The Spatial class provide several static access to the main methods to create, query, manipulate and transform geometries

Operators

Use as `Spatial.Operators` follow by the operator, usually one of Gaml language:

`union`, intersection, minus, and other cross geometry operations

Queries

closest, distance, overlapping, and other relative spatial relationship

Transpositions

enlarge, transpose, rotate, reduce and other specific transposition (like triangulation, squarification, etc.)

Punctal

operations relative to points

Type

`IType`: The main class to manipulate GamaType (main implementation of `IType`) is `Types`, that provides access to most common type manipulated in Gama

Opérateur de cast:

```
Types.get(IType.class)
```

IScope interface

An object of type `IScope` represents the context of execution of an agent (including experiments, simulations, and "regular" agents). Everywhere it is accessible (either passed as a parameter or available as an instance variable in some objects), it provides an easy access to a number of features: the current active agent, the shared random number generator, the global clock, the current simulation and experiment agents, the local variables declared in the current block, etc.

It also allows modifying this context, like changing values of local variables, adding new variables, although these functions should be reserved to very specific usages. Ordinarily, the scope is simply passed to core methods that allow to evaluate expressions, cast values, and so on.

Use of an IScope

A variable `scope` of type `IScope` can be used to:

- get the current agent with: `scope.getAgentScope()`

```
IAgent agent = scope.getAgentScope();
```

- evaluate an expression in the current scope:

```
String mes = Cast.asString(scope, message.value(scope));
```

- know whether the scope has been interrupted:

```
boolean b = scope.interrupted();
```

Exception

Exceptions in GAMA

An exception that can appear in the GAMA platform can be run using the `GamaRuntimeException` class. This class allows throwing an error (using `error(String, IScope)` method) or a warning (using `warning(String, IScope)` method).

In particular, it can be useful to catch the Java Exception and to throw a GAMA exception.

```
try {
    ...
} catch(Exception e) {
    throw GamaRuntimeException.error("informative message", scope);
}
```

Debug

Main class for debug is in ummisco.gama.dev.utils : [DEBUG](#)

- To turn GAMA Git version to debug mode change variable of the Debug class like: `GLOBAL_OFF = false`
- Turn on or off the debug for one class: `DEBUG.ON()` or `DEBUG.OFF()`
- You can benchmark a method call using : `DEBUG.TIME("Title to log", () -> methodToBenchmark(...))`
- You can use different built-in level to print: `DEBUG.ERR(string s)` `DEBUG.LOG(string s)` `DEBUG.OUT(Object message)`

Test

There are Gaml primitives and statement to define test:

```
test "Operator + (1)" {
    assert (circle(5) + 5).height with_precision 1 = 20.0;
    assert (circle(5) + 5).location with_precision 9 =
(circle(10)).location with_precision 9;
}
```

Everything can be made using Java Annotation (translated to Gaml test) :

```
examples = { @example (value="...",equals="...") }  
test = { "..." } // don't forget to turn test arg of examples to false
```

Core

The main plugin of the GAMA Platform that defines the core functionalities: most Gaml operators, statements, skills, types, etc.

Metamodel

`IAtom`, `IPopulation`, `IShape`, `ITopology`,

Ouputs

Util

1. Randomness in Gama: `msi.gama.util.random`

GamaRND is the main class that implements Random java class. It has several implementations and is mainly used with RandomUtils that define all the Gaml random operators

2. Graph in Gama:

3. File in Gama:

Operators

The packages where you can find all the operators defined in the core of Gama

Version: 1.9.1

Architecture of GAMA

GAMA is made of a number of Eclipse Java projects, some representing the core projects without which the platform cannot be run, others additional plugins adding functionalities or concepts to the platform.

Vocabulary: Each project is either designed as a **plugin** (containing an xml file "plugin.xml") or as a **feature** (containing an xml file "feature.xml").

- A **plugin** can be seen as a module (or bundle in the OSGI architecture), which can be necessary (the GAMA platform can't run without it) or optional (providing new functionalities to the platform). This decomposition between several plugins ensure the cohesion between functional blocks, each plugin has to be as independent as he can.
- A **feature** is a group of one or several modules (or plugin), which can be loaded. NB : Unlike a plugin, a feature does not include source code, but only two files : a build.properties and a feature.xml.

To see how to create a plugin and a feature, please read [this page](#).

Table of contents

- [Architecture of GAMA](#)
 - [The minimal configuration](#)
 - [Optional Plugins](#)
 - [Plugins present in the release version](#)
 - [Plugins not present by default in the release version](#)

- Plugins not designated to be in the release version
 - Unmaintained projects
 - Features
 - Models
 - Plugins overview

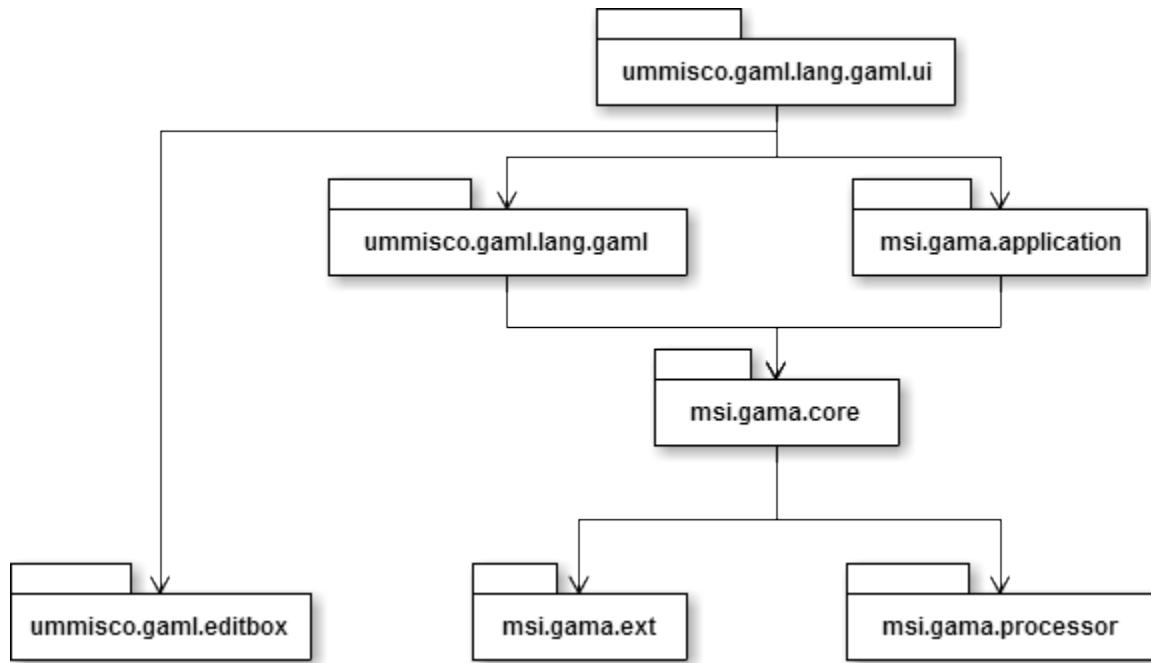
The minimal configuration

Here is the list of projects which have to be imported in order to run the GAMA platform, and to execute a simple model in gaml language:

- `msi.gama.core` : Encapsulates the core of the modeling and simulation facilities offered by the platform : runtime, simulation, meta-model, data structures, simulation kernel, scheduling, etc. It contains 2 main packages :
 - `msi.gama`
 - `msi.gaml`, which defines the GAML modeling language: keywords, operators, statements, species, skills
- `msi.gama.application` : Describes the graphical user interface (`msi.gama.gui` package). This project also contains the file `gama1.7.Eclipse3_8_2.product`, when you can configure the application (and also launch the application). It contains the following sub-packages :
 - `msi.gama.gui.displays`
 - `msi.gama.gui.navigator`
 - `msi.gama.gui.parameters`
 - `msi.gama.gui.swt`
 - `msi.gama.gui.views`
 - `msi.gama.gui.wizards`
 - `msi.gama.gui.viewers`
- `msi.gama.ext` : Gathers all the external libraries upon which GAMA relies upon

- `msi.gama.lang.gaml` : Contains the `gaml.xtext` file which defines the GAML grammar
- `msi.gama.lang.gaml.ui` : Contains the GAML Editor (syntax highlighting, code completion)
- `msi.gama.processor` : Is responsible for processing the annotations made in the Java source code and producing additions to GAML (Java, properties and documentation files), which are added into a source package called "gaml.additions" (containing two main generated files: `GamlAdditions.java` and `GamlDocumentation.java`). These additions are loaded automatically when GAMA launches, allowing extensions made by developers in other plugins to be recognized when their plugin is added to the platform.
- `ummisco.gaml.editbox` : Project used to define the edit boxes in the `gaml ui`.

Minimal configuration projects dependencies:



Optional Plugins

Plugins present in the release version

From this minimal configuration, it is possible to add some features. Here is the list of the features installed by default in the release version:

- `idees.gama.mapcomparison` : Contains some useful tools to do map comparaison
- `msi.gaml.extensions.fipa` : Provides some operators for communication between agents, using the FIPA standards
- `msi.gama.headless` : Enables to run simulations in console mode
- `simtools.gaml.extensions.traffic` : Provides operators and skills for traffic simulation
- `simtools.gaml.extensions.physics` : Physics engine, collision modelling, using the library JBullet
- `ummisco.gaml.extensions.maths` : Solving differential equation, using Euler methods and Runge Kutta.
- `irit.gaml.extensions.database` : Provides database manipulation tools, using SQL requests
- `irit.gaml.extensions.test` : Add unitary test statements
- `ummisco.gama.opengl` : Provide a 3D visualization using OpenGL.
- `simtools.gamanalyzer.fr` : Adding tools for the analysis of several execution result of a simulation (in order to find some correlations).
- `dream.gama.opengis` : Used to load some geographic information datas from online GIS server.
- `simtools.graphanalysis.fr` : Advanced graph operators

Plugins not present by default in the release version

Some other plugins are not present by default in the release version (because their use is very specific), but it's possible to install them through features. Here is the list of those plugins:

- `idees.gama.weka` : Data-mining operators, using the library Weka.
- `msi.gaml.architecture.simplebdi` : Architecture for using the Belief-Desire-Intention software model.
- `ummisco.gaml.extensions.sound` : Use of sound in simulations
- `ummisco.gaml.extensions.stats` : Advanced statistics operators
- `ummisco.gama.communicator` : Communication between several instances of GAMA
- `ummisco.gaml.extensions.rjava` : Adding the R language into GAMA for data mining

Plugins not designated to be in the release version

Other plugins will never be on the released version, and will never be loaded during the gama execution. They are just used in the "developer" version:

- `msi.gama.documentation` : Generate automatically the documentation in the wiki form (and also a pdf file)

Unmaintained projects

Some other projects are still in the git repository in case we need to work on it one day, but they are either unfinished, obsolete, or used in very rare situations (They are not delivered in release versions, of course). Here is the list:

- `cenres.gaml.extensions.hydro` : Provide some tools in order to create hydrology models
- `msi.gaml.extensions.traffic2d` : Provide some tools for traffic in 2 dimensions (deprecated, now replace by `msi.gaml.extensions.traffic`)
- `msi.gaml.extensions.humainmoving` : Provide a skill to represent human movement
- `ummisco.gama.gpu` : Computation directly on the GPU for more efficiency. Results or not conluant, slower than using CPU.
- `msi.gama.hpc` : "High Power Computing" to execute gama simulation in several computers.
- `msi.gaml.extensions.cplex` : Originaly designed to be able to run CPLEX function in GAMA. The CPLEX is a proprietary library, we can't deliver it in the project. Instead, we use a stub, "cplex.jar", that you can replace by the real cplex.jar file.
- `irit.maelia.gaml.additions` : Used for the project "Maelia". Provide the possibility to represent the computing time in a simulation.
- `msi.gama.display.web` : Originaly designed to run some GAMA simulation in a browser, inside gama application, using WebGL. Does not work for the moment
- `ummisco.miro.extension` : Once used for the "miro" project, no longer used.
- `ummisco.miro.extension.traffic` : Once used for the "miro" project, no longer used.

Features

- `ummisco.gama.feature.audio` : sound plugin
- `ummisco.feature.stats` : stats plugin
- `ummisco.gama.feature.opengl.jogl2` : gathers physics and opengl plugins
- `simtools.graphlayout.feature` : gathers core, ext, processor and graphanalysis plugins

- `ummisco.gama.feature.core` : gathers mapcomparison, database, test, application, core, ext, headless, gaml, gaml.ui, processor, fipa, traffic and maths plugins
- `ummisco.gama.feature.dependencies` : a bunch of libraries and plugins
- `other.gama.feature.plugins` gathers hydro, opengis, addition, web, hpc, cplex, traffic2d, communicator, gpu, stats, extensions and traffic plugins
- `ummisco.gama.feature.models` : model plugin
- `idees.gama.features.weka` : weka plugin
- `ummisco.gama.feature.jogl2.product` : gathering of the following features : core, dependencies, models, jogl2
- `ummisco.gama.feature.product` : gathering of the following features : core, dependencies, models, jogl1

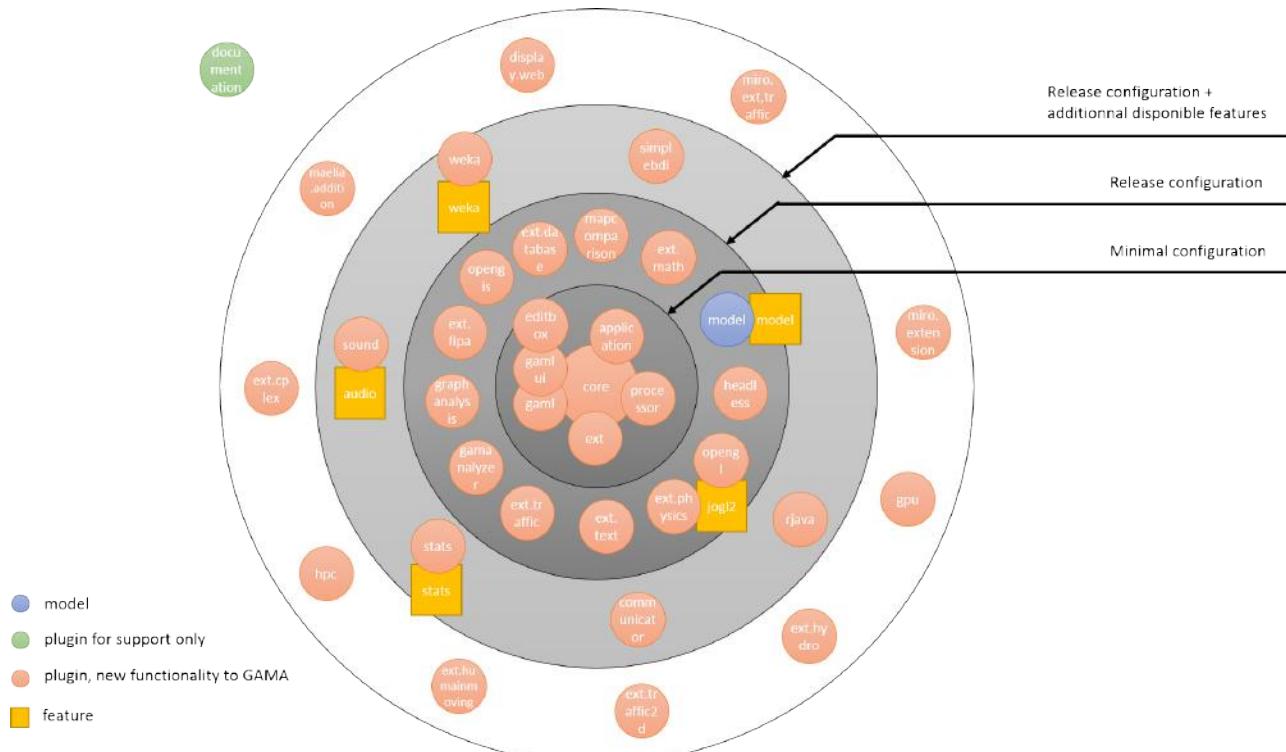
Models

Beside those plugins and features, a project dedicated to gather a bunch of examples is also in the git repository. It contains gaml code:

- `msi.gama.models`

Plugins overview

Global architecture of GAMA (nb: the features graphlayout, core, dependencies, plugins, jogl2.product and product are not represented here)



Version: 1.9.1

IScope interface

An object of type IScope represents the context of execution of an agent (including experiments, simulations, and "regular" agents). Everywhere it is accessible (either passed as a parameter or available as an instance variable in some objects), it provides an easy access to a number of features: the current active agent, the shared random number generator, the global clock, the current simulation and experiment agents, the local variables declared in the current block, etc.

It also allows modifying this context, like changing values of local variables, adding new variables, although these functions should be reserved to very specific usages. Ordinarily, the scope is simply passed to core methods that allow to evaluate expressions, cast values, and so on.

Use of an IScope

A variable `scope` of type `IScope` can be used to:

- get the current agent with: `scope.getAgentScope()`

```
IAgent agent = scope.getAgentScope();
```

- evaluate an expression in the current scope:

```
String mes = Cast.asString(scope, message.value(scope));
```

- know whether the scope has been interrupted:

```
boolean b = scope.interrupted();
```

Version: 1.9.1

Product your own release of GAMA

Install Maven if not already installed

Download the latest version of Maven here: <<https://maven.apache.org/download.cgi>>. Proceed to install it as explained on this page: <<https://maven.apache.org/install.html>>

Locate the `build.sh` shell script

It is located at the root of the `gama` Git repository on your computer. The easiest way to proceed is to select one of the GAMA projects in the Eclipse explorer and choose, in the contextual menu, `Show in > System Explorer`. Then open a shell with this path and `cd ...`. Alternatively, you can open a shell and `cd` to your Git repository and then inside `gama`.

Launch the script

Simply type `./build.sh` in your terminal and the build should begin and log its activity.

Locate the applications built by the script

They are in `ummisco.gama.product/target/products/`
`ummisco.gama.application.product` in their binary form or alternatively in
`ummisco.gama.product/target/products` in their zipped form.

Instruction for Travis build (Continuous Integration)

GAMA is built by Travis-ci.org. There are some triggers for developers to control travis:

- "ci skip": skip the build for a commit
- "ci deploy": deploy the artifacts/features to p2 server (currently to the ovh server of gama, www.gama-platform.org/updates)
- "ci clean": used with ci deploy, this trigger remove all old artifacts/features in server's p2 repository
- "ci docs": tell travis to regenerate the documentation of operators on wiki page, and update the website githubio
- "ci release": travis release zip package for OSs and place it on <https://github.com/gama-platform/gama/releases/tag/latest>
- "ci ext": The msi.gama.ext has big size, so it is not rebuilt every time, it will be compiled automatically only when it was changed, Or use this command to force travis to deploy msi.gama.ext
- "ci fullbuild": Full deploy all features/plugins

These instructions above can be used in 2 ways:

- Place them anywhere in the commit message, i.e: " fix bug #1111 ci deploy ci clean ci docs", " update readme ci skip "
- In Travis-ci, go to More Options -> Settings, add an environment variable named MSG, add the value as string, i.e.: "ci fullbuild ci deploy ci clean ci docs"

Version: 1.9.1

Generation of the documentation

Table of contents

- Requirements
 - Configuration
 - Generated files location
- Workflow to generate wiki files
- Workflow to generate PDF files
- Workflow to generate unit tests
- Main internal steps
 - Generate wiki files
 - Generate pdf files
 - Generate unit test files
- How to document
 - The @doc annotation
 - the @example annotation
 - How to document operators
 - How to document statements
 - How to document skills
- How to change the processor
- General workflow of file generation

The GAMA documentation comes in 2 formats: a set of wiki files available from the wiki section of the GitHub website and a PDF file. The PDF file is produced from the wiki files.

In the wiki files, some are hand-written by the GAMA community and some others are generated automatically from the Java code and the associated java annotations.

The section summarizes:

- how to generate this wiki files,
- how to generate the PDF documentation,
- how to generate the unit tests from the java annotations,
- how to add documentation in the java code.

Requirements

To generate automatically the documentation, the GAMA Git version is required. See [Install Git version](#) for more details.

Among all the GAMA plugins, the following ones are related to documentation generation:

- `msi.gama.processor`: the java preprocessor is called during java compilation of the various plugins and extract information from the java code and the java annotations. For each plugin it produces the `docGAMA.xml` file in the `gaml` directory.
- `msi.gama.documentation`: it contains all the java classes needed to gather all the `docGAMA.xml` files and generate wiki, pdf or unit test files.

In addition, the folder containing the wiki files is required. In the GitHub architecture, the wiki documentation is stored in a separate Git repository <https://github.com/>

`gama-platform/gama.wiki.git`. A local clone of this repository should thus be created:

1. Open the Git perspective:

- Windows > Open Perspective > Other...
- Choose `Git`

2. Click on "Clone a Git repository"

• In **Source Git repository** window:

- Fill in the URI label with: `https://github.com/gama-platform/gama.wiki.git`
- Other fields will be automatically filled in.

• In **Branch Selection** windows,

- check the master branch
- Next

• In **Local Destination** windows,

- Choose the directory in which the gama Git repository has been cloned
- Everything else should be unchecked
- Finish

3. In the **Git perspective** and the **Git Repositories** view, Right-Click on "Working Directory" inside the `gama.wiki` repository, and choose "Import projects"

• In the **Select a wizard to use for importing projects** window:

- "Import existing projects" should be checked
- "Working Directory" should be selected

• In **Import Projects** window:

- **Uncheck "Search for nested project"**
- Check the project `gama.wiki`
- Finish

2. Go back to the Java perspective: a `gama.wiki` plugin should have been added.

In order to generate the PDF file from the wiki files, we use an external application named **Pandoc**. Follow the [Pandoc installation instructions to install it](#). Specify the path to the pandoc folder in the file "Constants.java", in the static constant `CMD_PANDOC` : "*yourAbsolutePathToPandoc/pandoc*".

Note that Latex should be installed in order to be able to generate PDF files. Make sure you have already installed **Miktex** (for OS Windows and Mac). Specify the path to the miktex folder in the file "Constants.java", in the static constant `CMD_PDFLATEX` : "*yourAbsolutePathToMiktex/pdflatex*".

Configuration

The location where the files are generated (and other constants used by the generator) are defined in the file `msi.gama.documentation/src/msi/gama/doc/util/Constants.java`.

The use of Pandoc (path to the application and so on) is defined in the file `msi.gama.documentation/src/msi/gama/doc/util/ConvertToPDF.java`. *This should be changed in the future...*

Generated files location

The generated files are (by default) generated in various locations depending on their type:

- wiki files: they are generated in the plugin `gama.wiki`.
- pdf file: they are generated in the plugin `msi.gama.documentation`, in the folder `files/gen/pdf`.
- unit test files: they are generated in the plugin `msi.gama.models`, in the folder `models/Tests`.

Workflow to generate wiki files

The typical workflow to generate the wiki files is as follow:

- Clean and Build all the GAMA projects,
- Run the `MainGenerateWiki.java` file in the `msi.gama.documentation`,
- The wiki files are generated in the `gama.wiki` plugin.

Workflow to generate PDF files

The typical workflow to generate the wiki files is as follow:

- Clean and Build all the GAMA projects,
- In the file `mytemplate.tex`, specify the absolute path to your "gama_style.tex" (it should be just next to this file)
- Run the `MainGeneratePDF.java` file in the `msi.gama.documentation`, accepting all the packages install of latex,
- The wiki files are generated in the `msi.gama.documentation` plugin.

Note that generating the PDF takes a lot of time. Please be patient!

If you want to update the file "gama_style.sty" (for syntax coloration), you have to turn the flag "generateGamaStyle" to "true" (and make sure the file "keywords.xml" is already generated).

Workflow to generate unit tests

The typical workflow to generate the wiki files is as follow:

- Clean and Build all the GAMA projects,
- Run the `MainGenerateUnitTest.java` file in the `msi.gama.documentation`,
- The wiki files are generated in the `msi.gama.models` plugin.

Main internal steps

- Clean and Build all the GAMA projects will create a `docGAMA.xml` file in the `gaml` directory of each plugin,
- The `MainGenerateXXX.java` files then perform the following preparatory tasks:
 - they *prepare the gen folder* by deleting the existing folders and create all the folders that may contain intermediary generated folders
 - they merge all the `docGAMA.xml` files in a `docGAMAglobal.xml` file, created in the `files/gen/java2xml` folder. **Only the plugins that are referred in the product files are merged.**

After these common main first steps, each generator (wiki, pdf or unit test) performs specific tasks.

Generate wiki files

- The `docGAMAglobal.xml` is parsed in order to generate 1 wiki file per kind of keyword:
 - operators,
 - statements,
 - skills,
 - architectures,
 - built-in species,
 - constants and units.
 - in addition an index wiki file containing all the GAML keywords is generated.

- One wiki file is generated for each *extension* plugin, i.e. plugin existing in the Eclipse workspace but not referred in the product.

Generate pdf files

The pdf generator uses the table of content (toc) file located in the `files/input/toc` folder (`msi.gama.documentation` plugin) to organize the wiki files in a pdf file.

- `MainGeneratePDF.java` file parses the toc file and create the associated PDF file using the wiki files associated to each element of the toc. The generation is tuned using files located in the `files/input/pandocPDF` folder.

Generate unit test files

- `MainGenerateUnitTest.java` creates GAMA model files for each kind of keyword from the `docGAMAglobal.xml` file.

How to document

The documentation is generated from the Java code thanks to the Java additional processor, using mainly information from Java classes or methods and from the Java annotations. (see [the list of all annotations](#) for more details about annotations).

The `@doc` annotation

Most of the annotations can contain a `@doc` annotation, that can contain the main part of the documentation.

For example, the `inter` ([inter](#)) operator is commented using:

```

@doc(
    value = "the intersection of the two operands",
    comment = "both containers are transformed into sets (so without
duplicated element, cf. remove_duplicates operator) before the set
intersection is computed.",
    usages = {
        @usage(value = "if an operand is a graph, it will be transformed
into the set of its nodes"),
        @usage(value = "if an operand is a map, it will be transformed
into the set of its values", examples = {
            @example(value = "[1::2, 3::4, 5::6] inter [2,4]", equals =
"[2,4"]),
            @example(value = "[1::2, 3::4, 5::6] inter [1,3]", equals =
"[]") }),
        @usage(value = "if an operand is a matrix, it will be transformed
into the set of the lines", examples =
            @example(value = "matrix([[1,2,3],[4,5,4]]) inter [3,4]", equals
= "[3,4"]) },
        examples = {
            @example(value = "[1,2,3,4,5,6] inter [2,4]", equals = "[2,4"]),
            @example(value = "[1,2,3,4,5,6] inter [0,8]", equals = "[]") },
        see = { "remove_duplicates" })
)

```

This `@doc` annotation contains 5 parts:

- `value`: describes the documented element,
- `comment`: a general comment about the documented element,
- `usages`: a set of ways to use the documented element, each of them being in a `@usage` annotation. The usage contains mainly a description and set of examples,
- `examples`: a set of examples that are not related to a particular usage,
- `see`: other related keywords.

the `@example` annotation

This annotation contains a particular use example of the documented element. It is also used to generate unit test and patterns.

The simplest way to use it:

```
@example(value = "[1::2, 3::4, 5::6] inter [2,4]", equals = "[2,4]")
```

In this example:

- `value` contains an example of use of the operator,
- `equals` contains the expected results of expression in `value`.

This will become in the documentation:

```
list var3 <- [1::2, 3::4, 5::6] inter [2,4];      // var3 equals [2,4]
```

When no variable is given in the annotation, an automatic name is generated. The type of the variable is determined thanks to the return type of the operator with these parameters.

This example can also generate a unit test model. In this case, the value in the variable will be compared to the `equals` part.

By default, the `@example` annotation has the following default values:

- `isTestOnly` = `false`, meaning that the example will be added to the documentation too,
- `isExecutable` = `true`, meaning that content of `value` can be added in a model

and can be compiled (it can be useful to switch it to false, in a documentation example containing name of species that have not been defined),

- `test = true`, meaning that the content of value will be tested to the content of equals,
- `isPattern = false`.

How to document operators

A GAML operator is defined by a Java method annotated by the `@operator` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the operators are defined in the plugin `msi.gama.core` and in the package `msi.gaml.operators`.

The documentation generator will use information from:

- the `@operator` annotation:
 - `value`: it provides the name(s) of the operator (if an operator has several names, the other names will be considered as alternative names)
 - `category`: it is used to classify the operators in categories
- the `@doc` annotation,
- the method definition:
 - the return value type
 - parameters and their type (if the method is static, the IScope attribute is not taken into account)

How to document statements

A GAML statement is defined by a Java class annotated by the `@symbol` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the statements are defined in the plugin `msi.gama.core` and in the package `msi.gaml.statements`.

The documentation generator will use information from:

- `@symbol` annotation,
- `@facets` annotation (each facet can contain a documentation in a `@doc` annotation),
- `@inside` annotation (where the statement can be used),
- `@doc` annotation

How to document skills

A GAML skill is defined by a Java class annotated by the `@skill` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the skills are defined in the plugin `msi.gama.core` and in the package `msi.gaml.skills`.

The documentation generator will use information from:

- `@skill` annotation,
- `@vars` annotation (each var can contain a documentation in a `@doc` annotation),
- `@doc` annotation

How to change the processor

If you make some modifications in the plugin processor, you have to rebuild the .jar file associated to the processor to take into account the changes. Here are the several steps you have to do:

- In the `msi.gama.processor` plugin, click on `Generate Processor.jardesc` (in `processor`)
- Click on Finish (you can check that `msi.gama.processor` and

`ummisco.gama.annotations` are checked). Accept the warning popup.

- It should have changed the `processor / plugins / msi.gama.processor_1.4.0.jar` file.
- Right-click on the folder `processor` to refresh.

In case some projects have errors after the update of the processor:

- Clean and build the projects
- Close Eclipse and reopen it and clean and build the projects
- Check that Eclipse has been launched with the same JVM as GAMA. To this purpose, have a look at `Eclipse / About Eclipse`, `Installation details` and check the java version (i.e. after the `-vm` option). If it does not fit with the one used for eclipse plugin, change it (in the `eclipse.ini` file).

This following diagram explains roughly the workflow for the generation of the different files:

