GAMA v1.8.0 documentation

by GAMA team

http://gama-platform.org

Contents

Part I

Home

Chapter 1

GAMA

GAMA is a modeling and simulation development environment for building spatially explicit agent-based simulations.

- **Multiple application domains:** Use GAMA for whatever application domain you want.
- **High-level and Intuitive Agent-based language:** Write your models easily using GAML, a high-level and intuitive agent-based language.
- **GIS and Data-Driven models:** Instantiate agents from any dataset, including GIS data, and execute large-scale simulations (up to millions of agents).
- **Declarative user interface:** Declare interfaces supporting deep inspections on agents, user-controlled action panels, multi-layer 2D/3D displays & agent aspects.

Its latest version, **1.8**, can be freely downloaded or built from source, and comes pre-loaded with several models, tutorials and a complete on-line documentation.

Multiple application domains

GAMA has been developed with a very general approach and can be used for many application domains. Some additional plugins had been developed to fit with particular needs.

Example of application domains where GAMA is mostly present:

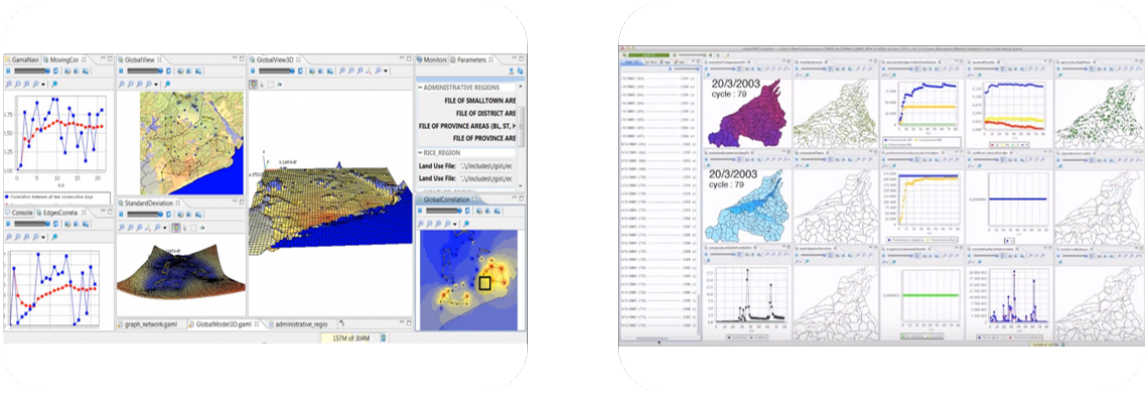


Figure 1.1: Multiple application domains

- Transport
- Urban planning
- Epidemiology
- Environment

Training sessions

Some training sessions about topics such as “urban management”, “epidemiology”, “risk management” are also provided by the team. Since GAMA is an open-source software that continues to grow, if you have any particular needs for improvement, feel free to share it to its active community!

High-level and intuitive agent-based language

Thanks to its high-level and intuitive language, GAMA has been developed to be used by non-computer scientists. You can declare your species, giving them some special behaviors, create them in your world, and display them in less than 10 minutes.

GAML is the language used in GAMA, coded in Java. It is an agent-based language, that provides you the possibility to build your model with several paradigms of modeling. Once your model is ready, some features allow you to explore and calibrate it, using the parameters you defined as input of your simulation.

Figure 1.2: High level language

We provide you a continual support through the active mailing list where the team will answer your questions. Besides, you can learn GAML on your own, following the step by step tutorial, or personal learning path in order reach the point you are interested in.

GIS and Data-Driven models

GAMA (GIS Agent-based Modeling Architecture) provides you, since its creation, the possibility to load easily GIS (Geographic Information System).

You can import a large number of data types, such as text, files, CSV, shapefile, OSM (open street map data), grid, images, SVG, but also 3D files, such as 3DS or OBJ, with their texture.

Some advanced features provide you the possibility to connect GAMA to databases, and also to use powerful statistical tools such as R.

GAMA has been used in large-scale projects, using a great number of agents (up to millions of agents).

Declarative user interface

GAMA provides you the possibility to have multiple displays for the same model. You can add as many visual representations as you want for the same model, in order

Figure 1.3: Data-driven models

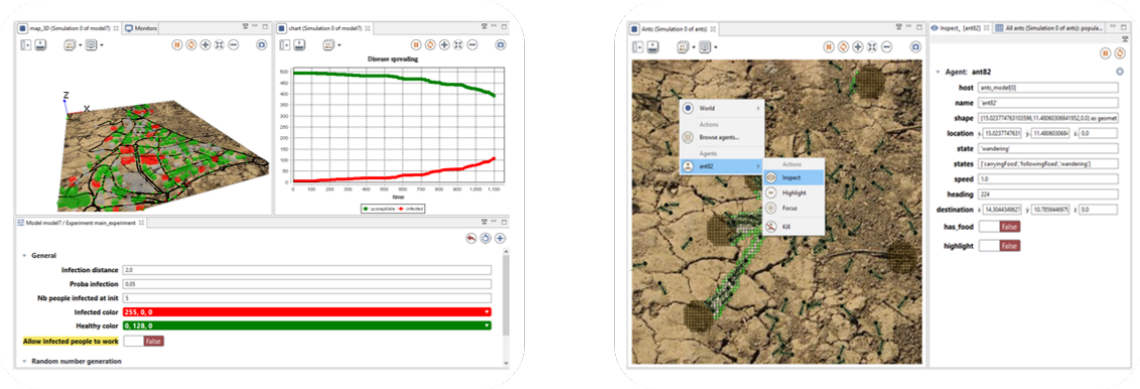


Figure 1.4: Declarative User Interface

to highlight a certain aspect of your simulation. Add easily new visual aspects to your agents.

Advanced 3D displays are provided: you can control lights, cameras, and also adding textures to your 3D objects. On the other hand, dedicated statements allow you to define easily charts, such as series, histogram, or pies.

During the simulations, some advanced features are available to inspect the population of your agents. To make your model more interactive, you can add easily some user-controlled action panels, or mouse events.

Development Team

GAMA is developed by several teams under the umbrella of the IRD/SU international research unit UMMISCO:

- UMI 209 UMMISCO, IRD, 32 Avenue Henri Varagnat, 93143 Bondy Cedex, France.
- DREAM Research Team, University of Can Tho, Vietnam (2011 - 2019).
- UMR 5505 IRIT, CNRS/University of Toulouse 1, France (2010 - 2019).
- UR MIAT, INRA, 24 Chemin de Borde Rouge, 31326 Castanet Tolosan Cedex, France (2016 - 2019).
- UMR 6228 IDEES, CNRS/University of Rouen, France (2010 - 2019).
- UMR 8623 LRI, CNRS/University Paris-Sud, France (2011 - 2019).
- MSI Research Team, Vietnam National University, Hanoi, Vietnam (2007 - 2015).

Citing GAMA

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

> Taillandier, P., Gaudou, B., Grignard, A.,Huynh, Q.-N., Marilleau, N., P. Caillou, P., Philippon, D., & Drogoul, A. (2019). Building, composing and experimenting complex spatial models with the GAMA platform. Geoinformatica, (2019), 23 (2), pp. 299-322, [doi:10.1007/s10707-018-00339-6]

or you can choose to cite the website instead:

> GAMA Platform website, http://gama-platform.org

A complete list of references (papers and PhD theses on or using GAMA) is available on the references page.

Figure 1.5: YourKit logo

Acknowledgement

YourKit supports open source projects with its full-featured Java Profiler. YourKit, LLC is the creator of YourKit Java Profiler and YourKit .NET Profiler, innovative and intelligent tools for profiling Java and .NET applications.

Part II

Introduction

Chapter 2

Introduction

GAMA is a simulation platform, which aims at providing field experts, modellers, and computer scientists with a complete modelling and simulation development

environment for building spatially explicit multi-agent simulations. It has been first developed by the Vietnamese-French research team MSI (located at IFI, Hanoi, and part of the IRD/SU International Research Unit UMMISCO) from 2007 to 2010, and is now developed by a consortium of academic and industrial partners led by UMMISCO, among which the University of Rouen, France, the University of Toulouse 1, France, the University of Orsay, France, the University of Can Tho, Vietnam, the National University of Hanoi, EDF R&D, France, and CEA LISC, France.

Some of the features of GAMA are illustrated in the videos above (more can be found in our Youtube channel).

Beyond these features, GAMA also offers:

- A complete modeling language, GAML, for modeling agents and environments
- A large and extensible library of primitives (agent's movement, communication, mathematical functions, graphical features, ...)

- A cross-platform reproducibility of experiments and simulations
- A powerful declarative drawing and plotting subsystem
- A flexible user interface based on the Eclipse platform
- A complete set of batch tools, allowing for a systematic or "intelligent" exploration of models parameters spaces

Documentation

The documentation of GAMA is available online on the wiki of the project. It is organized around a few central activities (installing GAMA, writing models, running experiments, developing new extensions to the platform) and provides complete references on both the GAML language, the platform itself, and the scientific aspects of our work (with a complete bibliography). Several tutorials are also provided in the documentation in order to minimize the learning curve, allowing users to build, step by step, the models corresponding to these tutorials, which are of course shipped with the platform.

The documentation can be accessed from the sidebar of this page. A good starting point for new users is the installation page.

A standalone version of the documentation, in PDF format, can be directly downloaded here

Source Code

GAMA can be downloaded as a regular application or built from source, which is necessary if you want to contribute to the platform. The source code is available from this GITHub repository:

```
https://github.com/gama-platform/gama
```

Which you can also browse from the web here. It is, in any case, recommended to follow the instructions on this page in order to build GAMA from source.

Copyright Information

Developers

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

- **Alexis Drogoul**, Senior Researcher at the IRD, member of the UMMISCO International Research Unit. Mostly working on agent-based modeling and simulation. Has contributed and still contributes to the original design of the platform, including the GAML language (from the meta-model to the editor) and simulation facilities like Java2D displays.
- **Patrick Taillandier**, Researcher at INRA, member of the MIAT Research Unit. Contributes since 2008 to the spatial and graph features (GIS integration, spatial operators). Currently working on new features related to graphical modeling, BDI agent architecture, and traffic simulation.
- **Benoit Gaudou**, Associate Professor at the University Toulouse 1 Capitole, member of the IRIT CNRS Mixed Research Unit. Contributes since 2010 to documentation and unit test generation and coupling mathematical (ODE and PDE) and agent paradigms.
- **Arnaud Grignard**, Research Scientist at MIT MediaLab, member of the CityScience Group, software engineer and PhD fellow (PDI-MSC) at SU. Contributes since 2011 to the development of new features related to visualization, interaction, online analysis and tangible interfaces.
- **Huynh Quang Nghi**, software engineering lecturer at CTU and PhD fellow (PDI-MSC) at SU. Contributes since 2012 to the development of new features related to GAML parser, coupling formalisms in EBM-ABM and ABM-ABM.
- **Truong Minh Thai**, software engineering lecturer at CTU and PhD fellow (PRJ322-MOET) at IRIT-UT1. Contributes since 2012 to the development of new features related to data management and analysis.

- **Nicolas Marilleau**, Researcher at the IRD, member of the UMMISCO International Research Unit and associate researcher at DISC team of FEMTO-ST institute. Contributes since 2010 to the development of headless mode and the high performance computing module.
- **Philippe Caillou**, Associate professor at the University Paris Sud 11, member of the LRI and INRIA project-team TAO. Contributes since 2012 and actually working on charts, simulation analysis and BDI agents.
- **Vo Duc An**, Post-doctoral Researcher, working on synthetic population generation in agent-based modelling, at the UMMISCO International Research Unit of the IRD. Has contributed to bringing the platform to the Eclipse RCP environment and to the development of several features (e.g., the FIPA-compliant agent communication capability, the multi-level architecture).
- **Truong Xuan Viet**, software engineering lecturer at CTU and PhD fellow (PDI-MSC) at SU. Contributes since 2011 to the development of new features related to R caller, online GIS (OPENGIS: Web Map Service - WMS, Web Feature Services - WMS, Google map, etc).
- Samuel Thiriot
- **Jean-Daniel.Zucker**, Senior Researcher at the IRD, member and director of the UMMISCO International Research Unit. Mostly working on Machine Learning and also optimization using agent-based modeling and simulation. Has contributed to different models and advised different students on GAMA since its beginning.

Citing GAMA

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

> Taillandier, P., Gaudou, B., Grignard, A., Huynh, Q.N., Marilleau, N., Caillou, P., Philippon, D., Drogoul, A. (2018), Building, composing and experimenting complex spatial models with the GAMA platform. In Geoinformatica, Springer, https://doi.org/10.1007/s10707-018-00339-6.

or you can choose to cite the website instead:

> GAMA Platform website, http://gama-platform.org

A complete list of references (papers and PhD theses on or using GAMA) is available on the references page.

Contact Us

To get in touch with the GAMA developers team, please sign in for the gama-platform@googlegroups.com mailing list. If you wish to contribute to the platform, you might want, instead or in addition, to sign in for the gama-dev@googlegroups.com mailing list. On both lists, we generally answer quite quickly to requests.

Finally, to report bugs in GAMA or ask for a new feature, please refer to these instructions to do so.

Part III

Changes from 1.6.1 to 1.8

Chapter 3

Changes from 1.6.1 to 1.8

Java version

Due to changes in the libraries used by GAMA 1.7 and 1.8, this version now **requires JDK/JVM 1.8** to run. Please note that GAMA **has not been tested with JDK 1.9 and 1.10**.

Changes between 1.6.1 and 1.7/1.8 that can influence the dynamics of models

- Initialization order between the initialization of variables and the execution of the `init` block in grids init -> vars in 1.6.1 / vars -> init in 1.7
- Initialization order of agents -> now, the init block of the agents are not executed at the end of the global init, but during it. put a sample model to explain the order of creation and its differences
- Initialization of vars to their default value map ? list ?
- Systematic casting and verification of types give examples
- header of CSV files: be careful, in GAMA 1.7, if the first line is detected as a header, it is not read when the file is casted as a matrix (so the first row of the matrix is not the header, but the first line of data) gives examples
- the step of batch experiments is now executed after all repetitions of simulations are done (not after each one). They can however be still accessed using the attributes `simulations` (see Batch.gaml in Models Library)

- signal and diffuse have been merged into a single statement
- facets do not accept a space between their identifier and the `:` anymore.
- simplification of equation/solve statements and deprecation of old facets
- in FIPA skill, `content` is replaced everywhere with `contents`
- in FIPA skill, `receivers` is replaced everywhere with **`to`**
- in FIPA skill, `messages` is replaced by `mailbox`
- The pseudo-attribute `user_location` has been removed (not deprecated, unfortunately) and replaced by the "unit" `#user_location`.
- The actions called by an `event` layer do not need anymore to define **`point`** and **`list`**<**`agent`**> arguments to receive the mouse location and the list of agents selected. Instead, they can now use `#user_location` and they have to compute the selected agents by themselves (using an arbitrary function).
- The random number generators now better handle seeding (larger range), but it can change the series of values previously obtained from a given seed in 1.6.1
- all models now have a starting_date and a current_date. They then don't begin at a hypothetical "zero" date, but at the epoch date defined by ISO 8601 (1970/1/1). It should not change models that don't rely on dates, except that:
- the `#year` (and its nicknames `#y`, `#years`) and `#month` (and its nickname `#month`) do not longer have a default value (of resp. 30 days and 360 days). Instead, they are always evaluated against the current_date of the model. If no starting_date is defined, the values of `#month` and `#year` will then depend on the sequence of months and year since epoch day.
- `as_time`, `as_system_time`, `as_date` and `as_system_date` have been removed

Chapter 4

Enhancements in 1.7/1.8

Simulations

- simulations can now be run in parallel withing an experiment (with their outputs, displays, etc.)
- batch experiments inherit from this possibility and can now run their repetitions in parallel too.
- concurrency between agents is now possible and can be controlled on a species/-grid/ask basis (from multi-threaded concurrency to complete parallelism within a species/grid or between the targets of an `ask` statement)

Language

- `gama` : a new immutable agent that can be invoked to change preferences or access to platform-only properties (like `machine-time`)
- `abort`: a new behavior (like `reflex` or **`init`**) that is executed once when the agent is about to die
- `try` and `catch` statements now provide a robust way to catch errors happening in the simulations.
- `super` (instead of `self`) and `invoke` (instead of `do`) can now be used to call an action defined in a parent species.
- **`date`** : new data type that offers the possibility to use a real calendar, to define a `starting_date` and to query a `current_date` from a simulation, to parse

dates from date files or to output them in custom formats. Dates can be added, subtracted, compared. Various new operators (**minus_months**, etc.) allow for a fine manipulation of their data. Time units (`#sec`, `#s`, `#mn`, `#minute`, `#h`, `#hour`, `#day`, etc.) can be used in conjunction with them. Interval of dates (date1 to date2) can be created and used as a basis for loops, etc. Various simple operators allow for defining conditions based on the current_date (after(date1), before(date2), since(date1), etc.).

- **font** type allows to define fonts more precisely in `draw` statements
- BDI control architecture for agents
- file management, new operators, new statements, new skills(?), new built-in variables, files can now download their contents from the web by using standard http: https: addresses instead of file paths.
- The `save` can now directly manipulate files and ... save them. So something like `save shape_file("bb.shp", my_agents collect each.shape);` is possible. In addition, a new facet **attributes** allows to define complex attributes to be saved.
- `assert` has a simpler syntax and can be used in any behaviour to raise an error if a condition is not met.
- `test` is a new type of experiments (`experiment aaa type: test ...`), equivalent to a `batch` with an exhaustive search method, which automatically displays the status of tests found in the model.
- new operators (**sum_of**, **product_of**, etc.)
- casting of files works
- co-modeling (importation of micro-models that can be managed within a macro-model)
- populations of agents can now be easily exported to CSV files using the `save` statement
- Simple `messaging` skill between agents

- Terminal commands can now be issued from within GAMA using the `console` operator
- New `status` statement allows to change the text of the status.
- light statement in 3D display provides the possibility to custom your lights (point lights, direction lights, spot lights)
- Displays can now inherit from other displays (facets **parent** and **virtual** to describe abstract displays)
- **on_change:** facet for attributes/parameters allows to define a sequence of statements to run whenever the value changes.

- species and experiment now support the **virtual** boolean facet (virtual species can not be instantiated, and virtual experiments do not show up).
- experiment now supports the auto_run boolean facet (to run automatically when launched)
- experiment now supports the benchmark boolean facet (to produce a CSV summary of the time spent in the different statements / operators of GAMA)
- experiments can now have their own file (xxx.experiment) and specify the model they are targeting by providing the path to the model in the new model: facet (similar to import).
- experiments can sport a new type: test, a specialised type of batch experiment that can be run automatically from the GUI or in headless and reports back the result of the tests found in its model

Data import

- draw of complex shapes through obj file
- new types fo files are taken into account: geotiff and dxf
- viewers for common files
- addition of plugin and test models

Navigator

- Shapefiles are now copied, pasted and deleted together with their support files
- External files are automatically linked from the workspace and the links are filed under an automatically created external folder in the project
- The "Refresh" command in the navigator pop-up refreshes the files, cleans the metadata and recompiles the models in order to obtain a "fresh" workspace again
- A search control allows to instantaneously find models based on their names (not contents)
- Wizards for creating .experiment file and test experiments
- The new project Wizard now leads by default to the new file wizard

Editor

- doc on built-in elements, templates, shortcuts to common tasks, hyperlinks to files used
- improvement in time, gathering of infos/todos
- warnings can be removed from model files
- resources / files can be dropped into editors to obtain declaration/import of the corresponding files

Headless

- A new option `-validate path/to/dir` allows to run a complete validation of all the models in the directory
- A new option `-test path/to/dir` allows to run all the tests defined in a directory

Models library

- New models (make a list)

Preferences

- For performances and bug fixes in displays
- For charts defaults

Simulation displays

- OpenGL displays should be up to 3 times faster in rendering
- fullscreen mode for displays (ESC key)
- CTRL+O for overlay and CTRL+L for layers side controls
- cleaner OpenGL displays (less garbage, better drawing of lines, rotation helper, sticky ROI, etc.)

- possibility to use a new OpenGl pipeline and to define keystoning parameters (for projections)
- faster java2D displays (esp. on zoom)
- better user interaction (mouse move, hover, key listener)
- a whole new set of charts
- getting values when moving the mouse on charts
- possibility to declare `permanent layout: + #splitted, #horizontal, #vertical, #stacked` in the `output` section to automatically layout the display view.
- Outputs can now be managed from the "Views" menu. Closed outputs can be reopened.
- Changing simulation names is reflected in their display titles (and it can be dynamic)
- OpenGL displays now handle rotations of 2D and 3D shapes, combinations of textures and colours, and keystoning

Error view

- Much faster (up to 100x !) display of errors
- Contextual menu to copy the text of errors to clipboard or open the editor on it

Validation

- Faster validation of multi-file models (x2 approx.)
- Much less memory used compared to 1.6.1 (/10 approx.)
- No more "false positive" errors

Console

- Interactive console allows to directly interact with agents (experiments, simulations and any agent) and get a direct feedback on the impact of code execution using a new interpreter integrated with the console. Available in the modeling perspective (to interact with the new `gama` agent) as well as the simulation perspective (to interact with the current `experiment` agent).

- Console now accepts colored text output

Monitor view

- monitors can have colors
- monitors now have contextual menus depending on the value displayed (save as CSV, inspect, browse...)

GAMA-wide online help on the language

- A global search engine is now available in the top-right corner of the GAMA window to look for GAML idioms

Serialization

- Serialize simulations and replay them
- Serialization and deserialization of agents between simulations

Allow TCP, UDP and MQTT communications between agents in different simulations

Part IV

Moving to 1.9

Chapter 5

Goals of GAMA 1.9 (see branch here)

- Get completely rid of GraphStream in favour of JGraphT for reading/writing graphs. Replace all odd reading operators by graph files for reading/writing
- Move to GeoTools 21.1, JTS 1.16.1, JGraphT 1.3.0, StreamEx 0.6.8
- Simplify the syntax by removing some alternatives (arguments passing to actions, for instance).

Part V

Platform

Chapter 6

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.

Chapter 7

Installation and Launching

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

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

Chapter 8

Installation

We made efforts to make the last release of GAMA (1.8.0) as easy as possible to install, by providing a version with an embedded Java JDK, limiting the installation to a 3-steps procedure: download, extract and launch.

Table of contents

Download GAMA

GAMA 1.8.0 (the last release) comes in 6 different versions:

- 2 versions for each of the 3 environments (by default in 64 bits) Windows, MacOS X and Linux (tested mainly on Ubuntu),
- For each OS, one version includes the Java JDK (1.8.0_161 in 64 bits) and one does not.

It is important to notice that each version has its own pros and contras:

- the version including the Java JDK is easier to install as it only requires to unzip the downloaded file and to run it. But the provided JDK is not automatically updated to fix security issues. This JDK should thus not be used with any other applications.
- the version without JDK requires **Java 1.8 Oracle JDK** to be installed on your computer (at least the update 161). The advantage of this version is that the file download is lighter and that the user can update the Java JDK to prevent new security vulnerabilities.

Note that the previous versions (GAMA 1.8RC2 and 1.7) came with 32 bits version for Windows and Linux (but without any version with an included Java JDK). You first need to determine which version to use (it depends on your computer, which may, or not, support 64 bits instructions, but also on the version of Java already installed, as the number of bits of the two versions must match). **It is not recommended to use it, as many issues have been fixed and many improvements have been introduced in the release**. Nevertheless, it can be downloaded from the page.

Install procedure

After having downloaded the chosen GAMA version from the Downloads page, you only have to extract the zip file wherever you want on your machine, and launch GAMA.

System Requirements

GAMA 1.8.0 requires approximately 540MB of disk space (resp. 250MB in its version without Java JDK) and a minimum of 4GB of RAM (to increase the portion of memory usable by GAMA, please refer to these instructions).

The version with JDK does not require the installation of any other software and in particular the Java JDK.

The version without JDK requires that **Java 1.8 Oracle JDK** is installed on your machine.

Please note that GAMA is not considered as compatible with Java 1.9 and Java 1.10 as it has not been tested under these environments.

Installation of Java

On all environments, the recommended Java Virtual Machine under which GAMA has been tested is the one distributed by Oracle (http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html). **Please make sure to install the JDK (Java Development Kit) and not the JRE (Java Runtime Environment)**. GAMA may work with the standard JRE, although it will be slower and may even crash (esp. under MacOS X).

On Mac OS X

The latest version of GAMA requires a JVM (or JDK or JRE) compatible with Java 1.8 to run.

Note for GAMA 1.6.1 users: if you plan to keep a copy of GAMA 1.6.1, you will need to have both Java 1.6 (distributed by Apple) and Java 1.8 (distributed by Oracle) installed at the same time. Because of this bug in SWT (https://bugs.eclipse.org/bugs/show_bug.cgi?id=374199), GAMA 1.6.1 will not run correctly under Java 1.8 (all the displays will appear empty). To install the JDK 1.6 distributed by Apple, follow the instructions here: http://support.apple.com/kb/DL1572. Alternatively, you might want to go to https://developer.apple.com/downloads and, after a free registration step if you're not an Apple Developer, get the complete JDK from the list of downloads.

On Windows

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

- To download the appropriate java version, follow this link: https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Execute the downloaded file
- You can check that a **Java\jre8** folder has been installed at the location **C:\Program Files**

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

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

On Ubuntu & Linux

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

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

The Oracle JDK License has changed for releases starting April 16, 2019. The result is that it's now more complicated to install Oracle JDK on Unix system than before.

If you want to simplify the process, you can download GAMA with an embarked JDK. But keep in mind that this JDK should only be used to run GAMA-Platform.

Install the Oracle JDK 8

If you still want to install Oracle JDK 8 on your machine, here are some workarounds: * For Debian based OS * For Arch-based OS

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

Install the OpenJDK 8

/! WARNING /!
OpenJDK is not the recommended way to run GAMA and is not (and won't be) supported.
We won't help you if you run in any trouble using this JDK.

Another solution will be to install OpenJDK, the free implementation under the GNU General Public License.

If you use a Debian based OS (Ubuntu, Linux Mint, ...), you need to do:

```
sudo apt-get install openjdk-8-jdk
```

If you use an Arch-based OS (Manjaro, Antergos, ...), you need to do:

```
sudo pacman -S jdk8-openjdk
```

If you use a Red Hat-based OS (CentOS, Fedora, Scientific Linux ...), you need to do:

```
su -c "yum install java-1.8.0-openjdk"
```

You can then switch between java version using:

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

Troubleshooting with Mac OS X Sierra

In some cases, "Archive utility.app" in MacOS may damage the files when extracting them from the zip or tar.gz archive files. This problem manifests itself by a dialog opening and explaining that the application is damaged and cannot be launched (see Issue 2082 and also this thread. In that case, to expand the files, consider using a different utility, like the free Stuffit Expander or directly from the command line.

Mac OS X Sierra has introduced a series of issues linked to the so-called "quarantine" mode (where applications downloaded from Internet prevent to use and update their internal components, such as the models of the library or the self-updating of the application). See this page for background information. To be certain that Gama will work, and until we find an easier solution, the installation should follow these steps:

1. Download the GAMA zip file
2. Unzip it (possibly with another archive utility, see above)
3. Copy and paste `Gama` in the `Applications` folder
4. Launch `Terminal.app`
5. Type `cd /Applications` and hit return.
6. Type `xattr -d -r com.apple.quarantine Gama.app/` and hit return to remove the quarantine attribute

From now on, Gama should be fully functional.

Chapter 9

Launching GAMA

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

Table of contents

Launching the Application

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

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

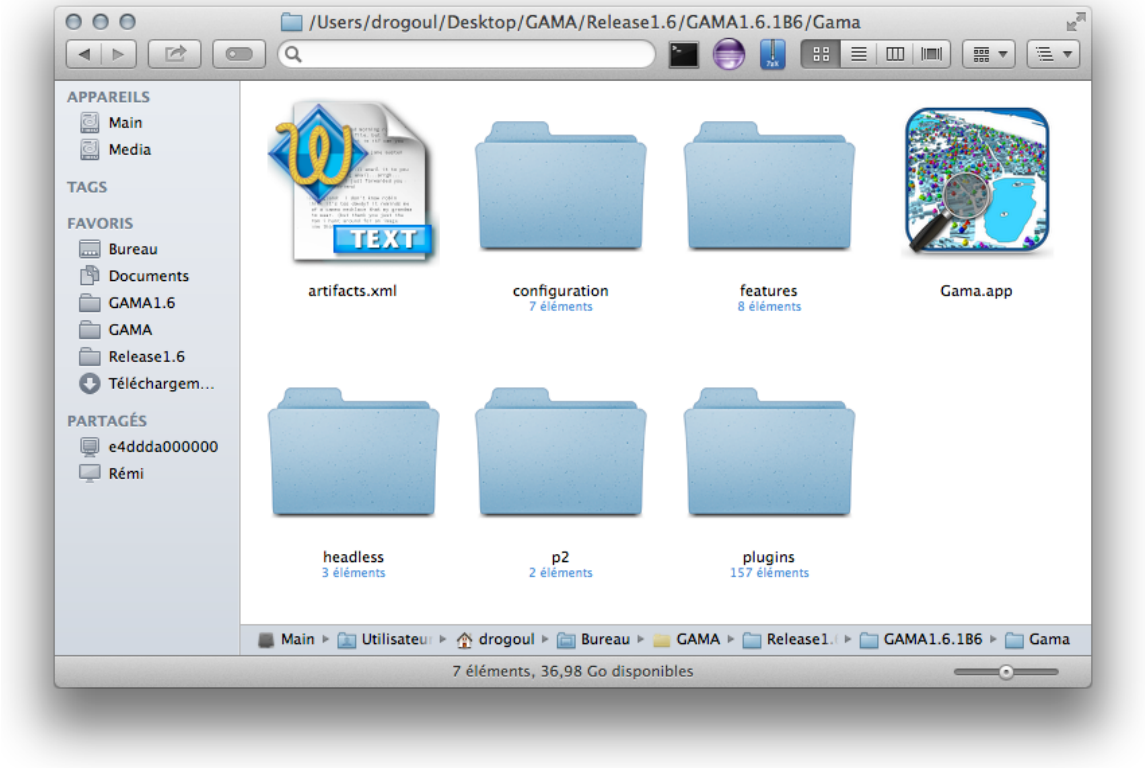


Figure 9.1: Eclipse folder.

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

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

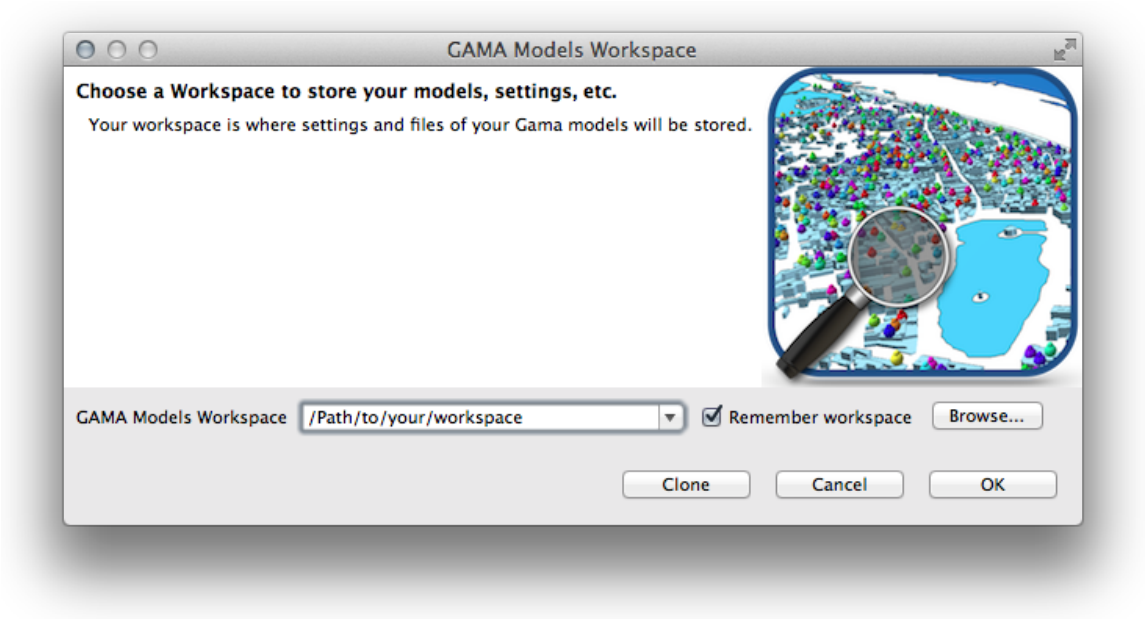


Figure 9.2: Window to choose the workspace.

Choosing a Workspace

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

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

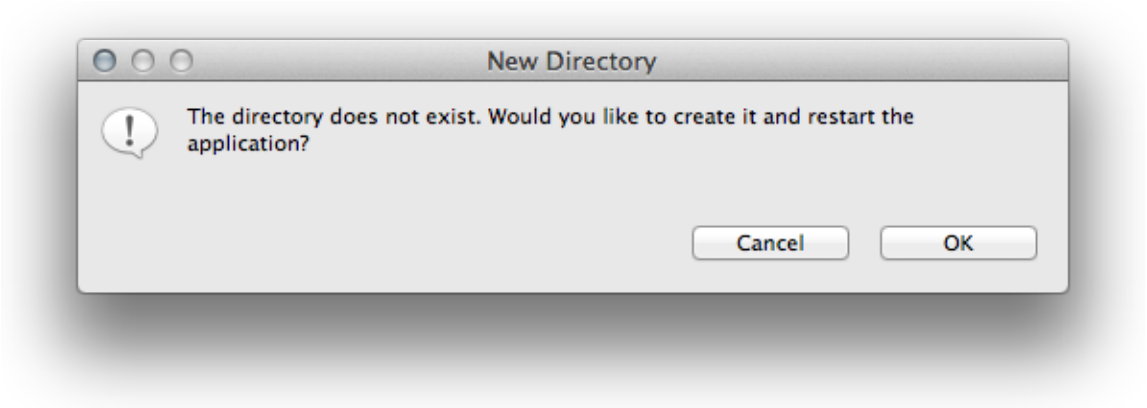


Figure 9.3: This pop-up appears when the user wants to create a new workspace. Click on OK.

Welcome Page

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

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

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

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

Figure 9.4: GAMA after the first launch.

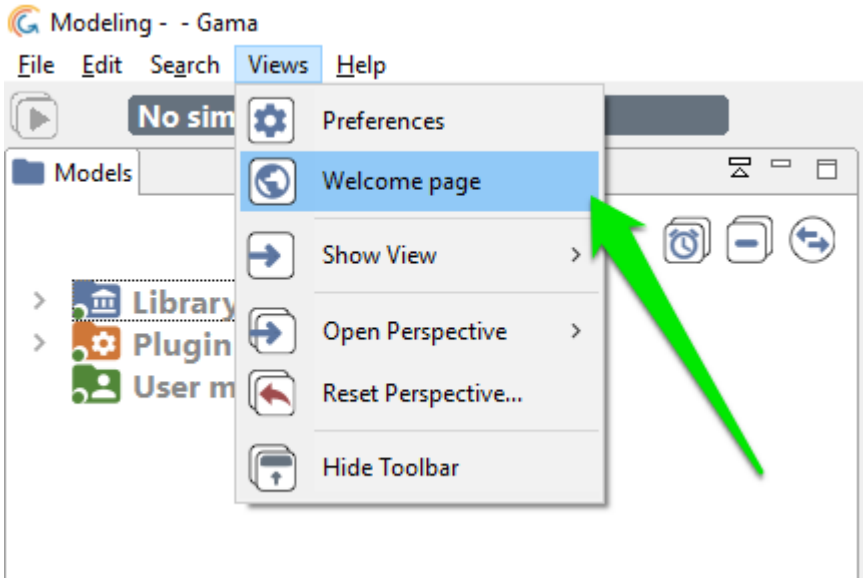


Figure 9.5: Menu to open new views.

Chapter 10

Headless Mode

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

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

Table of contents

Command

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

Bash Script

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

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

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

Experiment Input File

The XML input file contains for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Experiment_plan>
 <Simulation id="2" sourcePath="./predatorPrey/predatorPrey.gaml"
   finalStep="1000" experiment="predPrey">
  <Parameters>
    <Parameter name="nb_predator_init" type="INT" value="53" />
```

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

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

Heading

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

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

Parameters

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

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

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

Outputs

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

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

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

Output Directory

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

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

- with:
  - `simulation-output.xml`: containing the results

- `snapshot`: containing the snapshots produced during the simulation

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

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

Simulation Output

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

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

...
```

- With:
  - `<Simulation id="2" >` : block containing results of the simulation 2 (this Id is identified in the Input Experiment File)
  - `<Step id='1' > ... </Step>`: one block per step done. The id corresponds to the step number

Step

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

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

Variable

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

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

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

Snapshot files

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

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

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

Chapter 11

Updating GAMA

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

Table of contents

Manual Update

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

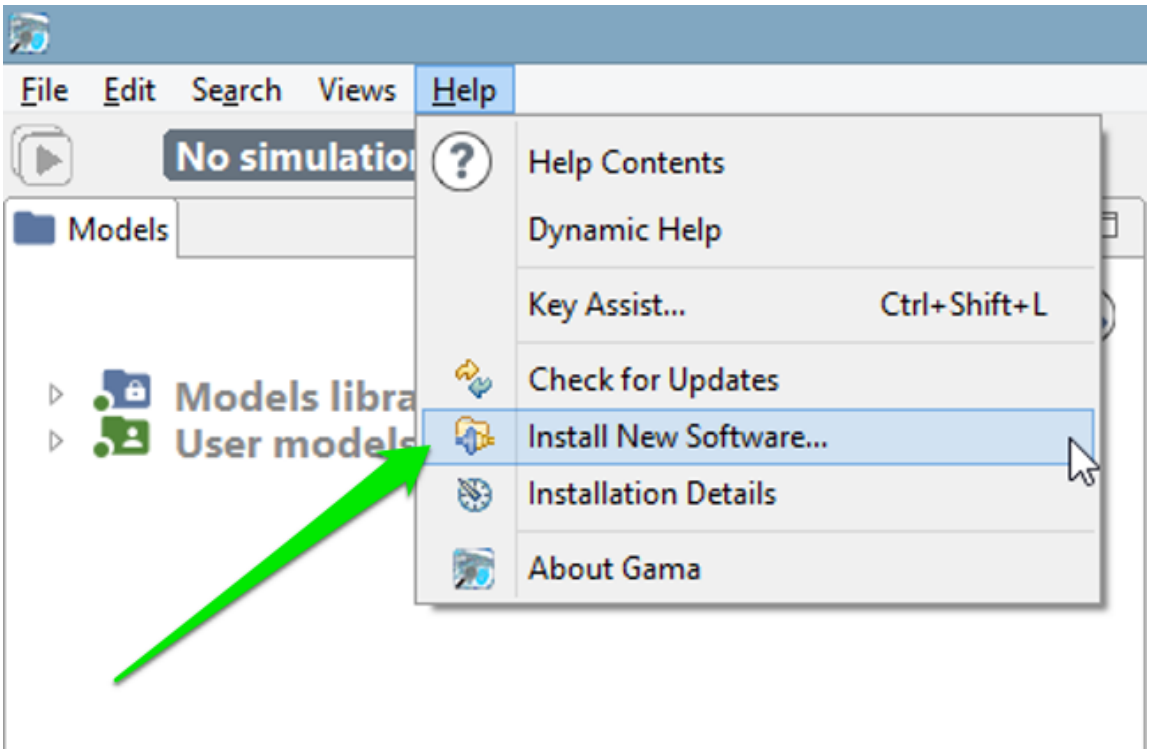


Figure 11.1: Menu to install new extensions to GAMA.

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

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

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

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

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

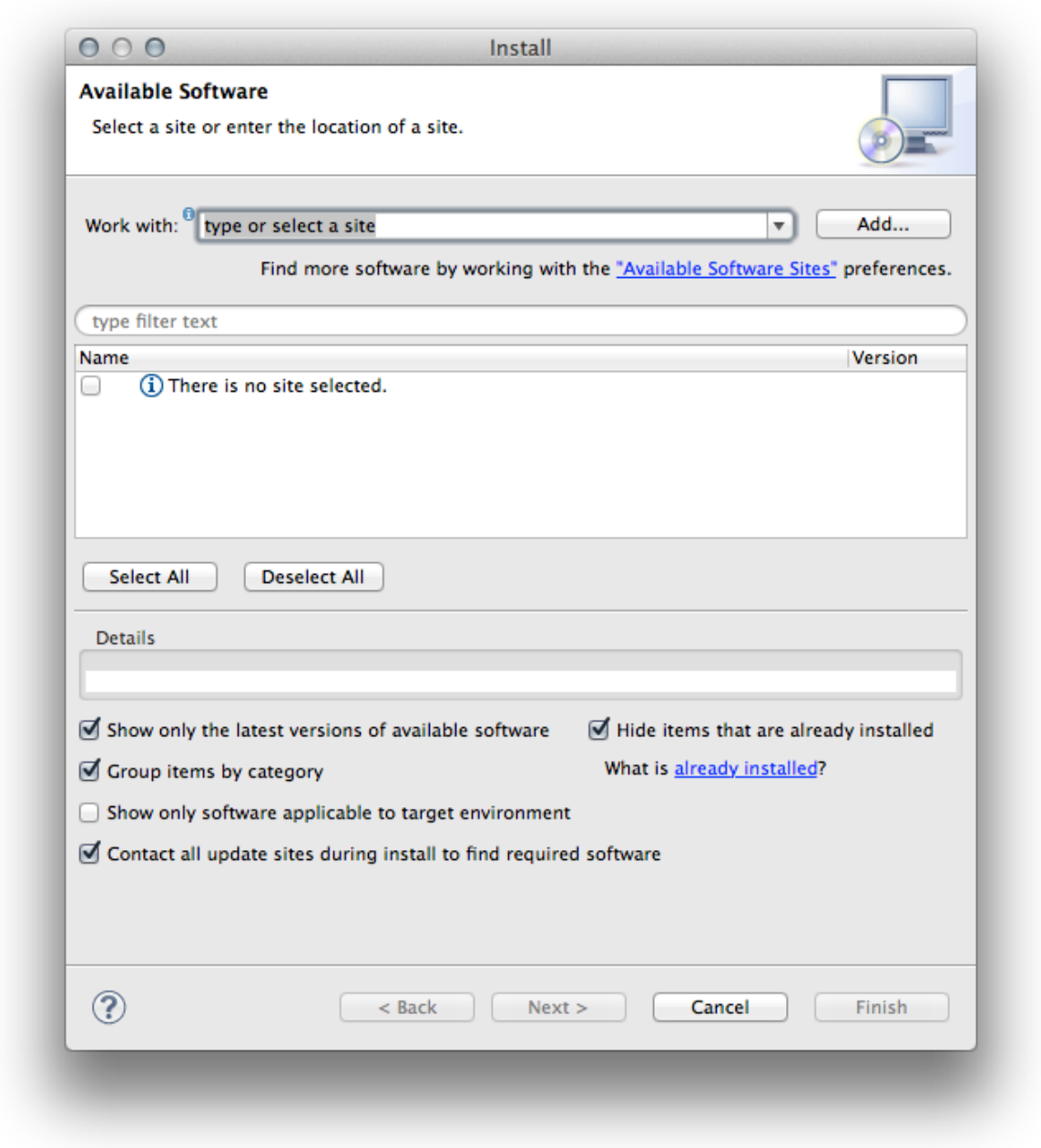


Figure 11.2: Window where the user enters the address of an update site and can choose plugins to install.

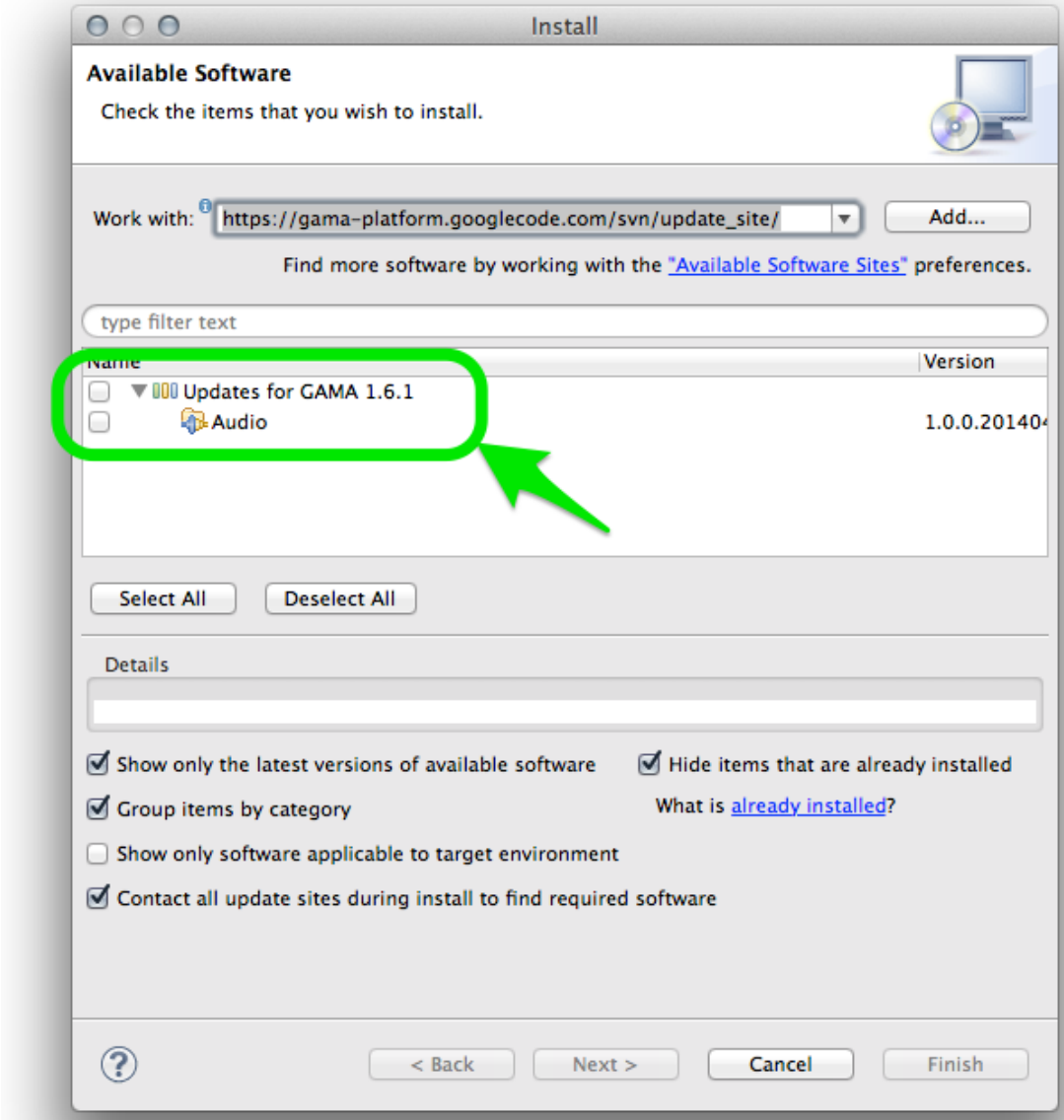


Figure 11.3: Display of the list of available extensions.

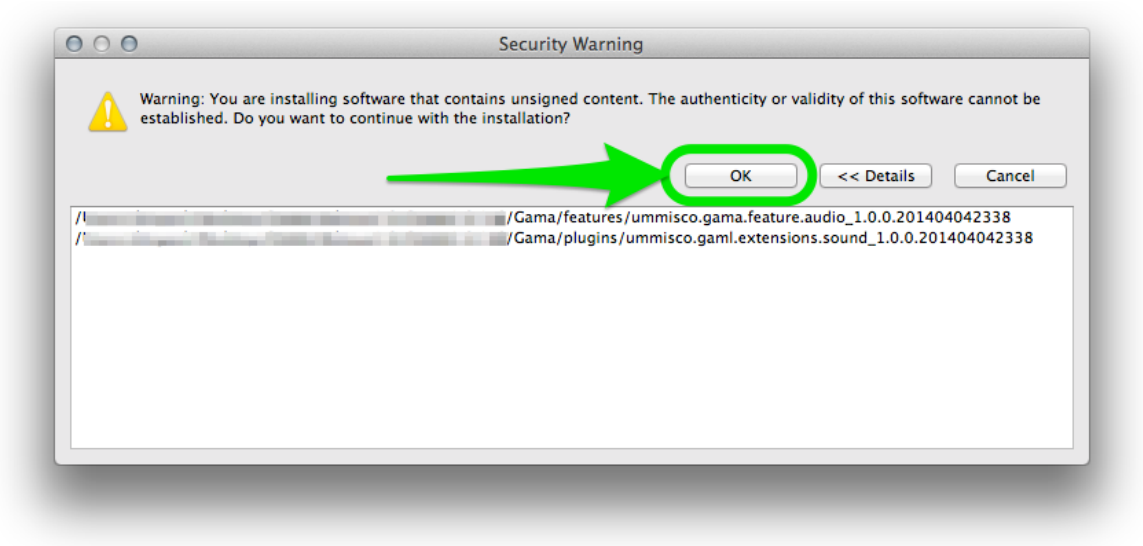


Figure 11.4: Warning window that can be dismissed.

Choose the ones you want to install (or update) and click "Next...". A summary page will appear, indicating which plugins will actually be installed (since some plugins might require additional plugins to run properly), followed by a license page that you have to accept. GAMA will then proceed to the installation (that can be canceled any time) of the plugins chosen.

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

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

Automatic Update

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

In the dialog that appears, navigate to "Install/Update > Automatic Updates". Then, enable the option using the check-box on the top of the dialog and choose the best

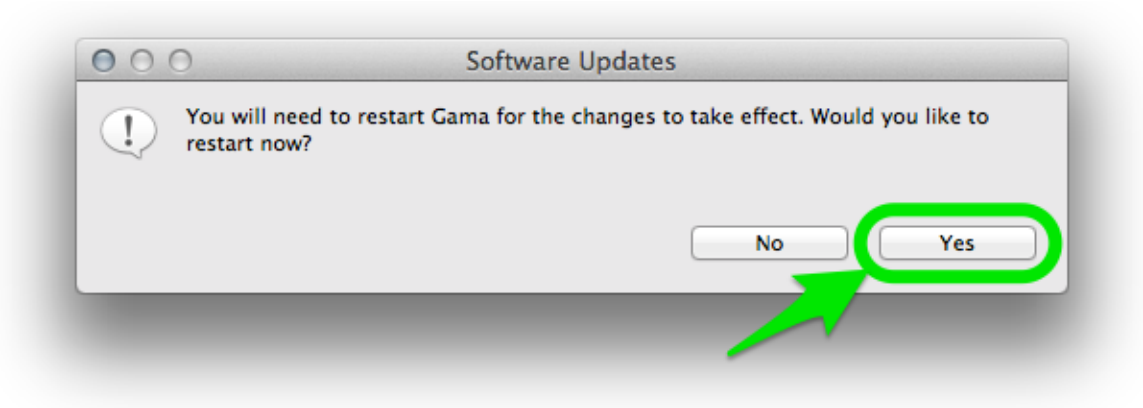


Figure 11.5: After installation, GAMA has to be restarted.

settings for your workflow. Clicking on "OK" will save these preferences and dismiss the dialog.

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

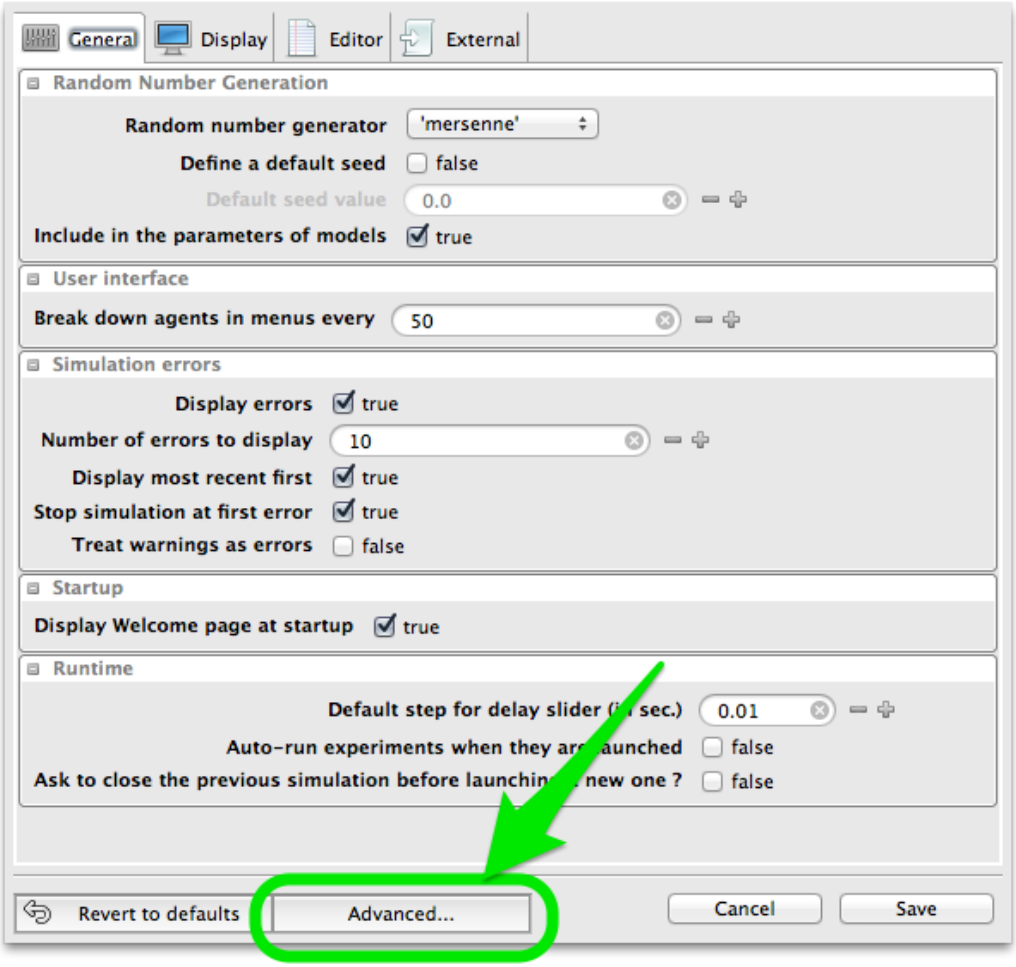


Figure 11.6: Button to give access to additional preferences.

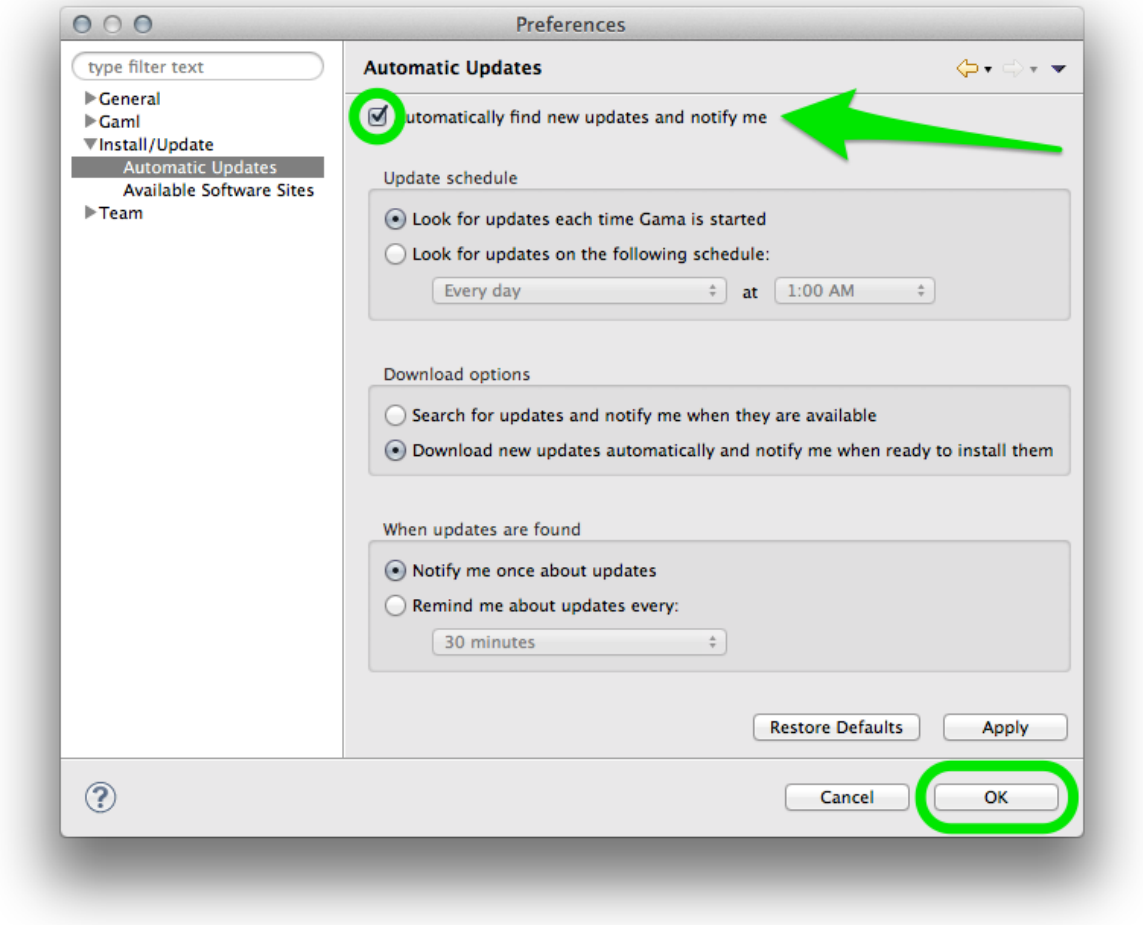


Figure 11.7: Check for automatic update.

Chapter 12

Installing Plugins

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

There are, however, three important restrictions:

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

With these restrictions in mind, it is however possible to install interesting additional plugins. We propose here a list of some of these plugins (known to work with GAMA), but feel free to either add a comment if you have tested plugins not listed here or

create an issue if a plugin does not work, in order for us to see what the requirements to make it work are and how we can satisfy them (or not) in GAMA.

Table of contents

Installation

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

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

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

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

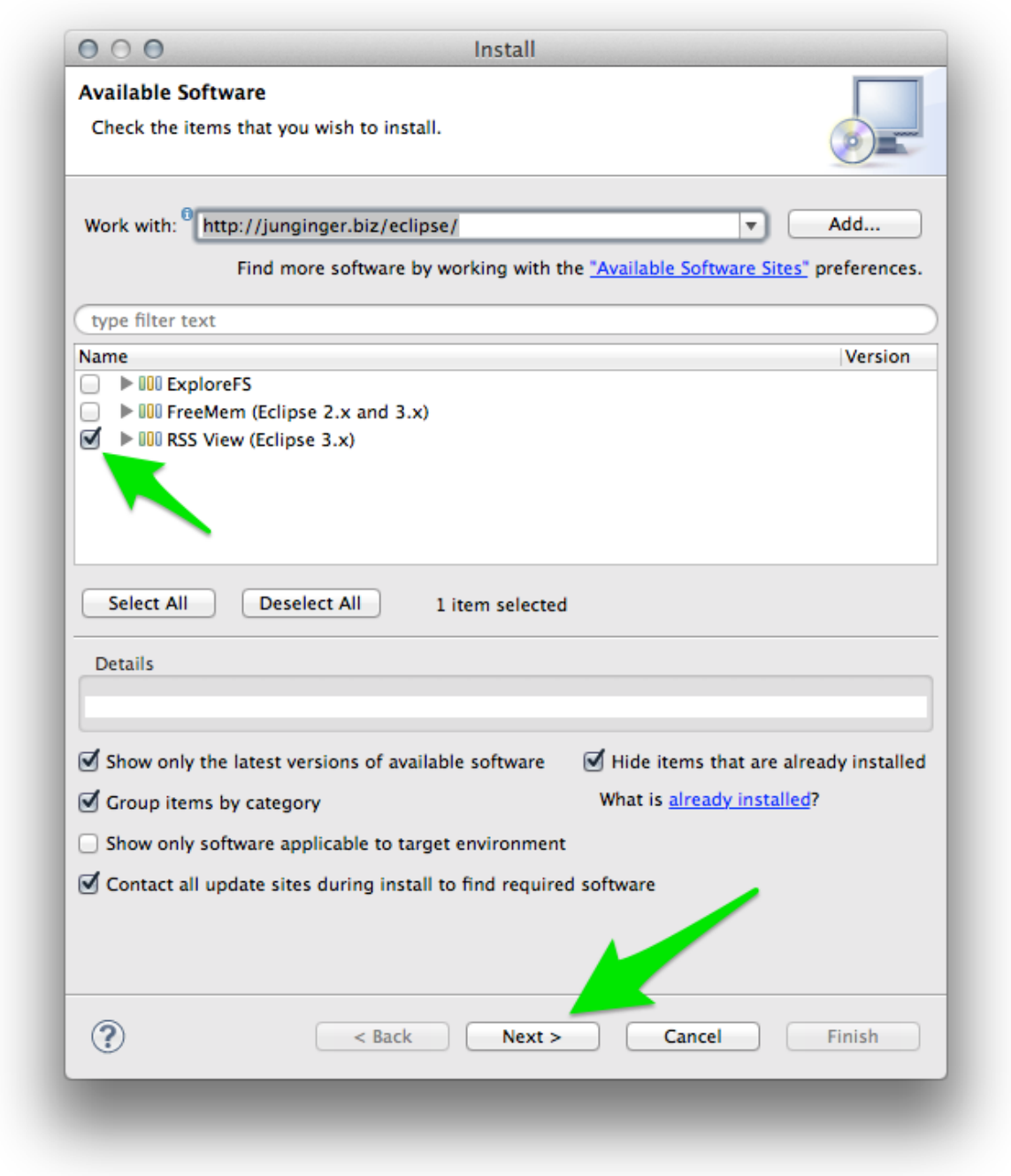
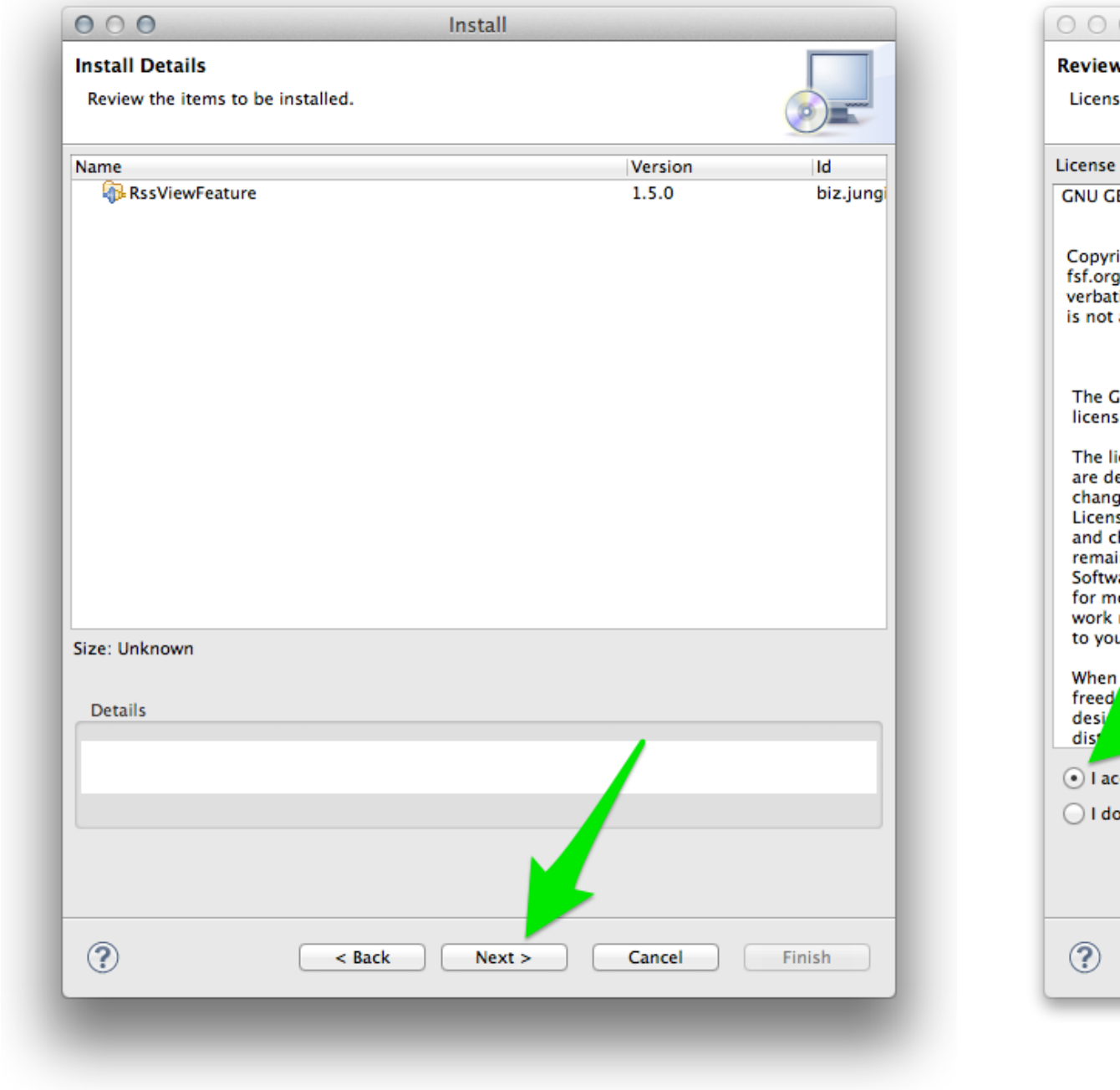


Figure 12.1: images/dialog_install_plugins.png



Once we dismiss the warning that the plugin is not signed and accept to restart

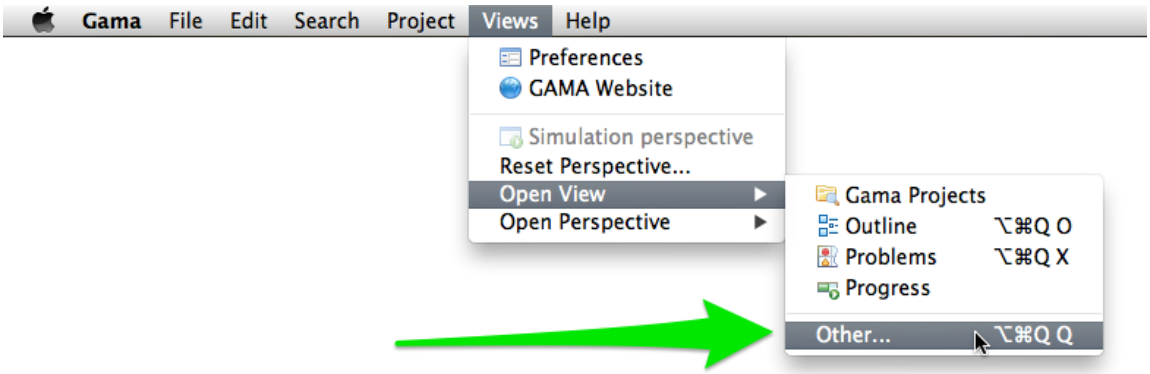


Figure 12.2: images/menu_other_views.png

GAMA, we can test the new plugin by going to the "Views" menu.

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

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

Selected Plugins

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

Overview

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

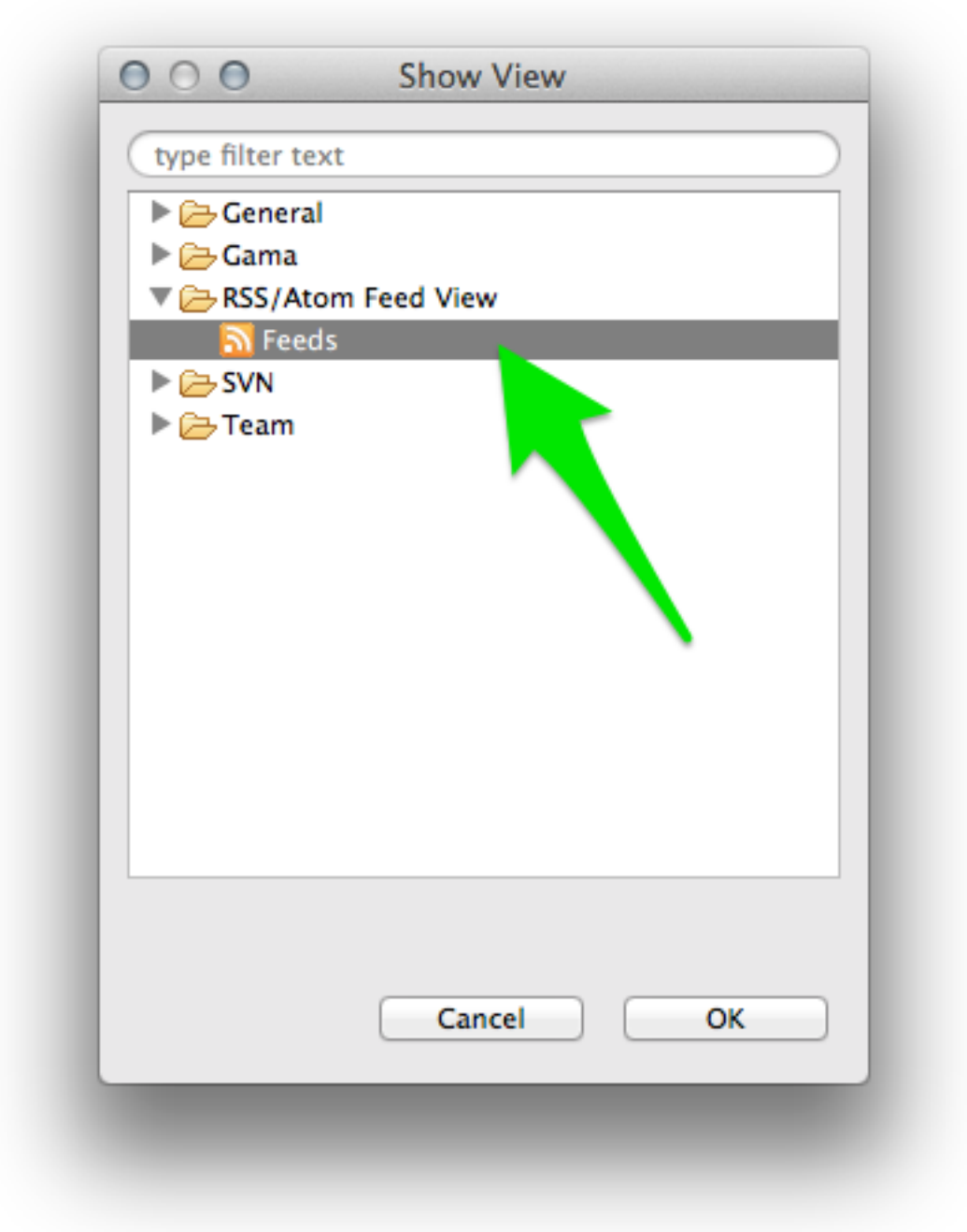


Figure 12.3: images/dialog_show_view.png

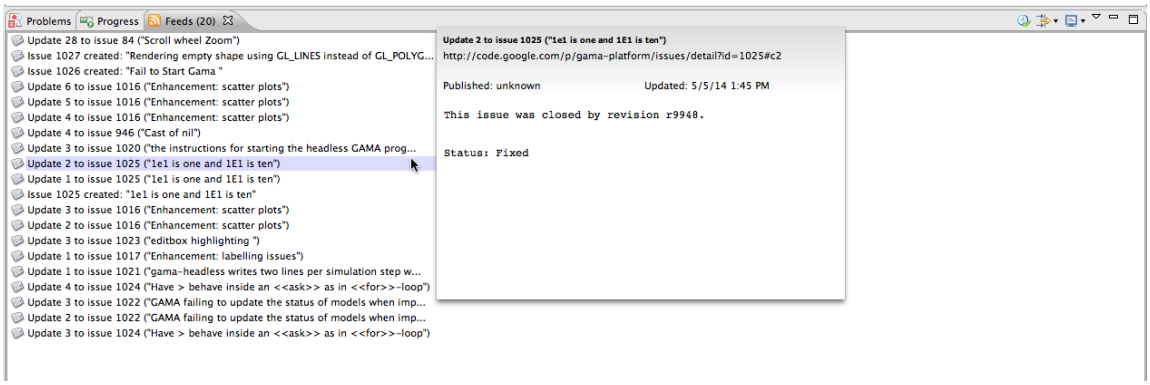


Figure 12.4: images/feed_working.png

Git

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

CKEditor

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

Startexplorer

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

whatever, or in a terminal).
- Update site: http://basti1302.github.com/startexplorer/update/

Pathtools

- Same purpose as StartExplorer, but much more complete, and additionally offers the possibility to add new commands to handle files (open them in specific editors, execute external programs on them, etc.). Very nice and professional. Works flawlessly in GAMA except that contributions to the toolbar are not accepted (so you have to rely on the commands present in the Navigator pop-up menu).
- Update site: http://pathtools.googlecode.com/svn/trunk/PathToolsUpdateSite/site.xml
- Website: https://pathtools.googlecode.com

CSV Edit

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

TM Terminal

- A powerful, yet simple, terminal which can connect locally or via ssh or other methods.
- Update site (general): http://download.eclipse.org/releases/mars/
- Select the following plugin: TM Terminal

Quickimage

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

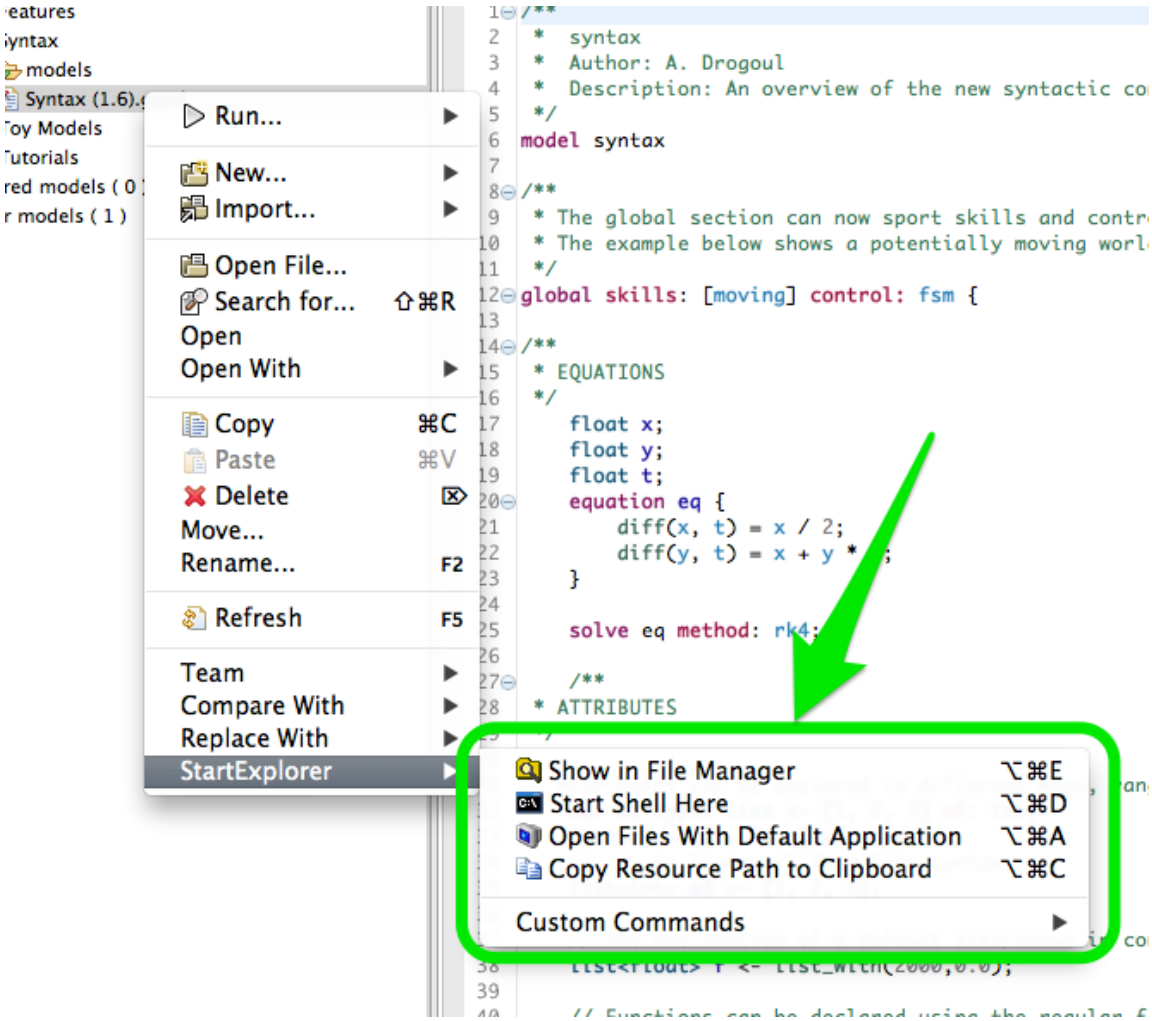


Figure 12.5: images/start_explorer.png

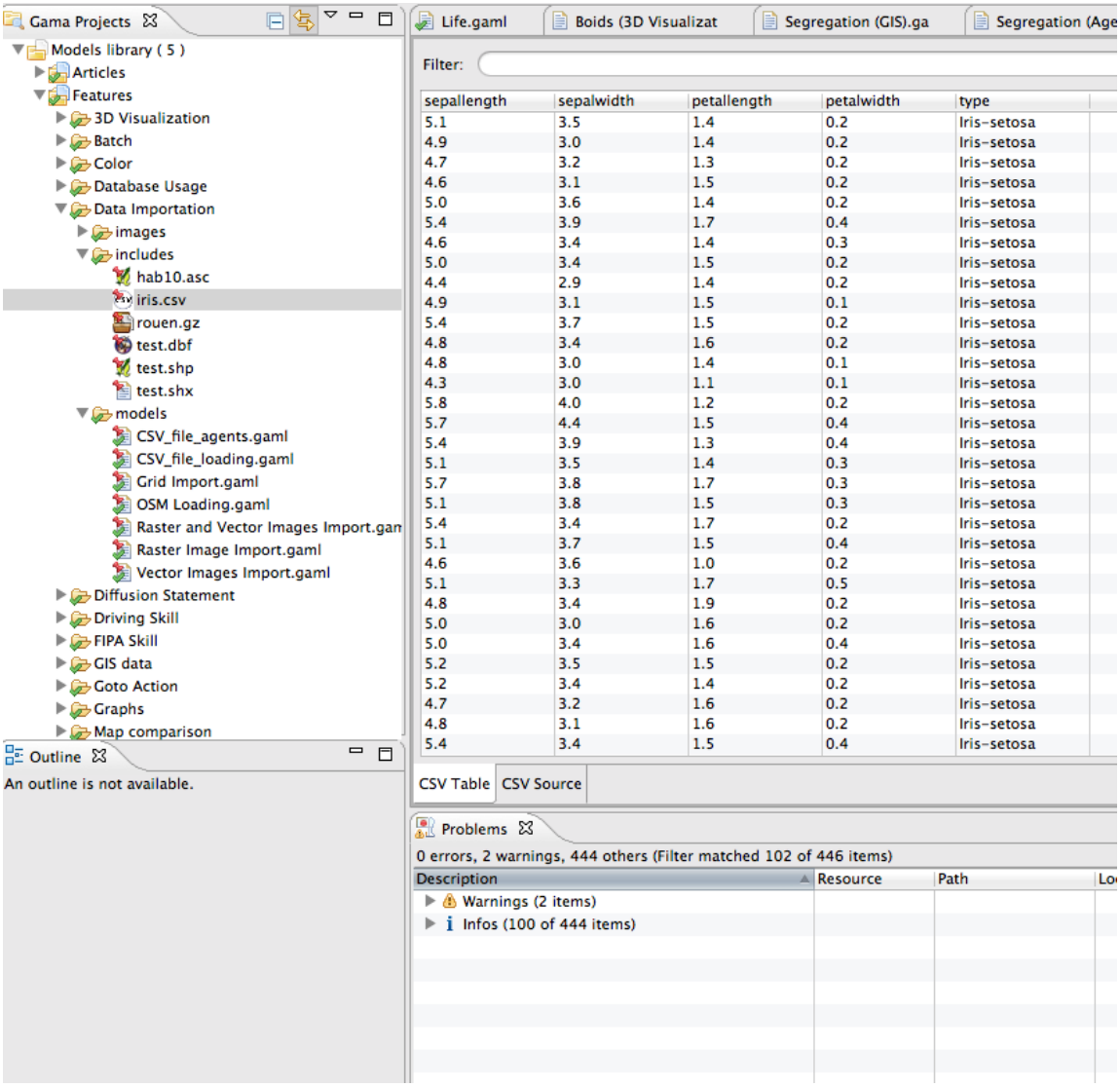


Figure 12.6: images/csv_edit.png

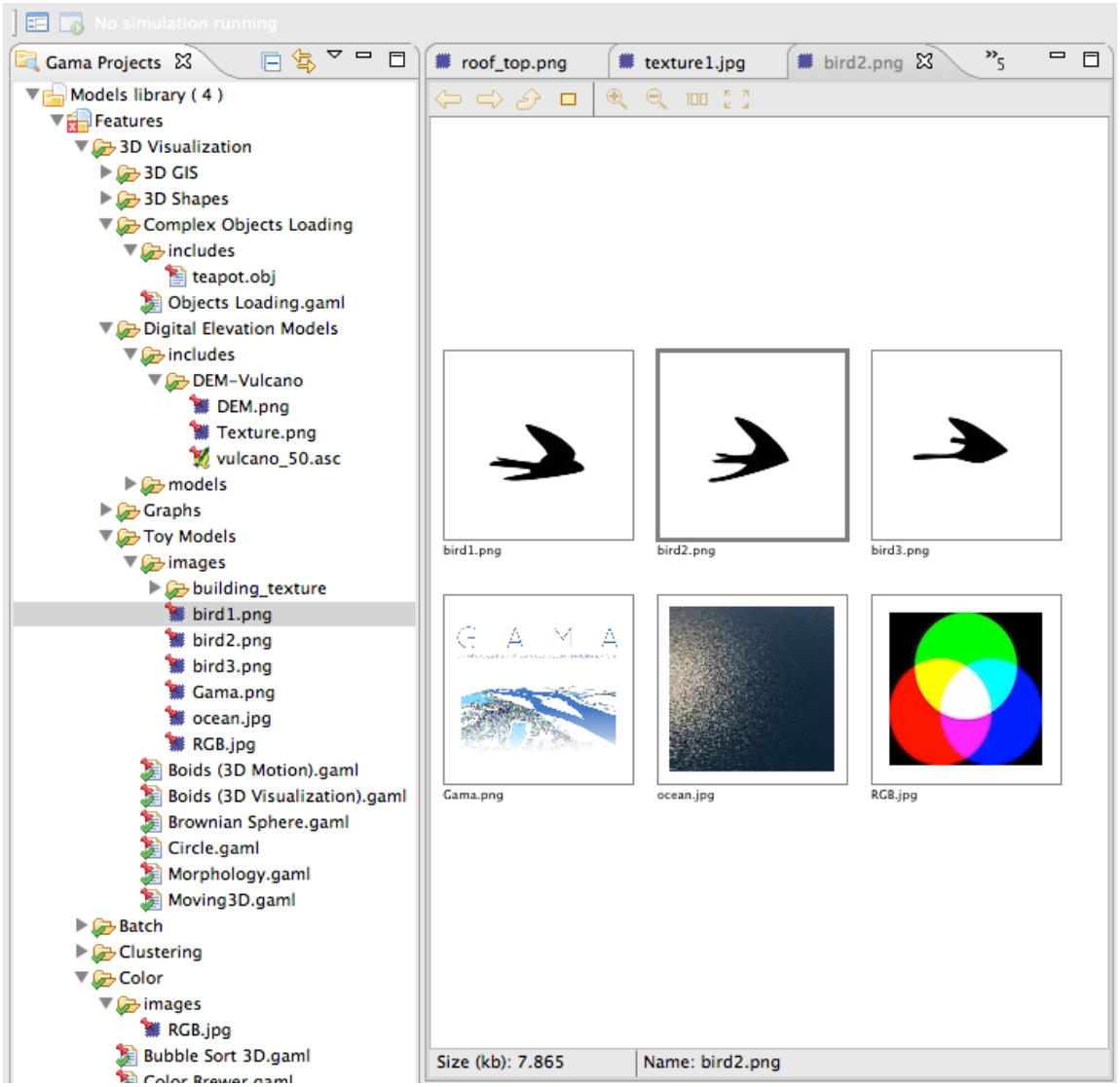


Figure 12.7: images/quick_image.png

Chapter 13

Troubleshooting

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

Table of contents

On Ubuntu (& Linux Systems)

Workaround if GAMA crashes when displaying web contents

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

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

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

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

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

Workaround if you use a GTK dark theme

If your Desktop Environment use GTK (list here) and you're using a dark theme on your session, GAMA will apply the same theme and look badly.

To solve it, you should change the variable environment on application startup. The easy way to keep that change is to modify the file *.desktop*

If you're running GAMA 1.8 or above

With the official GAMA 1.8 release, the display is based on GTK 3.

To change the display here, you should change the variable *GTK_THEME* to the theme that you prefer using. So the exec line for your desktop file will look like this :

```
Exec=env GTK_THEME=Matcha-aliz:light /path/to/../GAMA1.8/Gama
```

If you're running GAMA 1.8 - Release Candidate 2 or before

Before the official GAMA 1.8 release, the display is based on GTK 2

To change the display here, you should change the variable *GTK2_RC_FILES* to the theme that you prefer using. As the variable suggests, you should point directly to the style file and not just indicate the new theme. So the exec line for you desktop file will look like this :

```
Exec=env GTK2_RC_FILES=/usr/share/themes/Matcha-aliz/gtk-2.0/gtkrc /
   path/to/../GAMA1.8_RC2/Gama
```

On Windows

No common trouble...

On MacOS X

Workaround in case of glitches in the UI

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

Workaround in case of corrupted icons in menus under El Capitan

For some particular configurations (in particular some particular graphic cards), the icons of the menus (e.g. Edit menu) may be corrupted. This bug is documented for

all RCP products under El Capitan. See these references: https://bugs.eclipse.org/bugs/show_bug.cgi?id=479590 https://trac.filezilla-project.org/ticket/10669

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

Workaround in case of damaged extracted GAMA application under MacOSX Sierra

In some cases, "Archive utility.app" in MacOS may damage the files when extracting them from the zip or tar.gz archive files. This problem manifests itself by a dialog opening and explaining that the application is damaged and cannot be launched (see Issue 2082 and also this thread. In that case, to expand the files, consider using a different utility, like the free Stuffit Expander or directly from the command line.

MacOS Sierra has introduced a series of issues linked to the so-called "quarantine" mode (where applications downloaded from Internet prevent to use and update their internal components, such as the models of the library or the self-updating of the application). See this page for background information. To be certain that Gama will work, and until we find an easier solution, the installation should follow these steps:

1. Download the GAMA zip file
2. Unzip it (possibly with another archive utility, see above)
3. Copy and paste `Gama` in the `Applications` folder
4. Launch `Terminal.app`
5. Type `cd /Applications` and hit return.
6. Type `xattr -d -r com.apple.quarantine Gama.app/` and hit return to remove the quarantine attribute

From now on, Gama should be fully functional.

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

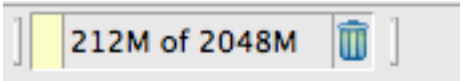


Figure 13.1: images/memory_status.png

will need between 40 and 200MB of memory to run "normally" and launch small models. Memory problems are easy to detect: on the bottom right corner of its window, GAMA will always display the status of the current memory. The first number represents the memory currently used (in MB), the second (always larger) the memory currently allocated by the JVM. And the little trash icon allows to "garbage collect" the memory still used by agents that are not used anymore (if any). If GAMA appears to hang or crash and if you can see that the two numbers are very close, it means that the memory required by GAMA exceeds the memory allocated.

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

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

Submitting an Issue

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

- A complete description of the problem and how it occurred.

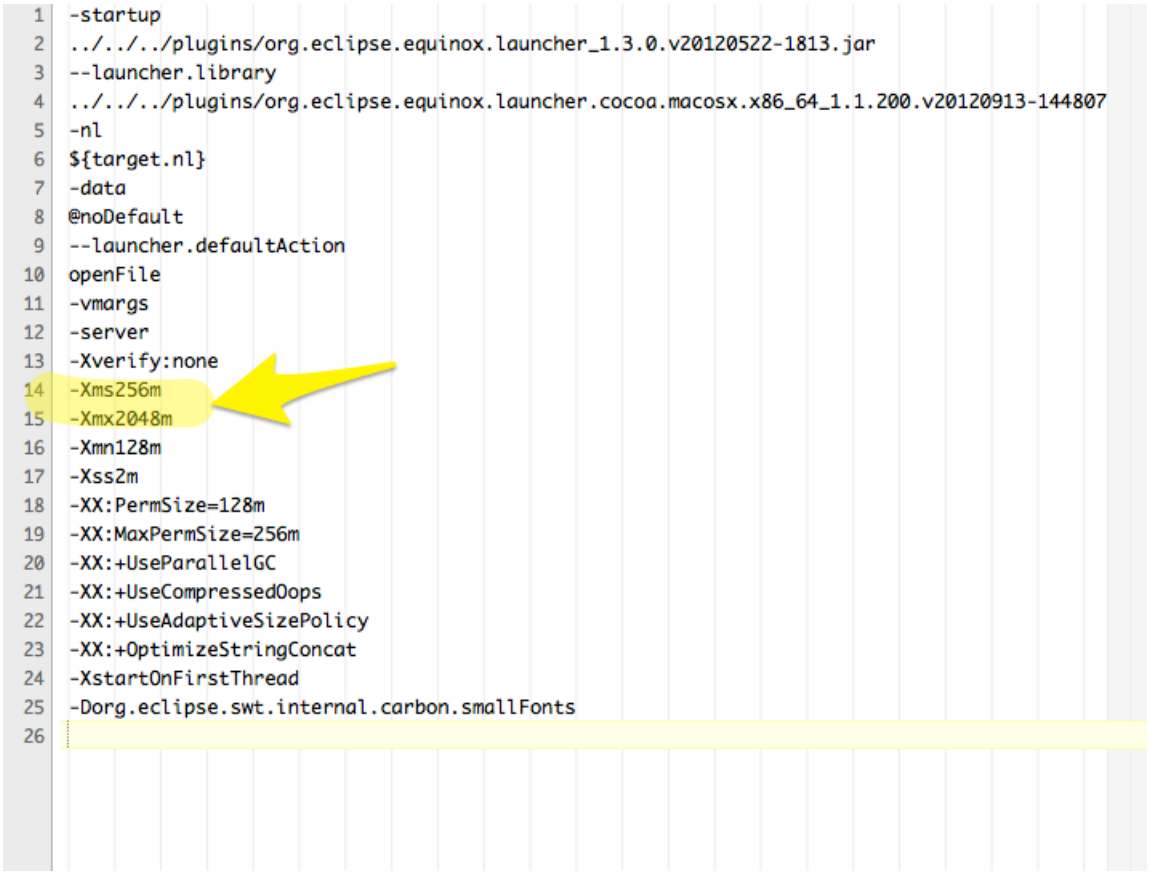


Figure 13.2: images/gama_ini.png

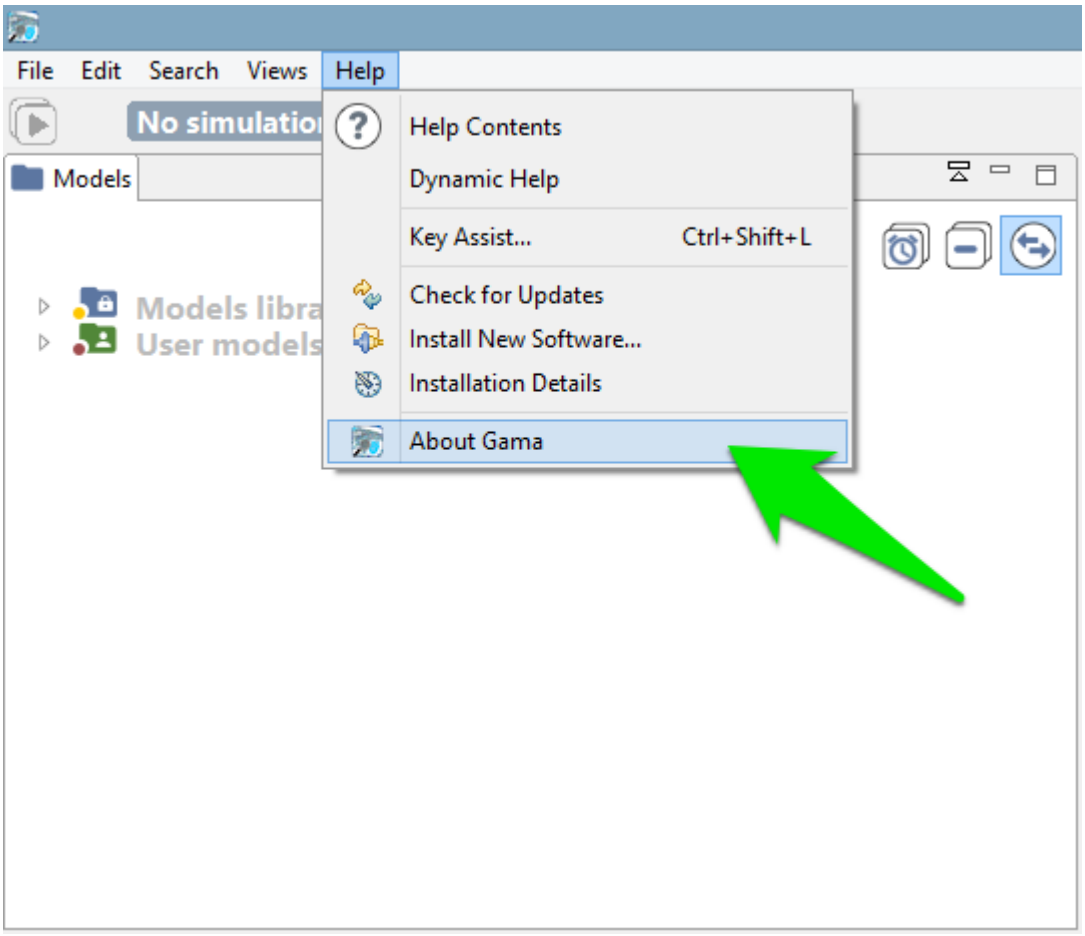


Figure 13.3: images/menu_about_gama.png

- The GAMA model or code you are having trouble with. If possible, attach a complete model.
- Screenshots or other files that help describe the issue.

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

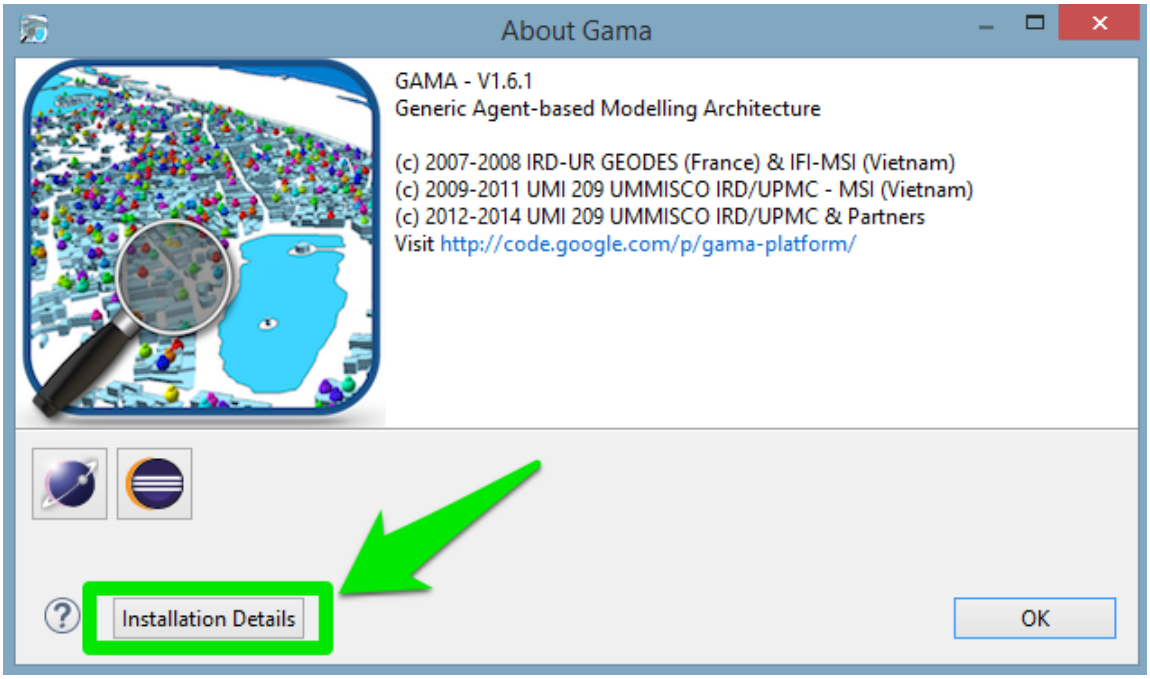


Figure 13.4: images/dialog_about_gama.png

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

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

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

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

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

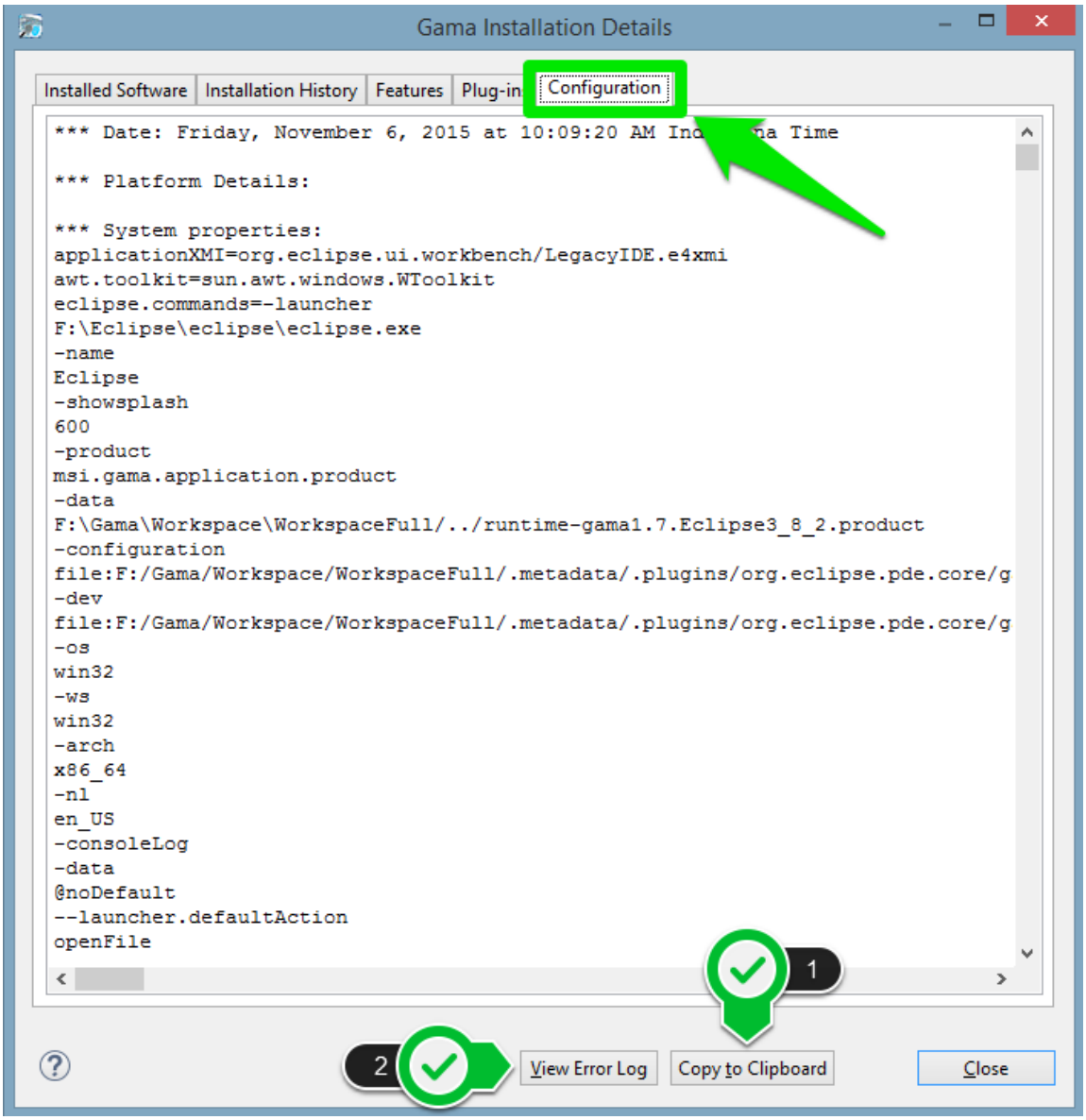


Figure 13.5: images/dialog_configuration.png

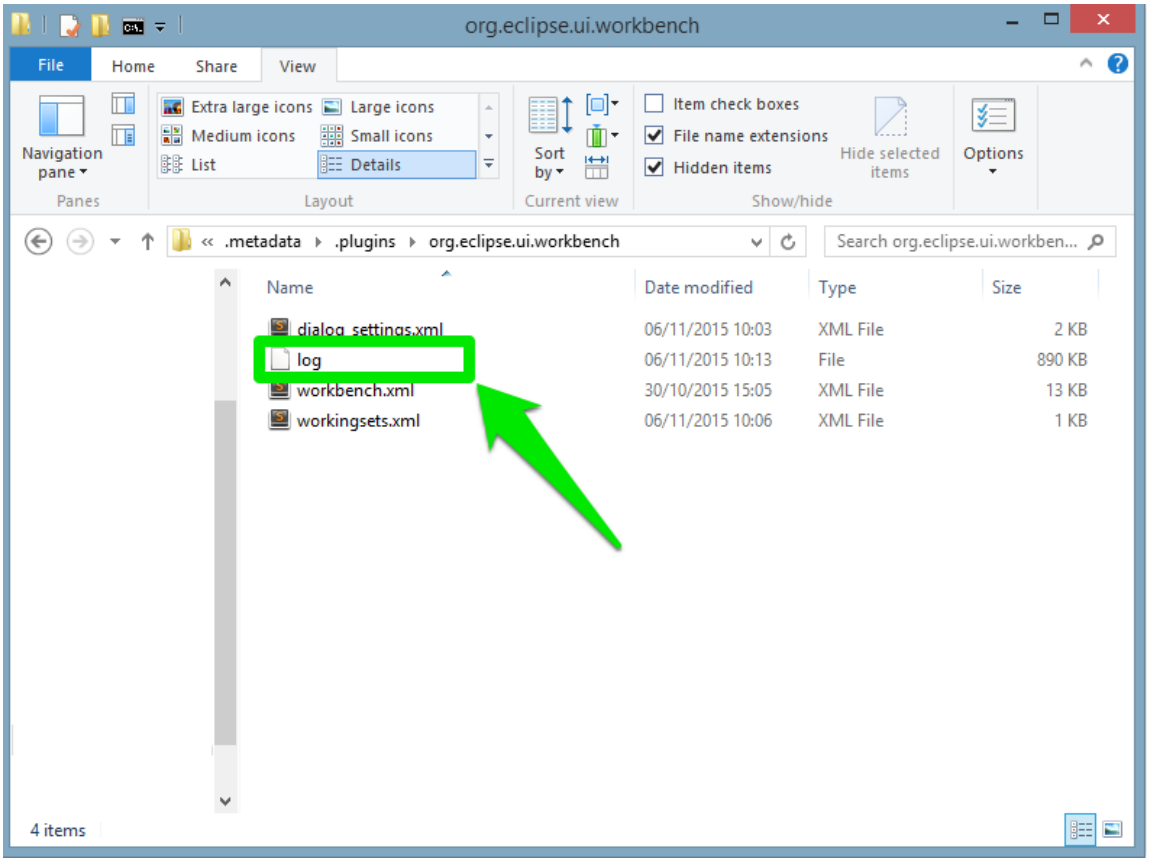


Figure 13.6: images/log_file.png

Chapter 14

Workspace, Projects and Models

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

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

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

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

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

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

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

Chapter 15

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

The Different Categories of Models

In the *Navigator*, models are organized in three different categories: *Models library*, *Plugin models*, and *User models*. This organization is purely logical and does not reflect where the models are actually stored in the workspace (or elsewhere). Whatever

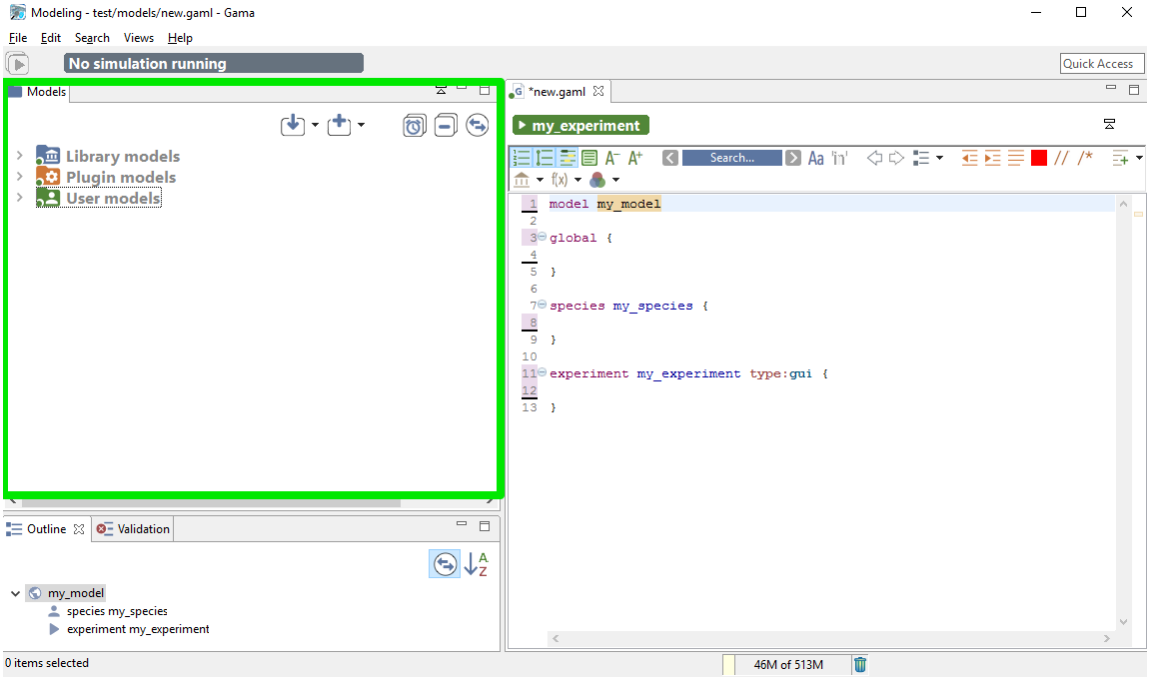


Figure 15.1: images/navigator_first.png

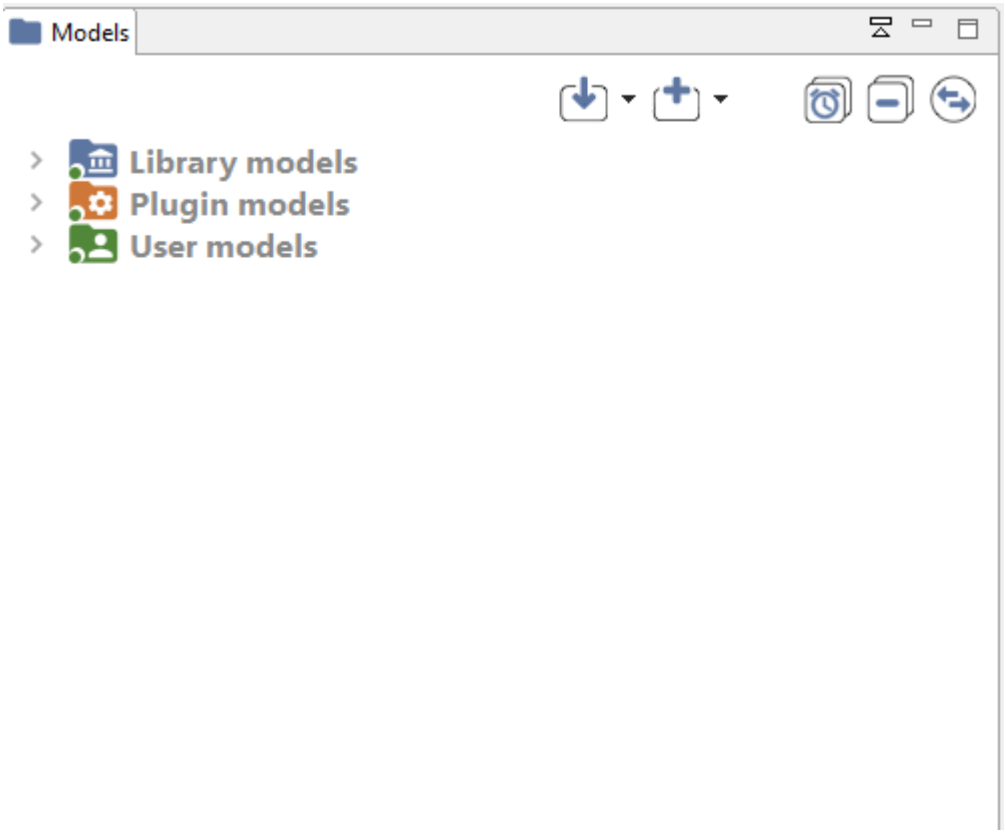


Figure 15.2: images/navigator_3_categories.png

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

Models library

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

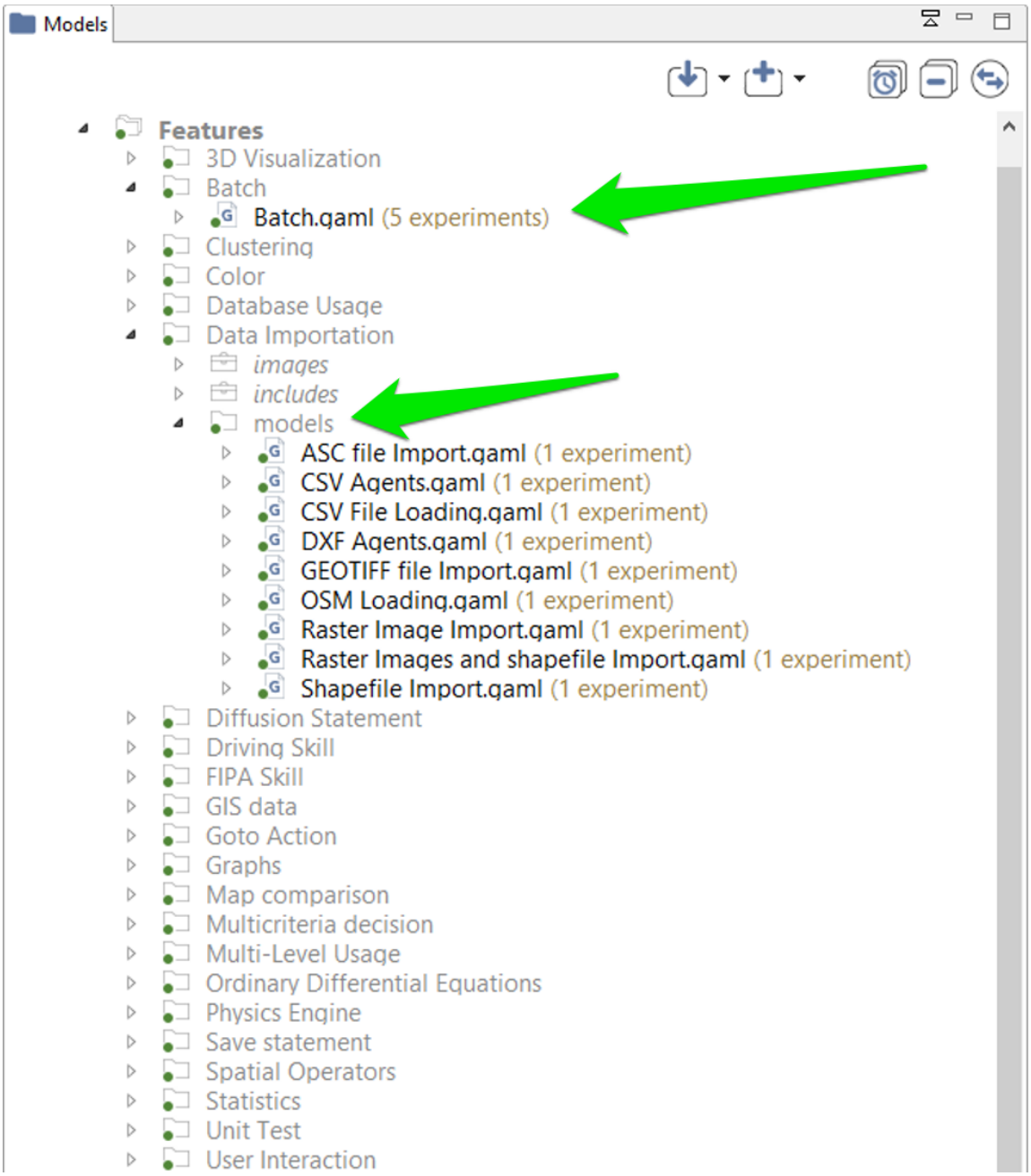


Figure 15.3: images/navigator_library_2_folders_expanded.png

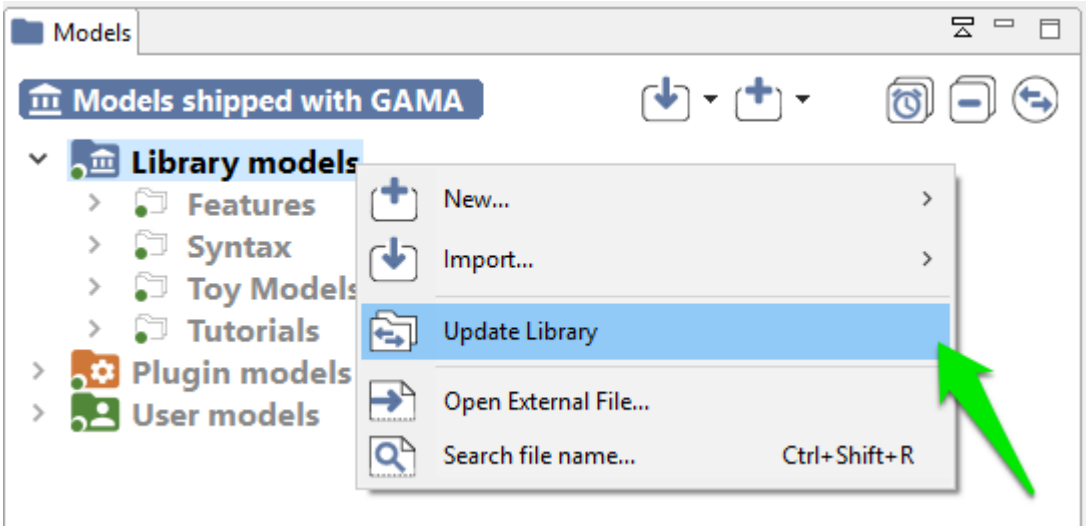
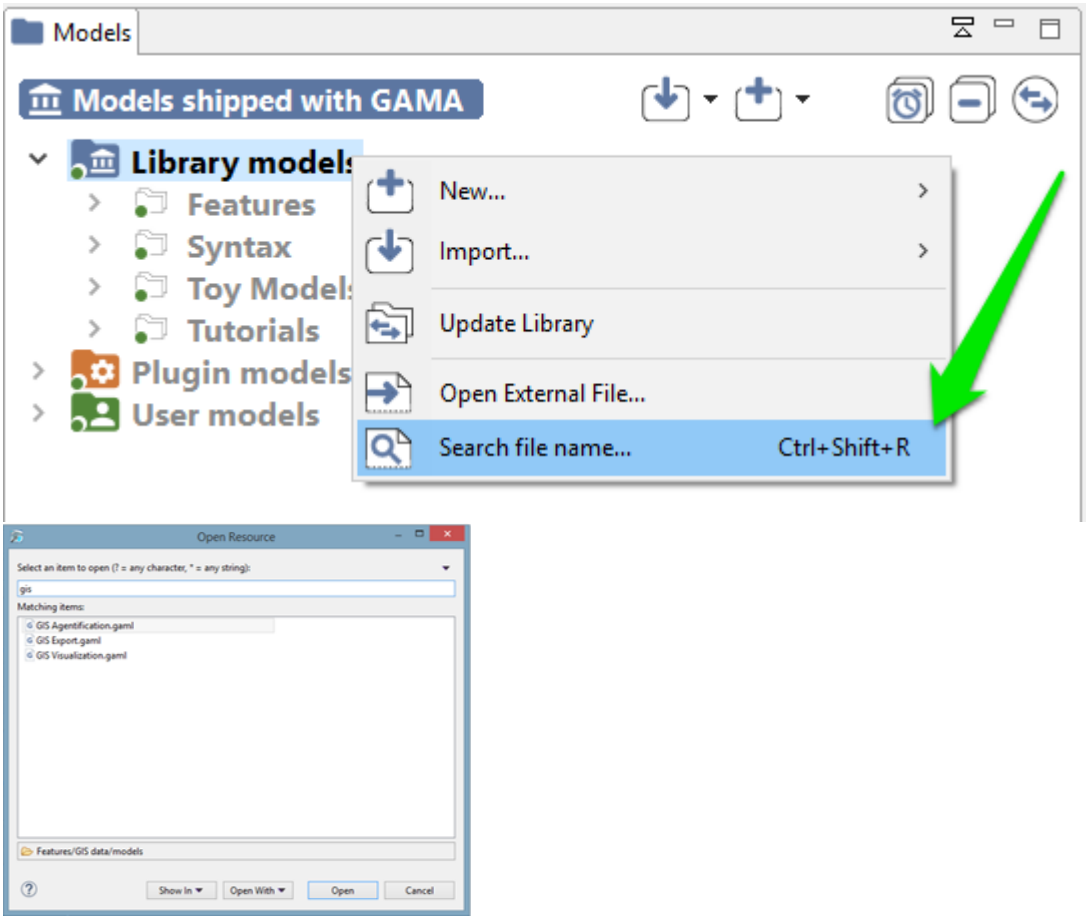


Figure 15.4: images/navigator_update_library.png

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

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



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.

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

User models

This category regroups all the projects that have been created or imported in the workspace by the user. Each project is actually a folder that resides in the folder

Figure 15.5: images/navigator_plugin_models.png

of the workspace (so they can be easily located from within the filesystem). 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".

Inspect Models

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

- **Experiment button** : Experiment button are present if your model contains experiments (it is usually the case !). To run the corresponding experiment, just click on it. To learn more about running experiments, jump into this section.
- **Require node** : The node *Require* is present if your model uses some plugins (additional or integrated by default). Each plugin is listed in this node, with a green icon if the plugin is already installed in your GAMA, and a red one if it is not the case. If the plugin you want in not installed, an error will be raised in your model. Please read about how to install plugins to learn some more about it.

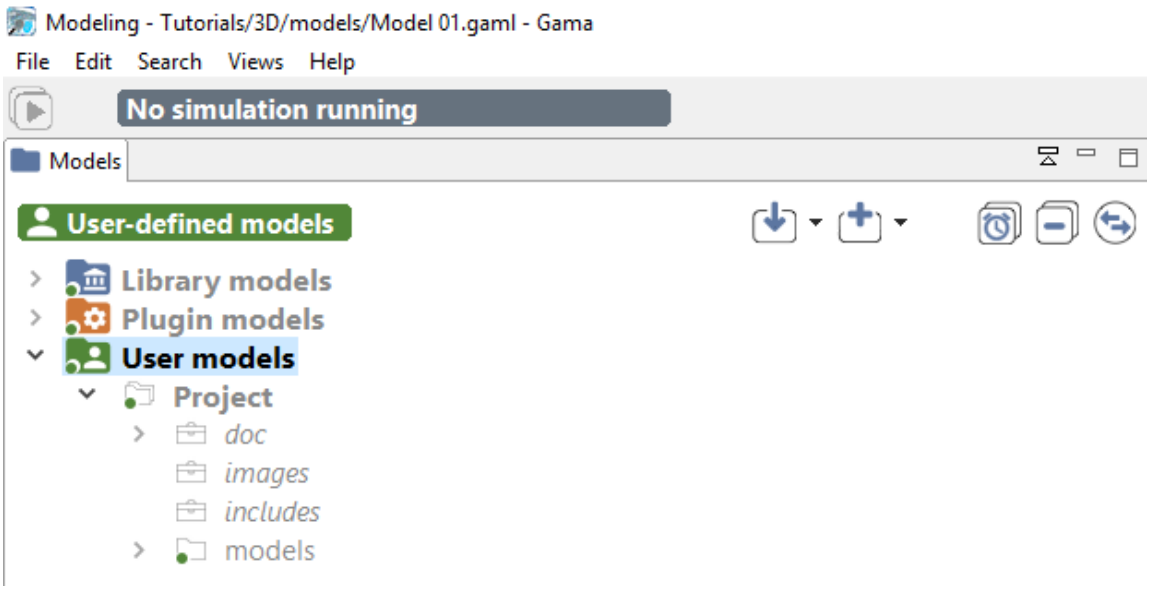
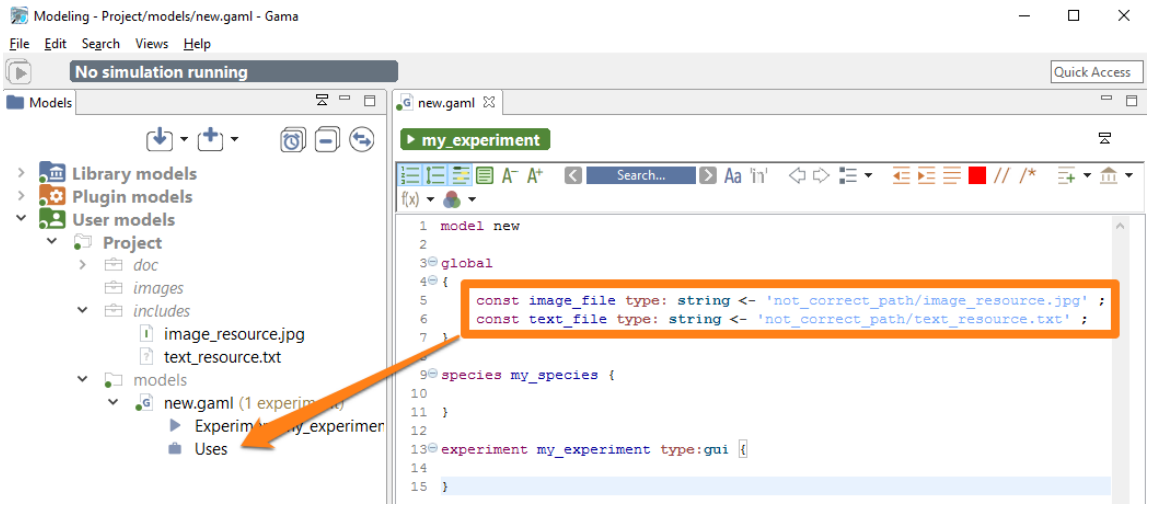


Figure 15.6: images/navigator_user_expanded.png

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



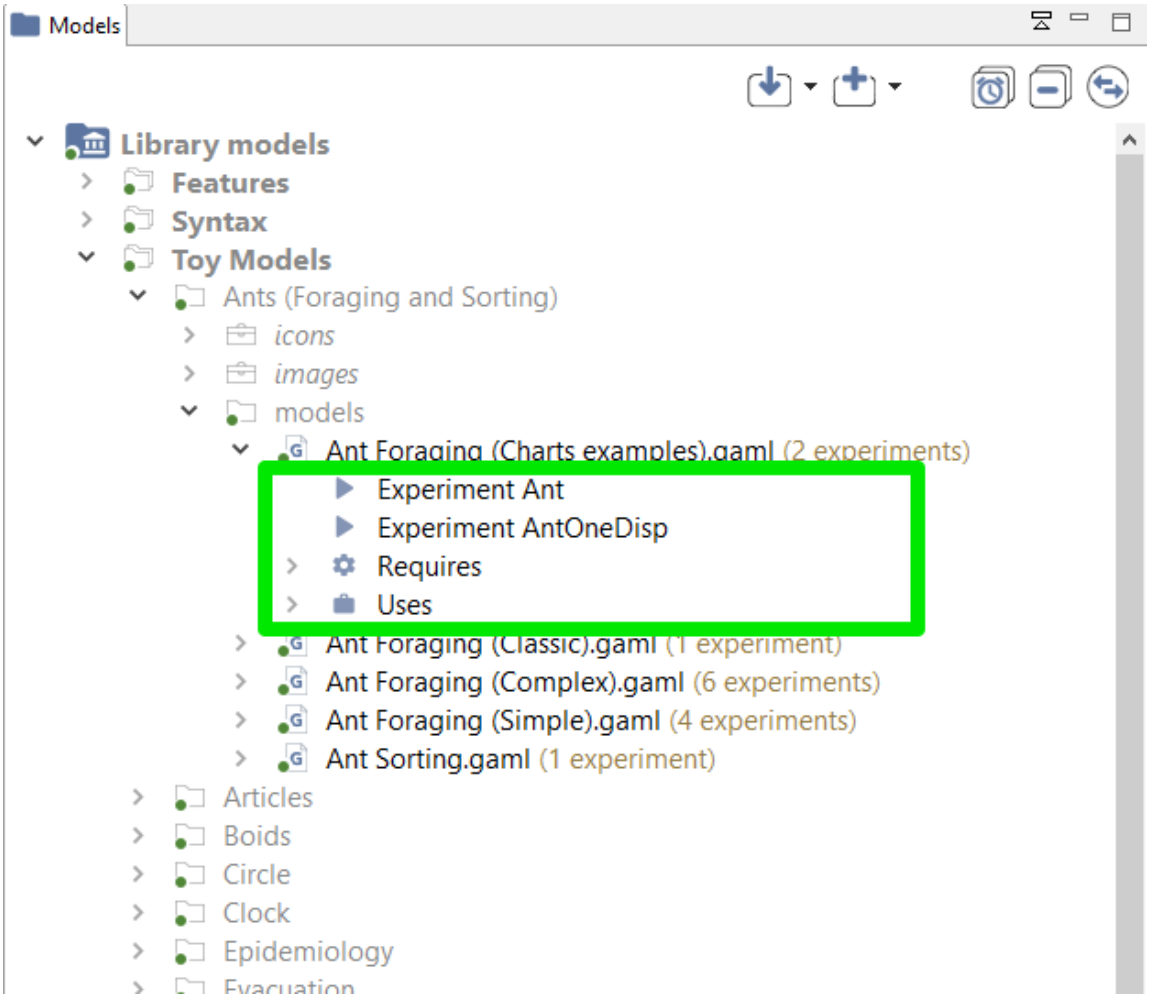


Figure 15.7: images/inspect_model.png

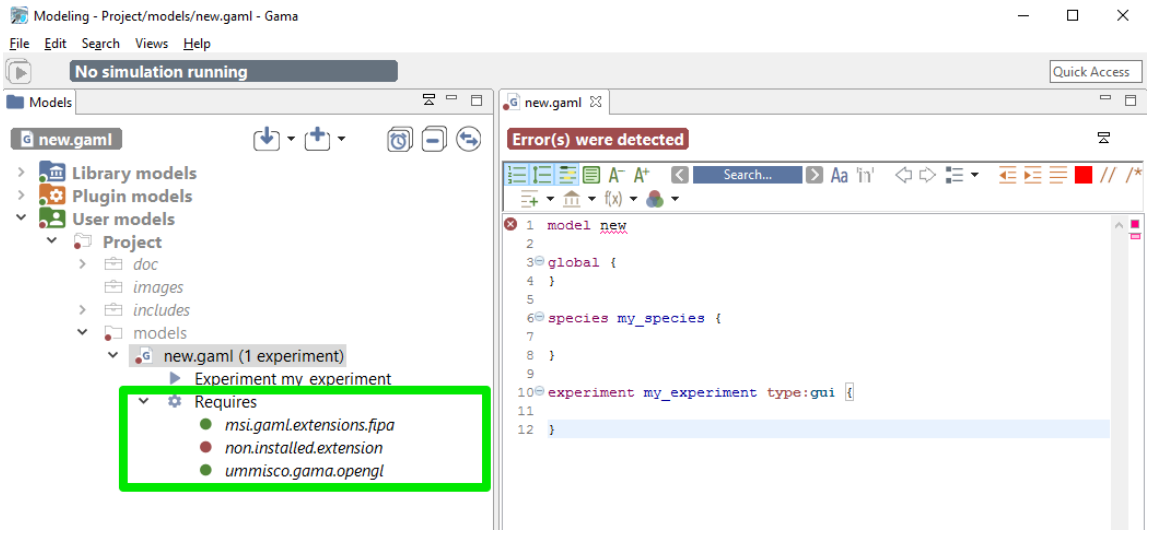
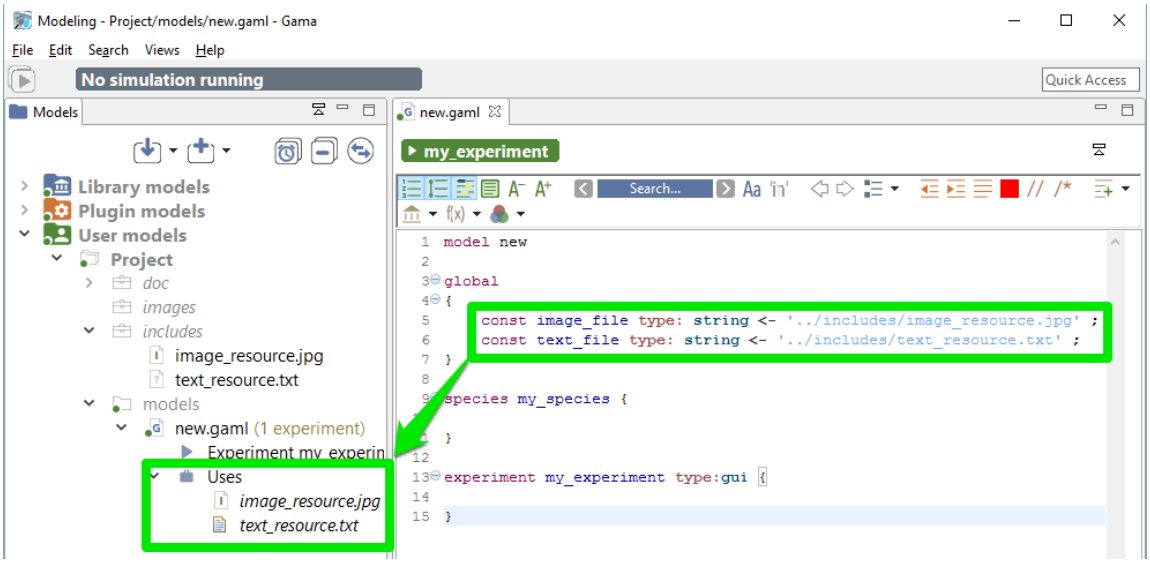


Figure 15.8: images/requires_plugin_not_found.png



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

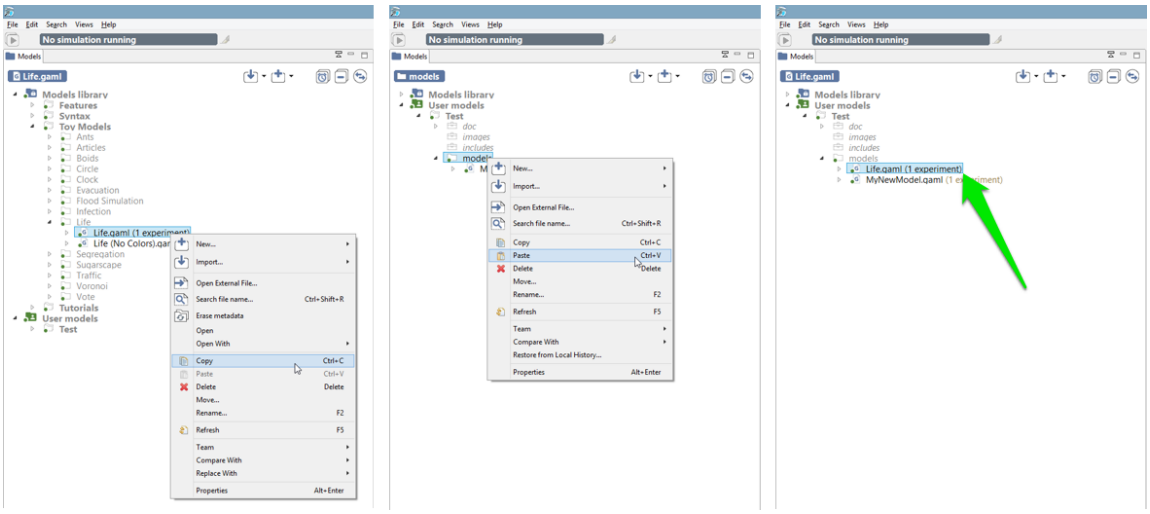
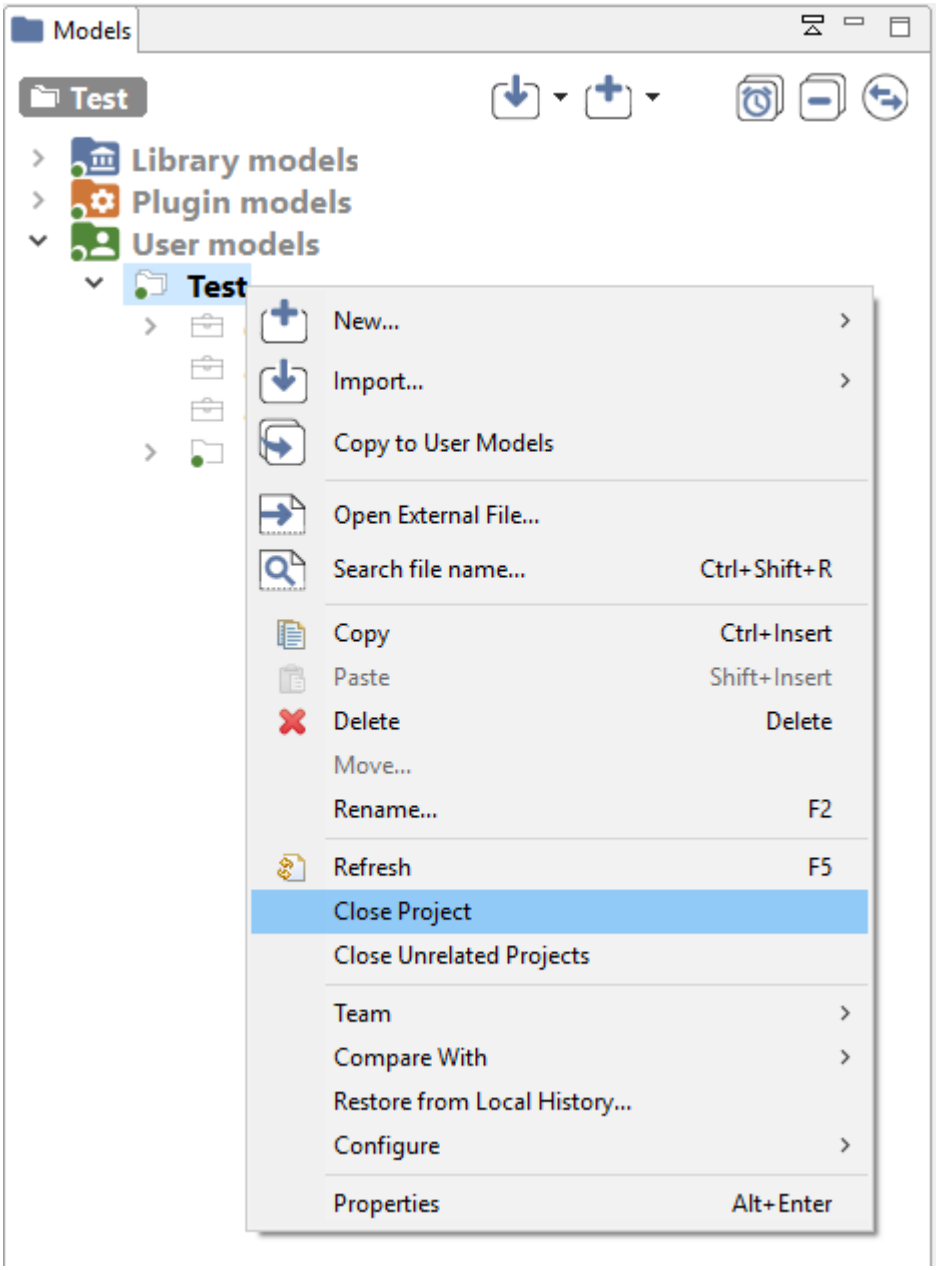


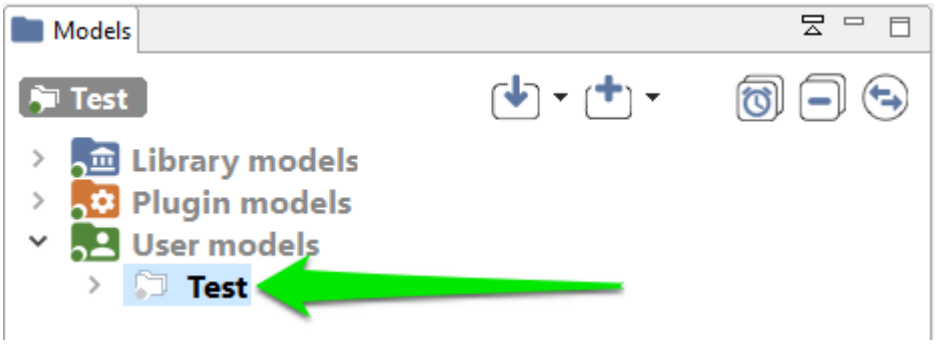
Figure 15.9: images/navigator_menu_copy_paste.png

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.

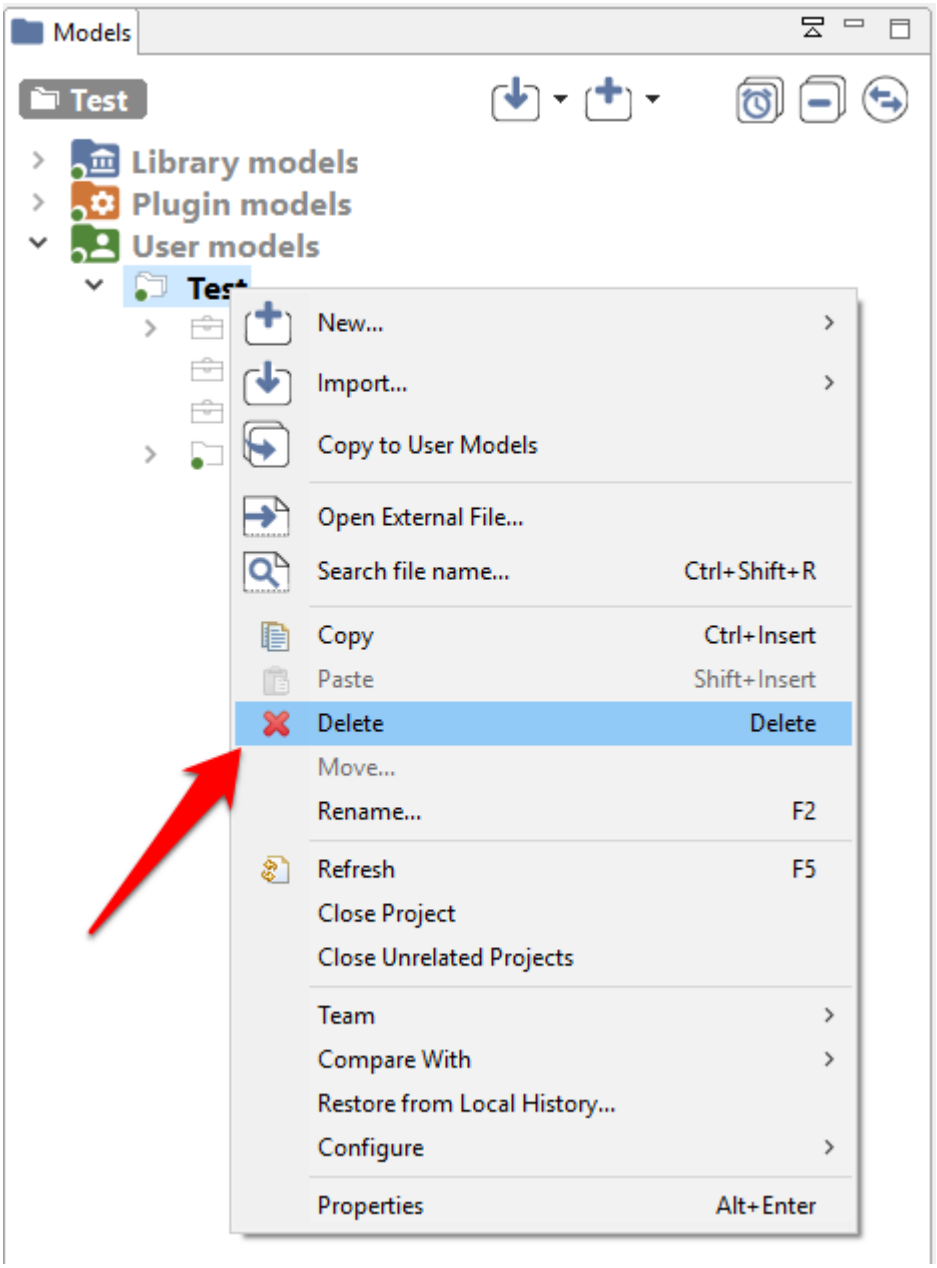
Closing and Deleting Projects

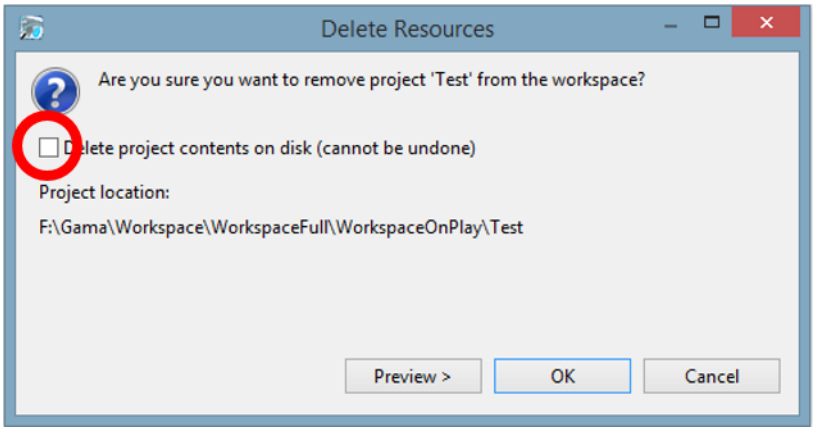
Users can choose to get rid of old projects by either **closing** or **deleting** them. Closing a project means that it will still reside in the workspace (and be still visible, although a bit differently, in the *Navigator*) but its model(s) won't participate to the build process and won't be displayable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


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



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


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


Delete Resources
Are you sure you want to remove project 'Test' from the workspace?
Delete project contents on disk (cannot be undone)
Project location:
F:\Gama\Workspace\WorkspaceFull\WorkspaceOnPlay\Test
Preview >
OK
Cancel

Chapter 16

Changing Workspace

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

Table of contents

Switching to another Workspace

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

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

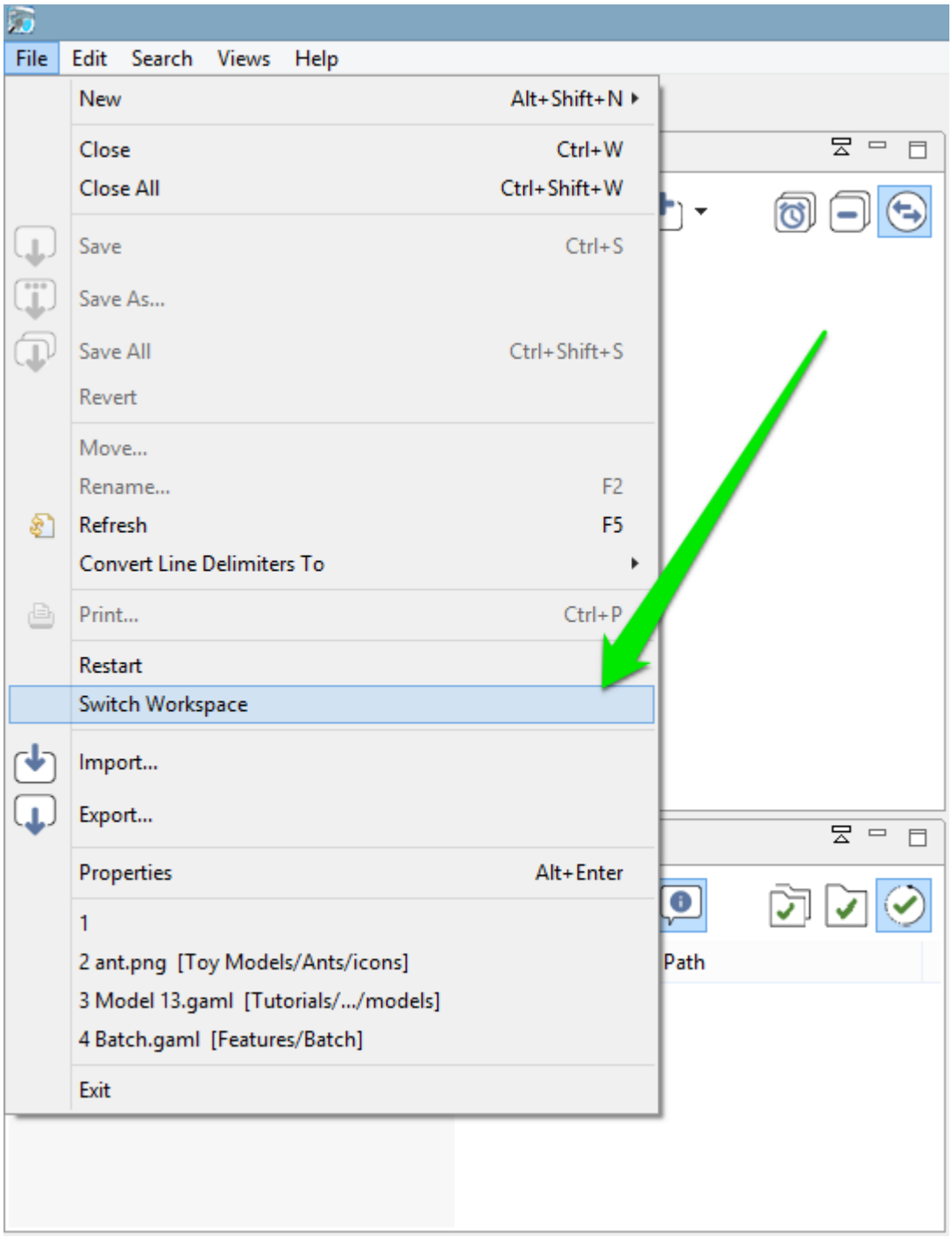


Figure 16.1: images/menu_switch.png

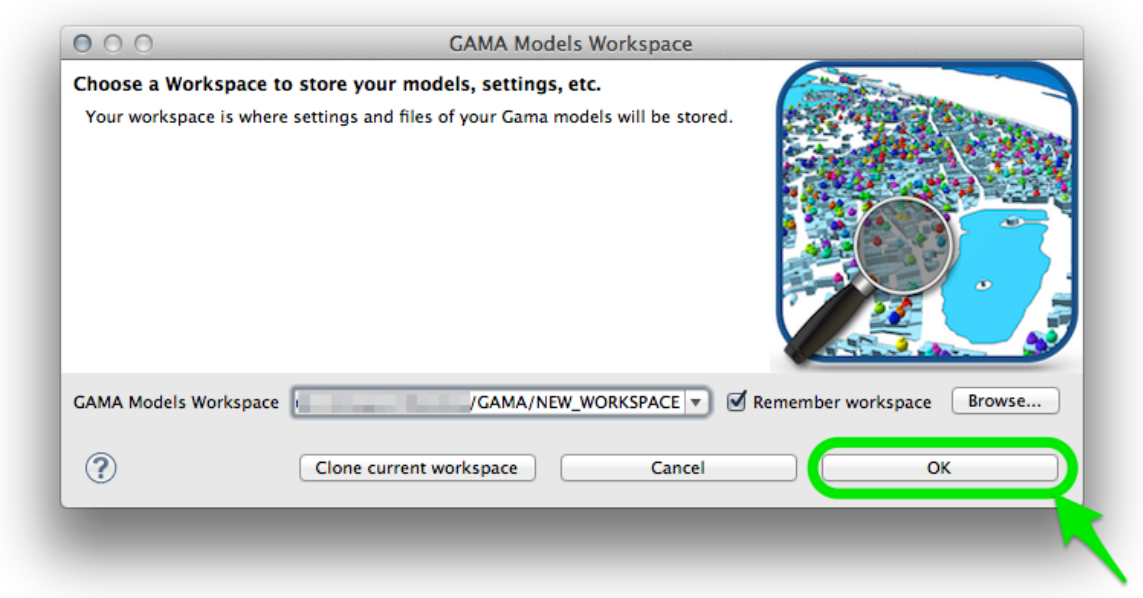


Figure 16.2: images/dialog_switch_ok.png

just repeat this step (the previous location is normally now accessible from the combo box).

Cloning the Current Workspace

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

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

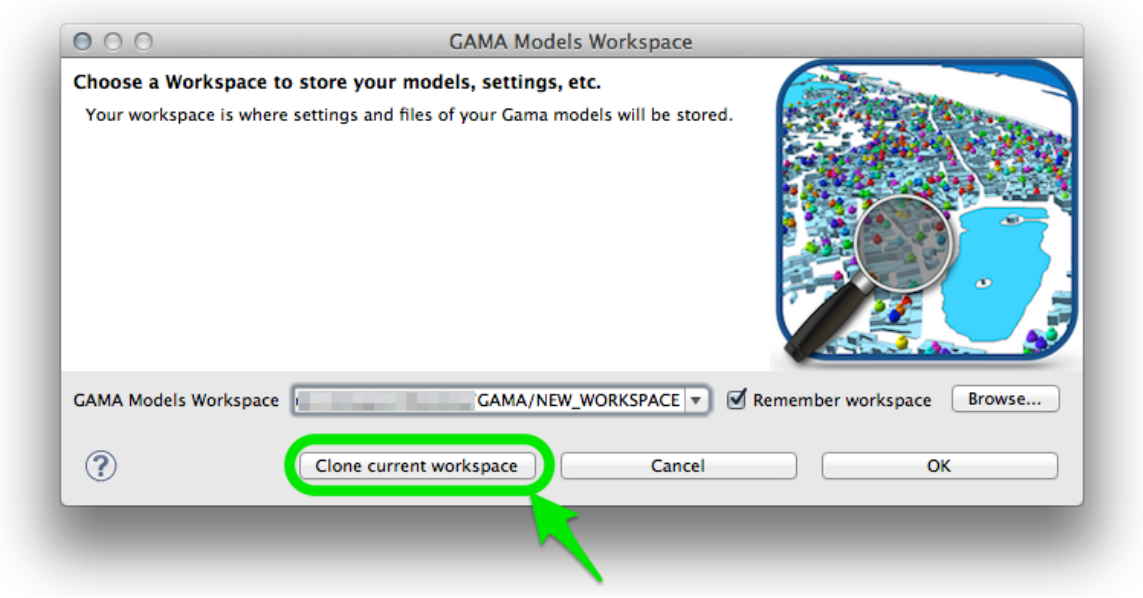


Figure 16.3: images/dialog_switch_clone.png

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).

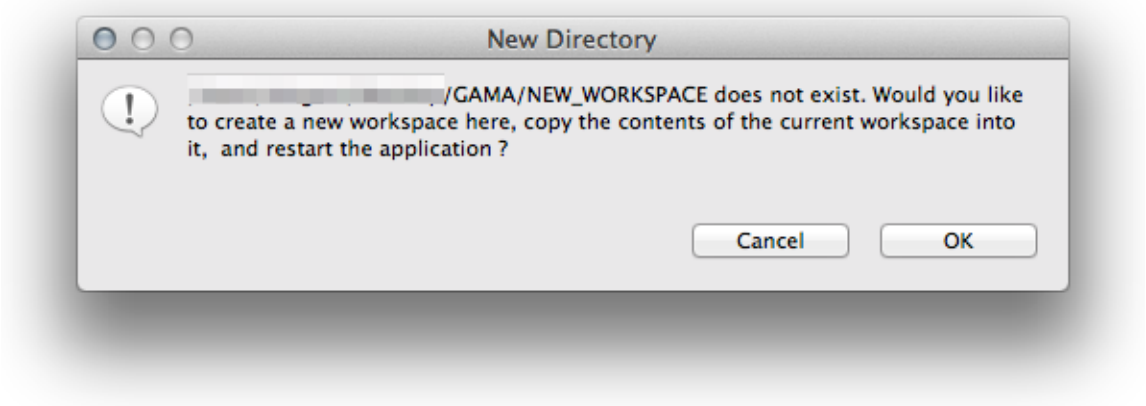


Figure 16.4: images/clone_confirm_new.png

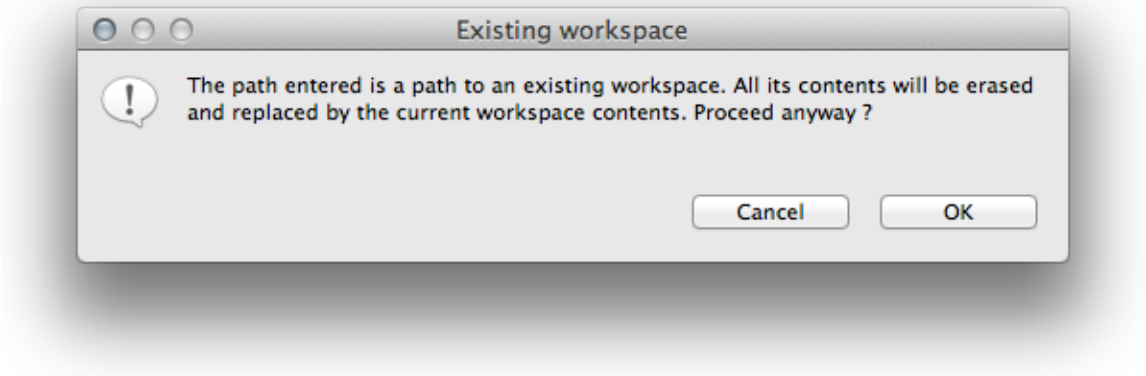


Figure 16.5: images/clone_confirm_existing.png

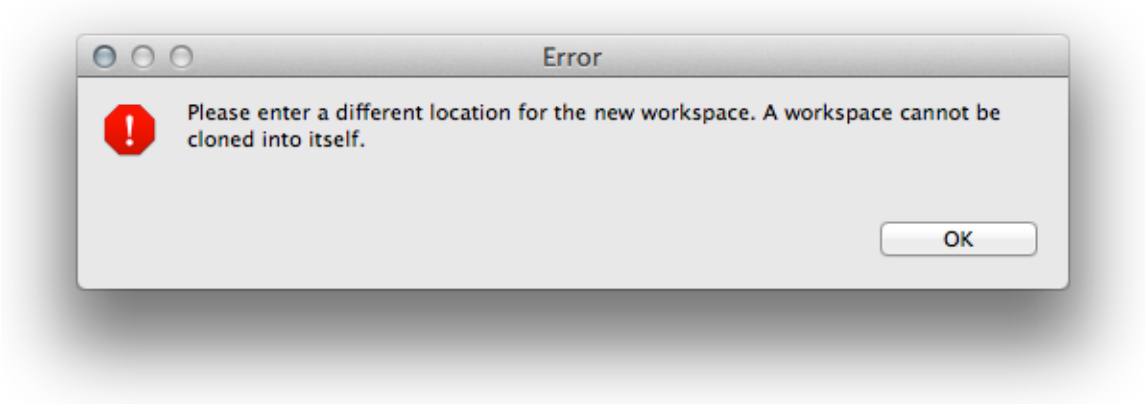


Figure 16.6: images/close_error_same.png

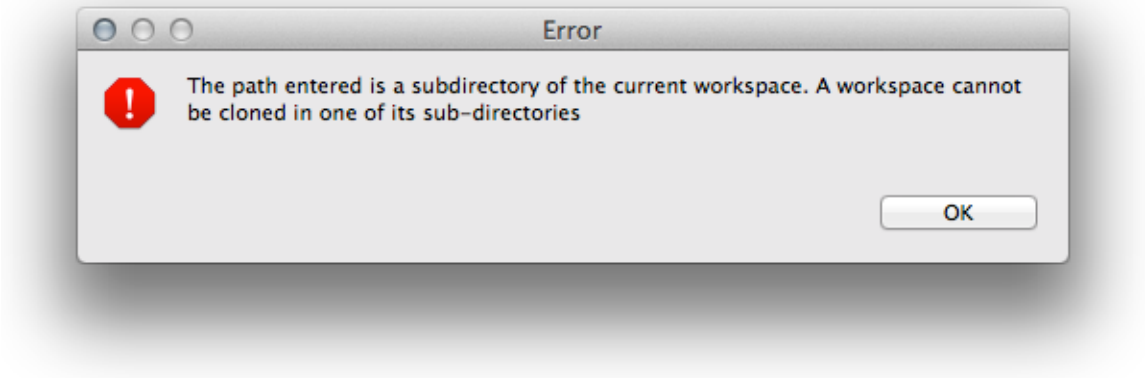


Figure 16.7: images/close_error_subdir.png

Chapter 17

Importing Models

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

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

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

Table of contents

The "Import..." Menu Command

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

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

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

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

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

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),

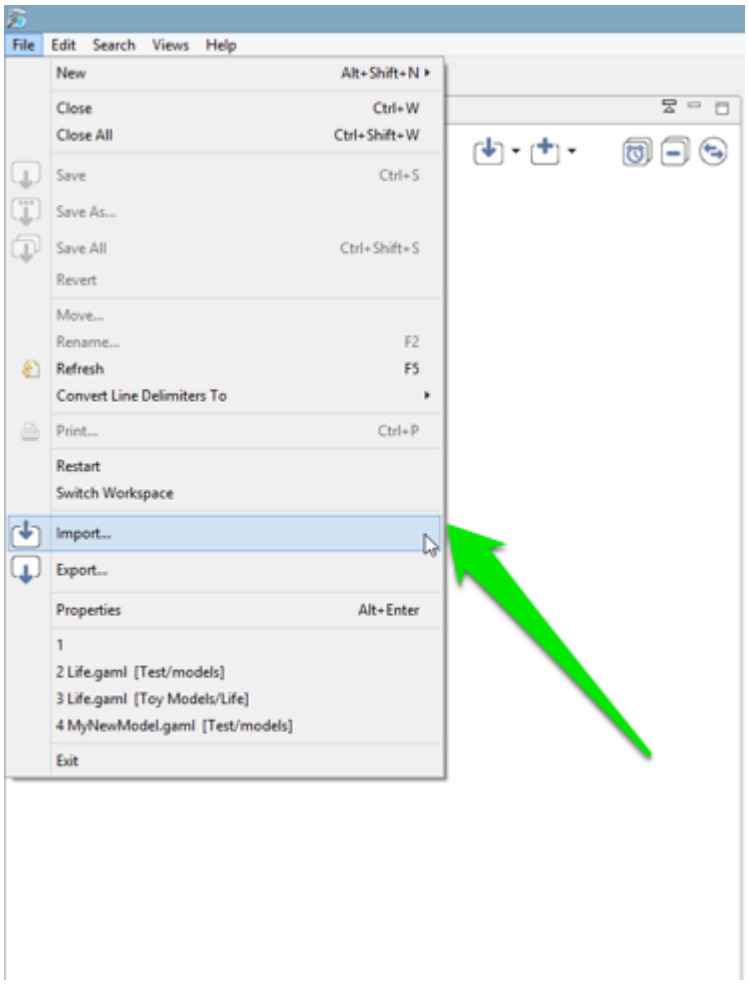


Figure 17.1: images/menu_file_import.png

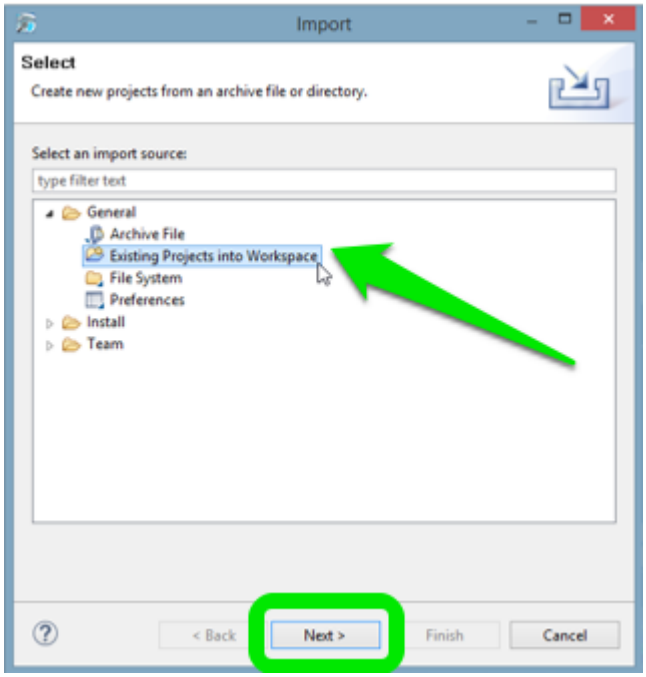


Figure 17.2: images/dialog_import.png

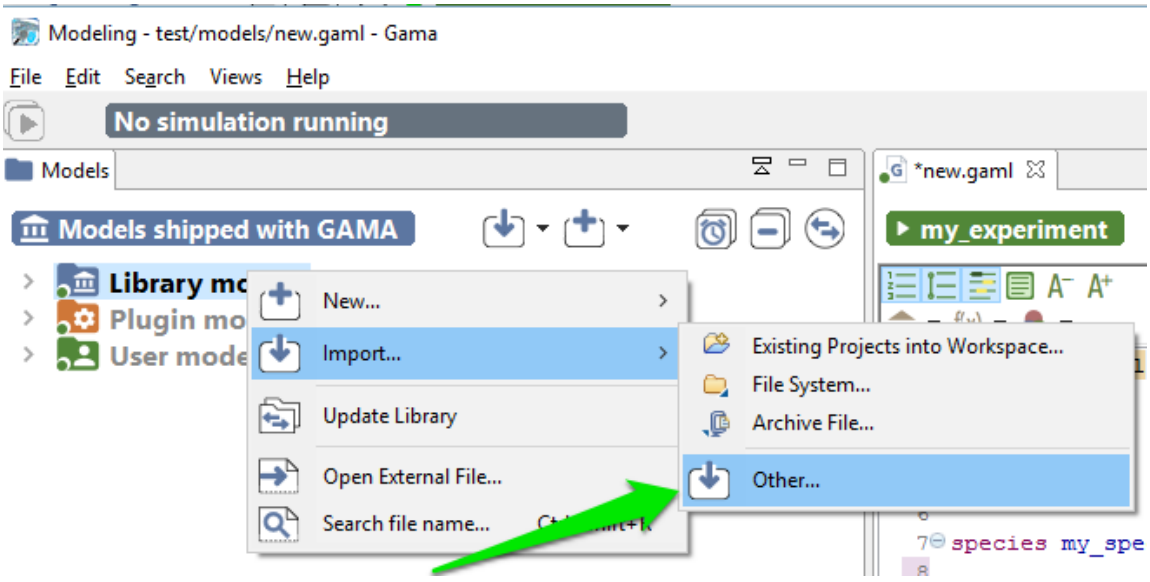


Figure 17.3: images/menu_navigator_import.png

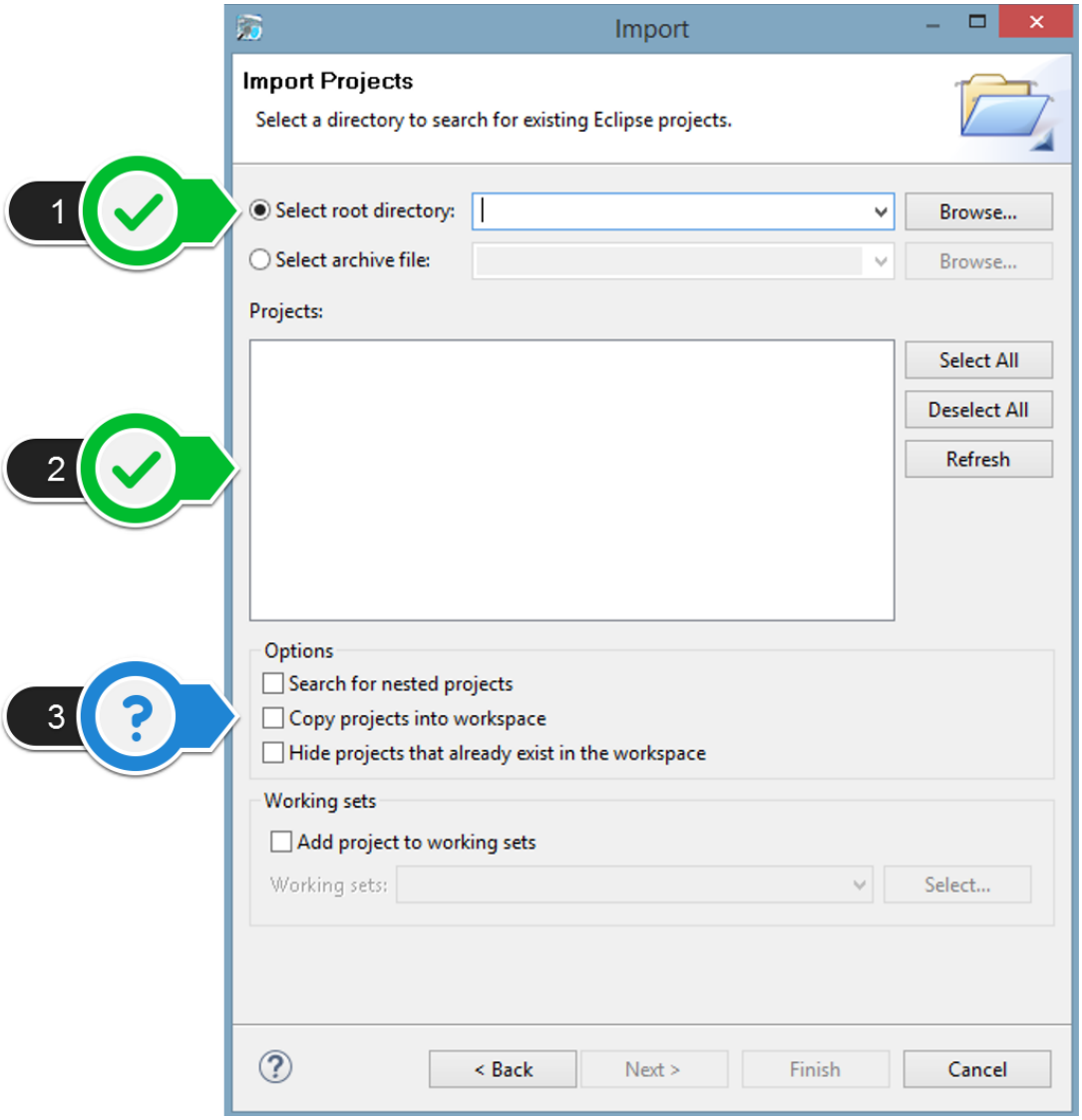


Figure 17.4: images/dialog_import_2.png

2. open an editor on the file selected.

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

Drag'n Drop / Copy-Paste Limitations

Currently, **there is no way** to drag and drop an entire project into GAMA *Navigator* (or to copy a project in the filesystem and paste it in the *Navigator*). Only individual model files, folders or resources can be moved this way (and they have to be dropped or pasted into existing projects).

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

Chapter 18

Editing models

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

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

This section covers the following sub-sections :

Chapter 19

The GAML Editor - Generalities

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

Table of contents

Creating a first model

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

A dialog is then displayed, offering you to enter the name of the project as well as its location on the filesystem. Unless you are absolutely sure of what you are doing, keep

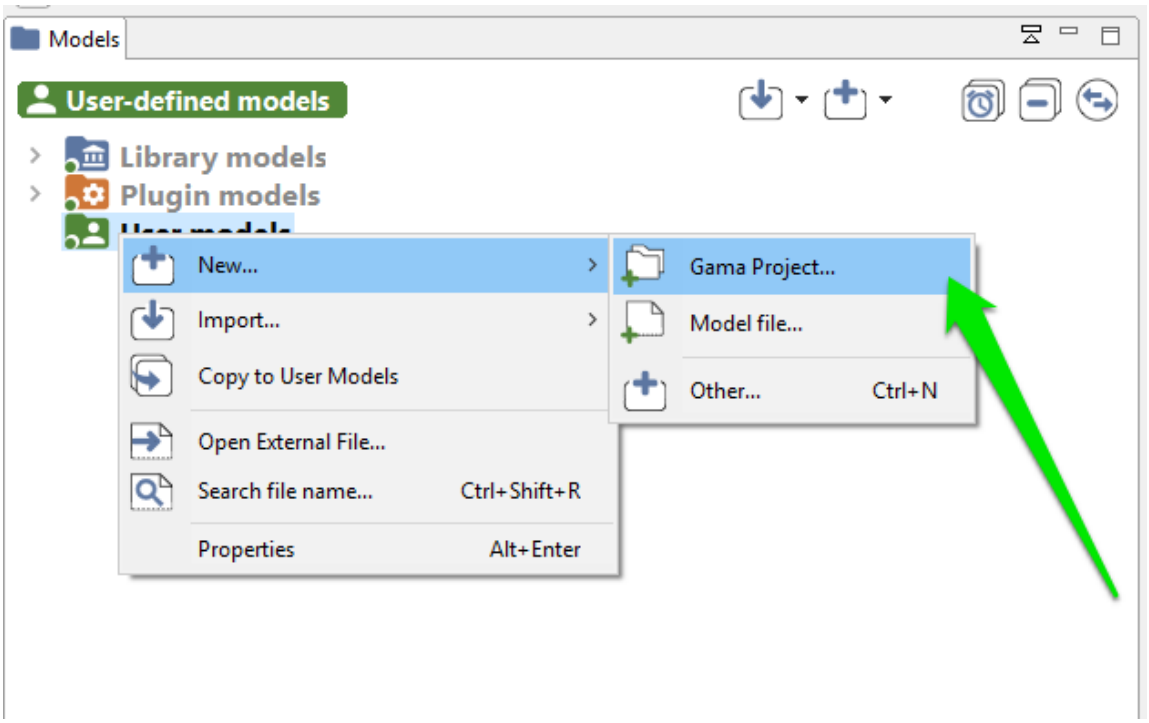


Figure 19.1: images/1.new_project.png

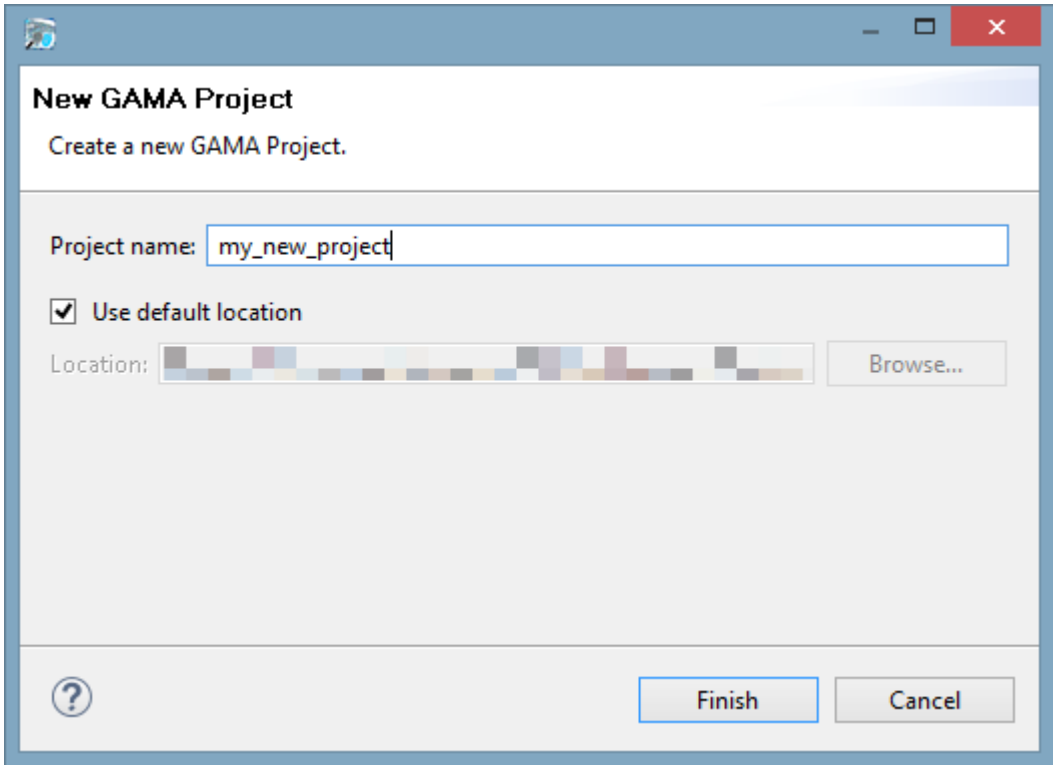


Figure 19.2: images/2.new_project2.png

the "Use default location" option checked. An error will be displayed if the project name already exists in the workspace, in which case you will have to change it. Two projects with similar names can not coexist in the workspace (even if they belong to different categories).

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

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

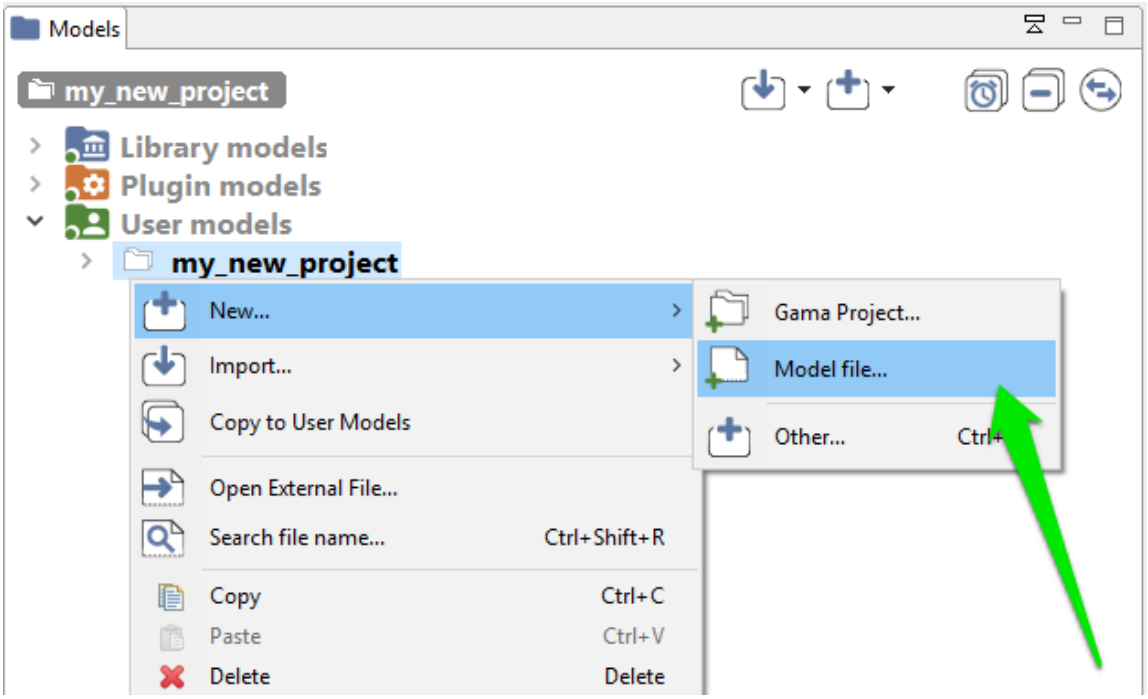


Figure 19.3: images/3.new_model.png

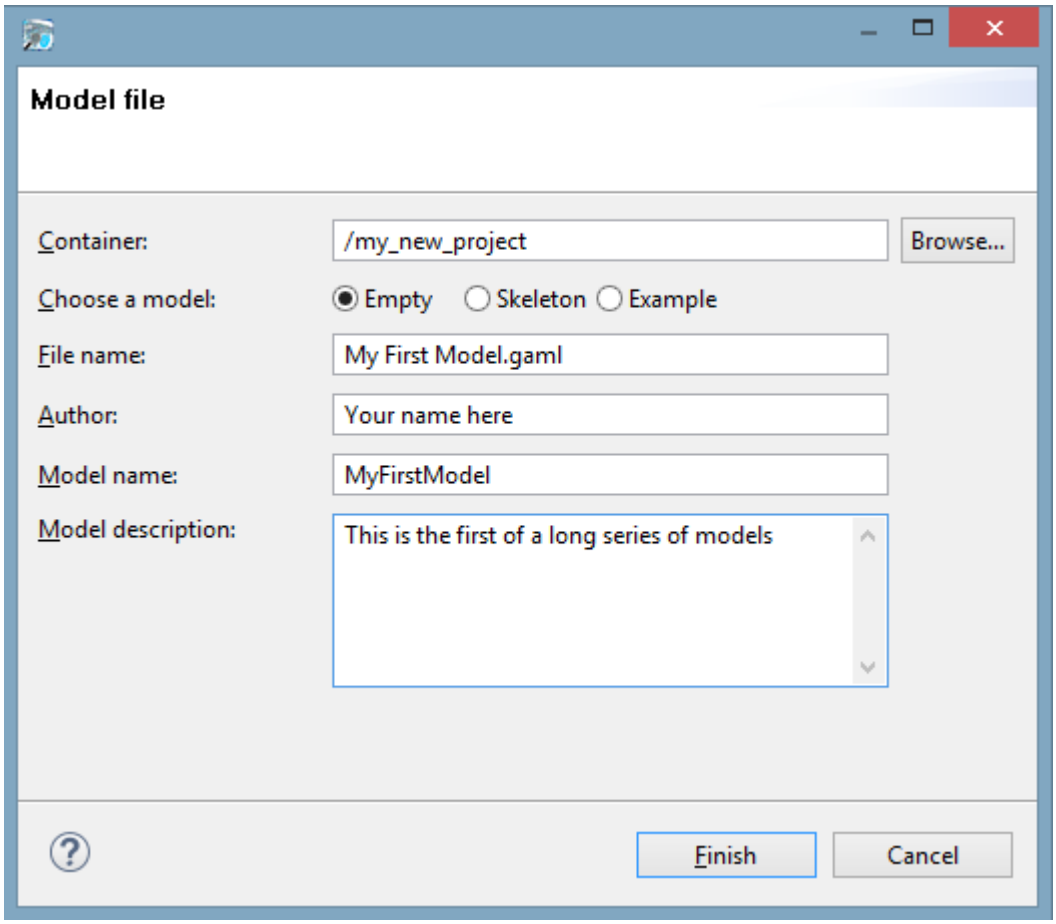


Figure 19.4: images/4.new_model2.png

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

Status of models in editors

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

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

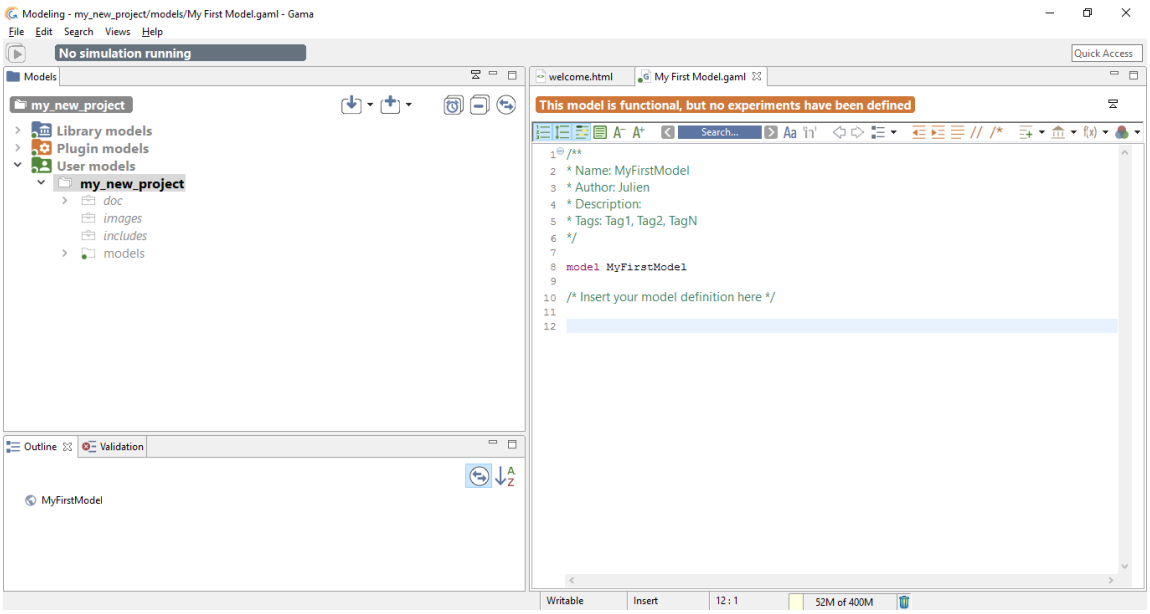


Figure 19.5: images/5.view_model.png

actually be in four different states, which are visually accessible above the editing area: *Functional* (orange color), *Experimentable* (green color), *InError* (red color), InImportedError_(yellow color). See the section on model compilation for more precise information about these statuses._

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

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

Reaching the *Experimentable* state requires that all errors are eliminated and that at least one experiment is defined in the model, which is the case now in our toy model. The experiment is immediately displayed as a button in the toolbar, and clicking on it will allow to launch this experiment on your model. See the section about running experiments for more information on this point.

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

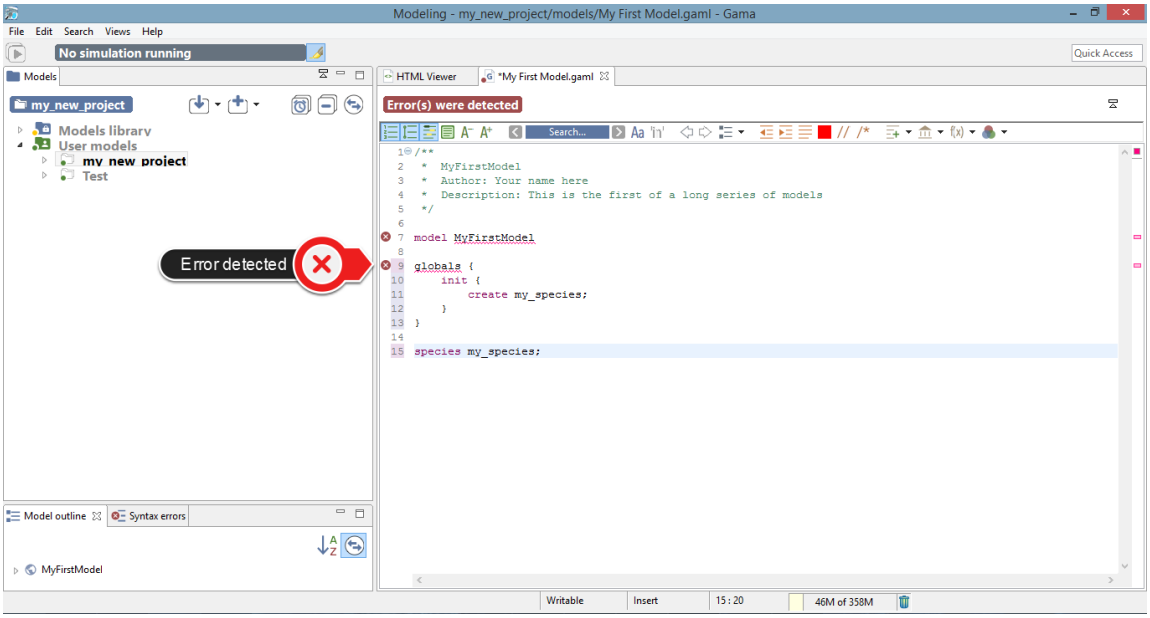


Figure 19.6: images/6.view_model_with_error.png

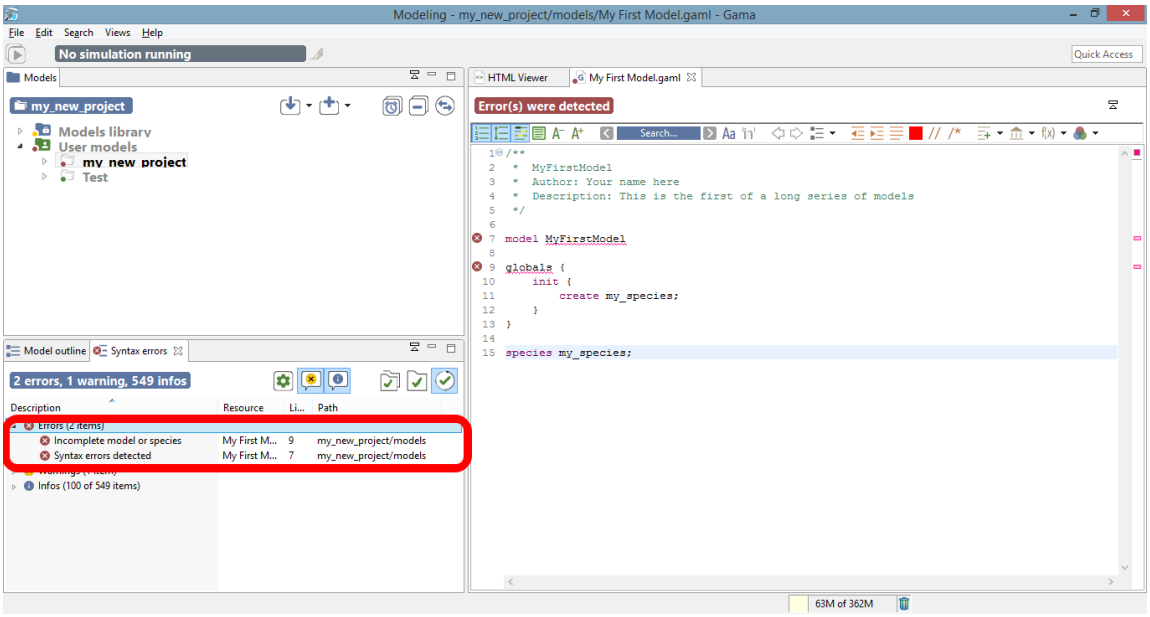


Figure 19.7: images/7.view_model_with_error_saved.png

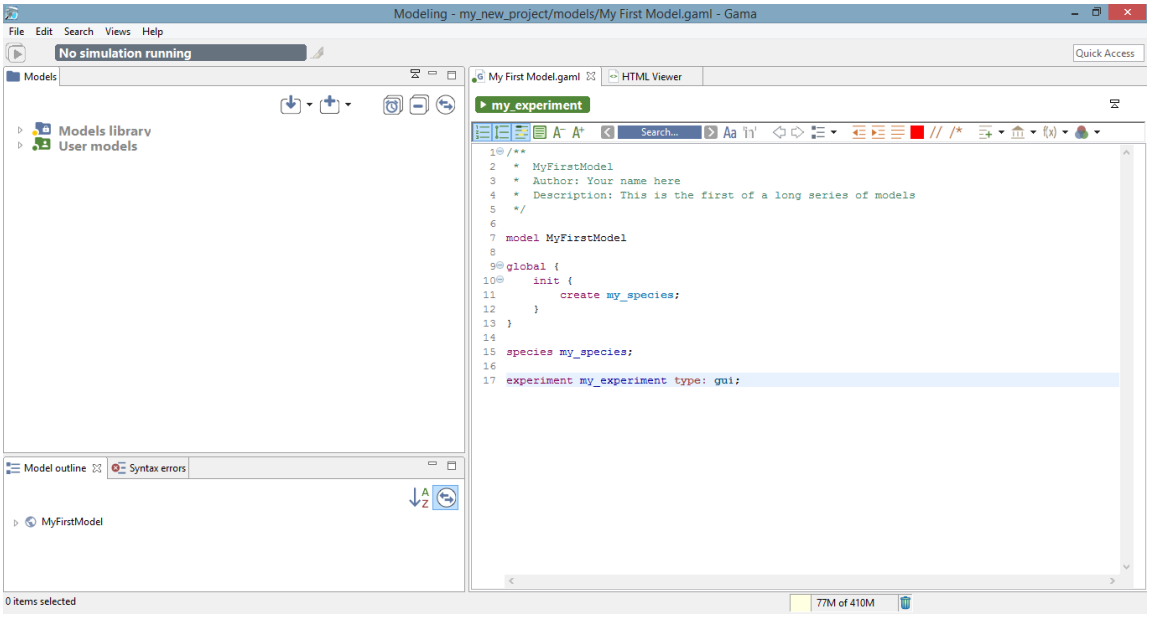


Figure 19.8: images/8.view_model_with_experiment.png

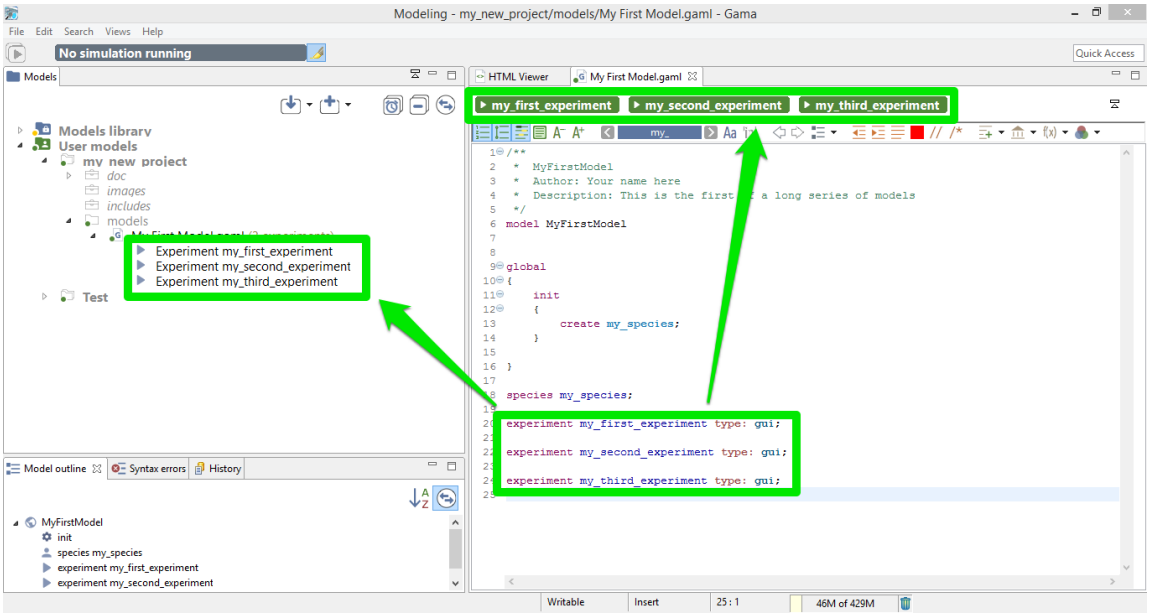


Figure 19.9: images/9.view_model_with_3_experiments.png

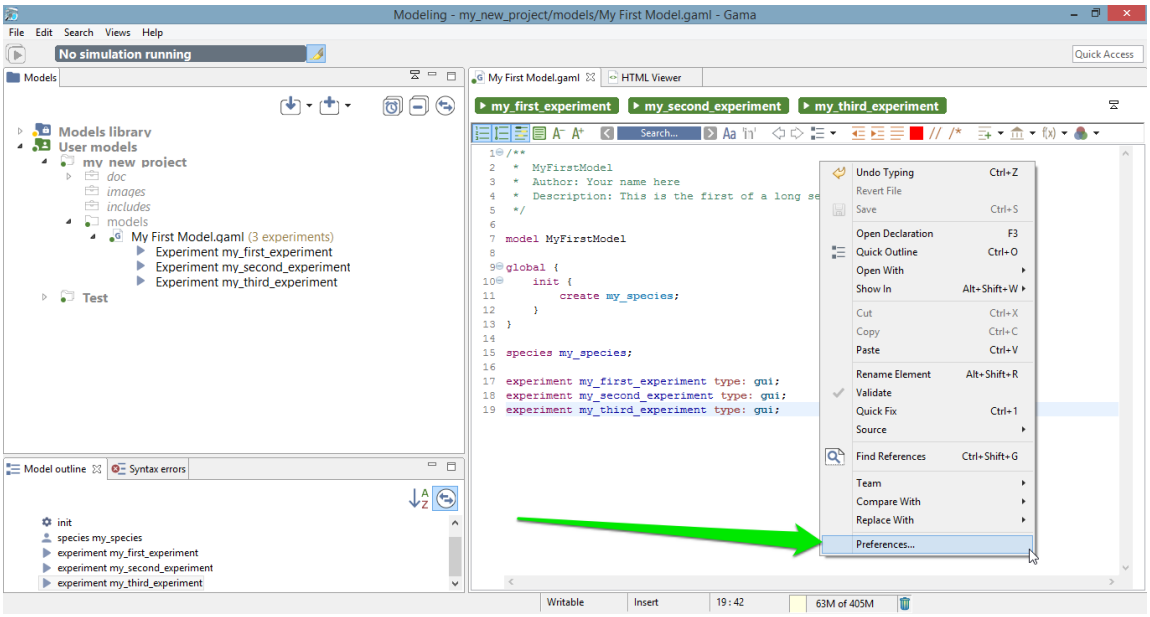


Figure 19.10: images/10.view_model_with_preferences.png

Editor Preferences

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

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

Additional informations in the Editor

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

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

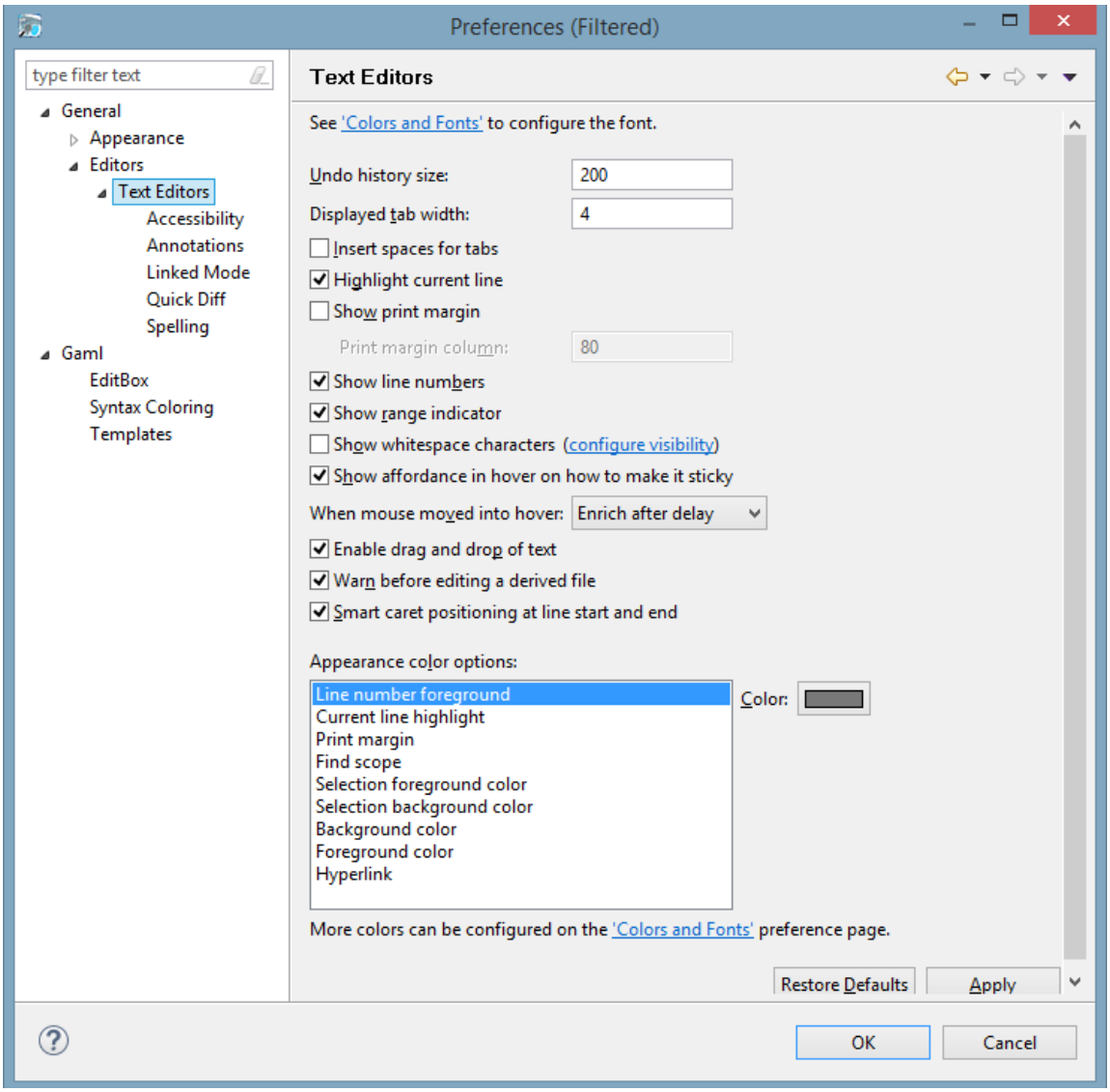


Figure 19.11: images/11.editor_preferences.png

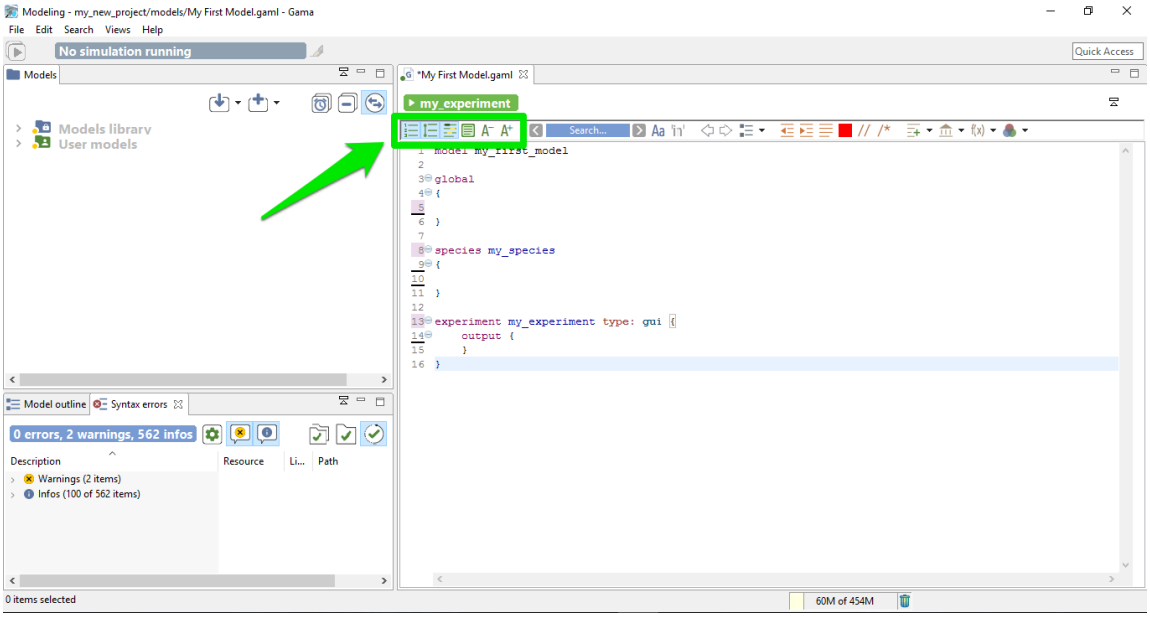


Figure 19.12: images/additional_informations_in_editor.png

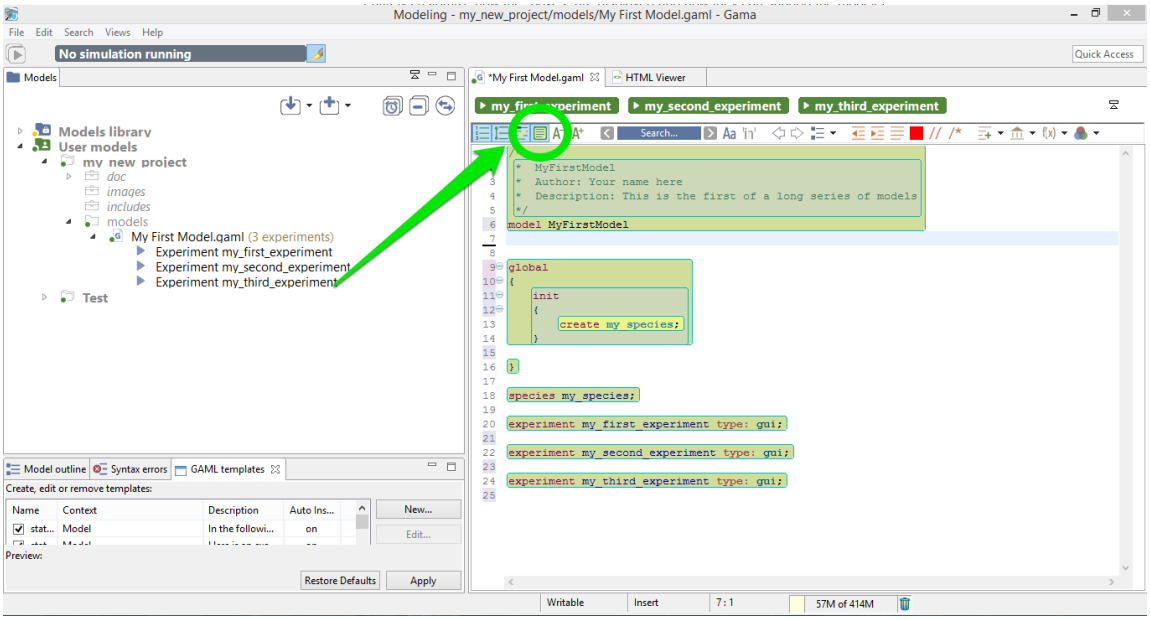


Figure 19.13: images/12.view_model_with_editbox_default.png

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

Multiple editors

GAMA inherits from Eclipse the possibility to entirely configure the placement of the views, editors, etc. This can be done by rearranging their position using the mouse (click and hold on an editor's title and move it around). In particular, you can have several editors side by side, which can be useful for viewing the documentation while coding a model.

Local history

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

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

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

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

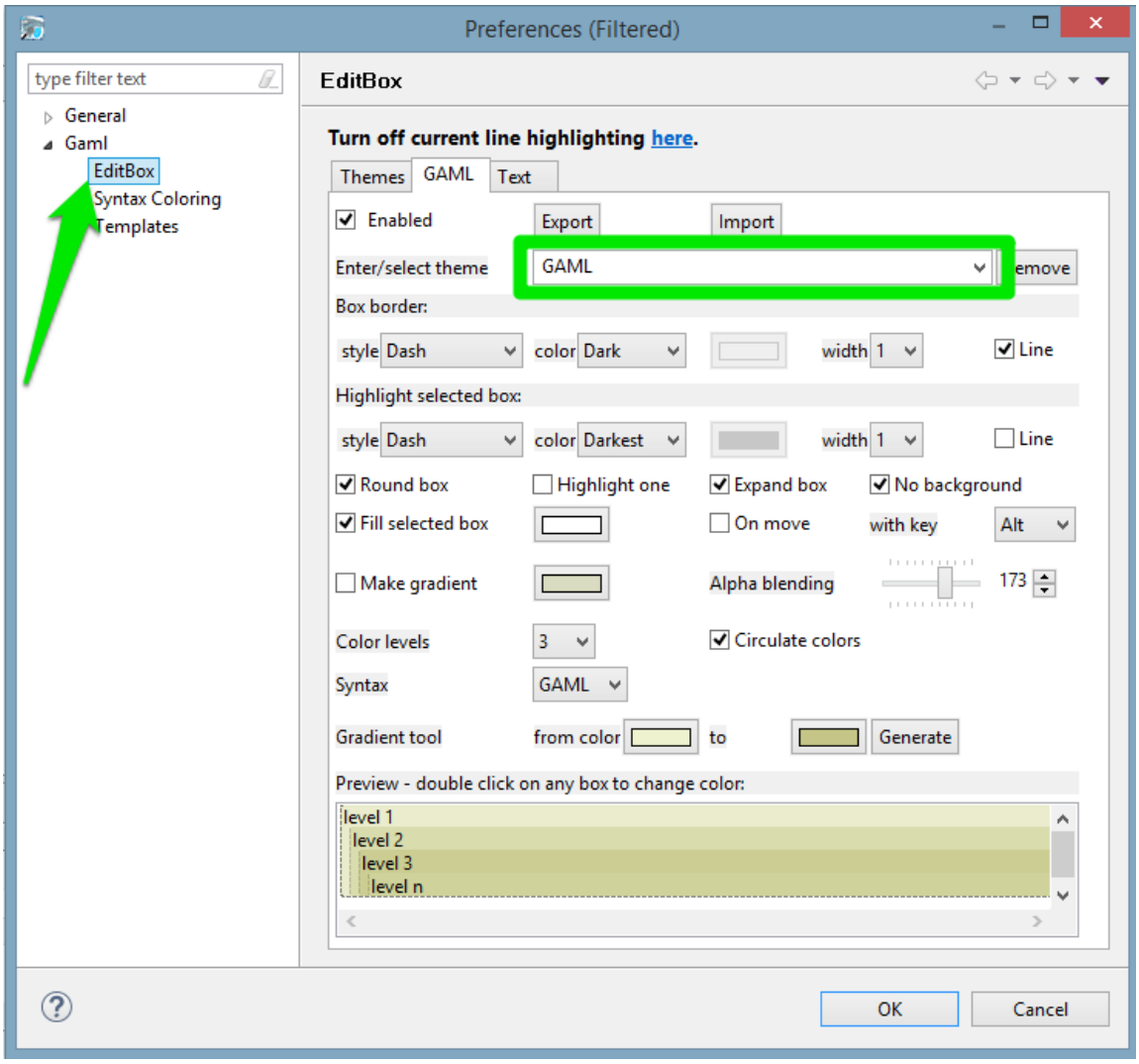


Figure 19.14: images/13.editbox_preferences.png

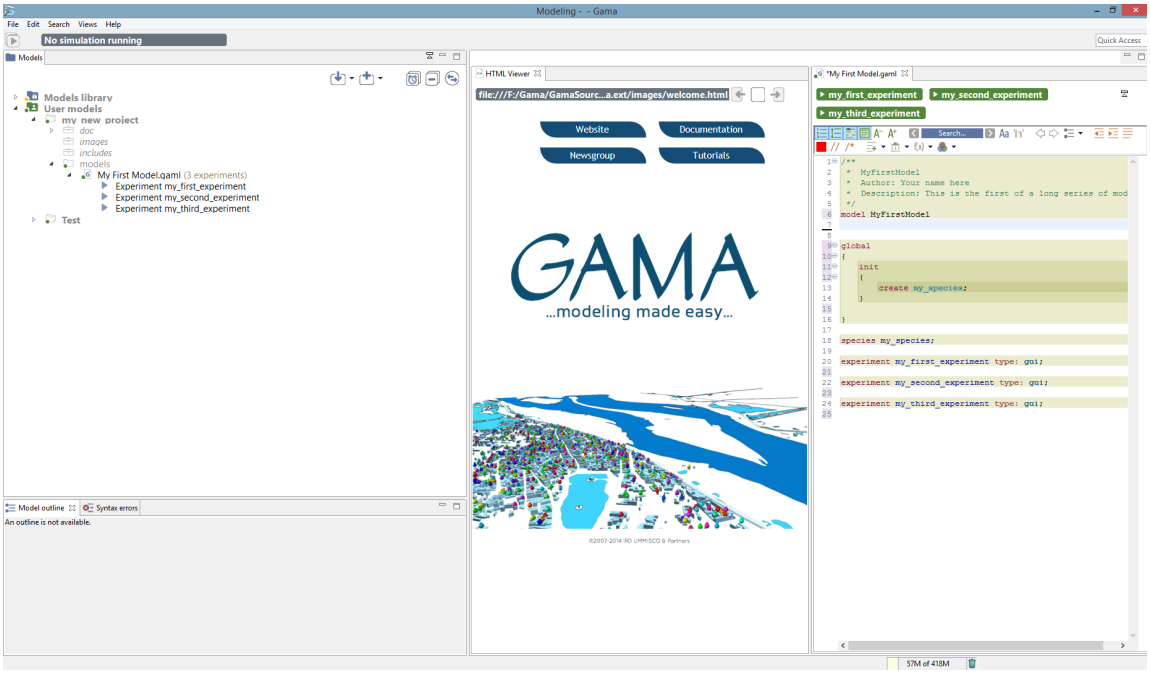


Figure 19.15: images/14.view_model_side_by_side.png

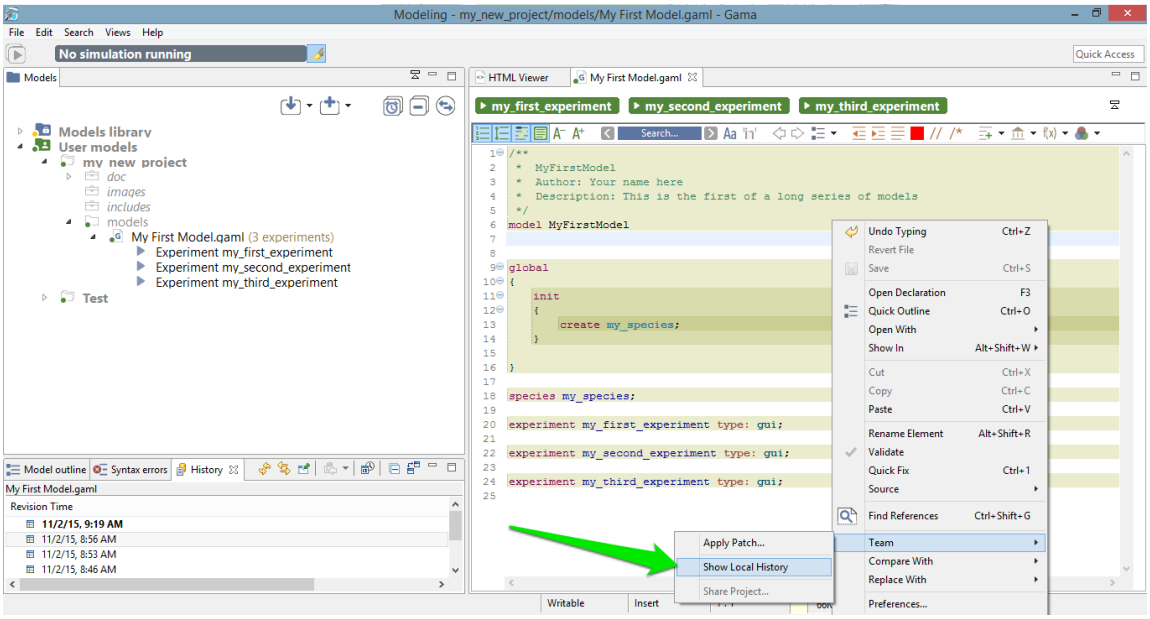


Figure 19.16: images/15.view_model_with_local_history_menu.png

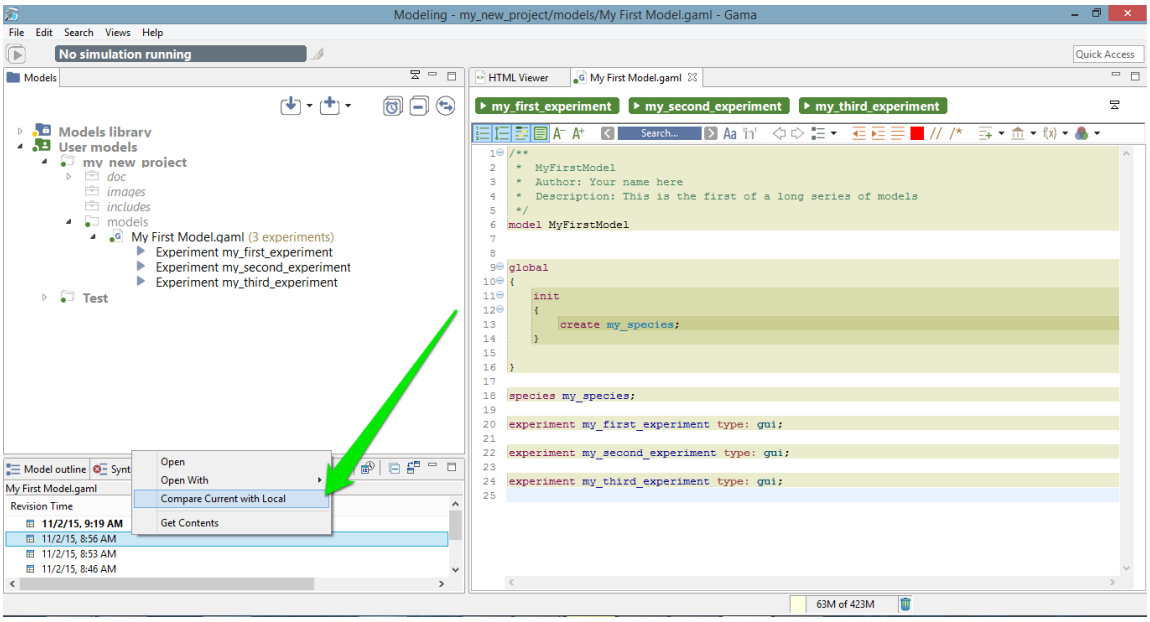


Figure 19.17: images/16.view_model_with_local_history_compare_menu.png

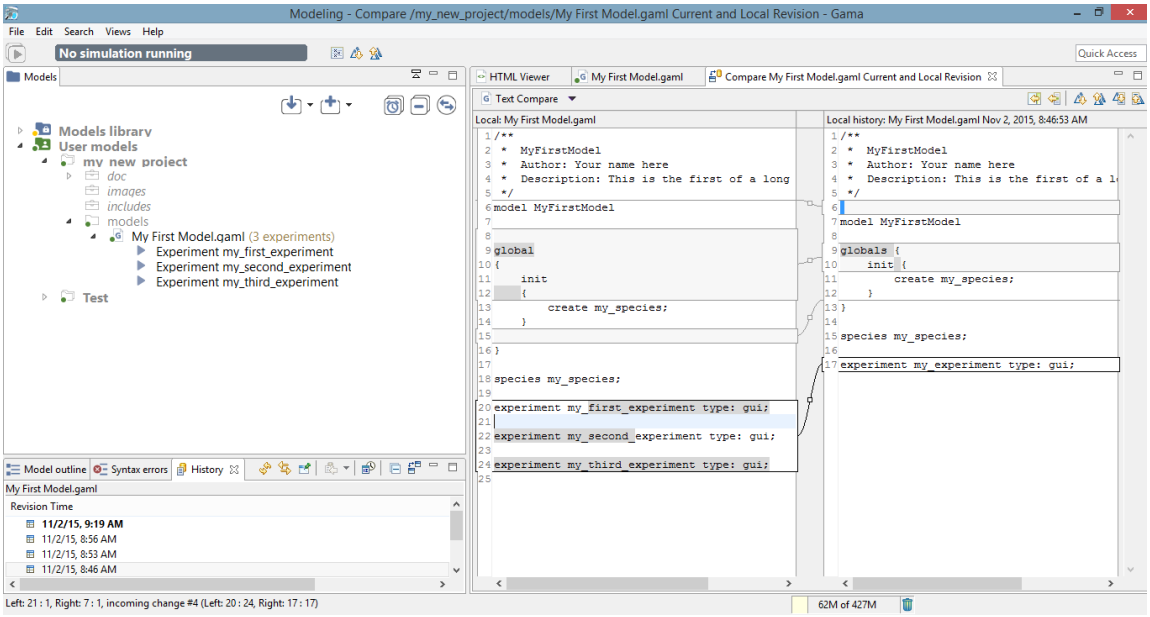


Figure 19.18: images/17.view_model_with_local_history_side_by_side.png

Chapter 20

The GAML Editor Toolbar

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

Table of contents

Visualization tools in the editor

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

Display the number of lines

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

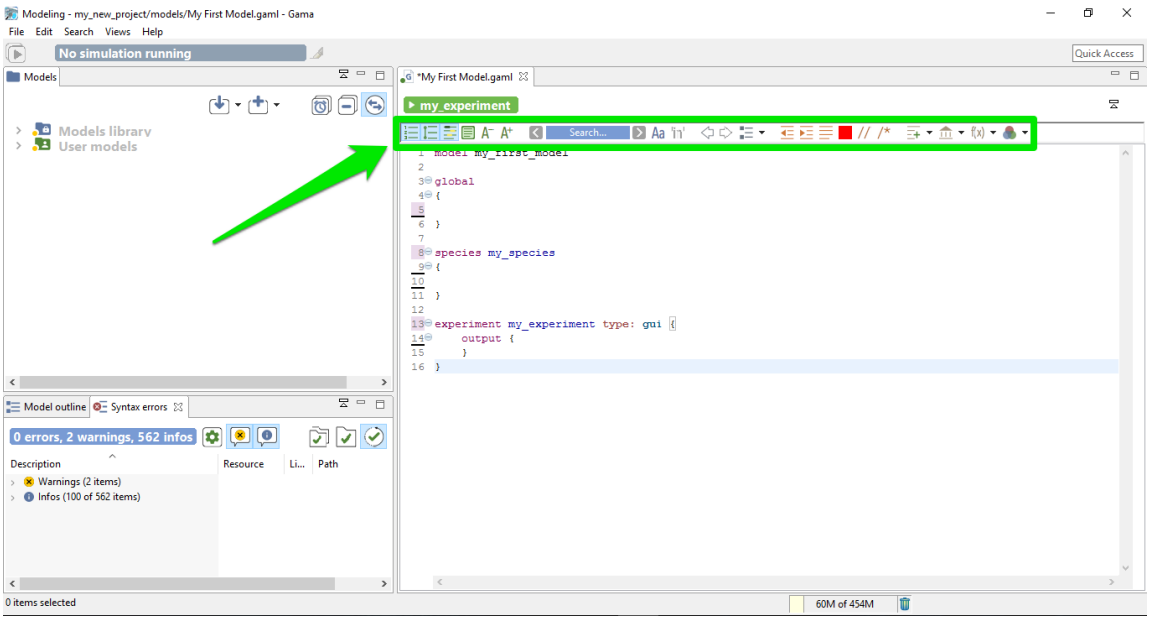


Figure 20.1: images/graphical_editor_toolbar.png

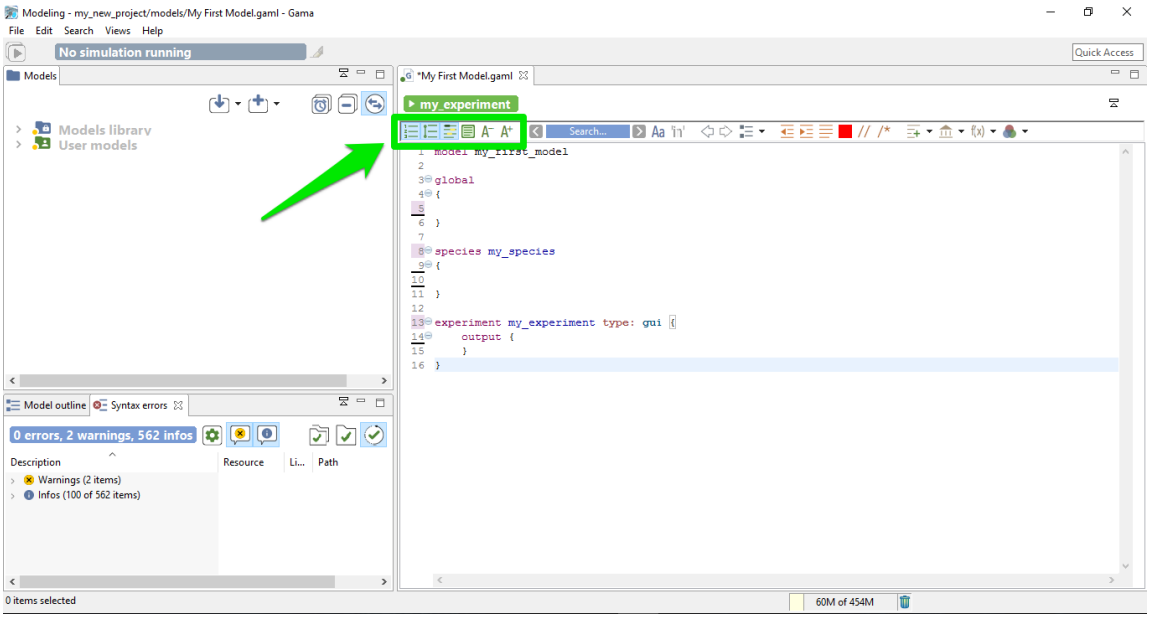


Figure 20.2: images/additional_informations_in_editor.png

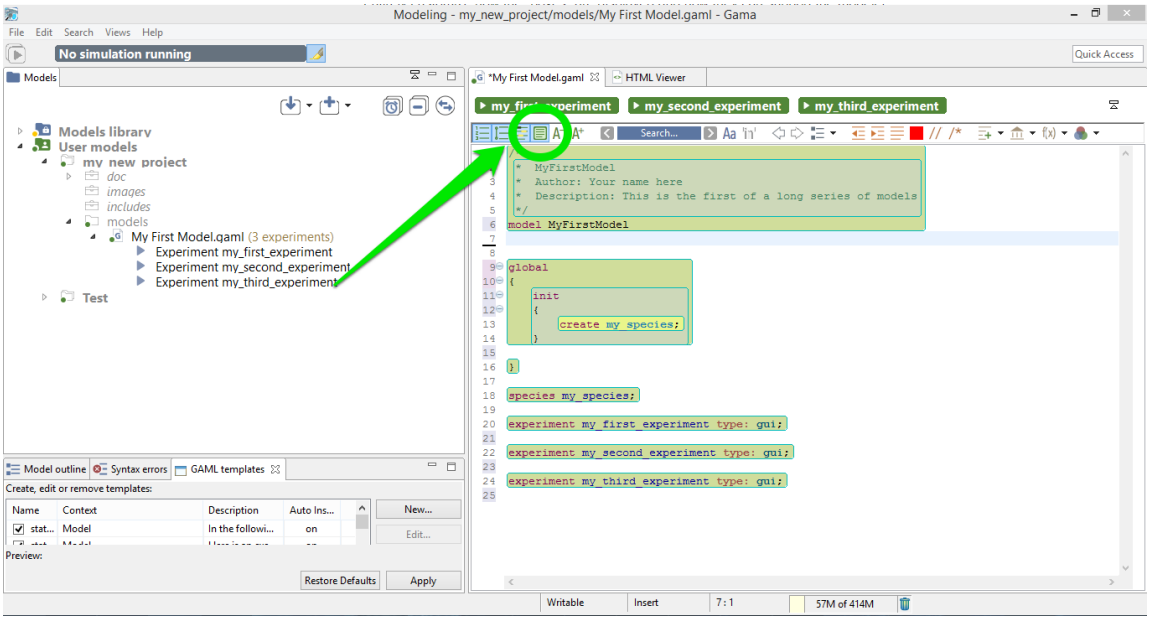


Figure 20.3: images/12.view_model_with_editbox_default.png

Expand / Collapse lines

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

Mark the occurrences

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

Display colorization of code section

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

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.

Change the font size

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

Navigation tools in the editor

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

The search engine

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

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

Previous / Next location

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

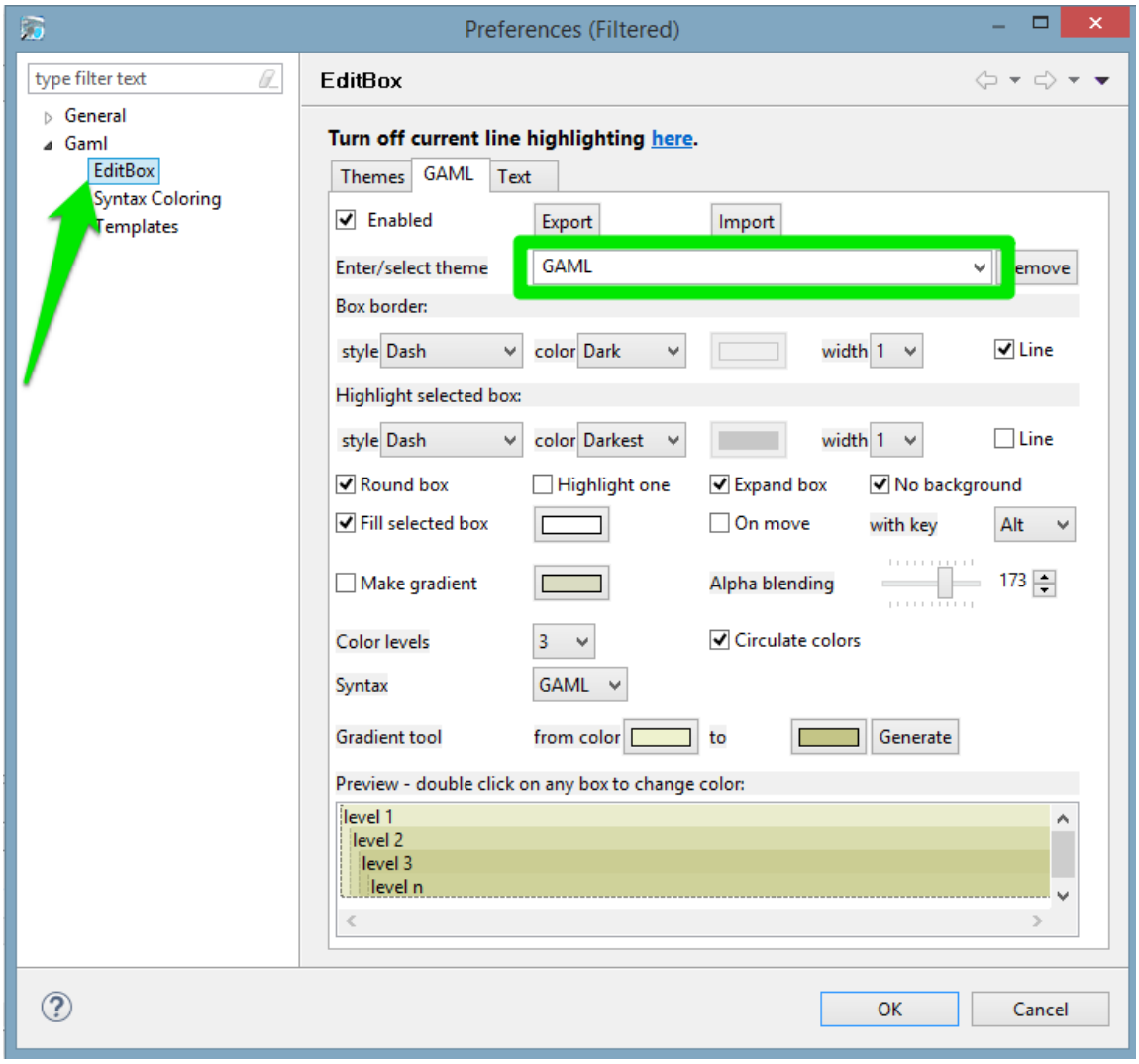


Figure 20.4: images/13.editbox_preferences.png

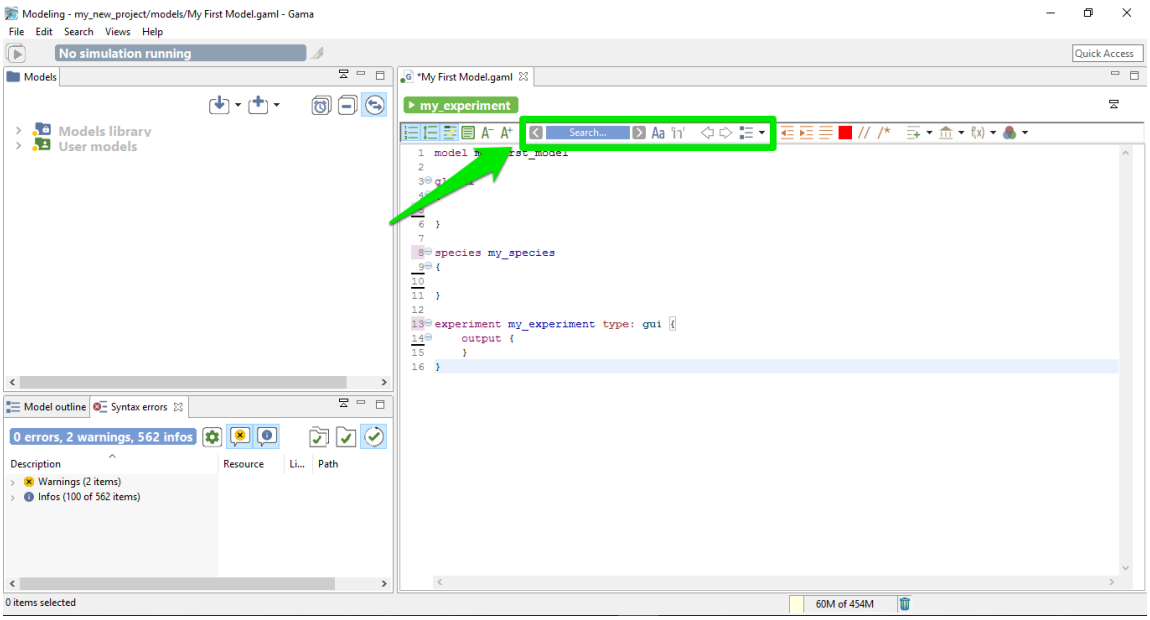


Figure 20.5: images/navigation_in_editor.png

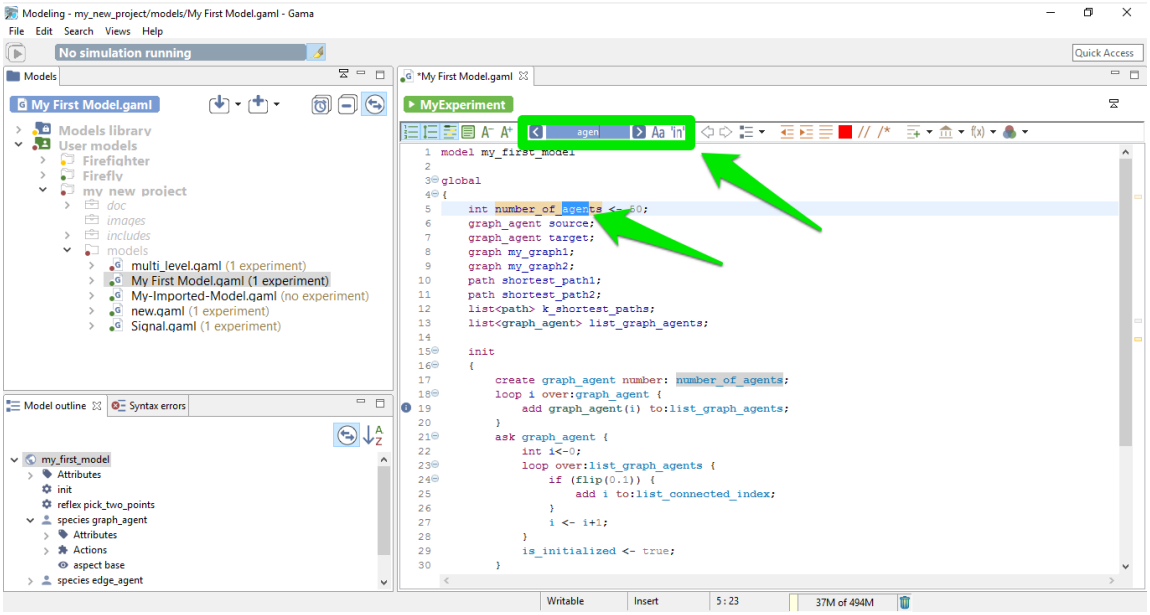


Figure 20.6: images/search_engine.png

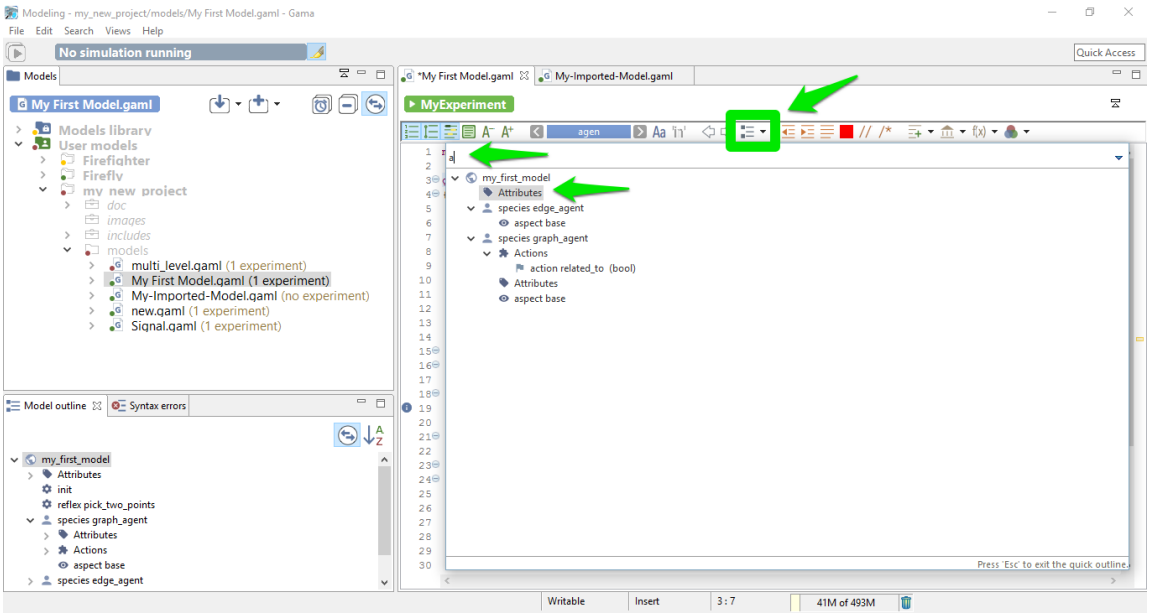


Figure 20.7: images/show_outline.png

Show outline

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

Format tools in the editor

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

Shift left / shift right

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

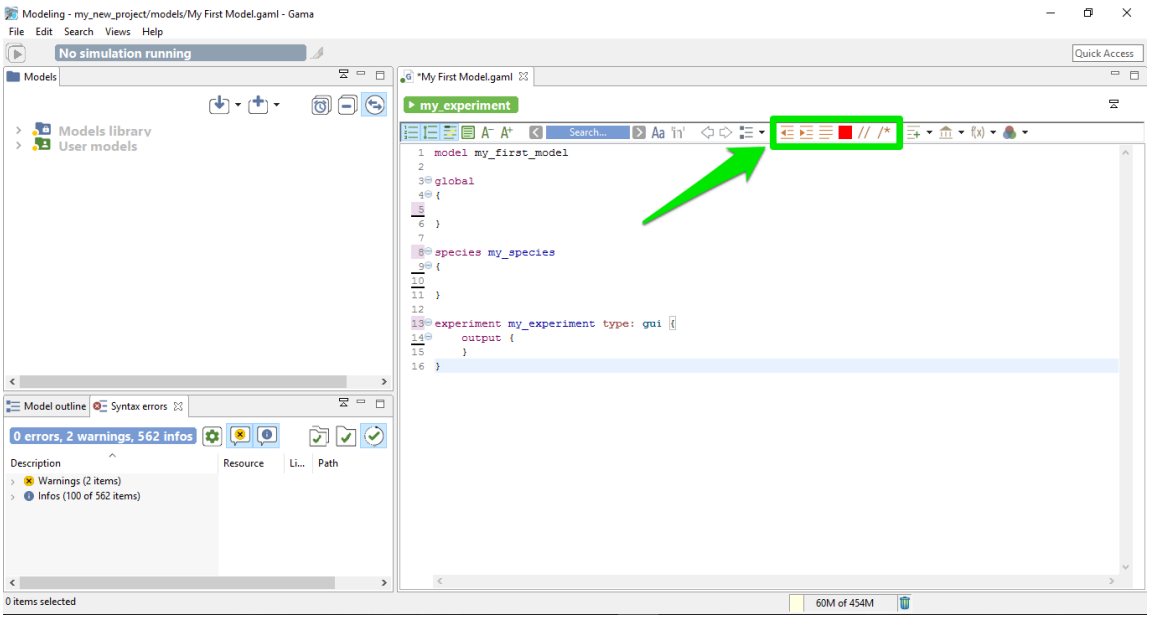


Figure 20.8: images/format_the_text_in_editor.png

Format

This useful feature re-indent automatically all your model.

Re-serialize

Re-serialize your model.

Comment

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

Vocabulary tools in the editor

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

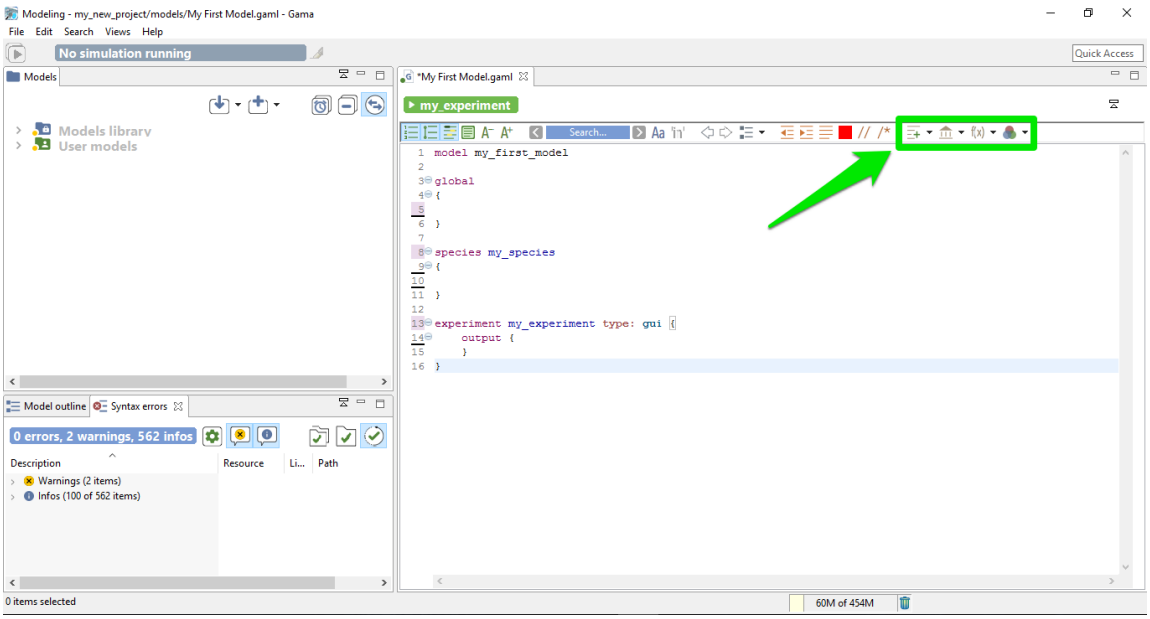


Figure 20.9: images/vocabulary_help_in_editor.png

Templates

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

Built-in attributes, built-in actions

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

. . . will generate the following code:

All the comments are generated automatically from the current documentation.

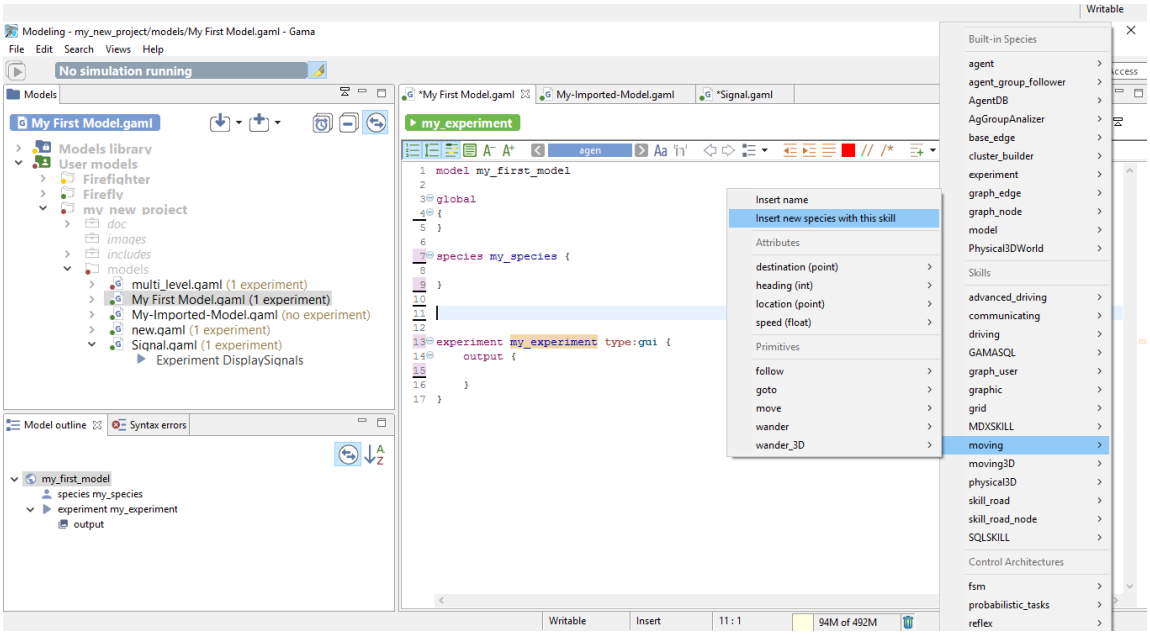


Figure 20.10: images/insert_species_with_moving_skill1.png

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

Figure 20.11: images/insert_species_with_moving_skill2.png

Operators

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

Colors

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

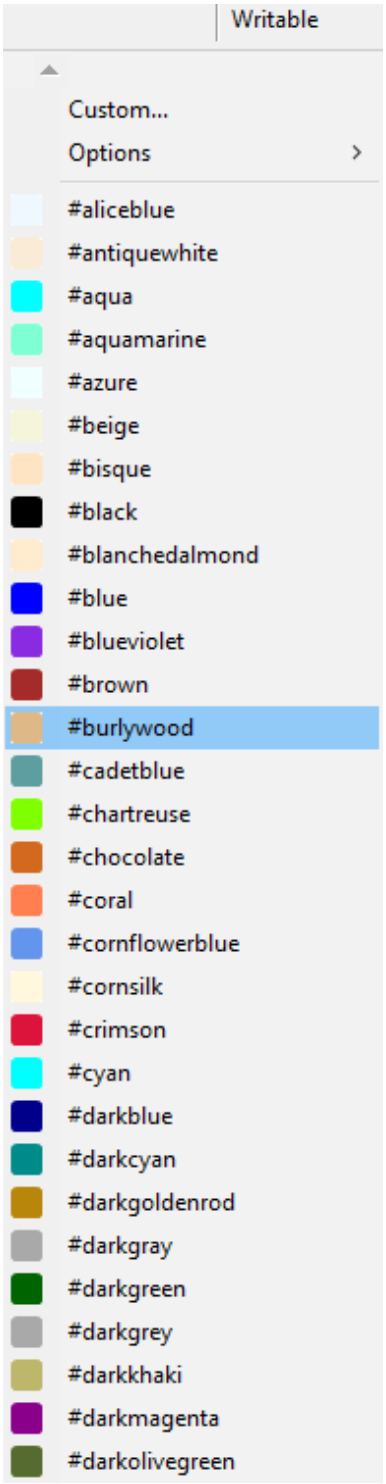


Figure 20.12: images/color.png

Chapter 21

Validation of Models

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

Table of contents

Syntactic errors

These errors are produced when the modeler enters a sentence that has no meaning in the grammar of GAML (see the documentation of the language). It can either be a

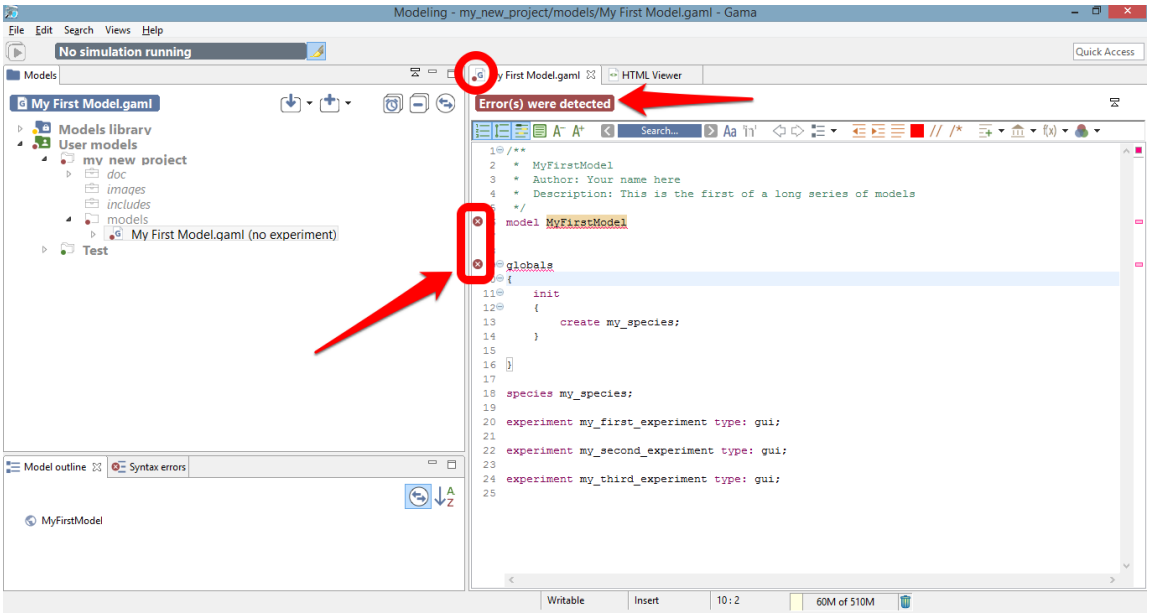


Figure 21.1: images/model_with_syntactic_errors.png

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.

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**).

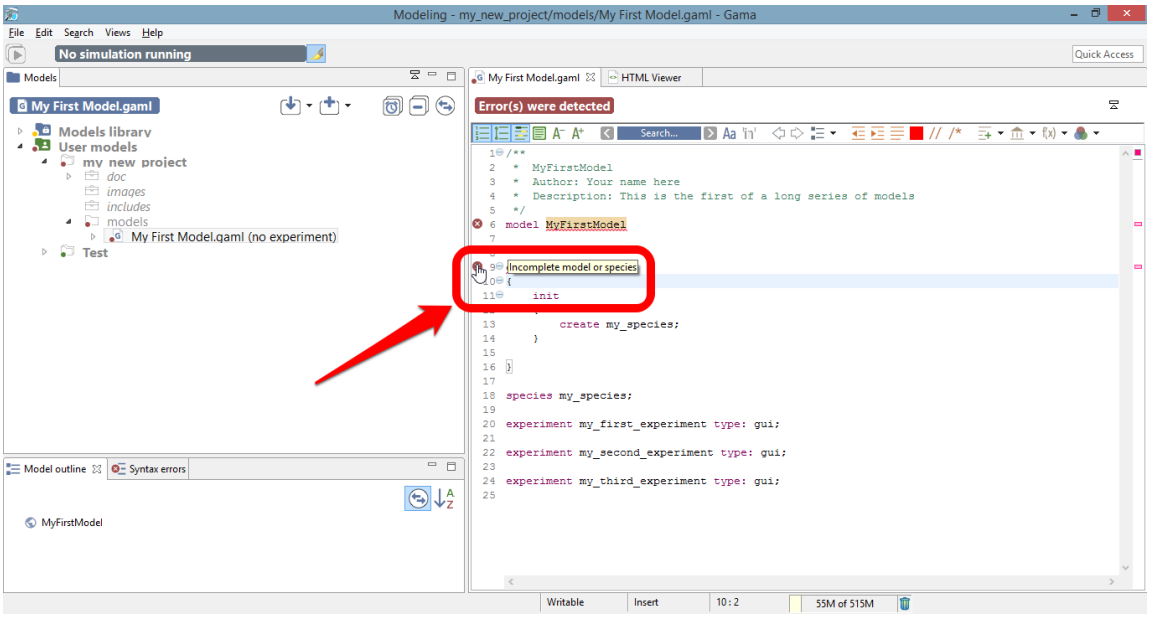


Figure 21.2: images/model_with_syntactic_errors_and_hover.png

Figure 21.3: images/semantic_error_detail.png

Semantic errors

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

This sentence below is **syntactically** correct:

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

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

Semantic errors are reported in a way similar to syntactic errors, except that no **marker** are displayed beside the model statement. The compiler tries to report

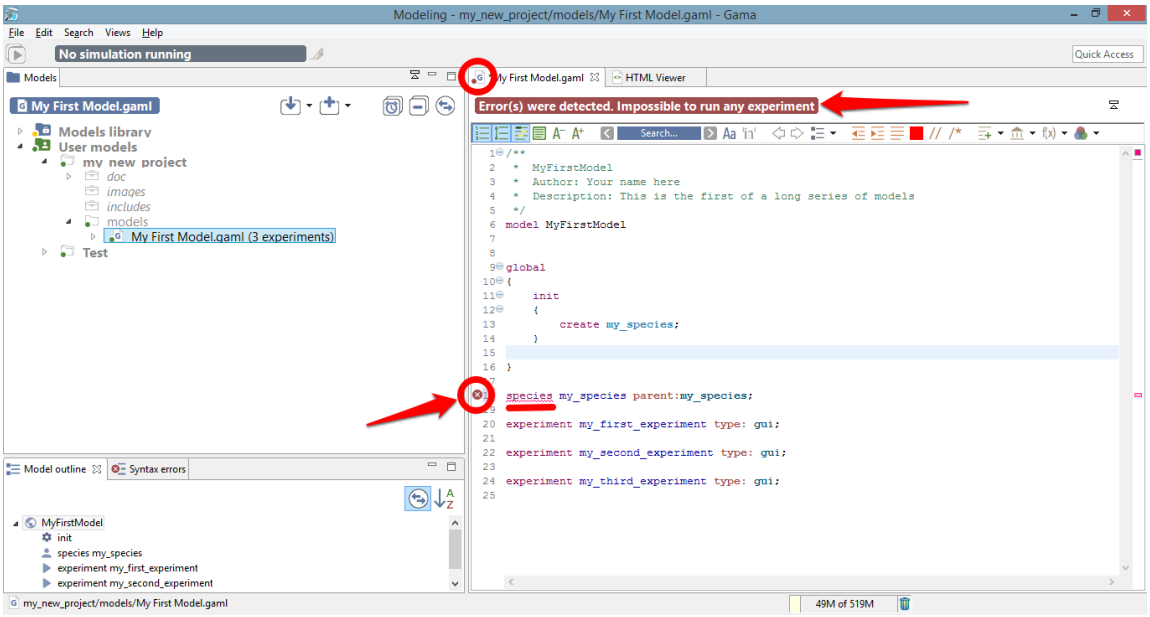


Figure 21.4: images/model_with_semantic_errors.png

them as precisely as possible, underlining the places where they have been found and outputting hopefully meaningful error messages. In the example below, for instance, we use a wrong number of arguments for defining a square geometry. Although the sentence is syntactically correct, GAMA will nevertheless issue an error and prevent the model from being experimentable.

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

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

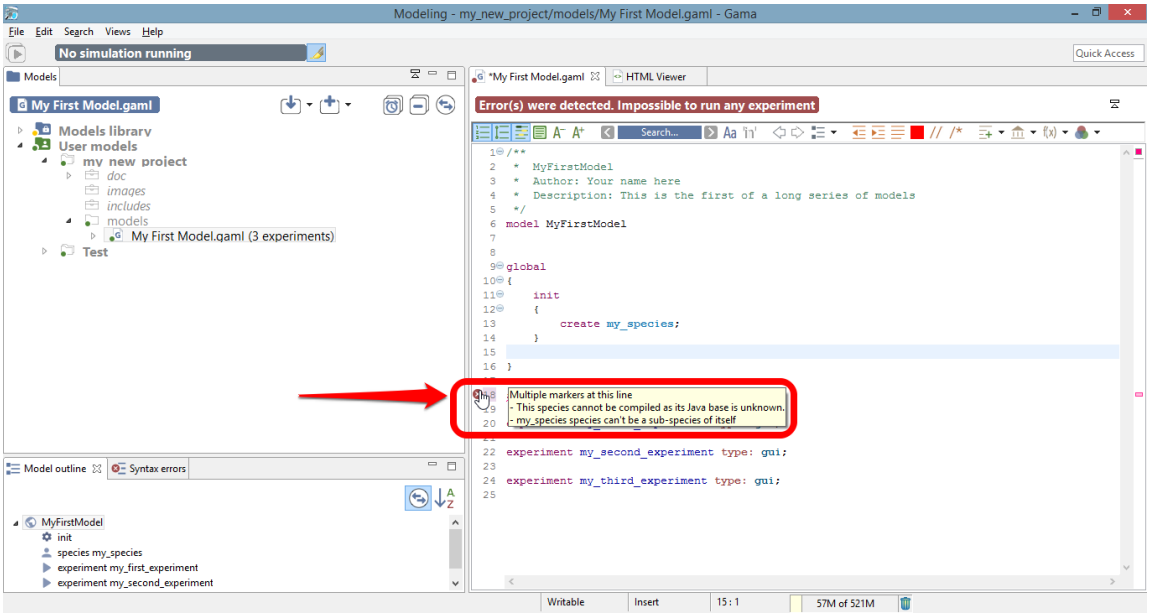


Figure 21.5: images/model_with_semantic_errors_and_hover.png

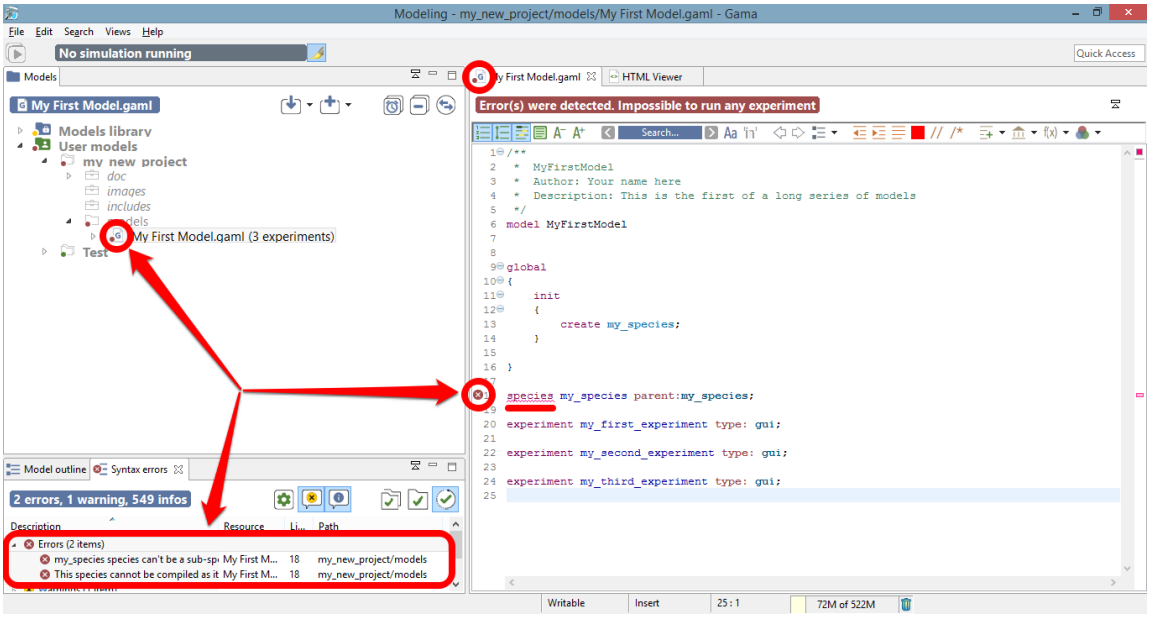


Figure 21.6: images/model_with_semantic_errors_saved.png

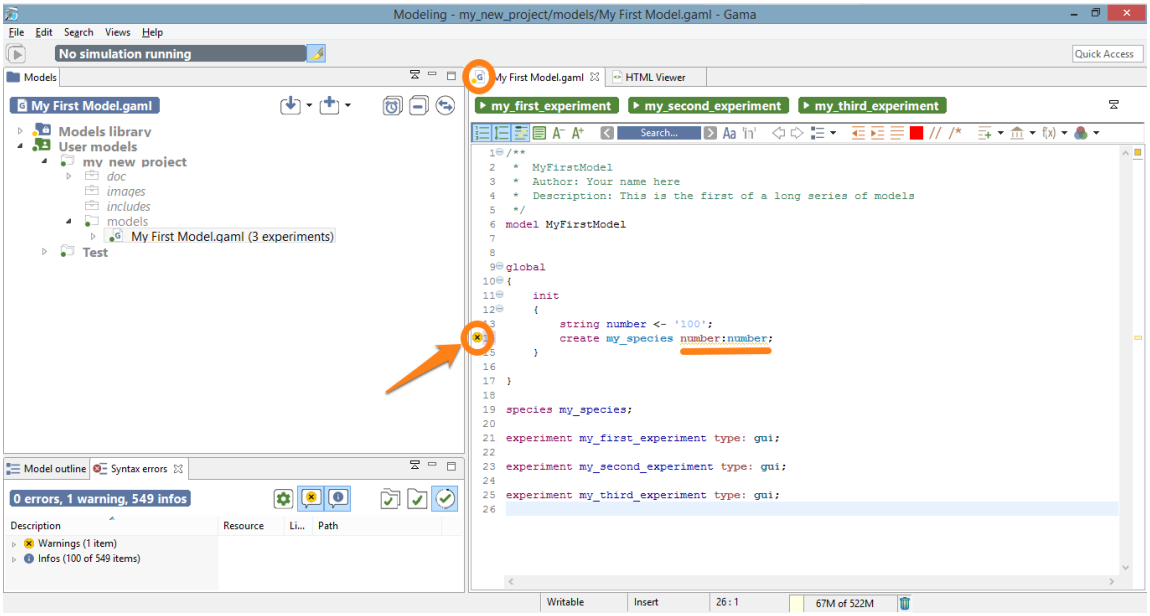


Figure 21.7: images/model_with_warnings.png

Semantic warnings

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

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

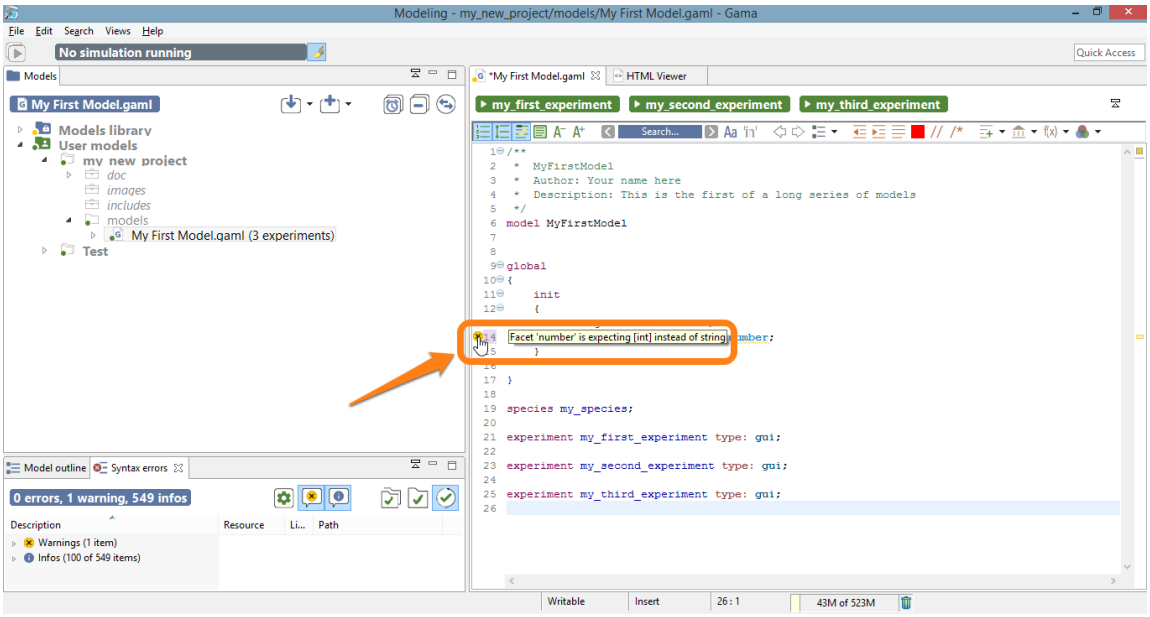


Figure 21.8: images/model_with_warnings_and_hover.png

Semantic information

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

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

Semantic documentation

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

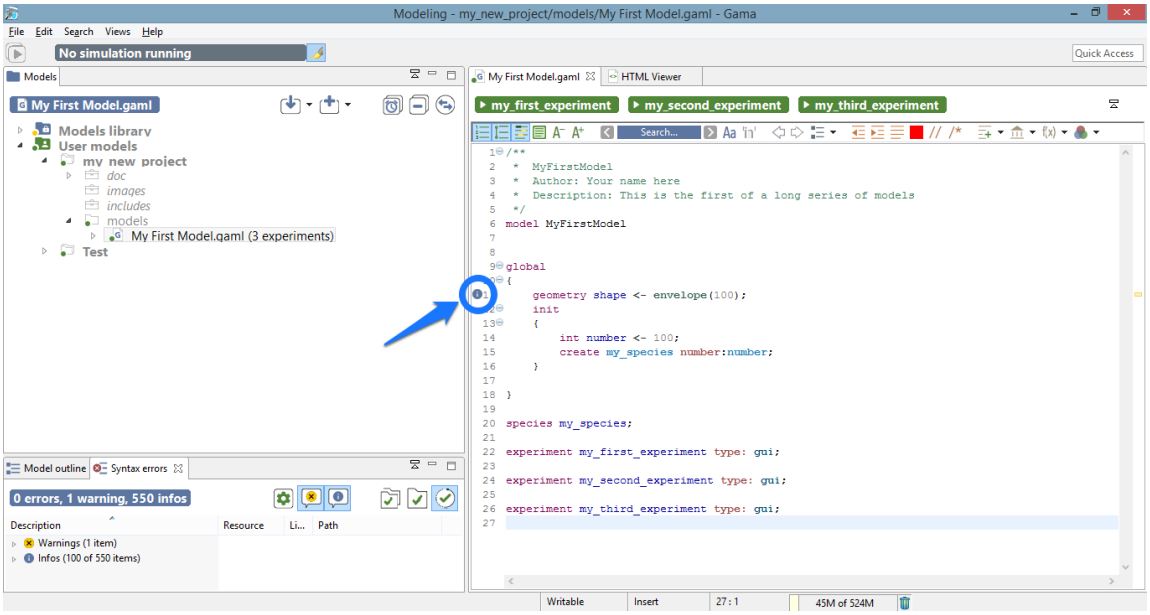


Figure 21.9: images/model_with_info.png

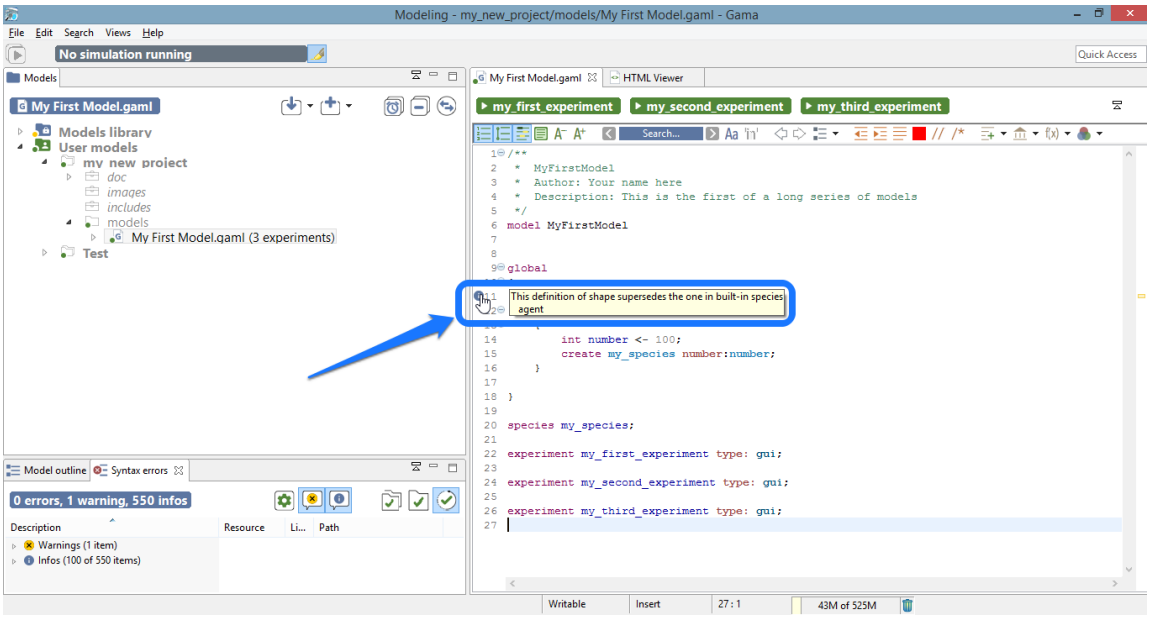


Figure 21.10: images/model_with_info_and_hover.png

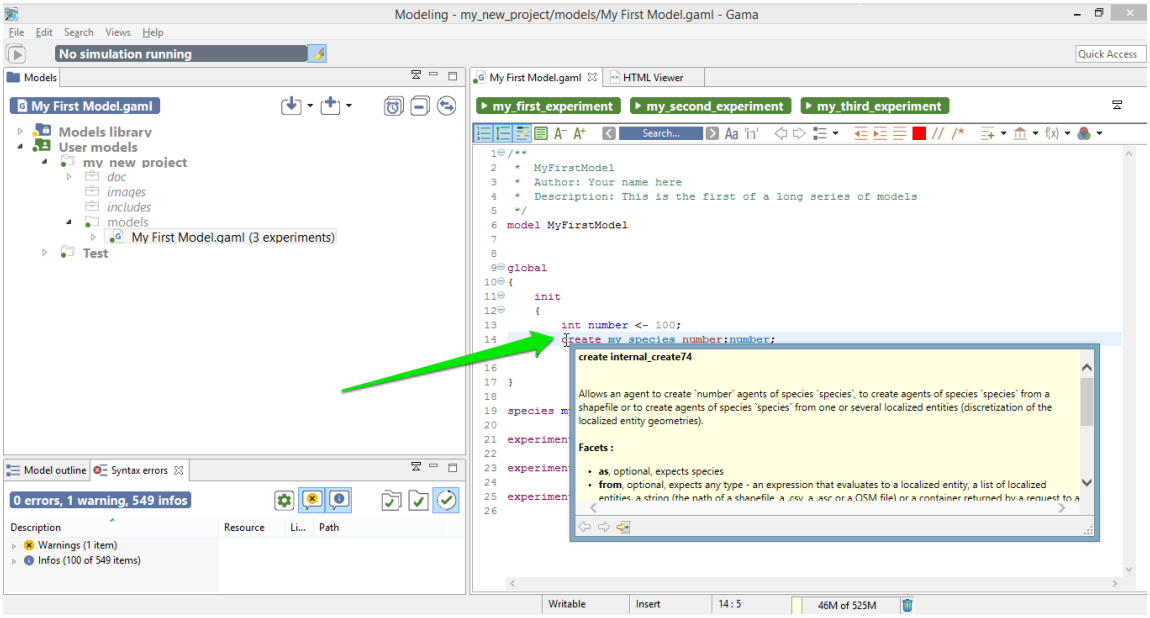


Figure 21.11: images/model_with_no_errors_and_hover.png

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:

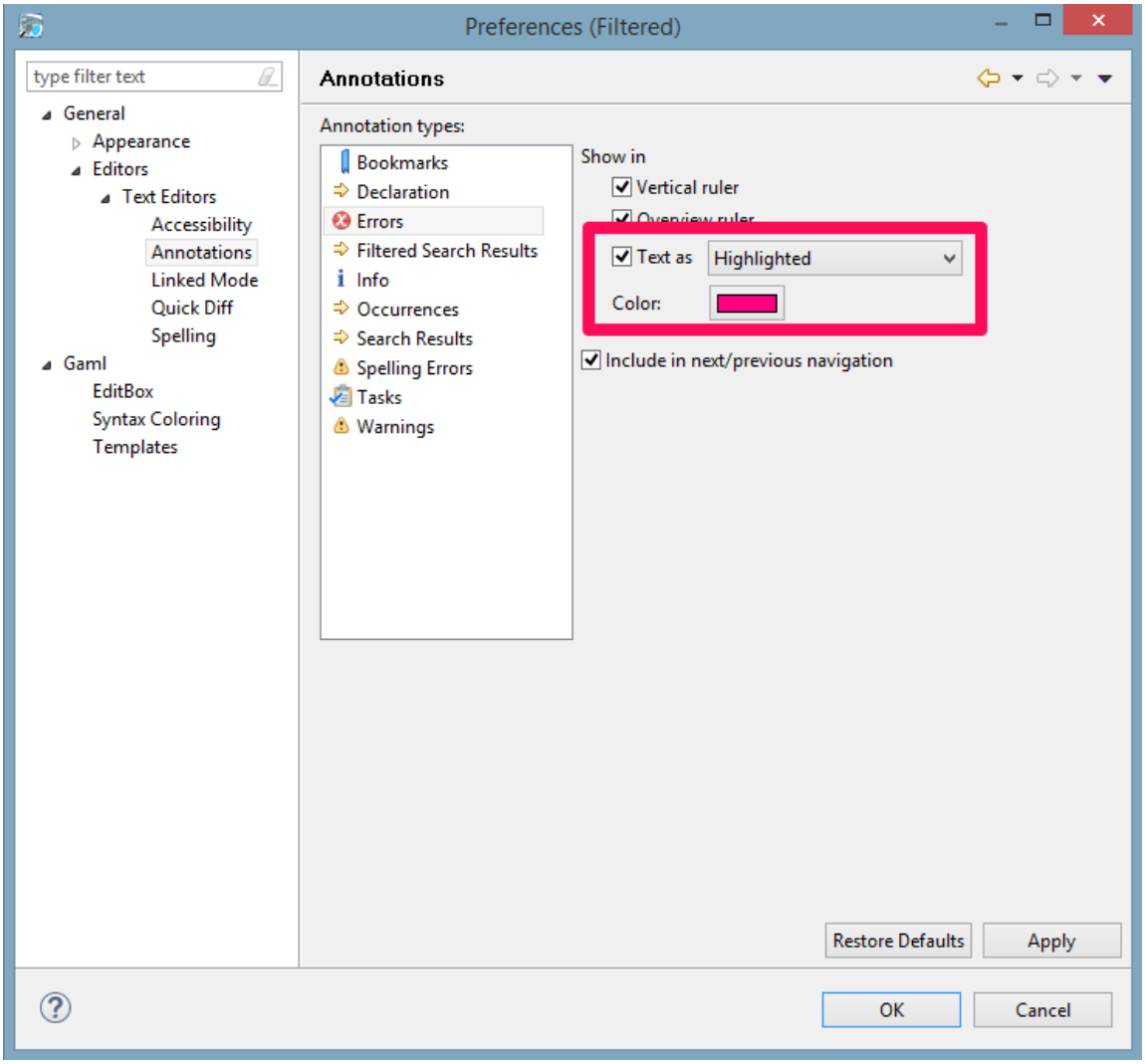


Figure 21.12: images/preferences_annotations.png

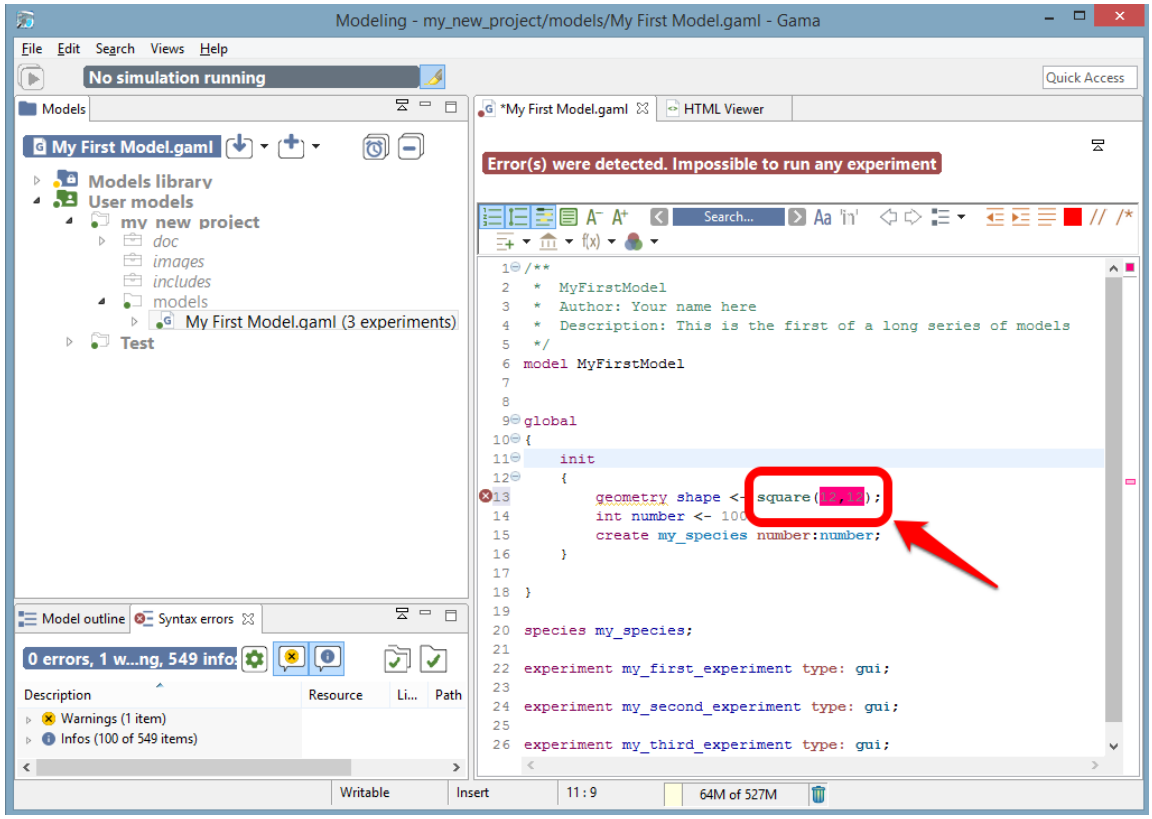


Figure 21.13: images/model_with_semantic_error_different_annotation.png

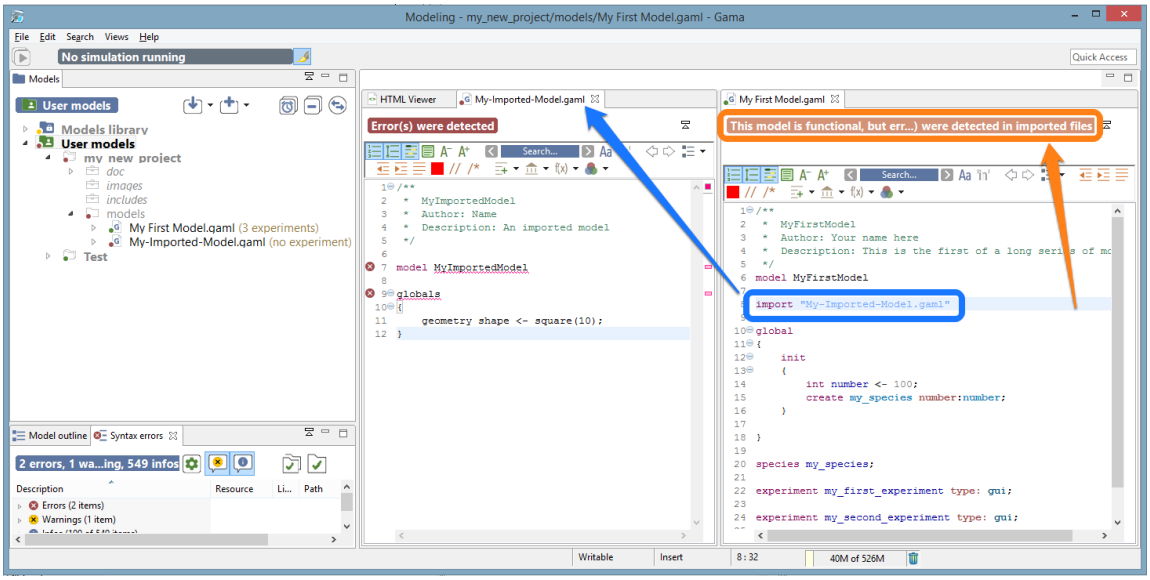


Figure 21.14: images/model_with_imported_errors.png

Errors in imported files

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

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

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

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

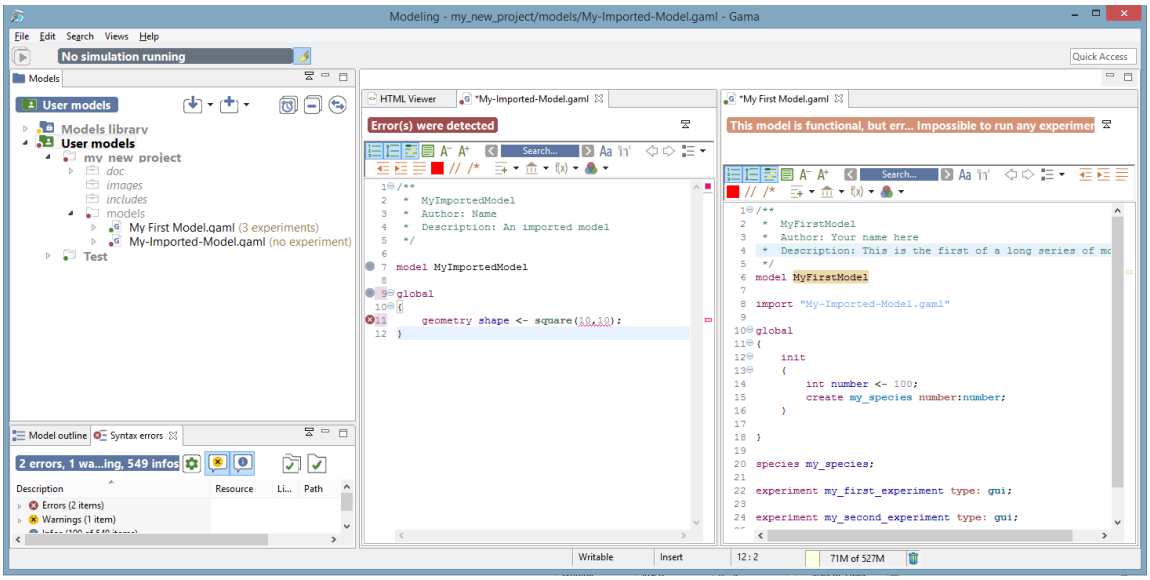


Figure 21.15: images/model_with_imported_semantic_error.png

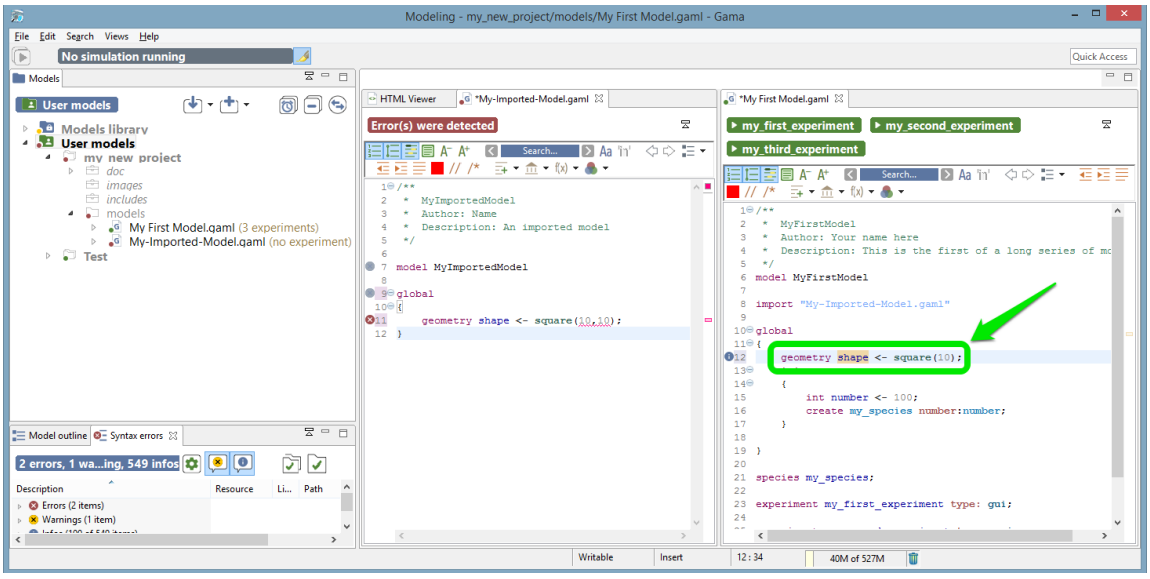


Figure 21.16: images/model_with_superseded_semantic_error.png

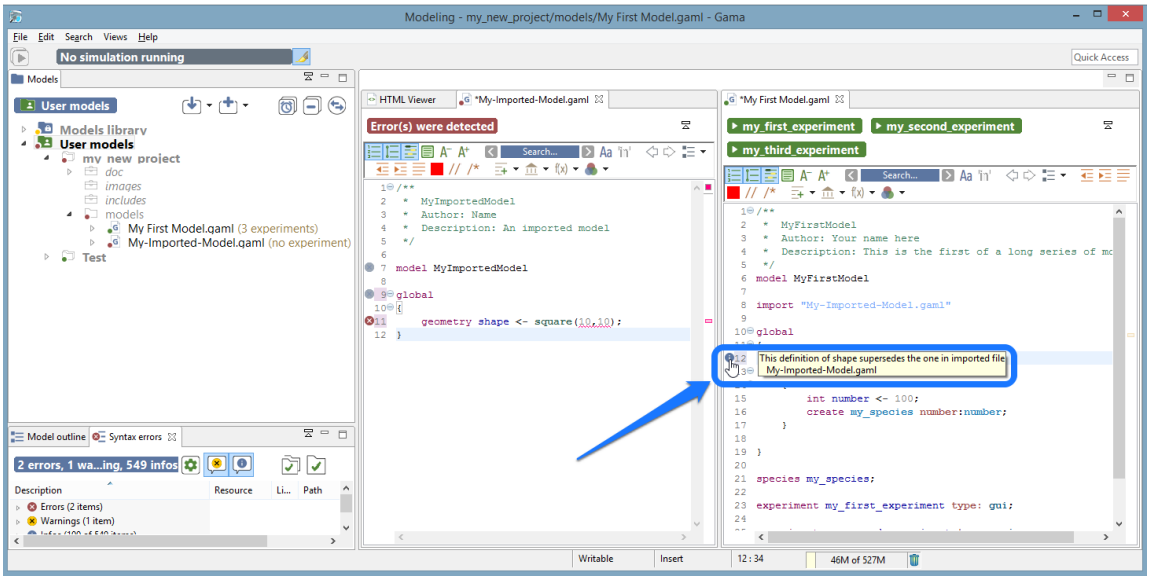


Figure 21.17: images/model_with_superseded_semantic_error_and_hover.png

Cleaning models

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

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

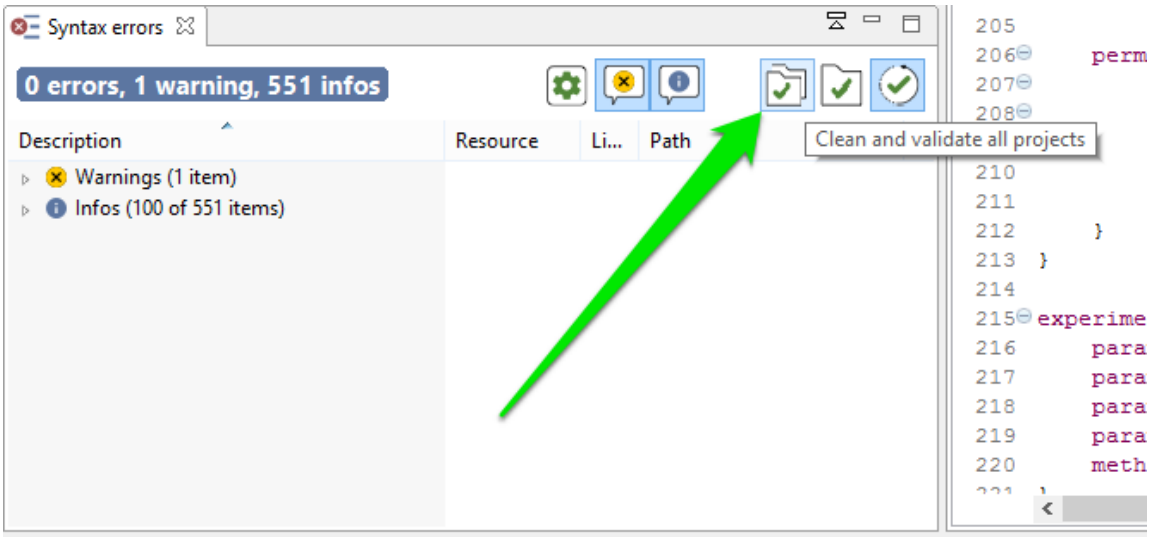


Figure 21.18: images/action_clean.png

Chapter 22

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.

Chapter 23

Launching Experiments from the User Interface

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

Table of contents

From an Editor

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

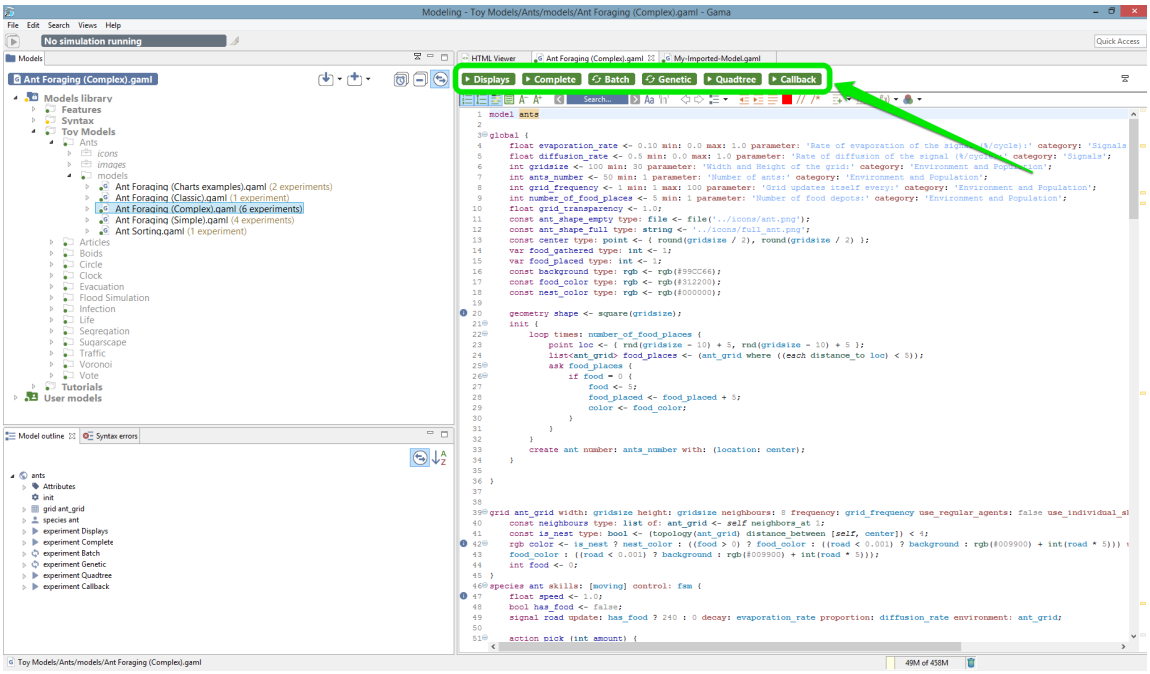


Figure 23.1: images/editor_launch.png

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

From the Navigator

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

Running Experiments Automatically

Once an experiment has been launched (unless it is run in headless mode, of course), it normally displays its views and waits from an input from the user, usually a click

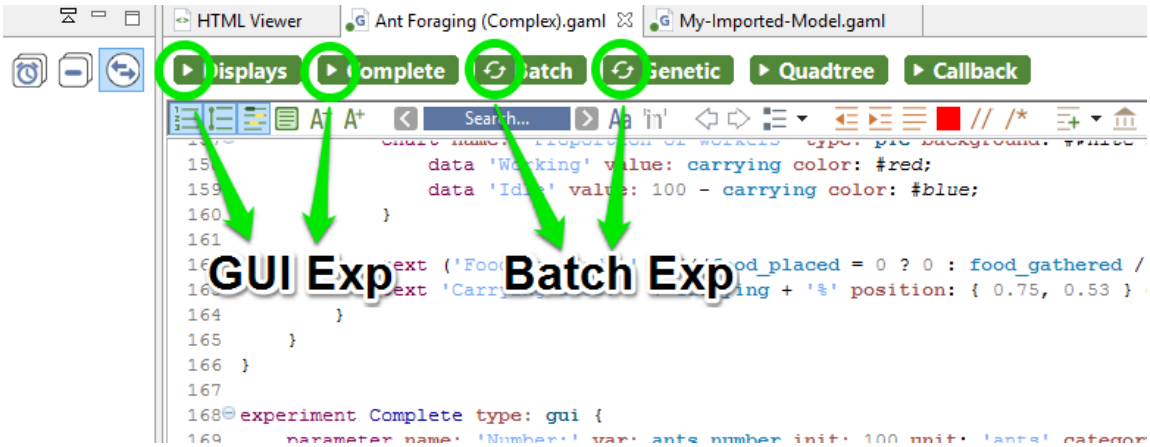


Figure 23.2: images/editor_different_types_of_experiment.png

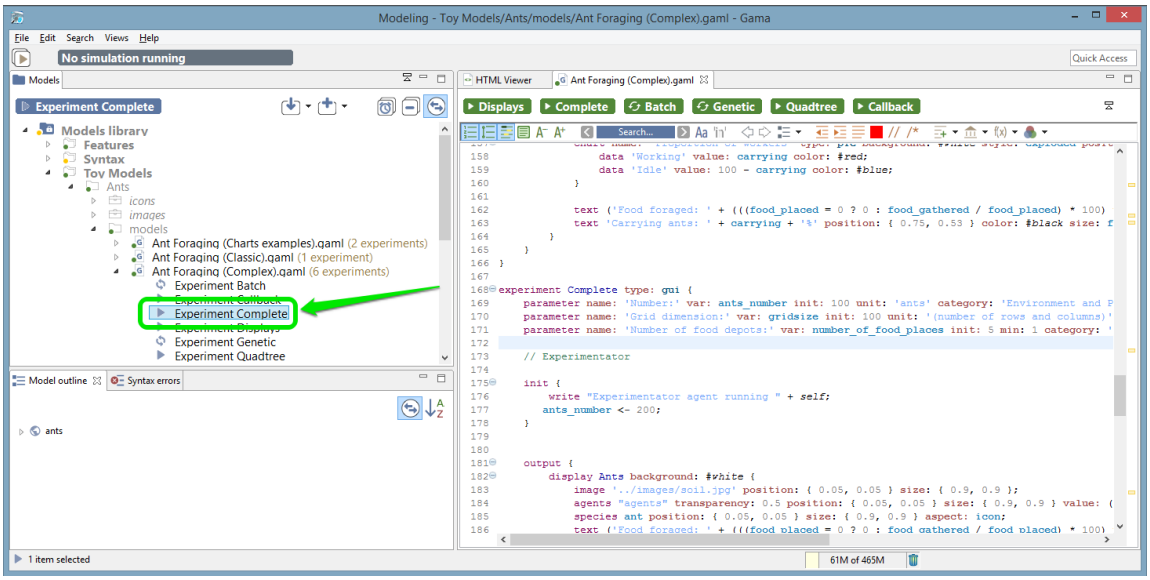


Figure 23.3: images/navigator_launch.png

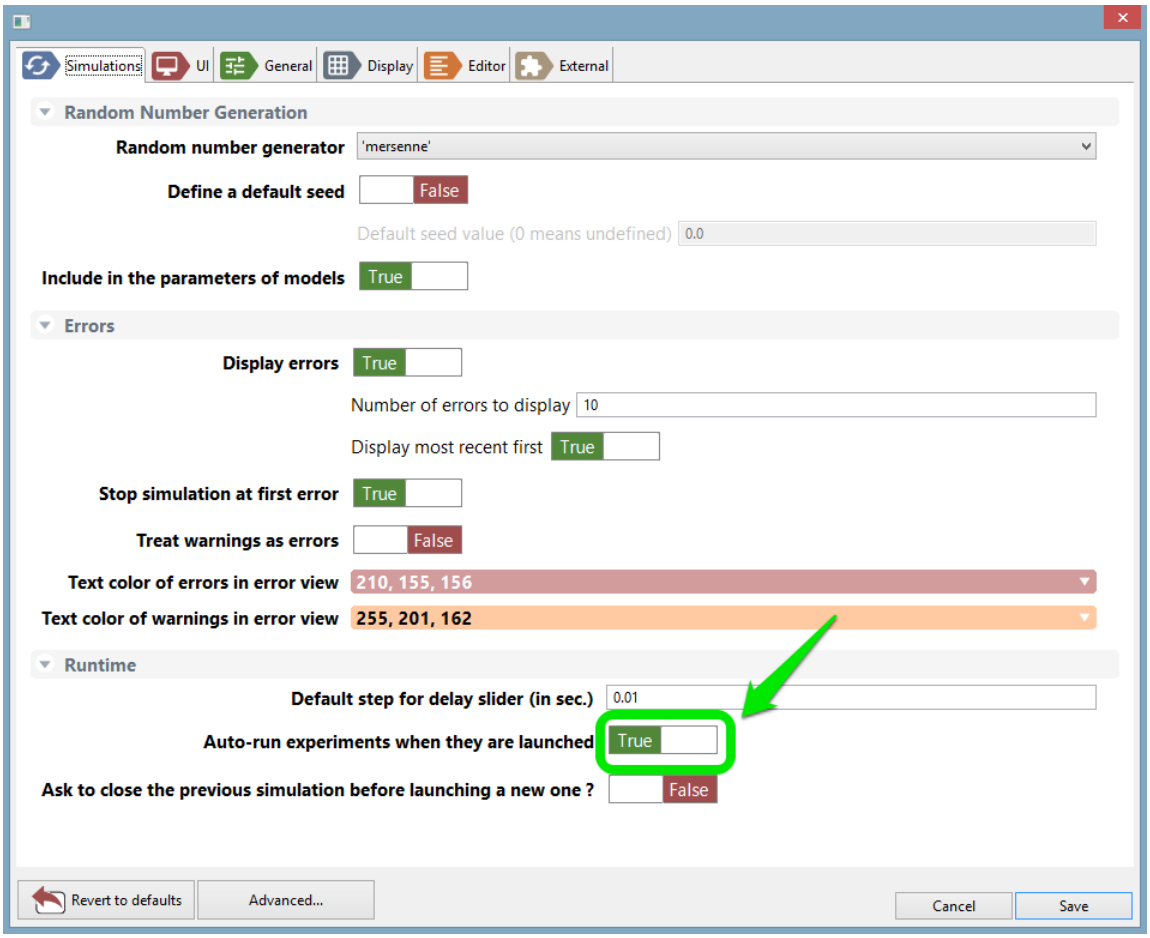


Figure 23.4: images/prefs_auto_run.png

on the "Run" or "Step" buttons (see here).

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

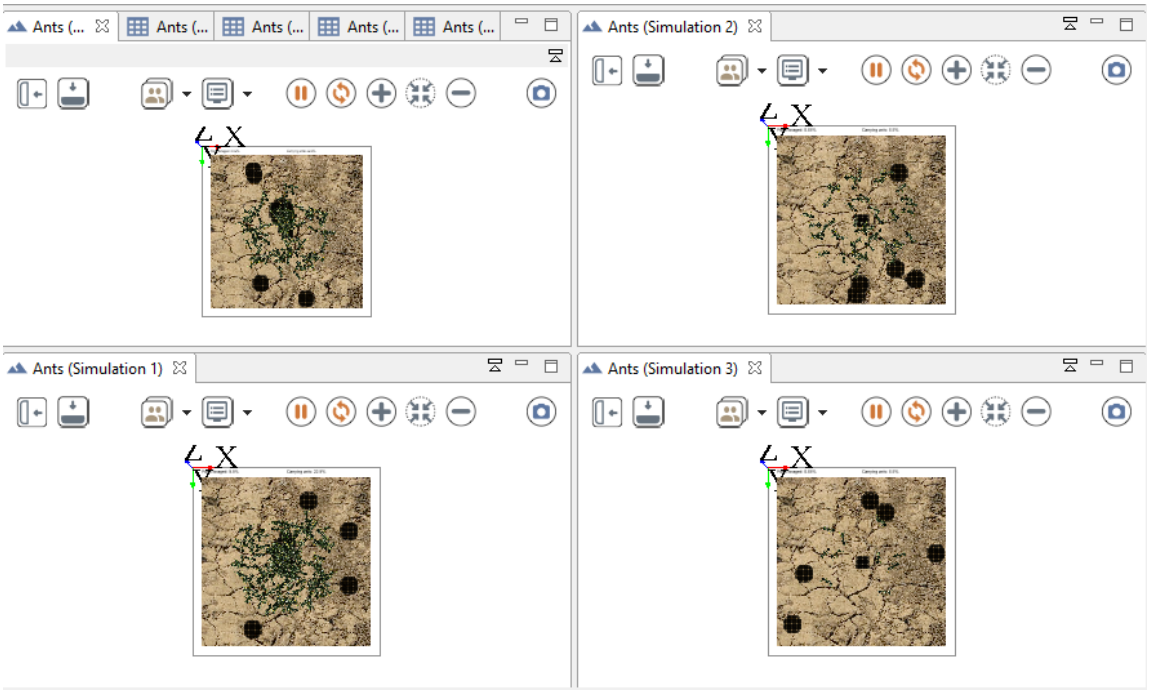


Figure 23.5: images/run_several_simulations.png

Running Several Simulations

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

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

Chapter 24

Experiments User Interface

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

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

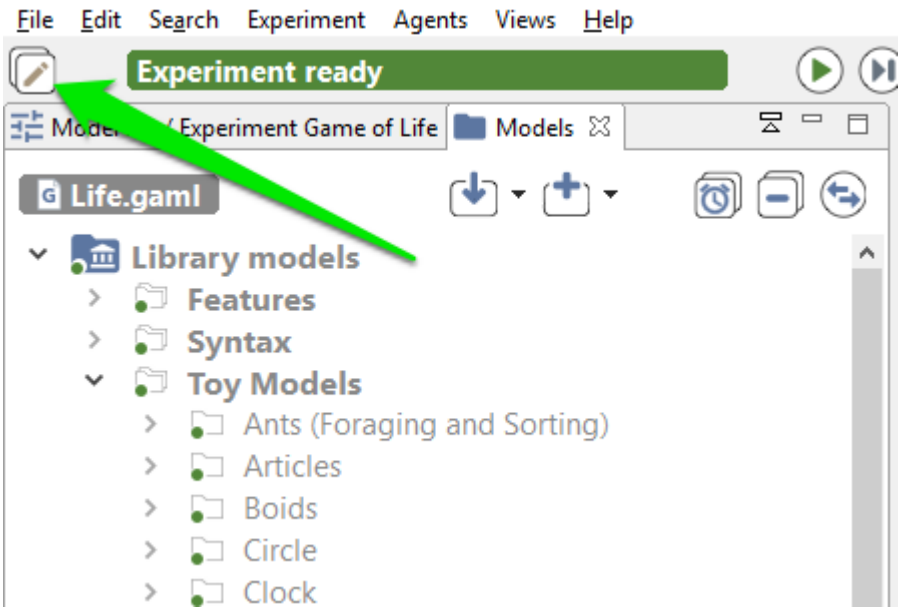


Figure 24.1: images/button_switch.png

Chapter 25

Menus and Commands

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

Table of contents

Experiment Menu

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

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

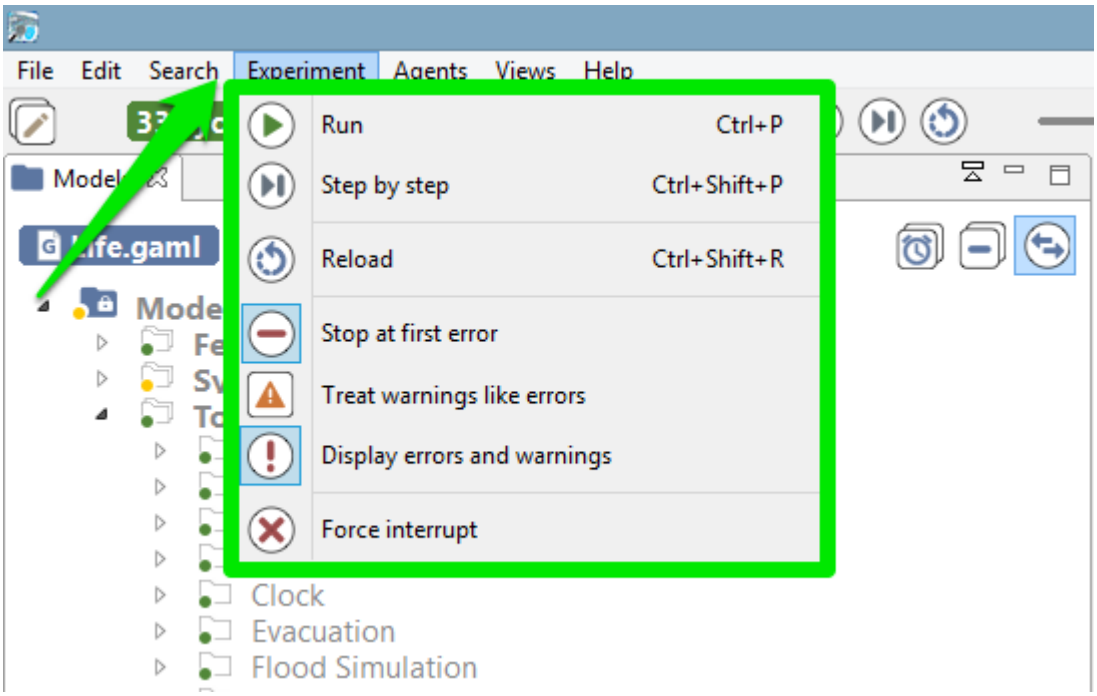


Figure 25.1: images/menu_experiment.png

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

Agents Menu

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

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

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

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

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

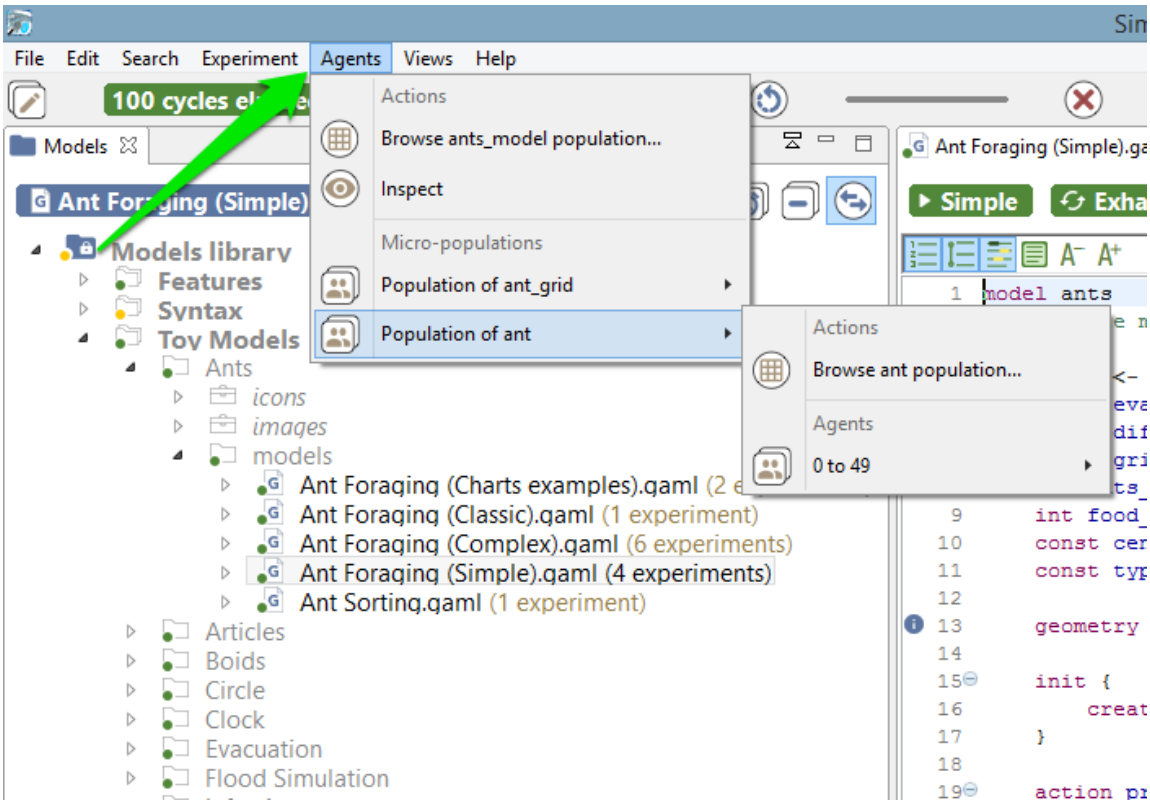


Figure 25.2: images/menu_agents.png

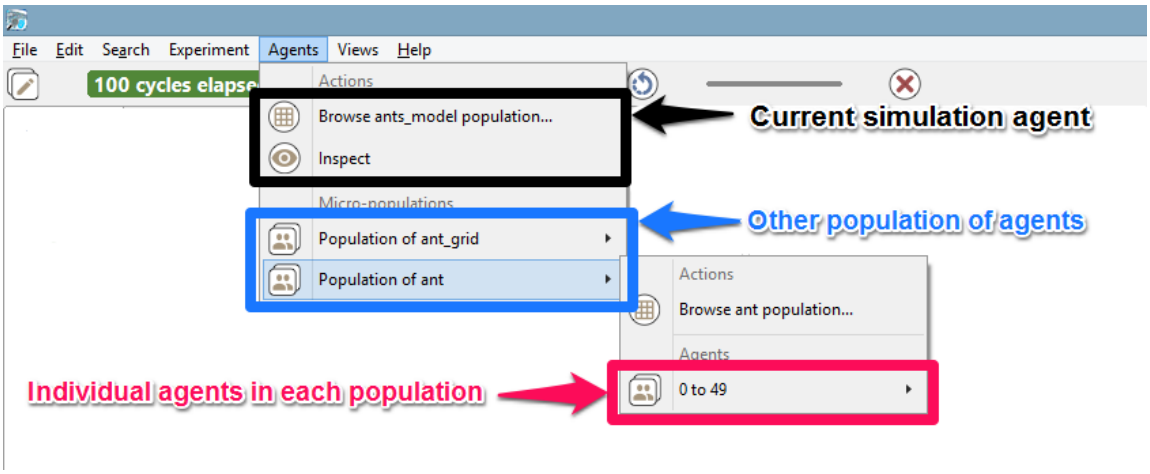


Figure 25.3: images/menu_agents_2.png

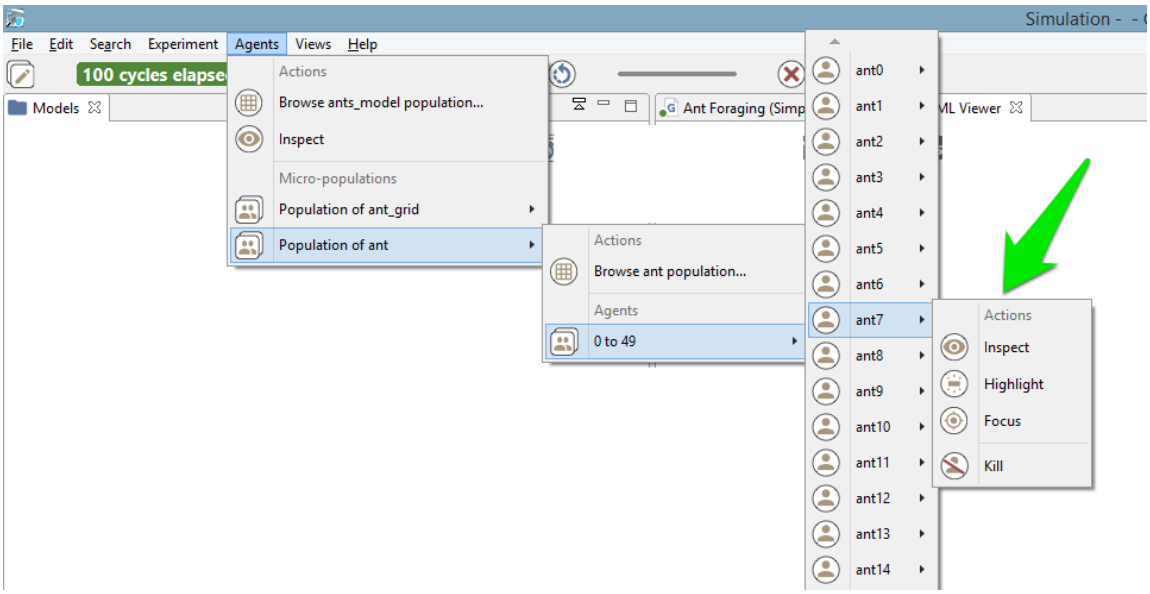


Figure 25.4: images/menu_agents_3.png

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

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

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

General Toolbar

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

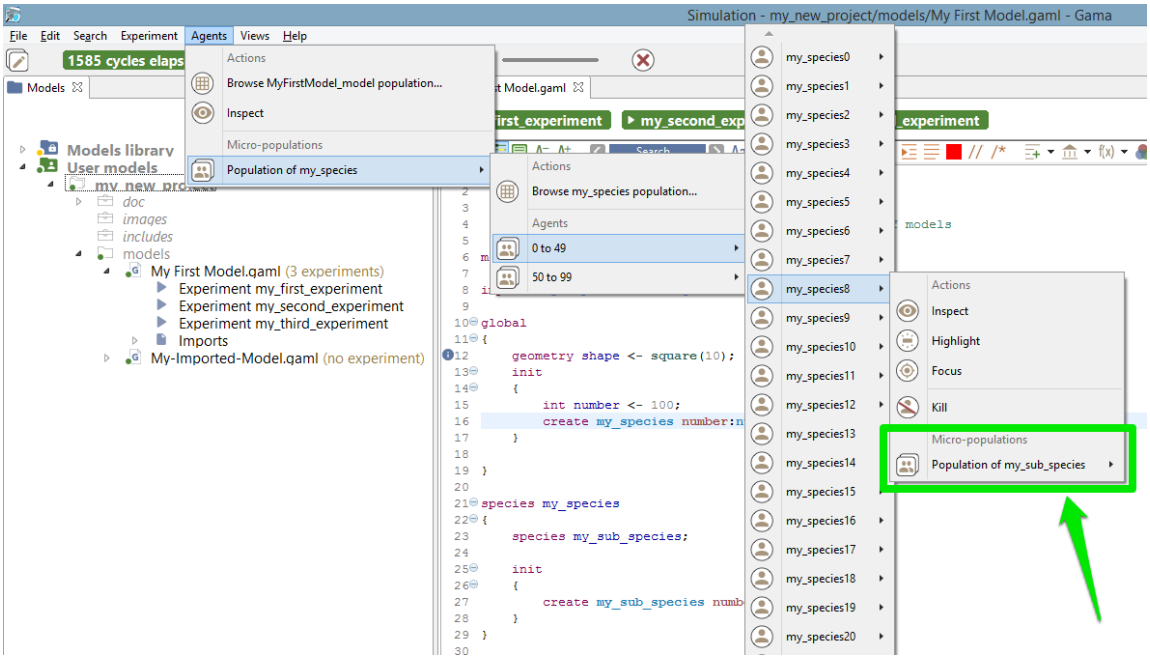


Figure 25.5: images/menu_agents_multi_level.png

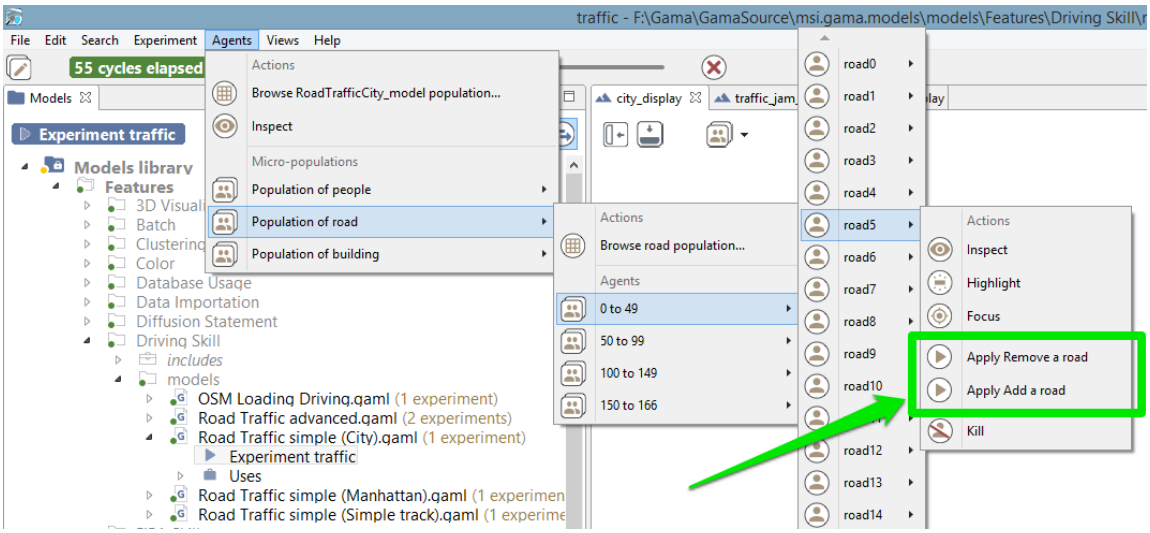


Figure 25.6: images/menu_agents_user_command.png

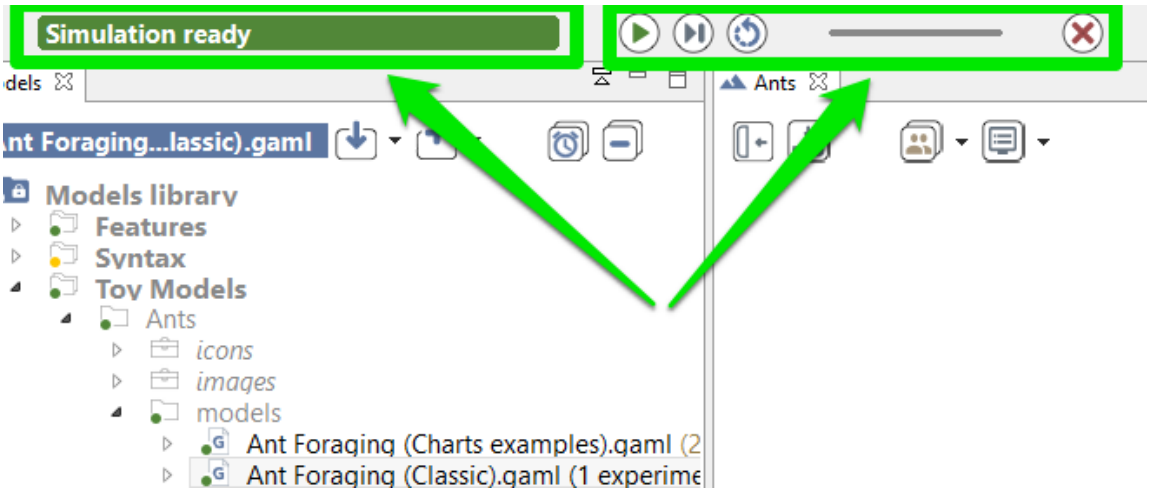


Figure 25.7: images/toolbar.png

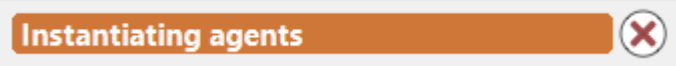


Figure 25.8: images/toolbar_instantiating_agents.png

This toolbar is voluntarily minimalist, with three buttons already present in the experiment menu (namely, "Play/Pause", "Step by Step" and "Reload"), which don't need to be detailed here, and two new controls ("Experiment status" and "Cycle Delay"), which are explained below.

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

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

Figure 25.9: images/toolbar_building_outputs.png

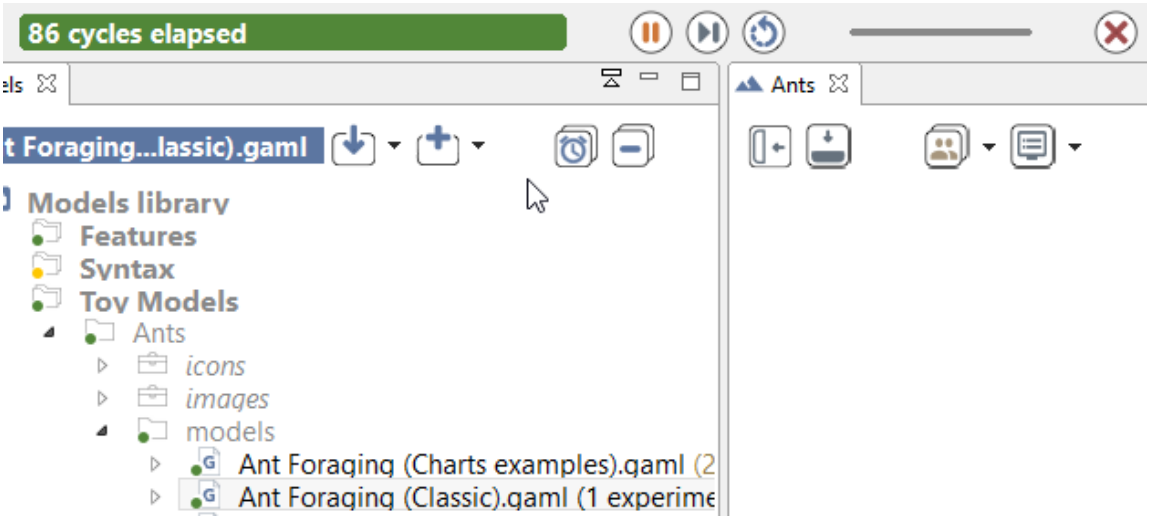


Figure 25.10: images/toolbar_running.png

currently managed by the experiment.

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

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

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

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

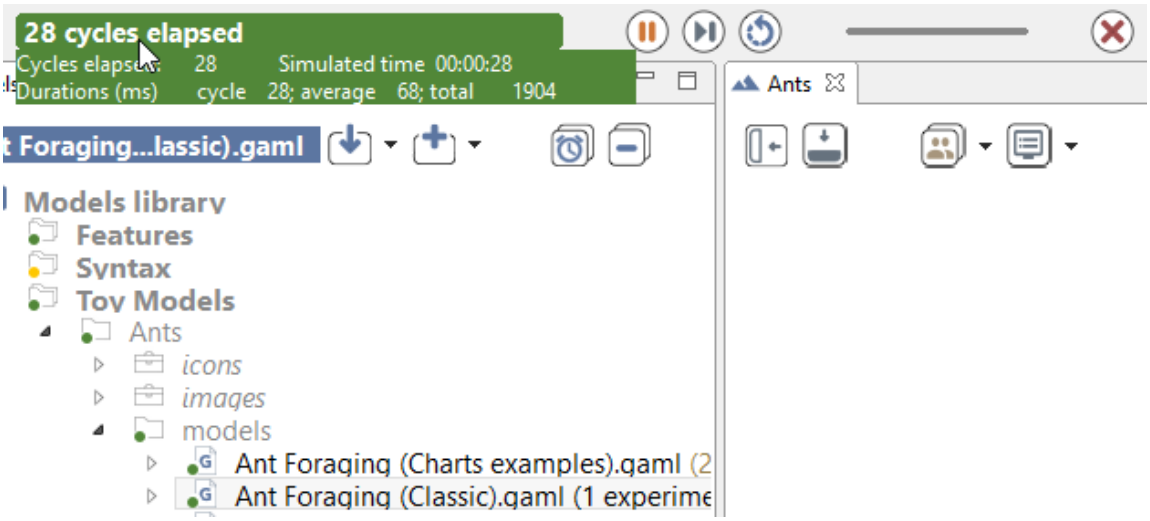


Figure 25.11: images/toolbar_running_with_info.png

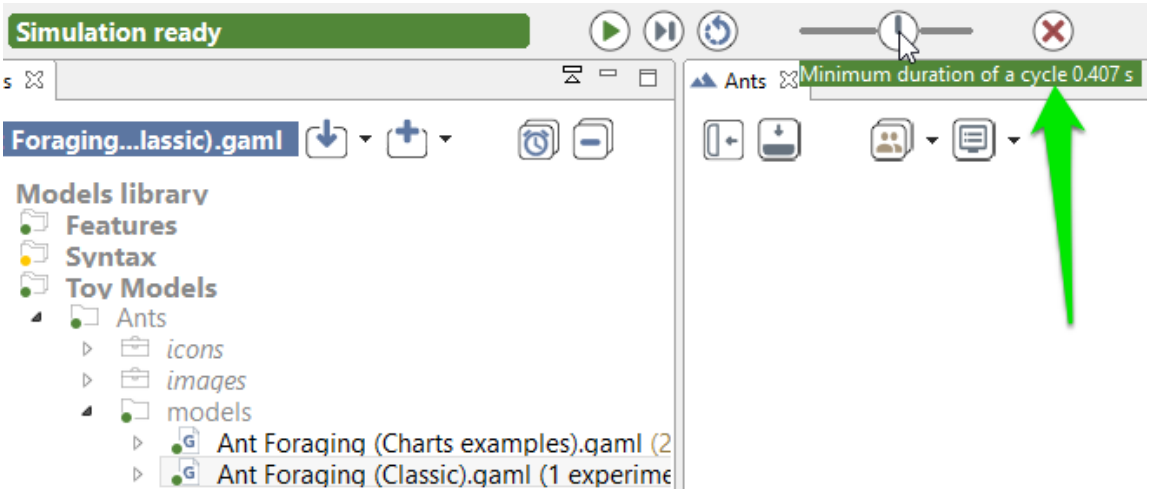


Figure 25.12: images/toolbar_running_with_delay.png

Chapter 26

Parameters View

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

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

Table of contents

Built-in parameters

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

- the Random Number Generator, with a choice between 3 RNG implementations,
- the Random Seed

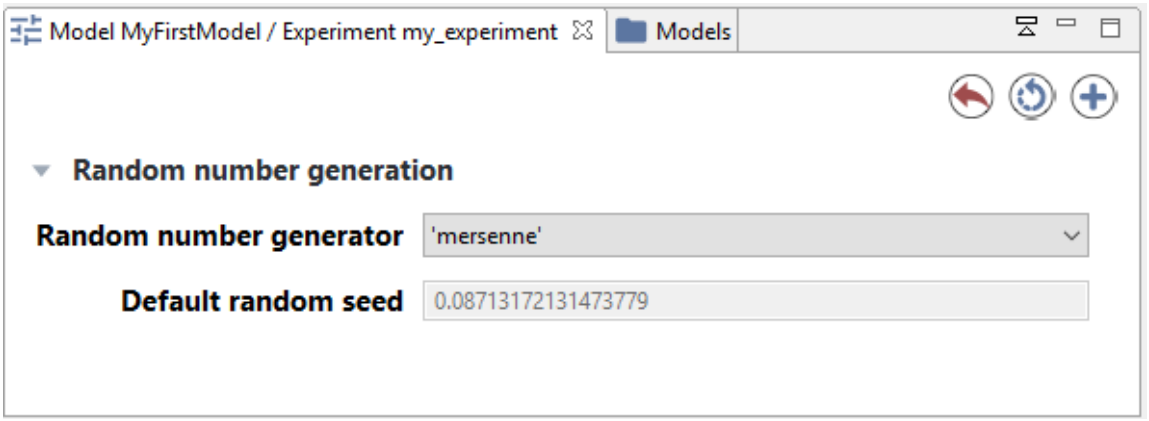


Figure 26.1: images/parameters_built_in.png

Parameters View

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

The above parameters view is generated from the following code:

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

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

    parameter "my_list" var: l <- [] category:"Complex types";
```

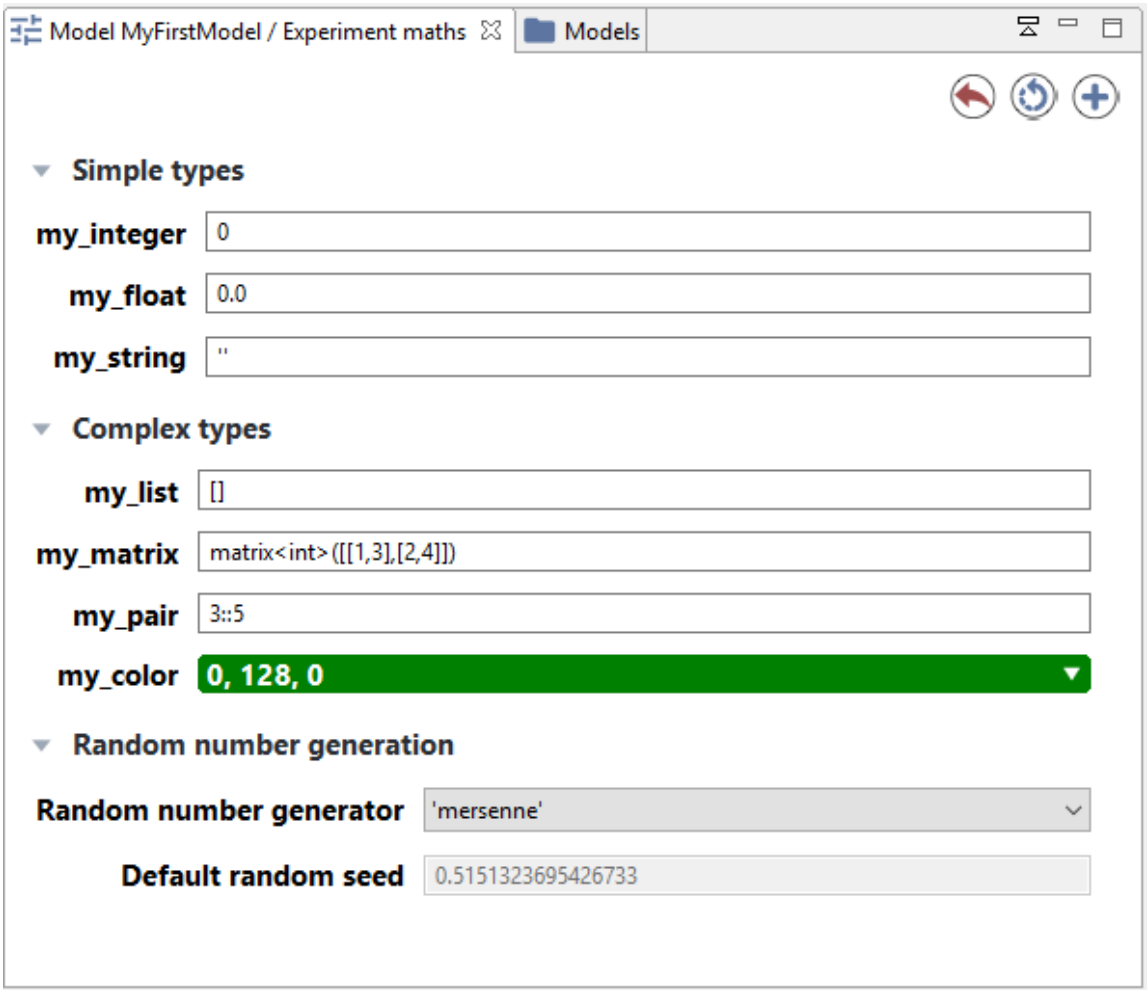


Figure 26.2: images/parameters.png

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

    output {}
}
```

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

Modification of parameters values

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

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

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

Chapter 27

Inspectors and monitors

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

- agent browser
- agent inspector

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

Table of contents

Agent Browser

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

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

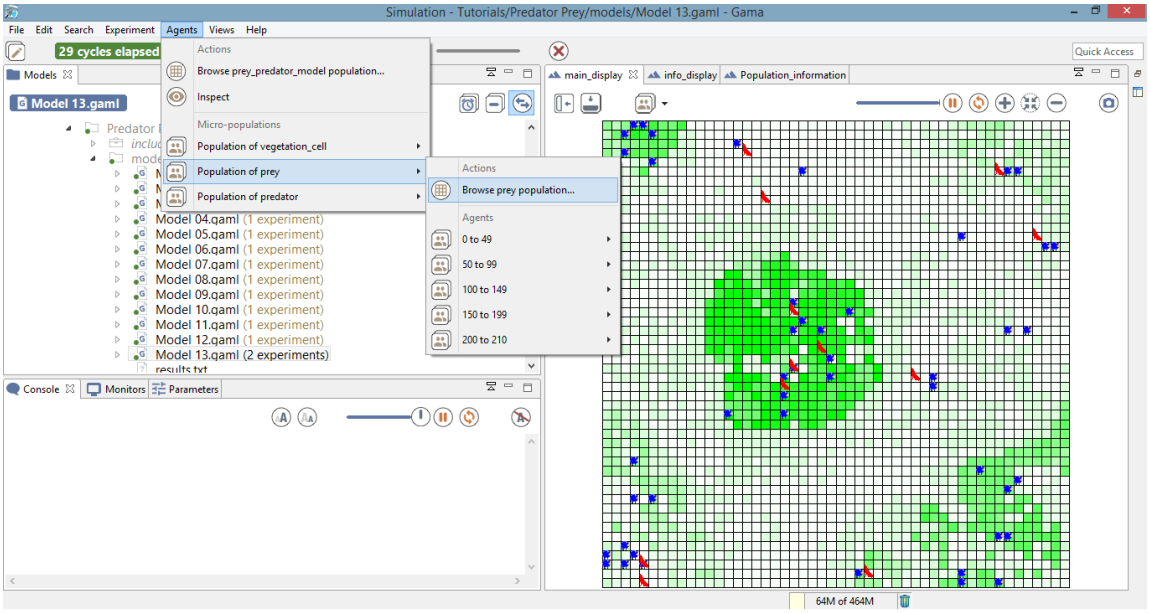


Figure 27.1: images/browse-menu.png

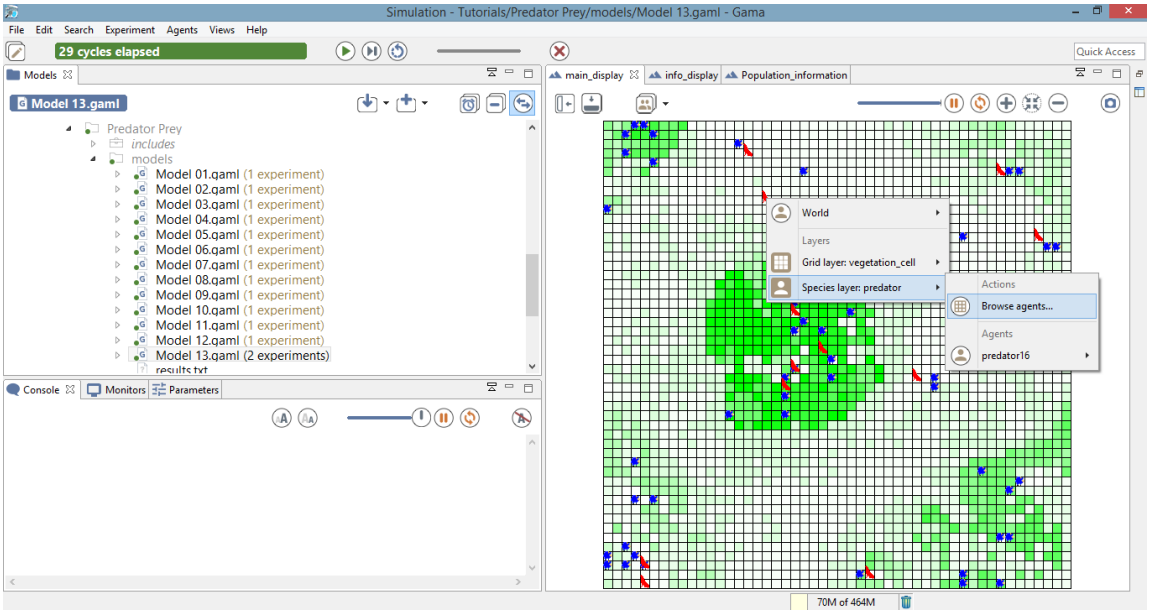


Figure 27.2: images/browse_right_clicking.png

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).

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

Agent Inspector

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

It is possible to «highlight» the selected agent.

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

Monitor

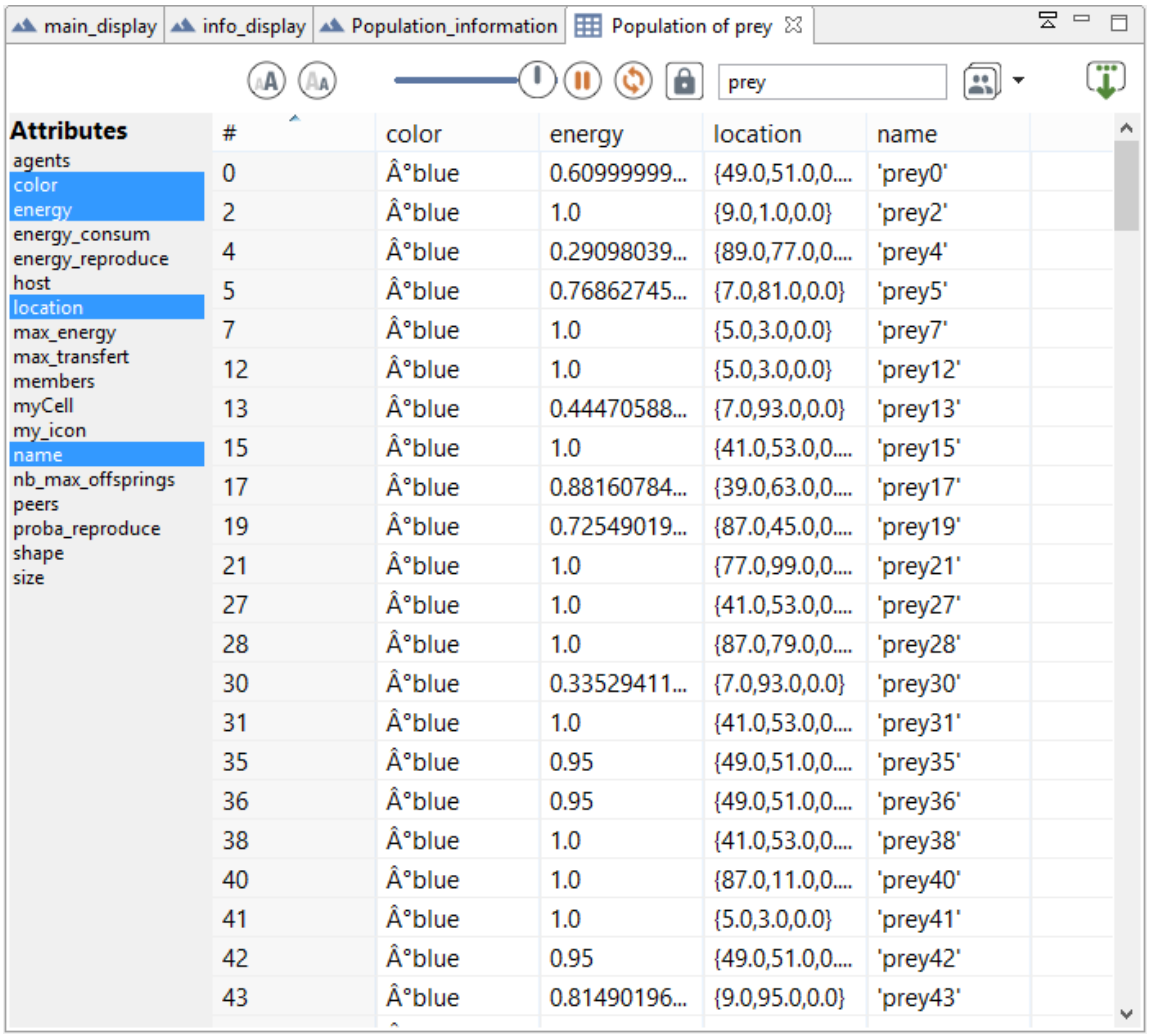
Monitors allow to follow the value of a GAML expression. For instance, the following monitor allows to follow the number of infected people agents during the simulation. The monitor is updated at each simulation step.

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

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

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

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



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

Figure 27.3: images/browse_result.png

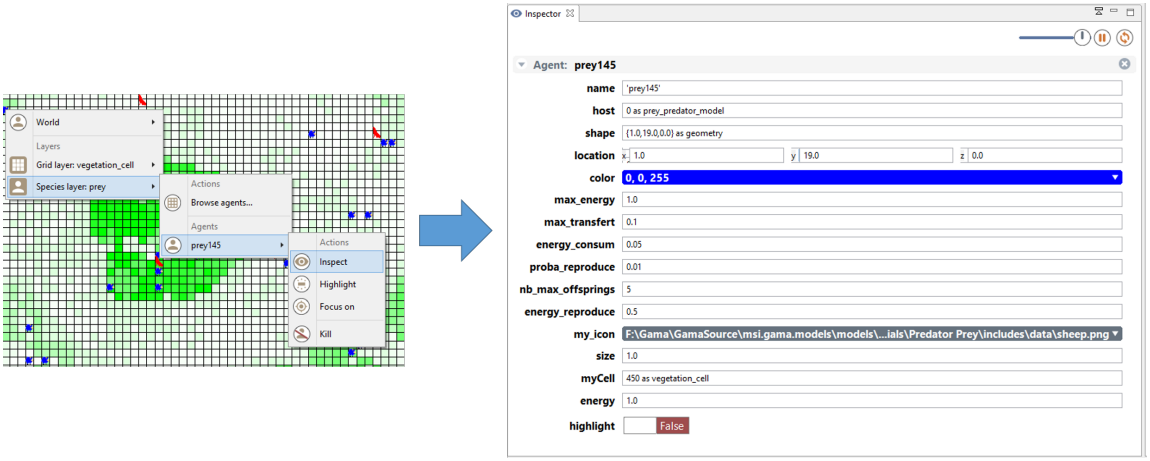


Figure 27.4: images/Agent_inspector.png

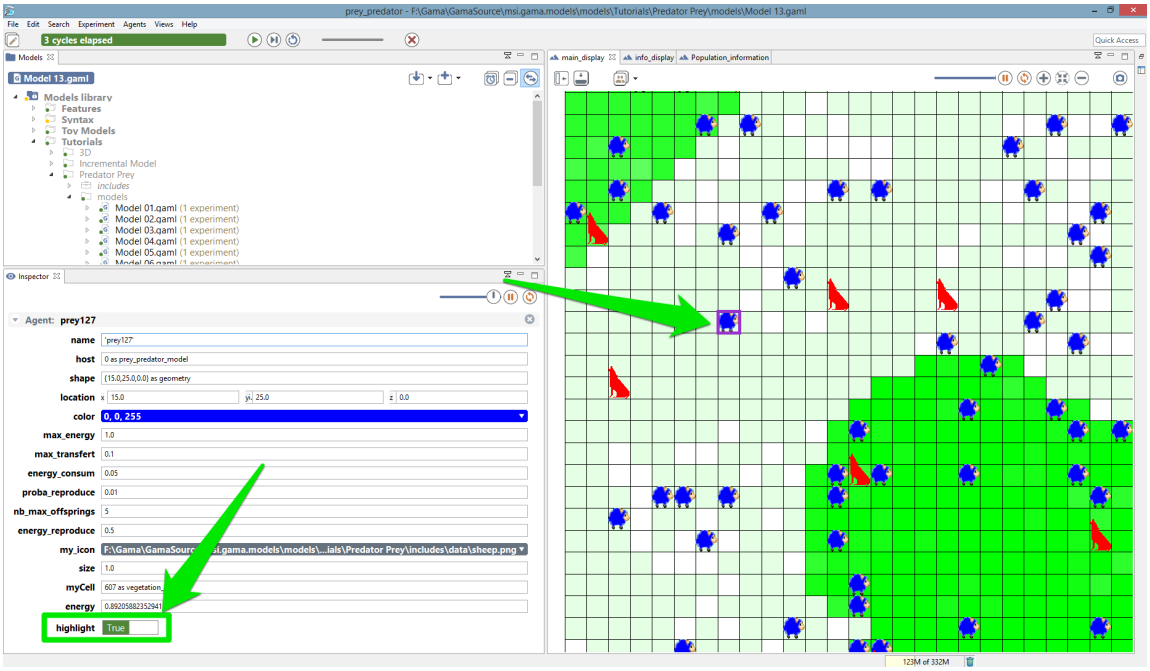


Figure 27.5: images/Inspector_highlight.png

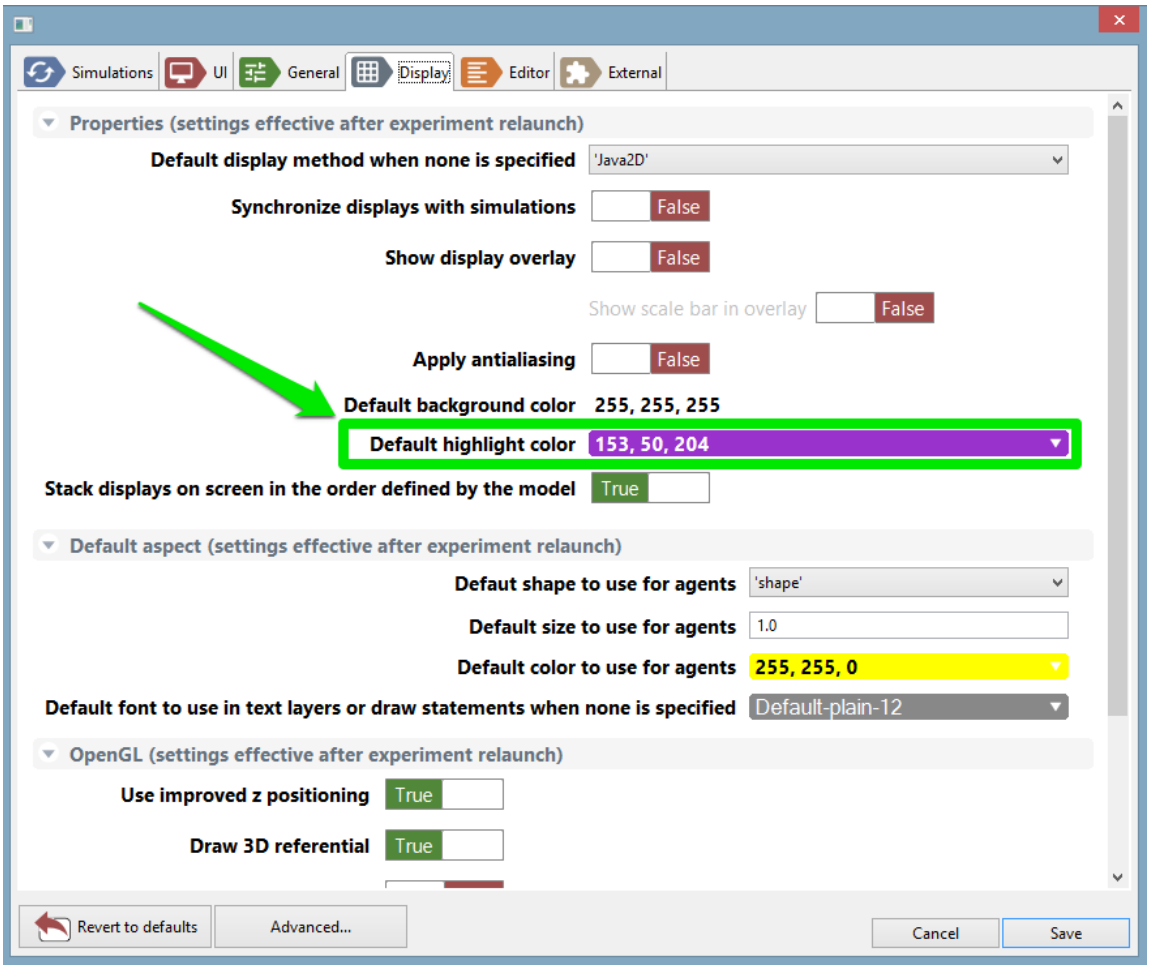


Figure 27.6: images/Inspector_change_highlight_color.png

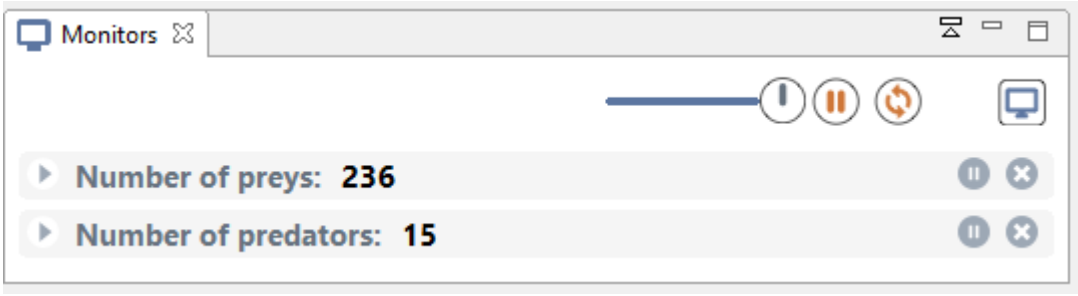


Figure 27.7: images/monitor.png

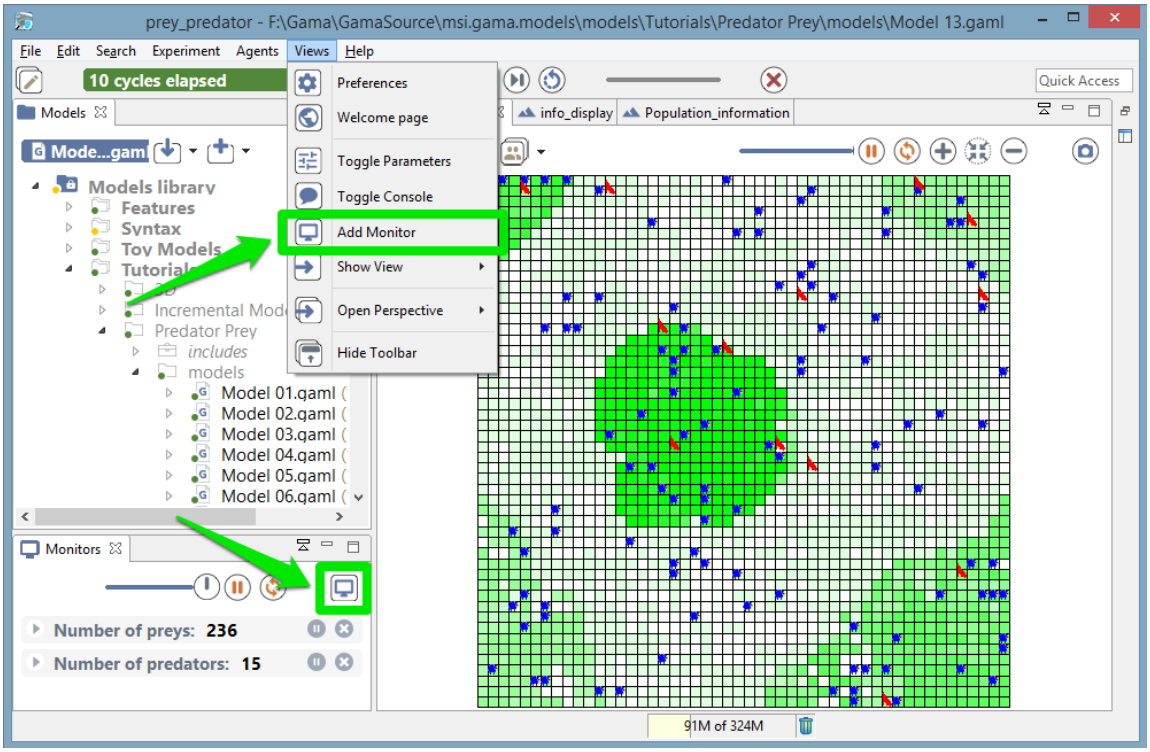


Figure 27.8: images/add_monitor.png

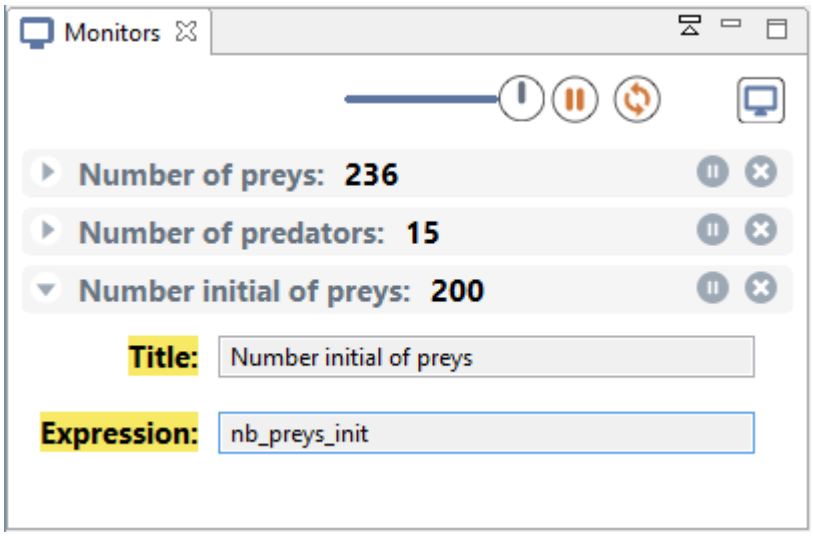


Figure 27.9: images/monitor_definition.png

Chapter 28

Displays

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

- java 2D displays
- OpenGL displays

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

Table of contents

Classical displays (java2D)

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

Figure 28.1: images/display-java2D.png

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

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

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

The Show/Hide side bar button opens a side panel in the display allowing the modeler to configure:

- **Properties** of the display: background and highlight color, display the scale bar
- For each layer, we can configure visibility, transparency, position and size of the layer. For grid layers, we can in addition show/hide grids. For species layers, we can also configure the displayed aspect. For text layers, we can the expression displayed with the color and the font.

The bottom overlay bar displays information about the way it is displayed:

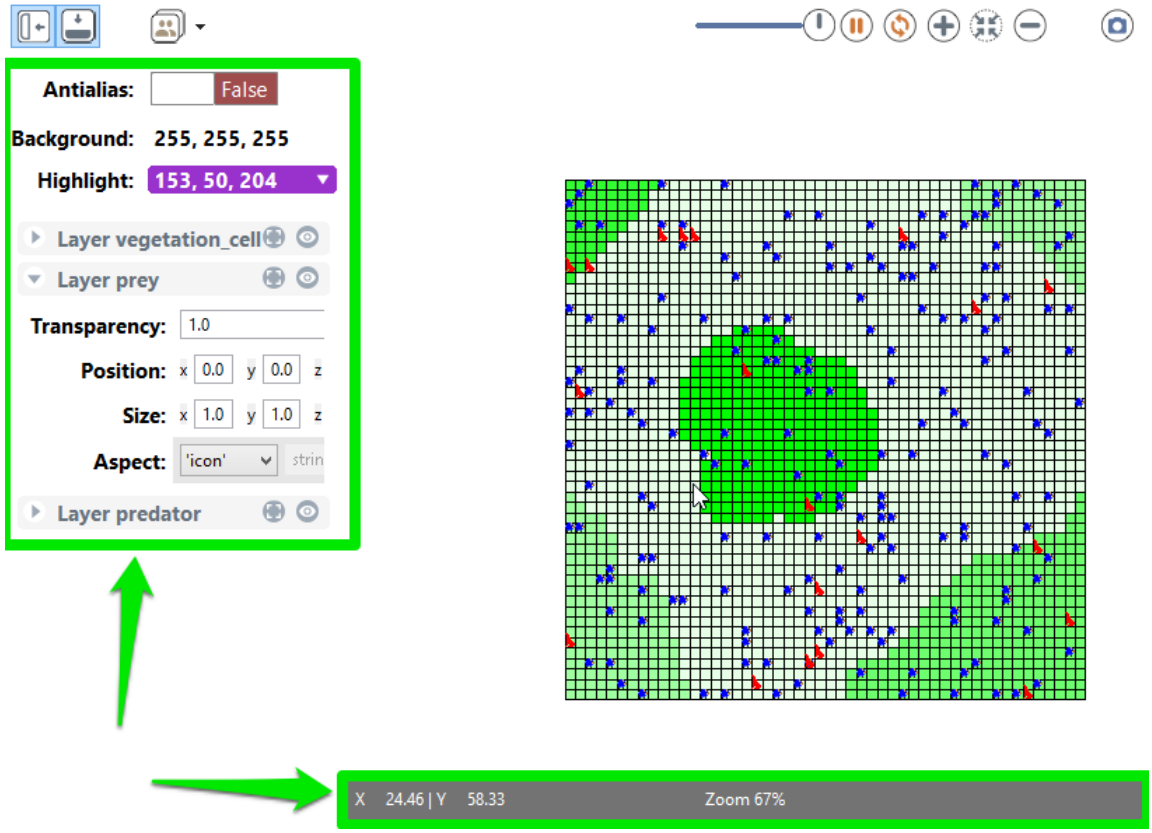


Figure 28.2: images/display-sidebar-overlay.png

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

OpenGL displays

The OpenGL display has an additional button **3D Options** providing 3D features:

- **Use FreeFly camera/Use Arcball camera**: switch between cameras, the default camera is the Arcball one,
- **Use mouse to rotate/Use mouse to drag** (only with Arcball camera): use left click for one of the 2 actions, left click + Ctrl for the other of the 2 actions.

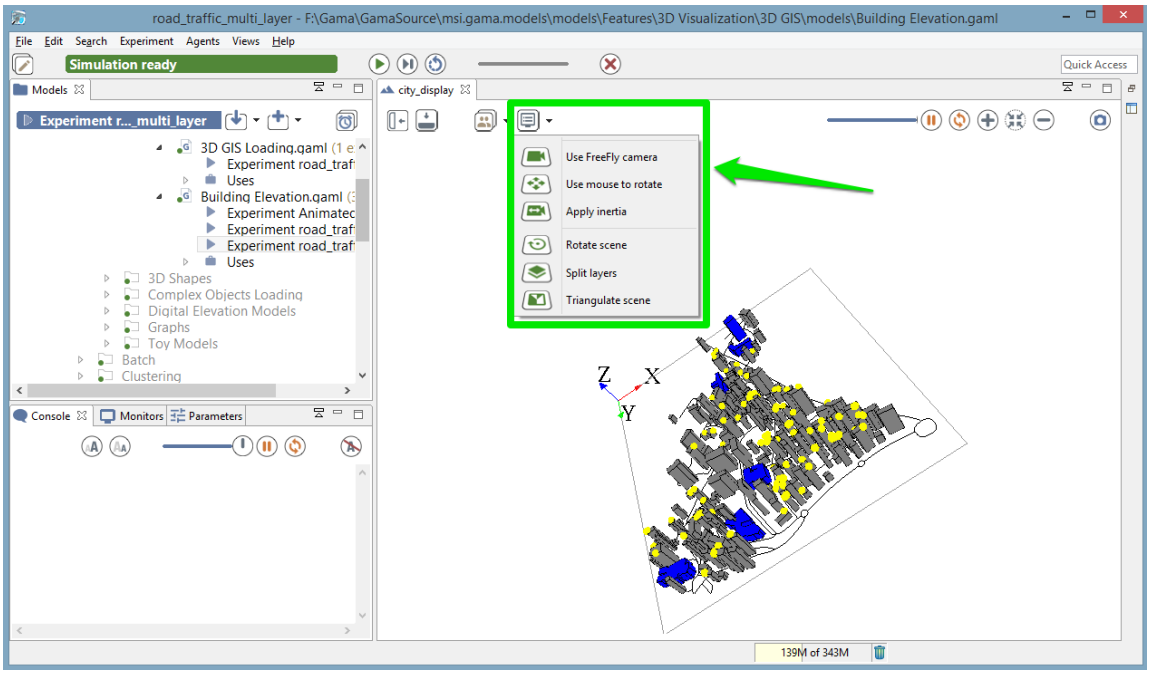


Figure 28.3: images/display-OpenGL.png

- **Apply inertia** (only with Arcball camera): in inertia mode, when the modeler stops moving the camera, there is no straight halt but a kind of inertia.
- **Rotate scene**: rotate the scene around an axis orthogonal to the scene,
- **Split layers/Merge layers**: display each layer at a distinct height,
- **Triangulate scene**: display the polygon primitives.

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

FreeFly camera commands

Key	Function
Double Click	Zoom Fit
+	Zoom In
-	Zoom Out
Up	Move forward
Down	Move backward

Key	Function
Left	Strafe left
Right	Strafe right
SHIFT+Up	Look up
SHIFT+Down	Look down
SHIFT+Left	Look left
SHIFT+Right	Look right
MOUSE	Makes the camera look up, down, left and right
MouseWheel	Zoom-in/out to the current target (center of the screen)

ArcBall camera commands

Key	Function
Double Click	Zoom Fit
+	Zoom In
-	Zoom Out
Up	Horizontal movement to the top
Down	Horizontal movement to the bottom
Left	Horizontal movement to the left
Right	Horizontal movement to the right
SHIFT+Up	Rotate the model up (decrease the phi angle of the spherical coordinates)
SHIFT+Down	Rotate the model down (increase the phi angle of the spherical coordinates)
SHIFT+Left	Rotate the model left (increase the theta angle of the spherical coordinates)
SHIFT+Right	Rotate the model right (decrease the theta angle of the spherical coordinates)
SPACE	Reset the pivot to the center of the envelope
KEYPAD 2,4,6,8	Quick rotation (increase/decrease phi/theta by 30°)
CMD+MOUSE1	Makes the camera rotate around the model
ALT+LEFT_-MOUSE	Enables ROI Agent Selection
SHIFT+LEFT_-MOUSE	Enables ROI Zoom

Key	Function
SCROLL	Zoom-in/out to the current target (center of the sphere)
WHEEL CLICK	Reset the pivot to the center of the envelope

Chapter 29

Batch Specific UI

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

Table of contents

Information bar

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

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

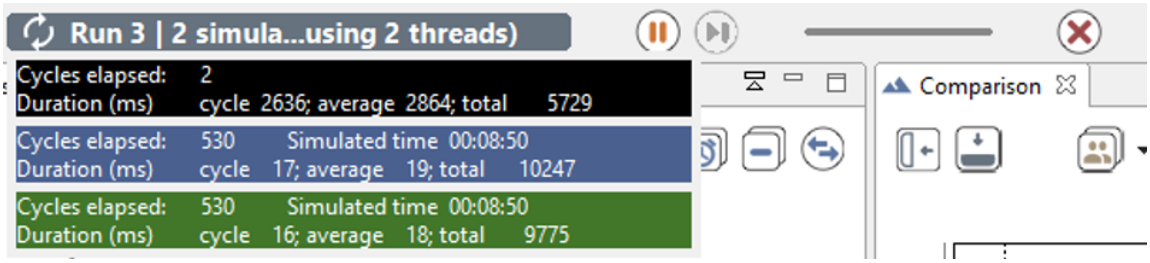


Figure 29.1: images/batch_Information_bar.png

Batch UI

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

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

   method exhaustive maximize: food_gathered;
```

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

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

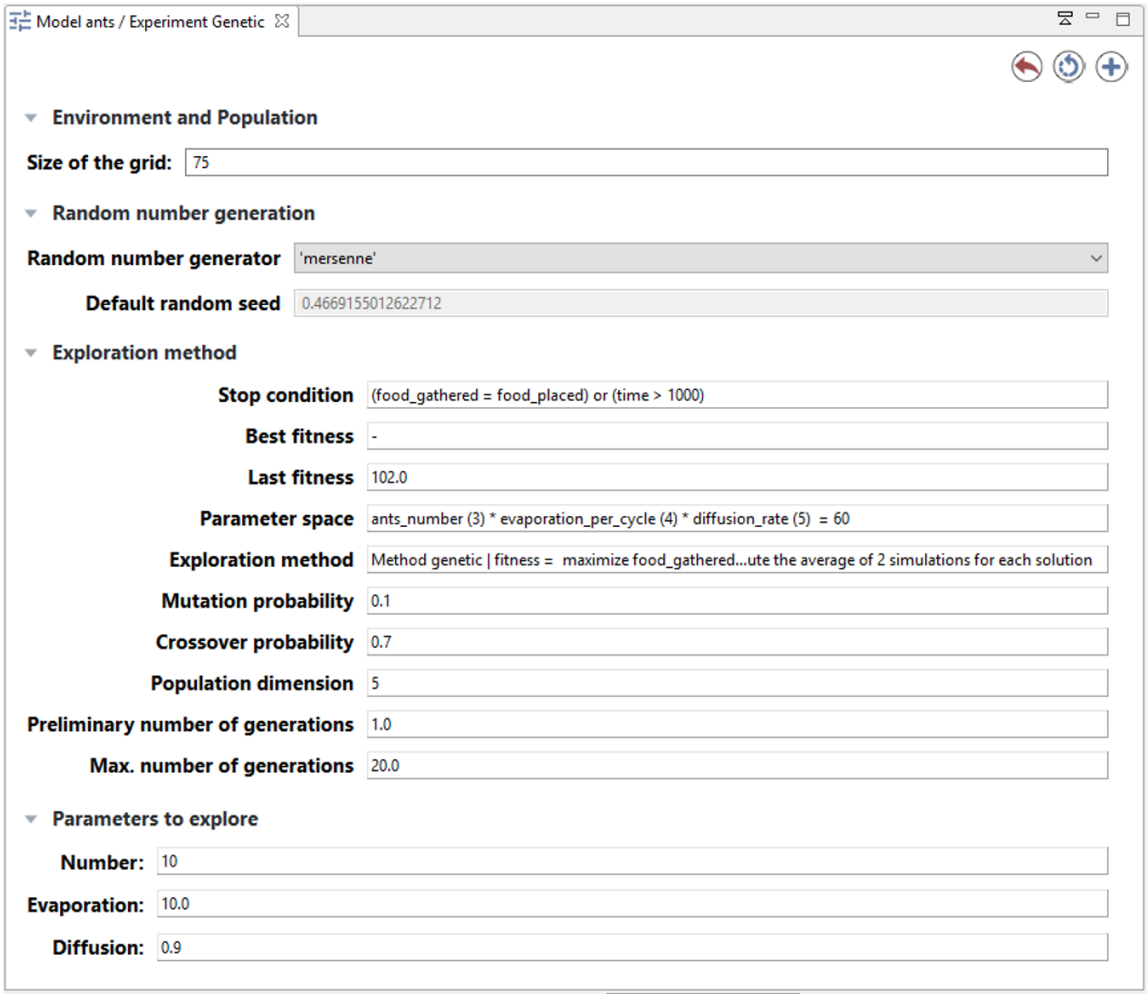


Figure 29.2: images/batch_Parameters_pane.png

Chapter 30

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.

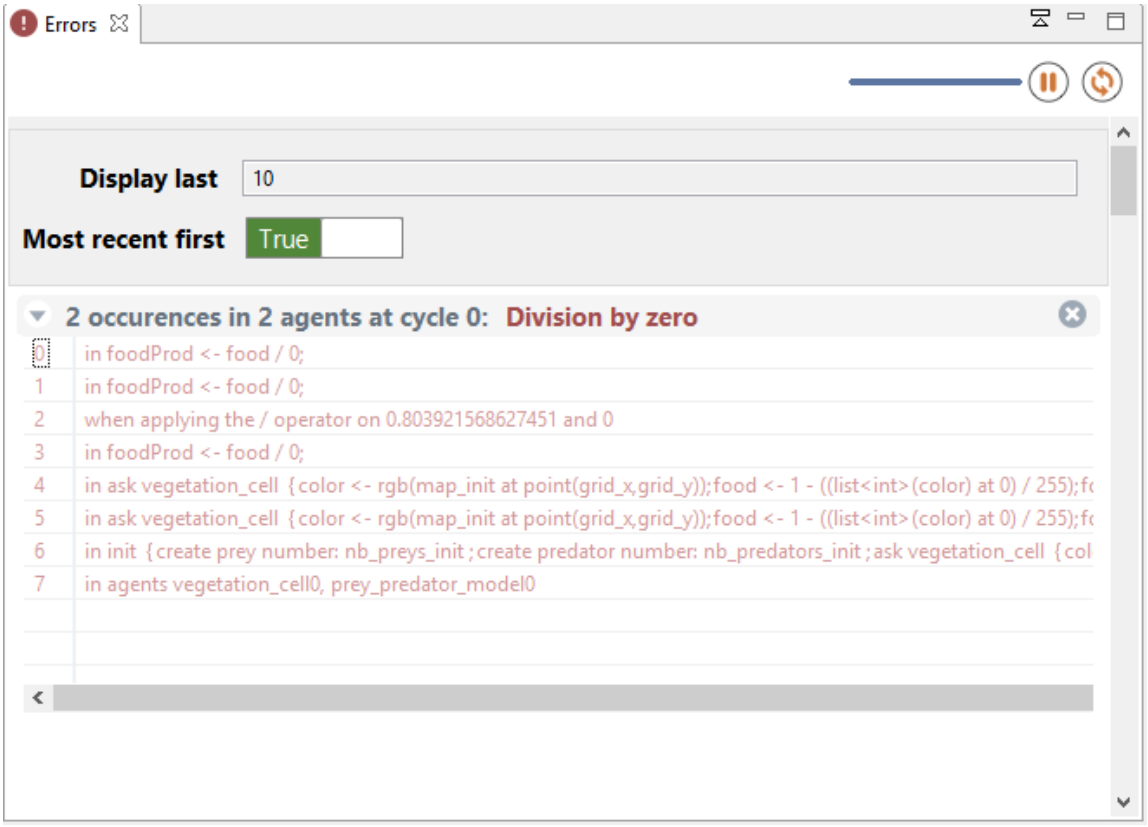


Figure 30.1: images/errors_view.png

Chapter 31

Preferences

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

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

Table of contents

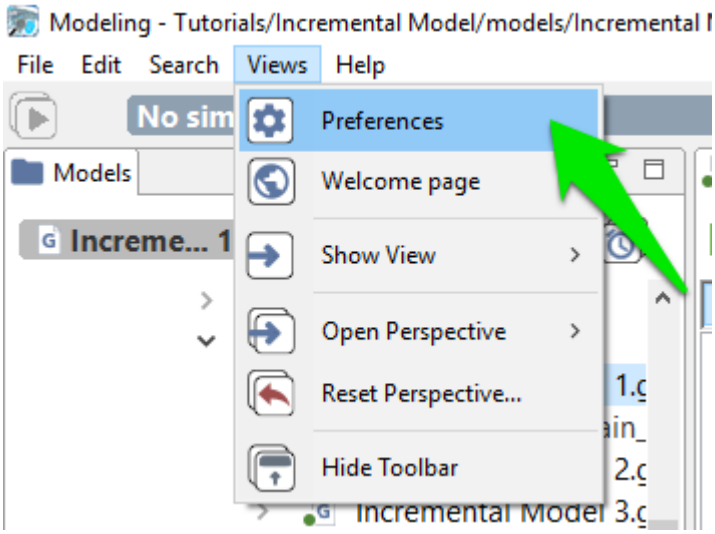


Figure 31.1: images/open_prefs.png

Opening Preferences

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

Simulation

- **Random Number Generation**: all the options pertaining to generating random numbers in simulations
  - Random Number Generator: the name of the generator to use by default (if none is specified in the model).
  - Define a default seed: whether or not a default seed should be used if none is specified in the model (otherwise it is chosen randomly by GAMA)
  - Default Seed value: the value of this default seed
  - Include in the parameters of models: whether the choice of generator and seed is included by default in the parameters views of experiments or not.
- **Errors**: how to manage and consider simulation errors

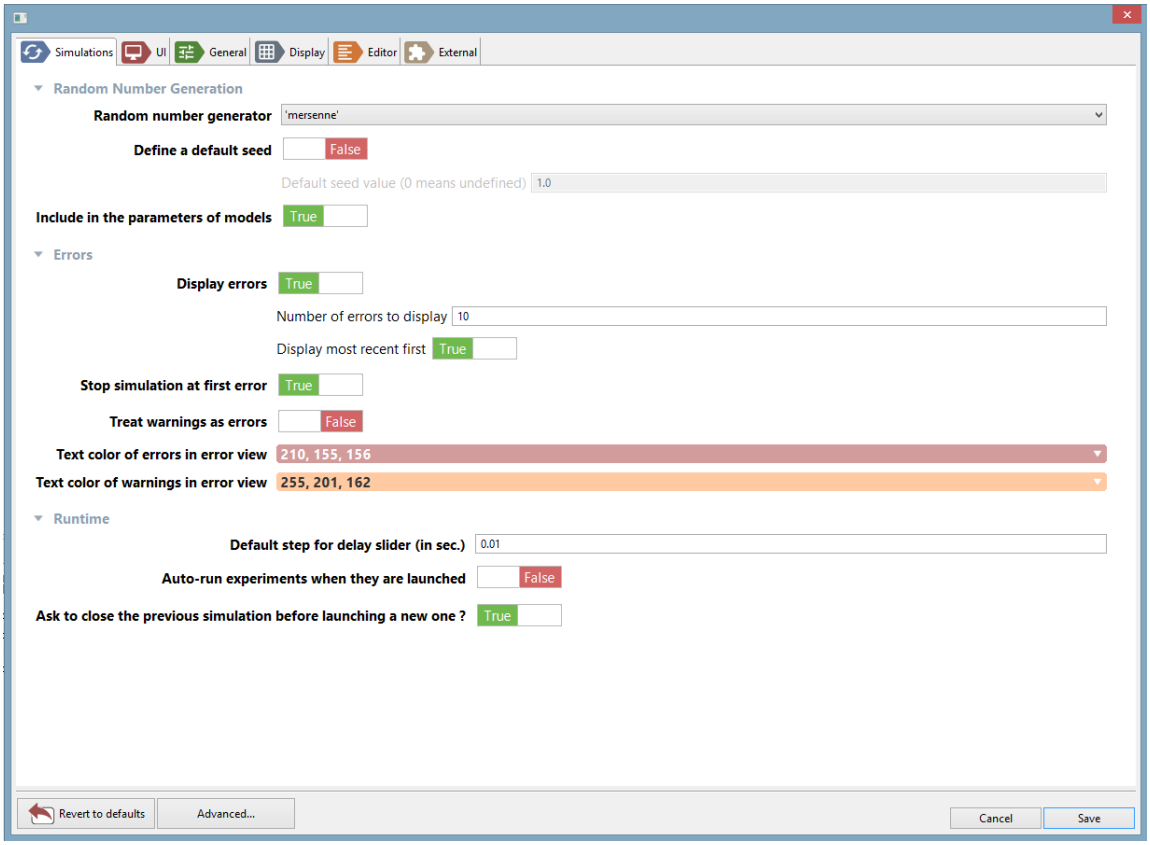


Figure 31.2: images/simulation.png

  - Display Errors: whether errors should be displayed or not.
  - Number of errors to display: how many errors should be displayed at once
  - Display most recent first: errors will be sorted in the inverse chronological order if true.
  - Stop simulation at first error: if false, the simulations will display the errors and continue (or try to).
  - Treat warnings as errors: if true, no more distinction is made between warnings (which do not stop the simulation) and errors (which can potentially stop it.

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

UI

- **Menus**
  - Break down agents in menu every: when inspecting a large number of agents, how many should be displayed before the decision is made to separate the population in sub-menus.
  - Sort colors menu by
  - Sort operators menu by
- **Console**
  - Max. number of characters to display in the console (-1 means no limit)
  - Max. number of characters to keep in memory when console is paused (-1 means no limit)
- **Icons**
  - Icons and buttons dark mode (restart to see the change): Change the highlight for the icons and the button.
  - Size of icons in the UI (restart to see the change): Size of the icons in pixel
- **Viewers**

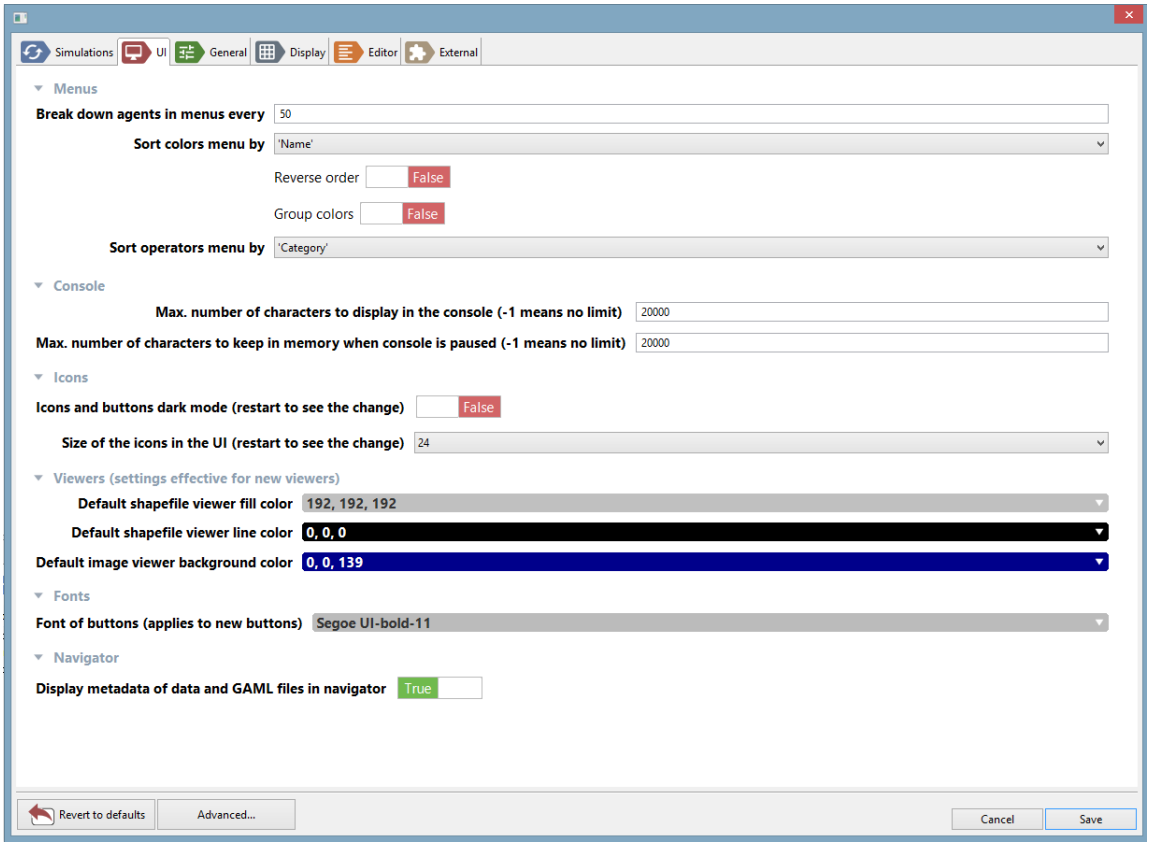


Figure 31.3: images/UI.png

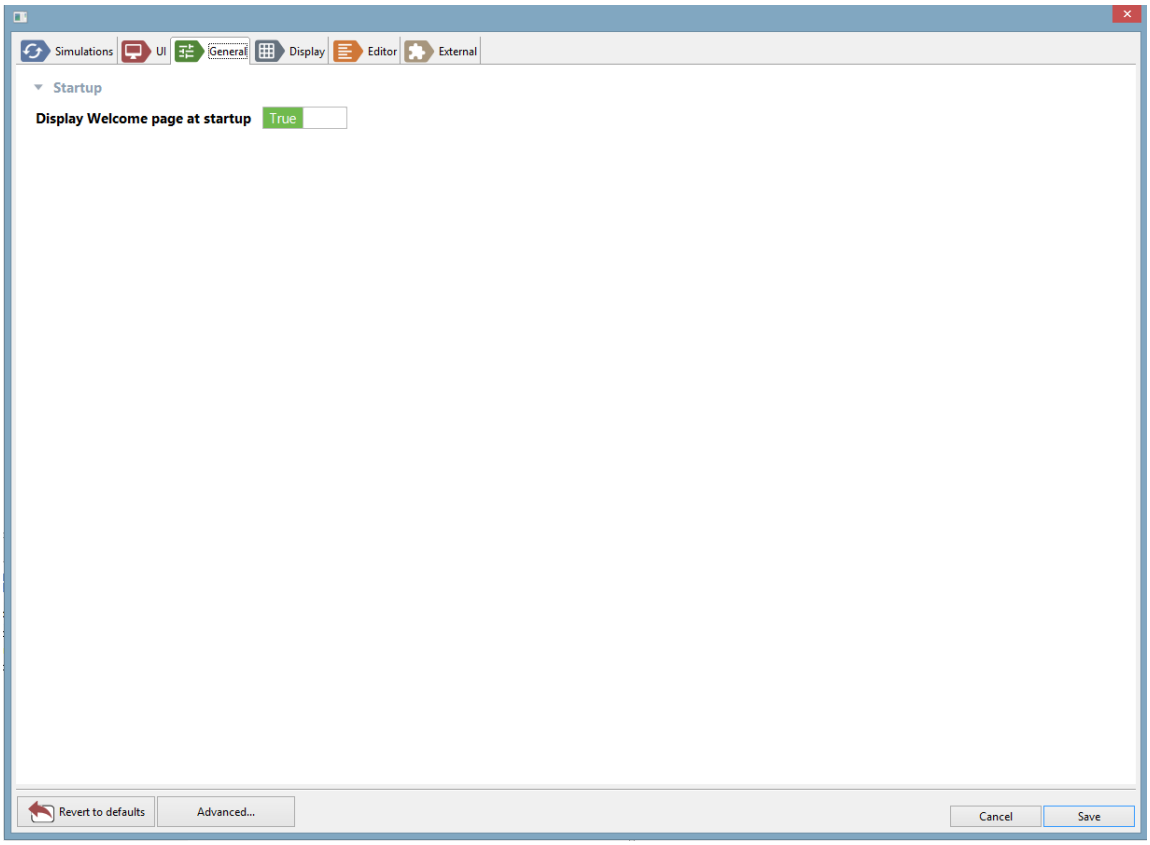


Figure 31.4: images/general.png

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

General

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

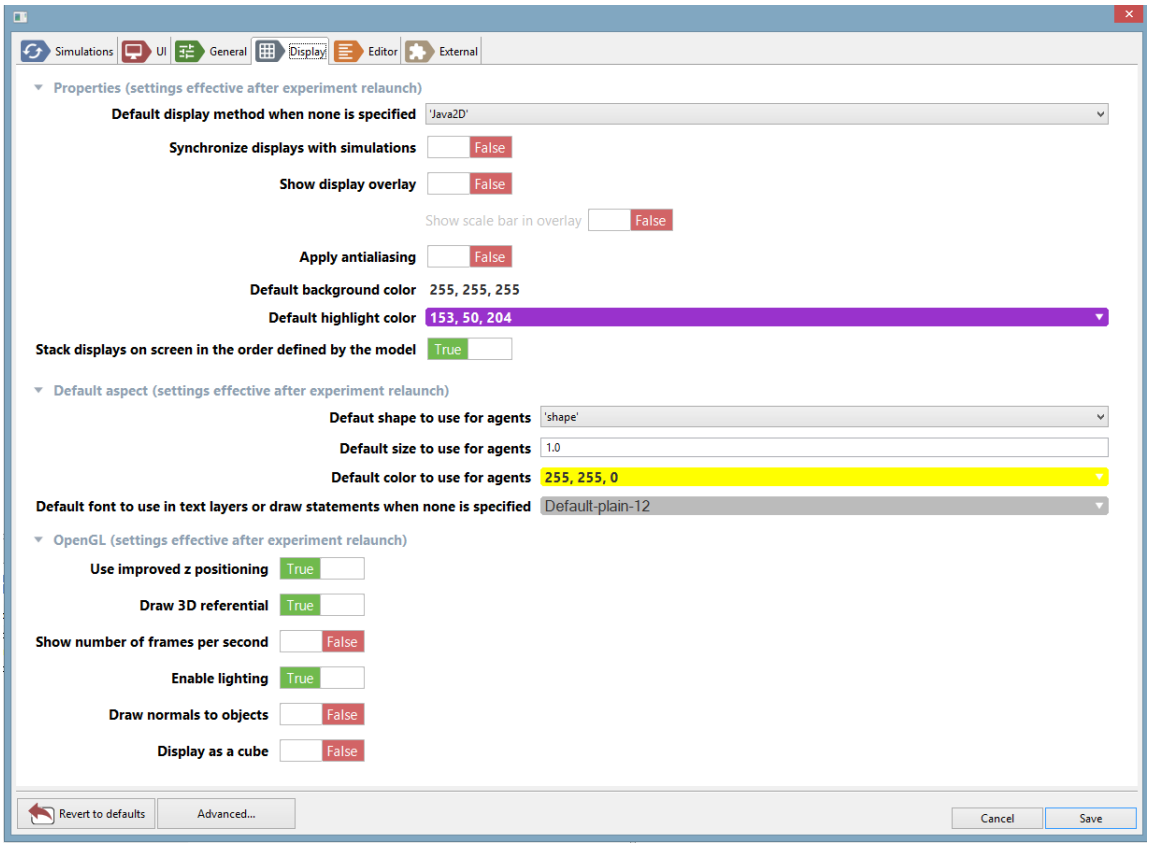


Figure 31.5: images/display.png

Display

- **Properties**: various properties of displays
  - Default display method: use either 'Java2D' or 'OpenGL' if nothing is specified in the declaration of a display.
  - Synchronize displays with simulations: if true, simulation cycles will wait for the displays to have finished their rendering before passing to the next cycle (this setting can be changed on an individual basis dynamically here).
  - Show display overlay: if true, the bottom overlay is visible when opening a display.
  - Show scale bar in overlay: if true, the scale bar is displayed in the bottom overlay.
  - Apply antialiasing: if true, displays are drawn using antialiasing, which is

slower but renders a better quality of image and text (this setting can be changed on an individual basis dynamically here).
  - Default background color: indicates which color to use when none is specified in the declaration of a display.
  - Default highlight color: indicates which color to use for highlighting agents in the displays.
  - Stack displays on screen...: if true, the display views, in case they are stacked on one another, will put the first display declared in the model on top of the stack.

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

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

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

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

Editor

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

- **Options**

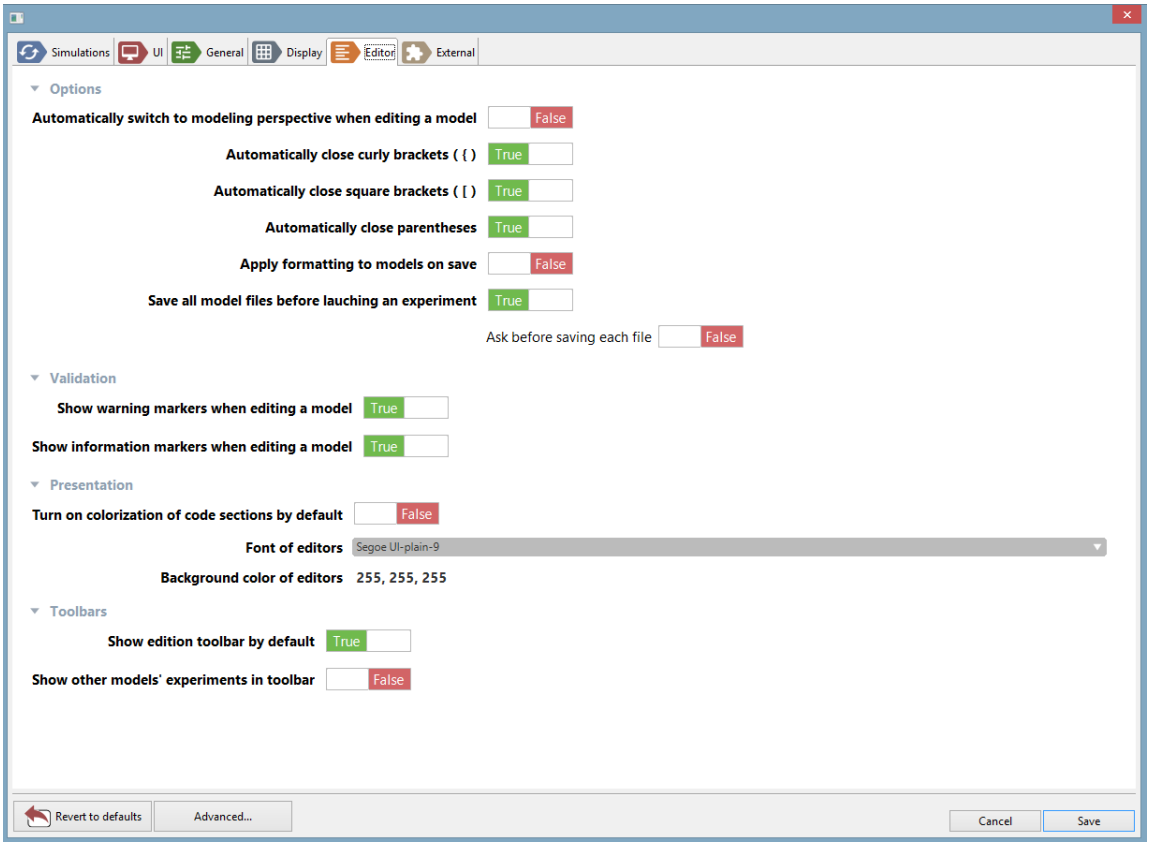


Figure 31.6: images/editor.png

  - Automatically switch to Modeling Persepective: if true, if a model is edited in the Simulation Perspective, then the perspective is automatically switched to Modeling (*inactive for the moment*)
  - Automatically close curly brackets ({)
  - Automatically close square brackets (])
  - Automatically close parenthesis
  - Mark occurrences of symbols in models: if true, when a symbol is selected in a model, all its occurrences are also highlighted.
  - Applying formatting to models on save: if true, every time a model file is saved, its code is formatted.
  - Save all model files before launching an experiment
  - Ask before saving each file

- **Validation**
  - Show warning markers when editing a model
  - Show information markers when editing a model

- **Presentation**
  - Turn on colorization of code sections by default
  - Font of editors
  - Background color of editors

- **Toolbars**
  - Show edition toolbar by default
  - Show other models' experiments in toolbar: if true, you are able to launch other models' experiments from a particular model.

External

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

- **Paths**
  - Path to Spatialite: the path to the Spatialite library (http://www.gaia-gis.it/gaia-sins/) in the system.
  - Path to RScript: the path to the RScript library (http://www.r-project.org) in the system.

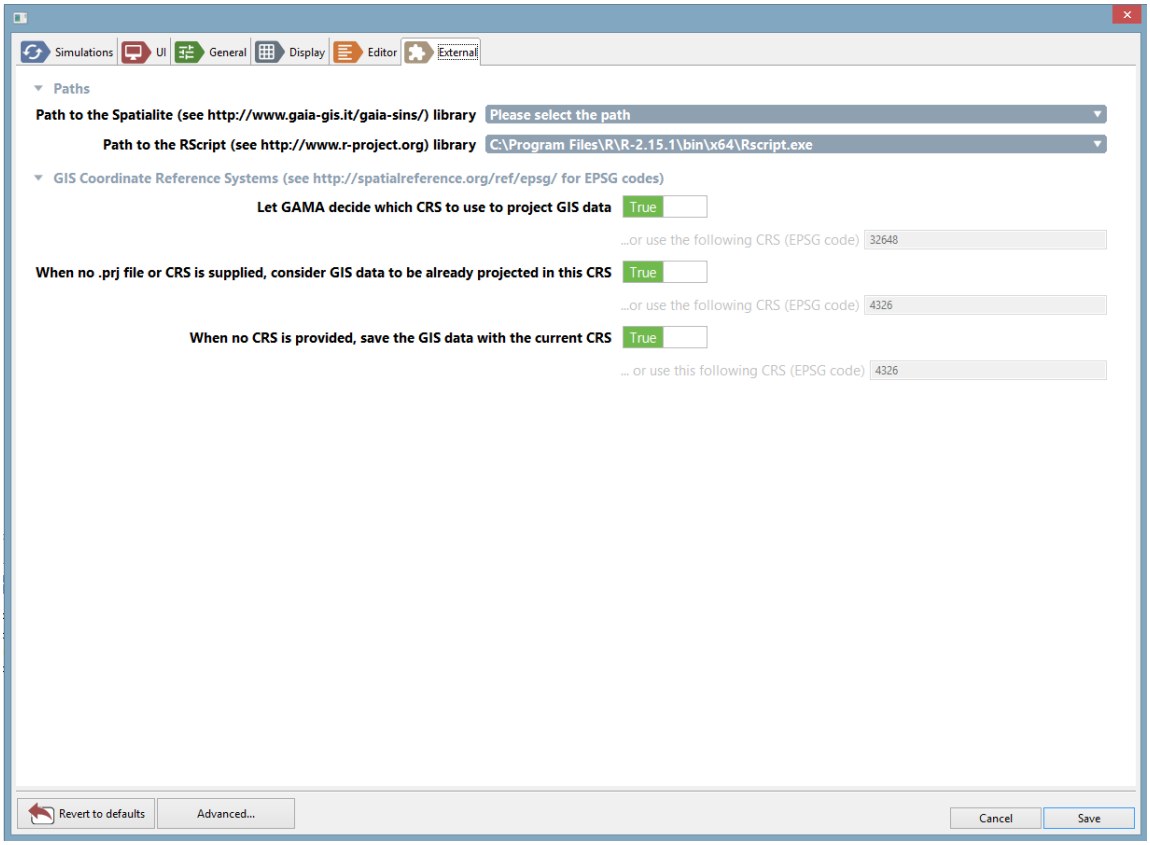


Figure 31.7: images/external.png

Figure 31.8: images/advanced.png

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

Advanced Preferences

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

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

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

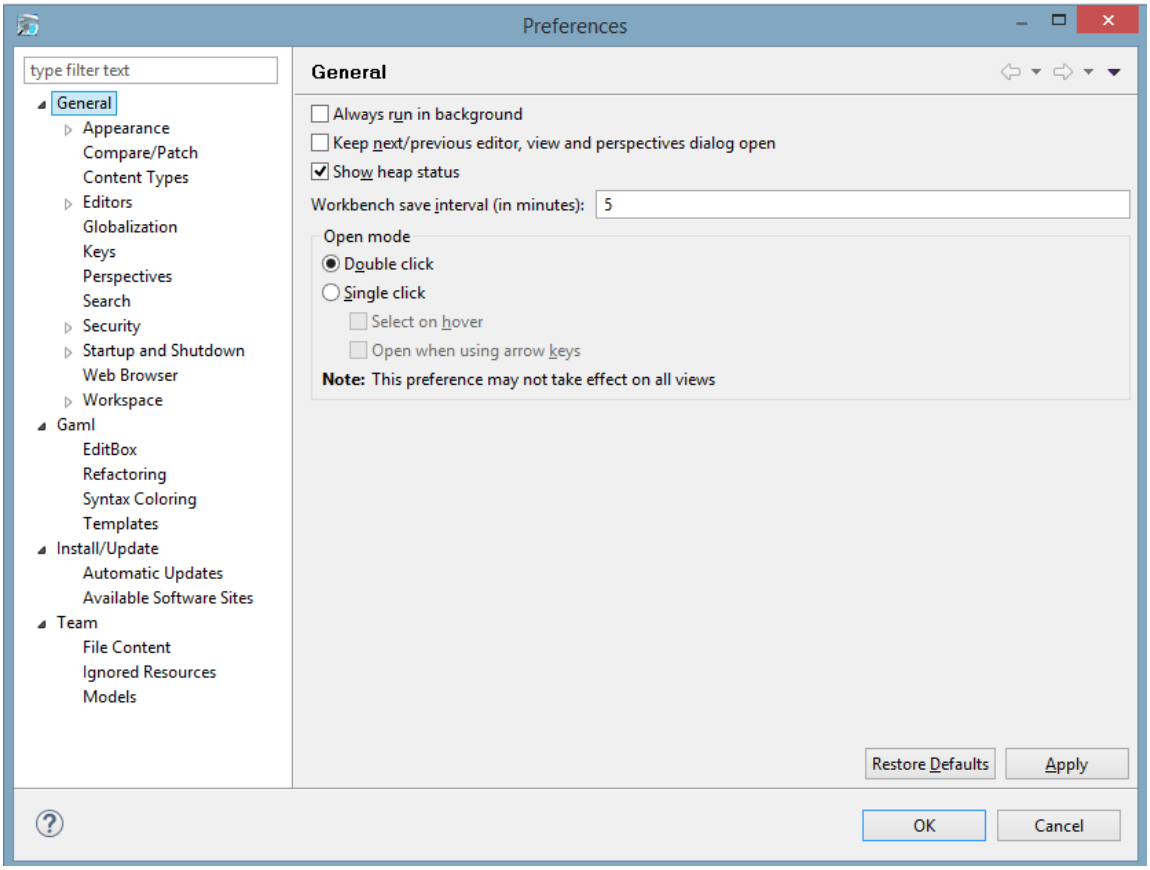


Figure 31.9: images/advanced_2.png

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).

Part VI

Learn GAML step by step

Chapter 32

Learn GAML Step by Step

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

How to proceed to learn better?

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

Although, if you really want to learn about a specific topic, our 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!

Chapter 33

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

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*).

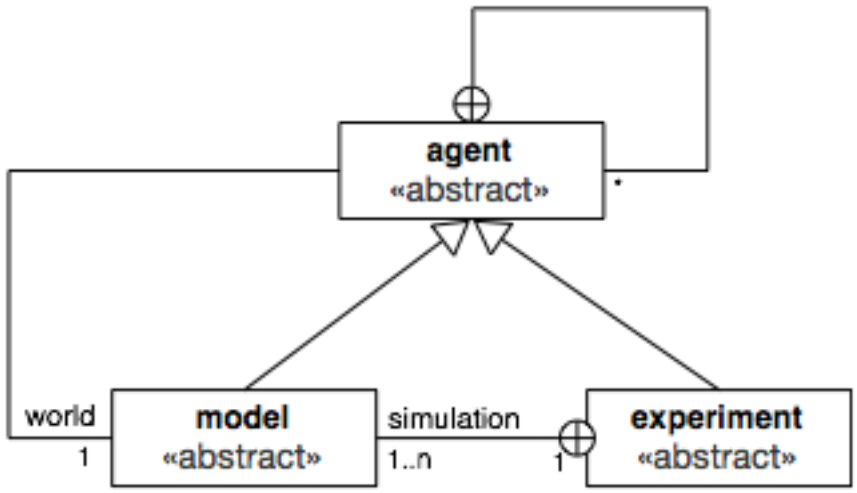


Figure 33.1: framework.png

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

Putting this all together, writing a model in GAML then consists in defining a species

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

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

Translation into a concrete syntax

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

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

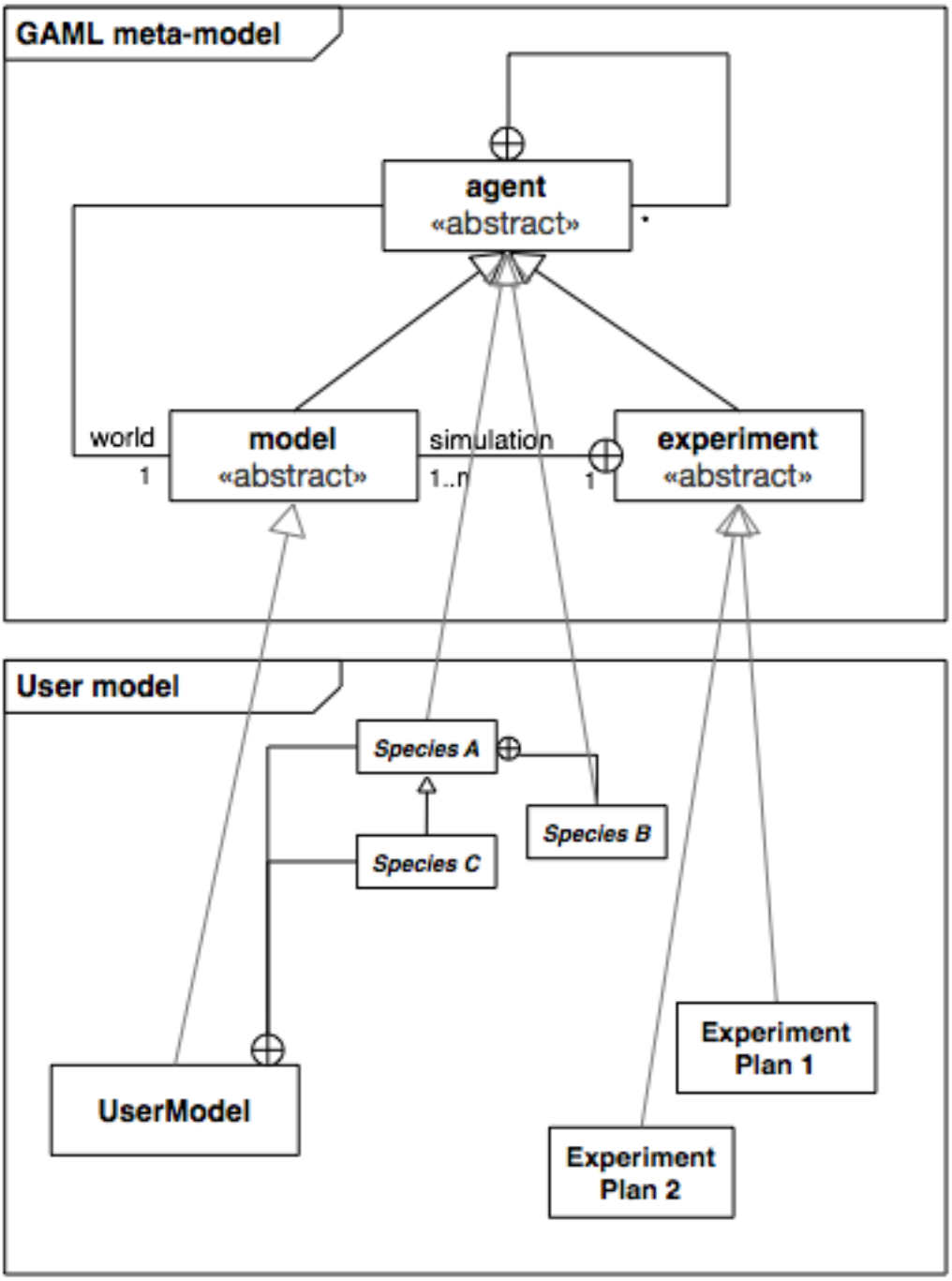


Figure 33.2: user_model.png

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)
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	-

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

Chapter 34

Start with GAML

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

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

Chapter 35

Organization of a model

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

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

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

Table of contents

Model Header (*model species*)

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

```
model name_of_the_model
```

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

```
model dummy
```

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

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

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

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

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

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

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

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

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

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

Species declarations

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

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

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

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

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

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

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

species A {...}

species B {...}
```

is completely equivalent to:

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

    species A {...}

    species B {...}
}
```

Experiment declarations

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

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

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

experiment second_experiment {
    // definition of parameters (inputs)

    // definition of output
}
```

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

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

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

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

Basic skeleton of a model

Here is the basic skeleton of a model :

```
model name_of_the_model

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

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

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

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

Chapter 36

Basic programming concepts in GAML

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

Index

- 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` scope, or inside the `species` scope.

```
typeName myVariableName;
```

Basic types

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

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

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

```
model firstModel

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

```
        write booleanVariable;
    }
}

experiment myExperiment
{
}
```

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

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

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

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

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

The point type

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

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

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

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

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

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

A world about dimensions

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

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

Declare variables using facet

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

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

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

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

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

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

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

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

Operators in GAMA

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

- Mathematical operators

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

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

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

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

Logical operators

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

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

Comparison operators

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

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

Type casting operators

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

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

Other operators

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

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

Some others are used in a more intuitive way:

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

Conditional structures

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

```
if (integerVariable<0) {
    write "my value is negative !! The exact value is " +
   integerVariable;
}
else if (integerVariable>0) {
    write "my value is positive !! The exact value is " +
   integerVariable;
}
else if (integerVariable=0) {
```

```
    write "my value is equal to 0 !!";
}
else {
    write "hey... This is not possible, right ?";
}
```

GAML also accepts ternary operator:

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

Loop

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

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

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

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

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

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

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

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

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

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

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

Manipulate containers

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

How to declare a list?

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

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

How to know the number of elements of a list?

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

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

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

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

How to get an element from a list?

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

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

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

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

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

How to know if an element exists in a list?

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

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

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

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

```
list<int> list_int <- [1,5,7,6,7];
remove from:list_int index:1; // remove the 2nd element of the list
write list_int; // the output is : [1,7,6,7]
remove item:7 from:list_int; // remove the 1st occurrence of 7
write list_int; // the output is : [1,6,7]
add item:9 to: list_int at: 2; // add 9 in the 3rd position
write list_int; // the output is : [1,6,9,7]
add 0 to: list_int; // add 0 in the last position
write list_int; // the output is : [1,6,9,7,0]
put 3 in: list_int at: 0; // put 3 in the 1st position
write list_int; // the output is : [3,6,9,7,0]
put 2 in: list_int key: 2; // put 2 in the 3rd position
write list_int; // the output is : [3,6,2,7,0]
```

How to add 2 lists?

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

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

How to browse a list?

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

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

How to filter a list?

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

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

Other useful operators for the manipulation of lists:

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

Random values

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

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

```
int var0 <- rnd (2);    // var0 equals 0, 1 or 2
float var1 <- rnd (1000) / 1000;    // var1 equals a float between 0
    and 1 with a precision of 0.001
```

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

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

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

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

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

You can use probabilistic laws, using operators such as **gauss**, **poisson**, **binomial**, or **truncated_gauss** (we invite you to read the documentation for those operators).

Chapter 37

Manipulate basic species

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

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

Chapter 38

The global species

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

Index

Declaration

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

```
global {
```

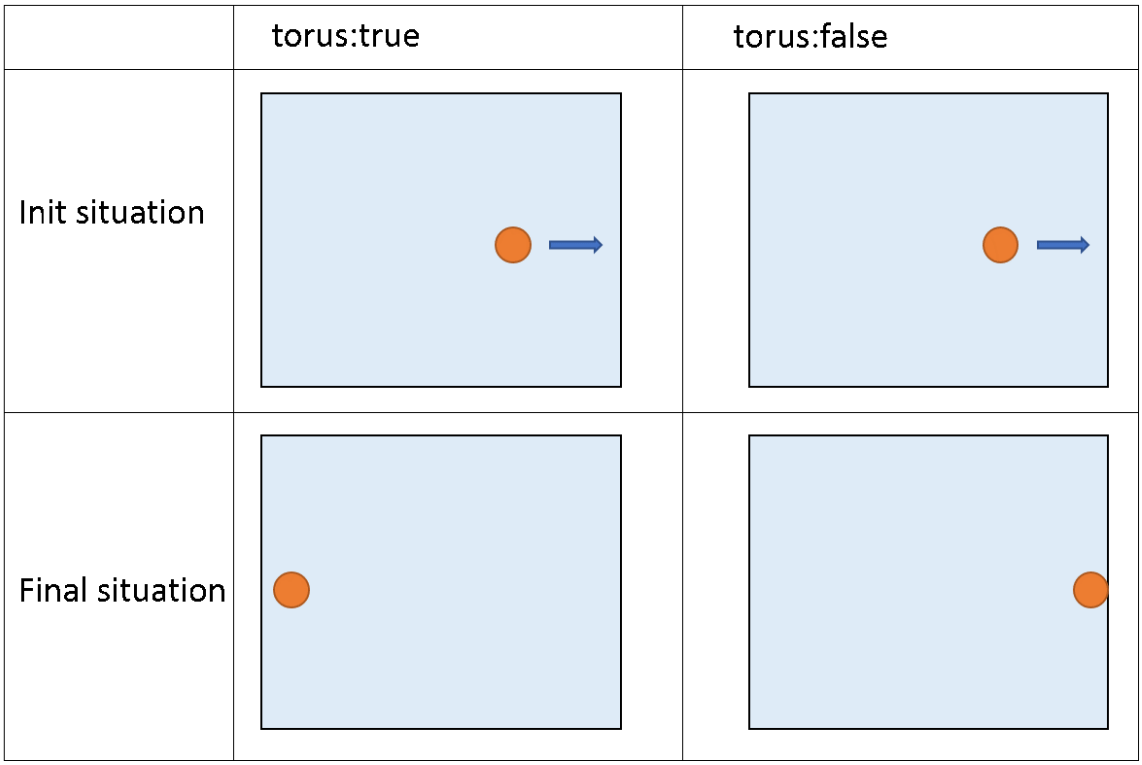


Figure 38.1: images/torus.png

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

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

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

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

Directly in the `global` scope, you have to declare all your global attributes (can be seen as "static members" in Java or C++). To declare them, proceed exactly as for

declaring basic variables. Those attributes are accessible wherever you want inside the species scope.

Environment size

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

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

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

Built-in attributes

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

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

world

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

cycle

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

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

step

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

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

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

time

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

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

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

duration

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

total_duration

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

average_duration

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

machine_time

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

agents

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

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

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

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

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

Built-in Actions

The global species is provided with two specific actions.

halt

- stops the simulation.

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

pause

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

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

The init statement

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

Chapter 39

Regular species

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

Index

Declaration

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

```
species my_specie {
}
```

or:

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

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

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

Built-in attributes

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

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

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

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

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

```
name_of_your_species.attribute_you_want
```

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

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

Here is the list of those attributes:

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

Built-in action

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

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

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

The init statement

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

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

The aspect statement

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

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

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

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

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

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

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

Instantiate an agent

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

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


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

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

```
model display_one_agent
```

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

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

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

Chapter 40

Defining actions and behaviors

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

Index

Action

Declare an action

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

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

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

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

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

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

Call an action

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

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

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

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

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

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

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

Behavior

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

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

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

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

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

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

Example

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

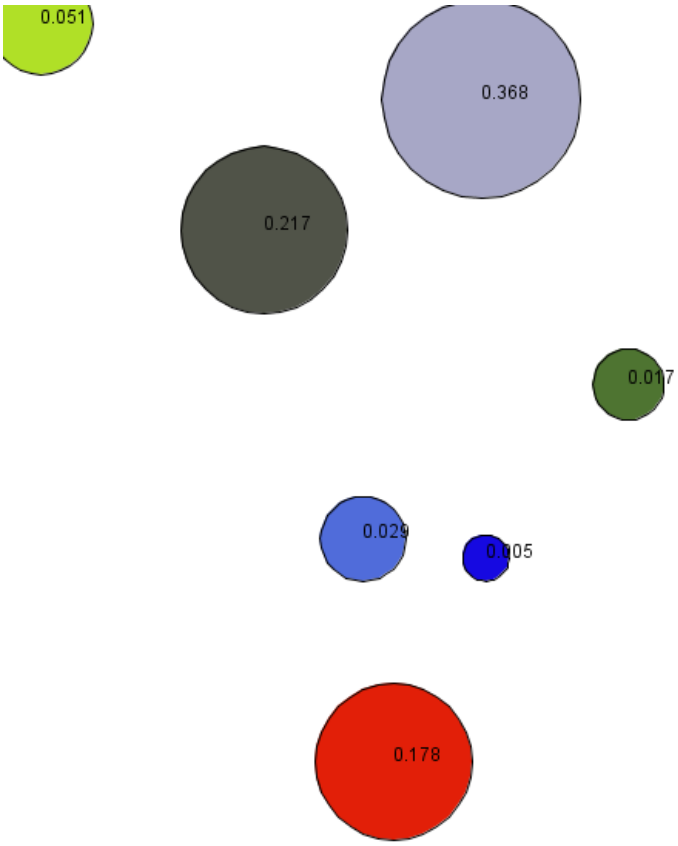


Figure 40.1: images/burst_the_baloon.png

are destroyed, the simulation stops. The volume of each balloon is displayed in the balloon position.

Here is one of the multiple possible implementation:

```
model burst_the_baloon

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

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

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

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

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

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

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

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

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

```
    }
}
```

Chapter 41

Interaction between agents

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

Index

The ask statement

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

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

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

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

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

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

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

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

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

```
        }
    }
}

species other_species {
}
```

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

```
ask other_species.population
```

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

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

Pseudo variables

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

```
species speciesA {
    init {
        name <- "speciesA";
    }
    reflex update {
        ask speciesB {
write name; // output : "speciesB"
write self.name; // output : "speciesB"
            write myself.name; // output : "speciesA"
        }
```

```
    }
}

species speciesB {
    init {
        name <- "speciesB";
    }
}
```

Now, if we introduce a third species, we can write an `ask` statement inside another.

```
species speciesA {
    init {
        name <- "speciesA";
    }
    reflex update {
        ask speciesB {
            write self.name; // output : "speciesB"
            write myself.name; // output : "speciesA"
            ask speciesC {
                write self.name; // output : "speciesC"
                write myself.name; // output : "speciesB"
            }
        }
    }
}

species speciesB {
    init {
        name <- "speciesB";
    }
}

species speciesC {
    init {
        name <- "speciesC";
    }
}
```

Nb: try to avoid multiple imbrications of ask statements. Most of the time, there is another way to do the same thing.

Some useful interaction operators

The operator **at_distance** can be used to know the list of agents that are in a certain distance from another agent.

```
species my_species {
    reflex update {
        list<agent> 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 fix number of agents with a circle shape. They can move randomly on the environment, and when they are close enough from another agent, a line is displayed between them. This line is destroyed when the distance between the two agents is too important. Hint: use the operator **polyline** to construct a line. List the points between angle brackets `[]`.

Here is one example of implementation:

```
model connect_the_neighbors

global{
```

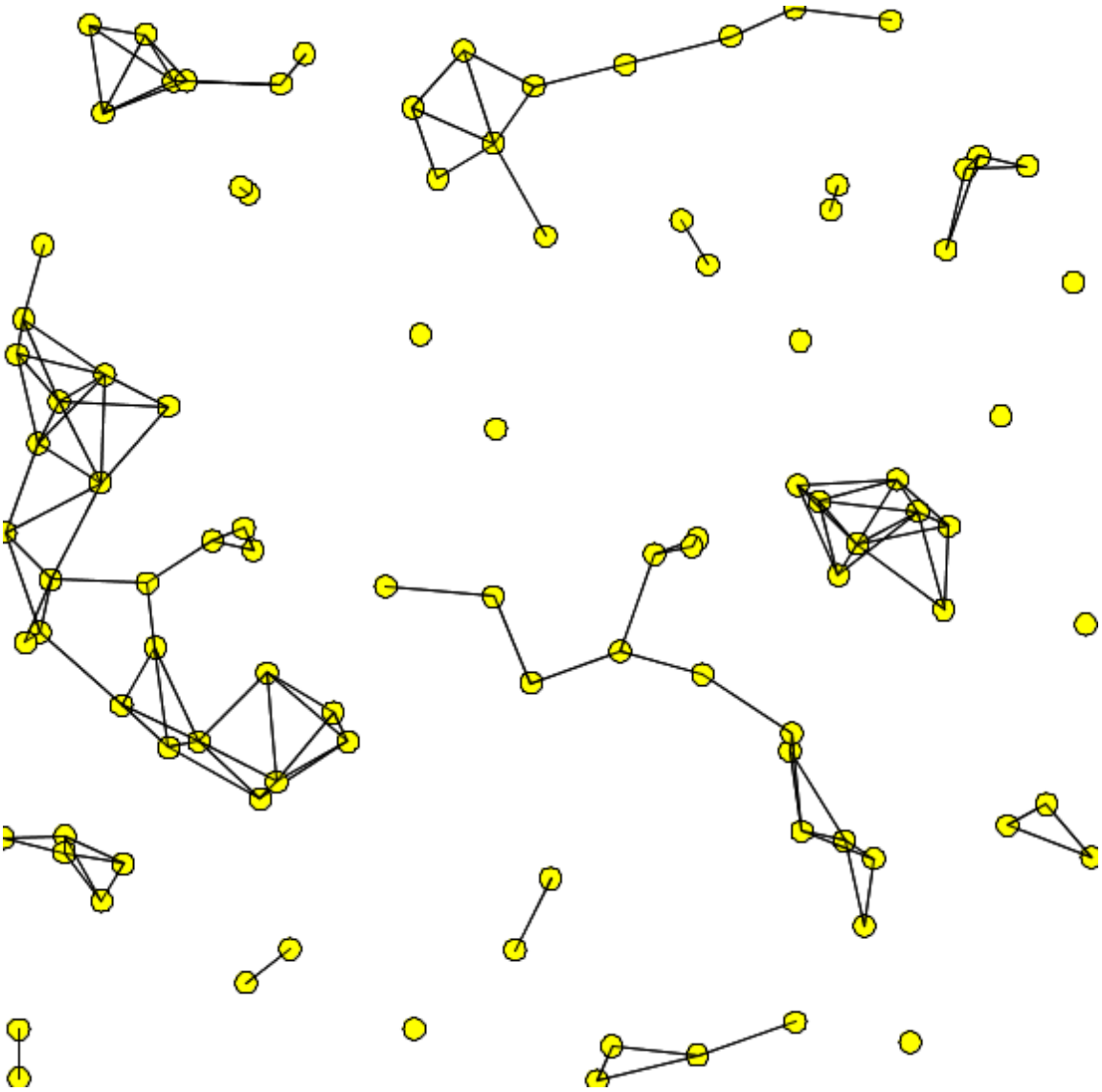


Figure 41.1: images/connect_the_neighbors.png

```
    float speed <- 0.2;
    float distance_to_intercept <- 10.0;
    int number_of_circle <- 100;
    init {
        create my_species number:number_of_circle;
    }
}

species my_species {
    reflex move {
        location <- {location.x+rnd(-speed,speed),location.y+rnd(-speed
   ,speed)};
    }
    aspect default {
        draw circle(1);
        ask my_species at_distance(distance_to_intercept) {
            draw polyline([self.location,myself.location]) color:#black
   ;
        }
    }
}

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

Chapter 42

Attaching Skills

GAMA allows to attach skills to agents through the facet **skills**. Skills are built-in modules that provide a set of related built-in attributes and built-in actions (in addition to those already proposed by GAMA) to the species that declare them.

Index

Skills

A declaration of skill is done by filling the **skills** facet in the species definition:

```
species my_species skills: [skill1,skill2] {
}
```

A very useful and common skill is the `moving` skill.

```
species my_species skills: [moving] {
}
```

Once your species has the moving skill, it earns automatically the following attributes: `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `follow`, `wander` and `wander_3D`.

Attributes:

- **`speed`** (float) designs the speed of the agent, in m/s.
- **`heading`** (int) designs the heading of an agent in degrees, which means that is the maximum angle the agent can turn around each step.
- **`destination`** (point) is the updated destination of the agent, with respect to its speed and heading. It's a read-only attribute, you can't change its value.

Actions:

`follow`

moves the agent along a given path passed in the arguments.

- returns: path
- **`speed`** (float): the speed to use for this move (replaces the current value of speed)
- **`path`** (path): a path to be followed.
- **`move_weights`** (map): Weights used for the moving.
- **`return_path`** (boolean): if true, return the path followed (by default: false)

`goto`

moves the agent towards the target passed in the arguments.

- returns: path
- **`target`** (agent,point,geometry): the location or entity towards which to move.

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **on** (graph): graph that restrains this move
- **recompute_path** (boolean): if false, the path is not recompute even if the graph is modified (by default: true)
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weights used for the moving.

move

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)
- **bounds** (geometry,agent): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

wander

Moves the agent towards a random location at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

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

- **amplitude** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)

- **bounds** (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

wander_3D

Moves the agent towards a random location (3D point) at the maximum distance (with respect to its speed). The heading of the agent is chosen randomly if no amplitude is specified. This action changes the value of heading.

- returns: path

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

- **amplitude** (int): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)

- **z_max** (int): the maximum altitude (z) the geometry can reach

- **bounds** (agent,geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

Other skills

A lot of other skills are available. Some of them can be built in skills, integrated by default in GAMA, other are linked to additional plugins.

This is the list of skills: `Advanced_driving`, `communication`, `driving`, `GAMASQL`, `graphic`, `grid`, `MDXSKILL`, `moving`, `moving3D`, `physical3D`, `skill_road`, `skill_road`, `skill_road_node`, `SQLSKILL`

Example

We can now build a model using the skill moving. Let's design 2 species, one is "species_red", the other is "species_green". Species_green agents are moving

randomly with a certain speed and a certain heading. Species_red agents wait for a species_green agent to be in a certain range of distance. Once it is the case, the agent move toward the species_green agent. A line link the red_species agent and its target.

Here is an example of implementation:

```
model green_and_red_species

global{
    float distance_to_intercept <- 10.0;
    int number_of_green_species <- 50;
    int number_of_red_species <- 50;
    init {
        create speciesA number:number_of_green_species;
        create speciesB number:number_of_red_species;
    }
}

species speciesA skills:[moving] {
    init {
        speed <- 1.0;
    }
    reflex move {
        do wander amplitude:90;
    }
    aspect default {
        draw circle(1) color:#green;
    }
}

species speciesB skills:[moving] {
    speciesA target;
    init {
        speed <- 0.0;
        heading <- 90;
    }
    reflex search_target when:target=nil {
        ask speciesA at_distance(distance_to_intercept) {
            myself.target <- self;
        }
    }
    reflex follow when:target!=nil {
        speed <- 0.8;
        do goto target:target;
    }
```

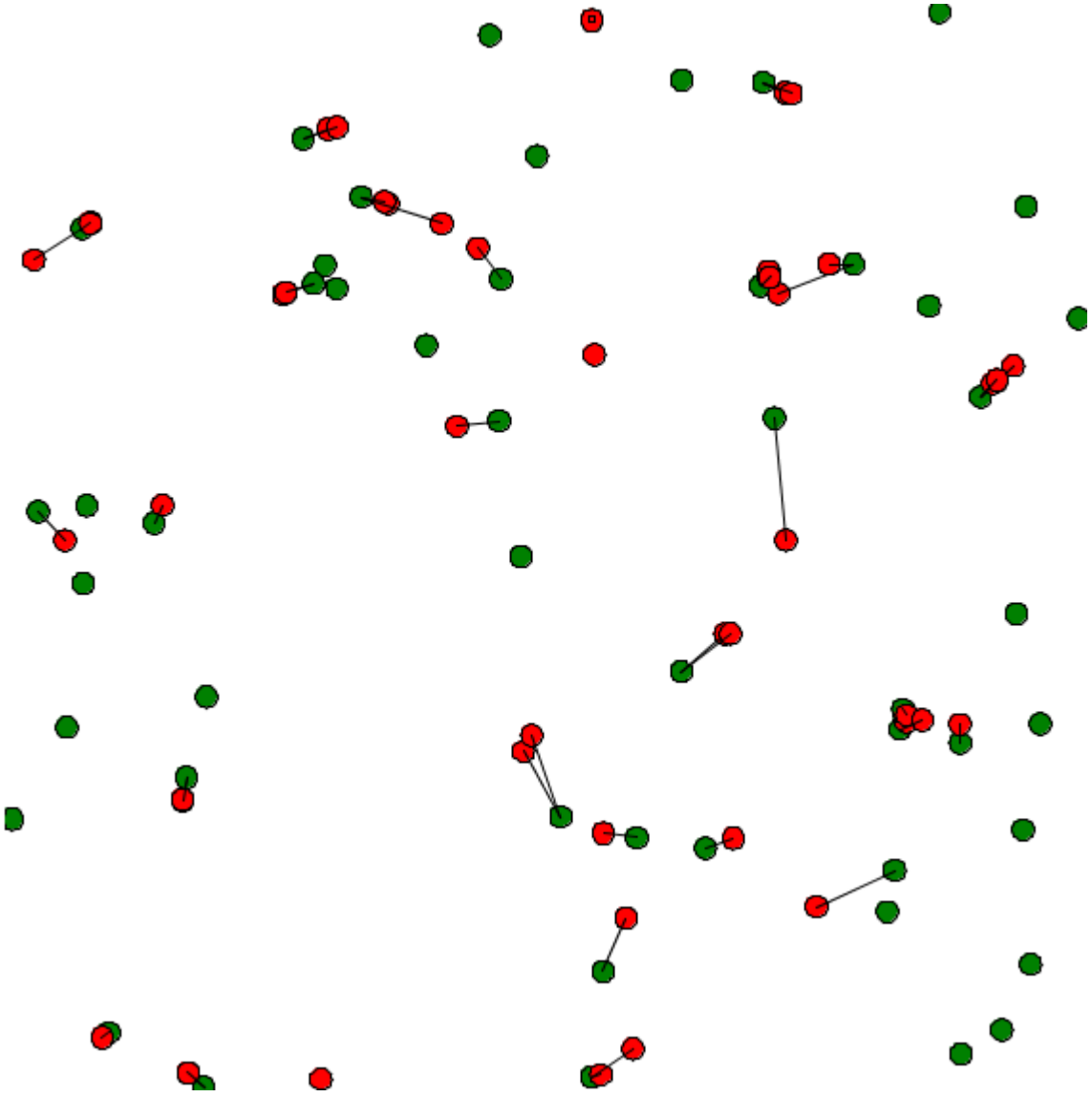


Figure 42.1: images/green_and_red_species.png

```
    aspect default {
        draw circle(1) color:#red;
        if (target!=nil) {
            draw polyline([self.location,target.location]) color:#black
   ;
        }
    }
}

experiment my_experiment type:gui
{
    output{
        display myDisplay {
            species speciesA aspect:default;
            species speciesB aspect:default;
        }
    }
}
```

Chapter 43

Inheritance

As for multiple programming languages, inheritance can be used in GAML. It is used to structure better your code when you have some complex models.

Index

Mother species / child species

To make a species inherit from a mother species, you have to add the facet **parent**, and specify the mother species.

```
species mother_species {
}

species child_species parent:mother_species {
}
```

Thus, all the attributes, actions and reflex of the mother species are inherited to the child species.

```
species mother_species {
    int attribute_A;
    action action_A {}
}

species child_species parent:mother_species {
    init {
        attribute_A <- 5;
        do action_A;
    }
}
```

If the mother species has a particular skill, its children will inherit all the attributes and actions.

```
species mother_species skills:[moving] {
}

species child_species parent:mother_species {
    init {
        speed <- 2.0;
    }
    reflex update {
        do wander;
    }
}
```

You can redefine an action or a reflex by declaring an action or a reflex with the same name.

Virtual action

You have also the possibility to declare a virtual action in the mother species, which means an action without implementation, by using the facet **virtual**:

```
action virtual_action virtual:true;
```

When you declare an action as virtual in a species, this species becomes abstract, which means you cannot instantiate agent from it. All the children of this species has to implement this virtual action.

```
species virtual_mother_species {
    action my_action virtual:true;
}

species child_species parent:virtual_mother_species {
    action my_action {
        // some statements
    }
}
```

Get all the subspecies from a species

If you declare a "mother" species, you create a "child" agent, then "mother" will return the population of agents "mother" and **not** the population of agents "child", as it is shown in the following example :

```
global
{
    init {
        create child number:2;
        create mother number:1;
    }
    reflex update {
        write length(mother); // will write 1 and not 3
    }
}

species mother {}

species child parent:mother {}
```

We 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 {}
```

Chapter 44

Defining advanced species

In the previous chapter, we saw how to declare and manipulate **regular species** and the **global species** (as a reminder, the instance of the **global species** is the **world agent**).

We will now see that GAMA provides you the possibility to declare some special species, such as **grids** or **graphs**, with their own built-in attributes and their own built-in actions. We will also see how to declare **mirror species**, which is a "copy" of a regular species, in order to give it an other representation. Finally, we will learn how to represent several agents through one unique agent, with **multi-level architecture**.

Chapter 45

Grid Species

A grid is a particular species of agents. Indeed, a grid is a set of agents that share a grid topology (until now, we only saw species with continuous topology). As other agents, a grid species can have attributes, attributes, behaviors, aspects However, contrary to regular species, grid agents are created automatically at the beginning of the simulation. It is thus not necessary to use the create statement to create them. Moreover, in addition to classic built-in variables, grid a provided with a set of additional built-in variables.

Index

Declaration

Instead of using the `species` keyword, use the keyword `grid` to declare a grid species. The grid species has exactly the same facets of the regular species, plus some others.

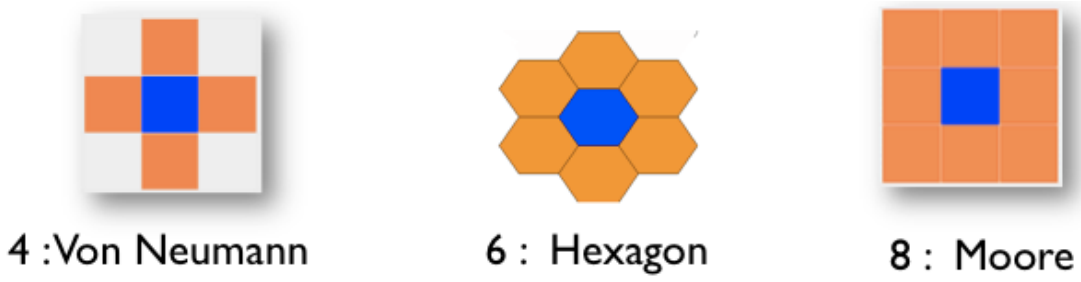


Figure 45.1: images/grid_neighbors.png

To declare a grid, you have to specify the number of columns and rows first. You can do it two different ways:

Using the two facets **width:** and **height:** to fix the number of cells (the size of each cells will be determined thanks to the environment dimension).

```
grid my_grid width:8 height:10 {
// my_grid has 8 columns and 10 rows
}
```

Using the two facets **cell_width:** and **cell_height:** to fix the size of each cells (the number cells will be determined thanks to the environment dimension).

```
grid my_grid cell_width:3 cell_height:2 {
// my_grid has cells with dimension 3m width by 2m height
}
```

By default, a grid is composed by 100 rows and 100 columns.

Another facet exists for grid only, very useful. It is the **neighbors** facet, used to determine how many neighbors has each cell. You can choose among 3 values: 4 (Von Neumann), 6 (hexagon) or 8 (Moore).

A grid can also be provided with specific facets that allows to optimize the computation time and the memory space, such as **use_regular_agents**, use_indivitual_shapes and **use_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;
    }
}
```

agents

return the set of agents located inside the cell. Note the use of this variable is deprecated. It is preferable to use the **inside** operator: //: # (keyword|operator_-inside)

```
grid cell width: 10 height: 10 neighbors: 4 {
    list<bug> bugs_inside -> {bug inside self};
}
```

color

The **color** built-in variable is used by the optimized grid display. Indeed, it is possible to use for grid agents an optimized aspect by using in a display the **grid** keyword. In this case, the grid will be displayed using the color defined by the **color** variable. The border of the cells can be displayed with a specific color by using the **lines** facet.

Here an example of the display of a grid species named **cell** with black border.

```
experiment main_xp type: gui{
    output {
        display map {
            grid cell lines: rgb("black") ;
        }
    }
}
```

neighbors

The **neighbors** built-in variable returns the list of cells at a distance of 1.

```
grid my_grid {
  reflex writeNeighbors {
    write neighbors;
  }
}
```

grid_value

The **grid_value** built-in variable is used when initializing a grid from grid file (see later). It is also used for the 3D representation of DEM.

Access to a cell

there are several ways to access to a specific cell:

- by a location: by casting a location to a cell (the unity (`#m`, `#cm`, etc...) is defined when you choose your environment size, in the global species.

```
   global {
        init {
            write "cell at {57.5, 45} :" + cell({57.5, 45});
        }
   }
```

```
    grid cell width: 10 height: 10 neighbors: 4 {
    }
```

- by the row and column indexes: like matrix, it is possible to directly access to a cell from its indexes

```
    global {
          init {
              write "cell [5,8] :" + cell[5,8];
          }
    }
    grid cell width: 10 height: 10 neighbors: 4 {
    }
```

The operator **grid_at** also exists to get a particular cell. You just have to specify the index of the cell you want (in x and y):

```
    global {
          init {
              agent cellAgent <- cell grid_at {5,8};
              write "cell [5,8] :" + cellAgent;
          }
    }
    grid cell width: 10 height: 10 neighbors: 4 {
    }
```

Display Grid

You can easily display your grid in your experiment as followed :

```
experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            grid MyGrid;
        }
    }
}
```

The grid will be displayed, using the color you defined for each cell (with the “color” built-in attribute). You can also show border of each cell by using the facet “lines:” and choosing a rgb color:

```
grid MyGrid lines:#black;
```

An other way to display a grid will be to define an aspect in your grid agent (the same way as for a regular species), and define your grid as a regular species then in your experiment, choosing your aspect :

```
grid MyGrid {
    aspect firstAspect {
        draw square(1);
    }
    aspect secondAspect {
        draw circle(1);
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: opengl {
            species MyGrid aspect:firstAspect;
        }
    }
}
```

Beware : don't use this second display when you have large grids : it's much slower.

Grid from a matrix

An easy way to load some values in a grid is to use matrix data. A **matrix** is a type of container (we invite you to learn some more about this useful type here). Once you have declared your matrix, you can set the values of your cells using the ask statement :

```
global {
  init {
    matrix data <- matrix([[0,1,1],[1,2,0]]);
    ask cell {
      grid_value <- float(data[grid_x, grid_y]);
    }
  }
}
```

Declaring larger matrix in GAML can be boring as you can imagine. You can load your matrix directly from a csv file with the operator **matrix** (used for the contruction of the matrix).

```
file my_file <- csv_file("path/file.csv","separator");
matrix my_matrix <- matrix(my_file);
```

You can try to read the following csv :

```
0,0,0,0,0,0,0,0,0,0,0
0,0,0,1,1,1,1,1,0,0,0
0,0,1,1,0,0,0,1,1,0,0
0,1,1,0,0,0,0,0,0,0,0
0,1,1,0,0,1,1,1,1,0,0
0,0,1,1,0,0,1,1,1,0,0
0,0,0,1,1,1,1,0,1,0,0
0,0,0,0,0,0,0,0,0,0,0
```

With the following model :

```
model import_csv

global {
  file my_csv_file <- csv_file("../includes/test.csv",",");
  init {
    matrix data <- matrix(my_csv_file);
    ask my_gama_grid {
      grid_value <- float(data[grid_x,grid_y]);
      write data[grid_x,grid_y];
    }
  }
}

grid my_gama_grid width: 11 height: 8 {
  reflex update_color {
    write grid_value;
    color <- (grid_value = 1) ? #blue : #white;
  }
}

experiment main type: gui{
  output {
    display display_grid {
      grid my_gama_grid;
    }
  }
```

```
}
```

For more complicated models, you can read some other files, such as ASCII files (asc), DEM files...

Example

To practice a bit those notions, we will build a quick model. A "regular" species will move randomly on the environment. A grid is displayed, and its cells becomes red when an instance of the regular species is waking inside this cell, and yellow when the regular agent is in the surrounding of this cell. If no regular agent is on the surrounding, the cell turns green.

Here is an example of implementation:

```
model my_grid_model

global{
    float max_range <- 5.0;
    int number_of_agents <- 5;
    init {
        create my_species number:number_of_agents;
    }
    reflex update {
        ask my_species {
            do wander amplitude:180;
            ask my_grid at_distance(max_range)
            {
                if(self overlaps myself)
                {
                    self.color_value <- 2;
                }
                else if (self.color_value != 2)
                {
                    self.color_value <- 1;
                }
            }
        }
        ask my_grid {
            do update_color;
        }
    }
}
```

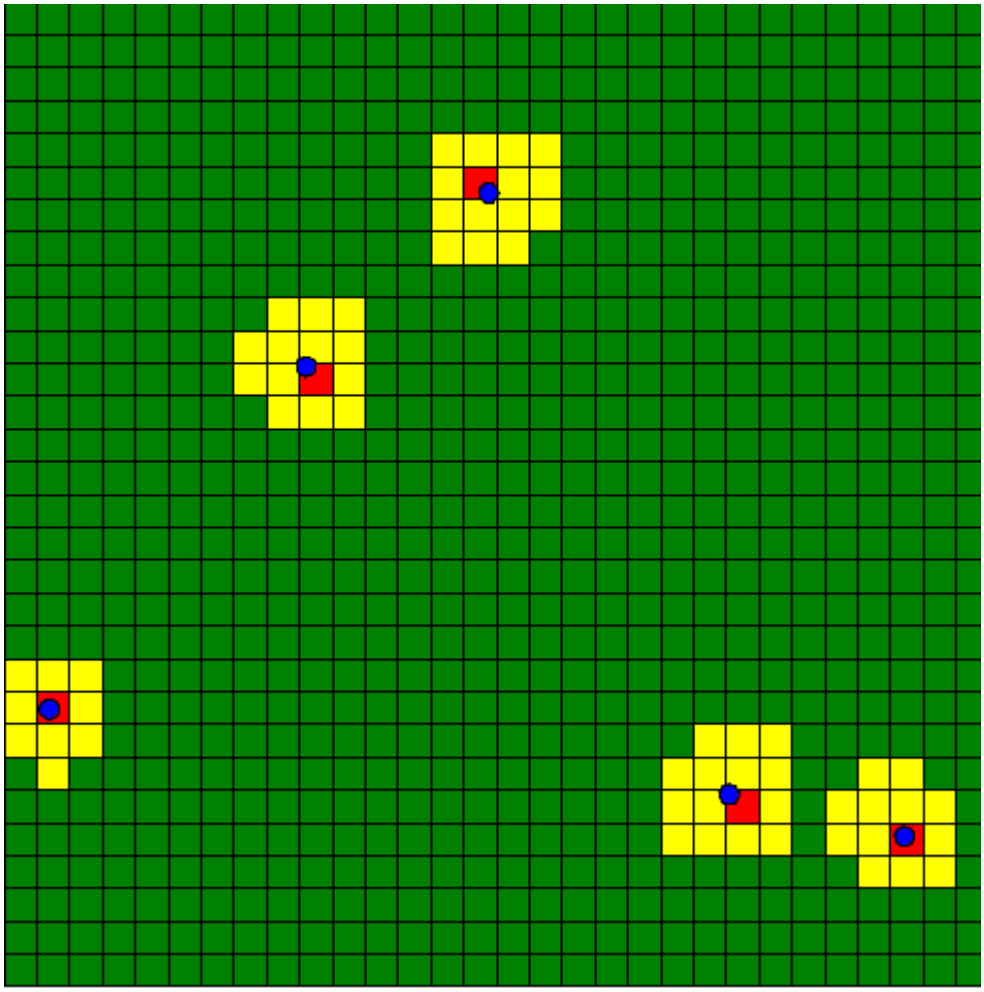


Figure 45.2: images/my_grid_model.png

```
species my_species skills:[moving] {
    float speed <- 2.0;
    aspect default {
        draw circle(1) color:#blue;
    }
}

grid my_grid width:30 height:30 {
    int color_value <- 0;
    action update_color {
        if (color_value = 0) {
            color <- #green;
        }
        else if (color_value = 1) {
            color <- #yellow;
        }
        else if (color_value = 2) {
            color <- #red;
        }
        color_value <- 0;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            grid my_grid lines:#black;
            species my_species aspect:default;
        }
    }
}
```

Chapter 46

Graph Species

Using a graph species enables to easily show interaction between agents of a same species. This kind of species is particularly useful when trying to show the interaction (especially the non-spatial one) that exist between agents.

Index

Declaration

Declare a graph with handmade agents

To instantiate this **graph** species, several steps must be followed. First the graph species must inherit from the abstract species `graph_node`, then the method `related_to` must be redefined and finally an auxiliary species that inherits from `base_edge` used to represent the edges of the generated graph must be declared. A graph node is an abstract species that must redefine one method called `related_to`.

```
species graph_agent parent: graph_node edge_species: edge_agent{
  bool related_to(graph_agent other){
    return true;
  }
}

species edge_agent parent: base_edge {
}
```

The method `related_to` returns a boolean, and take the agents from the current species in argument. If the method returns true, the two agents (the current instance and the one as argument) will be linked.

```
global{
    int number_of_agents <- 5;
    init {
        create graph_agent number:number_of_agents;
    }
}

species graph_agent parent: graph_node edge_species: edge_agent{
  bool related_to(graph_agent other){
    return true;
  }
  aspect base {
    draw circle(1) color:#green;
  }
}

species edge_agent parent: base_edge {
    aspect base {
    draw shape color:#blue;
  }
}
```

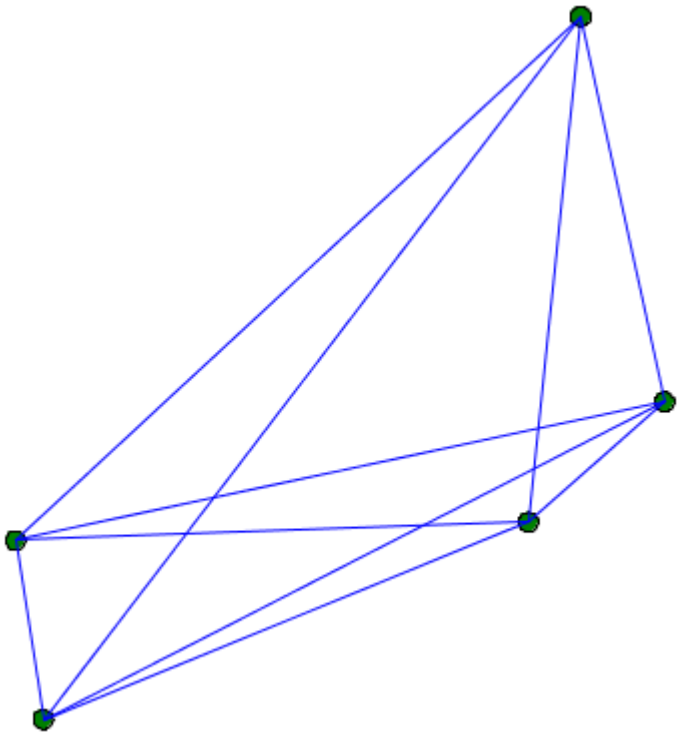


Figure 46.1: graph_related_to.png

```
experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species graph_agent aspect:base;
            species edge_agent aspect:base;
        }
    }
}
```

You can for example link 2 agents when they are closer than a certain distance. Beware: The topology used in graph species is the graph topology, and not the continuous topology. You can force the use of the continuous topology with the action `using` as follow:

```
bool related_to(graph_agent other){
    using topology:topology(world) {
        return (self.location distance_to other.location < 20);
```

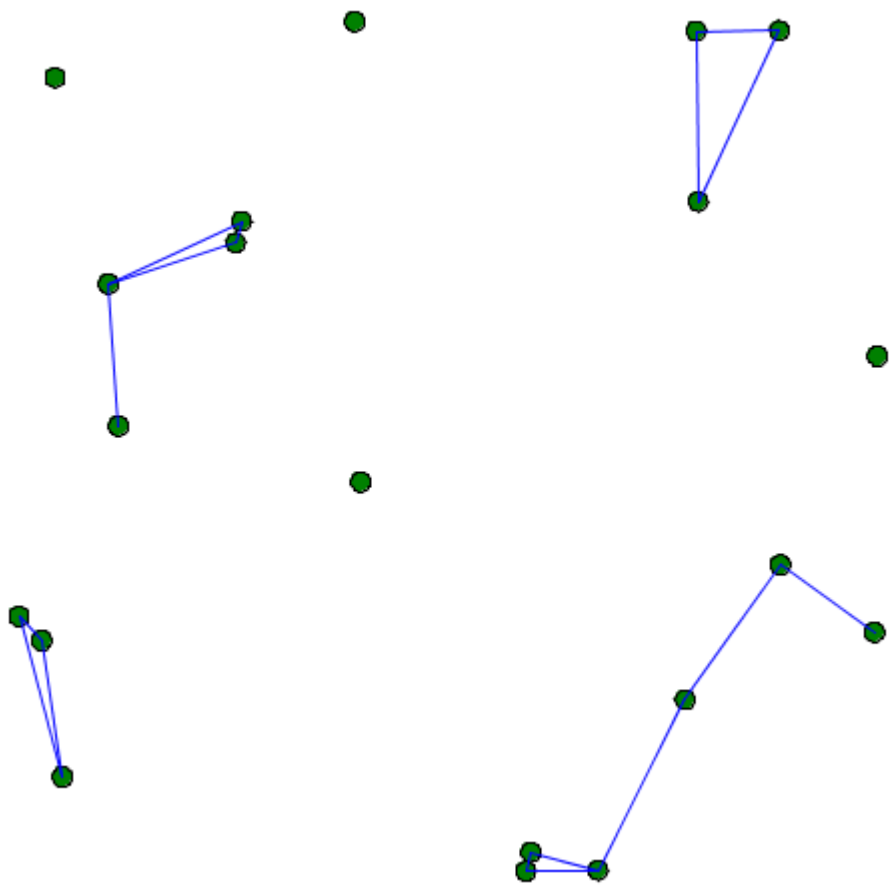


Figure 46.2: graph_related_to2.png

```
    }
}
```

The abstract mother species “graph_node” has an attribute “my_graph”, with the type “graph”. The graph type represent a graph composed of vertices linked with edges. This type has built-in attributes such as **edges** (the list of all the edges agents), or `vertices` (the list of all the vertices agents).

Declare a graph by using an geometry file

In most cases, you will have to construct a graph from an existing file (example: a "shp" file). In that case, you will have to first instantiate a species from the shape file (with the `create` statement, using the facet **`from`**). When, you will have to extract a graph from the agent, using the operator **`as_edge_graph`**.

```
model load_shape_file

global {
    file roads_shapefile <- file("../includes/road.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;

    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

experiment main_experiment type:gui{
    output {
        display map {
            species road aspect:geom;
        }
    }
}
```

Declare a graph with nodes and edges

Another way to create a graph is building it manually nodes by nodes, and then edges by edges, without using agent structures. Use the **`add_node`** operator and the **`add_edge`** operator to do so. Here is an example of how to do:

```
add point(0.0,0.0) to:nodes;
add point(90.0,90.0) to:nodes;
add point(20.0,20.0) to:nodes;
```

```
add point(40.0,50.0) to:nodes;
add point(100.0,0.0) to:nodes;

loop nod over:nodes {
    my_graph <- my_graph add_node(nod);
}

my_graph <- my_graph add_edge (nodes at 0::nodes at 2);
my_graph <- my_graph add_edge (nodes at 2::nodes at 3);
my_graph <- my_graph add_edge (nodes at 3::nodes at 1);
my_graph <- my_graph add_edge (nodes at 0::nodes at 4);
my_graph <- my_graph add_edge (nodes at 4::nodes at 1);
```

Using this solution, my_graph can have two types: it can be an a-spatial graph, or a spatial graph. The spatial graph will have a proper geometry, with segments that follow the position of your graph (you can access to the segments by using the built-in "segments"). The a-spatial graph will not have any shape.

```
global
{
    graph my_spatial_graph<-spatial_graph([]);
    graph my_aspatial_graph<-graph([]);

    init {
        point node1 <- {0.0,0.0};
        point node2 <- {10.0,10.0};
        my_spatial_graph <- my_spatial_graph add_node(node1);
        my_spatial_graph <- my_spatial_graph add_node(node2);
        my_spatial_graph <- my_spatial_graph add_edge(node1::node2);
        write my_spatial_graph.edges;
        // the output is [polyline ([{0.0,0.0,0.0},{10.0,10.0,0.0}])]
        my_aspatial_graph <- my_aspatial_graph add_node(node1);
        my_aspatial_graph <- my_aspatial_graph add_node(node2);
        my_aspatial_graph <- my_aspatial_graph add_edge(node1::node2);
        write my_aspatial_graph.edges;
        // the output is [{0.0,0.0,0.0}::{10.0,10.0,0.0}]
    }
}
```

Useful operators with graph

Knowing the degree of a node

The operator **degree_of** returns the number of edge attached to a node. To use it, you have to specify a graph (on the left side of the operator), and a node (on the right side of the operator).

The following code (to put inside the node species) displays the number of edges attached to each node:

```
aspect base
{
    draw text:string(my_graph degree_of node(5)) color:# black;
    status <- 0;
}
```

Get the neighbors of a node

To get the list of neighbors of a node, you should use the neighbors_of operator. On the left side of the operator, specify the graph you are using, and on the right side, specify the node. The operator returns the list of nodes located at a distance inferior or equal to 1, considering the graph topology.

```
species graph_agent parent: graph_node edge_species: edge_agent
{
  list<graph_agent> list_neighbors <- list<graph_agent>(my_graph
   neighbors_of (self));
}
```

Here is an example of model using those two previous concepts (a random node is chosen each step, displayed in red, and his neighbors are displayed in yellow):

```
model graph_model

global
{
    int number_of_agents <- 50;
    init
    {
        create graph_agent number: number_of_agents;
    }
```

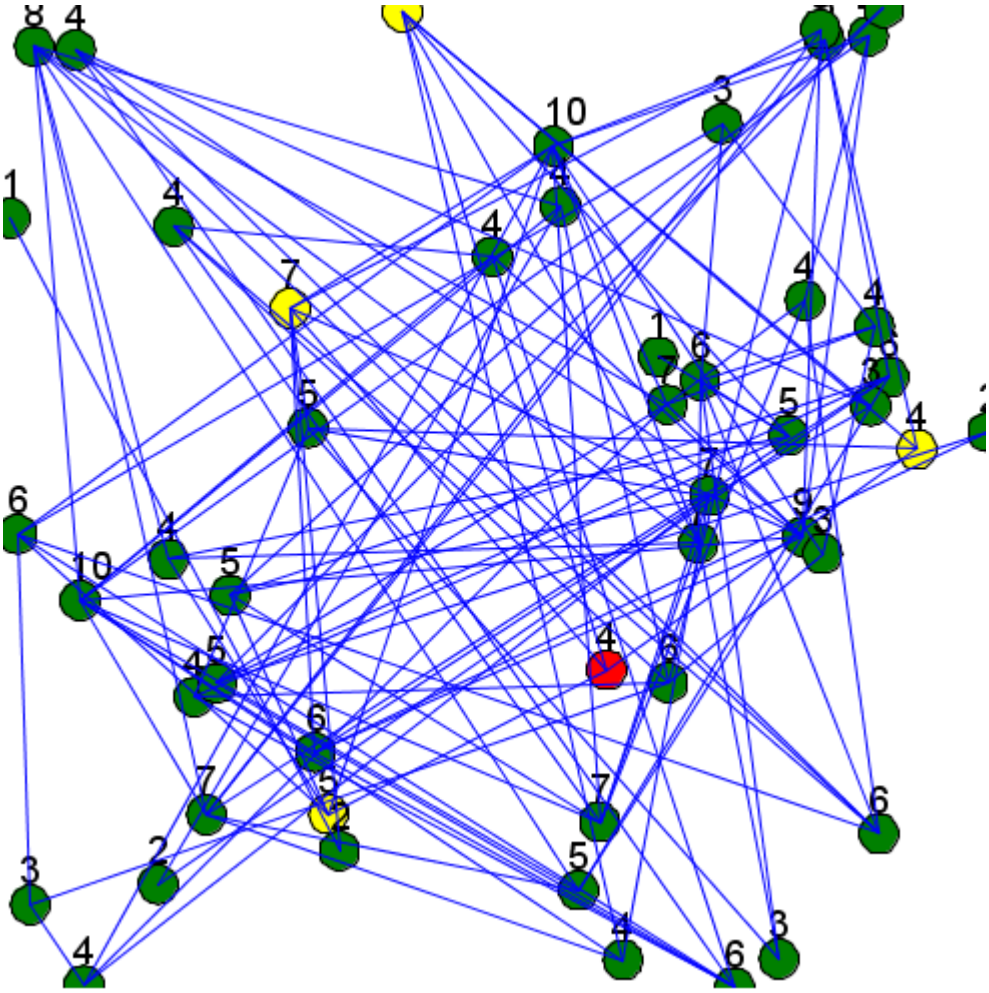


Figure 46.3: graph_model.png

```
    reflex update {
        ask graph_agent(one_of(graph_agent)) {
            status <- 2;
            do update_neighbors;
        }
    }
}

species graph_agent parent: graph_node edge_species: edge_agent
{
    int status <- 0;
    list<int> list_connected_index;

    init {
        int i<-0;
        loop g over:graph_agent {
            if (flip(0.1)) {
                add i to:list_connected_index;
            }
            i <- i+1;
        }
    }

    bool related_to(graph_agent other){
        if (list_connected_index contains (graph_agent index_of other))
    {
            return true;
        }
        return false;
    }

    action update_neighbors {

        list<graph_agent> list_neighbors <- list<graph_agent>(my_graph
   neighbors_of (self));

        loop neighb over:list_neighbors {
            neighb.status <- 1;
        }
    }

    aspect base
    {
        if (status = 0) {
            draw circle(2) color: # green;
```

```
        }
        else if (status = 1) {
            draw circle(2) color: # yellow;
        }
        else if (status = 2) {
            draw circle(2) color: # red;
        }
        draw text:string(my_graph degree_of self) color:# black size:4
    at:point(self.location.x-1,self.location.y-2);
        status <- 0;
    }
}

species edge_agent parent: base_edge
{
    aspect base
    {
        draw shape color: # blue;
    }
}

experiment MyExperiment type: gui
{
    output
    {
        display MyDisplay type: java2D
        {
            species graph_agent aspect: base;
            species edge_agent aspect: base;
        }
    }
}
```

Compute the shortest path

To compute the shortest path to go from a point to another, pick a source and a destination among the vertices you have for your graph. Store those values as point type.

```
point source;
point destination;
source <- point(one_of(my_graph.vertices));
destination <- point(one_of(my_graph.vertices));
```

Then, you can use the operator **path_between** to return the shortest path. To use this action, you have to give the graph, then the source point, and the destination point. This action returns a path type.

```
path shortest_path;
shortest_path <- path_between (my_graph, source,destination);
```

Another operator exists, **paths_between**, that returns a list of shortest paths between two points. Please read the documentation to learn more about this operator.

Here is an example of code that show the shortest path between two points of a graph:

```
model graph_model

global
{
    int number_of_agents <- 50;
    point source;
    point target;
    graph my_graph;
    path shortest_path;

    init
    {
        create graph_agent number: number_of_agents;
    }

    reflex pick_two_points {
        if (my_graph=nil) {
            ask graph_agent {
                myself.my_graph <- self.my_graph;
                break;
            }
        }
        shortest_path <- nil;
        loop while:shortest_path=nil {
            source <- point(one_of(my_graph.vertices));
            target <- point(one_of(my_graph.vertices));
            if (source != target) {
                shortest_path <- path_between (my_graph, source,target)
   ;
            }
        }
    }
}
```

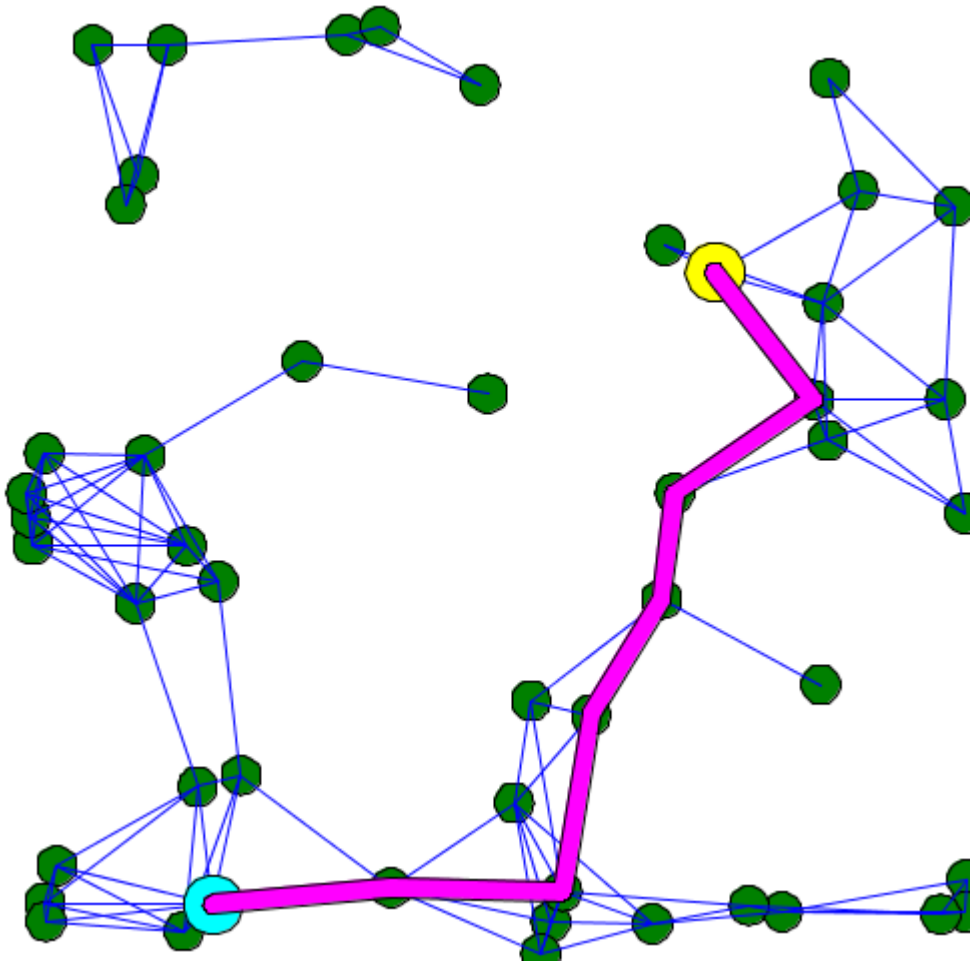


Figure 46.4: shortest_path.png

```
species graph_agent parent: graph_node edge_species: edge_agent
{
    list<int> list_connected_index;

    init {
        int i<-0;
        loop g over:graph_agent {
            if (flip(0.1)) {
                add i to:list_connected_index;
            }
            i <- i+1;
        }
    }

    bool related_to(graph_agent other) {
        using topology:topology(world) {
            return (self.location distance_to other.location < 20);
        }
    }

    aspect base {
        draw circle(2) color: # green;
    }
}

species edge_agent parent: base_edge
{
    aspect base {
        draw shape color: # blue;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species graph_agent aspect: base;
            species edge_agent aspect: base;
            graphics "shortest path" {
                if (shortest_path != nil) {
                    draw circle(3) at: source color: #yellow;
                    draw circle(3) at: target color: #cyan;
                    draw (shortest_path.shape+1) color: #magenta;
                }
            }
        }
```

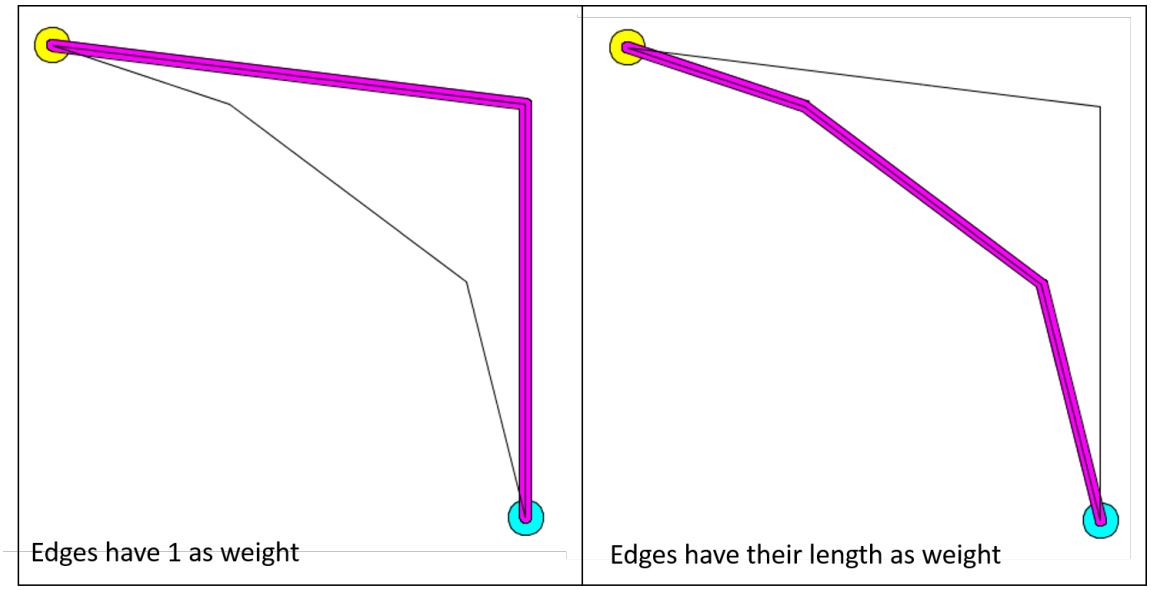


Figure 46.5: path_weight.png

```
    }
}
```

Control the weight in graph

You can add a map of weight for the edges that compose the graph. Use the operator **with_weights** to put weights in your graph. The graph has to be on the left side of the operator, and the map has to be on the right side. In the map, you have to put edges as key, and the weight for that edge as value. One common use is to put the distance as weight:

```
my_graph <- my_graph with_weights (my_graph.edges as_map (each::
    geometry(each).perimeter));
```

The calculation of shortest path can change according to the weight you choose for your edges. For example, here is the result of the calculation of the shortest path when all the edges have 1 as weight value (it is the default graph topology), and when the edges have their length as weight.

Here is an example of implementation:

```
model shortest_path_with_weight

global
{
    graph my_graph<-spatial_graph([]);
    path shortest_path;
    list<point> nodes;

    init
    {
        add point(10.0,10.0) to:nodes;
        add point(90.0,90.0) to:nodes;
        add point(40.0,20.0) to:nodes;
        add point(80.0,50.0) to:nodes;
        add point(90.0,20.0) to:nodes;

        loop nod over:nodes {
            my_graph <- my_graph add_node(nod);
        }

        my_graph <- my_graph add_edge (nodes at 0::nodes at 2);
        my_graph <- my_graph add_edge (nodes at 2::nodes at 3);
        my_graph <- my_graph add_edge (nodes at 3::nodes at 1);
        my_graph <- my_graph add_edge (nodes at 0::nodes at 4);
        my_graph <- my_graph add_edge (nodes at 4::nodes at 1);

        // comment/decomment the following line to see the difference.
        my_graph <- my_graph with_weights (my_graph.edges as_map (each
   ::geometry(each).perimeter));

        shortest_path <- path_between(my_graph,nodes at 0, nodes at 1);
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            graphics "shortest path" {
                if (shortest_path != nil) {
                    draw circle(3) at: point(shortest_path.source)
   color: #yellow;
                    draw circle(3) at: point(shortest_path.target)
   color: #cyan;
                    draw (shortest_path.shape+1) color: #magenta;
                }
```

```
                loop edges over: my_graph.edges {
                    draw geometry(edges) color: #black;
                }
            }
        }
    }
}
```

Chapter 47

Mirror species

A mirror species is a species whose population is automatically managed with respect to another species. Whenever an agent is created or destroyed from the other species, an instance of the mirror species is created or destroyed. Each of these 'mirror agents' has access to its reference agent (called its target). Mirror species can be used in different situations but the one we describe here is more oriented towards visualization purposes.

Index

Declaration

A mirror species can be defined using the **mirrors** keyword as following:

```
species B mirrors: A{
}
```

In this case, the species B mirrors the species A.

By default, the location of the species B will be random but in many cases, once want to place the mirror agent at the same location of the reference species. This can be 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 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{
    point location <- target.location update: {target.location.x,target
    .location.y,target.location.z+5};
```

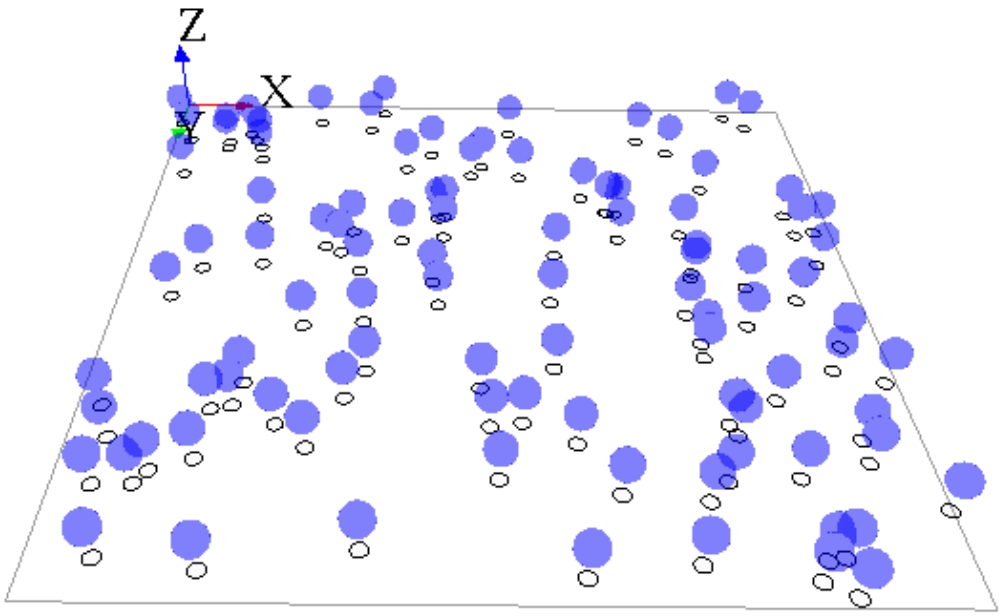


Figure 47.1: images/mirror_model.png

```
    aspect base {
        draw sphere(2) color: #blue;
    }
}

experiment mirroExp type: gui {
    output {
        display superposedView type: opengl{
          species A aspect: base;
          species B aspect: base transparency:0.5;
        }
    }
}
```

Chapter 48

Multi-level architecture

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

Index

Declaration of micro-species

A species can have other species as micro-species. The micro-species of a species is declared inside the species' declaration.

```
species macro_species {
      species micro_species_in_group {
      }
}
```

In the above example, “micro_species_in_group” is a micro-species of “macro_-species”. An agent of “macro_species” can have agents “micro_species_in_group” as micro-agents. Agents of “micro_species_in_group” have an agent of “macro_species” as “host” agent.

As the species “micro_species_in_group” is declared inside the species “macro_-species”, “micro_species_in_group” will return a list of “micro_species_in_group” agent inside the given “macro_species” agent.

```
global
{
    init {
        create macro_species number:5;
    }
}

species macro_species
{
    init {
        create micro_species_in_group number:rnd(10);
        write "the macro species named "+name+" contains "+length(
   micro_species_in_group)+" micro-species.";
    }

    species micro_species_in_group {
    }
}

experiment my_experiment type: gui {
}
```

In this above example, we create 5 macro-species, and each one of these macro-species create a random number of inner micro-species. We can see that “micro_species_-in_group” refers to the list of micro-species inside the given macro-species.

Access to micro-agents, host agent

To access to micro-agents (from a macro-agent), and to host agent (from a micro-agents), you have to use two built-in attributes.

The `members` built-in attribute is used inside the macro-agent, to get the list of all its micro-agents.

```
species macro_species
{
    init {
        create first_micro_species number:3;
        create second_micro_species number:6;
        write "the macro species named "+name+" contains "+length(
    members)+" micro-species.";
    }

    species first_micro_species {
    }

    species second_micro_species {
    }
}
```

The `host` built-in attribute is used inside the micro-agent to get the host macro-agent.

```
species macro_species {

    micro_species_in_group micro_agent;

    init {
        create micro_species_in_group number:rnd(10);
        write "the macro species named "+name+" contains "+length(
    members)+" micro-species.";
    }

    species micro_species_in_group {
        init {
            write "the micro species named "+name+" is hosted by "+host
    ;
        }
    }
}
```

NB: We already said that the world agent is a particular agent, instantiated just once. In fact, the world agent is the host of all the agents. You can try to get the host for a regular species, you will get the world agent itself (named as you named your model). You can also try to get the members of your world (from the global scope for example), and you will get the list of the agents presents in the world.

```
global
{
    init {
```

```
        create macro_species number:5;
        write "the world has "+length(members)+" members.";
    }
}

species macro_species
{
    init {
        write "the macro species named "+name+" is hosted by "+host;
    }
}
```

Representation of an entity as different types of agent

The multi-level architecture is often used in order to represent an entity through different types of agent. For example, an agent “bee” can have a behavior when it is alone, but when the agent is near from a lot of agents, he can changes his type to “bee_in_swarm”, defined as a micro-species of a macro-species “swarm”. Other example: an agent “pedestrian” can have a certain behavior when walking on the street, and then change his type to “pedestrian_in_building” when he is in a macro-species “building”. You have then to distinguish two different species to define your micro-species: - The first can be seen as a regular species (it is the “bee” or the “pedestrian” for instance). We will name this species as “micro_species”. - The second is the real micro-species, defined inside the macro-species (it is the “bee_in_swarm” or the “pedestrian_in_building” for instance). We will name this species as “micro_species_in_group”. This species has to inherit from the “micro_species”.

```
species micro_species {
}

species macro_species
{
    species micro_species_in_group parent: micro_species {
    }
}
```

Dynamic migration of agents

In our example about bees, a "swarm" entity is composed of nearby flying "bee" entities. When a "bee" entity approaches a "swarm" entity, this "bee" entity will become a member of the group. To represent this, the modeler lets the "bee" agent change its species to "bee_in_swarm" species. The "bee" agent hence becomes a "bee_in_swarm" agent. To change species of agent, we can use one of the following statements: `capture`, `release`, `migrate`.

The statement `capture` is used by the "macro_species" to capture one (or several) "micro_species" agent(s), and turn it (them) to a "micro_species_in_group". You can specify which agent (or list of agents) you want to capture by using the facet **`target`**. The facet **`as`** is used to cast the agent from "micro_species" to "micro_species_in_group". You can use the facet `return` to get the newly captured agent(s).

```
capture target:micro_species as:micro_species_in_group;
```

The statement `release` is used by the "macro_species" to release one (or several) "micro_species_in_group" agent(s), and turn it (them) to a "micro_species". You can specify which agent (or list of agents) you want to release by using the facet **`target`**. The facet **`as`** is used to cast the agent from "micro_species_in_group" to "micro_species". The facet **`in`** is used to specify the new host (by default, it is the host of the "macro_species"). You can use the facet `return` to get the newly released agent(s).

```
release target:list(micro_species_in_group) as:micro_species in:world;
```

The statement `migrate`, less used, permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Read the GAML Reference to learn more about this statement.

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;
```

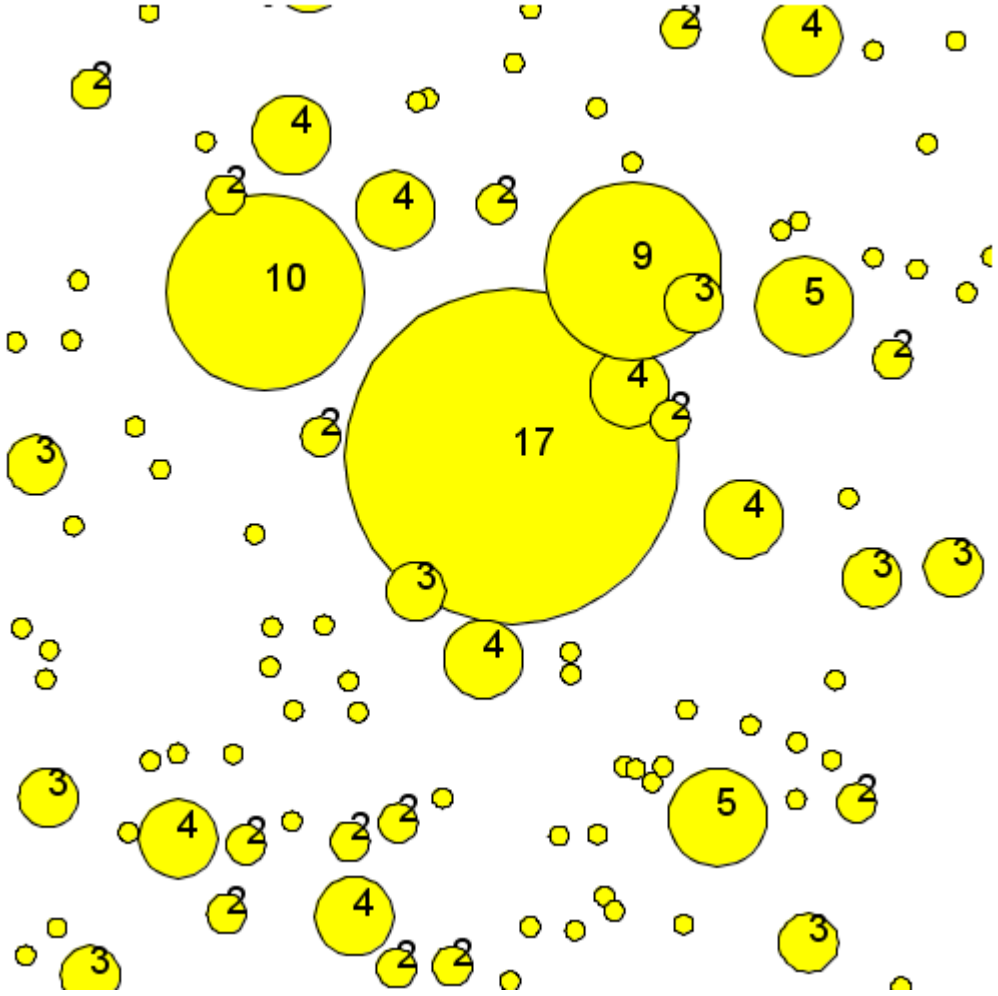


Figure 48.1: images/multilevel_model.png

```
    int capture_time <- 100;
    int remaining_release_time <- 0;
    int remaining_capture_time <- capture_time;
    init {
        create micro_species number:200;
    }
    reflex reflex_timer {
        if (remaining_release_time=1)
        {
            remaining_release_time <- 0;
            remaining_capture_time <- capture_time;
        }
        else if (remaining_capture_time=1)
        {
            remaining_capture_time <- 0;
            remaining_release_time <- release_time;
        }
        remaining_release_time <- remaining_release_time - 1;
        remaining_capture_time <- remaining_capture_time - 1;
    }
    reflex capture_micro_species when:(remaining_capture_time>0 and
   flip(0.1)) {
        ask macro_species {
            list<micro_species> micro_species_in_range <- micro_species
    at_distance 1;
            if (micro_species_in_range != []) {
                do capture_micro_species(micro_species_in_range);
            }
        }
        ask micro_species {
            list<micro_species> micro_species_list_to_be_captured <-
   micro_species at_distance 1;
            if(micro_species_list_to_be_captured != []) {
                create macro_species {
                    location <- myself.location;
                    add item:myself to:
   micro_species_list_to_be_captured;
                    do capture_micro_species(
   micro_species_list_to_be_captured);
                }
            }
        }
    }
}

species micro_species skills:[moving] {
```

```
    geometry shape <- circle(1);
    aspect base {
        draw shape;
    }
    reflex move{
        do wander;
    }
}

species macro_species {
    geometry shape <- circle(1) update:circle(length(members));

    species micro_species_in_group parent:micro_species {
    }

    action capture_micro_species(list<micro_species> micro_list) {
        loop mic_sp over:micro_list {
            capture mic_sp as:micro_species_in_group;
        }
    }

    reflex release_reflex when:(remaining_release_time>0 and flip(0.1))
    {
        release members as:micro_species /*in:world*/;
        do die;
    }

    aspect base {
        draw shape;
        draw text:string(length(members)) color:#black size:4;
    }
}

experiment MyExperiment type: gui {
    output {
        display MyDisplay type: java2D {
            species macro_species aspect: base;
            species micro_species aspect: base;
        }
    }
}
```

Chapter 49

Defining GUI Experiment

When you execute your simulation, you will often need to display some information. For each simulation, you can define some inputs and outputs:

- The inputs will be composed of parameters manipulated by the user for each simulation.
- The outputs will be composed of displays, monitors or output files. They will be define inside the scope `output`.

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

You can define two types of experiment (through the facet **`type`**):

- **`gui`** experiments (the default type) are used to play an experiment, and interpret its outputs.
- **`batch`** experiments are used to play an experiment several times (usually with other input values), used for model exploration. We will come back to this notion a bit further in the tutorial.

Inside experiment scope, you can access to some built-ins which can be useful, such as `minimum_cycle_duration`, to force the duration of one cycle.

```
experiment my_experiment type: gui {
    float minimum_cycle_duration <- 2.0#minute;
}
```

Other built-ins are available, to learn more about, go to the page **experiment built-in**.

In this part, we will focus on the **gui experiments**. We will start with learning how to **define input parameters**, then we will study the outputs, such as **displays**, **monitors and inspectors**, and **export files**. We will finish this part with how to define **user commands**.

Chapter 50

Defining Parameters

When playing simulation, you have the possibility to define input parameters, in order to change them and replay the simulation. Defining parameters allows to make the value of a global variable definable by the user through the user graphic interface.

Index

Defining parameters

You can define parameters inside the global scope, when defining your global variables with the facet `parameter`:

```
global
{
    int my_integer_global_value <- 5 parameter: "My integer global
   value";
}
```

When launching your experiment, the parameter will appear in your "Parameters" panel, with the name you chose for the `parameter` facet.

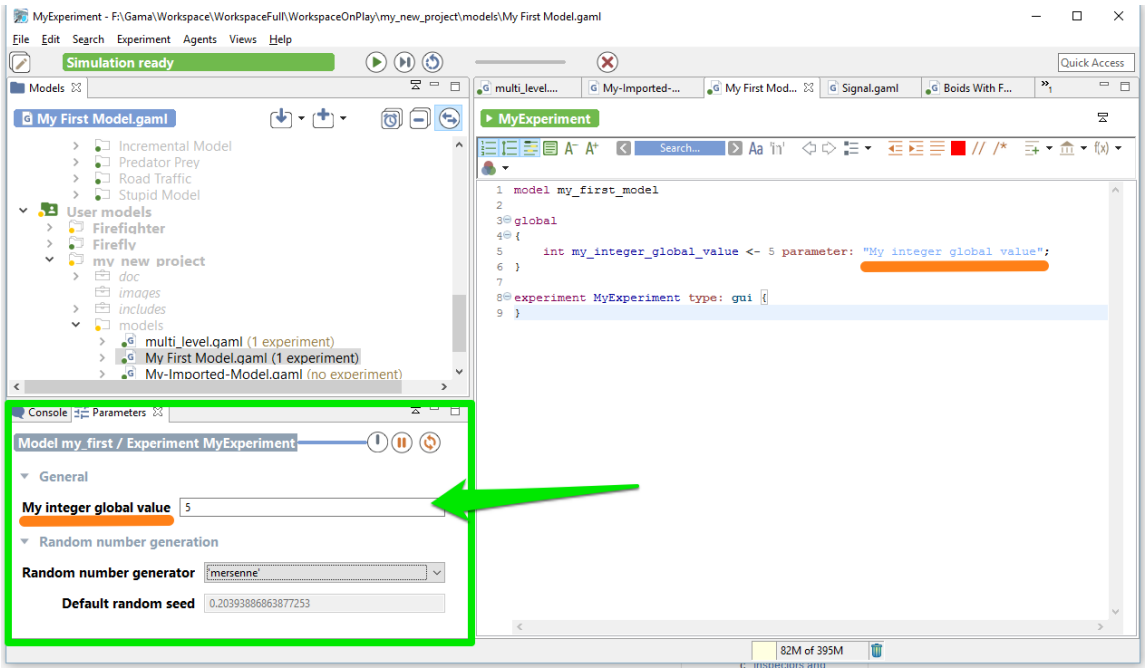


Figure 50.1: images/parameter1.png

You can also define your parameter inside the experiment, using the statement parameter. You have to specify first the name of your parameter, then the name of the global variable through the facet **var**.

```
global
{
    int my_integer_global_value <- 5;
}
```

experiment MyExperiment type: gui { parameter “My integer global value” var:my_integer_global_value; }

NB: This variable has to be initialized with a value. If you don’t want to initialize your value on the `global` scope, you can initialize the value directly on the parameter statement, using the facet **init**.

```
global
{
    int my_integer_global_value;
}
```

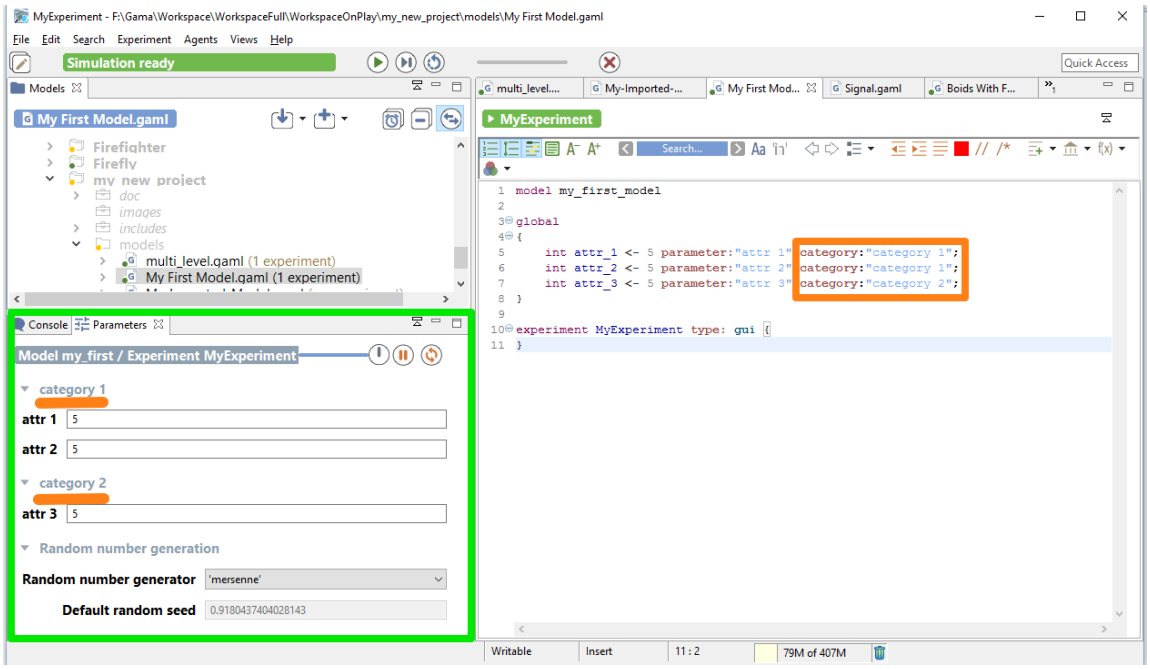


Figure 50.2: images/parameter2.png

```
experiment MyExperiment type: gui {
    parameter "My integer global value" var:my_integer_global_value
    init:5;
}
```

Additional facets

You can use some facets to arrange your parameters. For example, you can categorize your parameters under a label, using the facet **category**:

```
global
{
    int attr_1 <- 5 parameter:"attr 1" category:"category 1";
    int attr_2 <- 5 parameter:"attr 2" category:"category 1";
    int attr_3 <- 5 parameter:"attr 3" category:"category 2";
}
```

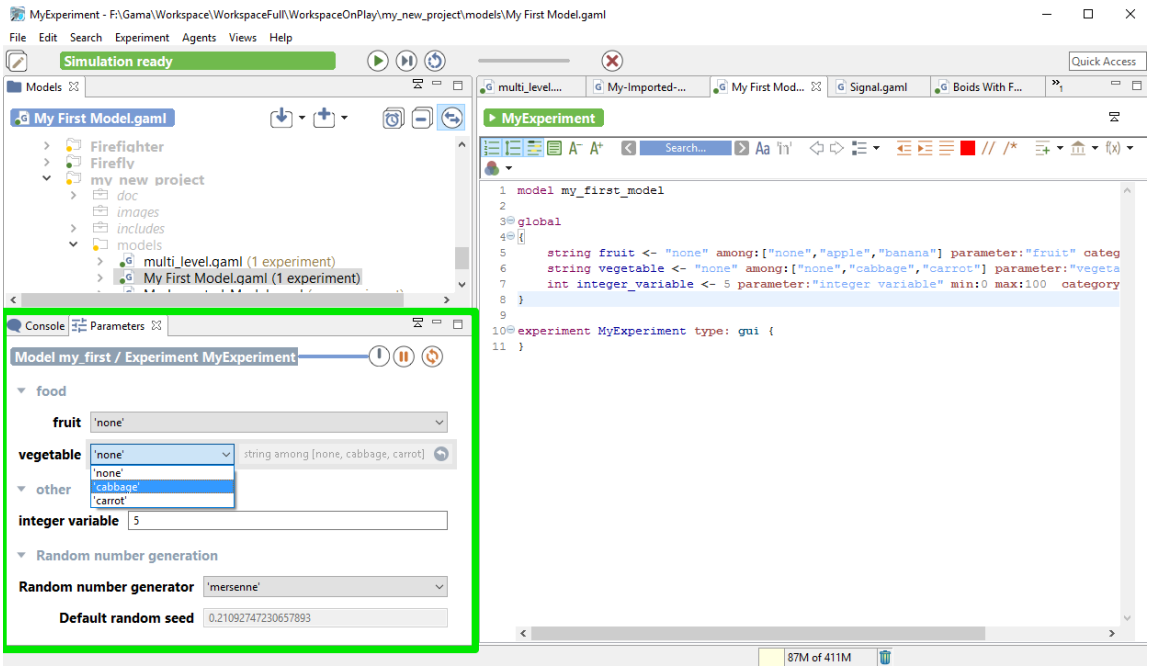


Figure 50.3: images/parameter3.png

You also can add some facets such as **min**, **max** or **among** to improve the declaration of the parameter.

```
global
{
    string fruit <- "none" among:["none","apple","banana"] parameter:"
    fruit" category:"food";
    string vegetable <- "none" among:["none","cabbage","carrot"]
    parameter:"vegetable" category:"food";
    int integer_variable <- 5 parameter:"integer variable" min:0 max
    :100  category:"other";
}

experiment MyExperiment type: gui {
}
```

Chapter 51

Defining displays (Generalities)

Index

Displays and layers

A display is the graphical output of your simulation. You can define several displays related with what you want to represent from your model execution. To define a display, use the keyword `display` inside the `output` scope, and specify a name (**`name`** facet).

```
experiment my_experiment type: gui {
    output {
        display "display1" {
        }
        display name:"display2" {
```

```
        }
    }
}
```

Other facets are available when defining your display:

- Use **background** to define a color for your background

```
display "my_display" background:#red
```

- Use **refresh** if you want to refresh the display when a condition is true (to refresh your display every number of steps, use the operator **every**)

```
display "my_display" refresh:every(10)
```

You can choose between two types of displays, by using the facet type:

- java2D displays will be used when you want to have 2D visualization. It is used for example when you manipulate charts. This is the default value for the facet type.
- opengl displays allows you to have 3D visualization.

You can save the display on the disk, as a png file, in the folder name_of_model/-models/snapshots, by using the facet **autosave**. This facet takes one a boolean as argument (to allow or not to save each frame) or a point (to define the size of your image). By default, the resolution of the output image is 500x500px (note that when no unit is provided, the unit is `#px` (pixel)).

```
display my_display autosave:true type:java2D {}
```

is equivalent to :

```
display my_display autosave:{500,500} type:java2D {}
```

Each display can be decomposed in one or several layers. Here is a screenshot (from the Toy Model Ant) to better understand those different notions we are about to tackle in this session.

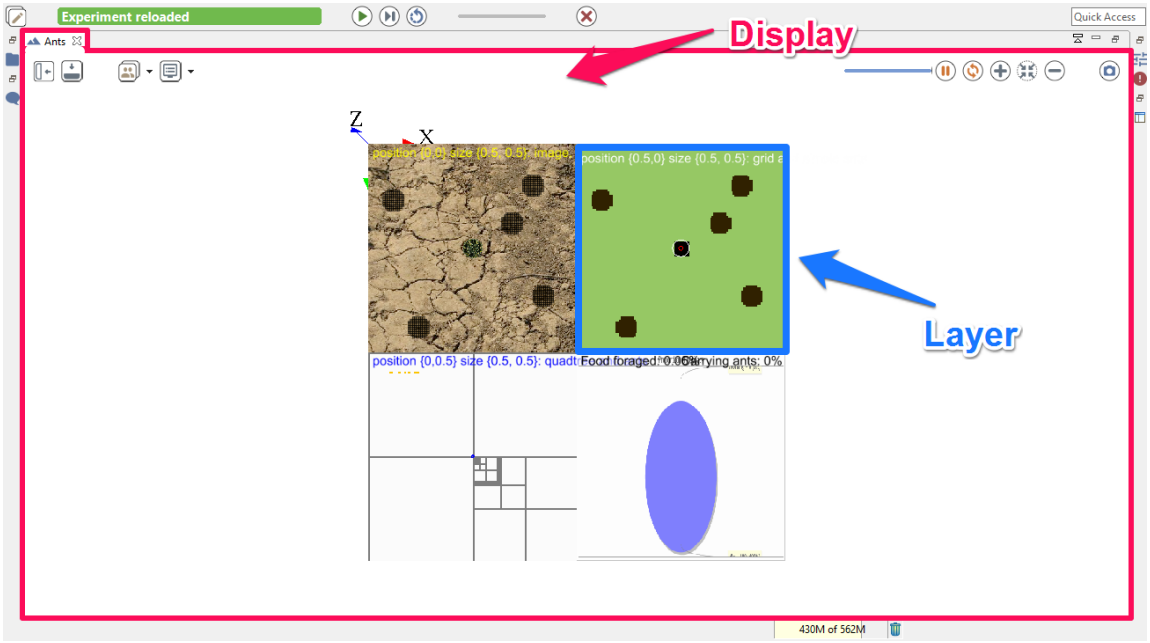


Figure 51.1: images/difference_layer_display.png

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.

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;
```

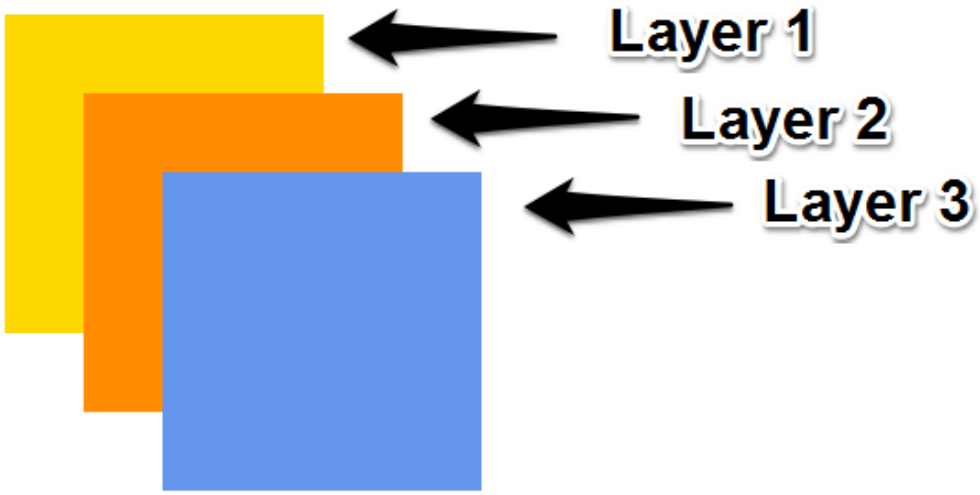


Figure 51.2: images/layers_order.png

```
            }
            graphics "layer3" {
                draw square(20) at:{20,20} color:#cornflowerblue;
            }
        }
    }
}
```

Will have this output:

Most of the layers have the **transparency** facet in order to see the layers which are under.

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" {
                draw square(20) at:{10,10} color:#darkorange;
            }
            graphics "layer2" transparency:0.5 {
                draw square(20) at:{15,15} color:#cornflowerblue;
            }
        }
    }
```

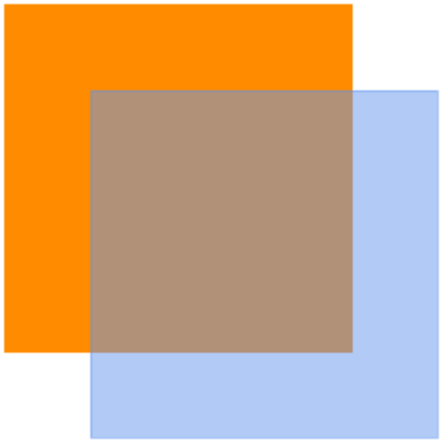


Figure 51.3: images/layers_transparency.png

```
}
```

To specify a position and a size for your layer, you can use the **position** and the **size** facets. The **position** facet is used with a point type, between {0,0} and {1,1}, which corresponds to the position of the upper left corner of your layer in percentage. Then, if you choose the point {0.5,0.5}, the upper left corner of your layer will be in the center of your display. By default, this value is {0,0}. The **size** facet is used with a point type, between {0,0} and {1,1} also. It corresponds to the size occupied by the layer in percentage. By default, this value is {1,1}.

```
experiment expe type:gui {
    output {
        display my_display {
            graphics "layer1" position:{0,0} size:{0.5,0.8} {
                draw shape color:#darkorange;
            }
            graphics "layer2" position:{0.3,0.1} size:{0.6,0.2} {
                draw shape color:#cornflowerblue;
            }
            graphics "layer3" position:{0.4,0.2} size:{0.3,0.8} {
                draw shape color:#gold;
            }
        }
    }
```

Figure 51.4: images/layers_size_position.png

```
}
```

NB: `displays` can have background, while `graphics` can't. If you want to put a background for your `graphics`, a solution can be to draw the `shape` of the world (which is, by default, a square 100m*100m).

A lot of other facets are available for the different layers. Please read the documentation of `graphics` for more information.

Example of layers

agents layer

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

Please read the documentation about `agents` statement if you are interested.

species layer

`species` allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

Please read the documentation about `species` statement if you are interested.

image layer

`image` allows modeler to display an image (e.g. as background of a simulation).

Please read the documentation about `image` statement if you are interested.

graphics layer

`graphics` allows the modeler to freely draw shapes/geometries/texts without having to define a species.

Please read the documentation about `graphics` statement if you are interested.

Chapter 52

Defining Charts

To visualize result and make analysis about you model, you will certainly have to use charts. You can define 3 types of charts in GAML: histograms, pie, and series. For each type, you will have to determine the data you want to highlight.

Index

Define a chart

To define a chart, we have to use the `chart` statement. A chart has to be named (with the **`name`** facet), and the type has to be specified (with the **`type`** facet). The value of the **`type`** facet can be `histogram`, `pie`, `series`, `scatter`, `xy`. A chart has to be defined inside a display.

```
experiment my_experiment type: gui {
    output {
        display "my_display" {
            chart "my_chart" type:pie {
            }
        }
```

```
    }
}
```

After declaring your chart, you have to define the data you want to display in your chart.

Data definition

Data can be specified with:

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

The `data` statement is used to specify which variable will be displayed. You have to give your data a name (that will be displayed in your chart), the value of the variable you want to follow (using the **`value`** facet). You can add come optional facets such as **`color`** to specify the color of your data.

```
global
{
    int numberA <- 2 update:numberA*2;
    int numberB <- 10000 update:numberB-1000;
}

experiment my_experiment type: gui {
    output {
        display "my_display" {
            chart "my_chart" type:pie {
                data "numberA" value:numberA color:#red;
                data "numberB" value:numberB color:#blue;
            }
        }
    }
}
```

(TODO_IMAGE)

The `datalist` statement is used several variables in one statement. Instead of giving simple values, datalist is used with lists.

```
datalist ["numberA","numberB"] value:[numberA,numberB] color:[#red,#
    blue];
```

[TODO] Datalist provides you some additional facets you can use. If you want to learn more about them, please read the documentation [URL]

Different types of chart

As we already said, you can display 3 types of graphs: the histograms, the pies and the series.

The histograms:

[TODO]

Chapter 53

Defining 3D Displays

Table of contents

OpenGL display

- Define the attribute type of the display with **`type`**`:opengl` in the output of your model (or use the preferences->display windows to use it by default):

```
output {
  display DisplayName type:opengl {
    species mySpecies;
  }
```

The opengl display share most of the feature that the java2D offers and that are described here.

Using 3D display offers many way to represent a simulation. A layer can be positioned and scaled in a 3D world. It is possible to superpose layer on different z value and display different information on the model at different position on the screen.

Position

Layer can be drawn on different position (x,y and z) value using the *position* facet

Size

Layer can be drawn with different size (x,y and z) using the *size* facet

Here is an example of display using all the previous facet (experiment factice to add to the model *Incremental Model 5*). You can also dynamically change those value by showing the side bar in the display.

```
experiment expe_test type:gui {
    output {
        display city_display type: opengl{
            species road aspect: geom refresh:false;
            species building aspect: geom transparency:0.5 ;
            species people aspect: sphere3D position:{0,0,0.1};
            species road aspect: geom size:{0.3,0.3,0.3};
        }
    }
}
```

Camera

Arcball Camera

FreeFly Camera

Figure 53.1: images/species_layer.png

Dynamic camera

User have the possibility to set dynamically the parameter of the camera (observer). The basic camera properties are its **position**, the **direction** in which is pointing, and its **orientation**. Those 3 parameters can be set dynamically at each iteration of the simulation.

Camera position

The facet `camera_pos(x,y,z)` places the camera at the given position. The default camera positon is *(world.width/2,world/height/2,world.maxDim*1.5)* to place the camera at the middle of the environement at an altitude that enables to see the entire environment.

Camera direction (Look Position)

The facet `camera_look_pos(x,y,z)` points the camera toward the given position. The default look position is *(world.width/2,world/height/2,0)* to look at the center of the environment.

Camera orientation (Up Vector)

The camera `camera_up_vector(x,y,z)` sets the *up vector* of the camera. The *up vector* direction in your scene is the *up* direction on your display screen. The default value is (0,1,0)

Here are some examples that can be done using those 3 parameters. You can test it by running the following model:

Default view

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

First person view

You can set the position as a first person shooter video game using:

```
display FirstPerson  type:opengl
camera_pos:{boids(1).location.x,-boids(1).location.y,10}
camera_look_pos:{cos(boids(1).heading)*world.shape.width,-sin(boids(1).
    heading)*world.shape.height,0}
camera_up_vector:{0.0,0.0,1.0}{
...
}
```

Third Person view

You can follow an agent during a simulation by positioning the camera above it using:

```
display ThirdPerson  type:opengl camera_pos:{boids(1).location.x,-boids
    (1).location.y,250}  camera_look_pos:{boids(1).location.x,-boids(1).
    location.y,boids(1).location.z}{
...
}
```

Lighting

In a 3D scene once can define light sources. The way how light sources and 3D object interact is called lighting. Lighting is an important factor to render realistic scenes.

In a real world, the color that we see depend on the interaction between color material surfaces, the light sources and the position of the viewer. There are four kinds of lighting called *ambient*, *diffuse*, *specular* and *emissive*.

Gama handle *ambient* and *diffuse* light.

- **ambient_light**: Allows to define the value of the ambient light either using an int (ambient_light:(125)) or a rgb color ((ambient_light:rgb(255,255,255)). default is rgb(125,125,125).
- **diffuse_light**: Allows to define the value of the diffuse light either using an int (diffuse_light:(125)) or a rgb color ((diffuse_light:rgb(255,255,255)). default is rgb(125,125,125).

- **diffuse_light_pos**: Allows to define the position of the diffuse light either using an point (diffuse_light_pos:{x,y,z}). default is {world.shape.width/2,world.shape.height/2,world.shape.width*2}.
- **is_light_on**: Allows to enable/disable the light. Default is true.
- **draw_diffuse_light**: Allows to enable/disable the drawing of the diffuse light. Default is false")),

Here is an example using all the available facet to define a diffuse light that rotate around the world.

```
display View1  type:opengl draw_diffuse_light:true ambient_light:(0)
   diffuse_light:(255) diffuse_light_pos:{50+ 150*sin(time*2),50,150*
   cos(time*2){
...
}
```

Chapter 54

Defining monitors and inspectors

Other outputs can be very useful to study better the behavior of your agents.

Index

Define a monitor

A monitor allows to follow the value of an arbitrary expression in GAML. It will appear, in the User Interface, in a small window on its own and be recomputed every time step (or according to its refresh facet).

Definition of a monitor:

```
monitor monitor_name value: an_expression refresh:boolean_statement;
```

with:

- **value**: mandatory, the expression whose value will be displayed by the monitor.
- **refresh**: bool statement, optional : the new value is computed if the bool statement returns true.

Example:

```
experiment my_experiment type: gui {
    output {
        monitor monitor_name value: cycle refresh:every(1);
    }
}
```

NB : you can also declare monitors during the simulation, by clicking on the button "Add new monitor", and specifying the name of the variable you want to follow.

Define an inspector

During the simulation, the user interface of GAMA provides the user the possibility to inspect an agent, or a group of agents. But you can also define the inspector you want directly from your model, as an output of the experiment.

Use the statement `inspect` to define your inspector, in the output scope of your gui experiment. The inspector has to be named (using the facet **`name`**), a value has to be specified (with the **`value`** facet).

```
inspect name:"inspector_name" value:the_value_you_want_to_display;
```

Note that you can inspect any type of species (regular species, grid species, even the world...).

The optional facet **`type`** is used to specify the type of your inspector. 2 values are possible:

- *agent* (default value) if you want to display the information as a regular agent inspector. Note that if you want to inspect a large number of agents, this can take a lot of time. In this case, prefer the other type *table*
- *table* if you want to display the information as an agent browser

The optional facet `attribute` is used to filter the attributes you want to be displayed in your inspector.

Beware: only one agent inspector (**`type:agent`**) can be used for an experiment. Beside, you can add as many agent browser (**`type:`**`table`) as you want for your experiment.

Example of implementation:

```
model new

global {
    init {
        create my_species number:3;
    }
}

species my_species {
    int int_attr <- 6;
    string str_attr <- "my_value";
    string str_attr_not_important <- "blabla";
}

grid my_grid_species width: 10 height: 10 {
    int rnd_value <- rnd(5);
}

experiment my_experiment type:gui {
    output {
        inspect name:"my_species_inspector" value:my_species attributes
    :["int_attr","str_attr"];
        inspect name:"my_species_browser" value:my_species type:table;
        inspect name:"my_grid_species_browser" value:5 among
    my_grid_species type:table;
    }
}
```

Another statement, `browse`, is doing a similar thing, but preferring the *table* type (if you want to browse an agent species, the default type will be the *table* type).

Chapter 55

Defining export files

Index

The Save Statement

Allows to save data in a file. The type of file can be “shp”, “text” or “csv”. The **save** statement can be use in an init block, a reflex, an action or in a user command. Do not use it in experiments.

Facets

- **to** (string): an expression that evaluates to an string, the path to the file
- data (any type), (omissible) : any expression, that will be saved in the file
- **crs** (any type): the name of the projectsion, e.g. crs:“EPSG:4326” or its EPSG id, e.g. crs:4326. Here a list of the CRS codes (and EPSG id): http://spatialreference.org
- **rewrite** (boolean): an expression that evaluates to a boolean, specifying whether the save will ecrase the file or append data at the end of it

- **type** (an identifier): an expression that evaluates to an string, the type of the output file (it can be only “shp”, “text” or “csv”)
- **with** (map):

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->"  + name + ":" + location) to: "save_data.txt
    " type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp" with: [name
    ::"nameAgent", location::"locationAgent"] crs: "EPSG:4326";
```

Export files in experiment

Displays are not the only output you can manage in GAMA. Saving data to a file during an experiment can also be achieved in several ways, depending on the needs of the modeler. One way is provided by the `save` statement, which can be used everywhere in a model or a species. The other way, described here, is to include an **output_file** statement in the output section.

```
output_file name:"file_name" type:file_type data:data_to_write;
```

with:

`file_type`: text, csv or xml `file_name`: string `data_to_write`: string

Example:

```
file name: "results" type: text data: time + "; " + nb_preys + ";" +
    nb_predators refresh:every(2);
```

Each time step (or according to the frequency defined in the **refresh** facet of the file output), a new line will be added at the end of the file. If **rewrite**: false is defined in its facets, a new file will be created for each simulation (identified by a timestamp in its name).

Optionally, a **footer** and a **header** can also be described with the corresponding facets (of type string).

Autosave

Image files can be exported also through the **autosave** facet of the display, as explained in this previous part.

Chapter 56

Defining user interaction

During the simulation, GAML provides you the possibility to define some function the user can execute during the execution. In this chapter, we will see how to define buttons to execute action during the simulation, how to catch click event, and how to use the user control architecture.

Index

Catch Mouse Event

You can catch mouse event during the simulation using the statement `event`. This statement has 2 required facets:

- **name** (identifier) : Specify which event do you want to trigger (among the following values : mouse_down, mouse_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;
    }
```

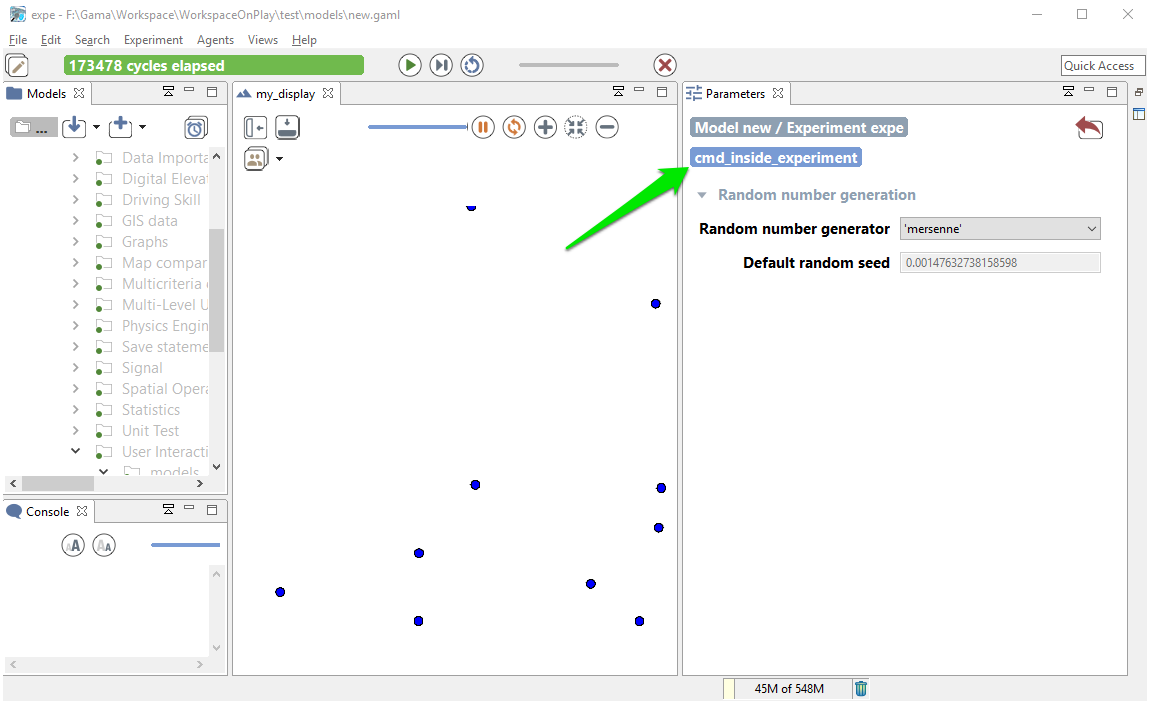


Figure 56.1: images/user_command_inside_expe.png

```
}

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 {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

During the execution, you have 2 ways to access to the action:

- When the agent is inspected, they appear as buttons above the agents' attributes

- 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.

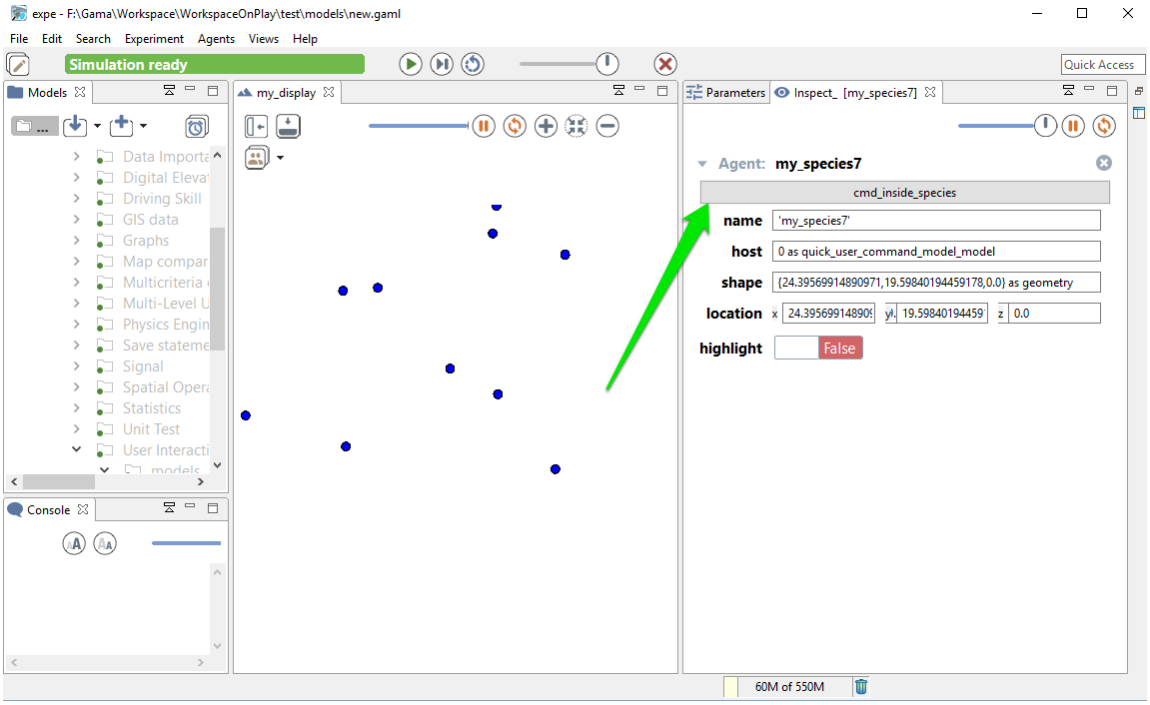


Figure 56.2: images/user_command_inside_species1.png

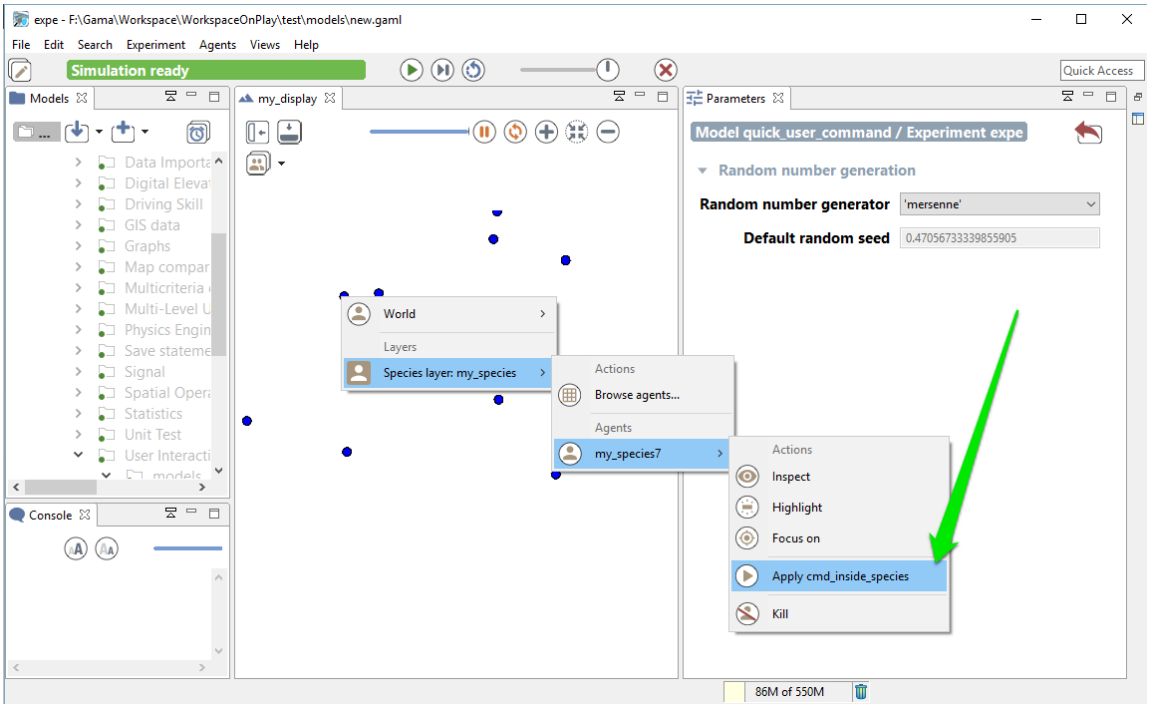


Figure 56.3: images/user_command_inside_species2.png

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 create",["
    Number" :: 100]);
      create my_species number: int(values at "Number");
```

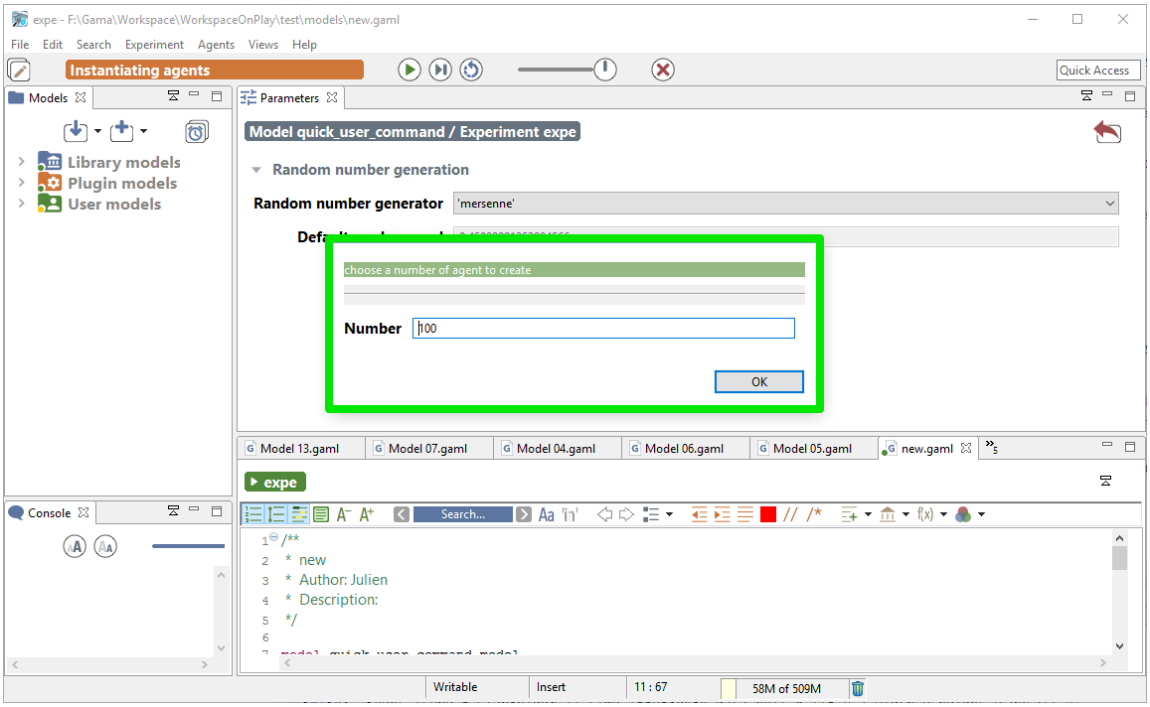


Figure 56.4: images/input_cmd.png

```
    }
}

species my_species {
    aspect base {
        draw circle(1) color:#blue;
    }
}

experiment expe type:gui {
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

When running this model, you will first have to input a number:

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.

Chapter 57

Exploring Models

We just learnt how to launch GUI Experiments from GAMA. A GUI Experiment will start with a particular set of input, compute several outputs, and will stop at the end (if asked).

In order to explore models (by automatically running the Experiment using several configurations to analyze the outputs), a first approach is to run several simulations from the same experiment, considering each simulation as an agent. A second approach, much more efficient for larger explorations, is to run an other type of experiment : the **Batch Experiment**.

We will start this part by learning how to **run several simulations** from the same experiment. Then, we will see how **batch experiments** work, and we will focus on how to use those batch experiments to explore models by using **exploration methods**.

Chapter 58

Run Several Simulations

To explore a model, the easiest and the most intuitive way to proceed is running several simulations with several parameter value, and see the differences from the output. GAMA provides you the possibility to launch several simulations from the GUI.

Index

Create a simulation

Let's remind you that in GAMA, everything is an **agent**. We already saw that the **"world" agent** is the **agent of the model**. The model is thus a **species**, called modelName_model :

```
model toto // <- the name of the species is "toto_model"
```

New highlight of the day : an **Experiment** is also an agent ! It's a special agent which will instantiate automatically an agent from the model species. You can then perfectly create agents (*model* agents) from your experiment, using the statement `create` :

```
model multi_simulations // the "world" is an instance of the "
    multi_simulations_model"

global {
}

experiment my_experiment type:gui  {
    init {
        create multi_simulations_model;
    }
}
```

This sort model will instantiate 2 simulations (two instance of the model) : one is created automatically by the experiment, and the second one is explicitly created through the statement `create`.

To simplify the syntax, you can use the built-in attribute `simulation` of your **experiment**. When you have a model called "multi_simulations", the two following lines are strictly equal :

```
create multi_simulations_model;
create simulation;
```

As it was the case for creating regular species, you can specify the parameters of your agent during the creation through the facet `with:` :

```
model multi_simulations

global {
    rgb bgd_color;
}

experiment my_experiment type:gui  {
    parameter name:"background color:" var:bgd_color init:#blue;
    init {
        create simulation with:[bgd_color::#red];
    }
    output {
        display "my_display" background:bgd_color{}
    }
```

```
}
```

Manipulate simulations

When you think the simulations as agents, it gives you a lot of new possibilities. You can for example create a reflex from your experiment, asking to create simulations **during the experiment execution** !

The following short model for example will create a new simulation at each 10 cycles :

```
model multi_simulations

global {
    init {
        write "new simulation created ! Its name is "+name;
    }
}

experiment my_experiment type:gui  {
    init {
    }
    reflex when:(mod(cycle,10)=0 and cycle!=0) {
        create simulation;
    }
    output {
    }
}
```

You may ask, what is the purpose of such a thing ? Well, with such a short model, it is not very interesting, for sure. But you can imagine running a simulation, and if the simulation reaches a certain state, it can be closed, and another simulation can be run instead with different parameters (a simulation can be closed by doing a "do die" on itself). You can also imagine to run two simulations, and to communicate from one to an other through the experiment, as it is shown in this easy model, where agents can move from one simulation to another :

```
model smallWorld

global {
    int grid_size <- 10;
    bool modelleft <- true;
    int id<- 0;
```

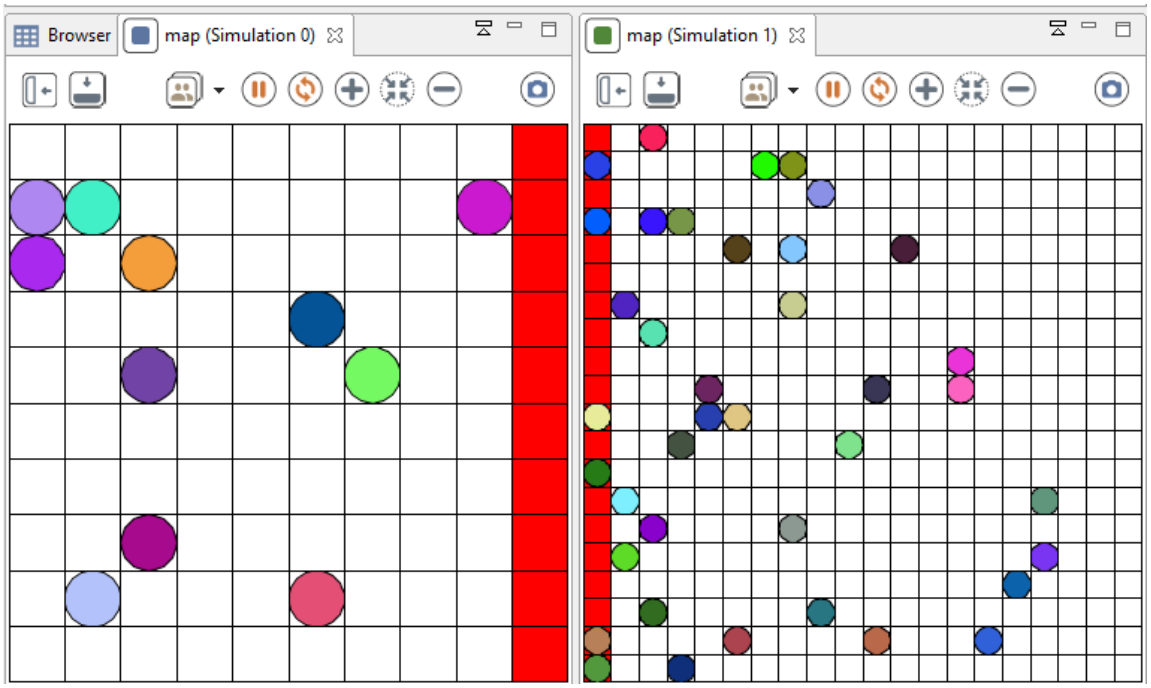


Figure 58.1: resources/images/exploringModel/change_world.png

```
    int nb_agents <- 50;

    init {
        create people number: nb_agents {
            my_cell <- one_of(cell);
            location <- my_cell.location;
        }
        if (modelleft) {
            ask cell where (each.grid_x = (grid_size - 1))  {
                color <- #red;
            }
        } else {
            ask cell where (each.grid_x = 0)  {
                color <- #red;
            }
        }
    }

    action changeWorld(rgb color, point loc) {
        create people with:[color::color, location::loc] {
            my_cell <- cell(location);
        }
    }
}

species people {
    rgb color <- rnd_color(255);
    cell my_cell;

    reflex move {
        if (modelleft and my_cell.color = #red) {
            ask smallWorld_model[1] {
                do changeWorld(myself.color, {100 - myself.location.x,
   myself.location.y});
            }
            do die;
        } else {
            list<cell> free_cells <- list<cell> (my_cell.neighbors)
   where empty(people inside each);
            if not empty(free_cells) {
                my_cell <- one_of(free_cells);
                location <- my_cell.location;
            }
        }

    }
```

```
    aspect default {
        draw circle(50/grid_size) color: color;
    }
}

grid cell width: grid_size height: grid_size;

experiment fromWorldToWorld type: gui {
    init {
         create simulation with:[grid_size::20, modelleft::false, id
    ::1, nb_agents::0];
    }

    output {
        display map {
            grid cell lines: #black;
            species people;
        }
    }
}
```

Here is an other example of application of application, available in the model library. Here we run 4 times the Ant Foraging model, with different parameters.

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 {
```

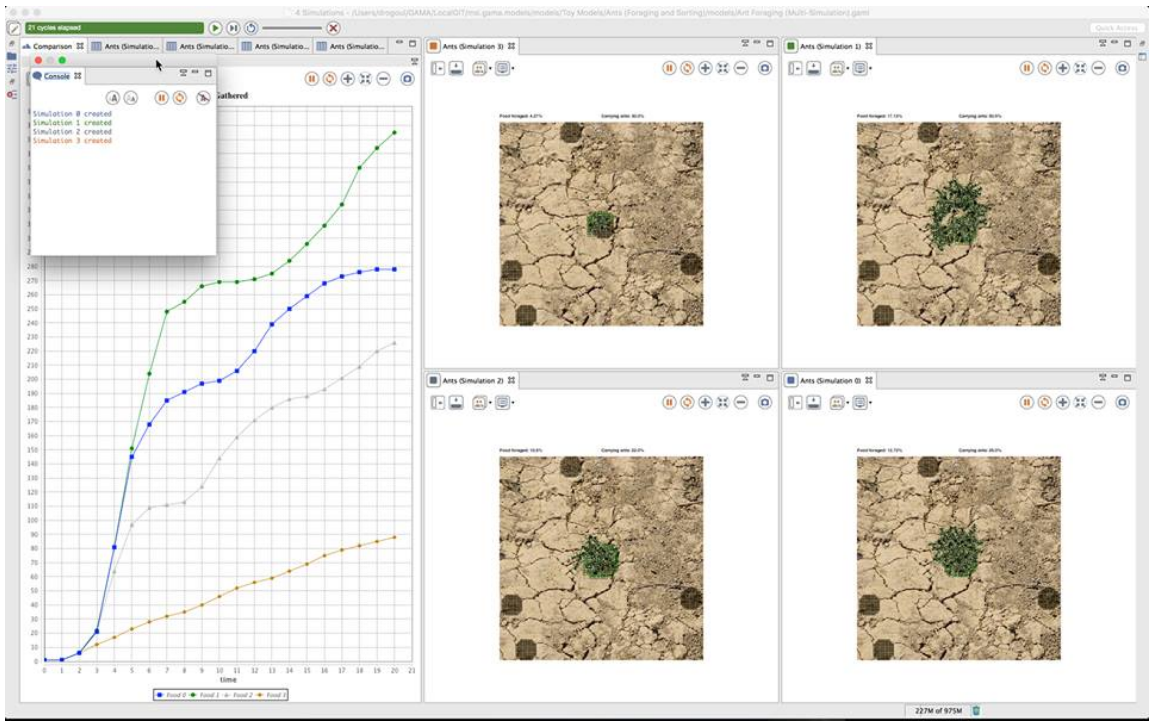


Figure 58.2: resources/images/exploringModel/multi_foraging.jpg

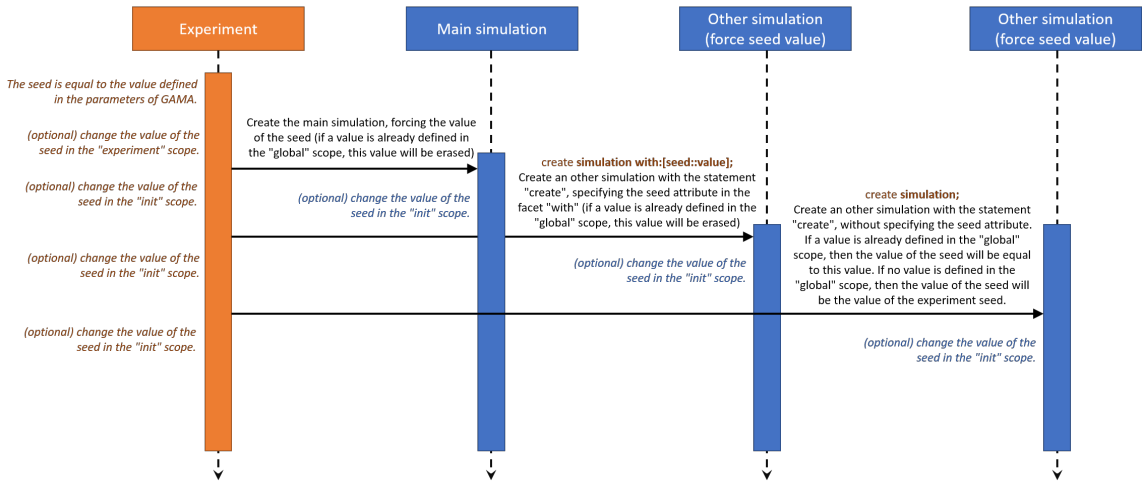


Figure 58.3: resources/images/exploringModel/sequence_diagram_seed_affectation.png

```
        seed<-10.0;
    }
}
```

Notice that if you affect the value of your seed built-in directly in the global scope, the affectation of the parameters (for instance specified with the facet `with` of the statement `create`), and the "init" will be done after will be done at the end.

Defining the seed from the experiment

The experiment agent also have a built-in attribute `seed`. The value of this seed is defined in your simulation preferences. The first simulation created is created **with the seed value of the experiment**.

The following sequence diagram can explain you better how the affectation of the seed attribute works :

The affectation of an attribute is always done in this order : (1) the attribute is affected with a specific value in the species scope. If no attribute value is specified, the value is a default value. (2) if a value is specified for this attribute in the `create` statement, then the attribute value is affected again. (3) the attribute value can be changed again in the `init` scope.

Run several simulations with the same random numbers

The following code shows how to run several simulations with a specific seed, determined from the experiment agent :

```
model multi_simulations

global {
    init {
        create my_species;
    }
}

species my_species skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#green;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
    float seed <- seedValue; // force the value of the seed.
    init {
        // create a second simulation with the same seed as the main
   simulation
        create simulation with:[seed::seedValue];
    }
    output {
        display my_display {
            species my_species aspect:base;
        }
    }
}
```

When you run this simulation, their execution is exactly similar.

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;
```

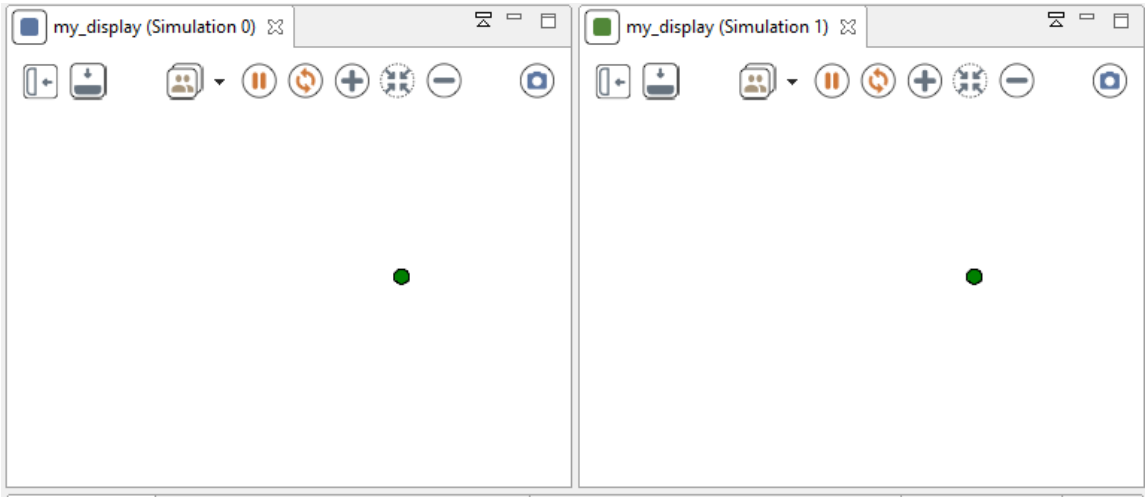


Figure 58.4: resources/images/exploringModel/same_simulation_one_agent.png

```
    init {
        create my_speciesA;
        create my_speciesB number:number_of_speciesB;
    }
}

species my_speciesA skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#green;
    }
}

species my_speciesB skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw circle(2) color:#red;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
```

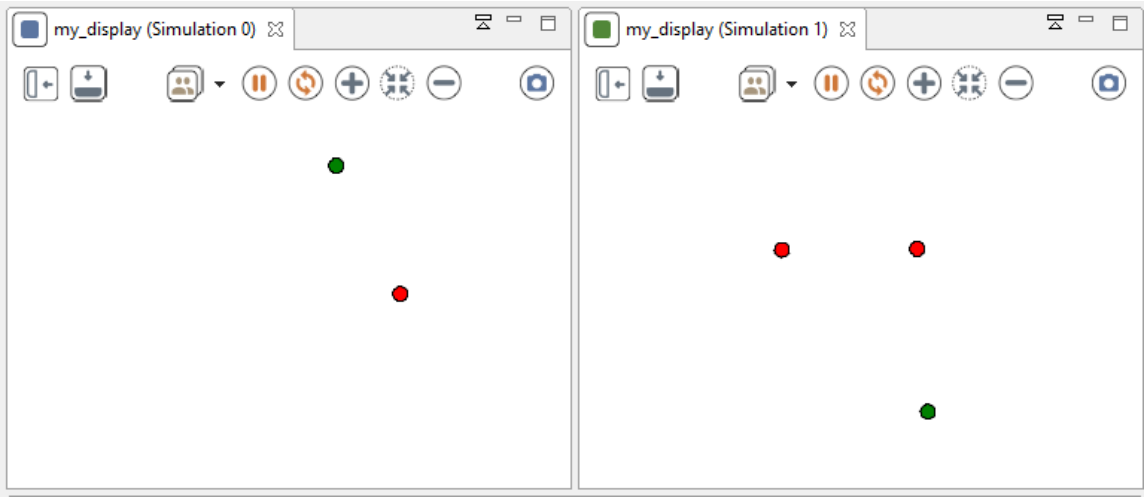


Figure 58.5: resources/images/exploringModel/same_simulation_2_species.png

```
    float seed <- seedValue; // force the value of the seed.
    init {
        create simulation with:[seed::seedValue,number_of_speciesB::2];
    }
    output {
        display my_display {
            species my_speciesA aspect:base;
            species my_speciesB aspect:base;
        }
    }
}
```

Then you run the experiment, you may find something strange...

Even if the first step seems ok (the greed agent and one of the two red agent is initialized with the same location), the simulation differs completly. You should have expected to have the same behavior for the greed agent in both of the simulation, but it is not the case. The explaination of this behavior is that a random number generator has generated more random numbers in the second simulation than in the first one.

If you don't understand, here is a short example that may help you to understand better :

```
model multi_simulations
```

```
global {
    int iteration_number <- 1;
    reflex update {
        float value;
        loop times:iteration_number {
            value<-rnd(10.0);
            write value;
        }
        write "cycle "+cycle+" in experiment "+name+" : "+value;
    }
}

experiment my_experiment type:gui  {
    float seedValue <- 10.0;
    float seed <- seedValue; // force the value of the seed.
    init {
        create simulation with:[seed::seedValue,iteration_number::2];
    }
    output {
    }
}
```

The output will be something like that :

```
7.67003069780383
cycle 0 in experiment multi_simulations_model0 : 7.67003069780383
7.67003069780383
0.22889843360303863
cycle 0 in experiment multi_simulations_model1 : 0.22889843360303863
0.22889843360303863
cycle 1 in experiment multi_simulations_model0 : 0.22889843360303863
4.5220913306263855
0.8363180333035425
cycle 1 in experiment multi_simulations_model1 : 0.8363180333035425
4.5220913306263855
cycle 2 in experiment multi_simulations_model0 : 4.5220913306263855
5.460148568140819
4.158355846617511
cycle 2 in experiment multi_simulations_model1 : 4.158355846617511
0.8363180333035425
cycle 3 in experiment multi_simulations_model0 : 0.8363180333035425
1.886091659169562
4.371253083874633
cycle 3 in experiment multi_simulations_model1 : 4.371253083874633
```

Which means :

Cycle	Value generated in simulation 0	Value generated in simulation 1
1	7.67003069780383	7.67003069780383
		0.22889843360303863
2	0.22889843360303863	4.5220913306263855
		0.8363180333035425
3	4.5220913306263855	5.460148568140819
		4.158355846617511

When writing your models, you have to be aware of this behavior. Remember that each simulation has it's own random number generator.

Change the RNG

The RNG (random number generator) can also be changed : `rng` is a string built-in attribute of the experiment (and also of the model). You can choose among the following rng : - mersenne (by default) - cellular - java

The following model shows how to run 4 simulations with the same seed but with some different RNG :

```
model multi_simulations

global {
    init {
        create my_species number:50;
    }
}

species my_species skills:[moving] {
    reflex update {
        do wander;
    }
    aspect base {
        draw square(2) color:#blue;
    }
}

experiment my_experiment type:gui  {
    float seed <- 10.0;
    init {
        create simulation with:[rng::"cellular",seed::10.0];
```

```
        create simulation with:[rng::"java",seed::10.0];
    }
    output {
        display my_display {
            species my_species aspect:base;
            graphics "my_graphic" {
                draw rectangle(35,10) at:{0,0} color:#lightgrey;
                draw rng at:{3,3} font:font("Helvetica", 20 , #plain)
   color:#black;
            }
        }
    }
}
```

Chapter 59

Defining Batch Experiments

Batch experiments allow to execute numerous successive simulation runs. They are used to explore the parameter space of a model or to optimize a set of model parameters.

A Batch experiment is defined by:

```
experiment exp_title type: batch {
   [parameter to explore]
   [exploration method]
   [reflex]
   [permanent]
}
```

Table of contents

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) A parameter configuration corresponds to a set of values assigned to each parameter. The attribute repeat specifies the number of times each configuration will be repeated, meaning that as many simulations will be run with the same parameter values. Different random seeds are given to the pseudo-random number generator. This allows to get some statistical power from the experiments conducted. Default value is 1.
- keep_seed: (boolean) If true, the same series of random seeds will be used from one parameter configuration to another. Default value is false.

```
experiment my_batch_experiment type: batch repeat: 5 keep_seed: true
    until: time = 300 {
    [parameter to explore]
    [exploration method]
}
```

Action _step_

The _step_ action of an experiment is called at the end of a simulation. It is possible to override this action to apply a specific action at the end of each simulation. Note that at the experiment level, you have access to all the species and all the global variables.

For instance, the following experiment runs the simulation 5 times, and, at the end of each simulation, saves the people agents in a shapefile.

```
experiment 'Run 5 simulations' type: batch repeat: 5 keep_seed: true
    until: ( time > 1000 ) {
     int cpt <- 0;
     action _step_ {
         save people type:"shp" to:"people_shape" + cpt + ".shp" with: [
    is_infected::"INFECTED",is_immune::"IMMUNE"];
         cpt <- cpt + 1;
     }
}
```

A second solution to achieve the same result is to use reflexes (see below).

Reflexes

It is possible to write reflexes inside a batch experiment. This reflex will be executed at the end of each simulation. For instance, the following reflex writes at the end of each simulation the value of the variable *food_gathered*:

```
reflex info_sim {
    write "Running a new simulation " + simulation + " -> " +
    food_gathered;
}
```

Permanent

The **permanent** section allows to define 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. 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;
        }
    }
}
```

Chapter 60

Exploration Methods

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

Table of contents

The method element

The optional method element controls the algorithm which drives the batch.

If this element is omitted, the batch will run in a classical way, changing one parameter value at each step until all the possible combinations of parameter values have been covered. See the Exhaustive exploration of the parameter space for more details.

When used, this element must contain at least a name attribute to specify the algorithm to use. It has theses facets:

- minimize or a maximize (mandatory for optimization method): a attribute defining the expression to be optimized.
- aggregation (optional): possible values ("min", "max"). Each combination of parameter values is tested **repeat** times. The aggregated fitness of one combination is by default the average of fitness values obtained with those repetitions. This facet can be used to tune this aggregation function and to choose to compute the aggregated fitness value as the minimum or the maximum of the obtained fitness values.
- other parameters linked to exploration method (optional) : see below for a description of these parameters.

Exemples of use of the method elements:

```
method exhaustive minimize: nb_infected ;

method genetic pop_dim: 3 crossover_prob: 0.7 mutation_prob: 0.1
    nb_prelim_gen: 1 max_gen: 5  minimize: nb_infected aggregation: "max
    ";
```

Exhaustive exploration of the parameter space

Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way.

Example (models/ants/batch/ant_exhaustive_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
    food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
}
```

The order of the simulations depends on the order of the param. In our example, the first combinations will be the followings:

- evaporation_rate = 0.1, diffusion_rate = 0.1, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 0.4, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 0.7, (2 times)
- evaporation_rate = 0.1, diffusion_rate = 1.0, (2 times)
- evaporation_rate = 0.2, diffusion_rate = 0.1, (2 times)
- ...

Note: this method can also be used for optimization by adding an method element with maximize or a minimize attribute:

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
    method exhaustive maximize: food_gathered;
}
```

Hill Climbing

Name: hill_climbing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article.

Algorithm:

```
Initialization of an initial solution s
iter = 0
While iter <= iter_max, do:
  Choice of the solution s' in the neighborhood of s that maximize the
   fitness function
  If f(s') > f(s)
    s = s'
  Else
    end of the search process
  EndIf
  iter = iter + 1
EndWhile
```

Method parameters:

- iter_max: number of iterations

Example (models/ants/batch/ant_hill_climbing_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
    food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
    method hill_climbing iter_max: 50 maximize : food_gathered;
}
```

Simulated Annealing

Name: annealing Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article.

Algorithm:

```
Initialization of an initial solution s
temp = temp_init
While temp > temp_end, do:
  iter = 0
  While iter < nb_iter_cst_temp, do:
    Random choice of a solution s2 in the neighborhood of s
    df = f(s2)-f(s)
    If df > 0
      s = s2
    Else,
      rand = random number between 0 and 1
      If rand < exp(df/T)
        s = s2
      EndIf
    EndIf
    iter = iter + 1
  EndWhile
```

```
  temp = temp * nb_iter_cst_temp
EndWhile
```

Method parameters:

- temp_init: Initial temperature
- temp_end: Final temperature
- temp_decrease: Temperature decrease coefficient
- nb_iter_cst_temp: Number of iterations per level of temperature

Example (models/ants/batch/ant_simulated_annealing_batch.xml):

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
    food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
    method annealing temp_init: 100  temp_end: 1 temp_decrease: 0.5
    nb_iter_cst_temp: 5 maximize: food_gathered;
}
```

Tabu Search

Name: tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article.

Algorithm:

```
Initialization of an initial solution s
tabuList = {}
iter = 0
While iter <= iter_max, do:
  Choice of the solution s2 in the neighborhood of s such that:
    s2 is not in tabuList
    the fitness function is maximal for s2
  s = s2
  If size of tabuList = tabu_list_size
```

```
    removing of the oldest solution in tabuList
  EndIf
  tabuList = tabuList + s
  iter = iter + 1
EndWhile
```

Method parameters:

- iter_max: number of iterations
- tabu_list_size: size of the tabu list

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
   food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
   'rate every cycle (1.0 means 100%)' step: 0.3;
    method tabu iter_max: 50 tabu_list_size: 5 maximize: food_gathered;
}
```

Reactive Tabu Search

Name: reactive_tabu Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle.

Method parameters:

- iter_max: number of iterations
- tabu_list_size_init: initial size of the tabu list
- tabu_list_size_min: minimal size of the tabu list
- tabu_list_size_max: maximal size of the tabu list
- nb_tests_wthout_col_max: number of movements without collision before shortening the tabu list
- cycle_size_min: minimal size of the considered cycles
- cycle_size_max: maximal size of the considered cycles

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
    food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
    method reactive_tabu iter_max: 50 tabu_list_size_init: 5
    tabu_list_size_min: 2 tabu_list_size_max: 10 nb_tests_wthout_col_max
    : 20 cycle_size_min: 2 cycle_size_max: 20 maximize: food_gathered;
}
```

Genetic Algorithm

Name: genetic Parameter definitions accepted: List with step and Explicit List. Parameter type accepted: all.

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article. The principle of GA is to search an optimal solution by applying evolution operators on an initial population of solutions There are three types of evolution operators:

- Crossover: Two solutions are combined in order to produce new solutions
- Mutation: a solution is modified
- Selection: only a part of the population is kept. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Representation of the solutions:

- Individual solution: {Param1 = val1; Param2 = val2; ...}
- Gene: Parami = vali

Initial population building: the system builds nb_prelim_gen random initial populations composed of pop_dim individual solutions. Then, the best pop_dim solutions are selected to be part of the initial population.

Selection operator: roulette-wheel selection: the probability to choose a solution is 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 mute to {Param1 = 3; Param2 = 4}

Crossover operator: A cut point is randomly selected and two new solutions are built by taking the half of each parent solution. Ex: let {Param1 = 4; Param2 = 1} and {Param1 = 2; Param2 = 3} be two solutions. The crossover operator builds two new solutions: {Param1 = 2; Param2 = 1} and {Param1 = 4; Param2 = 3}.

Method parameters:

- pop_dim: size of the population (number of individual solutions)
- crossover_prob: crossover probability between two individual solutions
- mutation_prob: mutation probability for an individual solution
- nb_prelim_gen: number of random populations used to build the initial population
- max_gen: number of generations

```
experiment Batch type: batch repeat: 2 keep_seed: true until: (
    food_gathered = food_placed ) or ( time > 400 ) {
    parameter 'Evaporation:' var: evaporation_rate among: [ 0.1 , 0.2 ,
     0.5 , 0.8 , 1.0 ] unit: 'rate every cycle (1.0 means 100%)';
    parameter 'Diffusion:' var: diffusion_rate min: 0.1 max: 1.0 unit:
    'rate every cycle (1.0 means 100%)' step: 0.3;
    method genetic maximize: food_gathered pop_dim: 5 crossover_prob:
    0.7 mutation_prob: 0.1 nb_prelim_gen: 1 max_gen: 20;
}
```

Chapter 61

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).

Chapter 62

Runtime Concepts

When a model is being simulated, a number of algorithms are applied, for instance to determine the order in which to run the different agents, or the order in which the initialization of agents is performed, etc. This section details some of them, which can be important when building models and understanding how they will be effectively simulated.

Table of contents

Simulation initialization

Once the user launches an experiment, GAMA starts the initialization of the simulation. First it creates a `world` agent.

It initializes all its attributes with their init values. This includes its shape (that will be used as environment of the simulation).

If a species of type grid exists in the model, agents of species are created.

Finally the **init** statement is executed. It should include the creation of all the other agents of regular species of the simulation. After their creation and initialization, they are added in the list `members` the `world` (that contains all the micro-agent of the `world`).

Agents Creation

Except `grid` agents, other agents are created using the `create` statement. It is used to allocate memory for each agent and to initialize all its attributes.

If no explicit initialization exists for an attribute, it will get the default value corresponding to its type.

The initialization of an attribute can be located at several places in the code; they are executed in the following order (which means that, if several ways are used, the attribute will finally have the value of the last applied one):

- using the **from:** facet of the `create` statement;
- in the embedded block of the `create` statement;
- in the attribute declaration, using the **init** facet;
- in the **init** block of the species.

Agents Step

When an agent is asked to *step*, it means that it is expected to update its variables, run its behaviors and then *step* its micro-agents (if any).

```
step of agent agent_a
    {
        species_a <- agent_a.species
        architecture_a <- species_a.architecture
        ask architecture_a to step agent_a {
             ask agent_a to update species_a.variables
             ask agent_a to run architecture_a.behaviors
        }

        ask each micro-population mp of agent_a to step {
            list<agent> sub-agents <- mp.compute_agents_to_schedule
```

```
            ask each agent_b of sub-agents to step //... recursive call
    ...
        }
    }
```

Schedule Agents

The global scheduling of agents is then simply the application of this previous *step* to the *experiment agent*, keeping in mind that this agent has only one micro-population (of simulation agents, each instance of the model species), and that the simulation(s) inside this population contain(s), in turn, all the "regular" populations of agents of the model.

To influence this schedule, then, one possible way is to change the way populations compute their lists of agents to schedule, which can be done in a model by providing custom definitions to the "schedules:" facet of one or several species.

A practical application of this facet is to reduce simulation artifacts created by the default scheduling of populations, which is sequential (i.e. their agents are executed in turn in their order of creation). To enable a pseudo-parallel scheduling based on a random scheduling recomputed at each step, one has simply to define the corresponding species like in the following example:

```
species A schedules: shuffle(A) {...}
```

Moving further, it is possible to enable a completely random scheduling that will eliminate the sequential scheduling of populations:

```
global schedules: [world] + shuffle(A + B + C) {...}

species A schedules: [] {...}
species B schedules: [] {...}
species C schedules: [] {...}
```

It is important to (1) explicitly invoke the scheduling of the world (although it doesn't have to be the first); (2) suppress the population-based scheduling to avoid having agent being scheduled 2 times (one time in the custom definition, one time by their population).

Other schemes are possible. For instance, the following definition will completely suppress the default scheduling mechanism to replace it with a custom scheduler that

will execute the world, then all agents of species A in a random way and then all agents of species B in their order of creation:

```
global schedules: [world] + shuffle(A) + B {...} // explicit scheduling
    in the world

species A schedules [];

species B schedules: [];
```

Complex conditions can be used to express which agents need to be scheduled each step. For instance, in the following definition, only agents of A that return true to a particular condition are scheduled:

```
species A schedules: A where each.can_be_scheduled() {

    bool can_be_scheduled() {
        ...
        returns true_or_false;
    }
}
```

Be aware that enabling a custom scheduling can potentially end up in non-functional simulations. For example, the following definitions will result in a simulation that will **never be executed**:

```
global schedules: [] {}; // the world is NEVER scheduled

species my_scheduler schedules: [world] ; // so its micro-species '
   my_scheduler' is NOT scheduled either.
```

and this one will result in an **infinite loop** (which will trigger a stack overflow at some point):

```
global {} // The world is normally scheduled...

species my_scheduler schedules: [world]; // ... but schedules itself
   again as a consequence of scheduling the micro-species 'my_scheduler
   '
```

Chapter 63

Optimizing Models

This page aims at presenting some tips to optimize the memory footprint or the execution time of a model in GAMA.

Note: since GAMA 1.6.1, some optimizations have become obsolete because they have been included in the compiler. They have, then, been removed from this page. For instance, writing 'rgb(0,0,0)' is now compiled directly as '#black'.

Table of contents

machine_time

In order to optimize a model, it is important to exactly know which part of the model take times. The simplest to do that is to use the **machine_time** built-in global variable that gives the current time in milliseconds. Then to compute the time taken by a statement, a possible way is to write:

```
float t <- machine_time;
// here a block of instructions that you consider as "critical"
// ...
write "duration of the last instructions: " + (machine_time - t);
```

Scheduling

If you have a species of agents that, once created, are not supposed to do anything more (i.e. no behavior, no reflex, their actions triggered by other agents, their attributes being simply read and written by other agents), such as a "data" grid, or agents representing a "background" (from a shape file, etc.), consider using the `schedules: []` facet on the definition of their species. This trick allows to tell GAMA to not schedule any of these agents.

```
grid my_grid height: 100 width: 100 schedules: []
{
      ...
}
```

The `schedules:` facet is dynamically computed (even if the agents are not scheduled), so, if you happen to define agents that only need to be scheduled every x cycles, or depending on a condition, you can also write `schedules:` to implement this. For instance, the following species will see its instances scheduled every 10 steps and only if a certain condition is met:

```
species my_species schedules: (every 10) ? (condition ? my_species :
    []) : []
{
     ...
}
```

In the same way, modelers can use the frequency facet to define when the agents of a species are going to be activated. By setting this facet to 0, the agents are never activated.

```
species my_species frequency: 0
{
     ...
}
```

Grid

Optimization Facets

In this section, we present some facets that allow to optimize the use of grid (in particular in terms of memories). Note that all these facet can be combined (see the Life model from the Models library).

use_regular_agents

If false, then a special class of agents is used. This special class of agents used less memories but has some limitation: the agents cannot inherit from a "normal" species, they cannot have sub-populations, their name cannot be modified, etc.

```
grid cell width: 50 height: 50 use_regular_agents: false ;
```

use_individual_shapes

If false, then only one geometry is used for all agents. This facet allows to gain a lot of memory, but should not be used if the geometries of the agents are often activated (for instance, by an aspect).

```
grid cell width: 50 height: 50 use_individual_shapes: false ;
```

Operators

List operators

first_with

It is sometimes necessary to randomly select an element of a list that verifies a certain condition. Many modelers use the **one_of** and the **where** operators to do this:

```
bug one_big_bug <- one_of (bug where (each.size > 10));
```

Whereas it is often more optimized to use the **shuffle** operator to shuffle the list, then the **first_with** operator to select the first element that verifies the condition:

```
bug one_big_bug <- shuffle(bug) first_with (each.size > 10);
```

where / count

It is quite common to want to count the number of elements of a list or a container that verify a condition. The obvious to do it is :

```
int n <- length(my_container where (each.size > 10));
```

This will however create an intermediary list before counting it, and this operation can be time consuming if the number of elements is important. To alleviate this problem, GAMA includes an operator called **count** that will count the elements that verify the condition by iterating directly on the container (no useless list created) :

```
int n <- my_container count (each.size > 10);
```

Spatial operators

container of agents in closest_to, at_distance, overlapping, inside

Several spatial query operators (such as **closest_to**, **at_distance**, **overlapping** or **inside**) allow to restrict the agents being queried to a container of agents. For instance, one can write:

```
agent closest_agent <- a_container_containing_agents closest_to self;
```

This expression is formally equivalent to :

```
agent closest_agent <- a_container_containing_agent with_min_of (each
    distance_to self);
```

But it is much faster **if your container is large**, as it will query the agents using a spatial index (instead of browsing through the whole container). Note that in some cases, when you have a small number of agents, the first syntax will be faster. The same applies for the other operators.

Now consider a very common case: you need to restrict the agents being queried, not to a container, but to a species (which, actually, acts as a container in most cases). For instance, you want to know which predator is the closest to the current agent. If we apply the pattern above, we would write:

```
predator closest_predator <- predator with_min_of (each distance_to
    self);
```

or

```
predator closest_predator <- list(predator) closest_to self;
```

But these two operators can be painfully slow if your species has many instances (even in the second form). In that case, always prefer using **directly** the species as the left member:

```
predator closest_ predator <- predator closest_to self;
```

Not only is the syntax clearer, but the speed gain can be phenomenal because, in that case, the list of instances is not used (we just check if the agent is an instance of the left species).

However, what happens if one wants to query instances belonging to 2 or more species ? If we follow our reasoning, the immediate way to write it would be (if predator 1 and predator 2 are two species):

```
agent closest_agent <- (list(predator1) + list(predator2)) closest_to
   self;
```

or, more simply:

```
agent closest_agent <- (predator1 + predator2) closest_to self;
```

The first syntax suffers from the same problem than the previous syntax: GAMA has to browse through the list (created by the concatenation of the species populations) to filter agents. The solution, then, is again to use directly the species, as GAMA is clever enough to create a temporary "fake" population out of the concatenation of several species, which can be used exactly like a list of agents, but provides the advantages of a species population (no iteration made during filtering).

Accelerate closest_to with a first spatial filtering

The **closest_to** operator can sometimes be slow if numerous agents are concerned by this query. If the modeler is just interested by a small subset of agents, it is possible to apply a first spatial filtering on the agent list by using the **at_distance** operator. For example, if the modeler wants first to do a spatial filtering of 10m:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self;
```

To be sure to find an agent, the modeler can use a test statement:

```
agent closest_agent <- (predator1 at_distance 10) closest_to self;
if (closest_agent = nil) {closest_agent  <- predator1 closest_to self;}
```

Displays

shape

It is quite common to want to display an agent as a circle or a square. A common mistake is to mix up the shape to draw and the geometry of the agent in the model.

If the modeler just wants to display a particular shape, he/she should not modify the agent geometry (which is a point by default), but just specify the shape to draw in the agent aspect.

```
species bug {
     int size <- rnd(100);

      aspect circle {
          draw circle(2) color: °blue;
      }
}
```

circle vs square / sphere vs cube

Note that in OpenGL and Java2D (the two rendering subsystems used in GAMA), creating and drawing a circle geometry is more time consuming than creating and drawing a square (or a rectangle). In the same way, drawing a sphere is more time consuming than drawing a cube. Hence, if you want to optimize your model displays and if the rendering does not explicitly need "rounded" agents, try to use squares/cubes rather than circles/spheres.

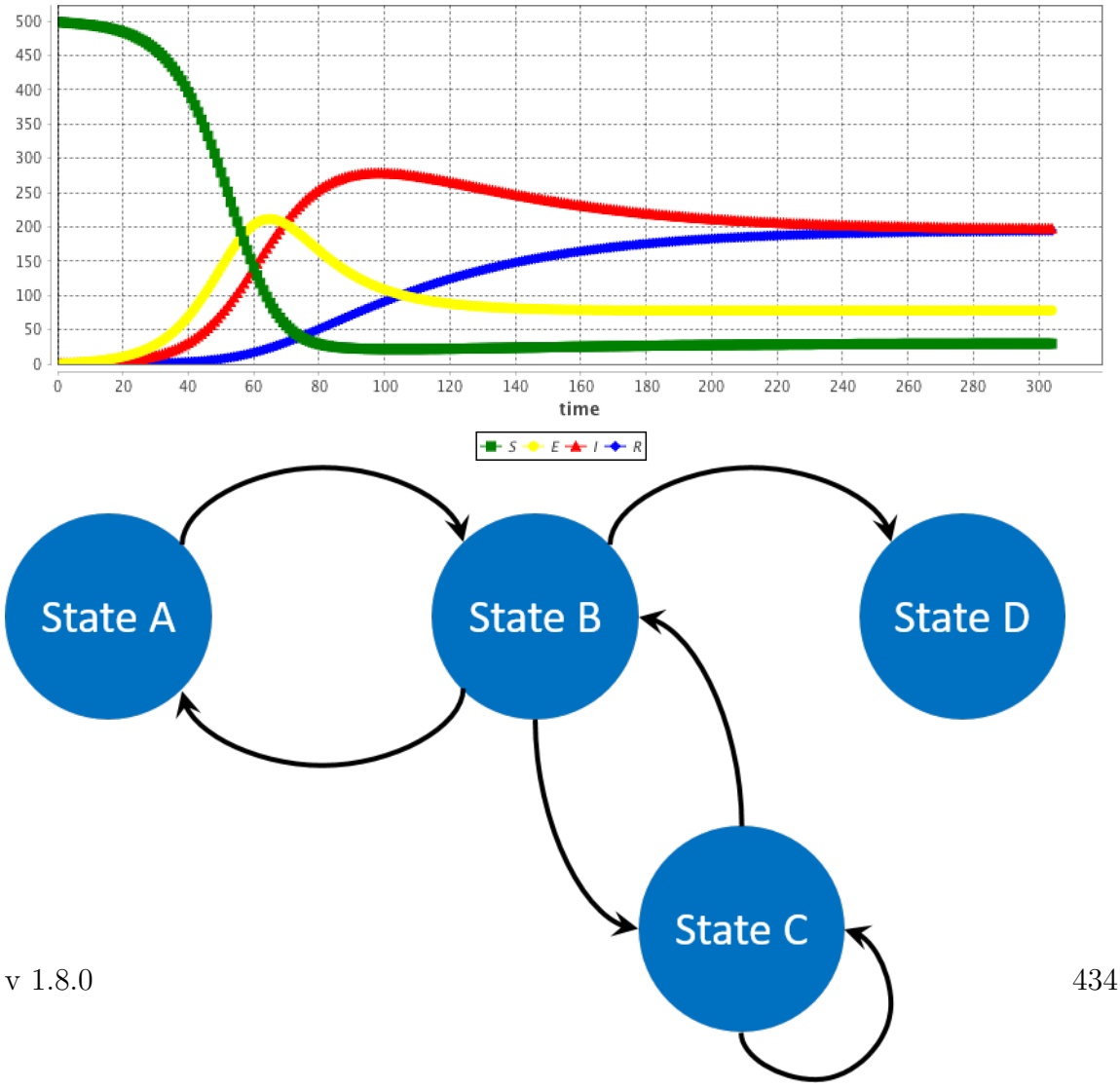
OpenGL refresh facets

In OpenGL display, it is possible to specify that it is not necessary to refresh a layer with the facet **refresh**. If a species of agents is never modified in terms of visualization (location, shape or color), you can set **refresh** to false. Example:

```
display city_display_opengl type: opengl{
     species building aspect: base refresh: false;
     species road aspect: base refresh: false;
     species people aspect: base;
}
```

Chapter 64

Multi-Paradigm Modeling



Multi-paradigm modeling is a research field focused on how to define a model semantically. From the beginning of this step by step tutorial, our approach is based on behavior (or reflex), for each agents. In this part, we will see that GAMA provides other ways to implement your model, using several control architectures. Sometime, it will be easier to implement your models choosing other paradigms.

In a first part, we will see how to use some **control architectures** which already exist in GAML, such as finite state machine architecture, task based architecture or user control architecture. In a second part, we will see an other approach, a math approach, through **equations**.

Chapter 65

Control Architectures

GAMA allows to attach built-in control architecture to agents.

These control architectures will give the possibility to the modeler to use for a species a specific control architecture in addition to the common behavior structure. Note that only one control architecture can be used per species.

The attachment of a control architecture to a species is done through the facets **control**.

For example, the given code allows to attach the `fsm` control architecture to the dummy species.

```
species dummy control: fsm {
}
```

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

- fsm: finite state machine based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is in one single state. Such an agent needs to have one initial state (the state in which it will be at its initialization)
- weighted_tasks: task-based control architecture. At any given time, only the task only the task with the maximal weight is executed.
- sorted_tasks: task-based control architecture. At any given time, the tasks are all executed in the order specified by their weights (highest first).

- probabilistic_tasks: task-based control architecture. This architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen at each step.
- user_only: allows users to take control over an agent during the course of the simulation. With this architecture, only the user control the agents (no reflexes).
- user_first: allows users to take control over an agent during the course of the simulation. With this architecture, the user actions are executed before the agent reflexes.
- user_last: allows users to take control over an agent during the course of the simulation. With this architecture, the user actions are executed after the agent reflexes.

Index

Finite State Machine

FSM (Finite State Machine) is a finite state machine based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is

in one single state. Such an agent needs to have one initial state (the state in which it will be at its initialization).

At each time step, the agent will:

- first (only if he just entered in its current state) execute statement embedded in the `enter` statement,
- then all the statements in the state statement
- it will evaluate the condition of each embedded transition statements. If one condition is fulfilled, the agent execute the embedded statements

Note that an agent executes only one state at each step.

Declaration

Using the FSM architecture for a species require to use the **control** facet:

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

State

Attributes

- initial: a boolean expression, indicates the initial state of agent.
- final: a boolean expression, indicates the final state of agent.

Sub Statements

- enter: a sequence of statements to execute upon entering the state.
- exit: a sequence of statements to execute right before exiting the state. Note that the `exit` statement will be executed even if the fired transition points to the same state (the FSM architecture in GAMA does not implement 'internal transitions' like the ones found in UML state charts: all transitions, even "self-transitions", follow the same rules).

- transition: allows to define a condition that, when evaluated to true, will designate the next state of the life cycle. Note that the evaluation of transitions is short-circuited: the first one that evaluates to true, in the order in which they have been defined, will be followed. I.e., if two transitions evaluate to true during the same time step, only the first one will be triggered.

Things worth to be mentioned regarding these sub-statements:

- Obviously, only one definition of exit and enter is accepted in a given state
- Transition statements written in the middle of the state statements will only be evaluated at the end, so, even if it evaluates to true, the remaining of the statements found after the definition of the transition will be nevertheless executed. So, despite the appearance, a transition written somewhere in the sequence will "not stop" the state at that point (but only at the end).

Definition

A state can contain several statements that will be executed, at each time step, by the agent. There are three exceptions to this rule:

1. statements enclosed in `enter` will only be executed when the state is entered (after a transition, or because it is the initial state).
2. Those enclosed in `exit` will be executed when leaving the state as a result of a successful transition (and before the statements enclosed in the transition).
3. Those enclosed in a transition will be executed when performing this transition (but after the `exit` sequence has been executed).

For example, consider the following example:

```
species dummy control: fsm {
    state state1 initial: true {
        write string(cycle) + ":" + name + "->" + "state1";
        transition to: state2 when: flip(0.5) {
            write string(cycle) + ":" + name + "->" + "transition to
   state1";
        }
        transition to: state3 when: flip(0.2) ;
    }
```

```
    state state2 {
        write string(cycle) + ":" + name + "->" + "state2";
        transition to: state1 when: flip(0.5) {
            write string(cycle) + ":" + name + "->" + "transition to
    state1";
        }
        exit {
            write string(cycle) + ":" + name + "->" + "leave state2";
        }
    }

    state state3 {
        write string(cycle) + ":" + name + "->" + "state3";
        transition to: state1 when: flip(0.5)  {
            write string(cycle) + ":" + name + "->" + "transition to
    state1";
        }
        transition to: state2 when: flip(0.2)  ;
    }
}
```

the dummy agents start at *state1*. At each simulation step they have a probability of 0.5 to change their state to *state2*. If they do not change their state to *state2*, they have a probability of 0.2 to change their state to *state3*. In *state2*, at each simulation step, they have a probability of 0.5 to change their state to *state1*. At last, in *step3*, at each simulation step they have a probability of 0.5 to change their state to *state1*. If they do not change their state to *state1*, they have a probability of 0.2 to change their state to *state2*.

Here a possible result that can be obtained with one dummy agent:

```
0:dummy0->state1
0:dummy0->transition to state1
1:dummy0->state2
2:dummy0->state2
2:dummy0->leave state2
2:dummy0->transition to state1
3:dummy0->state1
3:dummy0->transition to state1
4:dummy0->state2
5:dummy0->state2
5:dummy0->leave state2
5:dummy0->transition to state1
6:dummy0->state1
7:dummy0->state3
```

```
8:dummy0->state2
```

Task Based

GAMA integrated several **task-based** control architectures. Species can define any number of tasks within their body. At any given time, only one or several tasks are executed according to the architecture chosen:

- **weighted_tasks** : in this architecture, only the task with the maximal weight is executed.
- **sorted_tasks** : in this architecture, the tasks are all executed in the order specified by their weights (biggest first)
- **probabilistic_tasks**: this architecture uses the weights as a support for making a weighted probabilistic choice among the different tasks. If all tasks have the same weight, one is randomly chosen each step.

Declaration

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

```
species dummy control: weighted_tasks {
   ...
}
```

```
species dummy control: sorted_tasks {
   ...
}
```

```
species dummy control: probabilistic_tasks {
   ...
}
```

Task

Sub elements

Besides a sequence of statements like reflex, a task contains the following sub elements:

- weight: Mandatory. The priority level of the task.

Definition

As reflex, a task is a sequence of statements that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler chooses a task to execute based on its current priority weight value.

For example, consider the following example:

```
species dummy control: weighted_tasks {
    task task1 weight: cycle mod 3 {
        write string(cycle) + ":" + name + "->" + "task1";
    }
    task task2 weight: 2 {
        write string(cycle) + ":" + name + "->" + "task2";
    }
}
```

As the **weighted_tasks** control architecture was chosen, at each simulation step, the dummy agents execute only the task with the highest behavior. Thus, when *cycle modulo 3* is higher to 2, task1 is executed; when *cycle modulo 3* is lower than 2, task2 is executed. In case when *cycle modulo 3* is equal 2 (at cycle 2, 5, ...), the only the first task defined (here task1) is executed.

Here the result obtained with one dummy agent:

```
0:dummy0->task2
1:dummy0->task2
2:dummy0->task1
3:dummy0->task2
4:dummy0->task2
5:dummy0->task1
6:dummy0->task2
```

User Control Architecture

user_only, user_first, user_last

A specific type of control architecture has been introduced to allow users to take control over an agent during the course of the simulation. It can be invoked using

three different keywords: user_only, user_first, user_last.

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

If the control chosen is user_first, it means that the user controlled panel is opened first, and then the agent has a chance to run its "own" behaviors (reflexes, essentially, or "init" in the case of a "user_init" panel). If the control chosen is user_last, it is the contrary.

user_panel

This control architecture is a specialization of the Finite State Machine Architecture where the "behaviors" of agents can be defined by using new constructs called user_panel (and one user_init), mixed with "states" or "reflexes". This user_panel translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each user_panel, like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of user_commands which describe the different action buttons present in the panel.

user_commands can also accept inputs, in order to create more interesting commands for the user. This uses the user_input statement (and not operator), which is basically the same as a temporary variable declaration whose value is asked to the user. Example:

As user_panel is a specialization of state, the modeler has the possibility to describe several panels and choose the one to open depending on some condition, using the same syntax than for finite state machines : * either adding transitions to the user_panels, * or setting the state attribute to a new value, from inside or from another agent.

This ensures a great flexibility for the design of the user interface proposed to the user, as it can be adapted to the different stages of the simulation, etc...

Follows a simple example, where, every 10 steps, and depending on the value of an attribute called "advanced", either the basic or the advanced panel is proposed. (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;
  }

  user_panel "Basic Control" {
    user_command "Kill one cell" {
      ask (one_of(cell)){
        do die;
      }
    }
    user_command "Create one cell" {
      create cell ;
    }
    transition to: default when: true;
  }
  user_panel "Advanced Control" {
    user_command "Kill cells" {
      user_input "Number" returns: number type: int <- 10;
      ask (number among cell){
        do die;
      }
    }
    user_command "Create cells" {
      user_input "Number" returns: number type: int <- 10;
      create cell number: number ;
    }
    transition to: default when: true;
  }
}
```

The panel marked with the "*initial: true*" facet will be the one run first when the agent is supposed to run. If none is marked, the first panel (in their definition order) is chosen.

A special panel called user_init will be invoked only once when initializing the agent if it is defined. If no panel is described or if all panels are empty (i.e. no user_-commands), the control view is never invoked. If the control is said to be "user_only", the agent will then not run any of its behaviors.

user_controlled

Finally, each agent provided with this architecture inherits a boolean attribute called `user_controlled`. If this attribute becomes false, no panels will be displayed and the agent will run "normally" unless its species is defined with a `user_only` control.

Other Control Architectures

Some other control architectures are available in additional plugins. For instance, BDI (Belief, desire, intention) architecture is available. Feel free to read about it if you want to learn more.

You need some other control architectures for your model? Feel free to make your suggestion to the team of developer through the mailing list. Remember also that GAMA is an open-source platform, you can design your own control architecture easily. Go to the section Community/contribute if you want to jump into coding!

Chapter 66

Using Equations

Introduction

ODEs (Ordinary Differential Equations) are often used in ecology or in epidemiology to describe the macroscopic evolution over time of a population. Generally the whole population is split into several compartments. The state of the population is described by the number of individuals in each compartment. Each equation of the ODE system describes the evolution of the number of individual in a compartment. In such an approach individuals are not taken into account individually, with own features and behaviors. In contrary they are aggregated in a compartment and reduced to a number.

A classical example is the SIR epidemic model representing the spreading of a disease in a population. The population is split into 3 compartments: S (Susceptible), I (Infected), R (Recovered). (see below for the equation)

In general the ODE systems cannot be analytically solved, i.e. it is not possible to find the equation describing the evolution of the number of S, I or R. But these systems can be numerically integrated in order to get the evolution. A numerical integration computes step after step the value of S, I and R. Several integration methods exist (e.g. Euler, Runge-Kutta...), each of them being a compromise between accuracy and computation time. The length of the integration step has also a huge impact on precision. These models are deterministic.

This approach makes a lot of strong hypotheses. The model does not take into account space. The population is considered has infinite and homogeneously mixed, so that any agent can interact with any other one.

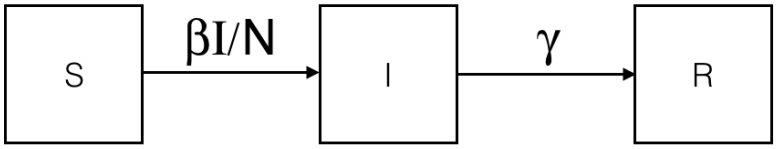


Figure 66.1: SIR-compartment.png

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I \\ \dfrac{dR}{dt} = \gamma I \end{cases}$$

Figure 66.2: SIR-equations.png

Example of a SIR model

In the SIR model, the population is split into 3 compartments: S (Susceptible), I (Infected), R (Recovered). This can be represented by the following Forrester diagram: boxes represent stocks (i.e. compartments) and arrows are flows. Arrows hold the rate of a compartment population flowing to another compartment.

The corresponding ODE system contains one equation per stock. For example, the I compartment evolution is influenced by an inner (so positive) flow from the S compartment and an outer (so negative) flow to the R compartment.

Integrating this system using the Runge-Kutta 4 method provides the evolution of S, I and R over time. The initial values are:

- S = 499

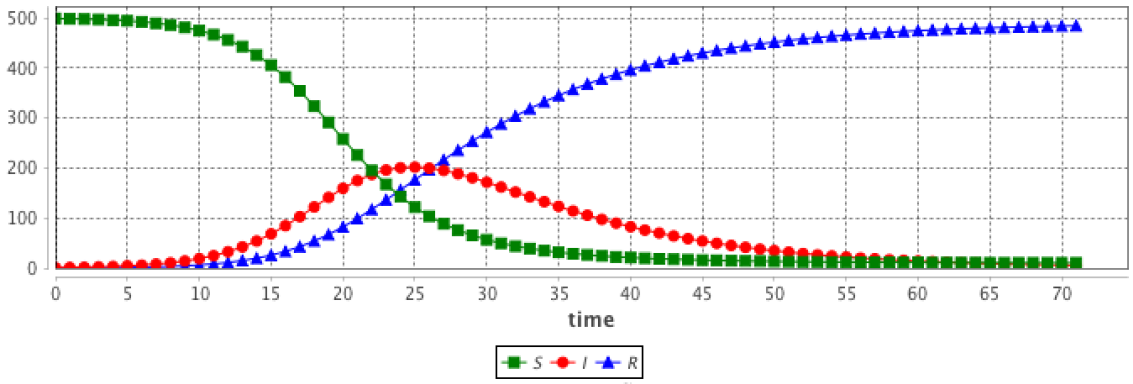


Figure 66.3: SIR-result.png

- I = 1
- R = 0
- beta = 0.4
- gamma = 0.1
- h = 0.1

Why and when can we use ODE in agent-based models ?

These hypotheses are very strong and cannot be fulfilled in agent-based models.

But in some multi-scale models, some entities can be close. For example if we want to implement a model describing the worldwide epidemic spread and the impact of air traffic on it, we cannot simulate the 7 billions people. But we can represent only cities with airports and airplanes as agents. In this case, cities are entities with a population of millions inhabitants, that will not been spatially located. As we are only interested in the disease spread, we are only interested in the number of infected people in the cities (and susceptibles and recovered too). As a consequence, it appears particularly relevant to describe the evolution of the disease in the city using a ODE system.

In addition these models have the advantage to not be sensible to population size in the integration process. Dozens or billions people does not bring a computation time increase, contrarily to agent-based models.

Use of ODE in a GAML model

A stereotypical use of ODE in a GAMA agent-based model is to describe species where some agents attributes evolution is described using an ODE system.

As a consequence, the GAML language has been increased by two main concepts (as two statements):

- equations can be written with the `equation` statement. An `equation` block is composed of a set of **`diff`** statement describing the evolution of species attributes.
- an equation can be numerically integrated using the `solve` statement

equation

Defining an ODE system

Defining a new ODE system needs to define a new `equation` block in a species. As 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 compartment:

```
species userSI {
    float t ;
    float I ;
    float S ;
    int N ;
    float beta<-0.4 ;
    float h ;

    equation eqSI {
        diff(S,t) = -beta * S * I / N ;
        diff(I,t) = beta * S * I / N ;
    }
}
```

This equation has to be defined in a species with `t`, `S` and `I` attributes. **`beta`** (and other similar parameters) can be defined either in the specific species (if it is specific to each agent) or in the `global` if it is a constant.

Note: the `t` attribute will be automatically updated using the `solve` statement; it contains the time elapsed in the equation integration.

Using a built-in ODE system

In order to ease the use of very classical ODE system, some built-in systems have been implemented in GAMA. For example, the previous SI system can be written as follows. Three additional facets are used to define the system:

- **type**: the identifier of the built-in system (here SI) (the list of all built-in systems are described below),
- **vars**: this facet is expecting a list of variables of the species, that will be matched with the variables of the system,
- **params**: this facet is expecting a list of variables of the species (of the global), that will be matched with the parameters of the system.

```
equation eqBuiltInSI type: SI vars: [S,I,t] params: [N,beta] ;
```

Split a system into several agents

An equation system can be split into several species and each part of the system are synchronized using the **simultaneously** facet of `equation`. The system split into several agents can be integrated using a single call to the `solve` statement. Notice that all the `equation` definition must have the same name.

For example, the SI system presented above can be defined in two different species `S_agt` (containing the equation defining the evolution of the S value) and `I_agt` (containing the equation defining the evolution of the I value). These two equations are linked using the **simultaneously** facet of the `equation` statement. This facet expects a set of agents. The integration is called only once in a simulation step, e.g. in the `S_agt` agent.

```
species S_agt {
    float t ;
    float Ssize ;

    equation evol simultaneously: [ I_agt ] {
        diff(Ssize, t) = (- sum(I_agt accumulate [each.beta * each.
   Isize]) * self.Ssize / N);
    }

    reflex solving {solve evol method : rk4 step : hKR4 ;}
```

```
}

species I_agt {
    float t ;
    float Isize ; // number of infected
    float beta ;

    equation evol simultaneously : [ S_agt ] {
        diff(Isize, t) = (beta * first(S_agt).Ssize * Isize / N);
    }
}
```

The interest is that the modeler can create several agents for each compartment, which different values. For example in the SI model, the modeler can choose to create 1 agent `S_agt` and 2 agents `I_agt`. The **`beta`** attribute will have different values in the two agents, in order to represent 2 different strains.

```
global {
    int number_S <- 495 ; // The number of susceptible
    int number_I <- 5   ; // The number of infected
    int nb_I <- 2;
    float gbeta  <- 0.3  ; // The parameter Beta

    int N <- number_S + nb_I * number_I ;
    float hKR4 <- 0.1 ;

    init {
        create S_agt {
            Ssize <- float(number_S) ;
        }
        create I_agt number: nb_I {
            Isize <- float(number_I) ;
            self.beta <- myself.gbeta + rnd(0.5) ;
        }
    }
}
```

The results are computed using the RK4 method with:

- number_S = 495
- number_I = 5

- nb_I = 2
- gbeta = 0.3

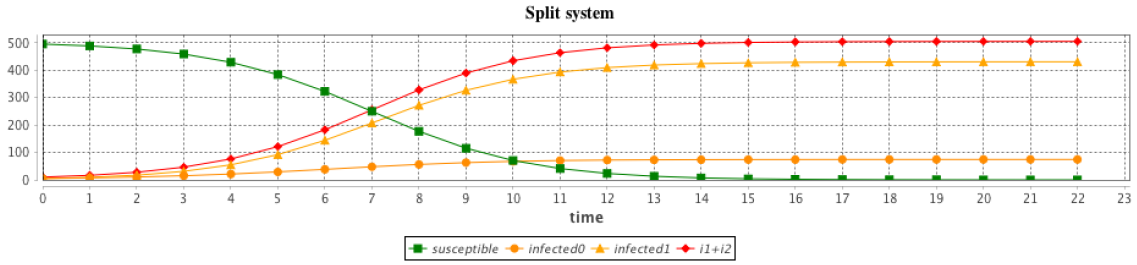


Figure 66.4: SI-split-results.png

- hKR4 = 0.1

`solve` an equation

The `solve` statement has been added in order to integrate numerically the equation system. It should be added into a reflex. At each simulation step, a step of the integration is executed, the length of the integration step is defined in the **`step`** facet. The `solve` statement will update the variables used in the equation system. The chosen integration method (defined in **`method`**) is Runge-Kutta 4 (which is very often a good choice of integration method in terms of accuracy).

```
reflex solving {
    solve eqSI method:rk4 step:h;
}
```

With a smaller integration step, the integration will be faster but less accurate.

More details

Details about the `solve` statement

The `solve` statement can have a huge set of facets (see [S_Statements#solve] for more details). The basic use of the `solve` statement requiers only the equation identifier. By default, the integration method is Runge-Kutta 4 with an integration step of `1`, which means that at each simulation step the equation integration is made over 1 unit of time (which is implicitly defined by the system parameter value).

```
solve eqSI ;
```

2 integration methods can be used:

- **method**: rk4 will use the Runge-Kutta 4 integration method
- **method**: dp853 will use the Dorman-Prince 8(5,3) integration method. The advantage of this method compared to Runge-Kutta is that it has an evaluation of the error and can use it to adapt the integration step size.

In order to synchronize the simulation step and the equation integration step, 2 facets can be used:

- **step**: **number**
- **cycle_length**: **number**

cycle_length (int): length of simulation cycle which will be synchronized with step of the integrator (default value: 1) step (float): integration step, use with most integrator methods (default value: 1)

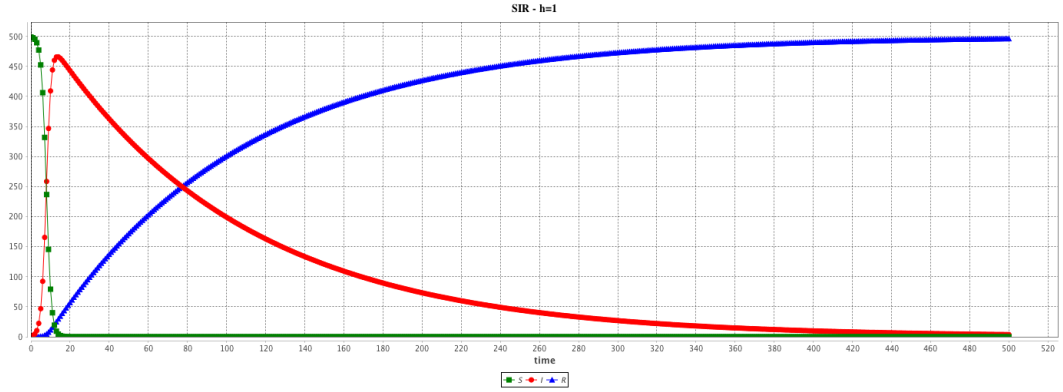
time_final (float): target time for the integration (can be set to a value smaller than t0 for backward integration) time_initial (float): initial time discretizing_step (int): number of discret beside 2 step of simulation (default value: 0) integrated_times (list): time interval inside integration process integrated_values (list): list of variables's value inside integration process

Some facets are specific to the DP853 integration methods: **max_step**, **min_step**, **scalAbsoluteTolerance** and **scalRelativeTolerance**.

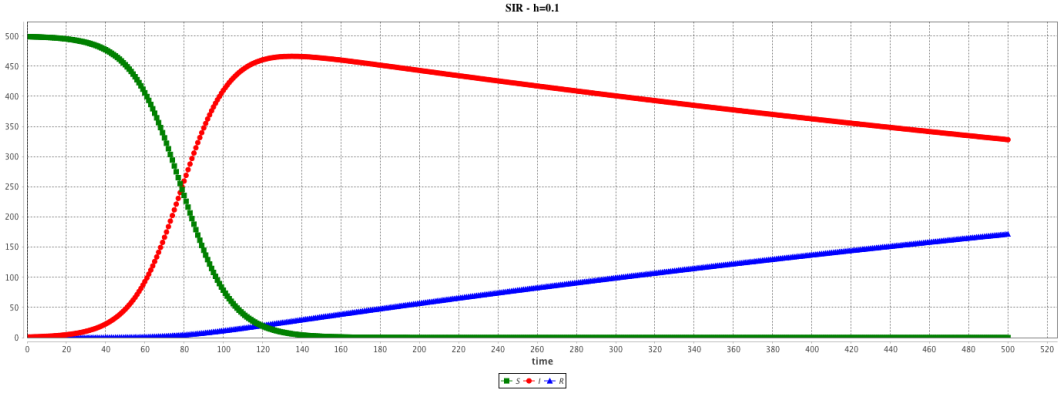
Example of the influence of the integration step

The **step** and **cycle_length** facets of the integration method may have a huge influence on the results. **step** has an impact on the result accuracy. In addition, it is possible to synchronize the step of the (agent-based) simulation and the (equation) integration step in various ways (depending on the modeler purpose) using the **cycle_length** facet: e.g. **cycle_length**: 10 means that 10 simulation steps are equivalent to 1 unit of time of the integration method.

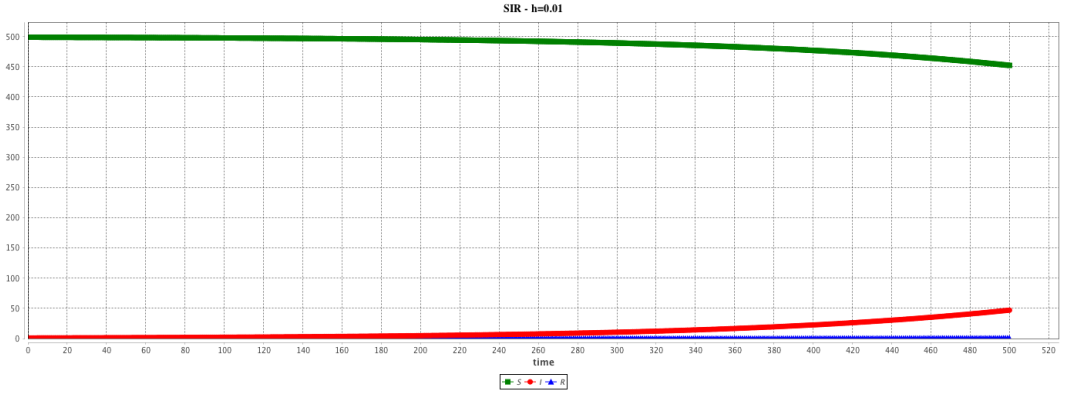
- `solve SIR method: "rk4" step: 1.0 cycle_length: 1.0 ;`



- `solve SIR method: "rk4" step: 0.1 cycle_length: 10.0 ;`



- `solve SIR method: "rk4" step: 0.01 cycle_length: 100.0 ;`



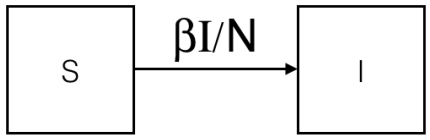


Figure 66.5: SI-compartment.png

$$\begin{cases} \frac{dS}{dt} = -\frac{\beta IS}{N} \\ \frac{dI}{dt} = \frac{\beta IS}{N} \end{cases}$$

Figure 66.6: SI-equations.png

List of built-in ODE systems

Several built-in equations have been defined.

`equation eqBuiltInSI type: SI vars: [S,I,t] params: [N,beta];`

This system is equivalent to:

```
equation eqSI {
    diff(S,t) = -beta * S * I / N ;
    diff(I,t) = beta * S * I / N ;
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- S = 499
- I = 1
- beta = 0.4
- h = 0.1

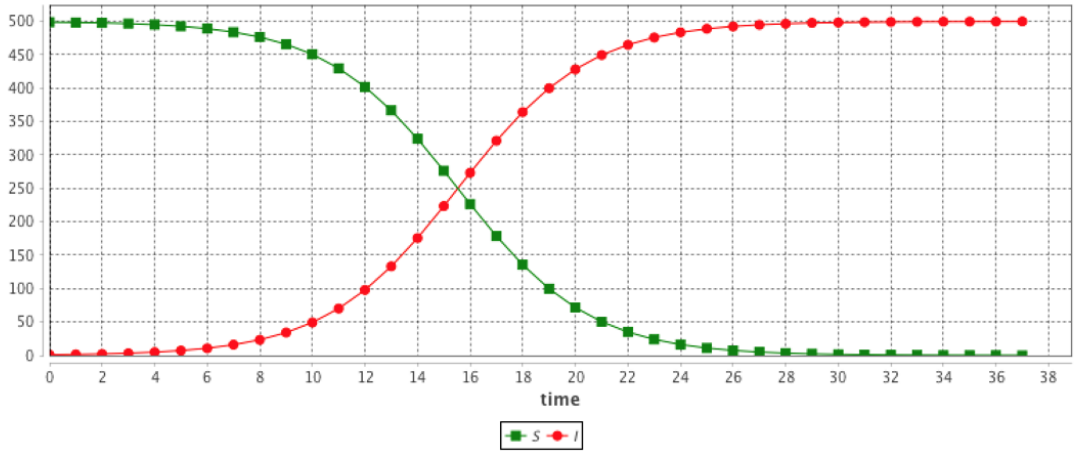


Figure 66.7: SI-result.png

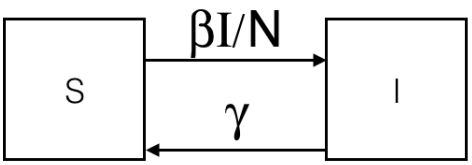


Figure 66.8: SIS-compartment.png

```
equation eqSIS type: SIS vars: [S,I,t] params:
[N,beta,gamma];
```

This system is equivalent to:

```
equation eqSIS {
    diff(S,t) = -beta * S * I / N + gamma * I;
    diff(I,t) = beta * S * I / N - gamma * I;
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- S = 499
- I = 1
- beta = 0.4

$$\begin{cases} \frac{dS}{dt} = -\frac{\beta IS}{N} + \gamma I \\ \frac{dI}{dt} = \frac{\beta IS}{N} - \gamma I \end{cases}$$

Figure 66.9: SIS-equations.png

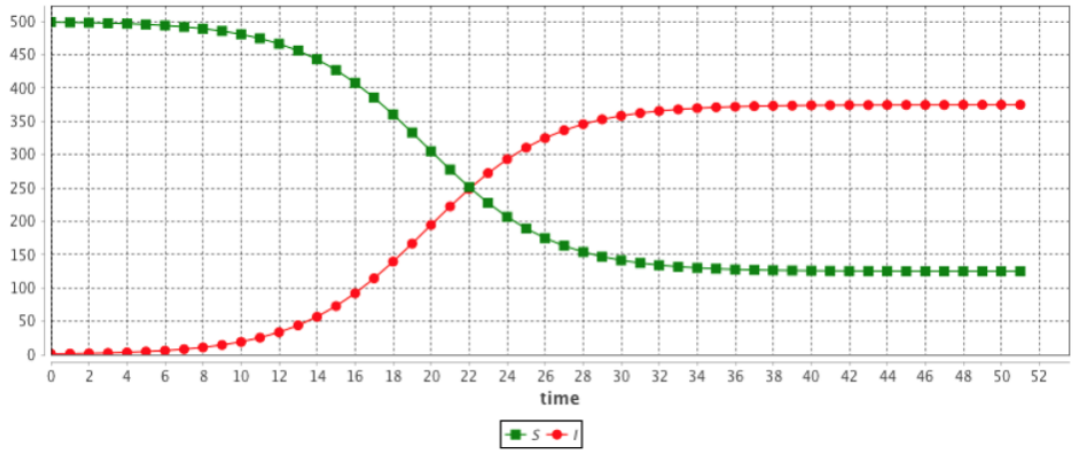


Figure 66.10: SIS-result.png

- gamma = 0.1
- h = 0.1

```
equation eqSIR type:SIR vars:[S,I,R,t] params:[N,beta,gamma]
;
```

This system is equivalent to:

```
equation eqSIR {
    diff(S,t) = (- beta * S * I / N);
    diff(I,t) = (beta * S * I / N) - (gamma * I);
    diff(R,t) = (gamma * I);
```

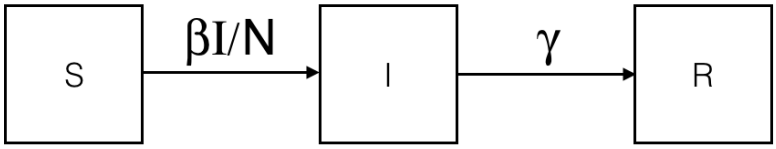


Figure 66.11: SIR-compartment.png

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N} \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I \\ \dfrac{dR}{dt} = \gamma I \end{cases}$$

Figure 66.12: SIR-equations.png

```
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- S = 499
- I = 1
- R = 0
- beta = 0.4
- gamma = 0.1
- h = 0.1

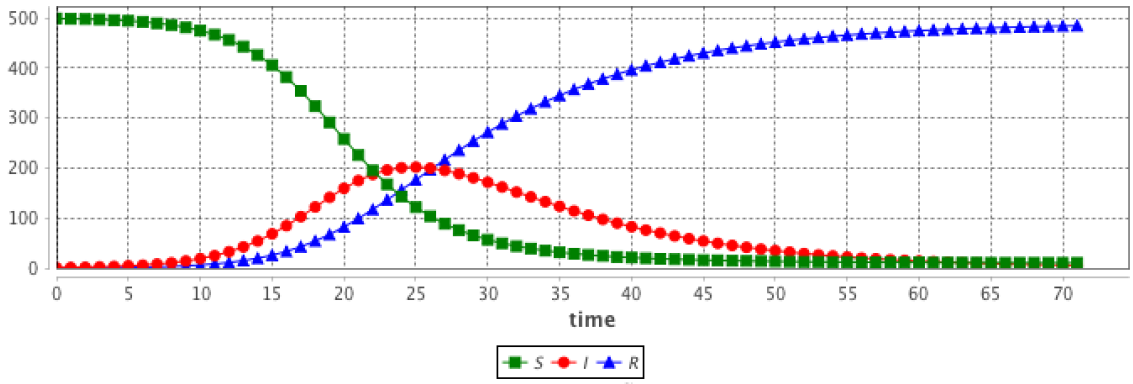


Figure 66.13: SIR-result.png

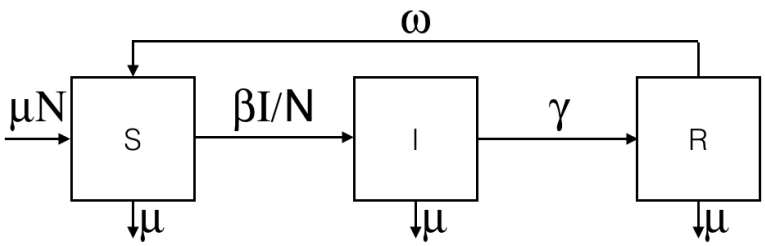


Figure 66.14: SIRS-compartment.png

```
equation eqSIRS type: SIRS vars: [S,I,R,t] params:
[N,beta,gamma,omega,mu] ;
```

This system is equivalent to:

```
equation eqSIRS {
     diff(S,t) = mu * N + omega * R + - beta * S * I / N - mu * S ;
     diff(I,t) = beta * S * I / N - gamma * I - mu * I ;
     diff(R,t) = gamma * I - omega * R - mu * R ;
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- S = 499
- I = 1
- R = 0

$$\begin{cases} \dfrac{dS}{dt} = \mu N + \omega R - \dfrac{\beta IS}{N} - \mu S \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I - \mu I \\ \dfrac{dR}{dt} = \gamma I - \omega R - \mu R \end{cases}$$

Figure 66.15: SIRS-equations.png

- beta = 0.4
- gamma = 0.01
- omega = 0.05
- mu = 0.01
- h = 0.1

equation eqSEIR type: SEIR vars: [S,E,I,R,t] params: [N,beta,gamma,sigma,mu] ;

This system is equivalent to:

```
equation eqSEIR {
    diff(S,t) = mu * N - beta * S * I / N - mu * S ;
    diff(E,t) = beta * S * I / N - mu * E - sigma * E ;
    diff(I,t) = sigma * E - mu * I - gamma * I;
    diff(R,t) = gamma * I - mu * R ;
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- S = 499
- E = 0
- I = 1

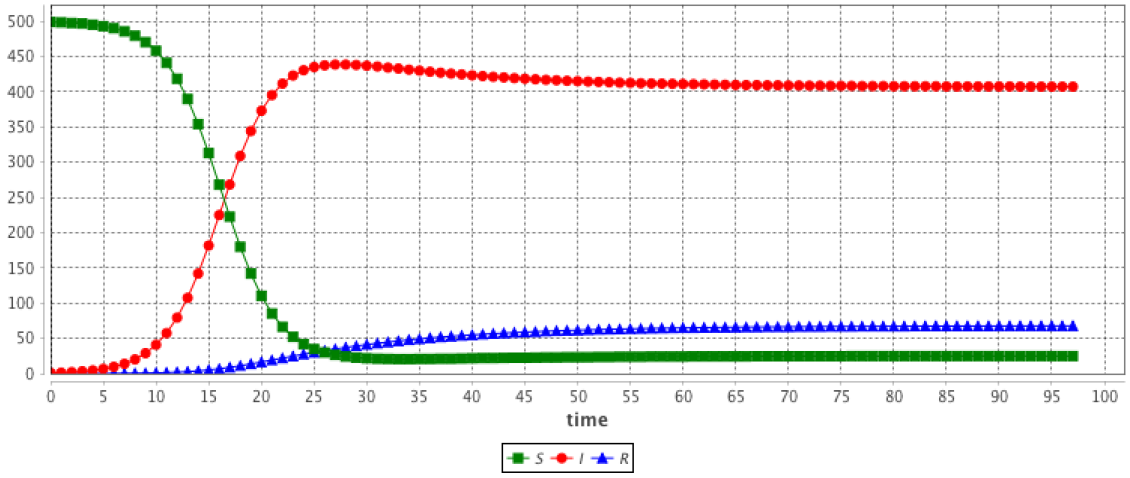


Figure 66.16: SIRS-result.png

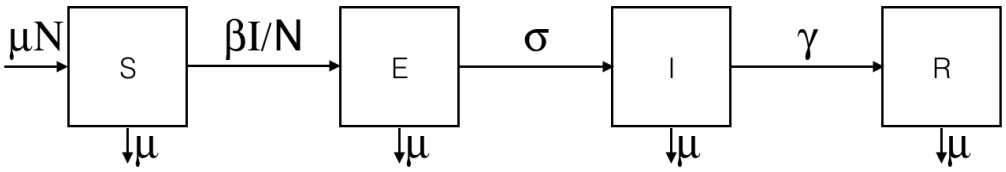


Figure 66.17: SEIR-compartment.png

$$\begin{cases} \frac{dS}{dt} = \mu N - \frac{\beta IS}{N} - \mu S \\ \frac{dE}{dt} = \frac{\beta IS}{N} - \sigma E - \mu E \\ \frac{dI}{dt} = \sigma E - \gamma I - \mu I \\ \frac{dR}{dt} = \gamma I - \mu R \end{cases}$$

Figure 66.18: SEIR-equations.png

- R = 0
- beta = 0.4
- gamma = 0.01
- sigma = 0.05
- mu = 0.01
- h = 0.1

```
equation eqLV type: LV vars: [x,y,t] params:
[alpha,beta,delta,gamma] ;
```

This system is equivalent to:

```
equation eqLV {
    diff(x,t) =   x * (alpha - beta * y);
    diff(y,t) = - y * (delta - gamma * x);
}
```

The results are provided using the Runge-Kutta 4 method using following initial values:

- x = 2

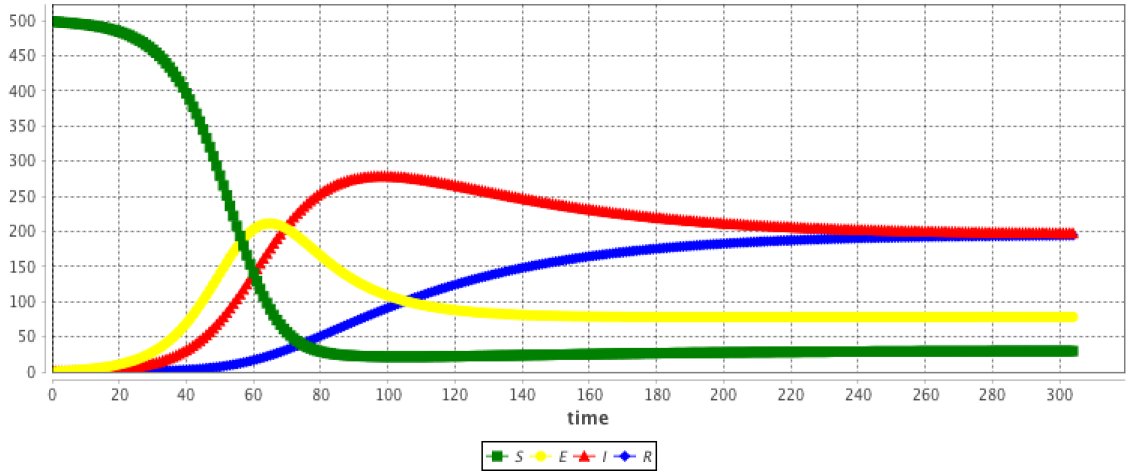


Figure 66.19: SEIR-result.png

$$\begin{cases} \dfrac{dx}{dt} = x*(\alpha - \beta y) \\ \dfrac{dy}{dt} = -y*(\delta - \gamma x) \end{cases}$$

Figure 66.20: LV-equations.png

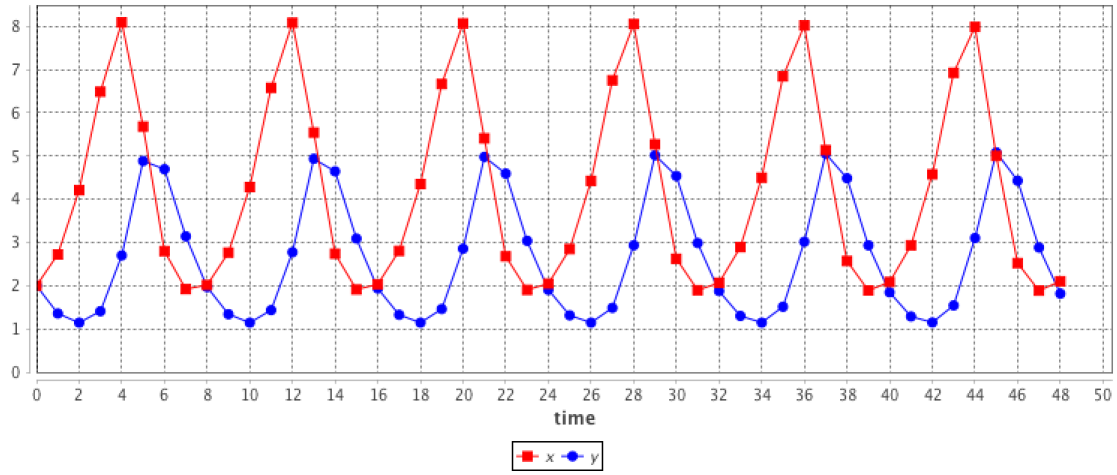


Figure 66.21: LV-result.png

- y = 2
- alpha = 0.8
- beta = 0.3
- gamma = 0.2
- delta = 0.85
- h = 0.1

Part VII

Recipes

Chapter 67

Recipes

Understanding the structure of models in GAML and gaining some insight of the language is required, but is usually not sufficient to build correct models or models that need to deal with specific approaches (like equation-based modeling). This section is intended to provide readers with practical "how to"s on various subjects, ranging from the use of database access to the design of agent communication languages. It is by no means exhaustive, and will progressively be extended with more "recipes" in the future, depending on the concrete questions asked by users.

Chapter 68

Manipulate OSM Datas

This section will be presented as a quick tutorial, showing how to proceed to manipulate OSM (Open street map) datas, and load them into GAMA. We will use the software QGIS to change the attributes of the OSM file.

From the website openstreetmap.org, we will chose a place (in this example, we will take a neighborhood in New York City). Directly from the website, you can export the chosen area in the osm format.

We have now to manipulate the attributes for the exported osm file. Several software are possible to use, but we will focus on QGIS, which is totally free and provides a lot of possibilities in term of manipulation of data.

Once you have installed correctly QGIS, launch QGIS Desktop, and start to import the topology from the osm file.

A message indicates that the import was successful. An output file .osm.db is created. You have now to export the topology to SpatiaLite.

Specify the path for your DataBase file, then choose the export type (in your case, we will choose the type "Polygons (closed ways)"), choose an output layer name. If you want to use the open street maps attributes values, click on "Load from DB", and select the attributes you want to keep. Click OK then.

A message indicates that the export was successful, and you have now a new layer created.

We will now manipulate the attributes of your datafile. Right click on the layer, and select "Open Attribute Table".

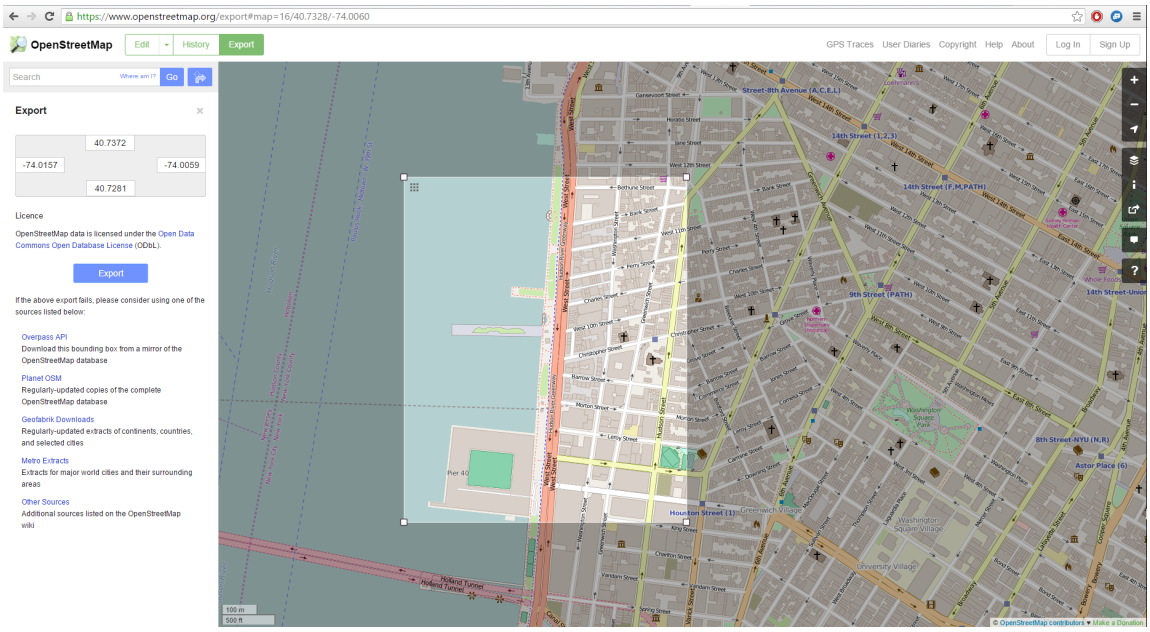


Figure 68.1: images/manipulate_OSM_file_1.png

The table of attribute appears. Select the little pencil on the top-left corner of the window to modify the table.

We will add an attribute manually. Click on the button “new column”, choose a name and a type (we will choose the type “text”).

A new column appears at the end of the table. Let’s fill some values (for instance blue / red). Once you finishes, click on the “save edit” button.

Our file is now ready to be exported. Right click on the layer, and click on “Save As”.

Choose “shapefile” as format, choose a save path and click ok.

Copy passed all the .shp created in the include folder of your GAMA project. You are now ready to write the model.

```
model HowToUseOpenStreetMap

global {
    // Global variables related to the Management units
    file shapeFile <- file('../includes/new_york.shp');

    //definition of the environment size from the shapefile.
```

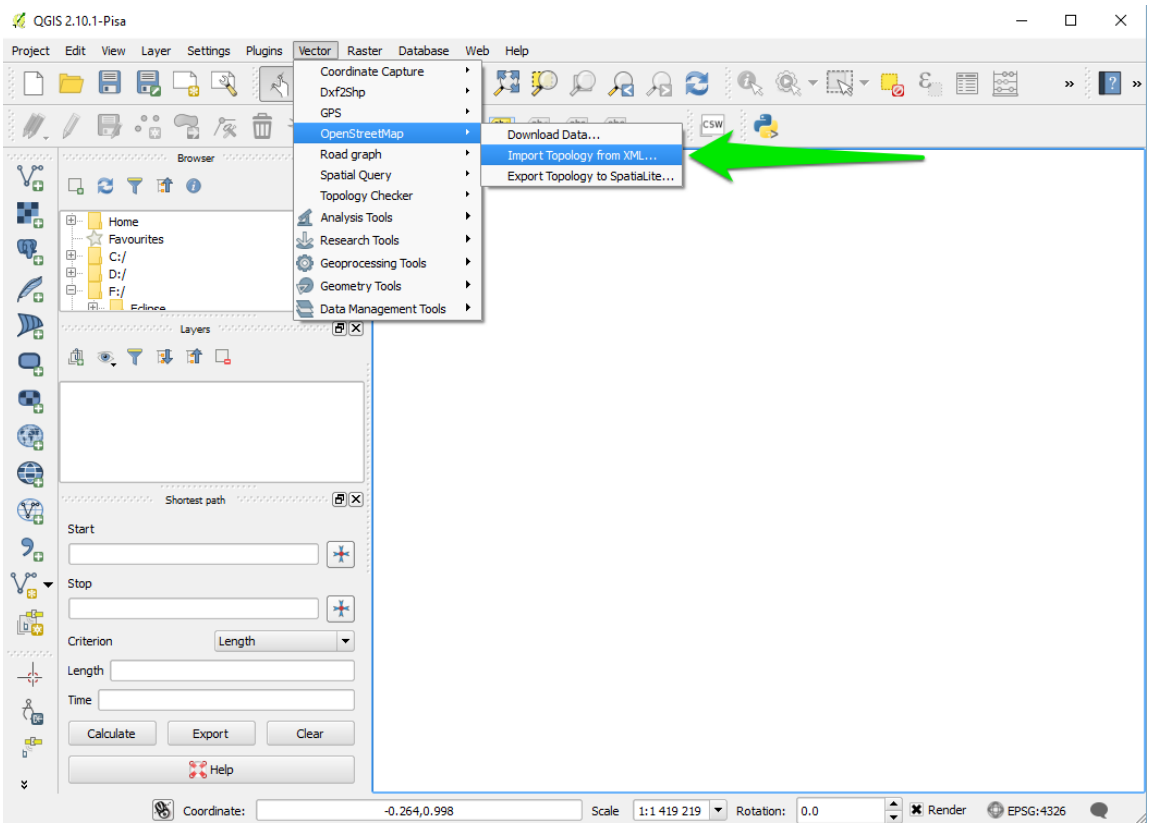


Figure 68.2: images/manipulate_OSM_file_2.png

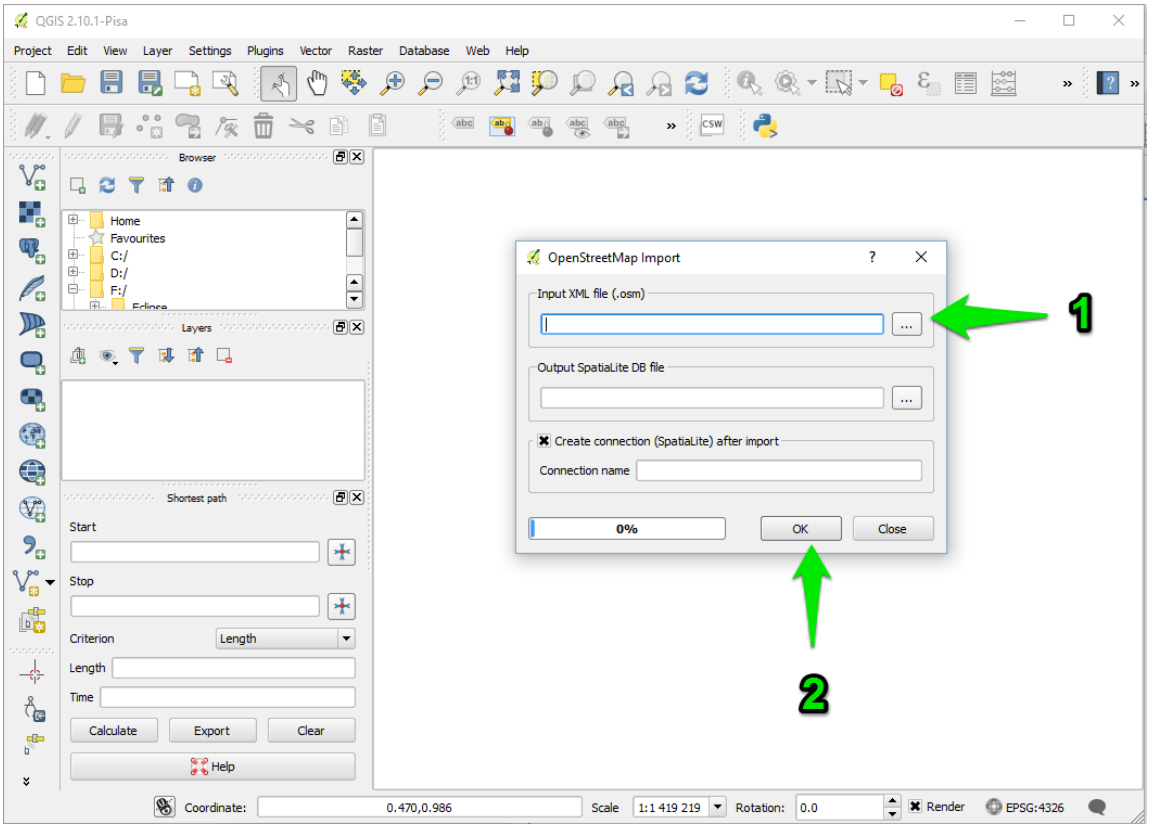


Figure 68.3: images/manipulate_OSM_file_3.png

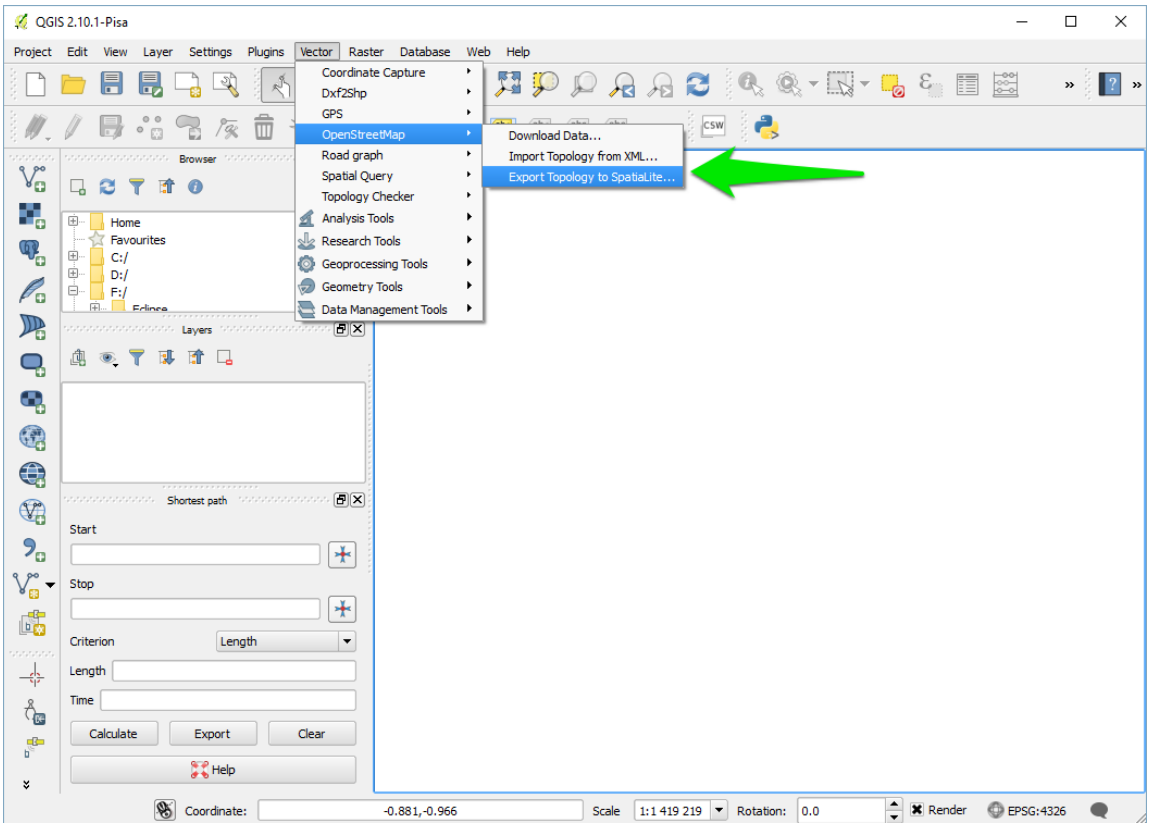


Figure 68.4: images/manipulate_OSM_file_4.png

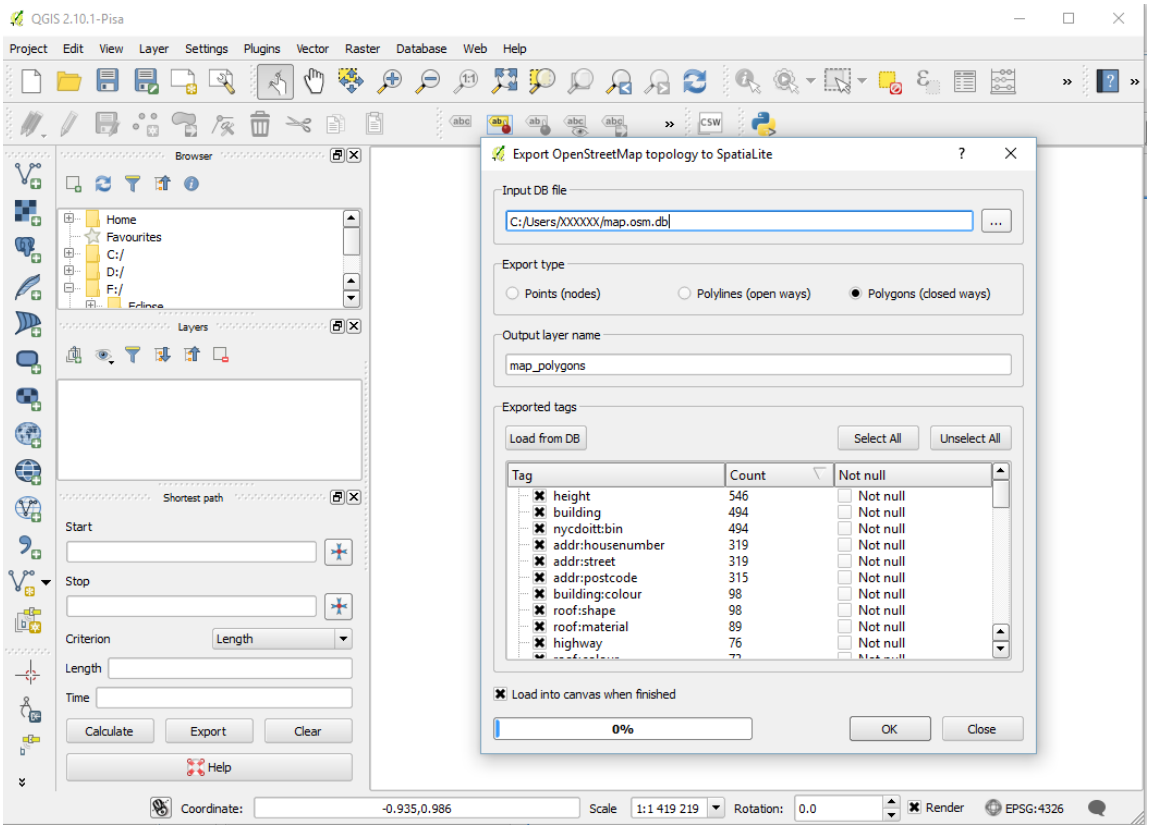


Figure 68.5: images/manipulate_OSM_file_5.png

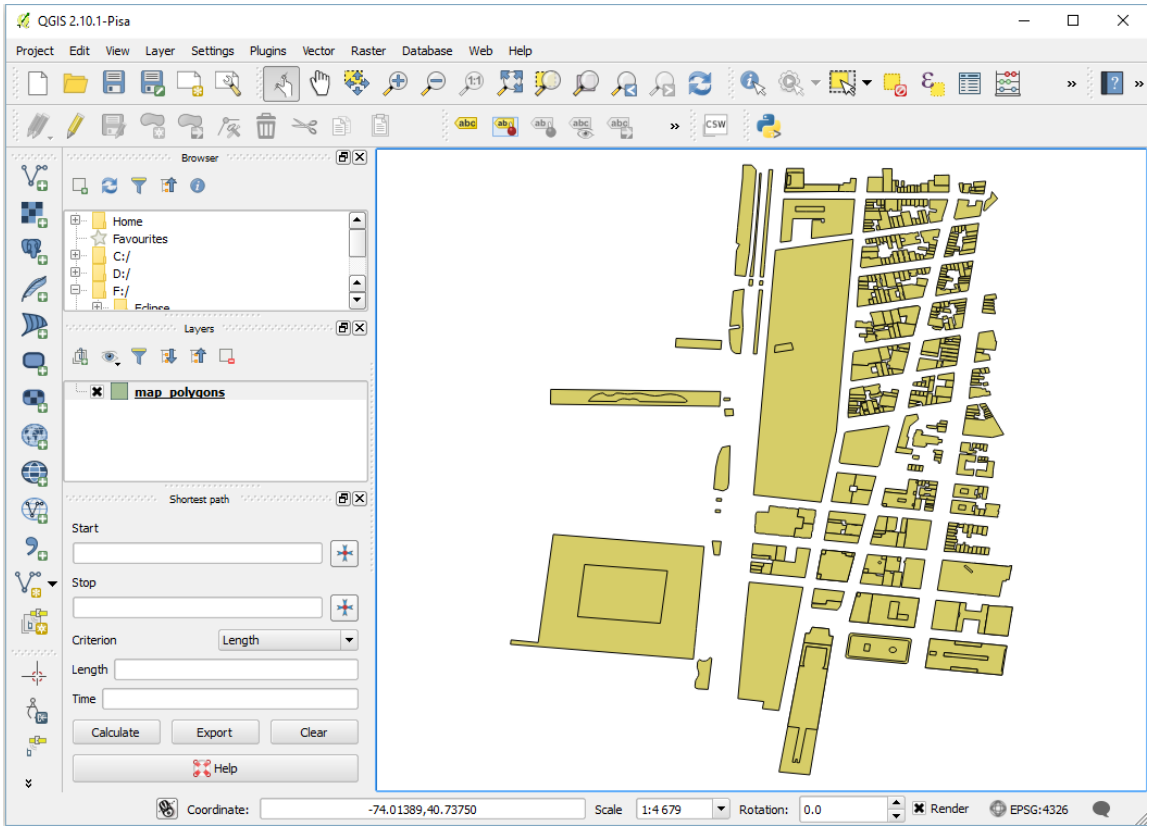


Figure 68.6: images/manipulate_OSM_file_6.png

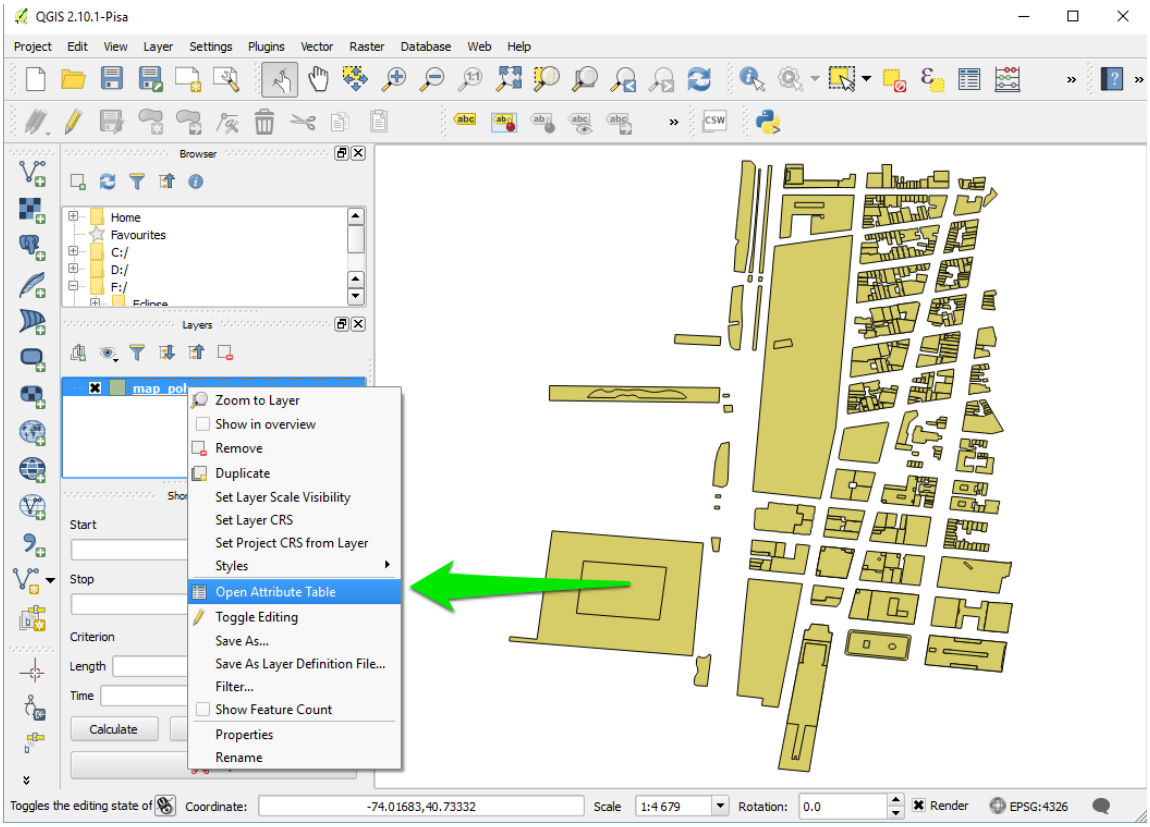


Figure 68.7: images/manipulate_OSM_file_7.png

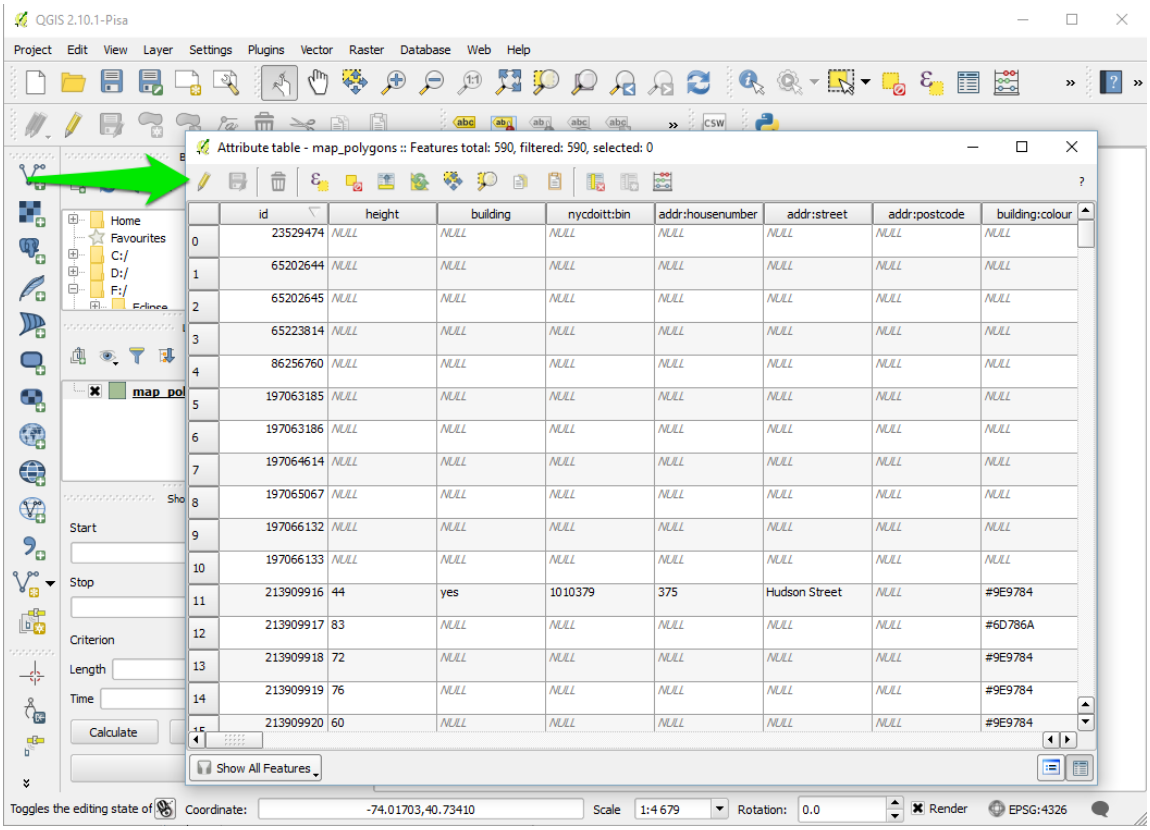


Figure 68.8: images/manipulate_OSM_file_8.png

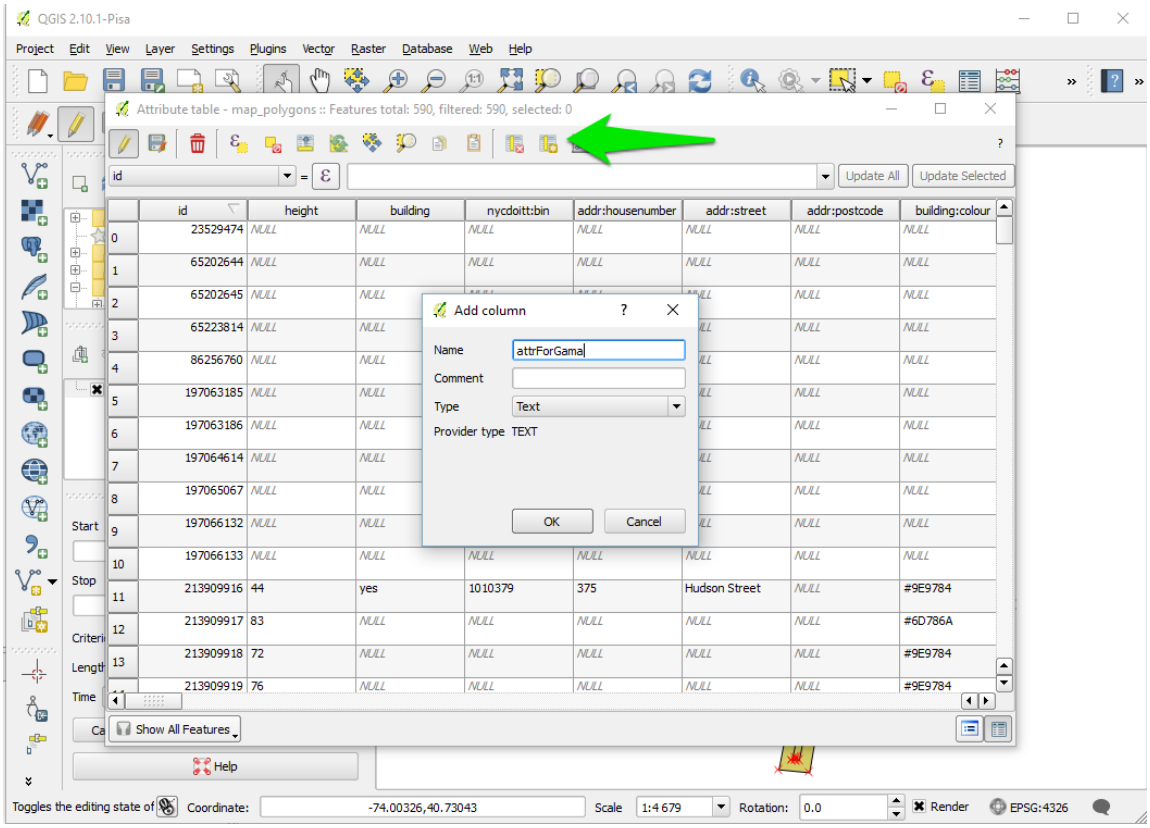


Figure 68.9: images/manipulate_OSM_file_9.png

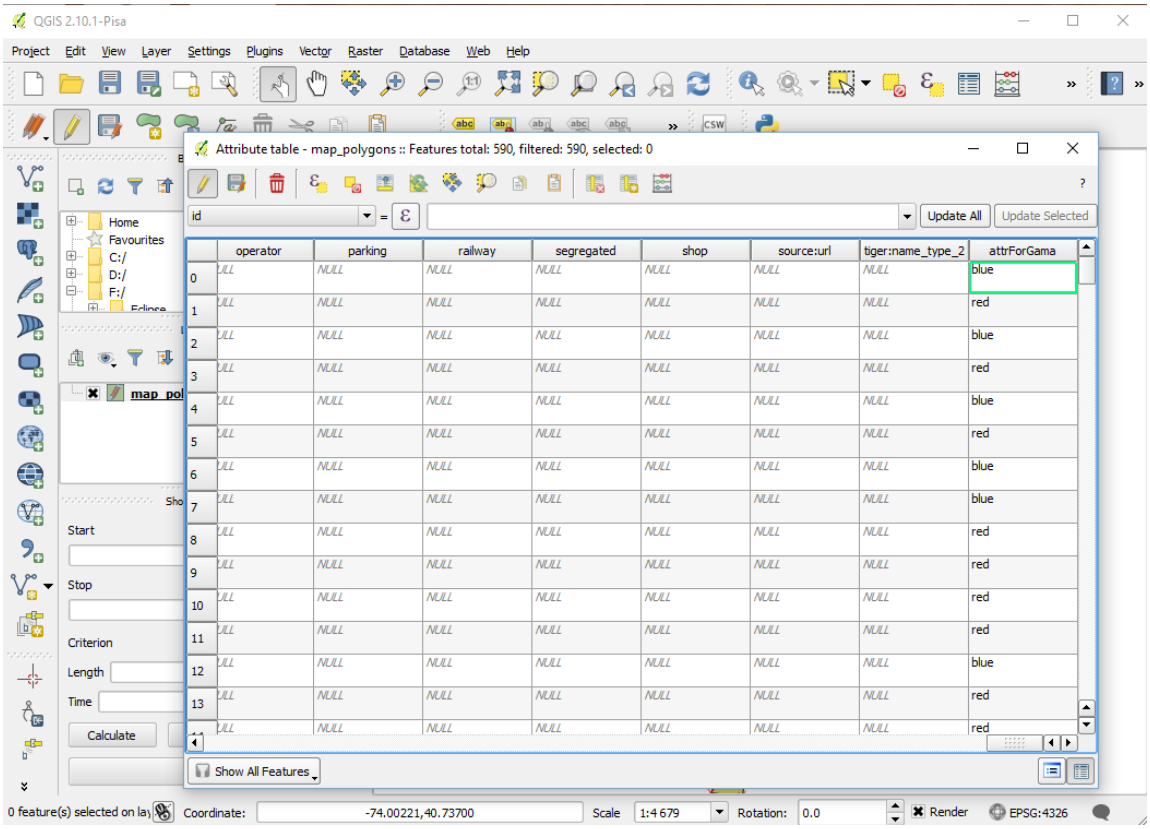


Figure 68.10: images/manipulate_OSM_file_10.png

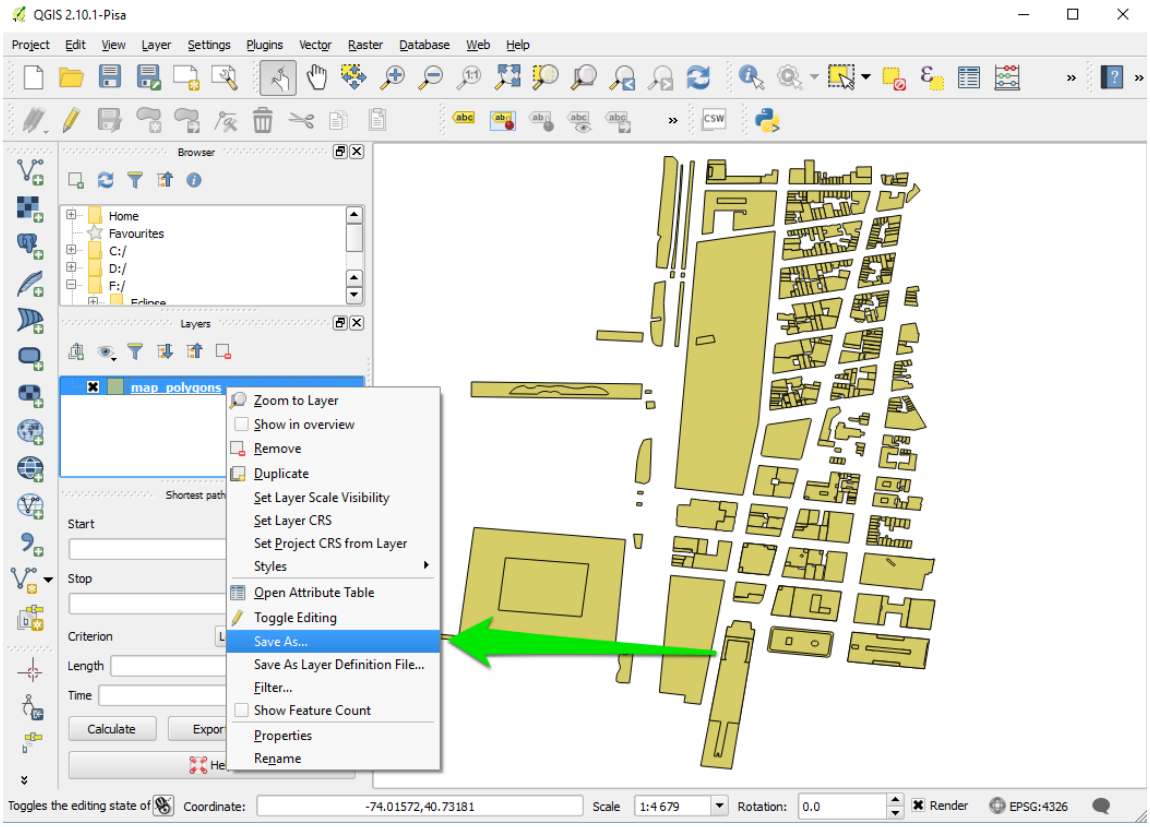


Figure 68.11: images/manipulate_OSM_file_11.png

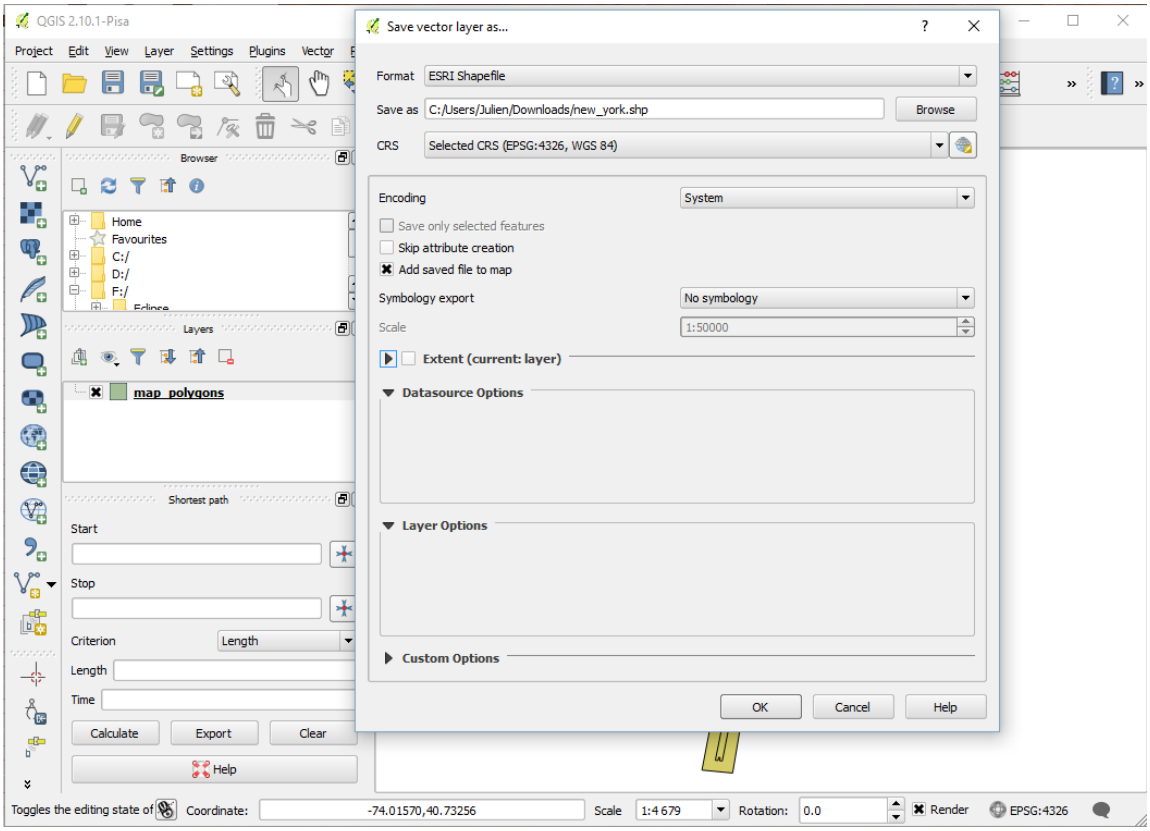


Figure 68.12: images/manipulate_OSM_file_12.png

```
    //Note that is possible to define it from several files by using:
    geometry shape <- envelope(envelope(file1) + envelope(file2) + ...);
    geometry shape <- envelope(shapeFile);

    init {
        //Creation of elementOfNewYork agents from the shapefile (and
    reading some of the shapefile attributes)
        create elementOfNewYork from: shapeFile
            with: [elementId::int(read('id')), elementHeight::int(read
    ('height')), elementColor::string(read('attrForGama'))] ;
    }
}

species elementOfNewYork{
    int elementId;
    int elementHeight;
    string elementColor;

    aspect basic{
        draw shape color: (elementColor = "blue") ? #blue : ( (
    elementColor = "red") ? #red : #yellow ) depth: elementHeight;
    }
}

experiment main type: gui {
    output {
        display HowToUseOpenStreetMap type:opengl {
            species elementOfNewYork aspect: basic;
        }
    }
}
```

Here is the result, with a special colorization of the different elements regarding to the value of the attribute “attrForGama”, and an elevation regarding to the value of the attribute “height”.

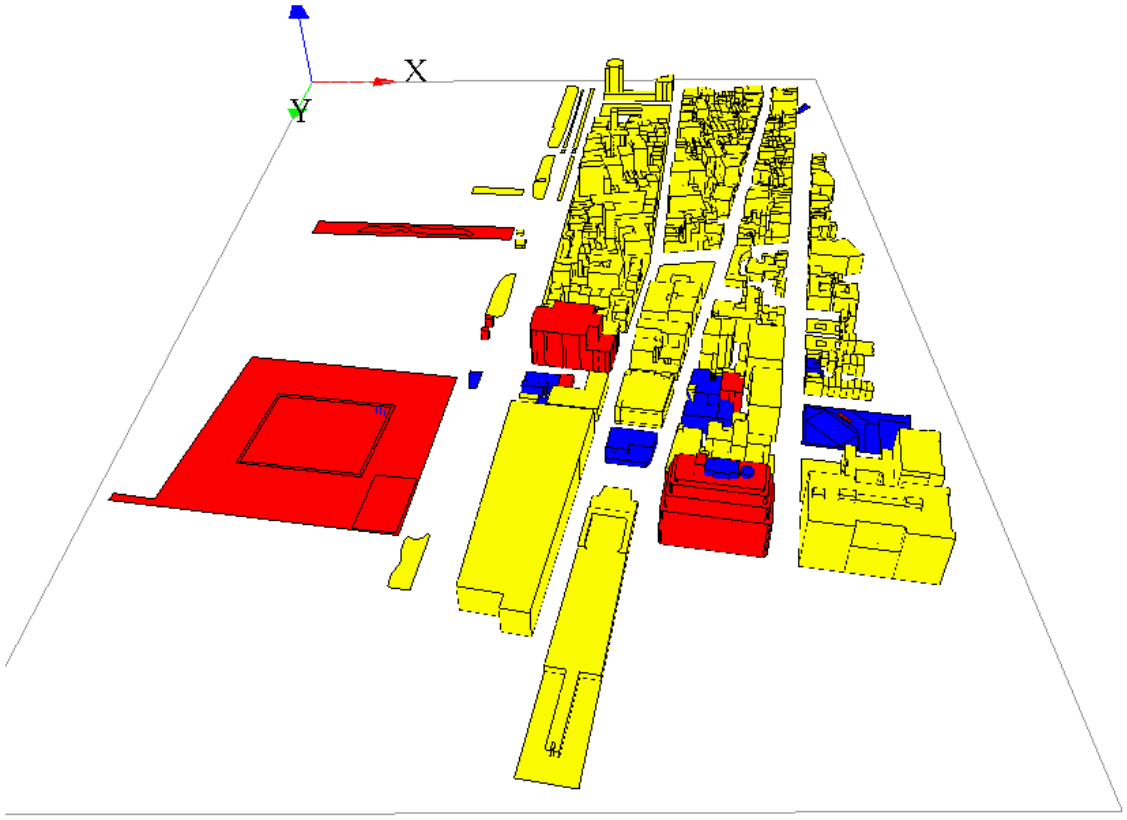


Figure 68.13: images/manipulate_OSM_file_13.png

Chapter 69

Implementing diffusion

GAMA provides you the possibility to represent and simulate the diffusion of a variable through a grid topology.

Index

Diffuse statement

The statement to use for the diffusion is `diffuse`. It has to be used in a `grid` species. The `diffuse` uses the following facets:

- **`var`** (an identifier), (omissible) : the variable to be diffused

- **`on`** (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- **`avoid_mask`** (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the variation value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- **`cycle_length`** (int): the number of diffusion operation applied in one simulation step
- **`mask`** (matrix): a matrix masking the diffusion (matrix created from a image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- **`matrix`** (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- **`method`** (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- **`min_value`** (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- **`propagation`** (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : S' = (S / N / proportion) - variation. The intensity of S is then diminished by S * proportion on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals 'gradient', the original intensity is not modified, and each neighbors receives the intensity : S / proportion - variation. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to

'diffusion'.

- **proportion** (float): a diffusion rate
- **radius** (int): a diffusion radius (in number of cells from the center)
- **variation** (float): an absolute value to decrease at each neighbors

To write a diffusion, you first have to declare a grid, and declare a special attribute for the diffusion. You will then have to write the `diffuse` statement in an other scope (such as the `global` scope for instance), which will permit the values to be diffused at each step. There, you will specify which variable you want to diffuse (through the **var** facet), on which species or list of agents you want the diffusion (through the **on** facet), and how you want this value to be diffused (through all the other facets, we will see how it works with matrix and with special parameters just after).

Here is the template of code we will use for the next following part of this page:

```
global {
    int size <- 64; // the size has to be a power of 2.
    cells selected_cells;

    // Initialize the emiter cell as the cell at the center of the word
    init {
        selected_cells <- location as cells;
    }
    // Affecting "1" to each step
    reflex new_Value {
        ask(selected_cells){
            phero <- 1.0;
        }
    }

    reflex diff {
        // Declare a diffusion on the grid "cells" and on "quick_cells
   ". The diffusion declared on "quick_cells" will make 10 computations
    at each step to accelerate the process.
        // The value of the diffusion will be store in the new variable
    "phero" of the cell.
        diffuse var: phero on: cells /*HERE WRITE DOWN THE DIFFUSION
   PROPERTIES*/;
    }
}


grid cells height: size width: size {
    // "phero" is the variable storing the value of the diffusion
```

```
    float phero  <- 0.0;
    // The color of the cell is linked to the value of "phero".
    rgb color <- hsb(phero,1.0,1.0) update: hsb(phero,1.0,1.0);
}


experiment diffusion type: gui {
    output {
        display a type: opengl {
            // Display the grid with elevation
            grid cells elevation: phero * 10 triangulation: true;
        }
    }
}
```

This model will simulate a diffusion through a grid at each step, affecting 1 to the center cell diffusing variable value. The diffusion will be seen during the simulation through a color code, and through the elevation of the cell.

Diffusion with matrix

A first way of specifying the behavior of your diffusion is using diffusion matrix. A diffusion matrix is a 2 dimension matrix `[n][m]` with **`float`** values, where both `n` and `m` have to be **pair values**. The most often, diffusion matrix are square matrix, but you can also declare rectangular matrix.

Example of matrix:

```
matrix<float> mat_diff <- matrix([
        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);
```

In the `diffuse` statement, you than have to specify the matrix of diffusion you want in the facet **`matrix`**.

```
diffuse var: phero on: cells matrix:mat_diff;
```

Using the facet **`propagation`**, you can specify if you want the value to be propagated as a *diffusion* or as a *gratient*.

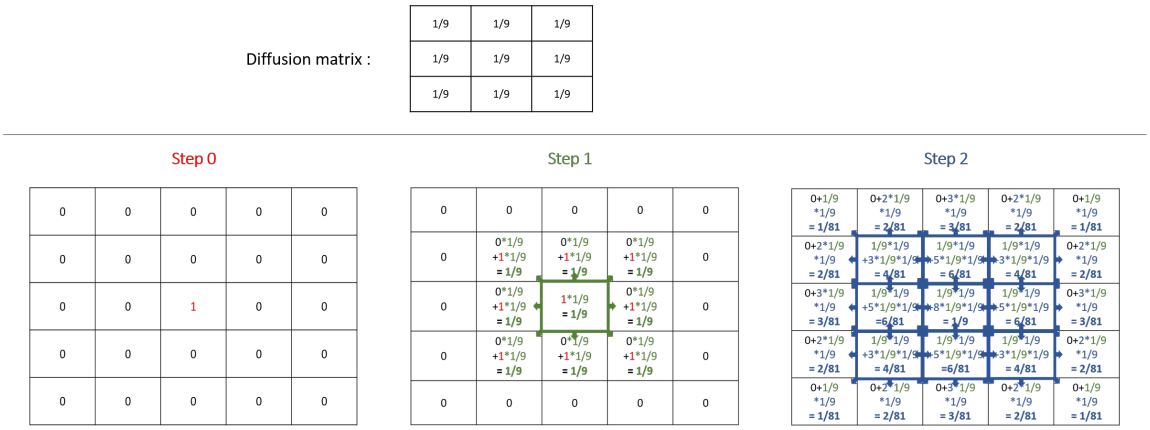


Figure 69.1: resources/images/recipes/diffusion_computation.png

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 :

Note that the sum of all the values diffused at the next step is equal to the sum of the values that will be diffused multiply by the sum of the values of the diffusion matrix. That means that if the sum of the values of your diffusion matrix is larger than 1, the values will increase exponentially at each step. The sum of the value of a diffusion matrix is usually equal to 1.

Here are some example of matrix you can use, played with the template model:

Gradient matrix

A `gradient` (use facet : **propagation**:`gradient`) is an other type of propagation. This time, only the larger value diffused will be chosen as the new one.

Note that unlike the *diffusion* propagation, the sum of your matrix can be greater than 1 (and it is the case, most often !).

Here are some example of matrix with gradient propagation:

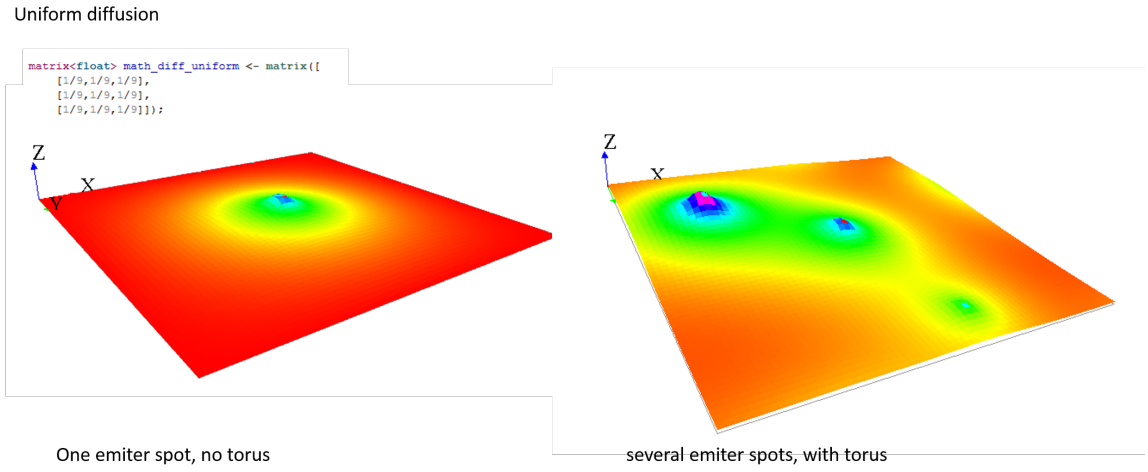


Figure 69.2: resources/images/recipes/uniform_diffusion.png

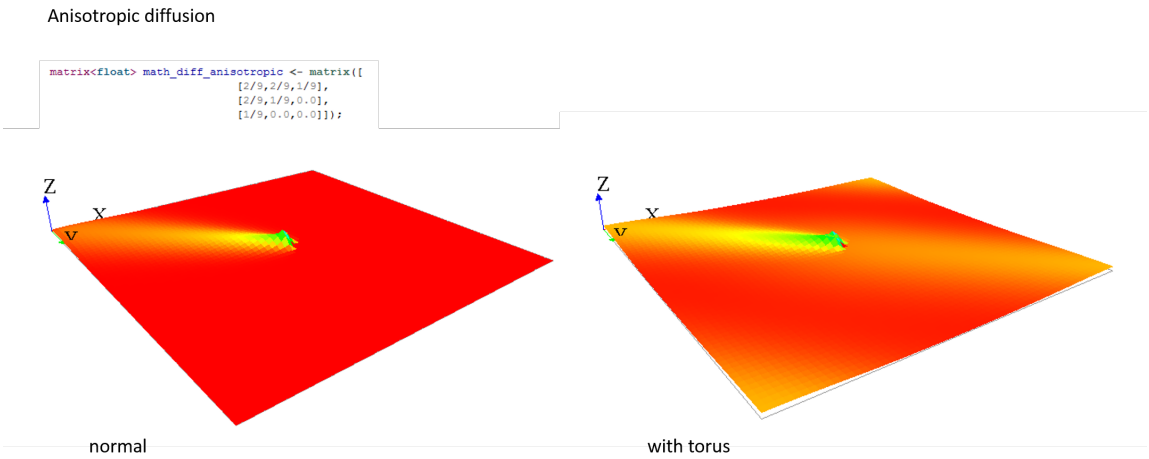


Figure 69.3: resources/images/recipes/anisotropic_diffusion.png

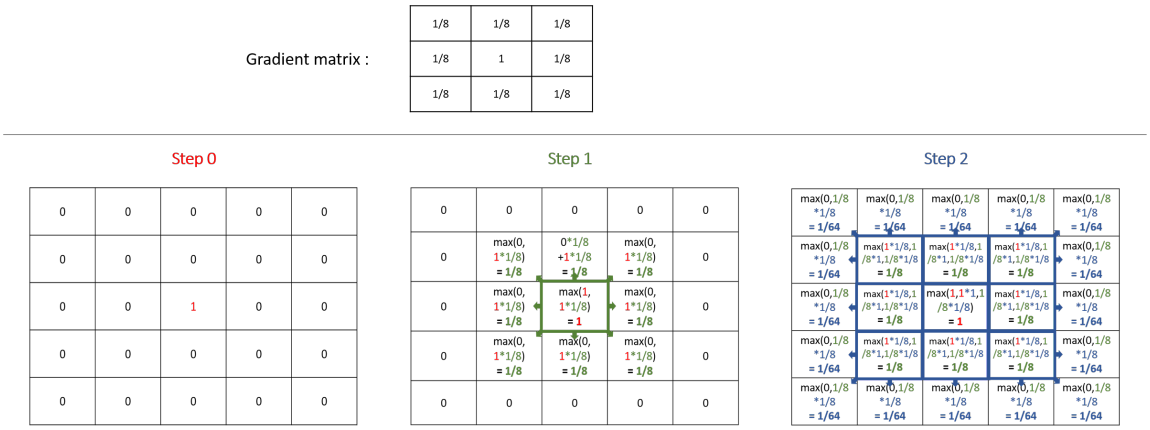


Figure 69.4: resources/images/recipes/gradient_computation.png

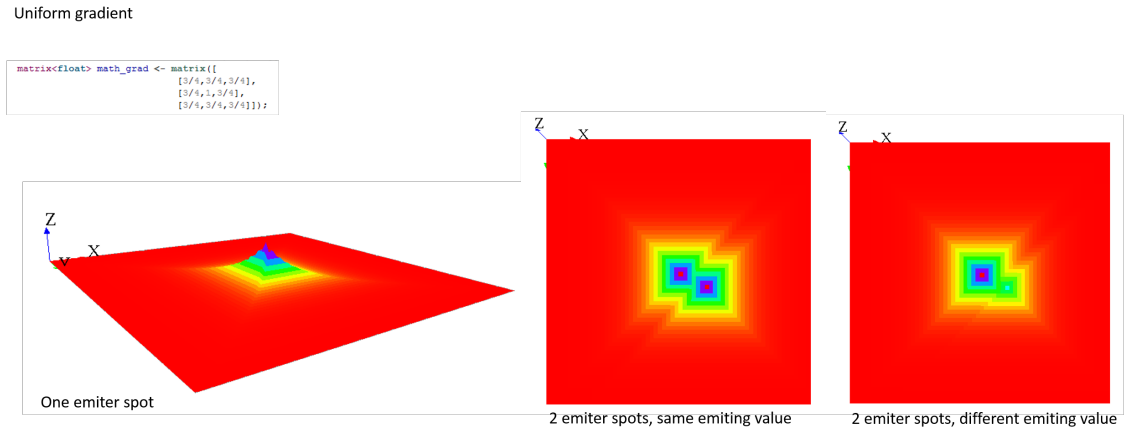


Figure 69.5: resources/images/recipes/uniform_gradient.png

Irregular gradient

```
matrix<float> math_grad <- matrix([
                           [2/4,3/4,3/4],
                           [1/4,1,1],
                           [2/4,3/4,3/4]]);
```

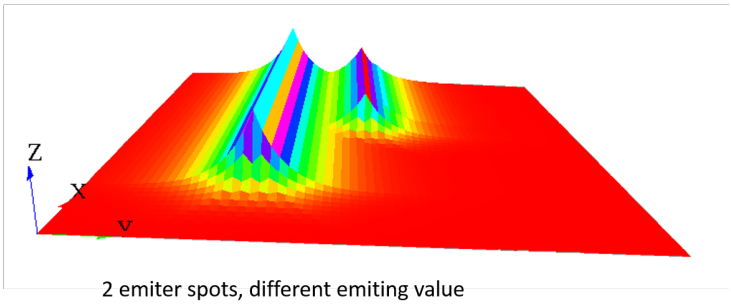


Figure 69.6: resources/images/recipes/irregular_gradient.png

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	3/81	2/81	1/81
2/81	3/81	6/81	3/81	2/81
3/81	6/81	1/9	6/81	3/81
2/81	3/81	6/81	3/81	2/81
1/81	2/81	3/81	2/81	1/81

Figure 69.7: resources/images/recipes/cycle_length.png

Compute multiple propagations at the same step

You can compute several times the propagation you want by using the facet **cycle_length**. GAMA will compute for you the corresponding new matrix, and will apply it.

Writing those two thinks are exactly equivalent (for diffusion):

```
    matrix<float> mat_diff <- matrix([
            [1/81,2/81,3/81,2/81,1/81],
            [2/81,4/81,6/81,4/81,2/81],
            [3/81,6/81,1/9,6/81,3/81],
            [2/81,4/81,6/81,4/81,2/81],
```

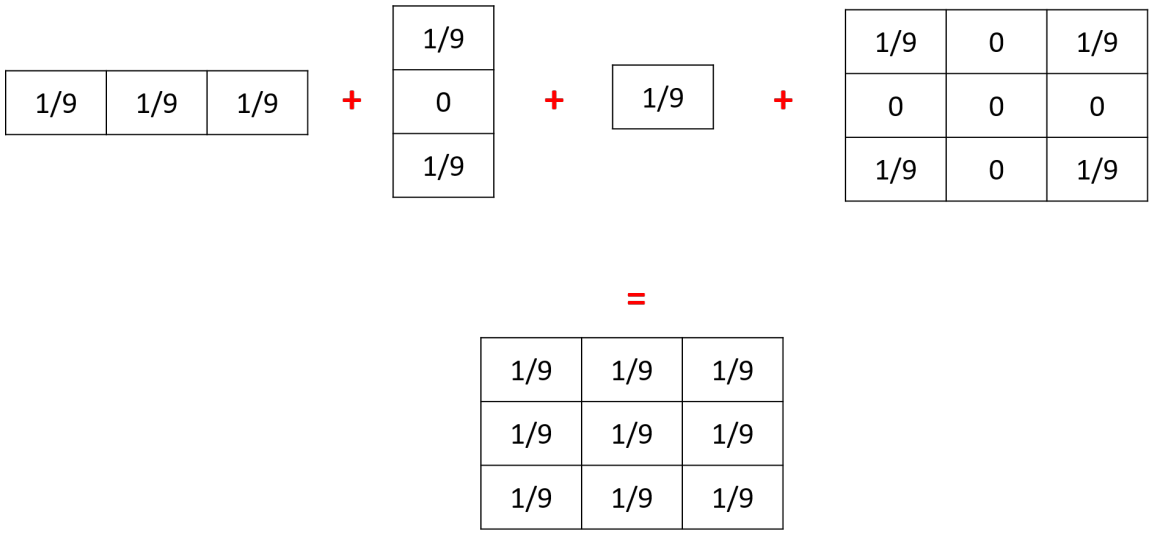


Figure 69.8: resources/images/recipes/addition_matrix.png

```
        [1/81,2/81,3/81,2/81,1/81]]);
reflex diff {
    diffuse var: phero on: cells matrix:mat_diff;
```

```
matrix<float> mat_diff <- matrix([
        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);
reflex diff {
    diffuse var: phero on: cells matrix:mat_diff cycle_length:2;
```

Executing several diffusion matrix

If you execute several times the statement `diffuse` with different matrix on the same variable, their values will be added (and centered if their dimension is not equal).

Thus, the following 3 matrix will be combined to create one unique matrix:

Diffusion with parameters

Sometimes writing diffusion matrix is not exactly what you want, and you may prefer to just give some parameters to compute the correct diffusion matrix. You can use the following facets in order to do that : **propagation**, **variation** and **radius**.

Depending on which **propagation** you choose, and how many neighbors your grid have, the propagation matrix will be compute differently. The propagation matrix will have the size : range*2+1.

Let's note **P** for the propagation value, **V** for the variation, **R** for the range and **N** for the number of neighbors.

- **With diffusion propagation**

For diffusion propagation, we compute following the following steps:

(1) We determine the "minimale" matrix according to N (if N = 8, the matrix will be `[[P/9,P/9,P/9][P/9,1/9,P/9][P/9,P/9,P/9]]`. if N = 4, the matrix will be `[[0,P/5,0][P/5,1/5,P/5][0,P/5,0]]`).

(2) If R != 1, we propagate the matrix R times to obtain a `[2*R+1][2*R+1]` matrix (same computation as for **cycle_length**).

(3) If V != 0, we substract each value by V*DistanceFromCenter (DistanceFromCenter depends on N).

Ex with the default values (P=1, R=1, V=0, N=8):

- **With gradient propagation**

The value of each cell will be equal to **P/POW(N,DistanceFromCenter)-DistanceFromCenter*V**. (DistanceFromCenter depends on N).

Ex with R=2, other parameters default values (R=2, P=1, V=0, N=8):

Note that if you declared a diffusion matrix, you cannot use those 3 facets (it will raise a warning). Note also that if you use parameters, you will only have uniform matrix.

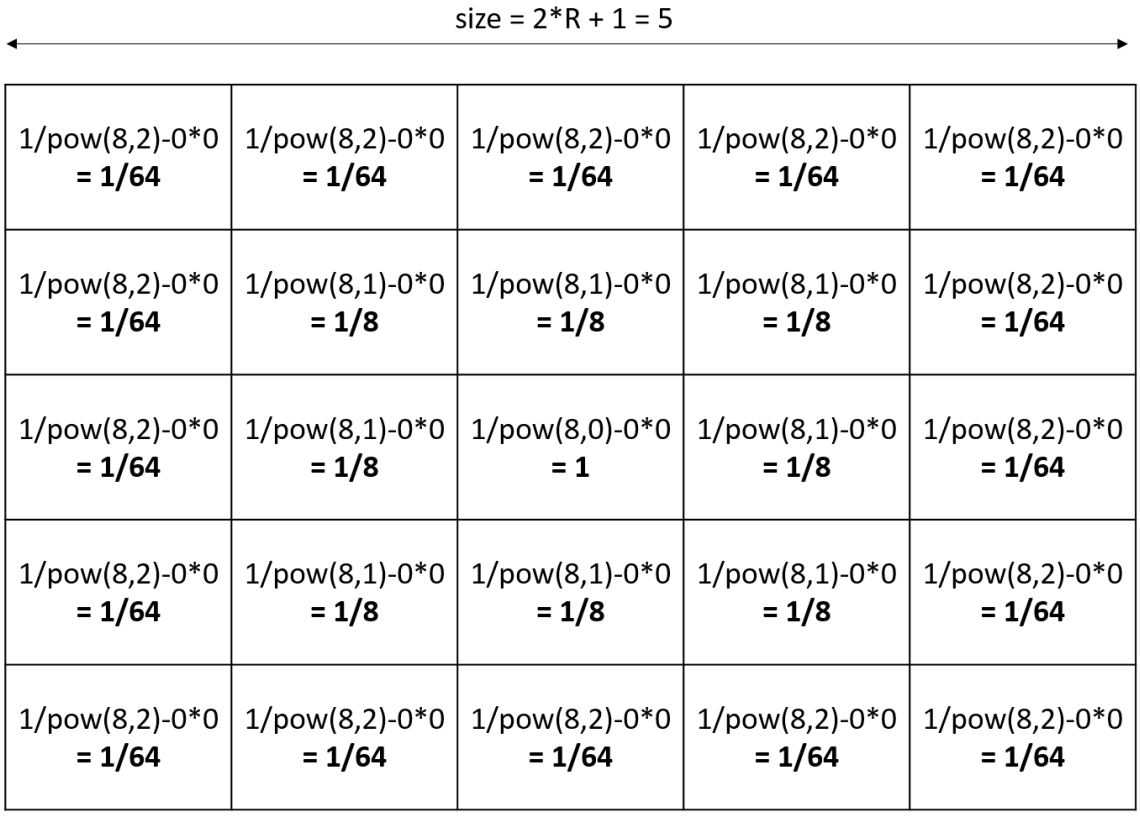


Figure 69.9: resources/images/recipes/gradient_computation_from_parameters.png

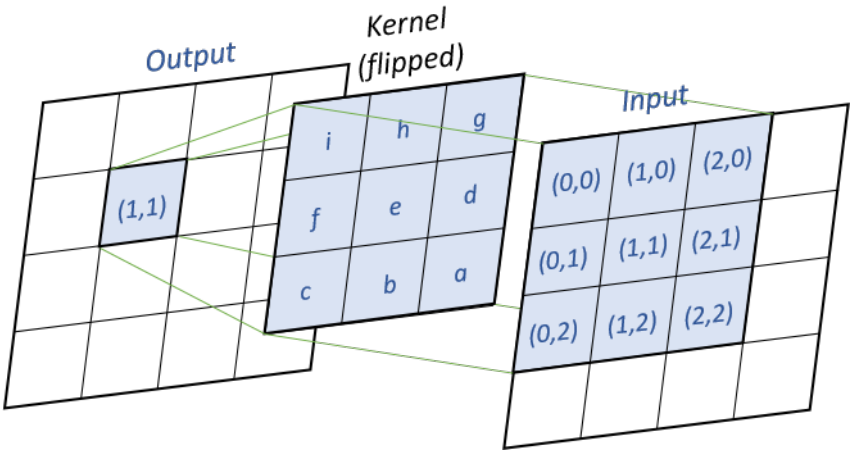


Figure 69.10: resources/images/recipes/convolution.png

Computation methods

You can compute the output matrix using two computation methods by using the facet **method** : the dot product and the convolution. Note that the result of those two methods is exactly the same (except if you use the **avoid_mask** facet, the results can be slightly differents between the two computations).

Convolution

`convolution` is the default computation method for the diffusion. For every output cells, we will multiply the input values and the flipped kernel together, as shown in the following image :

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]
```

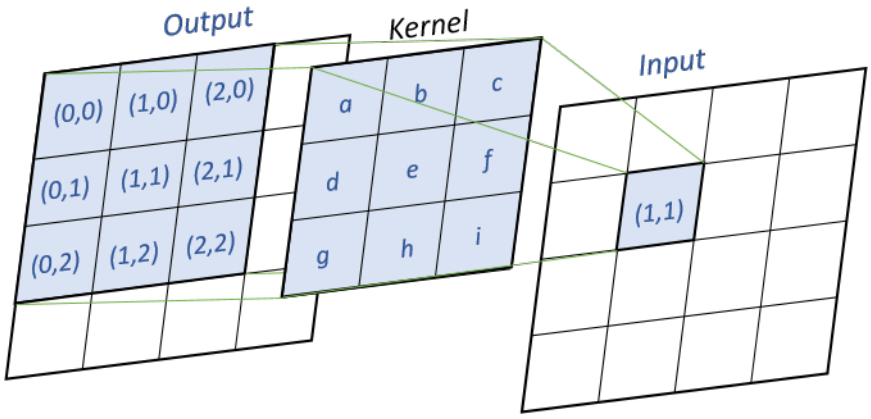


Figure 69.11: resources/images/recipes/dot_product.png

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 mask

Generalities

If you want to propagate some values in an heterogeneous grid, you can use some mask to forbid some cells to propagate their values.

You can pass a matrix to the facet **`mask`**. All the values smaller than `-1` will not propagate, and all the values greater or equal to `-1` will propagate.

A simple way to use mask is by loading an image :

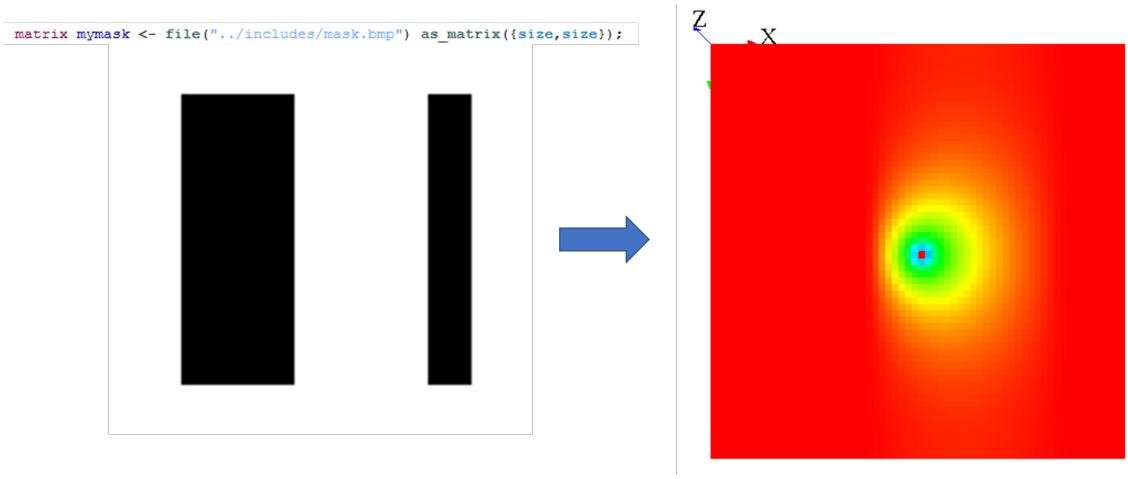


Figure 69.12: resources/images/recipes/simple_mask.png

Note that when you use the **on** facet for the `diffuse` statement, you can choose only some cells, and not every cells. In fact, when you restrain the values to be diffuse, it is exactly the same process as if you were defining a mask.

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([
```

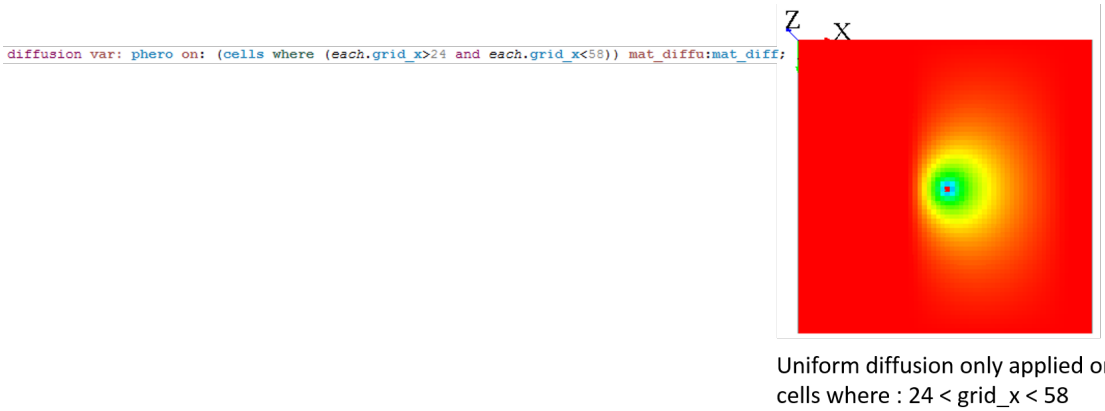


Figure 69.13: resources/images/recipes/mask_with_on_facet.png

```
            [1/9,1/9,1/9],
            [1/9,1/9,1/9],
            [1/9,1/9,1/9]]);

matrix<float> mat_diff_left_wall <- matrix([
            [0.0,0.0,2/9],
            [0.0,0.0,4/9],
            [0.0,0.0,2/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_x>30)) matrix:
   mat_diff;
    diffuse var: phero on: (cells where(each.grid_x=30)) matrix:
   mat_diff_left_wall;
}
```

Note that almost the same result can be obtained by using the facet **avoid_mask** : the value of all masked cells will remain at 0, and the value which was supposed to be affected to the masked cell will be distributed to the neighboring cells. Notice that the results can be slightly different if you are using the `convolution` or the `dot_product` method : the algorithm of redistribution of the value to the neighboring cells is a bit different. We advise you to use the `dot_product` with the **avoid_mask** facet, the results are more accurates.

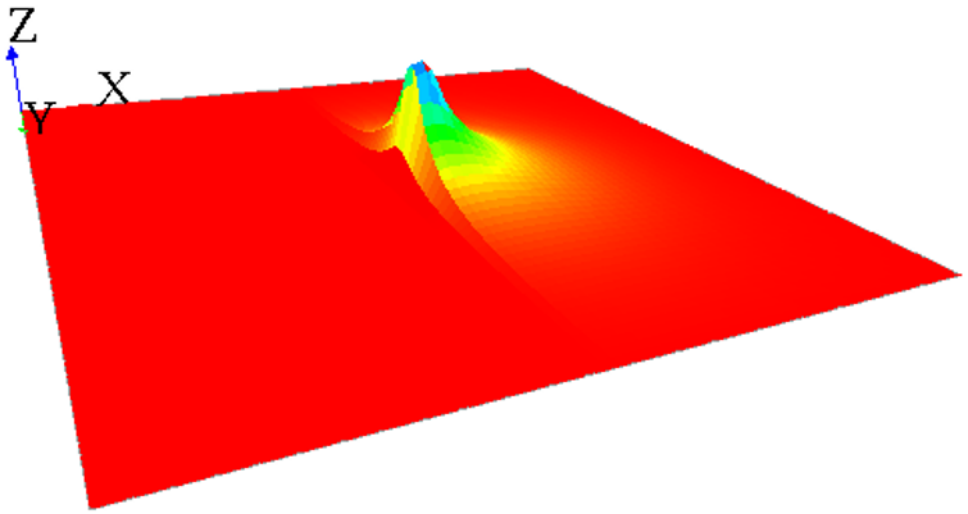


Figure 69.14: resources/images/recipes/wall_simulation.png

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 matrix : one for the uniform diffusion, one for the "north" wind and one for the "west" wind. The sum of the values for the 2 matrix meant to simulate the wind will be equal to 0 (as it will be add to the diffusion matrix).

```
matrix<float> mat_diff <- matrix([
        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);

matrix<float> mat_wind_from_west <- matrix([
        [-1/9,0.0,1/9],
        [-1/9,0.0,1/9],
        [-1/9,0.0,1/9]]);

matrix<float> mat_wind_from_north <- matrix([
        [-1/9,-1/9,-1/9],
        [0.0,0.0,0.0],
        [1/9,1/9,1/9]]);
```

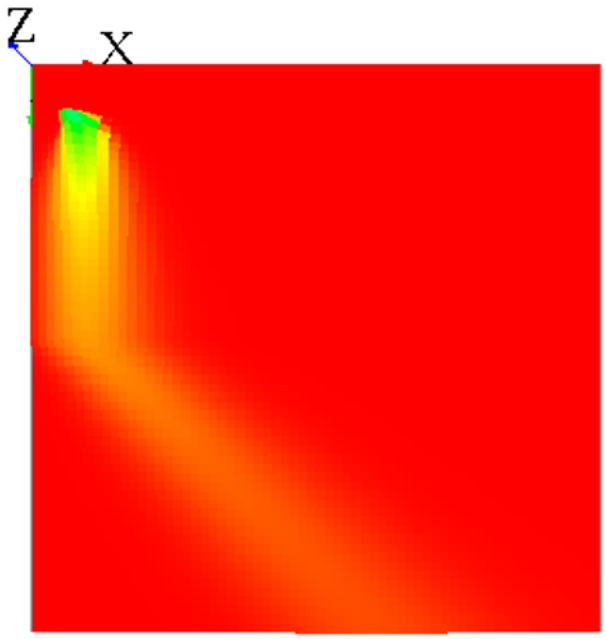


Figure 69.15: resources/images/recipes/diffusion_with_wind.png

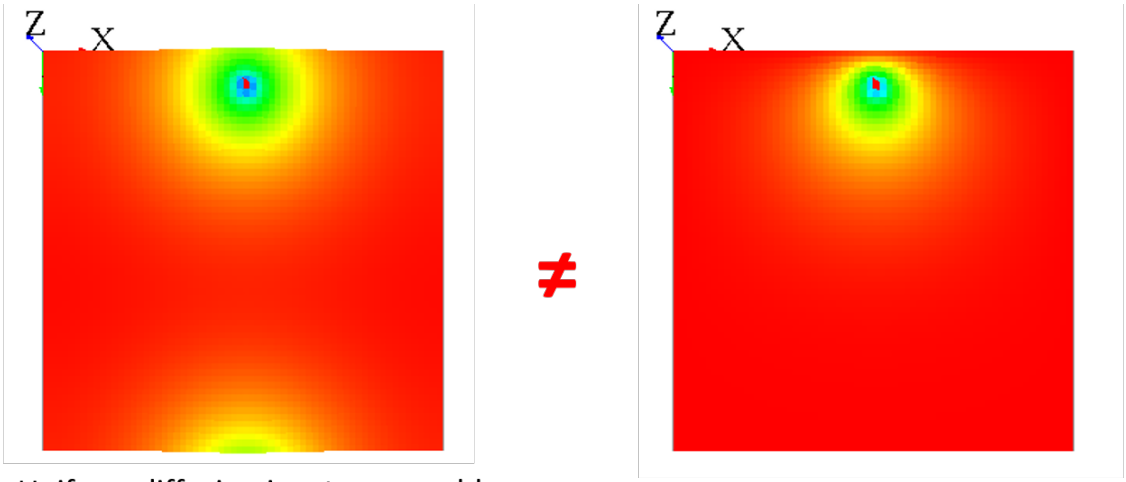
```
reflex diff {
    diffuse var: phero on: cells matrix:mat_diff;
    diffuse var: phero on: cells matrix:mat_wind_from_north;
    diffuse var: phero on: (cells where (each.grid_y>=32)) matrix:
   mat_wind_from_west;
}
```

Endless world

Note that when your world is not a torus, it has the same effect as a *mask*, since all the values outside from the world cannot diffuse some values back :

You can "fake" the fact that your world is endless by adding a different diffusion for the cells with `grid_x=0` to have almost the same result :

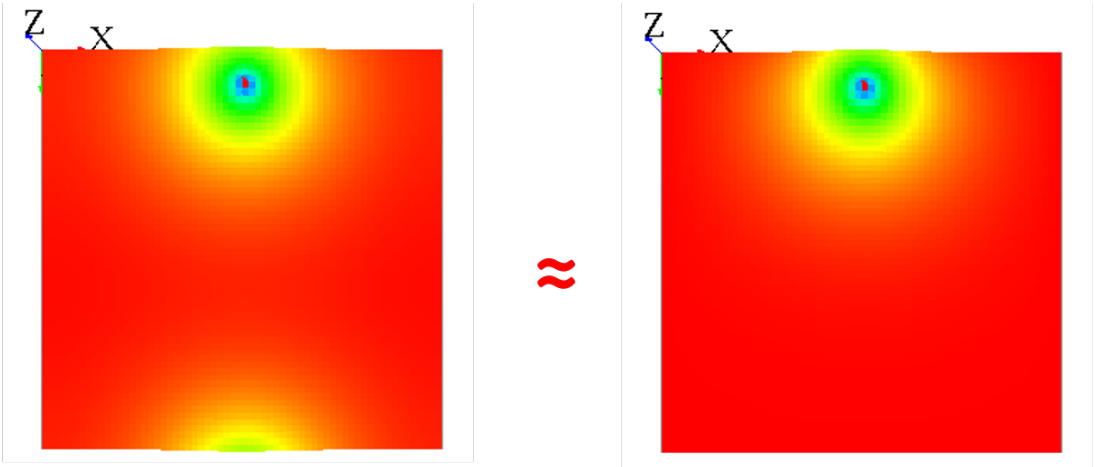
```
matrix<float> mat_diff <- matrix([
            [1/9,1/9,1/9],
            [1/9,1/9,1/9],
            [1/9,1/9,1/9]]);
```



Uniform diffusion in a torus world (after 500 cycles)

Uniform diffusion in a non-torus world (after 500 cycles)

Figure 69.16: resources/images/recipes/uniform_diffusion_near_edge.png



Uniform diffusion in a torus world (after 500 cycles)

Uniform diffusion in a non-torus world with simulation of endless world (after 500 cycles)

Figure 69.17: resources/images/recipes/uniform_diffusion_near_edge_with_-mask.png

```
matrix<float> mat_diff_upper_edge <- matrix([
            [0.0,0.0,0.0],
            [1/9+7/81,2/9+1/81,1/9+7/81],
            [1/9,1/9,1/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_y>0)) matrix:mat_diff
   ;
    diffuse var: phero on: (cells where(each.grid_y=0)) matrix:
   mat_diff_upper_edge;
}
```

Pseudo code

This section is more for a better understanding of the source code.

Here is the pseudo code for the computation of diffusion :

1) : Execute the statement `diffuse`, store the diffusions in a map (from class *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".
- Store the diffusion properties in a map
  - Map : ["method_diffu", "is_gradient", "matrix", "mask", "min_value"
   ] is value, ["var_diffu", "grid_name"] is key.
  - If the key exists in the map, try to "mix" the diffusions
    - If "method_diffu", "mask" and "is_gradient" equal for the 2
   diffusions, mix the diffusion matrix.
```

2) : At the end of the step, execute the diffusions (class *GridDiffuser*) :

```
- For each key of the map,
  - Load the couple "var_diffu" / "grid_name"
  - Build the "output" and "input" array with the dimension of the grid
    .
  - Initialize the "output" array with -Double.MAX_VALUE.
  - For each value of the map for that key,
    - Load all the properties : "method_diffu", "is_gradient", "matrix"
    , "mask", "min_value"
    - Compute :
      - If the cell is not masked, if the value of input is > min_value
    , diffuse to the neighbors.
        - If the value of the cell is equal to -Double.MAX_VALUE,
    remplace it by input[idx] * matDiffu[i][j].
        - Else, do the computation (gradient or diffusion).
    - Finish the diffusion :
      - If output[idx] > -Double.MAX_VALUE, write the new value in the
    cell.
```

Chapter 70

Using Database Access

Database features of GAMA provide a set of actions on Database Management Systems (DBMS) and Multi-Dimensional Database for agents in GAMA. Database features are implemented in the irit.gaml.extensions.database plug-in with these features:

- Agents can execute SQL queries (create, Insert, select, update, drop, delete) to various kinds of DBMS.
- Agents can execute MDX (Multidimensional Expressions) queries to select multidimensional objects, such as cubes, and return multidimensional cellsets that contain the cube's data.

These features are implemented in two kinds of component: *skills* (SQLSKILL, MDXSKILL) and agent (AgentDB)

SQLSKILL and AgentDB provide almost the same features (a same set of actions on DBMS) but with certain slight differences:

- An agent of species AgentDB will maintain a unique connection to the database during the whole simulation. The connection is thus initialized when the agent is created.
- In contrast, an agent of a species with the SQLSKILL skill will open a connection each time he wants to execute a query. This means that each action will be composed of three running steps:
  - Make a database connection.

  - Execute SQL statement.
  - Close database connection.

  An agent with the SQLSKILL spends lot of time to create/close the connection each time it needs to send a query; it saves the database connection (DBMS often limit the number of simultaneous connections). In contrast, an AgentDB agent only needs to establish one database connection and it can be used for any actions. Because it does not need to create and close database connection for each action: therefore, actions of AgentDB agents are executed faster than actions of SQLSKILL ones but we must pay a connection for each agent.

- With an inheritance agent of species AgentDB or an agent of a species using SQLSKILL, we can query data from relational database for creating species, defining environment or analyzing or storing simulation results into RDBMS. On the other hand, an agent of species with MDXKILL supports the OLAP technology to query data from data marts (multidimensional database). The database features help us to have more flexibility in management of simulation models and analysis of simulation results.

Description

- **Plug-in**: *irit.gaml.extensions.database*
- **Author**: TRUONG Minh Thai, Frederic AMBLARD, Benoit GAUDOU, Christophe SIBERTIN-BLANC

Supported DBMS

The following DBMS are currently supported:

- SQLite
- MySQL Server
- PostgreSQL Server
- SQL Server
- Mondrian OLAP Server

- SQL Server Analysis Services

Note that, other DBMSs require a dedicated server to work while SQLite on only needs a file to be accessed. All the actions can be used independently from the chosen DBMS. Only the connection parameters are DBMS-dependent.

SQLSKILL

Define a species that uses the SQLSKILL skill

Example of declaration:

```
species toto skills: [SQLSKILL] {
    //insert your descriptions here
}
```

Agents with such a skill can use additional actions (defined in the skill)

Map of connection parameters for SQL

In the actions defined in the SQLSkill, a parameter containing the connection parameters is required. It is a map with the following *key::value* pairs:

Key	Optional	Description
dbtype	No	DBMS type value. Its value is a string. We must use "mysql" when we want to connect to a MySQL. That is the same for "postgres", "sqlite" or "sqlserver" (ignore case sensitive)
host	Yes	Host name or IP address of data server. It is absent when we work with SQlite.
port	Yes	Port of connection. It is not required when we work with SQLite.
database	No	Name of database. It is the file name including the path when we work with SQLite.
user	Yes	Username. It is not required when we work with SQLite.
passwd	Yes	Password. It is not required when we work with SQLite.

Key	Optional	Description
srid	Yes	srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in *Preferences->External* configuration.

Table 1: Connection parameter description

Example: Definitions of connection parameter

```
// POSTGRES connection parameter
map <string, string>  POSTGRES <- [
     'host'::'localhost',
     'dbtype'::'postgres',
     'database'::'BPH',
     'port'::'5433',
     'user'::'postgres',
     'passwd'::'abc'];

//SQLite
map <string, string>  SQLITE <- [
    'dbtype'::'sqlite',
    'database'::'../includes/meteo.db'];

// SQLSERVER connection parameter
map <string, string> SQLSERVER <- [
    'host'::'localhost',
    'dbtype'::'sqlserver',
    'database'::'BPH',
    'port'::'1433',
    'user'::'sa',
    'passwd'::'abc'];

// MySQL connection parameter
map <string, string>  MySQL <- [
    'host'::'localhost',
    'dbtype'::'MySQL',
    'database'::'', // it may be a null string
    'port'::'3306',
    'user'::'root',
    'passwd'::'abc'];
```

Test a connection to database

Syntax: > ***testConnection*** *(params: connection_parameter)* The action tests the connection to a given database.

- **Return**: boolean. It is:
  - *true*: the agent can connect to the DBMS (to the given Database with given name and password)
  - *false*: the agent cannot connect
- **Arguments**:
  - *params*: (type = map) map of connection parameters
- **Exceptions**: *GamaRuntimeException*

Example: Check a connection to MySQL

```
if (self testConnection(params:MySQL)){
    write "Connection is OK" ;
}else{
    write "Connection is false" ;
}
```

Select data from database

Syntax: > ***select*** *(param: connection_parameter, select: selection_string,values: value_list)* The action creates a connection to a DBMS and executes the select statement. If the connection or selection fails then it throws a GamaRuntimeException.

- **Return**: list < list >. If the selection succeeds, it returns a list with three elements:
  - The first element is a list of column name.
  - The second element is a list of column type.
  - The third element is a data set.
- **Arguments**:
  - *params*: (type = map) map containing the connection parameters

- *select*: (type = string) select string. The selection string can contain question marks.
- *values*: List of values that are used to replace question marks in appropriate. This is an optional parameter.

- **Exceptions**: *GamaRuntimeException*

Example: select data from table points

```
map <string, string>   PARAMS <- ['dbtype'::'sqlite', 'database'::'../
    includes/meteo.db'];
list<list> t <- list<list> (self select(params:PARAMS,
                          select:"SELECT * FROM points ;"));
```

Example: select data from table point with question marks from table points

```
map <string, string>   PARAMS <- ['dbtype'::'sqlite', 'database'::'../
    includes/meteo.db'];
list<list> t <- list<list> (self select(params: PARAMS,
                                        select: "SELECT temp_min
    FROM points where (day>? and day<?);"
                                        values: [10,20] ));
```

Insert data into database

Syntax:

_**insert** (param: connection_parameter, into: table_name, columns: column_list, values: value‘_list) *The action creates a connection to a DBMS and executes the insert statement. If the connection or insertion fails then it throws a_GamaRuntimeException.*

- **Return**: int

If the insertion succeeds, it returns a number of records inserted by the insert.

- **Arguments**: params*: (type = map) map containing the connection parameters.* _into_: (type = string) table name. columns*: (type=list) list of column names of table. It is an optional argument. If it is not applicable then all columns of table are selected.* _values_: (type=list) list of values that are used to insert into table corresponding to columns. Hence the columns and values must have same size.
- **Exceptions**:_GamaRuntimeException

Example: Insert data into table registration

```
map<string, string> PARAMS <- ['dbtype'::'sqlite', 'database'::'../../
    includes/Student.db'];

do insert (params: PARAMS,
                into: "registration",
                values: [102, 'Mahnaz', 'Fatma', 25]);

do insert (params: PARAMS,
                 into: "registration",
                 columns: ["id", "first", "last"],
                 values: [103, 'Zaid tim', 'Kha']);

int n <- insert (params: PARAMS,
                        into: "registration",
                       columns: ["id", "first", "last"],
                       values: [104, 'Bill', 'Clark']);
```

Execution update commands

Syntax:

> ***executeUpdate*** *(param: connection_parameter, updateComm: table_name, values: value_list)* The action executeUpdate executes an update command (create/insert/delete/drop) by using the current database connection of the agent. If the database connection does not exist or the update command fails then it throws a GamaRuntimeException. Otherwise, it returns an integer value.

- **Return**: int. If the insertion succeeds, it returns a number of records inserted by the insert.

- **Arguments**:
  - *params*: (type = map) map containing the connection parameters
  - *updateComm*: (type = string) SQL command string. It may be commands: *create*, *update*, *delete* and *drop* with or without question marks.
  - *columns*: (type=list) list of column names of table.
  - *values*: (type=list) list of values that are used to replace question marks if appropriate. This is an optional parameter.
- **Exceptions**: *GamaRuntimeException*

Examples: Using action executeUpdate do sql commands (create, insert, update, delete and drop).

```
map<string, string> PARAMS <- ['dbtype'::'sqlite',  'database'::'../../
    includes/Student.db'];
// Create table
do executeUpdate (params: PARAMS,
                                updateComm: "CREATE TABLE registration"
                                                + "(id INTEGER PRIMARY KEY
    , "
                                                + " first TEXT NOT NULL, "
     + " last TEXT NOT NULL, "
                                                + " age INTEGER);");

// Insert into
do executeUpdate (params: PARAMS ,
                               updateComm: "INSERT INTO registration " +
    "VALUES(100, 'Zara', 'Ali', 18);");
do insert (params: PARAMS, into: "registration",
                columns: ["id", "first", "last"],
                values: [103, 'Zaid tim', 'Kha']);

// executeUpdate with question marks
do executeUpdate (params: PARAMS,
                               updateComm: "INSERT INTO registration " +
    "VALUES(?, ?, ?, ?);" ,
                               values: [101, 'Mr', 'Mme', 45]);

//update
int n <-  executeUpdate (params: PARAMS,
                                          updateComm: "UPDATE registration
     SET age = 30 WHERE id IN (100, 101)" );

// delete
```

```
int n <- executeUpdate (params: PARAMS,
                                        updateComm: "DELETE FROM
    registration where id=? ",
                                        values: [101] );

// Drop table
do executeUpdate (params: PARAMS, updateComm: "DROP TABLE registration"
    );
```

MDXSKILL

MDXSKILL plays the role of an OLAP tool using select to query data from OLAP server to GAMA environment and then species can use the queried data for any analysis purposes.

Define a species that uses the MDXSKILL skill

Example of declaration:

```
    species olap skills: [MDXSKILL]
     {
        //insert your descriptions here

     }
      ...
```

Agents with such a skill can use additional actions (defined in the skill)

Map of connection parameters for MDX

In the actions defined in the SQLSkill, a parameter containing the connection parameters is required. It is a map with following key::value pairs:

Key	Optional	Description
olaptype	No	OLAP Server type value. Its value is a string. We must use "SSAS/XMLA" when we want to connect to an SQL Server Analysis Services by using XML for Analysis. That is the same for "MONDRIAN/XML" or "MONDRIAN" (ignore case sensitive)
dbtype	No	DBMS type value. Its value is a string. We must use "mysql" when we want to connect to a MySQL. That is the same for "postgres" or "sqlserver" (ignore case sensitive)
host	No	Host name or IP address of data server.
port	No	Port of connection. It is no required when we work with SQLite.
database	No	Name of database. It is file name include path when we work with SQLite.
catalog	Yes	Name of catalog. It is an optional parameter. We do not need to use it when we connect to SSAS via XMLA and its file name includes the path when we connect a ROLAP database directly by using Mondrian API (see Example as below)
user	No	Username.
passwd	No	Password.

Table 2: OLAP Connection parameter description

Example: Definitions of OLAP connection parameter

```
//Connect SQL Server Analysis Services via XMLA
    map<string,string> SSAS <- [
                'olaptype'::'SSAS/XMLA',
                'dbtype'::'sqlserver',
                'host'::'172.17.88.166',
                'port'::'80',
                'database'::'olap',
                'user'::'test',
                'passwd'::'abc'];

//Connect Mondriam server via XMLA
    map<string,string>  MONDRIANXMLA <- [
                'olaptype'::"MONDRIAN/XMLA",
                'dbtype'::'postgres',
                'host'::'localhost',
```

```
                'port'::'8080',
                'database'::'MondrianFoodMart',
                'catalog'::'FoodMart',
                'user'::'test',
                'passwd'::'abc'];

//Connect a ROLAP server using Mondriam API
    map<string,string>  MONDRIAN <- [
                'olaptype'::'MONDRIAN',
                'dbtype'::'postgres',
                'host'::'localhost',
                'port'::'5433',
                'database'::'foodmart',
                'catalog'::'../includes/FoodMart.xml',
                'user'::'test',
                                'passwd'::'abc'];
```

Test a connection to OLAP database

Syntax:

> ***testConnection*** *(params: connection_parameter)* The action tests the connection to a given OLAP database.

- **Return**: boolean. It is:
  - *true*: the agent can connect to the DBMS (to the given Database with given name and password)
  - *false*: the agent cannot connect
- **Arguments**:
  - *params*: (type = map) map of connection parameters
- **Exceptions**: *GamaRuntimeException*

Example: Check a connection to MySQL

```
if (self testConnection(params:MONDIRANXMLA)){
    write "Connection is OK";
}else{
    write "Connection is false";
}
```

Select data from OLAP database

Syntax:

> ***select*** *(param: connection_parameter, onColumns: column_string, onRows: row_string from: cube_string, where: condition_string, values: value_list)* The action creates a connection to an OLAP database and executes the select statement. If the connection or selection fails then it throws a *GamaRuntimeException*.

- **Return**: list < list >. If the selection succeeds, it returns a list with three elements:
  - The first element is a list of column name.
  - The second element is a list of column type.
  - The third element is a data set.
- **Arguments**:
  - *params*: (type = map) map containing the connection parameters
  - *onColumns*: (type = string) declare the select string on columns. The selection string can contain question marks.
  - *onRows*: (type = string) declare the selection string on rows. The selection string can contain question marks.
  - *from*: (type = string) specify cube where data is selected. The cube_string can contain question marks.
  - where_: (type = string) specify the selection conditions. The condiction_string can contains question marks. This is an optional parameter. **values*: List of values that are used to replace question marks if appropriate. This is an optional parameter.
- **Exceptions**:_GamaRuntimeException

Example: select data from SQL Server Analysis Service via XMLA

```
if (self testConnection[ params::SSAS]){
    list l1  <- list(self select (params: SSAS ,
        onColumns: " { [Measures].[Quantity], [Measures].[Price] }",
        onRows:" { { { [Time].[Year].[All].CHILDREN } * "
        + " { [Product].[Product Category].[All].CHILDREN } * "
        +"{ [Customer].[Company Name].&[Alfreds Futterkiste], "
```

```
        +"[Customer].[Company Name].&[Ana Trujillo Emparedadosy helados
    ], "
        + "[Customer].[Company Name].&[Antonio Moreno Taquería] } } } "
     ,
        from : "FROM [Northwind Star] "));
    write "result1:"+ l1;
}else {
    write "Connect error";
}
```

Example: select data from Mondrian via XMLA with question marks in selection

```
if (self testConnection(params:MONDRIANXMLA)){
    list<list> l2  <- list<list> (self select(params: MONDRIANXMLA,
    onColumns:" {[Measures].[Unit Sales], [Measures].[Store Cost], [
    Measures].[Store Sales]} ",
    onRows:"  Hierarchize(Union(Union(Union({([Promotion Media].[All
    Media],"
    +" [Product].[All Products])}, "
    +" Crossjoin([Promotion Media].[All Media].Children, "
    +" {[Product].[All Products]})), "
    +" Crossjoin({[Promotion Media].[Daily Paper, Radio, TV]}, "
    +" [Product].[All Products].Children)), "
    +" Crossjoin({[Promotion Media].[Street Handout]}, "
    +" [Product].[All Products].Children))) ",
    from:" from [?] " ,
    where :" where [Time].[?] " ,
    values:["Sales",1997]));
    write "result2:"+ l2;
}else {
    write "Connect error";
}
```

AgentDB

AgentBD is a built-in species, which supports behaviors that look like actions in SQLSKILL but differs slightly with SQLSKILL in that it uses only one connection for several actions. It means that AgentDB makes a connection to DBMS and keeps that connection for its later operations with DBMS.

Define a species that is an inheritance of agentDB

Example of declaration:

```
species agentDB parent: AgentDB {
    //insert your descriptions here
}
```

Connect to database

Syntax:

> ***Connect*** *(param: connection_parameter)* This action makes a connection to DBMS. If a connection is established then it will assign the connection object into a built-in attribute of species (conn) otherwise it throws a GamaRuntimeException.

- **Return**: connection
- **Arguments**:
  - *params*: (type = map) map containing the connection parameters
- **Exceptions**: GamaRuntimeException

Example: Connect to PostgreSQL

```
// POSTGRES connection parameter
map <string, string>  POSTGRES <- [
                                        'host'::'localhost',
                                        'dbtype'::'postgres',
                                        'database'::'BPH',
                                        'port'::'5433',
                                        'user'::'postgres',
                                        'passwd'::'abc'];
ask agentDB {
      do connect (params: POSTGRES);
}
```

Check agent connected a database or not

Syntax:

> ***isConnected*** *(param: connection_parameter)* This action checks if an agent is connecting to database or not.

- **Return**: Boolean. If agent is connecting to a database then isConnected returns true; otherwise it returns false.
- **Arguments**:
  - *params*: (type = map) map containing the connection parameters

Example: Using action executeUpdate do sql commands (create, insert, update, delete and drop).

```
ask agentDB {
    if (self isConnected){
              write "It already has a connection";
    }else{
              do connect (params: POSTGRES);
        }
}
```

Close the current connection

Syntax:

> ***close*** This action closes the current database connection of species. If species does not has a database connection then it throws a GamaRuntimeException.

- **Return**: null

If the current connection of species is close then the action return null value; otherwise it throws a GamaRuntimeException.

Example:

```
ask agentDB {
    if (self isConnected){
          do close;
    }
}
```

Get connection parameter

Syntax:

getParameter This action returns the connection parameter of species.

- **Return**: map < string, string >

Example:

```
ask agentDB {
    if (self isConnected){
        write "the connection parameter: " +(self getParameter);
        }
}
```

Set connection parameter

Syntax:

setParameter *(param: connection_parameter)* This action sets the new values for connection parameter and closes the current connection of species. If it can not close the current connection then it will throw GamaRuntimeException. If the species wants to make the connection to database with the new values then action connect must be called.

- **Return**: null
- **Arguments**:
  - *params*: (type = map) map containing the connection parameters

- **Exceptions**: *GamaRuntimeException*

Example:

```
ask agentDB {
    if (self isConnected){
              do setParameter(params: MySQL);
              do connect(params: (self getParameter));
        }
}
```

Retrieve data from database by using AgentDB

Because of the connection to database of AgentDB is kept alive then AgentDB can execute several SQL queries with only one connection. Hence AgentDB can do actions such as **select**, **insert**, **executeUpdate** with the same parameters of those actions of SQLSKILL *except* ***params*** *parameter is always absent.*

Examples:

```
map<string, string> PARAMS <- ['dbtype'::'sqlite', 'database'::'../../
    includes/Student.db'];
ask agentDB {
   do connect (params: PARAMS);
   // Create table
   do executeUpdate (updateComm: "CREATE TABLE registration"
    + "(id INTEGER PRIMARY KEY, "
        + " first TEXT NOT NULL, " + " last TEXT NOT NULL, "
        + " age INTEGER);");
   // Insert into
   do executeUpdate ( updateComm: "INSERT INTO registration "
        + "VALUES(100, 'Zara', 'Ali', 18);");
   do insert (into: "registration",
     columns: ["id", "first", "last"],
     values: [103, 'Zaid tim', 'Kha']);
   // executeUpdate with question marks
   do executeUpdate (updateComm: "INSERT INTO registration VALUES(?, ?,
    ?, ?);",
     values: [101, 'Mr', 'Mme', 45]);
   //select
    list<list> t <- list<list> (self select(
     select:"SELECT * FROM registration;"));
    //update
```

```
    int n <-  executeUpdate (updateComm: "UPDATE registration SET age =
    30 WHERE id IN (100, 101)");
     // delete
     int n <- executeUpdate ( updateComm: "DELETE FROM registration
   where id=? ",  values: [101] );
     // Drop table
      do executeUpdate (updateComm: "DROP TABLE registration");
}
```

Using database features to define environment or create species

In Gama, we can use results of select action of SQLSKILL or AgentDB to create species or define boundary of environment in the same way we do with shape files. Further more, we can also save simulation data that are generated by simulation including geometry data to database.

Define the boundary of the environment from database

- **Step 1**: specify select query by declaration a map object with keys as below:

Key	Optional	Description
dbtype	No	DBMS type value. Its value is a string. We must use "mysql" when we want to connect to a MySQL. That is the same for "postgres", "sqlite" or "sqlserver" (ignore case sensitive)
host	Yes	Host name or IP address of data server. It is absent when we work with SQlite.
port	Yes	Port of connection. It is not required when we work with SQLite.
database	No	Name of database. It is the file name including the path when we work with SQLite.
user	Yes	Username. It is not required when we work with SQLite.
passwd	Yes	Password. It is not required when we work with SQLite.

Key	Optional	Description
srid	Yes	srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in Preferences->External configuration.
select	No	Selection string

Table 3: Select boundary parameter description

Example:

```
map<string,string> BOUNDS <- [
    //'srid'::'32648',
    'host'::'localhost',
        'dbtype'::'postgres',
    'database'::'spatial_DB',
    'port'::'5433',
        'user'::'postgres',
    'passwd'::'tmt',
    'select'::'SELECT ST_AsBinary(geom) as geom FROM bounds;' ];
```

- **Step 2**: define boundary of environment by using the map object in first step.

```
geometry shape <- envelope(BOUNDS);
```

Note: We can do the same way if we work with MySQL, SQLite, or SQLServer and we must convert Geometry format in GIS database to binary format.

Create agents from the result of a select action

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1**: Define a species with SQLSKILL or AgentDB

```
species toto skills: SQLSKILL {
    //insert your descriptions here
}
```

- **Step 2**: Define a connection and selection parameters

```
global {
    map<string,string>  PARAMS <- ['dbtype'::'sqlite','database'::'../
    includes/bph.sqlite'];
    string location <- 'select ID_4, Name_4, ST_AsBinary(geometry) as
    geom from vnm_adm4
                                        where id_2=38253 or id_2=38254;';
    ...
}
```

- **Step 3**: Create species by using selected results

```
init {
   create toto {
      create locations from: list(self select (params: PARAMS,
                                                          select:
    LOCATIONS))
                                    with:[ id:: "id_4", custom_name:: "
    name_4", shape::"geom"];
    }
   ...
}
```

Save Geometry data to database

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1**: Define a species with SQLSKILL or AgentDB

```
species toto skills: SQLSKILL {
    //insert your descriptions here
}
```

- **Step 2**: Define a connection and create GIS database and tables

```
global {
    map<string,string> PARAMS <-  ['host'::'localhost', 'dbtype'::'
   Postgres', 'database'::'',
                                                              'port
   '::'5433', 'user'::'postgres', 'passwd'::'tmt'];

    init {
        create toto ;
        ask toto {
            if (self testConnection[ params::PARAMS]){
                // create GIS database
                do executeUpdate(params:PARAMS,
                        updateComm: "CREATE DATABASE spatial_db with
   TEMPLATE = template_postgis;");
                    remove key: "database" from: PARAMS;
                put "spatial_db" key:"database" in: PARAMS;
                //create table
                            do executeUpdate params: PARAMS
                  updateComm : "CREATE TABLE buildings "+
                  "( "  +
                                  " name character varying(255), " +
                                     " type character varying(255), "
    +
                                     " geom GEOMETRY " +
                                  ")";
            }else {
                write "Connection to MySQL can not be established ";
            }
        }
    }
}
```

- **Step 3**: Insert geometry data to GIS database

```
ask building {
   ask DB_Accessor {
    do insert(params: PARAMS,
                        into: "buildings",
            columns: ["name", "type","geom"],
            values: [myself.name,myself.type,myself.shape];
   }
}
```

Chapter 71

Calling R

Introduction

R language is one of powerful data mining tools, and its community is very large in the world (See the website: http://www.r-project.org/). Adding the R language into GAMA is our strong endeavors to accelerate many statistical, data mining tools into GAMA.

RCaller 2.0 package (Website: http://code.google.com/p/rcaller/) is used for GAMA 1.6.1.

Table of contents

Configuration in GAMA

1) Install R language into your computer.
2) In GAMA, select menu option: **Edit/Preferences**.
3) In "**Config RScript's path**", browse to your "**Rscript**" file (R language installed in your system).

Notes: Ensure that install.packages("Runiversal") is already applied in R environment.

Calling R from GAML

Calling the built-in operators

Example 1

```
model CallingR

global {
    list X <- [2, 3, 1];
    list Y <- [2, 12, 4];

    list result;

    init{
        write corR(X, Y); // -> 0.755928946018454
        write meanR(X); // -> 2.0
    }
}
```

Calling R codes from a text file (.R,.txt) WITHOUT the parameters

Using **R_compute(String RFile)** operator. This operator DOESN'T ALLOW to add any parameters form the GAML code. All inputs is directly added into the R codes. **Remarks**: Don't let any white lines at the end of R codes. **R_compute** will return the last variable of R file, this parameter can be a basic type or a list. Please ensure that the called packages must be installed before using.

Example 2

```
model CallingR

global
{
    list result;

    init{
        result <- R_compute("C:/YourPath/Correlation.R");
        write result at 0;
    }
}
```

Above syntax is deprecated, use following syntax with R_file instead of R_compute:

```
model CallingR
```

```
global
{
    file result;

    init{
        result <- R_file("C:/YourPath/Correlation.R");
        write result.contents;
    }
}
```

Correlation.R file

```
x <- c(1, 2, 3)

y <- c(1, 2, 4)

result <- cor(x, y, method = "pearson")
```

Output

```
result::[0.981980506061966]
```

Example 3

```
model CallingR

global
{
    list result;

    init{
        result <- R_compute("C:/YourPath/RandomForest.R");

        write result at 0;
    }
}
```

RandomForest.R file

```
# Load the package:

library(randomForest)

# Read data from iris:

data(iris)

nrow<-length(iris[,1])

ncol<-length(iris[1,])

idx<-sample(nrow,replace=FALSE)

trainrow<-round(2*nrow/3)

trainset<-iris[idx[1:trainrow],]

# Build the decision tree:

trainset<-iris[idx[1:trainrow],]

testset<-iris[idx[(trainrow+1):nrow],]

# Build the random forest of 50 decision trees:

model<-randomForest(x= trainset[,-ncol], y= trainset[,ncol], mtry=3,
    ntree=50)

# Predict the acceptance of test set:

pred<-predict(model, testset[,-ncol], type="class")

# Calculate the accuracy:

acc<-sum(pred==testset[, ncol])/(nrow-trainrow)
```

Output

```
acc::[0.98]
```

Calling R codes from a text file (.R, .txt) WITH the parameters

Using **R_compute_param(String RFile, List vectorParam)** operator. This operator ALLOWS to add the parameters from the GAML code.

Remarks: Don't let any white lines at the end of R codes. **R_compute_param** will return the last variable of R file, this parameter can be a basic type or a list. Please ensure that the called packages must be installed before using.

Example 4

```
model CallingR

global
{
    list X <- [2, 3, 1];
    list result;

    init{
        result <- R_compute_param("C:/YourPath/Mean.R", X);
        write result at 0;
    }
}
```

Mean.R file

```
result <- mean(vectorParam)
```

Output

```
result::[3.33333333333333]
```

Example 5

```
model CallingR

global {
```

```
    list X <- [2, 3, 1];
    list result;

        init{
        result <- R_compute_param("C:/YourPath/AddParam.R", X);
        write result at 0;
    }
}
```

AddParam.R file

```
v1 <- vectorParam[1]

v2<-vectorParam[2]

v3<-vectorParam[3]

result<-v1+v2+v3
```

Output

```
result::[10]
```

Chapter 72

Using FIPA ACL

The communicating skill offers some actions and built-in variables which enable agents to communicate with each other using the FIPA interaction protocol. This document describes the built-in variables and actions of this skill. Examples are found in the models library bundled with GAMA.

Variables

- **accept_proposals (list)**: A list of 'accept_proposal' performative messages of the agent's mailbox having .
- **agrees (list)**: A list of 'accept_proposal' performative messages.
- **cancels (list)**: A list of 'cancel' performative messages.
- **cfps (list)**: A list of 'cfp' (call for proposal) performative messages.
- **conversations (list)**: A list containing the current conversations of agent. Ended conversations are automatically removed from this list.
- **failures (list)**: A list of 'failure' performative messages.
- **informs (list)**: A list of 'inform' performative messages.
- **messages (list)**: The mailbox of the agent, a list of messages of all types of performatives.
- **proposes (list)**: A list of 'propose' performative messages .
- **queries (list)**: A list of 'query' performative messages.
- **refuses (list)**: A list of 'propose' performative messages.
- **reject_proposals (list)**: A list of 'reject_proposals' performative messages.
- **requests (list)**: A list of 'request' performative messages.

- **requestWhens (list)**: A list of 'request-when' performative messages.
- **subscribes (list)**: A list of 'subscribe' performative messages.

Actions

accept_proposal

Replies a message with an 'accept_proposal' performative message

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

agree

Replies a message with an 'agree' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

cancel

Replies a message with a 'cancel' peformative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

cfp

Replies a message with a 'cfp' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

end_conversation

Replies a message with an 'end_conversation' peprformative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsible of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

failure

Replies a message with a 'failure' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

inform

Replies a message with an 'inform' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

propose

Replies a message with a 'propose' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

query

Replies a message with a 'query' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

refuse

Replies a message with a 'refuse' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

reject_proposal

Replies a message with a 'reject_proposal' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

reply

Replies a message. This action should be only used to reply a message in a 'no-protocol' conversation and with a 'user defined performative'. For performatives supported by GAMA (i.e., standard FIPA performatives), please use the 'action' with the same name of 'performative'. For example, to reply a message with a 'request' performative message, the modeller should use the 'request' action.

- returns: unknown
- message (message): The message to be replied
- performative (string): The performative of the replying message
- content (list): The content of the replying message

request

Replies a message with a 'request' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

send

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.Message
- receivers (list): A list of receiver agents
- content (list): The content of the message. A list of any GAML type
- performative (string): A string, representing the message performative
- protocol (string): A string representing the name of interaction protocol

start_conversation

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.Message
- receivers (list): A list of receiver agents
- content (list): The content of the message. A list of any GAML type
- performative (string): A string, representing the message performative
- protocol (string): A string representing the name of interaction protocol

subscribe

Replies a message with a 'subscribe' performative message.

- returns: unknown
- message (message): The message to be replied
- content (list): The content of the replying message

Chapter 73

Using GAMAnalyzer

Install

Go to Git View -> Click on Import Projects Add the dependencies in ummisco.gama.feature.dependencies

GamAnalyzer is a tool to monitor several multi-agents simulation

The “agent_group_follower” goal is to monitor and analyze a group of agent during several simulation. This group of agent can be chosen by the user according to criteria chosen by the user. The monitoring process and analysis of these agents involves the extraction, processing and visualization of their data at every step of the simulation. The data for each simulation are pooled and treated commonly for their graphic representation or clusters.

Built-in Variable

- **varmap**: All variable that can be analyzed or displayed in a graph.
- **numvarmap**: Numerical variable (on this variable all the aggregator numeric are computed).
- **qualivarmap**: All non numerical variable. Could be used for BDI to analyze beliefs.

- **metadatahistory**: See updateMetaDataHistory. This matrice store all the metadata like getSimulationScope(), getClock().getCycle(), getUniqueSimName(scope), rule, scope.getAgentScope().getName(), this.getName(), this.agentsCourants.copy(scope), this.agentsCourants.size(), this.getGeometry().
- **lastdetailedvarvalues**: store all the value (in varmap) for all the followed agent for the last iteration.
- **averagehistory**: Average value for each of the numvar
- **stdevhistory**: Std deviation value for each of the numvar
- **minhistory**: Min deviation value for each of the numvar
- **maxhistory**: Max deviation value for each of the numvar
- **distribhistoryparams**: Gives the interval of the distribution described in distribhistory
- **distribhistory**: Distribution of numvarmap
- **multi_metadatahistory**: Aggregate each metadatahistory for each experiment

Example

This example is based on a toy model which is only composed of wandering people. In this example we will use GamAnalyzer to follow the agent people.

```
agent_group_follower peoplefollower;
```

```
create agentfollower
{
  do analyse_cluster species_to_analyse:"people";
  peoplefollower<-self;
}
```

expGlobalNone

No clustering only the current agent follower is displayed

```
aspect base {
  display_mode <-"global";
  clustering_mode <-"none";
  draw shape color: #red;
}
```

expSimGlobalNone

The agent_group_follower corresponding to the current iteration and all the already launch experiments are displayed.

```
aspect simglobal{
  display_mode <-"simglobal";
  clustering_mode <-"none";
  draw shape color: #red;
  int curColor <-0;
  loop geom over: allSimShape{
    draw geom color:SequentialColors[curColor] at:{location.x,location.
   y,curColor*10};
    curColor <- curColor+1;
  }
}
```

expCluster

The agent group follower is divided in cluster computed thanks to a dbscan algorithm. Only the current agent_group_follower is displayed

```
aspect cluster {
  display_mode <-"global";
  clustering_mode <-"dbscan";
  draw shape color: #red;
}
```

expClusterSimGlobal

The agent_group_follower (made of different cluster) corresponding to the current iteration and all the already launch experiments are displayed.

```
aspect clusterSimGlobal {
  display_mode <-"simglobal";
  clustering_mode <-"dbscan";
  draw shape color: #red;
  int curColor <-0;
  loop geom over: allSimShape{
    draw geom color:SequentialColors[curColor] at:{location.x,location.
   y,curColor*10};
    curColor <- curColor+1;
  }
}
```

Chapter 74

Using BDI

Acteur Projet

A website (still in construction) of the ACTEUR project can be found here http://acteur-anr.fr/

This project lead to the BEN Architecture wich is a more complete version of the BDI architecture described on this page. A detailed description of BEN may be found here.

An introduction to cognitive agent

The belief-desire-intention software model (usually referred to simply, but ambiguously, as BDI) is a software model developed for programming intelligent agents.

- **Belief**: State of the agent.
- **Desire**: Objectives that the agent would like to accomplish.
- **Intention**: What the agent has chosen to do.
- **Plan**: Sequences of actions that an agent can perform to achieve one or more of its intensions.

Basic Example: A fire rescue model using cognitive agent

We introduce a simple example to illustrate the use of the BDI architecture.

This simple model consists in creating "cognitive" agent whose goal is to extinguish a fire. In a first approximation we consider only one static water area and fire area. The aim is not to have a realistic model but to illustrate how to give a "cognitive" behavior to an agent using the BDI architecture.

First let's create a BDI agent using the key control **simple_bdi** (A description of all existing control architectures is available here.)

Species Helicopter creation

```
species helicopter skills:[moving] control: simple_bdi{
...
}
```

Attributes

The species `helicopter` needs 2 attributes to represent the water value and its speed.

```
float waterValue;
float speed <- 10.0;
```

Predicates

The predicate are the structure that are used to define a belief, a desire or an intention. In this model we choose to declare 3 different predicates.

```
predicate patrol_desire <- new_predicate("patrol") with_priority 1;
predicate water_predicate <- new_predicate("has water", true)
    with_priority 3;
predicate no_water_predicate <- new_predicate("has water", false) ;
```

The **new_predicate()** tool creates a predicate. It needs a name (string type) and it can contain a map of values, a priority (double type) or a truth value (boolean

type). The **with_priority** tool add a priority to a predicate. The priority is used as an argument when the agent has to choose between two predicates (to choose an intention for example).

Initialization

The initialization consists in setting the attribute **waterValue** to 1 and to add one desire. Three optional parameters are also set. The first desire added in the desire base is the **patrol_desire** saying that the helicopter wants to patrol. The optional parameters are specific to the BDI plug-in. You can specify the commitment of an agent to his intentions and his plans with the variables intention_persistence and plan_persistence that are floats between 0.0 (no commitment) and 1.0. The variable probabilistic_choice is a boolean that enables the agent to use a probabilistic choice (when true) or a deterministic choice (when false) when trying to find a plan or an intention.

```
waterValue <-1.0;
do add_desire(patrol_desire);
intention_persistence <- 1.0;
plan_persistence <- 1.0;
probabilistic_choice <- false;
```

Perception

At each iteration, the helicopter has two perceptions to do. The first one is about itself. The helicopter needs to perceive if it has water or not. If it has water, it adds the belief corresponding belief and removes the belief that it does not have water. And if it does not have water, that is the contrary.

```
perceive target:self{
    if(waterValue>0){
        do add_belief(water_predicate);
        do remove_belief(no_water_predicate);
    }
    if(waterValue<=0){
        do add_belief(no_water_predicate);
        do remove_belief(water_predicate);
    }
}
```

The second perception is about the fires. Here, the fires are represented with the species **fireArea**. The helicopter has a radius of perception of 10 meters. If it perceives a fire, it will focus on the location of this fire. The **focus** tool create a belief with the same name as the focus (here, "fireLocation") and will store the value of the focused variable (here, the variable location from the specie fireArea) with a priority of 10 in this example. Once the fire is perceived, the helicopter removes its intention of patrolling.

```
perceive target:fireArea in: 10{
    focus fireLocation var:location priority:10;
    ask myself{
        do remove_intention(patrol_desire, true);
    }
}
```

Rules

The agent can use rules to create desires from beliefs. In this example, the agent has two rules. The first **rule** is to have a desire corresponding to the belief of a location of a fire. It means that when the agent has the belief that there is a fire in a particular location, it will have the desire to extinguish it. This permits to have the location value in the desire base. The second rule is to create the desire to have water when the agent has the belief that it not has water.

```
rule belief: new_predicate("fireLocation") new_desire:
   get_belief_with_name("fireLocation");
rule belief: no_water_predicate new_desire: water_predicate;
```

Plan

Patrolling

This plan will be used when the agent has the intention to patrol.

```
plan patrolling intention: patrol_desire{
  do wander;
}
```

stopFire

This plan is executed when the agent has the intention to extinguish a fire.

```
plan stopFire intention: new_predicate("fireLocation") {
    point target_fire <- point(get_current_intention().values["
   location_value"] );
    if(waterValue>0){
        if (self distance_to target_fire <= 1) {
            fireArea current_fire <- fireArea first_with (each.location
    = target_fire);
            if (current_fire != nil) {
                 waterValue <- waterValue - 1.0;
                 current_fire.size <-  current_fire.size - 1;
                 if ( current_fire.size <= 0) {
                    ask  current_fire {do die;}
                    do remove_belief(get_current_intention());
                    do remove_intention(get_current_intention(), true);
                    do add_desire(patrol_desire);
                }
            } else {
                do remove_belief(get_current_intention());
                do remove_intention(get_current_intention(), true);
                do add_desire(patrol_desire);
            }
        } else {
            do goto target: target_fire;
        }
    } else {
        do add_subintention(get_current_intention(),water_predicate,
   true);
        do current_intention_on_hold();
    }
}
```

gotoTakeWater

This plan is executed when the agent has the intention to have water.

```
plan gotoTakeWater intention: water_predicate {
        waterArea wa <- first(waterArea);
        do goto target: wa);
        if (self distance_to wa <= 1) {
            waterValue <- waterValue + 2.0;
    }
}
```

Plans can have other options. They can have a priority (with the facet priority), a boolean condition to start (with the facet when) or a boolean condition to stop (with the facet finished_when).

Rest of the code

Aspect of the helicopter

```
aspect base {
    draw circle(1) color: #black;
}
```

FireArea Species

```
species fireArea{
        float size <-1.0;

        aspect base {
          draw circle(size) color: #red;
        }
}
```

WaterArea Species

```
species waterArea{
    float size <-10.0;

    aspect base {
      draw circle(size) color: #blue;
    }
}
```

Chapter 75

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 PhD 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 practicle 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 tutorials or GAML References/build-in architecture/BDI.

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 personnality. Each part is made up of processes that

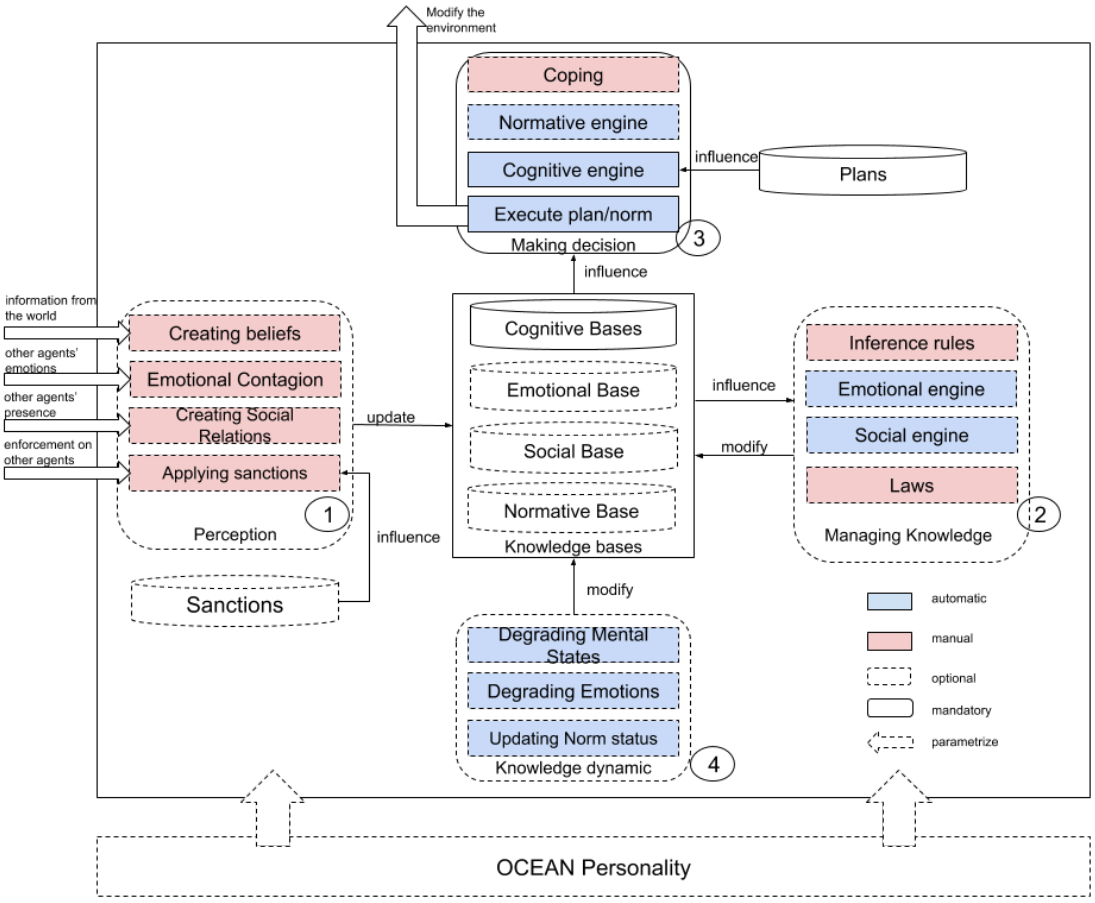


Figure 75.1: architectureBEN

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 **Pj(V)**. A predicate **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 simple_bdi 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_value chages 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 edfine for predicates.

Below is an example of how to define predicates in GAML:

```
predicate a <- new_predicate("test");
predicate a <- new_predicate("test",["value1"::10]);
predicate b <- new_predicate("test",agentBob);
predicate c <- new_predicate("test",false);
predicate d <- new_predicate("test",agenBob,false);
```

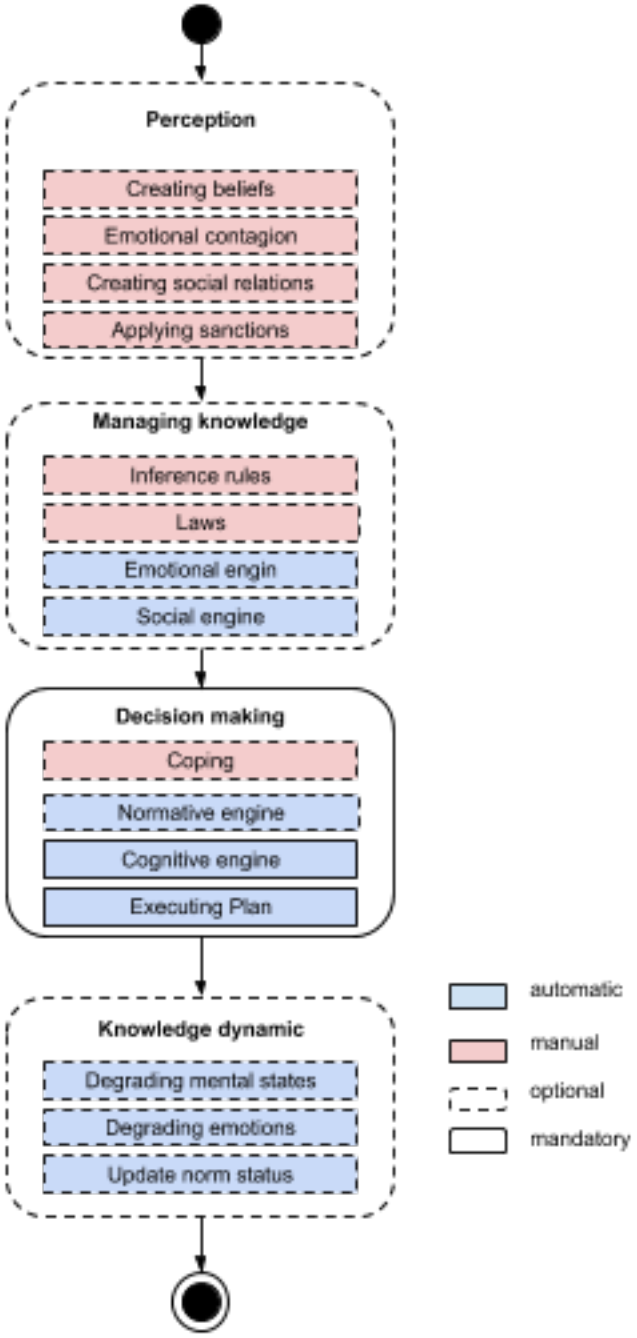


Figure 75.2: activityDiagram

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 **Mi(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 **Mi(PMEm)**. **Val**[**Mi(PMEm)**] represents the value attached to a particular cognitive mental state and **Li**[**Mi(PMEm)**] represents its lifetime.

The cognitive part of BEN is based on the BDI paradigm [**?**] 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 **Emi(P,Ag,I,De)** with the following elements :

- **Emi**: 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 $\mathbf{Em}_i\mathbf{(P,Ag)}$ and an emotion caused by any agent is written **{Emi(P)**. **I[Emi(P,Ag)]** stands for the intensity of a particular emotion and **De[Emi(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 cognitive mental states:

```
reflex testEmotion{
    predicate a <- new_predicate("test");
    do add_emotion("hope",a);
    do add_emotion("joy",a,intesity1,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, the 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 **Ri,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 Ri,j(L,D,S,F,T) is not equal by definition to Ri,j(L,D,S,F,T). **L[Ri,j]** stands for the

liking value of the social relation between agent i and agent j, **D[i,j]** stands for its dominance value, **S[Ri,j]** for its solidarity value, **F[Ri,j]** represents its familiarity value and **T[Ri,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_relation(new_social_relation(agentAlice));
    do add_social_relation(new_social_relation(agentBob
   ,0.5,-0.3,0.2,0.1,0.8));
    float a <- get_liking(get_social_relation(new_social_relation(
   agentBob)));
    set_dominance(get_social_relation(new_social_relation(agentBob))
   ,0.3);
}
```

Personality and additionnal 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 simple_bdi control architecture. They are called *openness*, *conscientiousness*, *extroversion*, *agreeableness*

and *neurotism*. To use this personality to automaticaly 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 = C1/2

Probability Keeping Intentions = C1/2

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 neurotism 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 manualy. 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 in **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 *BeliefA(X)* triggers multiple processes :

- it removes *BeliefA(not X)*.
- it removes *IntentionA(X)*.
- it removes *DesireA(X)* if *IntentionA(X)* has just been removed.
- it removes *UncertaintyA(X)* or *UncertaintyA(not X)*.
- it removes *ObligationA(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 *Chi x Rj* 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 automaticaly computed.

If Emj(P) already exists : *I[Emj(P)] = I[Emj(P)] + I[Emi(P)] x Chi x Rj* and *De[Emj(P)] = De[Emi(P)] if I[Emi(P)] > I[Emj(P)]* or *De[Emj(P)] = De[Emj(P)] if I[Emj(P)] > I[Emi(P)]*.

If Emj(P) does not already exist : *I[Emj(P)] = I[Emi(P)] x Chi x Rj* and *De[Emj(P)] = De[Emi(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 ocument.

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 knowledges

The second step of the architecture, corresponding to the module number 2, consists in 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. Theses 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 modelers 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:

- **Joyi(Pj,j)** = Beliefi(Pj) & Desirei(P)
- **Sadnessi(Pj,j)** = Beliefi(Pj) & Desirei(not P)

Their initial intensity is computed according to the following equation with N the neurotism component from the OCEAN model:

I[Emi(P)] = V[Beliefi(P)] x V[Desirei(P)] x (1+(0,5-N))

The emotions about uncertainties are fear and hope and are defined this way:

- **Hopei(Pj,j)** = Uncertaintyi(Pj) & Desirei(P)
- **Feari(Pj,j)** = Uncertaintyi(Pj) & Desirei(not P)

Their initial intensity is computed according to the following equation:

I[Emi(P)] = V[Uncertaintyi(P)] x V[Desirei(P)] x (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:

- **Satisfactioni(Pj,j)** = Hopei(Pj,j) & Beliefi(Pj)
- **Disappointmenti(Pj,j)** = Hopei(Pj,j) & Beliefi(not Pj)
- **Reliefi(Pj,j)** = Feari(Pj,j) & Beliefi(not Pj)
- **Fear confirmedi(Pj,j)** = Feari(Pj,j) & Beliefi(Pj)

Their initial intensity is computed according to the following equation with Em'i(P) the emotion of fear/hope.

I[Emi(P)] = V[Beliefi(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 a 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 fori(P,j)** = L[Ri,j]>0 & Joyj(P)
- **Sorry fori(P,j)** = L[Ri,j]>0 & Sadnessj(P)

Their initial intensity is computed according to the following equation with A the agreeableness value from the OCEAN model.

I[Emi(P)] = I[Emj(P)] x L[Ri,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:

- **Resentmenti(P,j)** = L[Ri,j]<0 & Joyj(P)
- **Gloatingi(P,j)** = L[Ri,j]<0 & Sadnessj(P)

Their initial intensity is computed according to the following equation. This equation can be seen as the inverse of Equation (**??**), 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[Emi(P)] = I[Emj(P)] x |L[Ri,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:

- **Pridei(Pi,i)** = Beliefi(Pi) & Ideali(Pi) & V[Ideali(Pi)]>0
- **Shamei(Pi,i)** = Beliefi(Pi) & Ideali(Pi) & V[Ideali(Pi)]<0
- **Admirationi(Pj,j)** = Beliefi(Pj) & Ideali(Pj) & V[Ideali(Pj)]>0
- **Reproachi(Pj,j)** = Beliefi(Pj) & Ideali(Pj) & V[Ideali(Pj)]<0

Their initial intensity is computed according to the following equation with O the openness value from the OCEAN model:

I[Emi(P)] = V[Beliefi(P)] x |V[Ideali(P)]| x (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:

- **Gratificationi(Pi,i)** = Pridei(Pi,i) & Joyi(Pi)
- **Remorsei(Pi,i)** = Shamei(Pi,i) & Sadnessi(Pi)
- **Gratitudei(Pj,j)** = Admirationi(Pj,j) & Joyi(Pj)
- **Angeri(Pj,j)** = Reproachi(Pj,j) & Sadnessi(Pj)

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[Emi(P)] = I[Em'i(P)] x I[Em"i(P)]

In order to keep the initial intensity of each emotion between 0 and 1, each equation is truncated between 0 an 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[Emi(P)] = N x I[Emi(P)] x 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 relation 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** Ri,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 [**?**]. 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 aL a coefficient depending of the agent's personality, indicating the importance of emotions in the process, and which is described below.

L[Ri,j]=L[Ri,j]+|L[Ri,j]|(1-|L[Ri,j]|)S[Ri,j] + aL (1-|L[Ri,j]|)(mPos-mNeg)

aL = 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 aD a coefficient depending on the agent's personality, indicating the importance of emotions in the process.

D[Ri,j]=D[Ri,j] + aD (1-|D[Ri,j]|)(mSE-mOE)

aD = 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[Ri,j]=S[Ri,j] + S[Ri,j] x (1-S[Ri,j]) x (aS1 (sim-opp)/(NbKnow) - aS2 mNeg))

aS1 = 1-O

aS2 = 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[Ri,j]=F[Ri,j] x (1+L[Ri,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 to obligations, desires, plans and norms serves as a probability.

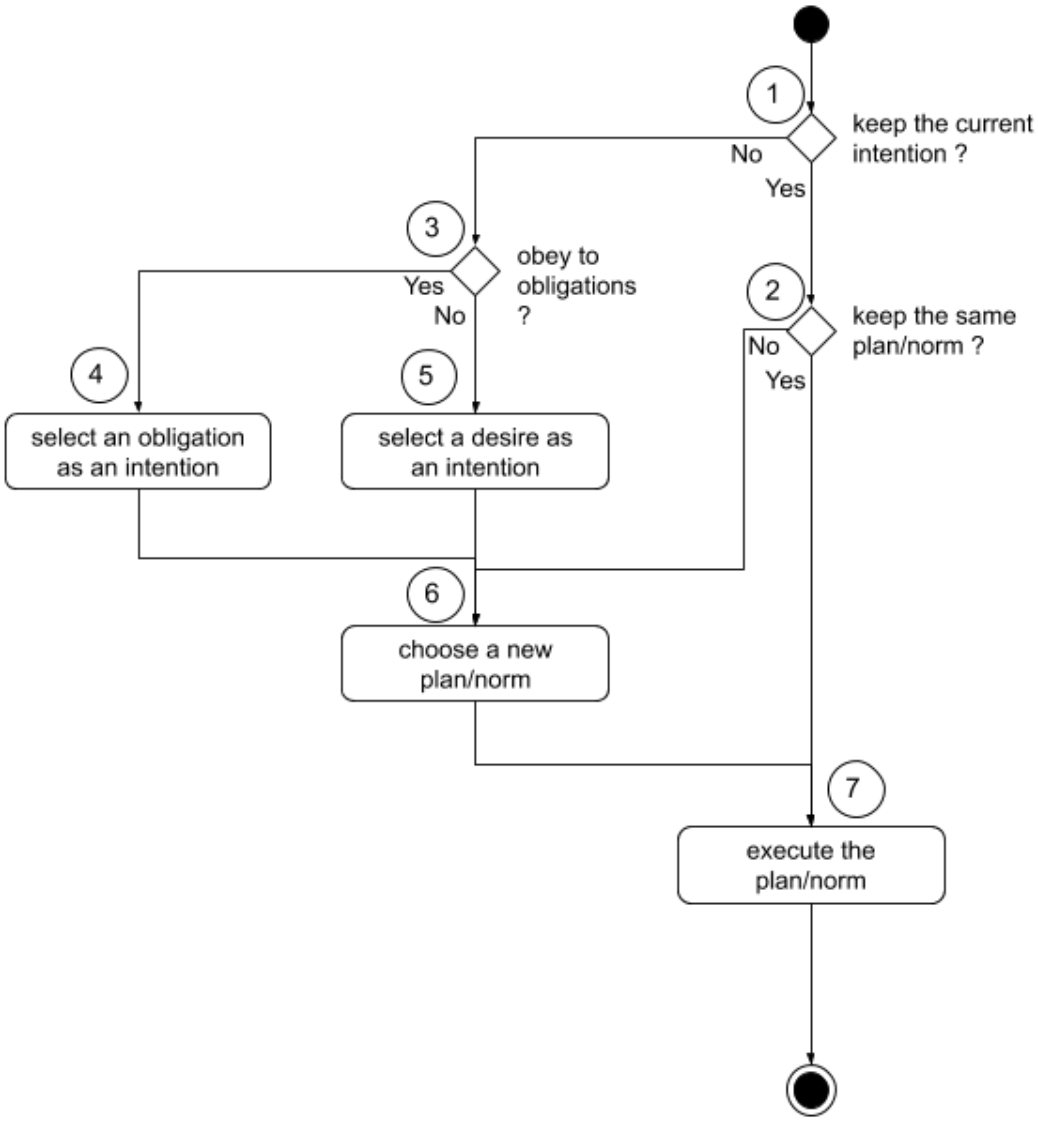


Figure 75.3: cognitive engine

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 wich 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 **Pli(Int,Cont,Pr,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 with 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 {
...
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.

In BEN, this concept of a behavior which may be disobeyed is formally represented by a norm possessed by agent i **Noi(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 instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
    each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
    mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        loop known_empty_goldmine over: get_beliefs_with_name(
    empty_mine_location) {
            ask my_friends {
                do add_belief(known_empty_goldmine);
            }
        }

        do remove_intention(share_information, true);
    }
...
}
```

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 in reducing their lifetime. When the lifetime is null, the mental state is removed from its base. The **degradation of emotions** consists in 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.

Chapter 76

Advanced Driving Skill

This page aims at presenting how to use the advanced driving skill in models.

The use of the advanced driving skill requires to use 3 skills:

- **Advanced driving skill**: dedicated to the definition of the driver species. It provides the driver agents with variables and actions allowing to move an agent on a graph network and to tune its behavior.
- **Road skill**: dedicated to the definition of roads. It provides the road agents with variables and actions allowing to registers agents on the road.
- **RoadNode skill**: dedicated to the definition of node. It provides the node agents with variables allowing to take into account the intersection of roads and the traffic signals.

Table of contents

Structure of the network: road and roadNode skills

The advanced driving skill is versatile enough to be usable with most of classic road GIS data, in particular OSM data. We use a classic format for the roads and nodes. Each road is a polyline composed of road sections (segments). Each road has a target node and a source node. Each node knows all its input and output roads. A road is considered as directed. For bidirectional roads, 2 roads have to be defined corresponding to both directions. Each road will be the **linked_road** of the other. Note that for some GIS data, only one road is defined for bidirectional roads, and the nodes are not explicitly defined. In this case, it is very easy, using the GAML language, to create the reverse roads and the corresponding nodes (it only requires few lines of GAML).

images/roads_structure.PNG

A lane can be composed of several lanes and the vehicles will be able to change at any time its lane. Another property of the road that will be taken into account is the maximal authorized speed on it. Note that even if the user of the plug-in has no information about these values for some of the roads (the OSM data are often incomplete), it is very easy using the GAML language to fill the missing value by a default value. It is also possible to change these values dynamically during the simulation (for example, to take into account that after an accident, a lane of a road is closed or that the speed of a road is decreased by the authorities).

images/roads.PNG

The **road skill** provides the road agents with several variables that will define the road properties:

- **lanes**: integer, number of lanes.
- **maxspeed**: float; maximal authorized speed on the road.
- **linked_road**: road agent; reverse road (if there is one).
- **source_node**: node agent; source node of the road.
- **target_node**: node agent; target node of the road.

It provides as well the road agents with one read only variable:$

- **agents_on**: list of list (of driver agents); for each lane, the list of driver agents on the road.

The **roadNode skill** provides the road agents with several variables that will define the road properties:

- **roads_in**: list of road agents; the list of road agents that have this node for target node.
- **roads_out**: list of road agents; the list of road agents that have this node for source node.
- **stop**: list of list of road agents; list of stop signals, and for each stop signal, the list of concerned roads.**

It provides as well the road agents with one read only variable:

- **block**: map: key: driver agent, value: list of road agents; the list of driver agents blocking the node, and for each agent, the list of concerned roads.

Advanced driving skill

Each driver agent has a planned trajectory that consists in a succession of edges. When the driver agent enters a new edge, it first chooses its lane according to the traffic density, with a bias for the rightmost lane. The movement on an edge is inspired by the Intelligent Driver Model. The drivers have the possibility to change their lane at any time (and not only when entering a new edge).

The **advanced driving skill** provides the driver agents with several variables that will define the car properties and the personality of the driver:

- **final_target**: point; final location that the agent wants to reach (its goal).
- **vehicle_length**: float; length of the vehicle.
- **max_acceleration**: float; maximal acceleration of the vehicle.
- **max_speed**: float; maximal speed of the vehicle.
- **right_side_driving**: boolean; do drivers drive on the right side of the road?
- **speed_coef**: float; coefficient that defines if the driver will try to drive above or below the speed limits.
- **security_distance_coeff**: float; coefficient for the security distance. The security distance will depend on the driver speed and on this coefficient.
- **proba_lane_change_up**: float; probability to change lane to a upper lane if necessary (and if possible).

- **proba_lane_change_down**: float; probability to change lane to a lower lane if necessary (and if possible).
- **proba_use_linked_road**: float; probability to take the reverse road if necessary (if there is a reverse road).
- **proba_respect_priorities**: float; probability to respect left/right (according to the driving side) priority at intersections.
- **proba_respect_stops**: list of float; probabilities to respect each type of stop signals (traffic light, stop sign...).
- **proba_block_node**: float; probability to accept to block the intersecting roads to enter a new road.

It provides as well the driver agents with several read only variables:

- **speed**: float; speed expected according to the road **max_value**, the car properties, the personality of the driver and its **real_speed**.
- **real_speed**: float; real speed of the car (that takes into account the other drivers and the traffic signals).
- **current_path**: path (list of roads to follow); the path that the agent is currently following.
- **current_target**: point; the next target to reach (sub-goal). It corresponds to a node.
- **targets**: list of points; list of locations (sub-goals) to reach the final target.
- **current_index**: integer; the index of the current goal the agent has to reach.
- **on_linked_road**: boolean; is the agent on the linked road?

Of course, the values of these variables can be modified at any time during the simulation. For example, the probability to take a reverse road (**proba_use_linked_road**) can be increased if the driver is stucked for several minutes behind a slow vehicle.

In addition, the advanced driving skill provides the driver agents with several actions:

- **compute_path**: arguments: a graph and a target node. This action computes from a graph the shortest path to reach a given node.
- **drive**: no argument. This action moves the driver on its current path according to the traffic condition and the driver properties (vehicle properties and driver personality).

the **drive** action works as follow: while the agent has the time to move (**remaining_time > 0**), it first defines the speed expected. This speed is computed from the **max_speed** of the road, the current **real_speed**, the **max_speed**, the **max_acceleration** and the **speed_coef** of the driver (see Equation 1).

```
speed_driver = Min(max_speed_driver, Min(real_speed_driver +
   max_acceleration_driver,max_speed_road * speed_coef_driver))
```

Then, the agent moves toward the current target and compute the remaining time. During the movement, the agents can change lanes (see below). If the agent reaches its final target, it stops; if it reaches its current target (that is not the final target), it tests if it can cross the intersection to reach the next road of the current path. If it is possible, it defines its new target (target node of the next road) and continues to move.

images/drive_action.png

The function that defines if the agent crosses or not the intersection to continue to move works as follow: first, it tests if the road is blocked by a driver at the intersection (if the road is blocked, the agent does not cross the intersection). Then, if there is at least one stop signal at the intersection (traffic signal, stop sign. . .), for each of these signals, the agent tests its probability to respect or not the signal (note that the agent has a specific probability to respect each type of signals). If there is no stopping signal or if the agent does not respect it, the agent checks if there is at least one vehicle coming from a right (or left if the agent drives on the left side) road at a distance lower than its security distance. If there is one, it tests its probability to respect this priority. If there is no vehicle from the right roads or if it chooses to do not respect the right priority, it tests if it is possible to cross the intersection to its target road without blocking the intersection (i.e. if there is enough space in the target road). If it can cross the intersection, it crosses it; otherwise, it tests its probability to block the node: if the agent decides nevertheless to cross the intersection, then the perpendicular roads will be blocked at the intersection level (these roads will be unblocked when the agent is going to move).

images/stop_at_intersection.png

Concerning the movement of the driver agents on the current road, the agent moves from a section of the road (i.e. segment composing the polyline) to another section according to the maximal distance that the agent can moves (that will depend on the remaining time). For each road section, the agent first computes the maximal distance it can travel according the remaining time and its speed. Then, the agent computes

its security distance according to its speed and its **security_distance_coeff**. While its remaining distance is not null, the agent computes the maximal distance it can travel (and the corresponding lane), then it moves according to this distance (and update its current lane if necessary). If the agent is not blocked by another vehicle and can reach the end of the road section, it updates its current road section and continues to move.

images/follow_driving.png

The computation of the maximal distance an agent can move on a road section consists in computing for each possible lane the maximal distance the agent can move. First, if there is a lower lane, the agent tests the probability to change its lane to a lower one. If it decides to test the lower lane, the agent computes the distance to the next vehicle on this lane and memorizes it. If this distance corresponds to the maximal distance it can travel, it chooses this lane; otherwise it computes the distance to the next vehicle on its current lane and memorizes it if it is higher than the current memorized maximal distance. Then if the memorized distance is lower than the maximal distance the agent can travel and if there is an upper lane, the agents tests the probability to change its lane to a upper one. If it decides to test the upper lane, the agent computes the distance to the next vehicle on this lane and memorizes it if it is higher than the current memorized maximal distance. At last, if the memorized distance is still lower than the maximal distance it can travel, if the agent is on the highest lane and if there is a reverse road, the agent tests the probability to use the reverse road (linked road). If it decides to use the reverse road, the agent computes the distance to the next vehicle on the lane 0 of this road and memorizes the distance if it is higher than the current memorized maximal distance.

images/define_max_dist.png

Application example

We propose a simple model to illustrate the driving skill. We define a driver species. When a driver agent reaches its destination, it just chooses a new random final target. In the same way, we did not define any specific behavior to avoid traffic jam for the driver agents: once they compute their path (all the driver agents use for that the same road graph with the same weights), they never re-compute it even if they are stucked in a traffic jam. Concerning the traffic signals, we just consider the traffic lights (without any pre-processing: we consider the raw OSM data). One step of the simulation represents 1 second. At last, in order to clarify the explanation of the

model, we chose to do not present the parts of the GAML code that concern the simulation visualization.

images//sim_snapshot.png

The following code shows the definition of species to represent the road infrastructure:

```
species road skills: [skill_road] {
  string oneway;
}

species node skills: [skill_road_node] {
  bool is_traffic_signal;
  int time_to_change <- 100;
  int counter <- rnd (time_to_change) ;

  reflex dynamic when: is_traffic_signal {
    counter <- counter + 1;
    if (counter >= time_to_change) {
      counter <- 0;
      stop[0] <-empty(stop[0])? roads_in : [];
    }
  }
}
```

In order to use our driving skill, we just have to add the **skill_road_node** to the **node** species and the **skill_road** to the **road** species. In addition, we added to the road species a variable called **oneway** that will be initialized from the OSM data and that represents the traffic direction (see the OSM map features for more details). Concerning the node, we defined 3 new attributes:

- **is_traffic_signal**: boolean; is the node a traffic light?
- **time_to_change**: integer; represents for the traffic lights the time to pass from the red light to the green light (and vice versa).
- **counter**: integer; number of simulation steps since the last change of light color (used by the traffic light nodes).

In addition, we defined for the **node** species a reflex (behavior) called **dynamic** that will be activated only for traffic light nodes and that will increment the **counter** value. If this counter is higher than **time_to_change**, this variable is set to 0, and the node change the value of the **stop** variable: if the traffic light was green (i.e. there is no road concerns by this stop sign), the list of block roads is set by all the roads

that enter the node; if the traffic light was red (i.e. there is at least one road concerns by this stop sign), the list of block roads is set to an empty list.

The following code shows the definition of driver species:

```
species driver skills: [advanced_driving] {
  reflex time_to_go when: final_target = nil {
    current_path <- compute_path(
        graph: road_network, target: one_of(node));
  }
  reflex move when: final_target != nil {
    do drive;
  }
}
```

In order to use our driving plug-in, we just have to add the **advanced_driving** to the **driver** species. For this species, we defined two reflexes:

- **time_to_go**: activated when the agent has no final target. In this reflex, the agent will randomly choose one of the nodes as its final target, and computed the path to reach this target using the
- **road_network** graph. Note that it will have been possible to take into account the knowledge that each agent has concerning the road network by defining a new variable of type map (dictionary) containing for each road a given weight that will reflect the driver knowledge concerning the network (for example, the known traffic jams, its favorite roads....) and to use this map for the path computation.
- **move**: activated when the agent has a final target. In this reflex, the agent will drive in direction of its final target.

We describe in the following code how we initialize the simulation:

```
init {
  create node from: file("nodes.shp") with:[
    is_traffic_signal::read("type")="traffic_signals"];

  create road from: file("roads.shp")
    with:[lanes::int(read("lanes")),
    maxspeed::float(read("maxspeed")),
    oneway::string(read("oneway"))]
    {
      switch oneway {
```

```
        match "no" {
          create road {
            lanes <- myself.lanes;
            shape <- polyline(reverse
              (myself.shape.points));
            maxspeed <- myself.maxspeed;
            linked_road <- myself;
            myself.linked_road <- self;
          }
        }
        match "-1" {
          shape <- polyline(reverse(shape.points));
        }
      }
    }
  }
  map general_speed_map <-  road as_map
    (each::(each.shape.perimeter/(each.maxspeed)));

  road_network <-  (as_driving_graph(road, node))
     with_weights general_speed_map;

  create driver number: 10000 {
    location <- one_of(node).location;
    vehicle_length <- 3.0;
    max_acceleration <- 0.5 + rnd(500) / 1000;
    speed_coeff <- 1.2 - (rnd(400) / 1000);
    right_side_driving <- true;
    proba_lane_change_up <- rnd(500) / 500;
    proba_lane_change_down <- 0.5+ (rnd(250) / 500);
    security_distance_coeff <- 3 - rnd(2000) / 1000);
    proba_respect_priorities <- 1.0 - rnd(200/1000);
    proba_respect_stops <- [1.0 - rnd(2) / 1000];
    proba_block_node <- rnd(3) / 1000;
    proba_use_linked_road <- rnd(10) / 1000;
  }
}
```

In this code, we create the node agents from the node shapefile (while reading the attributes contained in the shapefile), then we create in the same way the road agents. However, for the road agents, we use the **oneway** variable to define if we should or not reverse their geometry (**oneway** = "-1") or create a reverse road (**oneway** = "no"). Then, from the road and node agents, we create a graph (while taking into account the **maxspeed** of the road for the weights of the edges). This graph is

the one that will be used by all agents to compute their path to their final target. Finally, we create 1000 driver agents. At initialization, they are randomly placed on the nodes; their vehicle has a length of 3m; the maximal acceleration of their vehicle is randomly drawn between 0.5 and 1; the speed coefficient of the driver is randomly drawn between 0.8 and 1.2; they are driving on the right side of the road; their probability of changing lane for a upper lane is randomly drawn between 0 and 1.0; their probability of changing lane for a lower lane is randomly drawn between 0.5 and 1.0; the security distance coefficient is randomly drawn between 1 and 3; their probability to respect priorities is randomly drawn between 0.8 and 1; their probability to respect light signal is randomly drawn between 0.998 and 1; their probability to block a node is randomly drawn between 0 and 0.003; their probability to use the reverse road is randomly drawn between 0 and 0.01;

The complete code of the model with the data can be found here

Chapter 77

Manipulate Dates

Managing Time in Models

If some models are based on a abstract time - only the number of cycles is important - others are based on a real time. In order to manage the time, GAMA provides some tools to manage time.

First, GAMA allows to define the duration of a simulation step. It provides access to different time variables. At last, since GAMA 1.7, it provides a date variable type and some global variables allowing to use a real calendar to manage time.

Definition of the step and use of temporal unity values

GAMA provides three important global variables to manage time:

- `cycle` (int - not modifiable): the current simulation step - this variable is incremented by 1 at each simulation step
- **`step`** (float - can be modified): the duration of a simulation step (in seconds). By default the duration is one second.
- `time` (float - not modifiable): the current time spent since the beginning of the simulation - this variable is computed at each simulation step by: time = cycle * step.

The value of the cycle and time variables are shown in the top left (green rectangle) of the simulation interface. Clicking on the green rectangle allows to display either the number cycles or the time variable. Concerning this variable, it is presented following a years - month - days - hours - minutes - seconds format. In this presentation, every months are considered as being composed of 30 days (the different number of days of months are not taken into account).

Concerning the step facet, the variable can be modified by the modeler. A classic way of doing it consists in reediting the variable in the global section:

```
global {
        float step <- 1 #hour;
}
```

In this example, each simulation step will represent 1 hour. This time will be taken into account for all actions based on time (e.g. moving actions).

Note that the value of the step variable should be given in seconds. To facilitate the definition of the step value and of all expressions based on time, GAMA provides different built-in constant variables accessible with the "`#`" symbol:

- `#s` : second - 1 second
- `#mn` : minute - 60 seconds
- `#hour` : hour - 60 minutes - 3600 seconds
- `#day` : day - 24 hours - 86400 seconds
- `#month` : month - 30 days - 2592000 seconds
- `#year` : year - 12 month - 3.1104E7

The date variable type and the use of a real calendar

Since GAMA 1.7, it is possible to use a real calendar to manage the time. For that, the modeler have just to define the starting date of the simulation. This variable is of type date which allow to represent a date and time. A date variable has several attributes:

- `year` (int): the year component of the date
- `month` (int): the month component of the date

- `day` (int): the day component of the date
- `hour` (int): the hour component of the date
- `minute` (int): the minute component of the date
- `second` (int): the second component of the date
- `day_of_week` (int): the day of the week
- `week_of_year` (int): the week of the year

Several ways can be used to define a date. The simplest consists in using a list of int values: [year,month of the year,day of the month, hour of the day, minute of the hour, second of the minute]

```
date my_date <- date([2010,3,23,17,30,10]); // the 23th of March 2010,
    at 17:30:10
```

Another way consists in using a string with the good format:

```
date my_date <- date("2010-3-23T17:30:10+07:00");
```

Note that the current date can be access through the #now built-in variable (variable of type date).

In addition, GAMA provides different useful operators working on dates. For instance, it is possible to compute the duration in seconds between 2 dates using the "-" operator. The result is given in seconds:

```
float d <- starting_date - my_date;
```

It is also possible to add or subtract a duration (in seconds) to a date:

```
write "my_date + 10: " + (my_date + 10);
write "my_date - 10: " + (my_date - 10);
```

At last, it is possible to add or subtract a duration (in years, months, weeks, days, hours, minutes, seconds) to a date:

```
write "my_date add_years 1: " + (my_date add_years 1);
write "my_date add_months 1: " + (my_date add_months 1);
write "my_date add_weeks 1: " + (my_date add_weeks 1);
write "my_date add_days 1: " + (my_date add_days 1);
write "my_date add_hours 1: " + (my_date add_hours 1);
write "my_date add_minutes 1: " + (my_date add_minutes 1);
write "my_date add_seconds 1: " + (my_date add_seconds 1);
```

```
write "my_date subtract_years 1: " + (my_date subtract_years 1);
write "my_date subtract_months 1: " + (my_date subtract_months 1);
write "my_date subtract_weeks 1: " + (my_date subtract_weeks 1);
write "my_date subtract_days 1: " + (my_date subtract_days 1);
write "my_date subtract_hours 1: " + (my_date subtract_hours 1);
write "my_date subtract_minutes 1: " + (my_date subtract_minutes 1);
write "my_date subtract_seconds 1: " + (my_date subtract_seconds 1);
```

For the modelers, two global date variable are available:

- `starting_date`: date considered as the beginning of the simulation
- `current_date`: current date of the simulation

By default, these variables are nil. Defining a value of the starting_date allows to change the normal time management of the simulation by a more realistic one (using calendar):

```
global {
     date starting_date <- date([1979,12,17,19,45,10]);
}
```

When a variable is set to this variable, the current_date variable is automatically initialized with the same value. However, at each simulation step, the current_date variable is incremented by the step variable. The value of the current_date will replace the value of the time variable in the top left green panel.

Note that you have to be careful, when a real calendar is used, the built-in constants `#month` and `#year` should not be used as there are not consistent with the calendar (where month can be composed of 28, 29, 30 or 31 days).

Chapter 78

Implementing light

When using opengl display, GAMA provides you the possibility to manipulate one or several lights, making your display more realistic. Most of the following screenshots will be taken with the following short example gaml :

```
model test_light

grid cells {
    aspect base {
        draw square(1) at:{grid_x,grid_y} color:#white;
    }
}
experiment my_experiment type:gui{
    output {
        display my_display type:opengl background:#darkblue {
            species cells aspect:base;
            graphics "my_layer" {
                draw square(100) color:#white at:{50,50};
                draw cube(5) color:#lightgrey at:{50,30};
                draw cube(5) color:#lightgrey at:{30,35};
                draw cube(5) color:#lightgrey at:{60,35};
                draw sphere(5) color:#lightgrey at:{10,10,2.5};
                draw sphere(5) color:#lightgrey at:{20,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,30,2.5};
                draw sphere(5) color:#lightgrey at:{40,60,2.5};
                draw cone3D(5,5) color:#lightgrey at:{55,10,0};
                draw cylinder(5,5) color:#lightgrey at:{10,60,0};
            }
        }
    }
```

```
}
```

Index

Light generalities

Before going deep into the code, here is a quick explanation about how light works in opengl. First of all, you need to know that there are 3 types of lights you can manipulate : the **ambient light**, the **diffuse light** and the **specular light**. Each "light" in opengl is in fact composed of those 3 types of lights.

Ambient light

The **ambient light** is the light of your world without any lighting. If a face of a cube is not stricken by the light rays for instance, this face will appear totally black if there is no ambient light. To make your world more realistic, it is better to have an ambient light. An ambient light has then no position or direction. It is equally distributed to all the objects of your scene.

Here is an example of our GAML scene using only ambient light (color red) :

Diffuse light

The **diffuse light** can be seen as the light rays : if a face of a cube is stricken by the diffuse light, it will take the color of this diffuse light. You have to know that the more perpendicular the face of your object will be to the light ray, the more lightened the face will be.

A diffuse light has then a direction. It can have also a position. Your have 2 categories of diffuse light : the **positional lights**, and the **directional lights**.

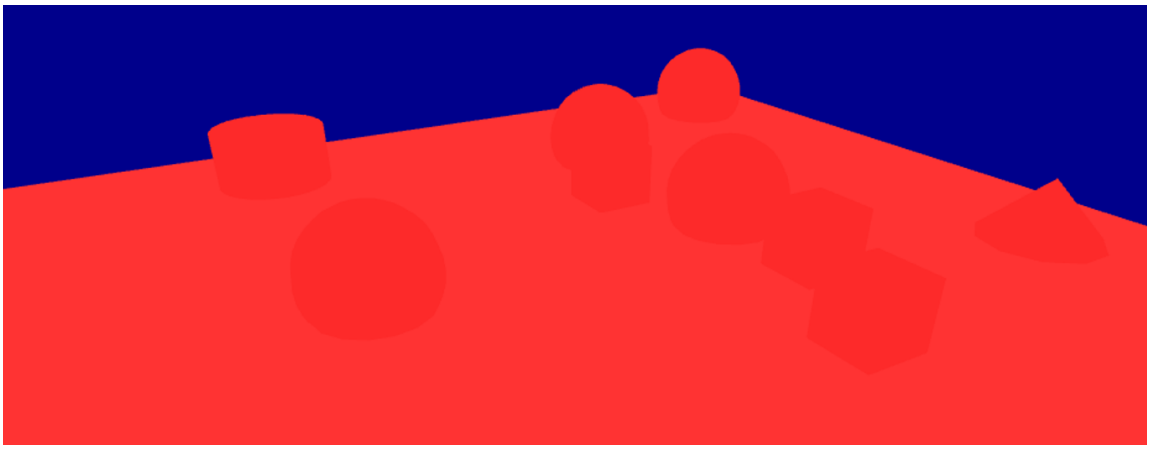


Figure 78.1: resources/images/lightRecipes/ambient_light.png

Positional lights

Those lights have a position in your world. It is the case of **point lights** and **spot lights**.

- Point lights

Points lights can be seen as a candle in your world, diffusing the light equally in all the direction.

Here is an example of our GAML scene using only diffuse light, with a point light (color red, the light source is displayed as a red sphere) :

- Spot lights

Spot lights can be seen as a torch light in your world. It needs a position, and also a direction and an angle.

Here is an example of our GAML scene using only diffusion light, with a spot light (color red, the light source is displayed as a red cone) :

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

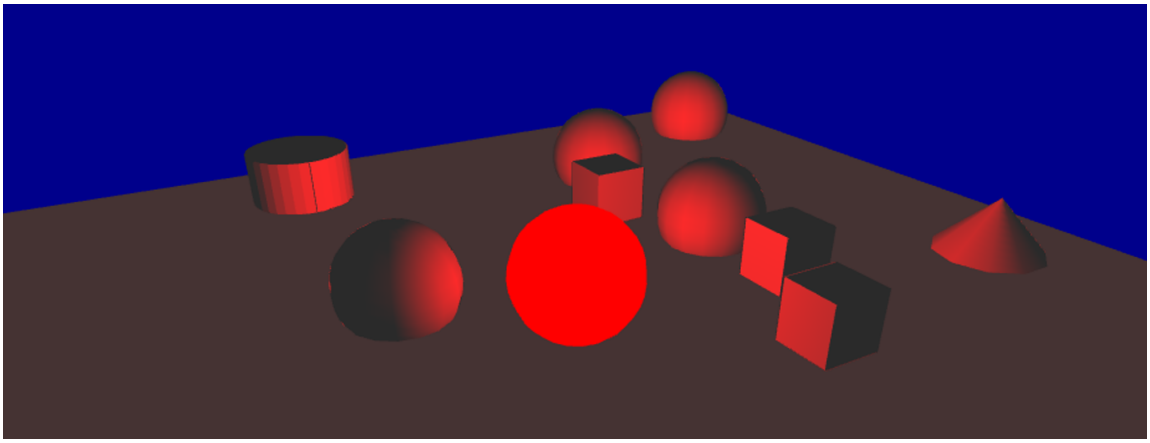


Figure 78.2: resources/images/lightRecipes/point_light.png

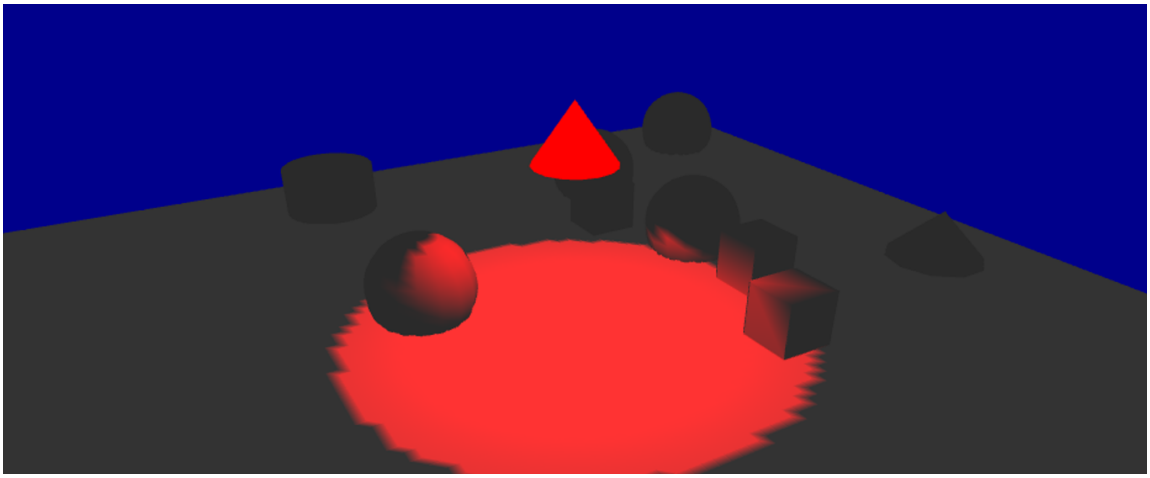


Figure 78.3: resources/images/lightRecipes/spot_light.png

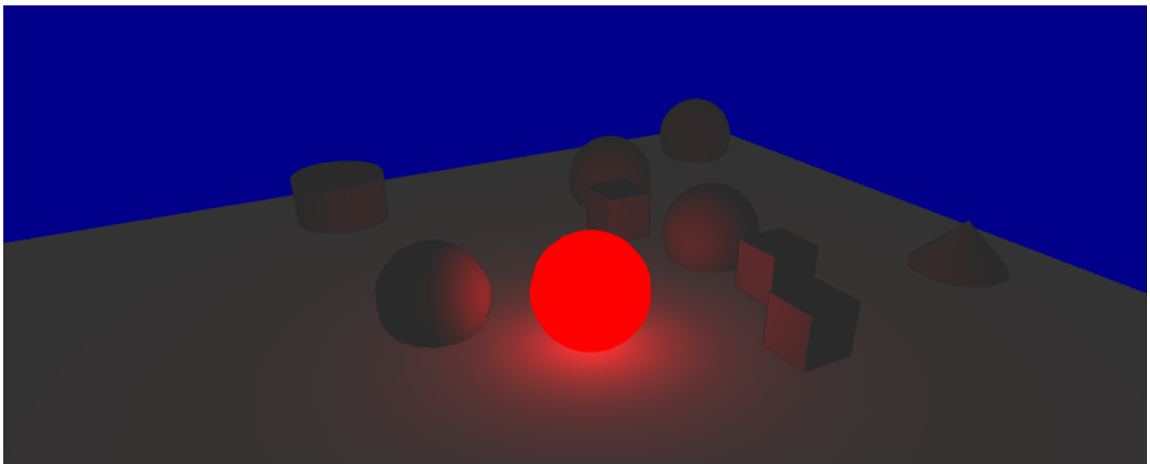


Figure 78.4: resources/images/lightRecipes/point_light_with_attenuation.png

/ (1 + constante_attenuation + linear_attenuation * d + quadratic_attenuation * d)) By changing those 3 values (constante_attenuation, linear_attenuation and quadratic_attenuation), you can control the way light is diffused over your world (if your world is "foggy" for instance, you may turn your linear and quadratic attenuation on). Note that by default, all those attenuation are equal to 0.

Here is an example of our GAML scene using only diffusion light, with a point light with linear attenuation (color red, the light source is displayed as a red sphere):

Directional lights

Directional lights have no real "position" : they only have a direction. A directional light will strike all the objects of your world with the same direction. An example of directional light you have in the real world would be the light of the sun : the sun is so far away from us that you can consider that the rays have the same direction and the same intensity wherever they strike. Since there is no position for directional lights, there is no attenuation either.

Here is an example of our GAML scene using only diffusion light, with a directional light (color red) :

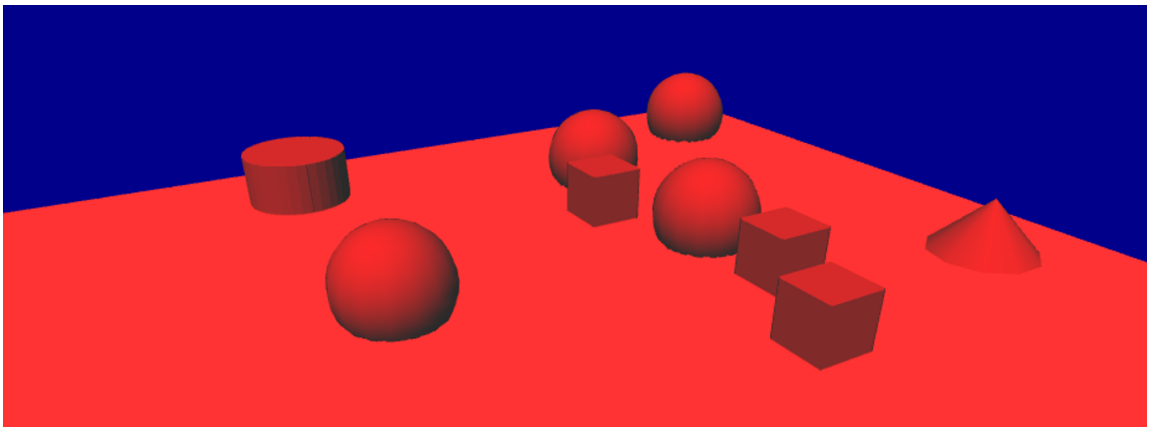


Figure 78.5: resources/images/lightRecipes/direction_light.png

Specular light

This is a more advanced concept, giving an aspect a little bit "shinny" to the objects stricken by the specular light. It is used to simulate the interaction between the light and a special material (ex : wood, steel, rubber...). This specular light is not implemented yet in gama, only the two others are.

Default light

In your opengl display, without specifying any light, you will have only one light, with those following properties :

Those values have been chosen in order to have the same visual effect in both opengl and java2D displays, when you display 2D objects, and also to have a nice "3D effect" when using the opengl displays. We chose the following setting by default:

- The ambient light value: rgb(127,127,127,255)
- diffuse light value: rgb(127,127,127,255)
- type of light: direction
- direction of the light: (0.5,0.5,-1);

Here is an example of our GAML scene using the default light:

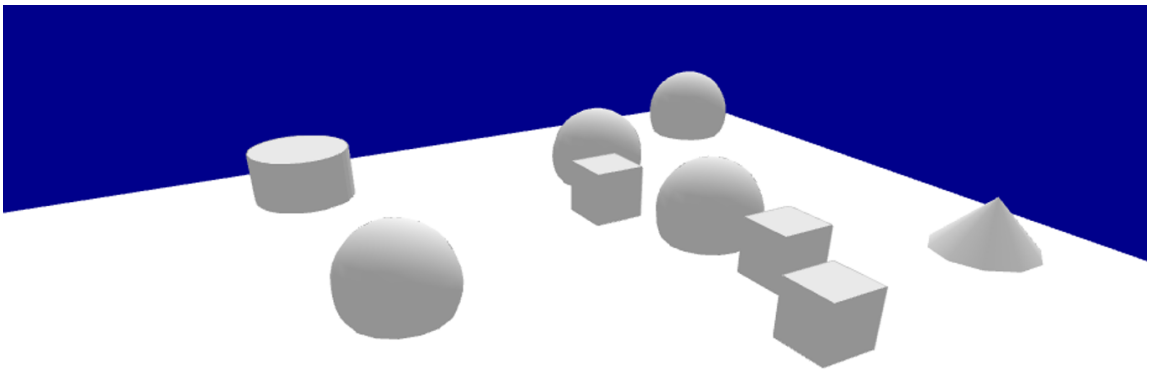


Figure 78.6: resources/images/lightRecipes/default_light.png

Custom lights

In your opengl display, you can create up to 8 lights, giving them the properties you want.

Ambient light

In order to keep it simple, the ambient light can be set directly when you are declaring your display, through the facet **ambient_light**. You will have one only ambient light.

```
experiment my_experiment type:gui {
    output {
        display "my_display" type:opengl ambient_light:100 {
        }
    }
}
```

Note for developers : Note that this ambient light is set to the GL_LIGHT0. This GL_LIGHT0 only contains an ambient light, and no either diffuse nor specular light.

Diffuse light

In order to add lights, or modifying the existing lights, you can use the statement `light`, inside your `display` scope :

```
experiment my_experiment type:gui {
    output {
        display "my_display" type:opengl {
            light id:0;
        }
    }
}
```

This statement has just one non-optional facet: the facet "id". Through this facet, you can specify which light you want. You can control 7 lights, through an integer value between 1 and 7. Once you are manipulating a light through the `light` statement, the light is turned on. To switch off the light, you have to add the facet **`active`**, and turn it to `false`. The light you are declaring through the `light` statement is in fact a "diffuse" light. You can specify the color of the diffuse light through the facet **`color`** (by default, the color will be turned to white). An other very important facet is the **`type`** facet. This facet accepts a value among **`direction`**, **`point`** and `spot`.

Declaring direction light

A direction light, as explained in the first part, is a light without any position. Instead of the facet **`position`**, you will use the facet **`direction`**, giving a 3D vector.

Example of implementation:

```
light id:1 type:direction direction:{1,1,1} color:rgb(255,0,0);
```

Declaring point light

A point light will need a facet **`position`**, in order to give the position of the light source.

Example of implementation of a basic point light:

```
light id:1 type:point position:{10,20,10} color:rgb(255,0,0);
```

You can add, if you want, a custom attenuation of the light, through the facets **`linear_attenuation`** or **`quadratic_attenuation`**.

Example of implementation of a point light with attenuation :

```
light id:1 type:point position:{10,20,10} color:rgb(255,0,0)
   linear_attenuation:0.1;
```

Declaring spot light

A spot light will need the facet **position** (a spot light is a positionnal light) and the facet **direction**. A spot light will also need a special facet **spot_angle** to determine the angle of the spot (by default, this value is set to 45 degree).

Example of implementation of a basic spot light:

```
light id:1 type:spot position:{0,0,10} direction:{1,1,1} color:rgb
   (255,0,0) spot_angle:20;
```

Same as for point light, you can specify an attenuation for a spot light.

Example of implementation of a spot light with attenuation :

```
light id:1 type:spot position:{0,0,10} direction:{1,1,1} color:rgb
   (255,0,0) spot_angle:20;
```

Note that when you are working with lights, you can display your lights through the facet `draw light` to help you implementing your model. The three types of lights are displayed differently:

- The **point** light is represented by a sphere with the color of the diffuse light you specified, in the position of your light source.
- The **spot** light is represented by a cone with the color of the diffuse light you specified, in the position of your light source, the orientation of your light source. The size of the base of the cone will depend of the angle you specified.
- The **direction** light, as it has no real position, is represented with arrows a bit above the world, with the direction of your direction light, and the color of the diffuse light you specified.

Note for developers: Note that, since the GL_LIGHT0 is already reserved for the ambient light (only !), all the other lights (from 1 to 7) are the lights from GL_LIGHT1 to GL_LIGHT7.

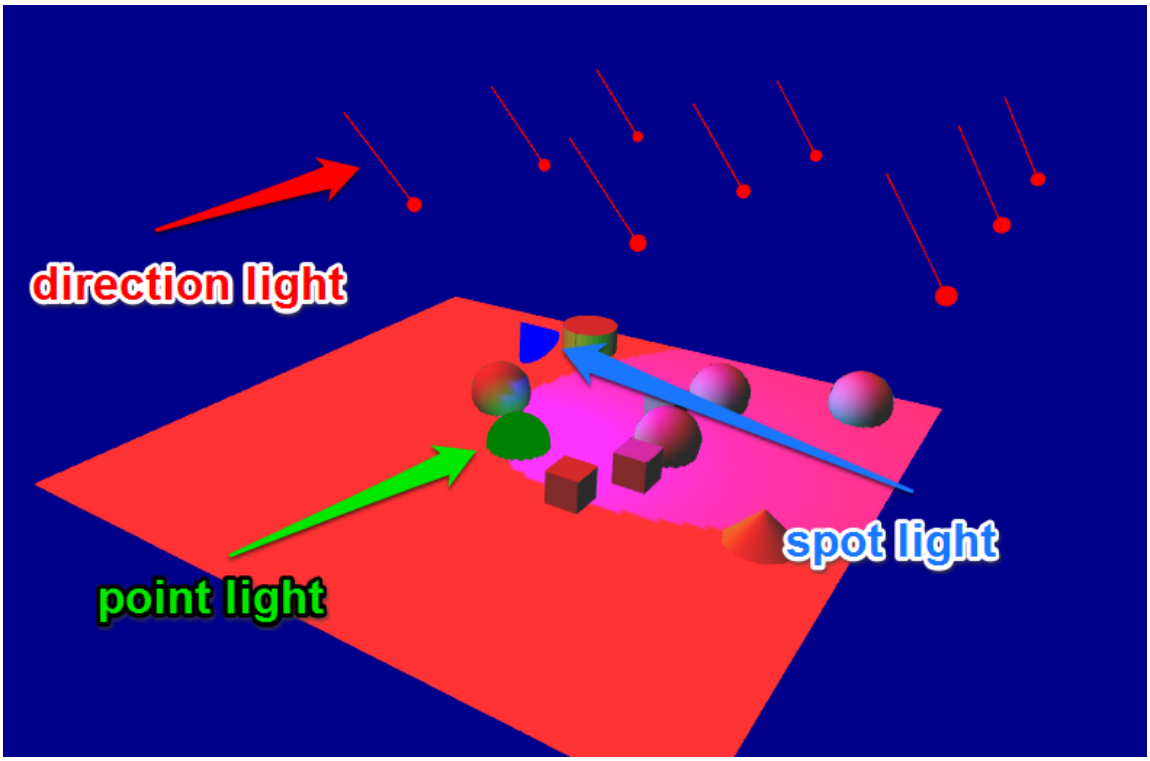


Figure 78.7: resources/images/lightRecipes/draw_light.png

Chapter 79

Using Comodel

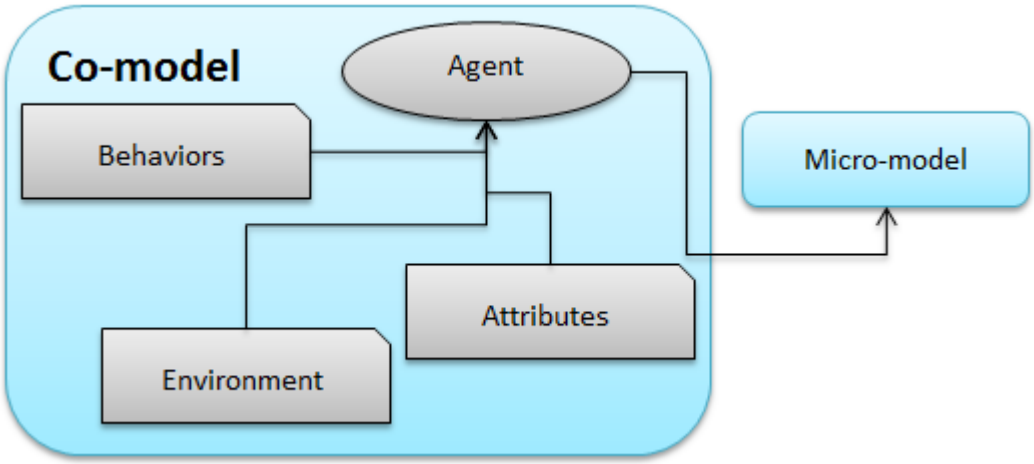
Introduction

In the trend of developing complex system of multi-disciplinary, composing and coupling models are days by days become the most attractive research objectives. GAMA is supporting the co-modelling and co-simulation which are suppose to be the common coupling infrastructure.

Example of a Comodel

A Comodel is a model, especially an multi-agent-based, compose several sub-model, called micro-model. A comodel itself could be also a micro-model of an other comodel. From the view of a micro-model, comodel is called a macro-model.

A micro-model must be import, instantiate, and life-control by macro-model.



Why and when can we use Comodel ?

to be completed...

Use of Comodel in a GAML model

The GAML language has been evolve by extend the import section. The old importation tell the compiler to merge all imported elements into as one model, but the new one allows modellers to keep the elements come from imported models separately with the caller model.

Defining a micro-model

Defining a micro-model of comodel is to import an existing model with an alias name. The syntax is:

```
import <path to the GAML model> as <identifier>
```

The identifier is then become the new name of the micro-model.

Instantiate a micro-model

After the importation and giving an identifier, micro-model must be explicitly instantiated. It could be done by create statement.

```
create <micro-model identifier> . <experiment name> [optional parameter
    ];
```

THe is an expriment inside micro-model. This syntax will generate an experiment agent and attach an implicitly simulation.

Note: Creation of multi-instant is not create multi-simulation, but multi-experiment. Modellers could create a experiment with multi-simulation by explicitly do the init inside the experiment scope.

Control micro-model life-cycle

A micro-model can be control as the normal agent by asking the correspond identifier, and also be destroy by the 'o die' statement. As fact, it can be recreate any time we need.

```
ask (<micro-model identifier> . <experiment name>  at <number> ) .
    simulation {
        ...
}
```

Visualize micro-model

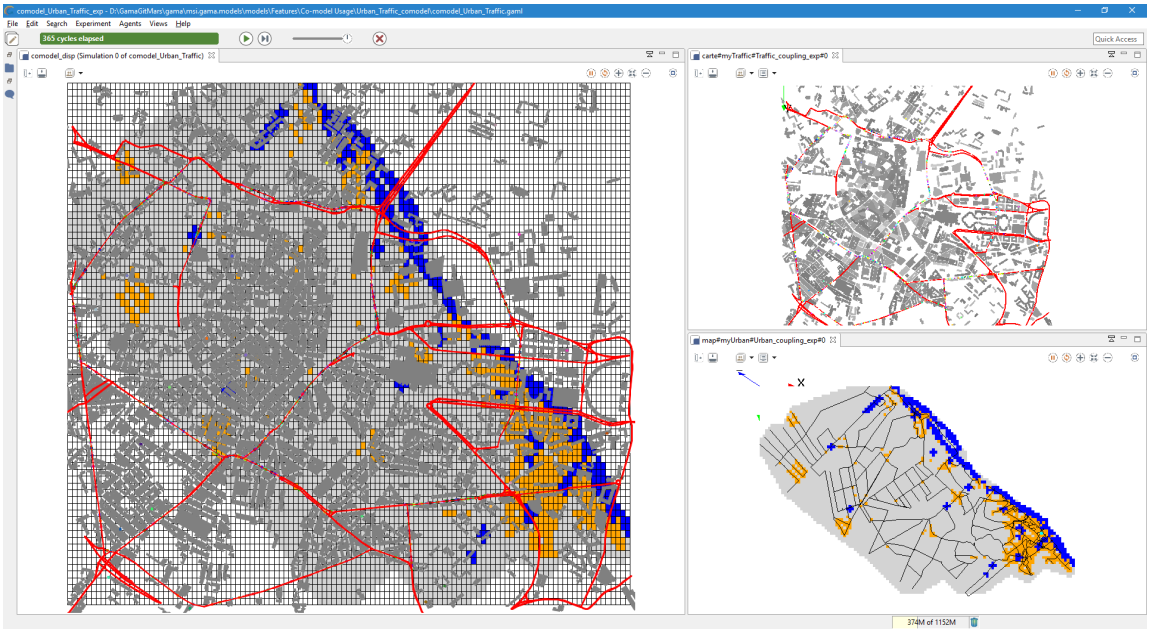
The micro-model species could display in comodel with the support of agent layer

```
agents "name of layer" value: (<micro-model> . <experiment name> at <
    number>).<get List of agents>;
```

More details

Example of the comodel

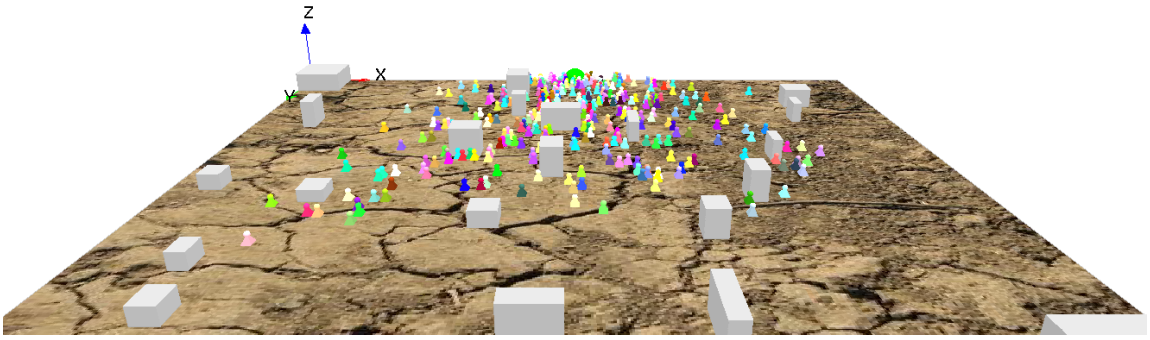
Urbanization model with Traffic model



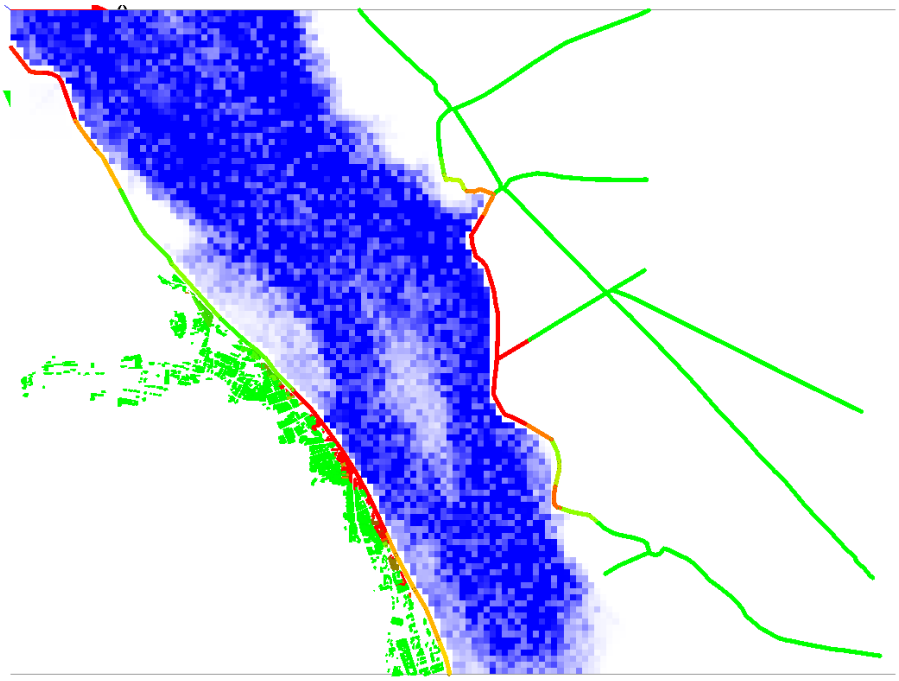
Flood model with Evacuation model

Reusing of two existing models:Flood Simulation and Evacuation.

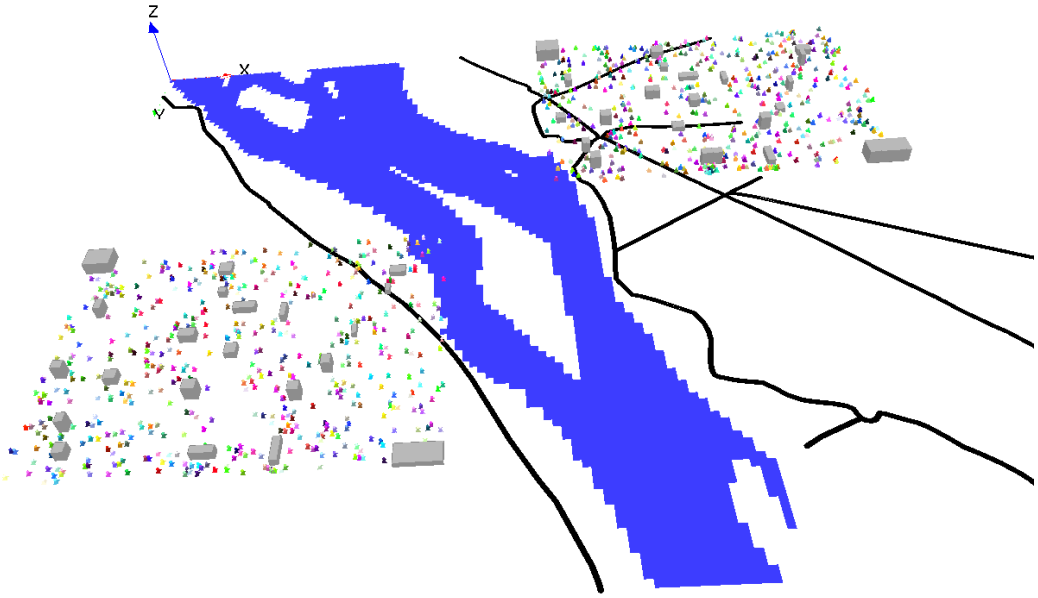
Toy Models/Evacuation/models/continuous_move.gaml



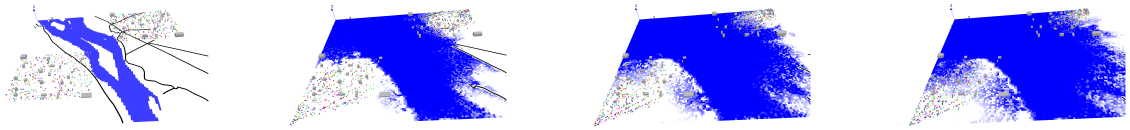
Toy Models/Flood Simulation/models/Hydrological Model.gaml



The comodel explore the effect of flood on evacuation plan:



Simulation results:



Chapter 80

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 step forward and backward in a simulation,
- The possibility to save the state of a simulation in a file and to restore a simulation from this file.

Save a simulation

```
experiment saveSimu type: gui {

    reflex store when: cycle = 5 {
        write "================ START SAVE + self " + " - " + cycle ;
        write "Save of simulation : " + saveSimulation('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;
        }
    }
}
```

Saved simulation file type: gsim

Other serialization operators

Chapter 81

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 asymetric: the simulations should contain a server and some clients to communicate. Message exchanges are made between agents through this server. 3 protocols are supported (TCP, UDP and MQTT):

- **when TCP or UDP protocols are used:** one agent of the simulation is the server and the other ones are the clients.
- **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.

Which protocol to use ?

In the GAMA network, 3 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**: this protocol can be used both to communicate between GAMA applications or between GAMA and an external application.

Disclaimer

In all the models using any network communication, the server should be launched before the clients. As a consequence, when TCP 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 possibles 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.
- **UDP**:
  - **protocol**: the 2 possibles 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)

Sending messages

To send any message, the agent has to use the `send` action:

```
    do send to: "send" contents: name + " " + cycle + " sent to server"
   ;
```

Receiving messages

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 new messages (with action `has_more_message()`) and fetch it (that is get 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;
}
```

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;
    }
}
```

Broadcasting a message to all the agents members of a given group

Each time an agent create 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 whose 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 peference page.
- examples model can be found in the GAMA model library, in: `Plugin models > Network`.

Chapter 82

Editing 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 here, with the same model as an example. 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 bit more technical to handle than the general GAMA use-case, and the following commands and code have been solely tested on a Linux Ubuntu 15.04 machine, x86_64 architecture, with kernel 3.19.0-82-generic. Java version is 1.8.0_121 (java version "1.8.0_121")

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). RECOMMENED: 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 1.7.0 V7
    *"
echo "* http://gama-platform.org
    *"
echo "* (c) 2007-2016 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 commmand, 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>
     <Outputs>
         <Output id="1" name="main_display" framerate="10" />
         <Output id="2" name="number_of_preys" framerate="1" />
         <Output id="3" name="number_of_predators" framerate="1" />
         <Output id="4" name="duration" framerate="1" />
```

```
    </Outputs>
</Simulation>
</Experiment_plan>
```

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.XXXXXXXXXXX/models/Tutorials/Predator Prey/models` (with XXXXXXXXXXX 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 `$OUPUT_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" ;
        parameter "Prey max energy: " var: prey_max_energy category: "
    Prey" ;
        parameter "Prey max transfert: " var: prey_max_transfert
    category: "Prey" ;
        parameter "Prey energy consumption: " var: prey_energy_consum
    category: "Prey" ;
        output {
            display main_display {
                grid vegetation_cell lines: #black ;
                species prey aspect: base ;
            }
            monitor "Number of preys" value: nb_preys ;
        }
    }
```

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. Theses 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 :

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 :

bash 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/tutoGama-Headless/../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.

Chapter 83

The Graphical Editor

The graphical editor that 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.

Table of contents

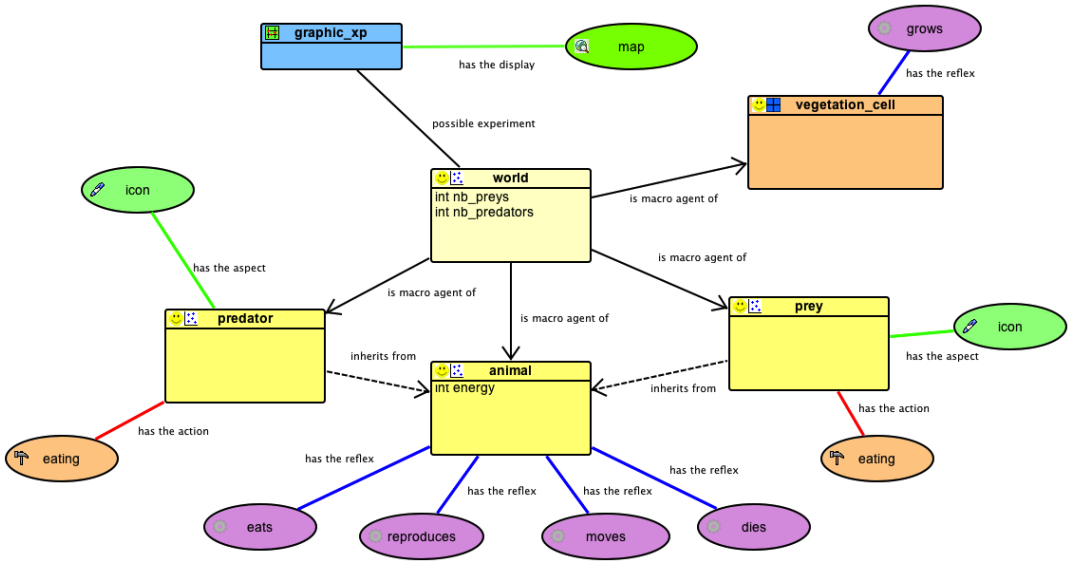


Figure 83.1: images/graphical_editor/gm_predator_prey.png

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**

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>"

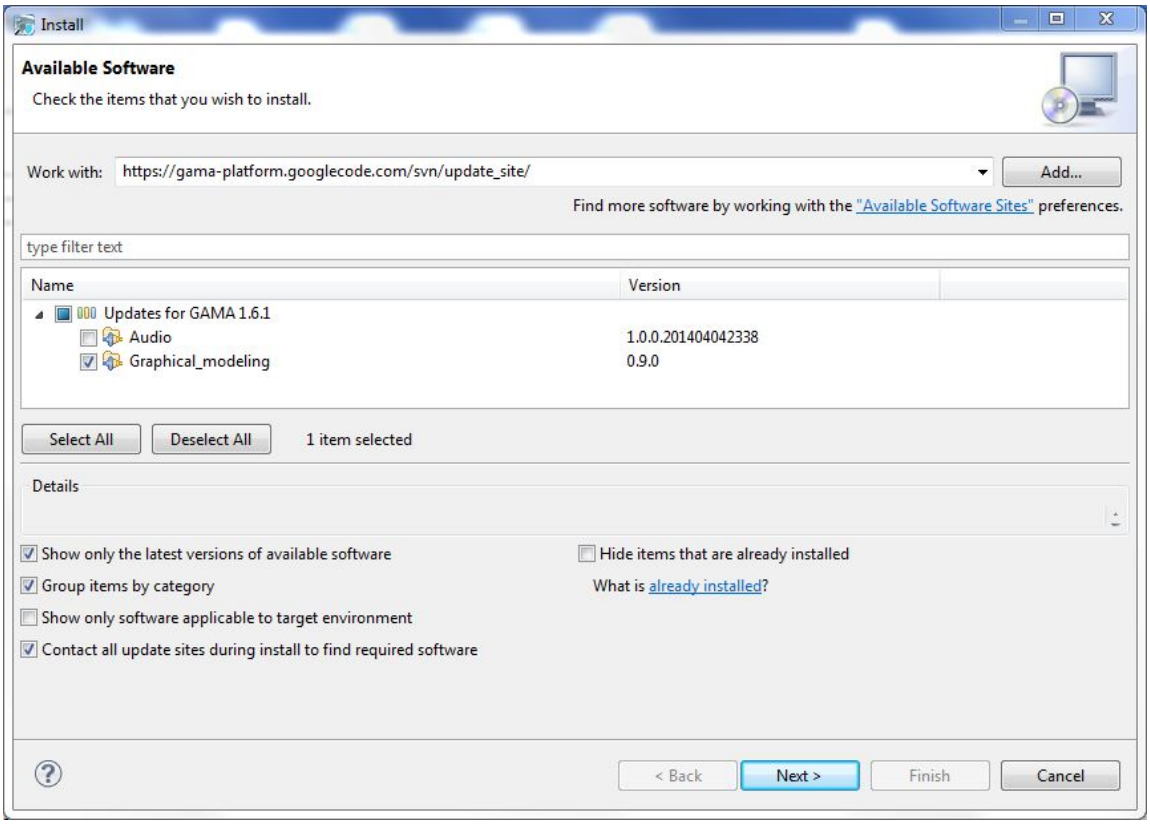


Figure 83.2: install

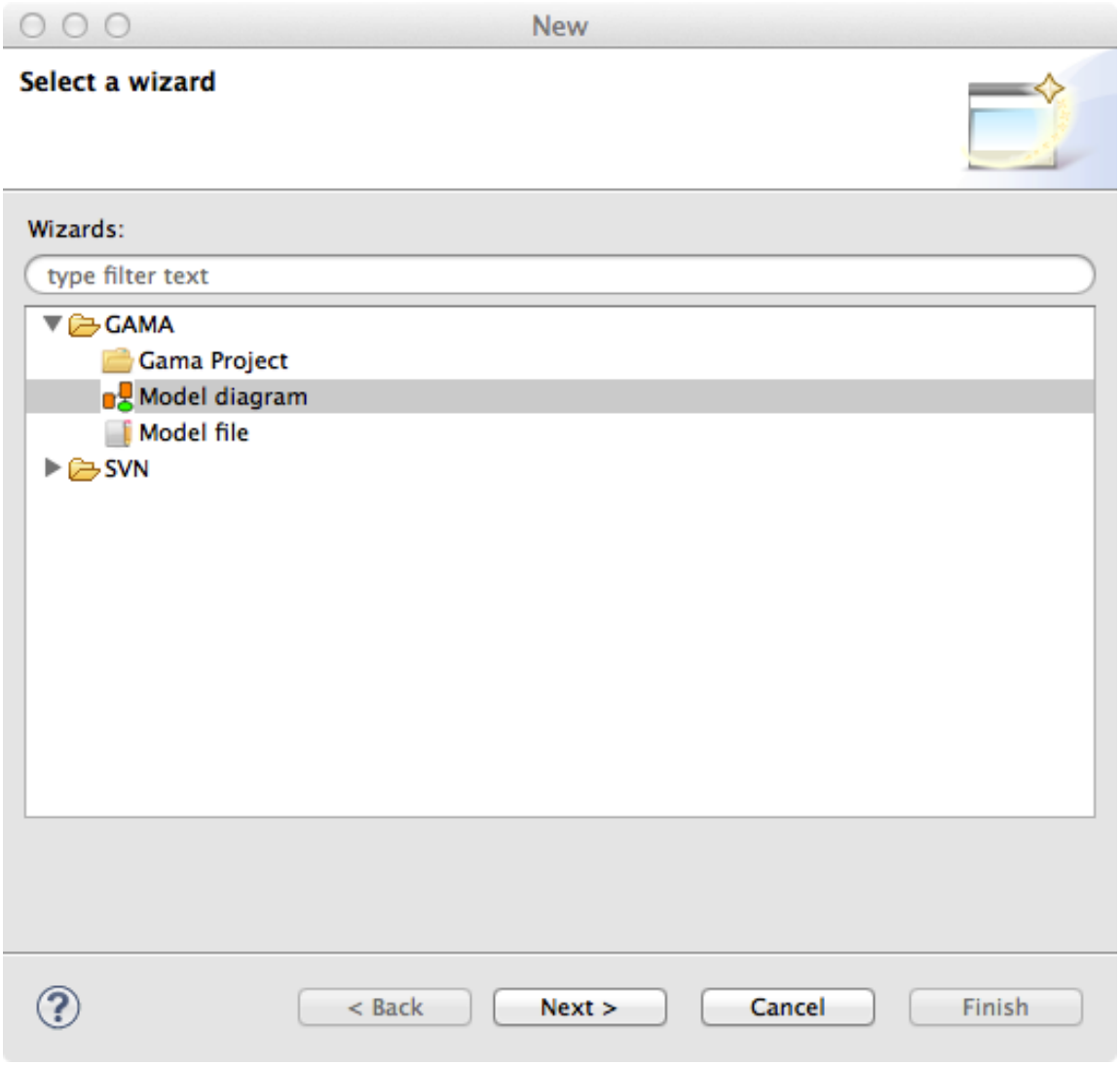


Figure 83.3: images/graphical_editor/newDiagram.png

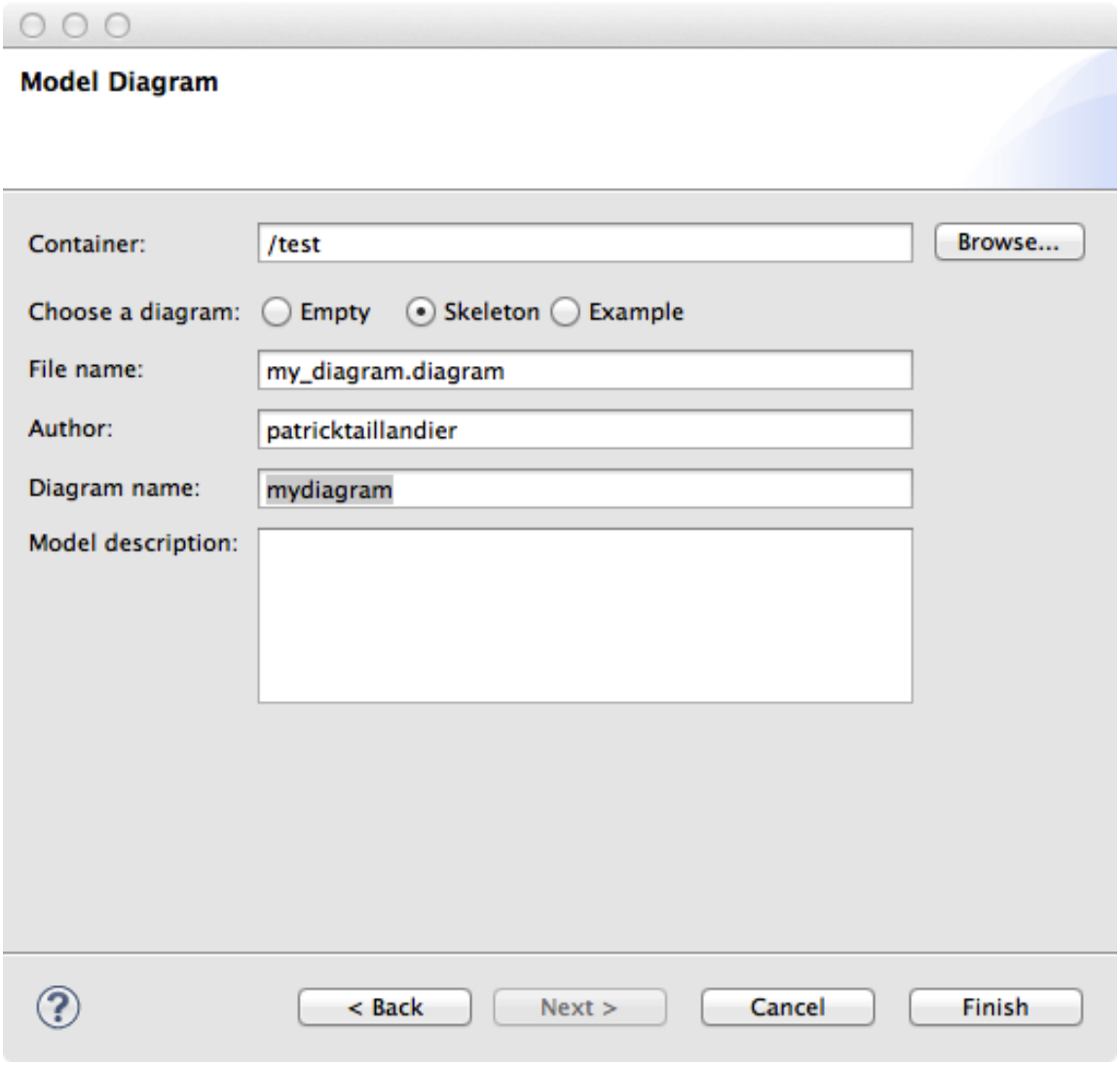


Figure 83.4: images/graphical_editor/modeldiagramNew.png

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:

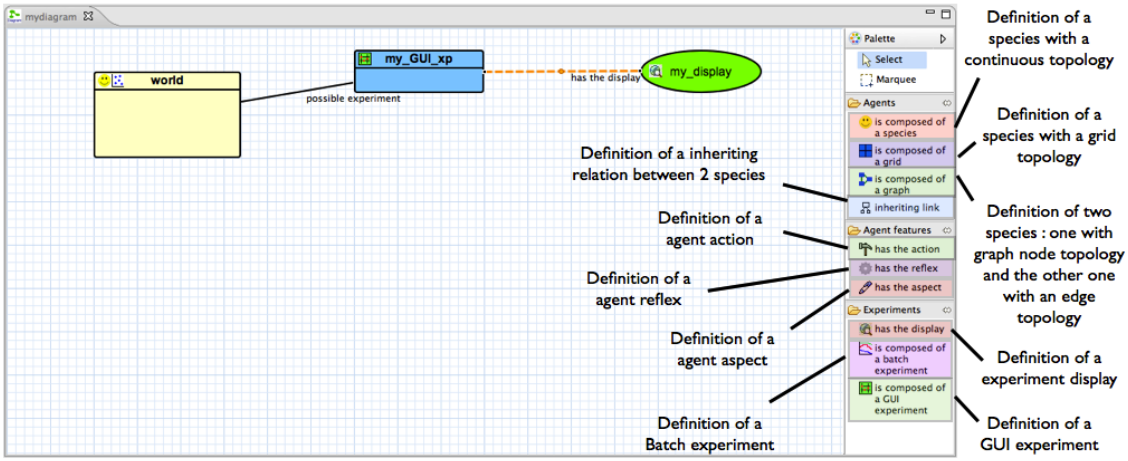


Figure 83.5: images/graphical_editor/framework.png

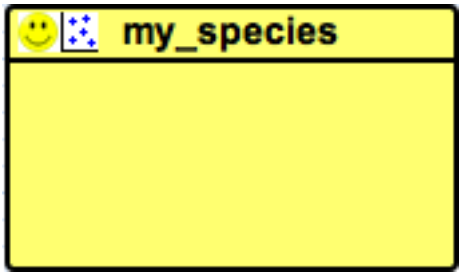


Figure 83.6: images/graphical_editor/species.png

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**: -

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.

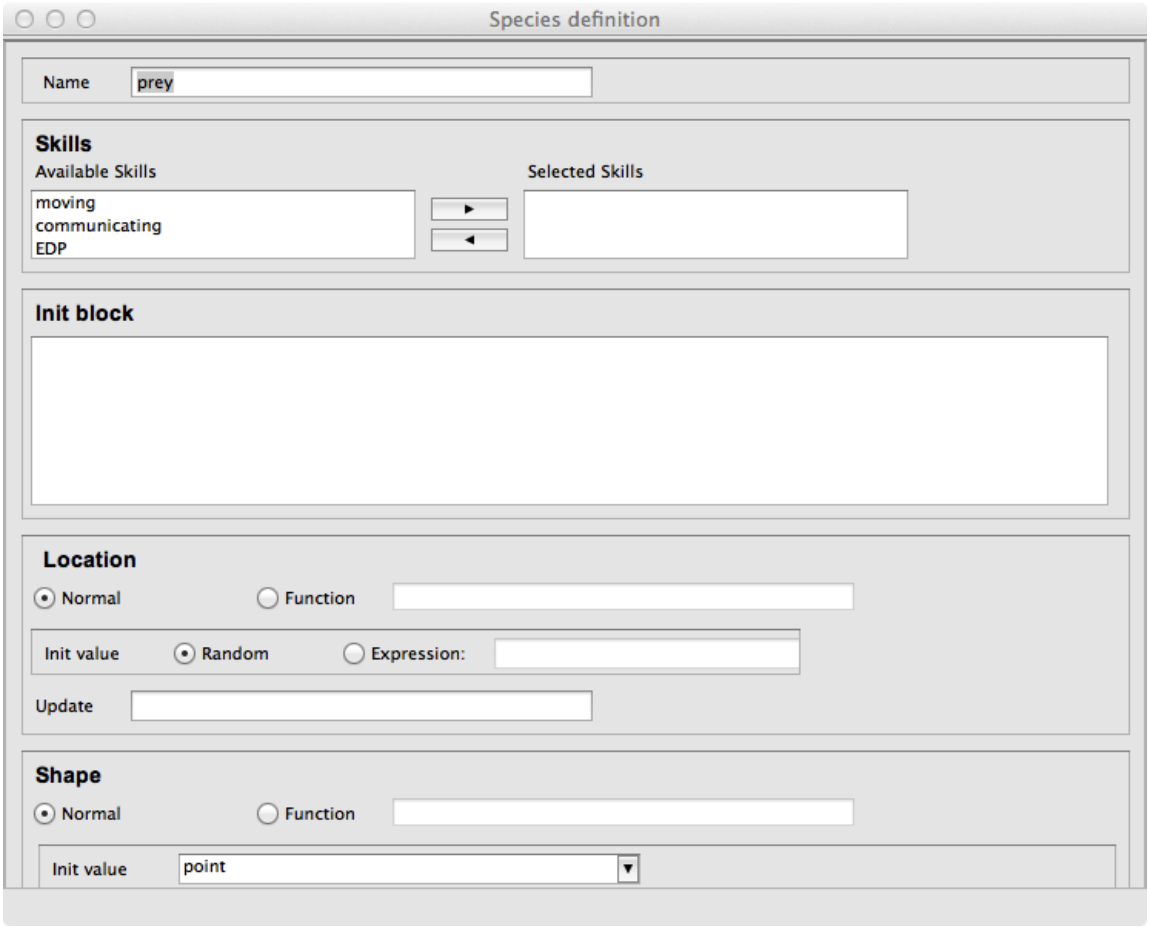


Figure 83.7: images/graphical_editor/Frame_Speciesdef1.png

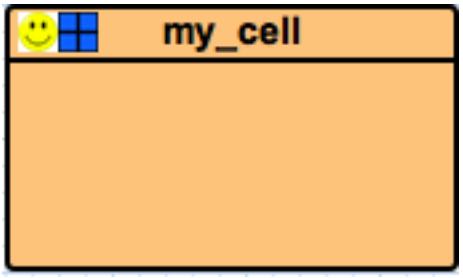


Figure 83.8: images/graphical_editor/grid.png

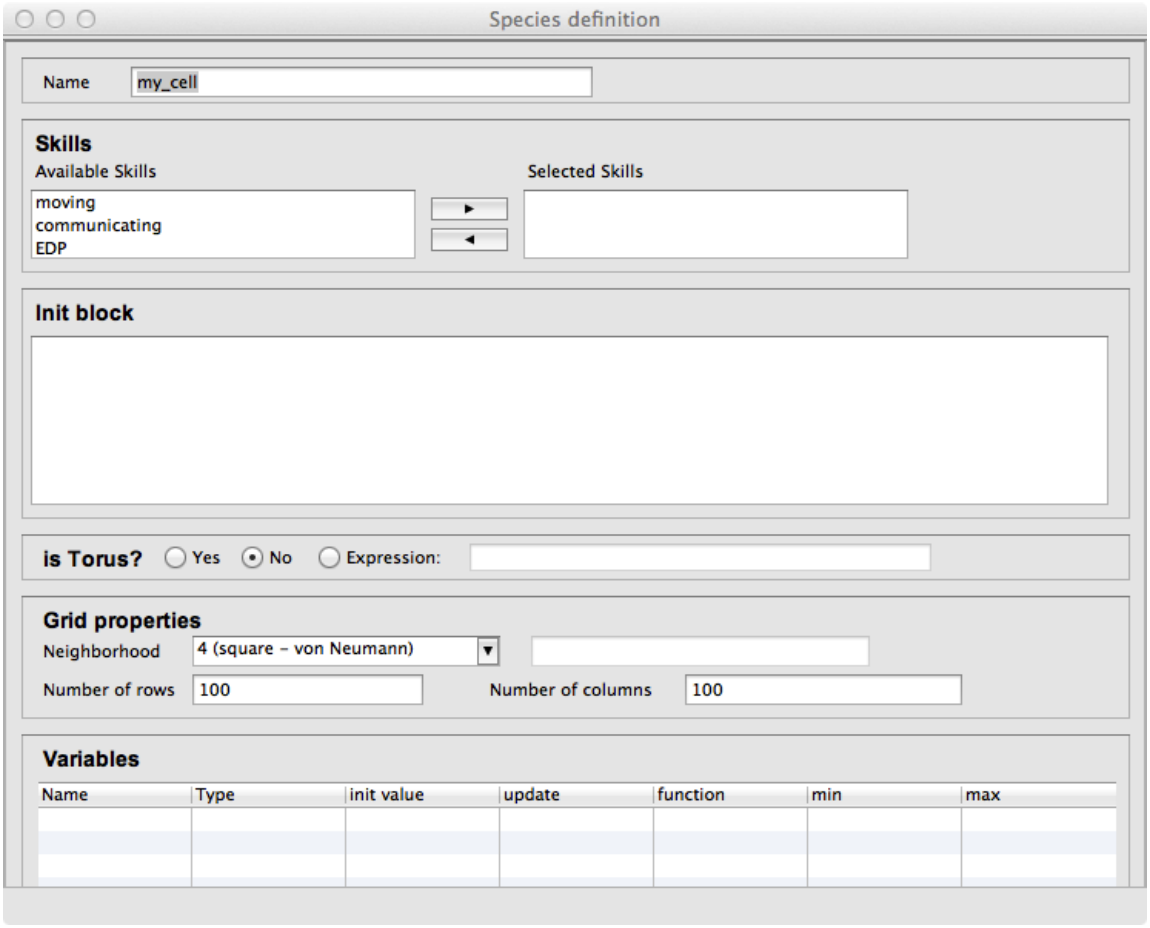


Figure 83.9: images/graphical_editor/Frame_grid.png

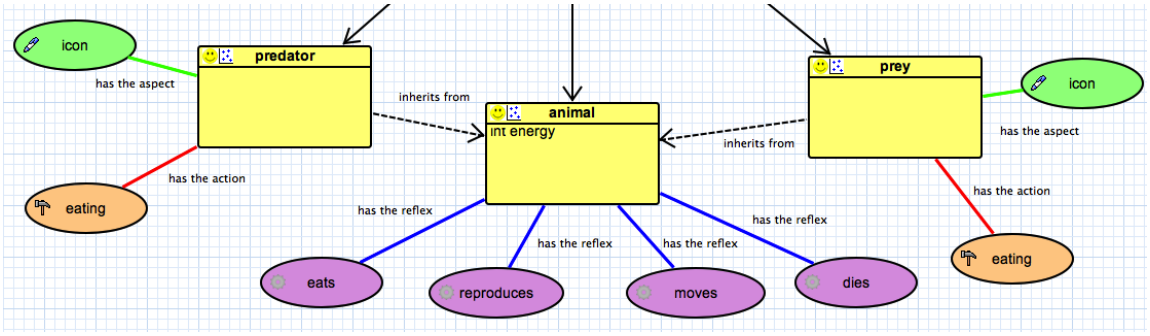


Figure 83.10: images/graphical_editor/inhereting_link.png

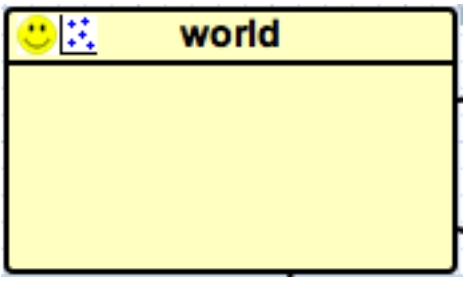


Figure 83.11: images/graphical_editor/world.png

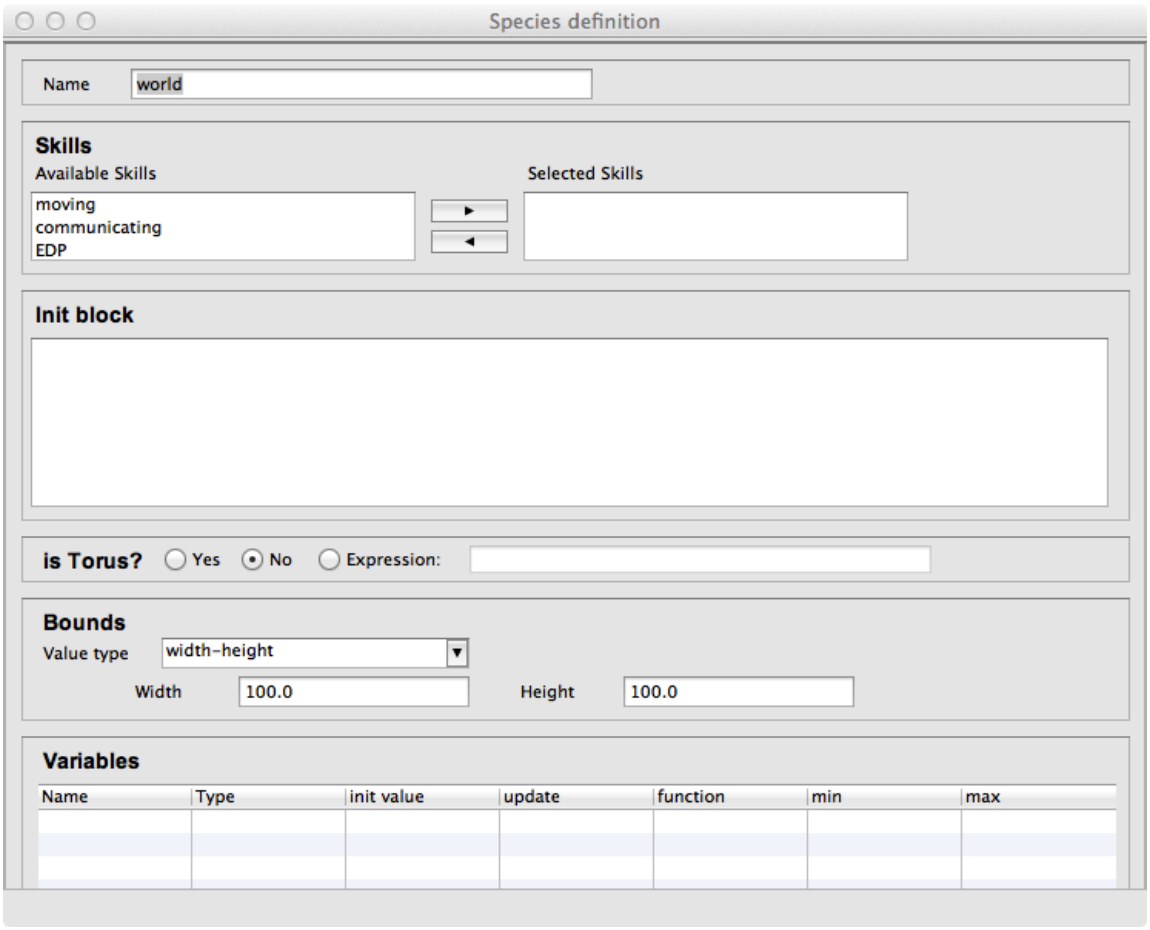


Figure 83.12: images/graphical_editor/Frame_world.png

Figure 83.13: images/graphical_editor/action.png

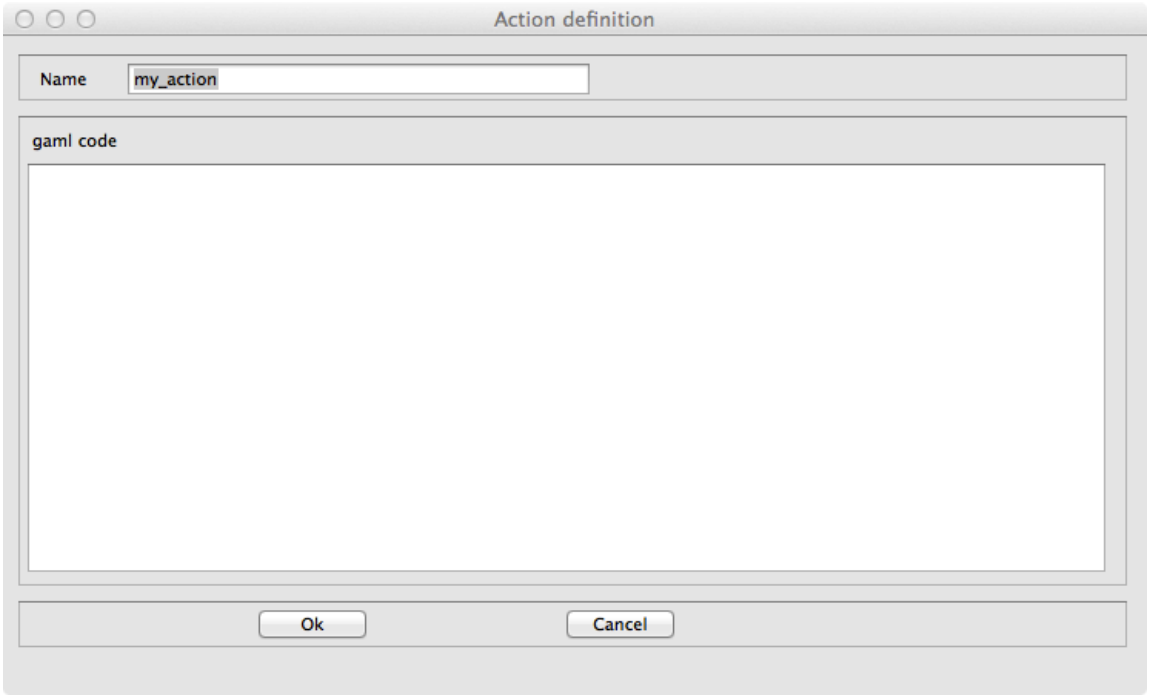


Figure 83.14: images/graphical_editor/Frame_action.png

agent features

action

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

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

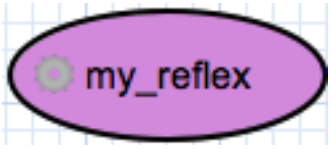


Figure 83.15: images/graphical_editor/reflex.png

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**: -

equation

The equation feature allows the modeler to define an equation for a species.

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

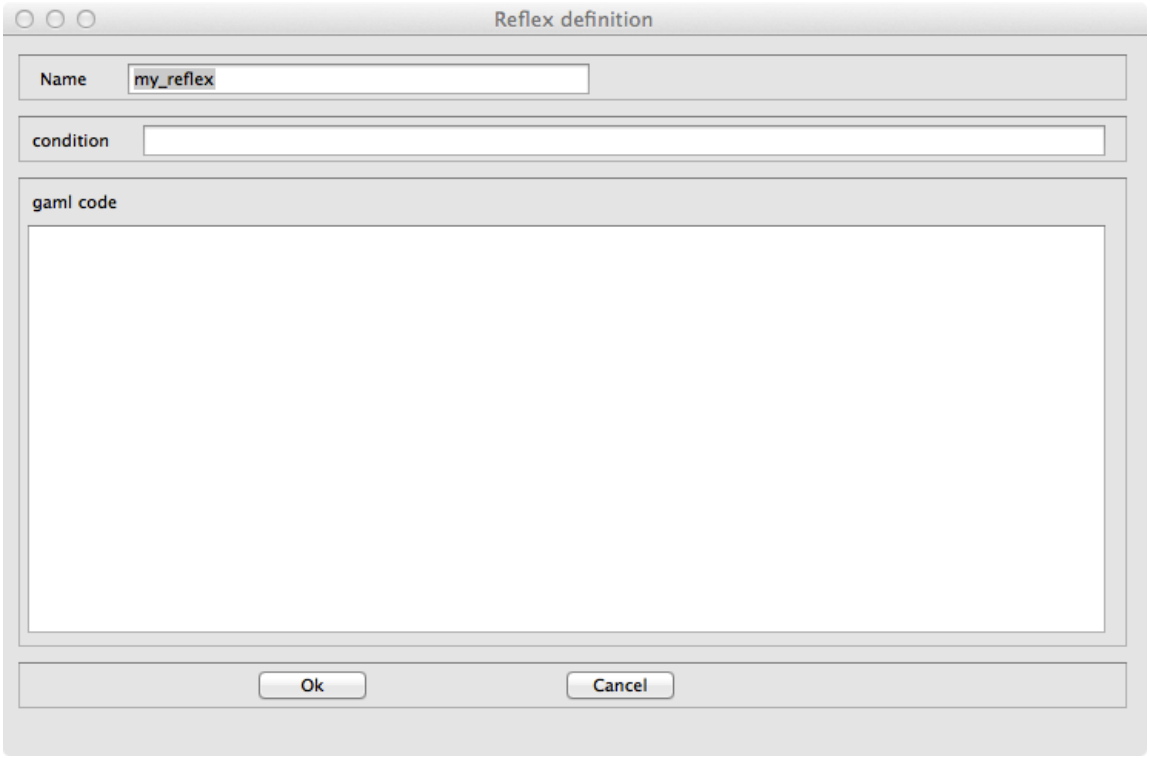


Figure 83.16: images/graphical_editor/Frame_reflex.png

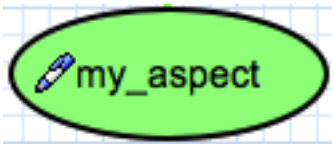


Figure 83.17: images/graphical_editor/aspect.png

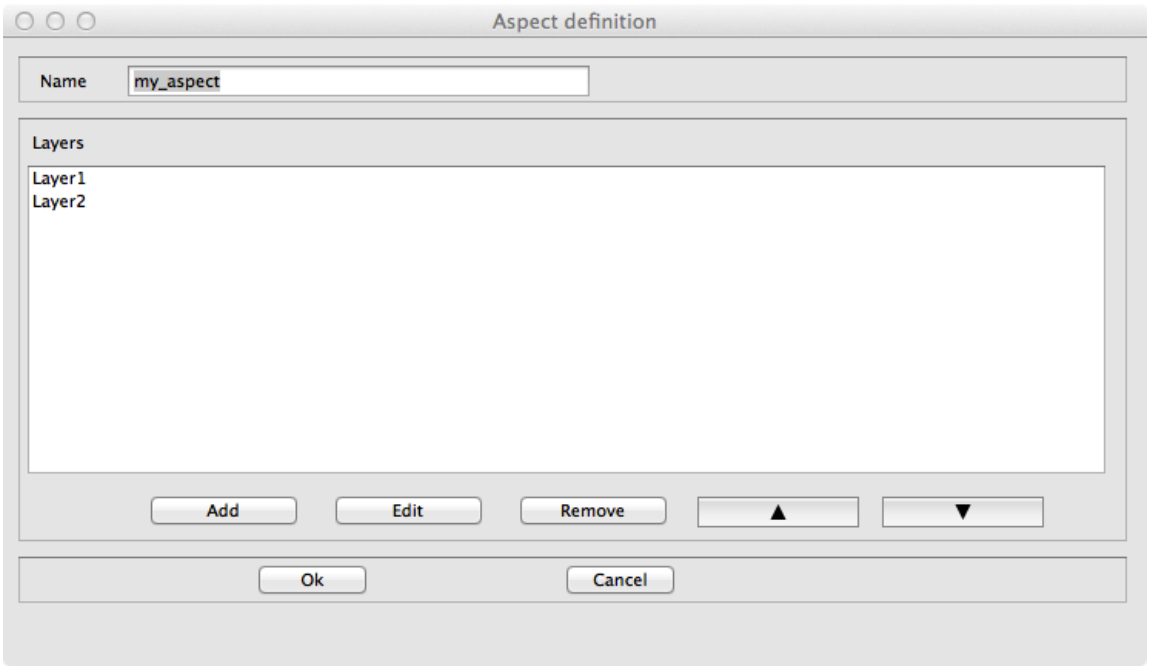


Figure 83.18: images/graphical_editor/Frame_aspect.png

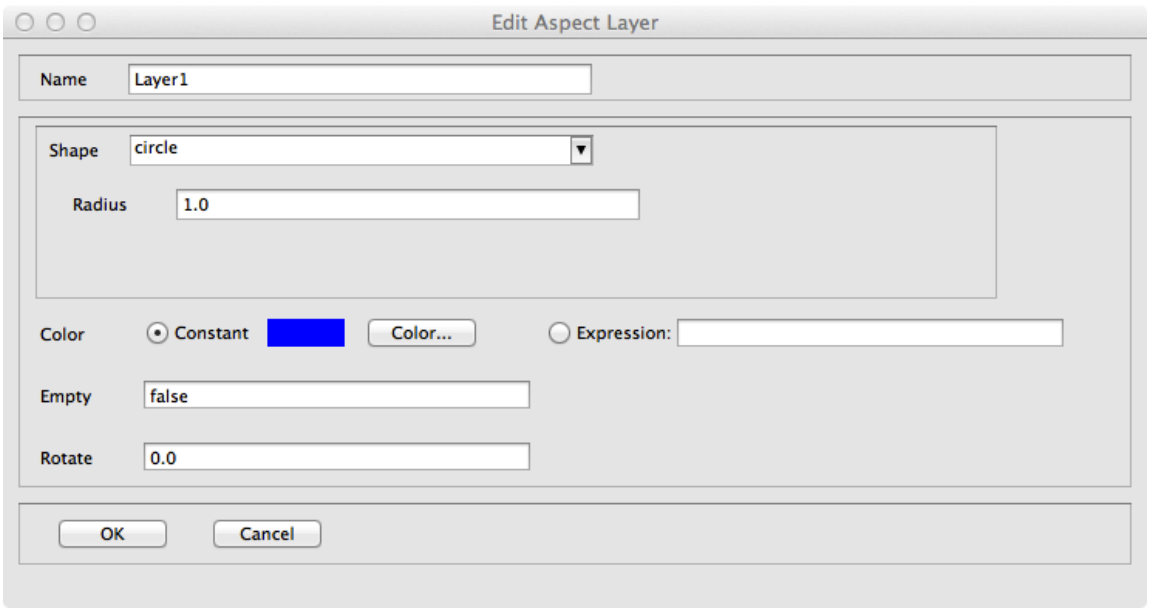


Figure 83.19: images/graphical_editor/Frame_Aspect_layer.png

Figure 83.20: images/graphical_editor/equation.png

Figure 83.21: images/graphical_editor/Frame_Equation.png

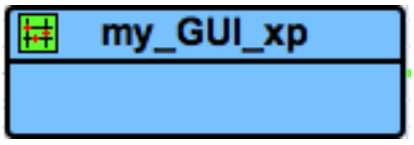


Figure 83.22: images/graphical_editor/guiXP.png

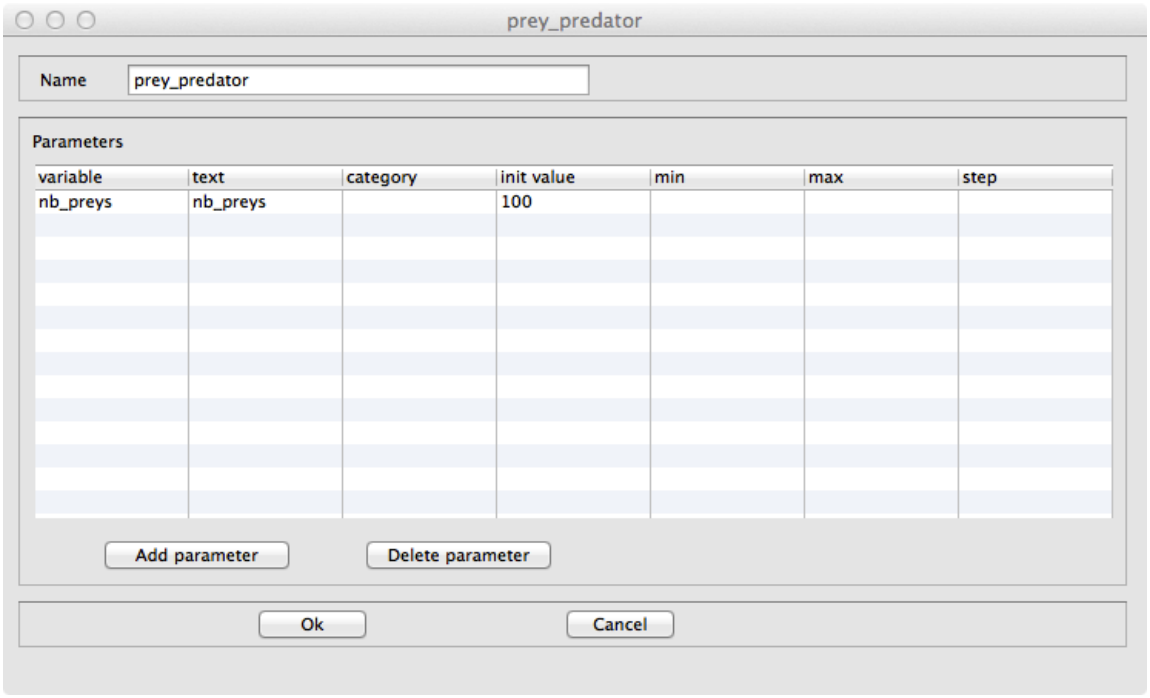


Figure 83.23: images/graphical_editor/Frame_Experiment.png

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**: -

Figure 83.24: images/graphical_editor/display.png

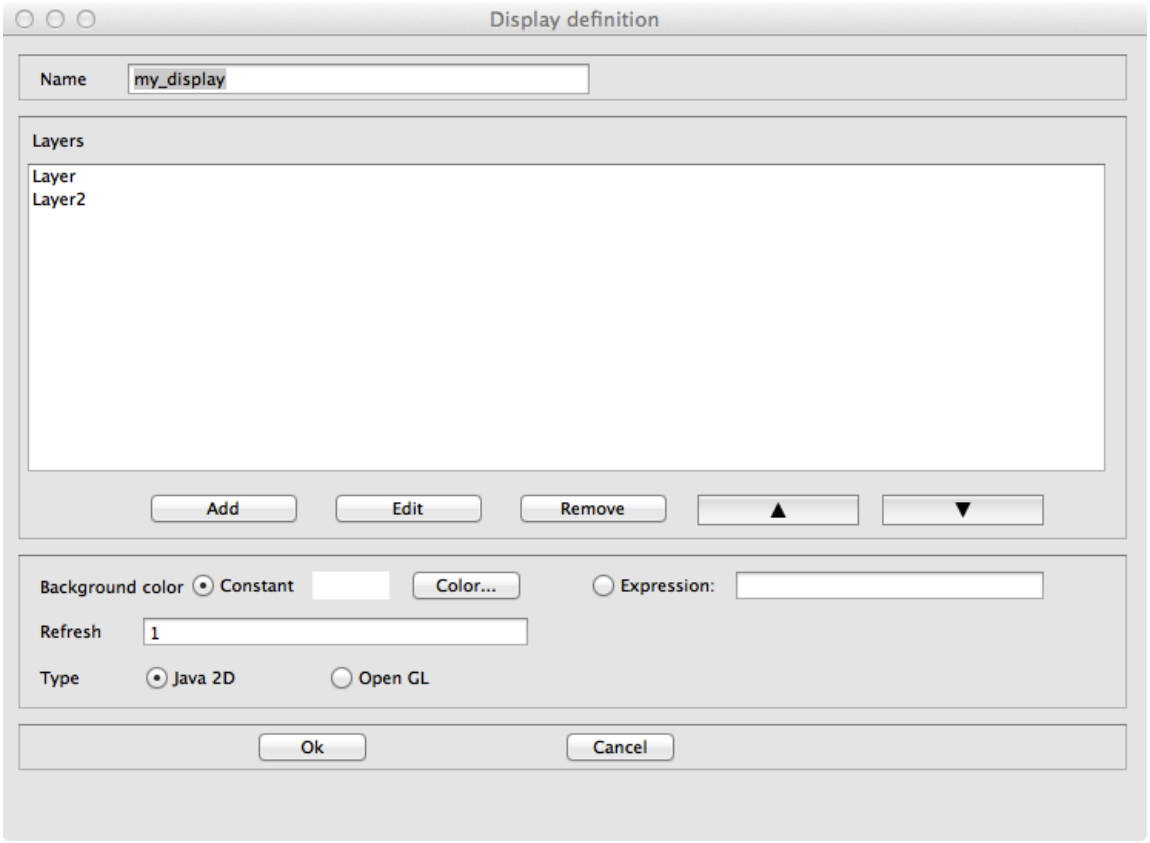


Figure 83.25: images/graphical_editor/Frame_display.png

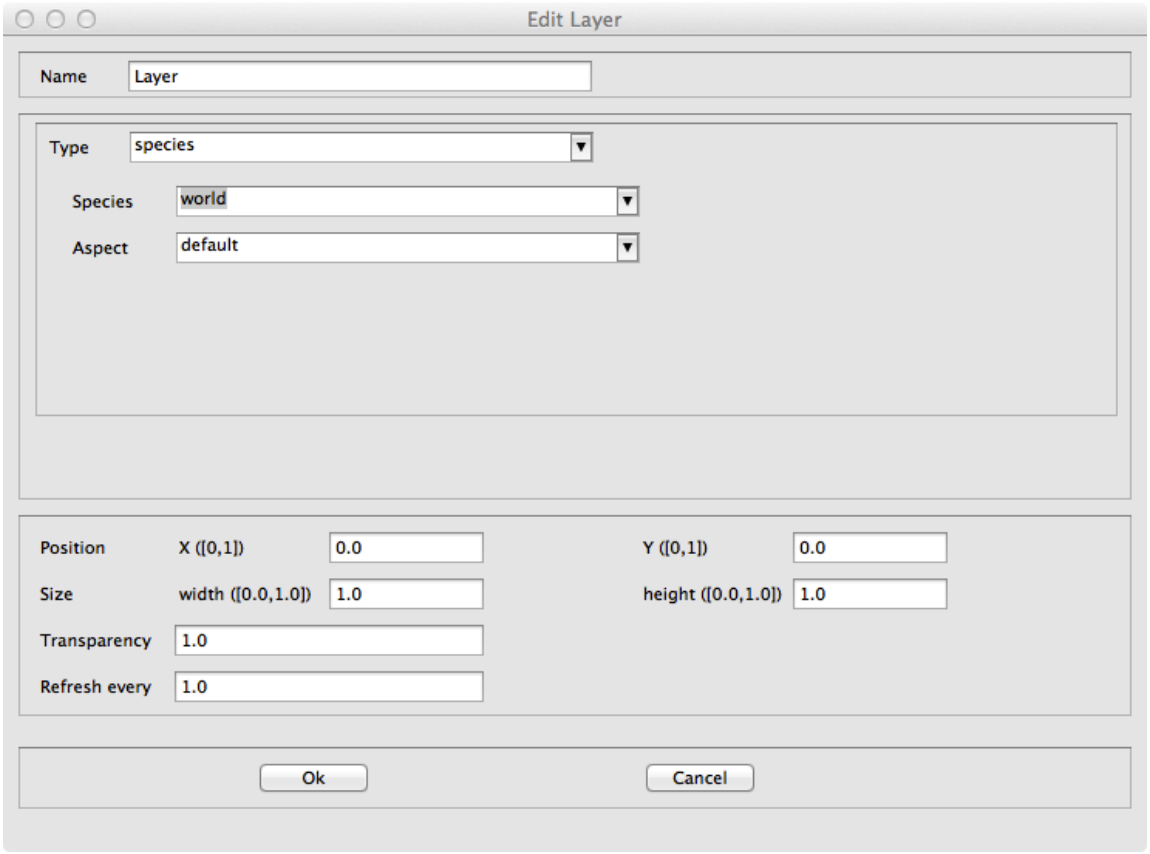


Figure 83.26: images/graphical_editor/Frame_layer_display.png

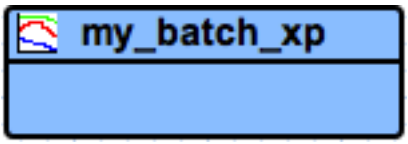


Figure 83.27: images/graphical_editor/batchxp.png

resources/images/graphicalEditor/plan.png

Figure 83.28: images/graphical_editor/plan.png

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**: -

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.

Figure 83.29: images/graphical_editor/Frame_plan.png

Figure 83.30: images/graphical_editor/rule.png

- **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**: -

Figure 83.31: images/graphical_editor/Frame_rule.png

Figure 83.32: images/graphical_editor/perception.png

Figure 83.33: images/graphical_editor/Frame_perception.png

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**: -

Figure 83.34: images/graphical_editor/state.png

Figure 83.35: images/graphical_editor/Frame_state.png

Figure 83.36: images/graphical_editor/task.png

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**: -

Figure 83.37: images/graphical_editor/Frame_task.png

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).

Chapter 84

FAQ (Frequently Asked Questions)

Can we record a video from an experiment ?

No, we cannot directly. But you have two alternatives: - With the set of images generated with the `autosave` facet of an experiment, you can construct your own video file using powerful software such as ffmpeg. - You can directly record the video stream using software such as VLC Media Player or QuickTime.

Chapter 85

Known issues

Crash when using openGL on Windows

If you are using GAMA with Windows, and your video card is a Radeon AMD, then GAMA can crash while running a simulation using OpenGL. To avoid this issue, you have to disable your video card. This will slow down a bit the performances, but at least you will be able to run GAMA without those annoying crashes.

To disable your video card, open the control panel, click on Hardware and Sound / Devices and Printers / Device manager, and then right click on your video card (as shown in the following image)

Grid not displayed right using openGL

When you try to display a grid with opengl, the cells have not a regular shape (as it is shown in the following image)

The reason of this problem is that we can only map a grid of 2^n x 2^n cells in the plan. Here are some solutions for this problem:

- Choose a grid with 2^n*2^n dimension (such as 16x16, or 32x32)
- Display the grid in java2D
- Display the grid as *species*, and not as *grid* (note that the difference in term of performance between displaying a grid as a *grid* and as a *species* is not so important for opengl displays. It has originally been done for java2D displays)

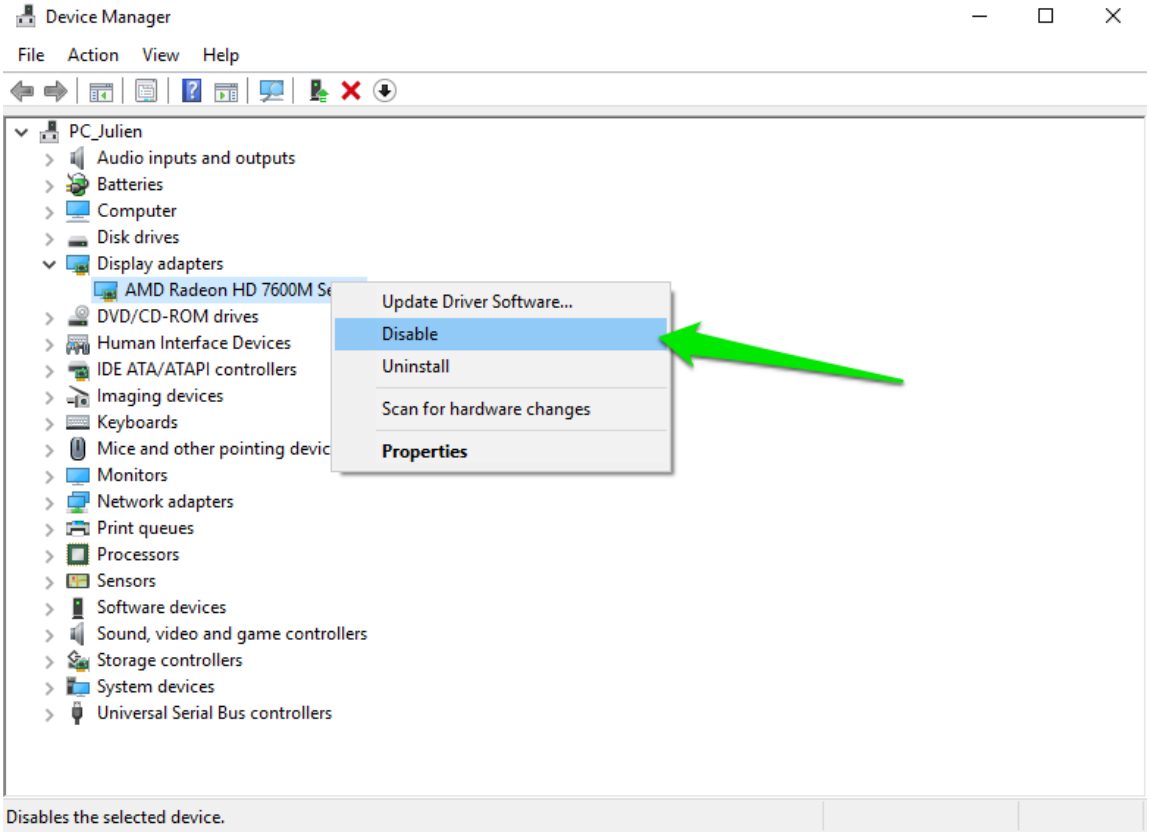


Figure 85.1: resources/images/recipes/disable_amd_radeon.png

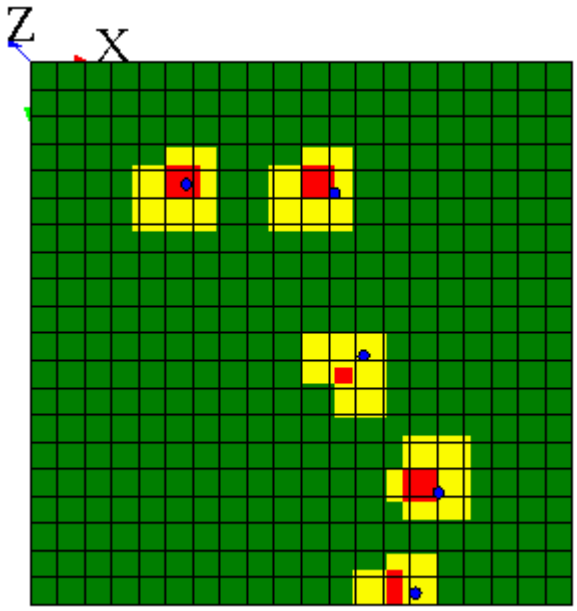


Figure 85.2: resources/images/recipes/grid_display_problem.png

Part VIII

GAML References

Chapter 86

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.

Chapter 87

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

Variables

- **host** (-29): 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

- returns: unknown

step

- returns: unknown

AgentDB

Variables

- **agents** (**list**): Returns the list of agents for the population(s) of which the receiver agent is a direct or undirect host

- **members** (**list**): Returns the list of agents for the population(s) of which the receiver agent is a direct host

Actions

close

- returns: unknown

connect

- returns: unknown
- **params** (map): Connection parameters

executeUpdate

- returns: int
- **updateComm** (string): SQL commands such as Create, Update, Delete, Drop with question mark
- **values** (list): List of values that are used to replace question mark

getParameter

- returns: unknown

insert

- returns: int
- **into** (string): Table name
- **columns** (list): List of column name of table

- **values** (list): List of values that are used to insert into table. Columns and values must have same size

isConnected

- returns: bool

select

- returns: list
- **select** (string): select string
- **values** (list): List of values that are used to replace question marks

setParameter

- returns: unknown
- **params** (map): Connection parameters

testConnection

- returns: bool
- **params** (map): Connection parameters

timeStamp

- returns: float

base_edge

Variables

- source (agent): The source agent of this edge
- target (agent): The target agent of this edge

Actions

---

experiment

Experiments that declare a graphical user interface

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
- 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 for this simulation. Three different ones are at the disposal of the modeler: mersenne represents the default generator, based on the Mersenne-Twister algorithm. Very reliable; cellular is a cellular automaton based generator that should be a bit faster, but less reliable; and java invokes the standard Java generator

- **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
- **simulation** (-27): contains a reference to the current simulation being run by this experiment
- **simulations** (**list**): contains the list of currently running simulations
- **warnings** (boolean): The value of the preference 'Consider warnings as errors'
- **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

- returns: unknown

update_outputs

Forces all outputs to refresh, optionally recomputing their values

- returns: unknown
- **recompute** (boolean): Whether or not to force the outputs to make a computation step

graph_edge

Variables

- **source** (**agent**): The source agent of this edge
- **target** (**agent**): The target agent of this edge

Actions

---

graph_node

Variables

- **my_graph** (**graph**): A reference to the graph containing the agent

Actions

related_to

This operator should never be called

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

---

physical_world

The base species for agents that act as a 3D physical world

Variables

- **agents** (**list**): The list of agents registered in this physical world
- **gravity** (**float**): Define if the value for the gravity
- **use_gravity** (boolean): Define if the physical world has a gravity or not

Actions

compute_forces

- returns: unknown
- **step** (float): allows to define the time step considered for the physical world agent. If not defined, the physical world agent will use the step global variable.

Chapter 88

The 'agent' built-in species (Under Construction)

As described in the presentation of GAML, the hierarchy of species derives from a single built-in species called **agent**. All its components (attributes, actions) will then be inherited by all direct or indirect children species (including `model` and `experiment`), with the exception of species that explicitly mention `use_minimal_agents: true` as a facet, which inherit from a stripped-down version of **agent** (see below).

agent attributes

agent defines several attributes, which form the minimal set of knowledge any agent will have in a model. *

agent actions

Chapter 89

The 'model' built-in species (Under Construction)

As described in the presentation of GAML, any model in GAMA is a species (introduced by the keyword `global`) which directly inherits from an abstract species called `model`. This abstract species (sub-species of **`agent`**) defines several attributes and actions that can then be used in any global section of any model.

`model` attributes

`model` defines several attributes, which, in addition to the attributes inherited from **`agent`**, form the minimal set of knowledge a model can manipulate. *

`model` actions

Chapter 90

The 'experiment' built-in species (Under Construction)

As described in the presentation of GAML, any experiment attached to a model is a species (introduced by the keyword `experiment` which directly or indirectly inherits from an abstract species called `experiment` itself. This abstract species (sub-species of **`agent`**) defines several attributes and actions that can then be used in any experiment.

`experiment` attributes

`experiment` defines several attributes, which, in addition to the attributes inherited from **`agent`**, form the minimal set of knowledge any experiment will have access to.

`experiment` actions

Chapter 91

Built-in Skills

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

Introduction

Skills are built-in modules, written in Java, that provide a set of related built-in variables and built-in actions (in addition to those already provided by GAMA) to the species that declare them. A declaration of skill is done by filling the skills attribute in the species definition:

```
species my_species skills: [skill1, skill2] {
    ...
}
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. An example of skill is the `moving` skill.

So, for instance, if a species is declared as:

```
species foo skills: [moving]{
...
}
```

Its agents will automatically be provided with the following variables : `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `wander`, `follow` in addition to those built-in in species and declared by the modeller. Most of these variables, except the ones marked read-only, can be customized and modified like normal variables by the modeller. For instance, one could want to set a maximum for the speed; this would be done by redeclaring it like this:

```
float speed max:100 min:0;
```

Or, to obtain a speed increasing at each simulation step:

```
float speed max:100 min:0  <- 1 update: speed * 1.01;
```

Or, to change the speed in a behavior:

```
if speed = 5 {
    speed <- 10;
}
```

Table of Contents

---

advanced_driving

Variables

- **current_index** (**int**): the current index of the agent target (according to the targets list)
- **current_lane** (**int**): the current lane on which the agent is

- **current_path** (**path**): the current path that tha agent follow
- **current_road** (**agent**): current road on which the agent is
- **current_target** (**point**): the current target of the agent
- **distance_to_goal** (**float**): euclidean distance to the next point of the current segment
- **final_target** (**point**): the final target of the agent
- **max_acceleration** (**float**): maximum acceleration of the car for a cycle
- **max_speed** (**float**): maximal speed of the vehicle
- **min_safety_distance** (**float**): the minimal distance to another driver
- **min_security_distance** (**float**): the minimal distance to another driver
- **on_linked_road** (boolean): is the agent on the linked road?
- **proba_block_node** (**float**): probability to block a node (do not let other driver cross the crossroad)
- **proba_lane_change_down** (**float**): probability to change lane to a lower lane (right lane if right side driving) if necessary
- **proba_lane_change_up** (**float**): probability to change lane to a upper lane (left lane if right side driving) if necessary
- **proba_respect_priorities** (**float**): probability to respect priority (right or left) laws
- **proba_respect_stops** (**list**): probability to respect stop laws - one value for each type of stop
- **proba_use_linked_road** (**float**): probability to change lane to a linked road lane if necessary

- **real_speed** (**float**): the actual speed of the agent (in meter/second)
- **right_side_driving** (boolean): are drivers driving on the right size of the road?
- **safety_distance_coeff** (**float**): the coefficient for the computation of the the min distance between two drivers (according to the vehicle speed - security_distance =max(min_security_distance, security_distance_coeff * min(self.real_speed, other.real_speed))
- **security_distance_coeff** (**float**): the coefficient for the computation of the the min distance between two drivers (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 driver want to reach (according to the max speed of the road)
- **targets** (**list**): the current list of points that the agent has to reach (path)
- **vehicle_length** (**float**): the length of the vehicle (in meters)

Actions

advanced_follow_driving

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

- returns: float

- **path** (path): a path to be followed.
- **target** (point): the target to reach
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **time** (float): time to travel

compute_path

action to compute a path to a target location according to a given graph

- returns: path
- **graph** (graph): the graph on wich compute the path
- **target** (agent): the target node to reach
- **source** (agent): the source node (optional, if not defined, closest node to the agent location)
- **on_road** (agent): the road on which the agent is located (optional)

drive

action to drive toward the final target

- returns: void

drive_random

action to drive by chosen randomly the next road

- returns: void
- **proba_roads** (map): a map containing for each road (key), the probability to be selected as next road (value)

external_factor_impact

action that allows to define how the remaining time is impacted by external factor

- returns: float
- **new_road** (agent): the road on which to the driver wants to go
- **remaining_time** (float): the remaining time

is_ready_next_road

action to test if the driver can take the given road at the given lane

- returns: bool
- **new_road** (agent): the road to test
- **lane** (int): the lane to test

lane_choice

action to choose a lane

- returns: int
- **new_road** (agent): the road on which to choose the lane

path_from_nodes

action to compute a path from a list of nodes according to a given graph

- returns: path
- **graph** (graph): the graph on wich compute the path
- **nodes** (list): the list of nodes composing the path

speed_choice

action to choose a speed

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

test_next_road

action to test if the driver can take the given road

- returns: bool
- **new_road** (agent): the road to test

---

driving

Variables

- **lanes_attribute** (**string**): the name of the attribut of the road agent that determine the number of road lanes
- **living_space** (**float**): the min distance between the agent and an obstacle (in meter)
- **obstacle_species** (**list**): the list of species that are considered as obstacles
- **speed** (**float**): the speed of the agent (in meter/second)
- **tolerance** (**float**): the tolerance distance used for the computation (in meter)

Actions

follow_driving

moves the agent along a given path passed in the arguments while considering the other agents in the network.

- returns: path
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **path** (path): a path to be followed.
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weigths used for the moving.
- **living_space** (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- **tolerance** (float): tolerance distance used for the computation (replaces the current value of tolerance)
- **lanes_attribute** (string): the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

goto_driving

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

- returns: path
- **target** (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): Weigths used for the moving.
- **living_space** (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- **tolerance** (float): tolerance distance used for the computation (replaces the current value of tolerance)
- **lanes_attribute** (string): the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

---

fipa

The fipa skill offers some primitives and built-in variables which enable agent to communicate with each other using the FIPA interaction protocol.

Variables

- **accept_proposals** (**list**): A list of 'accept_proposal' performative messages 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 'propose' 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.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

agree

Replies a message with an 'agree' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

cancel

Replies a message with a 'cancel' peformative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

cfp

Replies a message with a 'cfp' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

end_conversation

Reply a message with an 'end_conversation' peprformative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsible of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

failure

Replies a message with a 'failure' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

inform

Replies a message with an 'inform' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

propose

Replies a message with a 'propose' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

query

Replies a message with a 'query' performative message.

- returns: unknown

- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

refuse

Replies a message with a 'refuse' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The contents of the replying message

reject_proposal

Replies a message with a 'reject_proposal' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

reply

Replies a message. This action should be only used to reply a message in a 'no-protocol' conversation and with a 'user defined performative'. For performatives supported by GAMA (i.e., standard FIPA performatives), please use the 'action' with the same name of 'performative'. For example, to reply a message with a 'request' performative message, the modeller should use the 'request' action.

- returns: unknown
- **message** (24): The message to be replied

- **performative** (string): The performative of the replying message
- **contents** (list): The content of the replying message

request

Replies a message with a 'request' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

send

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.FIPAMessage
- **to** (list): A list of receiver agents
- **contents** (list): The content of the message. A list of any GAML type
- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

start_conversation

Starts a conversation/interaction protocol.

- returns: msi.gaml.extensions.fipa.FIPAMessage
- **to** (list): A list of receiver agents

- **contents** (list): The content of the message. A list of any GAML type
- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

subscribe

Replies a message with a 'subscribe' performative message.

- returns: unknown
- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

MDXSKILL

This skill allows agents to be provided with actions and attributes in order to connect to MDX databases

Variables

Actions

select

- returns: list
- **params** (map): Connection parameters
- **onColumns** (string): select string with question marks

- **onRows** (list): List of values that are used to replace question marks
- **from** (list): List of values that are used to replace question marks
- **where** (list): List of values that are used to replace question marks
- **values** (list): List of values that are used to replace question marks

testConnection

- returns: bool
- **params** (map): Connection parameters

timeStamp

- returns: float

---

messaging

A simple skill that provides agents with a mailbox than can be filled with messages

Variables

- **mailbox** (**list**): The list of messages that can be consulted by the agent

Actions

send

- returns: msi.gama.extensions.messaging.GamaMessage

- **to** (any type): The agent, or server, to which this message will be sent to
- **contents** (any type): The contents of the message, an arbitrary object

---

moving

The moving skill is intended to define the minimal set of behaviours required for agents that are able to move on different topologies

Variables

- **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.

- 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** (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 recompute even if the graph is modified (by default: true)
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weights used for the moving.

move

moves the agent forward, the distance being computed with respect to its speed and heading. The value of the corresponding variables are used unless arguments are passed.

- returns: path

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

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** (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

---

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.

- returns: path
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (int): int, optional, the direction to take for this move (replaces the current value of heading)
- **pitch** (int): int, optional, the direction to take for this move (replaces the current value of pitch)
- **heading** (int): int, optional, the direction to take for this move (replaces the current value of roll)
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry

network

The network skill provides new features to let agents exchange message through network.

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 as a server.

- returns: void
- **protocol** (string): protocol type (MQTT (by default), TCP, UDP): the possible value ares 'udp_server', 'udp_emitter', 'tcp_server', 'tcp_client', otherwise the MQTT protocol is used.
- **port** (int): Port number
- **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)

execute

- returns: string
- **command** (string): command to execute

fetch_message

- returns: msi.gama.extensions.messaging.GamaMessage

has_more_message

- returns: bool

join_group

allow an agent to join a group of agents in order to broadcast messages to other membersor to receive messages sent by other members. Note that all members of the group called : "ALL".

- returns: void
- **with_name** (string): name of the group

leave_group

leave a group of agents. The leaving agent will not receive any message from the group. Overwhise, it can send messages to the left group

- returns: void
- **with_name** (string): name of the group the agent wants to leave

simulate_step

Simulate a step to test the skill. It must be used for Gama-platform test only

- returns: void

---

physics

Variables

- **ang_damping** (**float**): angular dampling
- **collisionBound** (**map**): map describing the shape of the agent. format for sphere: ['shape'::'sphere', 'radius'::10.0]; for floor: ['shape'::'floor', 'x'::100.0, 'y'::100.0, 'z'::2.0]; if not defined, the shape attribute of the agent is used.
- **friction** (**float**): coefficient of friction of the agent
- **lin_damping** (**float**): linear dampling
- **mass** (**float**): mass of the agent
- **restitution** (**float**): coefficient of restitution force of the agent
- **velocity** (**list**): velocity of the agent

Actions

---

skill_road

Variables

- **agents_on** (**list**): for each lane of the road, the list of agents for each segment
- **all_agents** (**list**): the list of agents on the road
- **lanes** (**int**): the number of lanes
- **linked_road** (-199): the linked road: the lanes of this linked road will be usable by drivers on the road
- **maxspeed** (**float**): the maximal speed on the road
- **source_node** (**agent**): the source node of the road
- **target_node** (**agent**): the target node of the road

Actions

register

register the agent on the road at the given lane

- returns: void
- **agent** (agent): the agent to register on the road.
- **lane** (int): the lane index on which to register; if lane index >= number of lanes, then register on the linked road

unregister

unregister the agent on the road

- returns: void
- **agent** (agent): the agent to unregister on the road.

---

skill_road_node

Variables

- **block** (**map**): define the list of agents blocking the node, and for each agent, the list of concerned roads
- **priority_roads** (**list**): the list of priority roads
- **roads_in** (**list**): the list of input roads
- **roads_out** (**list**): the list of output roads
- **stop** (**list**): define for each type of stop, the list of concerned roads

Actions

SQLSKILL

This skill allows agents to be provided with actions and attributes in order to connect to SQL databases

Variables

Actions

executeUpdate

- returns: int
- **params** (map): Connection parameters
- **updateComm** (string): SQL commands such as Create, Update, Delete, Drop with question mark
- **values** (list): List of values that are used to replace question mark

getCurrentDateTime

- returns: string
- **dateFormat** (string): date format examples: 'yyyy-MM-dd' , 'yyyy-MM-dd HH:mm:ss'

getDateOffset

- returns: string
- **dateFormat** (string): date format examples: 'yyyy-MM-dd' , 'yyyy-MM-dd HH:mm:ss'
- **dateStr** (string): Start date
- **offset** (string): number on day to increase or decrease

insert

- returns: int

- **params** (map): Connection parameters
- **into** (string): Table name
- **columns** (list): List of column name of table
- **values** (list): List of values that are used to insert into table. Columns and values must have same size

list2Matrix

- returns: matrix
- **param** (list): Param: a list of records and metadata
- **getName** (boolean): getType: a boolean value, optional parameter
- **getType** (boolean): getType: a boolean value, optional parameter

select

- returns: list
- **params** (map): Connection parameters
- **select** (string): select string with question marks
- **values** (list): List of values that are used to replace question marks

testConnection

- returns: bool
- **params** (map): Connection parameters

timeStamp

- returns: float

Chapter 92

Built-in Architectures

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

INTRODUCTION

Table of Contents

---

fsm

Variables

- **state** (string): Returns the current state in which the agent is
- **states** (list): Returns the list of all possible states the agents can be in

Actions

---

parallel_bdi

compute the bdi architecture in parallel

Variables

Actions

---

probabilistic_tasks

Variables

Actions

---

reflex

Variables

Actions

---

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): thecurrent 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 automaticaly 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 personnality 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** (546704): predicate to add as a belief
- **strength** (float): the stregth 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** (546706): emotion to add as a belief
- **strength** (float): the stregth 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** (546708): predicate to add as a belief
- **strength** (float): the stregth of the belief
- **lifetime** (int): the lifetime of the belief

add_desire

adds the predicates is in the desire base.

- returns: bool
- **predicate** (546704): predicate to add as a desire
- **strength** (float): the stregth of the belief
- **lifetime** (int): the lifetime of the belief
- **todo** (546704): add the desire as a subintention of this parameter

add_desire_emotion

adds the emotion in the desire base.

- returns: bool
- **emotion** (546706): emotion to add as a desire
- **strength** (float): the stregth of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (546704): 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** (546708): mental_state to add as a desire
- **strength** (float): the stregth of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (546704): add the desire as a subintention of this parameter

add_directly_belief

add the belief in the belief base.

- returns: bool
- **belief** (546708): belief to add in th belief base

add_directly_desire

add the desire in the desire base.

- returns: bool
- **desire** (546708): desire to add in th belief base

add_directly_ideal

add the ideal in the ideal base.

- returns: bool
- **ideal** (546708): ideal to add in the ideal base

add_directly_uncertainty

add the uncertainty in the uncertainty base.

- returns: bool
- **uncertainty** (546708): uncertainty to add in the uncertainty base

add_emotion

add the emotion to the emotion base.

- returns: bool
- **emotion** (546706): emotion to add to the base

add_ideal

add a predicate in the ideal base.

- returns: bool
- **predicate** (546704): 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** (546706): 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** (546708): 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** (546704): predicate to check

- **strength** (float): the stregth 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** (546706): emotion to add as an intention
- **strength** (float): the stregth 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** (546708): predicate to add as an intention
- **strength** (float): the stregth of the belief
- **lifetime** (int): the lifetime of the belief

add_obligation

add a predicate in the ideal base.

- returns: bool
- **predicate** (546704): 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** (546707): social link to add to the base

add_subintention

adds the predicates is in the desire base.

- returns: bool
- **predicate** (546708): the intention that receives the sub_intention
- **subintentions** (546704): 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** (546704): predicate to add
- **strength** (float): the stregth of the belief
- **lifetime** (int): the lifetime of the belief

add_uncertainty_emotion

add a predicate in the uncertainty base.

- returns: bool
- **emotion** (546706): emotion to add as an uncertainty
- **strength** (float): the stregth 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** (546708): mental state to add as an uncertainty
- **strength** (float): the stregth 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 who 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 who 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` (546704): 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** (546706): 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** (546708): 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
- **predicate** (546704): predicate to check

get_beliefs_metal_state

get the list of bliefs in the belief base containing the mental state

- returns: list
- **mental_state** (546708): mental state to check

get_beliefs_with_name

get the list of predicates is in the belief base with the given name.

- returns: list
- **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** (546704): 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** (546708): 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
- **predicate** (546704): 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** (546708): 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
- **name** (string): name of the predicates to check

get_emotion

get the emotion in the emotion base (if several, returns the first one).

- returns: emotion
- **emotion** (546706): emotion to get

get_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** (546704): 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** (546708): mental state to return

get_intention

get the predicates in the intention base (if several, returns the first one).

- returns: mental_state
- **predicate** (546704): 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** (546708): 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
- **predicate** (546704): 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** (546708): mental state to check

get_intentions_with_name

get the list of predicates is in the belief base with the given name.

- returns: list
- **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** (546704): 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

get_social_link

get the social link (if several, returns the first one).

- returns: social_link
- **social_link** (546707): 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** (546704): 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** (546708): mental state to return

has_belief

check if the predicates is in the belief base.

- returns: bool
- **predicate** (546704): predicate to check

has_belief_mental_state

check if the mental state is in the belief base.

- returns: bool
- **mental_state** (546708): 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** (546704): predicate to check

has_desire_mental_state

check if the mental state is in the desire base.

- returns: bool
- **mental_state** (546708): 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** (546706): 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** (546704): predicate to check

has_ideal_mental_state

check if the mental state is in the ideal base.

- returns: bool
- **mental_state** (546708): 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** (546704): predicate to check

has_social_link

check if the social link base.

- returns: bool
- **social_link** (546707): 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** (546704): predicate to check

has_uncertainty_mental_state

check if the mental state is in the uncertainty base.

- returns: bool
- **mental_state** (546708): 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** (546704): 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** (546708): 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** (546704): predicate to remove

remove_belief

removes the predicate from the belief base.

- returns: bool
- **predicate** (546704): predicate to remove

remove_belief_mental_state

removes the mental state from the belief base.

- returns: bool
- **mental_state** (546708): mental state to remove

remove_desire

removes the predicates from the desire base.

- returns: bool
- **predicate** (546704): predicate to remove from desire base

remove_desire_mental_state

removes the mental state from the desire base.

- returns: bool
- **mental_state** (546708): mental state to remove from desire base

remove_emotion

removes the emotion from the emotion base.

- returns: bool
- **emotion** (546706): emotion to remove

remove_ideal

removes the predicates from the ideal base.

- returns: bool
- **predicate** (546704): predicate to remove

remove_ideal_mental_state

removes the mental state from the ideal base.

- returns: bool
- **mental_state** (546708): metal state to remove

remove_intention

removes the predicates from the intention base.

- returns: bool
- **predicate** (546704): 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** (546708): intention's mental state to remove
- **desire_also** (boolean): removes also desire

remove_obligation

removes the predicates from the obligation base.

- returns: bool
- **predicate** (546704): predicate to remove

remove_social_link

removes the social link from the social relation base.

- returns: bool
- **social_link** (546707): 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** (546704): predicate to remove

remove_uncertainty_mental_state

removes the mental state from the uncertainty base.

- returns: bool
- **mental_state** (546708): mental state to remove

replace_belief

replace the old predicate by the new one.

- returns: bool
- **old_predicate** (546704): predicate to remove
- **predicate** (546704): predicate to add

---

sorted_tasks

Variables

Actions

---

user_first

Variables

Actions

---

user_last

Variables

Actions

---

user_only

Variables

Actions

---

weighted_tasks

Variables

Actions

Chapter 93

Statements

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

Table of Contents

Statements by kinds

- **Batch method**

  - annealing, exhaustive, genetic, hill_climbing, reactive_tabu, tabu,

- **Behavior**
  - aspect, coping, norm, plan, reflex, rule, sanction, state, task, test, user_init, user_panel,

- **Behavior**
  - aspect, coping, norm, plan, reflex, rule, sanction, state, task, test, user_init, user_panel,

- **Experiment**
  - experiment,

- **Layer**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,

- **Output**
  - display, inspect, layout, monitor, output, output_file, permanent,

- **Parameter**
  - parameter,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**

  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Sequence of statements or action**
  - action, ask, benchmark, capture, catch, create, default, else, enter, equation, exit, if, loop, match, migrate, perceive, release, run, setup, start_simulation, switch, trace, transition, try, user_command, using,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**
  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Single statement**

  - =, add, assert, break, conscious_contagion, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, law, let, put, remove, return, rule, save, set, simulate, socialize, solve, status, unconscious_contagion, user_input, warn, write,

- **Species**
  - species,

- **Variable (container)**
  - Variable_container,

- **Variable (number)**
  - Variable_number,

- **Variable (regular)**
  - Variable_regular,

Statements by embedment

- **Behavior**
  - add, ask, assert, benchmark, capture, conscious_contagion, create, diffuse, do, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, run, save, set, simulate, socialize, solve, start_simulation, status, switch, trace, transition, try, unconscious_contagion, using, warn, write,
- **Environment**
  - species,
- **Experiment**

  - action, annealing, exhaustive, genetic, hill_climbing, output, parameter, permanent, reactive_tabu, reflex, rule, setup, simulate, state, tabu, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,

- **Layer**
  - add, 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,

- **Sequence of statements or action**
  - add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, simulate, 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, simulate, 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, simulate, using,

- **display**
  - agents, camera, chart, display_grid, display_population, event, graphics, image, light, overlay,
- **display_population**
  - display_population,
- **equation**
  - =,
- **fsm**
  - state, user_panel,
- **if**
  - else,
- **output**
  - display, inspect, 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,

- **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...;
}
```

=

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.

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)

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

Definition

Allows to define in a species, model or experiment a new action that can be called elsewhere.

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)

Usages

- The simplest syntax to define an action that does not take any parameter and does not return anything is:

```
action simple_action {
   // [set of statements]
}
```

- If the action needs some parameters, they can be specified betwee, braquets after the identifier of the action:

```
action action_parameters(int i, string s){
   // [set of statements using i and s]
}
```

- If the action returns any value, the returned type should be used instead of the "action" keyword. A return statement inside the body of the action statement is mandatory.

```
int action_return_val(int i, string s){
   // [set of statements using i and s]
   return i + i;
}
```

- If virtual: is true, then the action is abstract, which means that the action is defined without body. A species containing at least one abstract action is abstract. Agents of this species cannot be created. The common use of an abstract action is to define an action that can be used by all its sub-species, which should redefine all abstract actions and implements its body.

```
species parent_species {
   int virtual_action(int i, string s);
}

species children parent: parent_species {
   int virtual_action(int i, string s) {
      return i + i;
   }
}
```

- See also: do,

Embedments

- The `action` statement is of type: **Sequence of statements or action**
- The `action` statement can be embedded into: Species, Experiment, Model,
- The `action` statement embeds statements: assert, return,

---

add

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

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

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

Definition

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

Facets

- **value** (container): the set of agents to display
- **name** (a label), (omissible) : Human readable title of the layer
- aspect (an identifier): the name of the aspect that should be used to display the species
- **fading** (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false
- focus (agent): the agent on which the camera will be focused (it is dynamically computed)
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are 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.
- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, useful in case of agents that do not move)
- **selectable** (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- trace (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency applied to this layer between 0 (solid) and 1 (totally transparent)

Usages

- The general syntax is:

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

```
}
```

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

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

- See also: display, chart, event, graphics, display_grid, image, overlay, display_population,

Embedments

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

---

annealing

Definition

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article and BatchExperiments.

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **aggregation** (a label), takes values in: {min, max}: the agregation method
- **maximize** (float): the value the algorithm tries to maximize
- **minimize** (float): the value the algorithm tries to minimize
- **nb_iter_cst_temp** (int): number of iterations per level of temperature
- **temp_decrease** (float): temperature decrease coefficient
- **temp_end** (float): final temperature
- **temp_init** (float): initial temperature

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

Definition

Allows an agent, the sender agent (that can be the GlobalSpecies), 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.

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.

Usages

- Ask a set of receiver agents, stored in a container, to perform a block of statements. The block is evaluated in the context of the agents' species

```
ask ${receiver_agents} {
     ${cursor}
}
```

- Ask one agent to perform a block of statements. The block is evaluated in the context of the agent's species

```
ask ${one_agent} {
     ${cursor}
}
```

- If the species of the receiver agent(s) cannot be determined, it is possible to force it using the **as** facet. An error is thrown if an agent is not a direct or undirect instance of this species

```
ask${receiver_agent(s)} as: ${a_species_expression} {
     ${cursor}
}
```

- To ask a set of agents to do something only if they belong to a given species, the **of_species** operator can be used. If none of the agents belong to the species, nothing happens

```
ask ${receiver_agents} of_species ${species_name} {
     ${cursor}
}
```

- Any statement can be declared in the block statements. All the statements will be evaluated in the context of the receiver agent(s), as if they were defined in their species, which means that an expression like `self` will represent the receiver agent and not the sender. If the sender needs to refer to itself, some of its own attributes (or temporary variables) within the block statements, it has to use the keyword `myself`.

```
species animal {
    float energy <- rnd (1000) min: 0.0;
    reflex when: energy > 500 { // executed when the energy is above
   the given threshold
         list<animal> others <- (animal at_distance 5); // find all the
    neighboring animals in a radius of 5 meters
         float shared_energy  <- (energy - 500) / length (others); //
   compute the amount of energy to share with each of them
         ask others { // no need to cast, since others has already been
    filtered to only include animals
              if (energy < 500) { // refers to the energy of each
   animal in others
                   energy <- energy + myself.shared_energy; //
   increases the energy of each animal
                   myself.energy <- myself.energy - myself.
   shared_energy; // decreases the energy of the sender
              }
         }
    }
}
```

- If the species of the receiver agent cannot be determined, it is possible to force it by casting the agent. Nothing happens if the agent cannot be casted to this species

Embedments

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

---

aspect

Definition

Aspect statement is used to define a way to draw the current agent. Several aspects can be defined in one species. It can use attributes to customize each agent's aspect.

The aspect is evaluate for each agent each time it has to be displayed.

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)

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

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

propagagated but invalidates the test (in case of a warning, it partially invalidates it). Otherwise, it is normally propagated

Facets

- **value** (boolean), (omissible) : a boolean expression. If its evaluation is true, the assertion is successful. Otherwise, an error (or a warning) is raised.
- **warning** (boolean): if set to true, makes the assertion emit a warning instead of an error

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

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

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

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:

---

break

Definition

break allows to interrupt the current sequence of statements.

Facets

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

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.

Facets

- **name** (string), (omissible) : The name of the camera
- **location** (point): The location of the camera in the world
- **look_at** (point): The location that the camera is looking
- **up_vector** (point): The up-vector of the camera.

Usages

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

Embedments

- The camera statement is of type: **Layer**
- The camera statement can be embedded into: display,

- The camera statement embeds statements:

---

capture

Definition

Allows an agent to capture other agent(s) as its micro-agent(s).

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)

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. MultiLevelArchitecture#declaration-of-micro-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

Definition

This statement cannot be used alone

Facets

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:

---

chart

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).

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_font** (string): Label font face
- **label_font_size** (int): Label font size
- **label_font_style** (an identifier), takes values in: {plain, bold, italic}: the style used to display labels
- **legend_font** (string): Legend font face
- **legend_font_size** (int): Legend font size
- **legend_font_style** (an identifier), takes values in: {plain, bold, italic}: the style used to display legend
- **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-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.
- **reverse_axes** (boolean): reverse X and Y axis (for example to get horizental bar charts
- **series_label_position** (an identifier), takes values in: {default, none, legend, onchart, yaxis, xaxis}: Position of the Series names: default (best guess), none, legend, onchart, xaxis (for category plots) or yaxis (uses the first serie name).

- **size** (point): the layer resize factor: {1,1} refers to the original size whereas {0.5,0.5} divides by 2 the height and the width of the layer. In case of a 3D layer, a 3D point can be used (note that {1,1} is equivalent to {1,1,0}, so a resize of a layer containing 3D objects with a 2D points will remove the elevation)
- **style** (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded, default}: The sub-style style, also default style for the series.
- **tick_font** (string): Tick font face
- **tick_font_size** (int): Tick font size
- **tick_font_style** (an identifier), takes values in: {plain, bold, italic}: the style used to display ticks
- **tick_line_color** (rgb): the tick lines color
- **title_font** (string): Title font face
- **title_font_size** (int): Title font size
- **title_font_style** (an identifier), takes values in: {plain, bold, italic}: the style used to display titles
- **title_visible** (boolean): chart title visible
- **type** (an identifier), takes values in: {xy, scatter, histogram, series, pie, radar, heatmap, box_whisker}: the type of chart. It could be histogram, series, xy, pie, radar, heatmap or box whisker. The difference between series and xy is that the former adds an implicit x-axis that refers to the numbers of cycles, while the latter considers the first declaration of data to be its x-axis.
- **x_label** (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 horyzontal 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_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).

Usages

- The general syntax is:

```
display chart_display {
   chart "chart name" type: series [additional options] {
      [Set of data, datalists statements]
   }
}
```

- See also: display, agents, event, graphics, display_grid, image, overlay, quadtree, display_population, text,

Embedments

- The chart statement is of type: **Layer**
- The chart statement can be embedded into: display,
- The chart statement embeds statements: add, ask, data, datalist, do, put, remove, set, simulate, using,

conscious_contagion

Definition

enables to directly add an emotion of a perceived specie if the perceived agent ges a patricular emotion.

Facets

- **emotion_created** (546706): the emotion that will be created with the contagion

- **emotion_detected** (546706): the emotion that will start the contagion
- **name** (an identifier), (omissible) : the identifier of the unconscious contagion
- **charisma** (float): The charisma value of the perceived agent (between 0 and 1)
- **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

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:

coping

Definition

enables to add or remove mantal states depending on the emotions of the agent, after the emotional engine and before the cognitive or normative engine.

Facets

- **name** (an identifier), (omissible) : The name of the rule
- **belief** (546704): The mandatory belief
- **beliefs** (list): The mandatory beliefs
- **desire** (546704): The mandatory desire
- **desires** (list): The mandatory desires
- **emotion** (546706): The mandatory emotion
- **emotions** (list): The mandatory emotions
- **ideal** (546704): The mandatory ideal
- **ideals** (list): The mandatory ideals
- **lifetime** (int): the lifetime value of the mental state created
- **new_belief** (546704): The belief that will be added
- **new_beliefs** (list): The belief that will be added
- **new_desire** (546704): The desire that will be added
- **new_desires** (list): The desire that will be added
- **new_emotion** (546706): The emotion that will be added
- **new_emotions** (list): The emotion that will be added
- **new_ideal** (546704): 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** (546704): The uncertainty that will be added
- **obligation** (546704): 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** (546704): The belief that will be removed
- **remove_beliefs** (list): The belief that will be removed

- **remove_desire** (546704): The desire that will be removed
- **remove_desires** (list): The desire that will be removed
- **remove_emotion** (546706): The emotion that will be removed
- **remove_emotions** (list): The emotion that will be removed
- **remove_ideal** (546704): The ideal that will be removed
- **remove_ideals** (list): The ideals that will be removed
- **remove_intention** (546704): The intention that will be removed
- **remove_obligation** (546704): 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** (546704): The uncertainty that will be removed
- **strength** (any type in [float, int]): The stregth of the mental state created
- **threshold** (float): Threshold linked to the emotion.
- **uncertainties** (list): The mandatory uncertainties
- **uncertainty** (546704): The mandatory uncertainty
- **when** (boolean):

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

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).

Facets

- species (any type in [species, agent]), (omissible) : an expression that evaluates to a species, the species of the agents to be created. In the case of simulations, the name 'simulation', which represents the current instance of simulation, can also be used as a proxy to their species
- **as** (species):
- **from** (any type): an expression that evaluates to a localized entity, a list of localized entities, a string (the path of a file), a file (shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a database
- **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

Usages

- Its simple syntax to create an_int agents of species a_species is:

```
create a_species number: an_int;
create species_of(self) number: 5 returns: list5Agents;
5
```

- In GAML modelers can create agents of species a_species (**with** two **attributes**type**and**nature**with** types corresponding **to** the types **of** the shapefile **attributes**) **from** a shapefilethe_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 read(int), the index of the column
```

- Similarly to the creation from shapefile, modelers can create agents from a set of geometries. In this case, one agent per geometry will be created (with the geometry as shape)

```
create species_of(self) from: [square(4),circle(4)];    // 2 agents
   have been created, with shapes respectively square(4) and circle(4)
```

- Created agents are initialized following the rules of their species. If one wants to refer to them after the statement is executed, the returns keyword has to be defined: the agents created will then be referred to by the temporary variable it declares. For instance, the following statement creates 0 to 4 agents of the same species as the sender, and puts them in the temporary variable children for later use.

```
create species (self) number: rnd (4) returns: children;
ask children {
        // ...
}
```

- If one wants to specify a special initialization sequence for the agents created, create provides the same possibilities as ask. This extended syntax is:

```
create a_species number: an_int {
     [statements]
}
```

- The same rules as in ask apply. The only difference is that, for the agents created, the assignments of variables will bypass the initialization defined in species. For instance:

```
create species(self) number: rnd (4) returns: children {
     set location <- myself.location + {rnd (2), rnd (2)}; // tells the
    children to be initially located close to me
     set parent <- myself; // tells the children that their parent is
   me (provided the variable parent is declared in this species)
}
```

- Desprecated uses:

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

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

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

Definition

This statement allows to describe the values that will be displayed on the chart.

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, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified 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)

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

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.

Facets

- **`value`** (list): the values to display. Has to be a matrix, a list or a List of List. Each element can be a number (series/histogram) or a list with two values (XY chart)
- **`legend`** (list), (omissible) : the name of the series: a list of strings (can be a variable with dynamic names)
- **`accumulate_values`** (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- **`color`** (list): list of colors, for heatmaps can be a list of [minColor,maxColor] or [minColor,medColor,maxColor]
- **`fill`** (boolean): Marker filled (true) or not (false), same for all series.
- **`line_visible`** (boolean): Line visible or not (same for all series)
- **`marker`** (boolean): marker visible or not
- **`marker_shape`** (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker. Same one for all series.
- **`marker_size`** (list): the marker sizes to display. Can be a list of numbers (same size for each marker of the series) or a list of list (different sizes by point)
- **`style`** (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded}: Style for the serie (if not the default one sepecified on chart statement)

- **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)

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

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).

Facets

- **value** (any type), (omissible) : The value or values this statement tries to match

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

Definition

This statements allows a value to diffuse among a species on agents (generally on a grid) depending on a given diffusion matrix.

Facets

- **`var`** (an identifier), (omissible) : the variable to be diffused

- **`on`** (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- **`avoid_mask`** (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the proportion value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- **`cycle_length`** (int): the number of diffusion operation applied in one simulation step
- **`mask`** (matrix): a matrix masking the diffusion (matrix created from a image for example). The cells corresponding to the values smaller than "-1" in the mask matrix will not diffuse, and the other will diffuse.
- **`matrix`** (matrix): the diffusion matrix ("kernel" or "filter" in image processing). Can have any size, as long as dimensions are odd values.
- **`method`** (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- **`min_value`** (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.

- **propagation** (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals 'diffusion', the intensity of a signal is shared between its neighbors with respect to 'proportion', 'variation' and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : S' = (S / N / proportion) - variation. The intensity of S is then diminished by S * proportion on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals 'gradient', the original intensity is not modified, and each neighbors receives the intensity : S / proportion - variation. If multiple propagation occur at once, only the maximum intensity is kept on each place. If 'propagation' is not defined, it is assumed that it is equal to 'diffusion'.
- **proportion** (float): a diffusion rate
- **radius** (int): a diffusion radius (in number of cells from the center)
- **variation** (float): an absolute value to decrease at each neighbors

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

Definition

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

Facets

- **`name`** (a label), (omissible) : the identifier of the display
- **`ambient_light`** (any type in [int, rgb]): Allows to define the value of the ambient light either using an int (ambient_light:(125)) or a rgb color ((ambient_light:rgb(255,255,255)). default is rgb(127,127,127,255)
- **`autosave`** (any type in [boolean, point]): Allows to save this display on disk. A value of true/false will save it at a resolution of 500x500. A point can be passed to personalize these dimensions
- **`background`** (rgb): Allows to fill the background of the display with a specific color
- **`camera_interaction`** (boolean): If false, the user will not be able to modify the position and the orientation of the camera, and neither using the ROI. Default is true.
- **`camera_lens`** (int): Allows to define the lens of the camera
- **`camera_look_pos`** (point): Allows to define the direction of the camera
- **`camera_pos`** (any type in [point, agent]): Allows to define the position of the camera
- **`camera_up_vector`** (point): Allows to define the orientation of the camera
- **`draw_diffuse_light`** (boolean): Allows to show/hide a representation of the lights. Default is false.

- **draw_env** (boolean): Allows to enable/disable the drawing of the world shape and the ordinate axes. Default can be configured in Preferences
- focus (geometry): the geometry (or agent) on which the display will (dynamically) focus
- **fullscreen** (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. 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)
- **rotate** (float): Set the angle for the rotation around the Z axis
- **show_fps** (boolean): Allows to enable/disable the drawing of the number of frames per second
- **synchronized** (boolean): Indicates whether the display should be directly synchronized with the simulation
- **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.

Usages

- The general syntax is:

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

- Each display can include different layers (like in a GIS).

```
display gridWithElevationTriangulated type: opengl ambient_light: 100 {
    grid cell elevation: true triangulation: true;
    species people aspect: base;
}
```

Embedments

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

---

display_grid

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).

Facets

- **species** (species), (omissible) : the species of the agents in the grid

- **elevation** (any type in [matrix, float, int, boolean]): Allows to specify the elevation of each cell, if any. Can be a matrix of float (provided it has the same size than the grid), an int or float variable of the grid species, or simply true (in which case, the variable called 'grid_value' is used to compute the elevation of each cell)
- **grayscale** (boolean): if true, givse a grey value to each polygon depending on its elevation (false by default)
- **hexagonal** (boolean):
- **lines** (rgb): the color to draw lines (borders of cells)
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 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.
- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **selectable** (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **text** (boolean): specify whether the attribute used to compute the elevation is displayed on each cells (false by default)
- **texture** (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 rate of the agents (between 0 and 1, 1 means no transparency)
- **triangulation** (boolean): specifies whther the cells will be triangulated: if it is false, they will be displayed as horizontal squares at a given elevation, whereas if it is true, cells will be triangulated and linked to neighbors in order to have a continuous surface (false by default)

Usages

- The general syntax is:

```
display my_display {
   grid ant_grid lines: #black position: { 0.5, 0 } size: {0.5,0.5};
}
```

- To display a grid as a DEM:

```
display my_display {
    grid cell texture: texture_file text: false triangulation: true
   elevation: true;
}
```

- See also: display, agents, chart, event, graphics, image, overlay, display_population,

Embedments

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

---

display_population

Definition

The display_population statement is used using the species keyword. It allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

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.
- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **selectable** (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- trace (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Usages

- The general syntax is:

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

- Species can be superposed on the same plan (be careful with the order, the last one will be above all the others):

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

```
    species agent2 aspect: base;
    species agent3 aspect: base;
}
```

- Each species layer can be placed at a different z value using the opengl display. position:{0,0,0} means the layer will be placed on the ground and position:{0,0,1} means it will be placed at an height equal to the maximum size of the environment.

```
display my_display type: opengl{
    species agent1 aspect: base ;
    species agent2 aspect: base position:{0,0,0.5};
    species agent3 aspect: base position:{0,0,1};
}
```

- See also: display, agents, chart, event, graphics, display_grid, image, overlay,

Embedments

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

---

do

Definition

Allows the agent to execute an (built-in or defined by the modeler) action. For a list of built-in actions available in every species, see RegularSpecies#built-in-actions; for the built-in action available in the `world` agent, see the GlobalSpecies#built-in-actions; for the actions defined by the different skills, see this BuiltInSkills. Finally, see DefiningActionsAndBehaviors#action.

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

Usages

- The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

```
do name_of_action_or_primitive;
```

- In case the action expects one or more arguments to be passed, they are defined by using facets (enclosed tags or a map are now deprecated):

```
do name_of_action_or_primitive arg1: expression1 arg2: expression2;
```

- In case the result of the action needs to be made available to the agent, the action can be called with the agent calling the action (`self` when the agent itself calls the action) instead of `do`; the result should be assigned to a temporary variable:

```
type_returned_by_action result <- self name_of_action_or_primitive [];
```

- In case of an action expecting arguments and returning a value, the following syntax is used:

```
type_returned_by_action result <- self name_of_action_or_primitive [
   arg1::expression1, arg2::expression2];
```

- Deprecated uses: following uses of the `do` statement (still accepted) are now deprecated:

```
// Simple syntax:
do action: name_of_action_or_primitive;

// In case the result of the action needs to be made available to the
   agent, the `returns` keyword can be defined; the result will then be
    referred to by the temporary variable declared in this attribute:
do name_of_action_or_primitive returns: result;
do name_of_action_or_primitive arg1: expression1 arg2: expression2
   returns: result;
type_returned_by_action result <- name_of_action_or_primitive(self, [
   arg1::expression1, arg2::expression2]);

// In case the result of the action needs to be made available to the
   agent
let result <- name_of_action_or_primitive(self, []);

// In case the action expects one or more arguments to be passed, they
   can also be defined by using enclosed `arg` statements, or the `with
   ` facet with a map of parameters:
do name_of_action_or_primitive with: [arg1::expression1, arg2::
   expression2];

or

do name_of_action_or_primitive {
     arg arg1 value: expression1;
     arg arg2 value: expression2;
     ...
}
```

Embedments

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

draw

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.

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
- **empty** (boolean): a condition specifying whether the geometry is empty or full
- **end_arrow** (any type in [int, float]): the size of the arrow, located at the end of the drawn geometry
- **font** (any type in [19, string]): the font used to draw the text, if any. Applying this facet to geometries or images has no effect. You can construct here your font with the operator "font". ex : **font:** **font**("Helvetica", 20 , #plain)
- **lighted** (boolean): Whether the object should be lighted or not (only applicable in the context of opengl displays)

- **material** (25): Set a particular material to the object (only if you use it in an "opengl2" display).
- **perspective** (boolean): Whether to render the text in perspective or facing the user. Default is true.
- **rotate** (any type in [float, int, pair]): orientation of the shape/text/icon; can be either an int/float (angle) or a pair float::point (angle::rotation axis). The rotation axis, when expressed as an angle, is by defaut `{0,0,1}`
- **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 in [string, list, file]): 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

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

Definition

This statement cannot be used alone

Facets

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

Definition

enables to make conscious or unconscious emotional contagion

Facets

- **emotion_detected** (546706): the emotion that will start the contagion
- **name** (an identifier), (omissible) : the identifier of the emotional contagion
- **charisma** (float): The charisma value of the perceived agent (between 0 and 1)
- **decay** (float): The decay value of the emotion added to the agent
- **emotion_created** (546706): 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

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

Definition

applay a sanction if the norm specified is violated, or a reward if the norm is applied by the perceived agent

Facets

- **name** (an identifier), (omissible) : the identifier of the enforcement
- law (string): The law to enforce
- norm (string): The norm to enforce
- **obligation** (546704): 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

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

Definition

In an FSM architecture, `enter` introduces a sequence of statements to execute upon entering a state.

Facets

Usages

- In the following example, at the step it enters into the state s_init, the message 'Enter in s_init' is displayed followed by the display of the state name:

```
state s_init {
    enter { write "Enter in" + state; }
        write "Enter in" + state;
    }
    write state;
}
```

- See also: state, exit, transition,

Embedments

- The `enter` statement is of type: **Sequence of statements or action**
- The `enter` statement can be embedded into: state,
- The `enter` statement embeds statements:

---

equation

Definition

The equation statement is used to create an equation system from several single equations.

Facets

- **name** (an identifier), (omissible) : the equation identifier
- **params** (list): the list of parameters used in predefined equation systems
- **simultaneously** (list): a list of species containing a system of equations (all systems will be solved simultaneously)
- **type** (an identifier), takes values in: {SI, SIS, SIR, SIRS, SEIR, LV}: the choice of one among classical models (SI, SIS, SIR, SIRS, SEIR, LV)
- **vars** (list): the list of variables used in predefined equation systems

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 Equations 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

Definition

The statement makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

Facets

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

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

Definition

event allows to interact with the simulation by capturing mouse or key events and doing an action. This action needs to be defined in 'global' 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

Facets

- **name** (an identifier), (omissible) : the type of event captured: can be "mouse_up", "mouse_down", "mouse_move", "mouse_exit", "mouse_enter" or a character

- **action** (26): Either a block of statements to execute in the context of the experiment or 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 peripheric used to generate events. Defaults to 'default', which encompasses keyboard and mouse
- **unused** (an identifier), takes values in: {mouse_up, mouse_down, mouse_move, mouse_enter, mouse_exit}: an unused facet that serves only for the purpose of declaring the string values

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, overlay, display_population,

Embedments

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

---

exhaustive

Definition

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way. See BatchExperiments.

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 to use (either min or max)
- **maximize** (float): the value the algorithm tries to maximize
- **minimize** (float): the value the algorithm tries to minimize

Usages

- As other batch methods, the basic syntax of the exhaustive statement uses **method** exhaustive instead of the expected exhaustive **name**: **id** :

```
method exhaustive [facet: value];
```

- For example:

```
method exhaustive maximize: food_gathered;
```

Embedments

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

---

exit

Definition

In an FSM architecture, exit introduces a sequence of statements to execute right before exiting the state.

Facets

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

Definition

Declaration of a particular type of agent that can manage simulations

Facets

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

- **title** (a label):

- **type** (a label), takes values in: {batch, memorize, gui, test, headless}: 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 frequence 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 hom 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)

Usages

Embedments

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

---

focus

Definition

enables to directly add a belief from the variable of a perceived specie.

Facets

- **agent_cause** (agent): the agentCause value of the created belief (can be nil
- **belief** (546704): The predicate to focus on the beliefs of the other agent
- **desire** (546704): The predicate to focus on the desires of the other agent
- **emotion** (546706): 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** (546704): 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** (546704): 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

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

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

Facets

- **value** (any type), (omissible) : The agent, list of agents, geometry to focus on

Usages

- Focuses on an agent, a geometry, a set of agents, etc. . .)

```
focus_on my_species(0);
```

Embedments

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

---

genetic

Definition

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article and BatchExperiments. 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).

Facets

- **`name`** (an identifier), (omissible) : The name of this method. For internal use only
- **`aggregation`** (a label), takes values in: {min, max}: the agregation 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 determistic one (keep the best solutions)

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

Definition

graphics allows the modeler to freely draw shapes/geometries/texts in a display without having to define a species. It works exactly like a species RegularSpecies#the-aspect-statement: the Statements#draw can be used in the same way.

Facets

- **name** (a label), (omissible) : the human readable title of the graphics
- **fading** (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false

- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are 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.
- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- trace (any type in [boolean, int]): Allows to aggregate the visualization at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Usages

- The general syntax is:

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

- See also: display, agents, chart, event, graphics, display_grid, image, overlay, display_population,

Embedments

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

- The graphics statement embeds statements:

---

highlight

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

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

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

Definition

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article and BatchExperiments.

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **aggregation** (a label), takes values in: {min, max}: the agregation method
- **iter_max** (int): number of iterations
- **maximize** (float): the value the algorithm tries to maximize
- **minimize** (float): the value the algorithm tries to minimize

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

Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true.

Facets

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

Usages

- The generic syntax is:

```
if bool_expr {
    [statements]
}
```

- Optionally, the statements to execute when the condition evaluates to false can be defined in a following statement else. The syntax then becomes:

```
if bool_expr {
    [statements]
}
else {
    [statements]
}
string valTrue <- "";
if true {
    valTrue <- "true";
}
else {
    valTrue <- "false";
}//valTrue equals "true"
string valFalse <- "";
if false {
    valFalse <- "true";
```

```
}
else {
    valFalse <- "false";
}//valFalse equals "false"
```

- ifs and elses can be imbricated as needed. For instance:

```
if bool_expr {
    [statements]
}
else if bool_expr2 {
    [statements]
}
else {
    [statements]
}
```

Embedments

- The `if` statement is of type: **Sequence of statements or action**
- The `if` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `if` statement embeds statements: else,

---

image

Definition

`image` 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.

Facets

- **name** (any type in [string, file]), (omissible) : Human readable title of the image layer
- **color** (rgb): in the case of a shapefile, this the color used to fill in geometries of the shapefile. In the case of an image, it is used to tint the image
- **file** (any type in [string, file]): the name/path of the image (in the case of a raster image)
- **gis** (any type in [file, string]): the name/path of the shape file (to display a shapefile as background, without creating agents from it)
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in [0,1[, the position is relative to the size of the environment (e.g. {0.5,0.5} refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 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.
- **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)
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **transparency** (float): the transparency rate of the agents (between 0 and 1, 1 means no transparency)

Usages

- The general syntax is:

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

- For instance, in the case of a bitmap image

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

- Or in the case of a shapefile:

```
display my_display {
   image testGIS gis: "../includes/building.shp" color: rgb('blue');
}
```

- It is also possible to superpose images on different layers in the same way as for species using opengl display:

```
display my_display {
  image image1 file:"../images/image1.jpg";
  image image2 file:"../images/image2.jpg";
  image image3 file:"../images/image3.jpg" position: {0,0,0.5};
}
```

- See also: display, agents, chart, event, graphics, display_grid, overlay, display_population,

Embedments

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

---

inspect

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.

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

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

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.

Facets

- **name** (an identifier), (omissible) : The name of the law
- **all** (boolean): add an obligation for each belief
- **belief** (546704): The mandatory belief
- **beliefs** (list): The mandatory beliefs
- **lifetime** (int): the lifetime value of the mental state created
- **new_obligation** (546704): 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 stregth of the mental state created
- **threshold** (float): Threshold linked to the obedience value.
- **when** (boolean):

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

Definition

Represents the layout of the display views of simulations and experiments

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
- **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)

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

Definition

Allows to declare a temporary variable of the specified type and to initialize it with a value

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

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

Definition

`light` allows to define diffusion lights in your 3D display.

Facets

- **id** (int), (omissible) : a number from 1 to 7 to specify which light we are using
- **active** (boolean): a boolean expression telling if you want this light to be switch on or not. (default value : true)
- **color** (any type in [int, rgb]): an int / rgb / rgba value to specify the color and the intensity of the light. (default value : (127,127,127,255)).
- **direction** (point): the direction of the light (only for direction and spot light). (default value : {0.5,0.5,-1})
- **draw_light** (boolean): draw or not the light. (default value : false).
- **linear_attenuation** (float): the linear attenuation of the positionnal light. (default value : 0)
- **position** (point): the position of the light (only for point and spot light). (default value : {0,0,1})
- **quadratic_attenuation** (float): the linear attenuation of the positionnal light. (default value : 0)
- **spot_angle** (float): the angle of the spot light in degree (only for spot light). (default value : 45)
- **type** (a label): the type of light to create. A value among {point, direction, spot}. (default value : direction)
- **update** (boolean): specify if the light has to be updated. (default value : true).

Usages

- The general syntax is:

```
light 1 type:point position:{20,20,20} color:255, linear_attenuation
    :0.01 quadratic_attenuation:0.0001 draw_light:true update:false
light 2 type:spot position:{20,20,20} direction:{0,0,-1} color:255
    spot_angle:25 linear_attenuation:0.01 quadratic_attenuation:0.0001
    draw_light:true update:false
light 3 type:point direction:{1,1,-1} color:255 draw_light:true update:
    false
```

- See also: display,

Embedments

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

---

loop

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.

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

Usages

- The basic syntax for repeating a fixed number of times a set of statements is:

```
loop times: an_int_expression {
    // [statements]
}
```

- The basic syntax for repeating a set of statements while a condition holds is:

```
loop while: a_bool_expression {
    // [statements]
}
```

- The basic syntax for repeating a set of statements by progressing over a container of a point is:

```
loop a_temp_var over: a_collection_expression {
    // [statements]
}
```

- The basic syntax for repeating a set of statements while an index iterates over a range of values with a fixed step of 1 is:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 {
    // [statements]
}
```

- The incrementation step of the index can also be chosen:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 step:
   int_expression3 {
    // [statements]
}
```

- In these latter three cases, the name facet designates the name of a temporary variable, whose scope is the loop, and that takes, in turn, the value of each of the element of the list (or each value in the interval). For example, in the first instance of the “loop over” syntax :

```
int a <- 0;
loop i over: [10, 20, 30] {
    a <- a + i;
} // a now equals 60
```

- The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to facets take an integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the inclusive interval (i.e. [to, from]). If the step is not defined, it is assumed to be equal to 1 or -1, depending on the direction of the range. If it is defined, its sign will be respected, so that a positive step will never allow the loop to enter a loop from i to j where i is greater than j

```
list the_list <-list (species_of (self));
loop i from: 0 to: length (the_list) - 1 {
     ask the_list at i {
        // ...
     }
} // every  agent of the list is asked to do something
```

Embedments

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

---

match

Definition

In a switch. . . match structure, the value of each match block is compared to the value in the switch. If they match, the embedded statement set is executed. Three kinds of match can be used

Facets

- **`value`** (any type), (omissible) : The value or values this statement tries to match

Usages

- match block is executed if the switch value is equals to the value of the match:

```
switch 3 {
   match 1 {write "Match 1"; }
   match 3 {write "Match 2"; }
}
```

- match_between block is executed if the switch value is in the interval given in value of the match_between:

```
switch 3 {
   match_between [1,2] {write "Match OK between [1,2]"; }
   match_between [2,5] {write "Match OK between [2,5]"; }
}
```

- match_one block is executed if the switch value is equals to one of the values of the match_one:

```
switch 3 {
   match_one [0,1,2] {write "Match OK with one of [0,1,2]"; }
   match_between [2,3,4,5] {write "Match OK with one of [2,3,4,5]"; }
}
```

- See also: switch, default,

Embedments

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

migrate

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.

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

Usages

- It can be used in a 3-levels model, in case where individual agents can be captured into group meso agents and groups into clouds macro agents. migrate is used to allows agents captured by groups to migrate into clouds. See the model 'Balls, Groups and Clouds.gaml' in the library.

```
migrate ball_in_group target: ball_in_cloud;
```

- See also: capture, release,

Embedments

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

monitor

Definition

A monitor allows to follow the value of an arbitrary expression in GAML.

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)

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:

---

norm

Definition

a norm indicates what action the agent has to do in a certain context and with and obedience value higher than the threshold

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 instananeous
- **intention** (546704): the intention triggering the norm
- **lifetime** (int): the lifetime of the norm
- **obligation** (546704): 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

Usages

Embedments

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

---

output

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.

Facets

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

Definition

Represents an output that writes the result of expressions into a 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

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

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.

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.

- **right** (any type): an expression that will be evaluated and displayed in the right section of the bottom overlay
- **rounded** (boolean): Whether or not the rectangular shape of the overlay should be rounded. True by default
- **size** (point): extent of the layer in the view from its position. Coordinates in [0,1[are treated as percentages of the total surface of the view, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Unlike 'position', no elevation can be provided with the z coordinate
- **transparency** (float): the transparency rate of the overlay (between 0 and 1, 1 means no transparency) when it is displayed inside the view. The bottom overlay will remain at 0.75

Usages

- To display information in the bottom overlay, the syntax is:

```
overlay "Cycle: " + (cycle) center: "Duration: " + total_duration + "ms
    " right: "Model time: " + as_date(time,"") color: [#yellow, #orange,
     #yellow];
```

- See also: display, agents, chart, event, graphics, display_grid, image, display_population,

Embedments

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

---

parameter

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.

Facets

- **var** (an identifier): the name of the variable (that should be declared in the global)
- **name** (a label), (omissible) : The message displayed in the interface
- **among** (list): the list of possible values
- **category** (a label): a category label, used to group parameters in the interface
- **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 (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 (they are otherwise disabled)
- **init** (any type): the init value
- **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
- **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

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

Definition

Allow the agent, with a bdi architecture, to perceive others agents

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`** (546706): The emotion needed to do the perception
- **`in`** (any type in [float, geometry]): a float or a geometry. If it is a float, it's a radius of a detection area. If it is a geometry, it is the area of detection of others species.

- **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

Usages

- the basic syntax to perceive agents inside a circle of perception

```
perceive name_of-perception target: the_agents_you_want_to_perceive in:
    a_distance when: a_certain_condition {
Here you are in the context of the perceived agents. To refer to the
   agent who does the perception, use myself.
If you want to make an action (such as adding a belief for example),
   use ask myself{ do the_action}
}
```

Embedments

- The perceive statement is of type: **Sequence of statements or action**
- The perceive statement can be embedded into: Species, Model,
- The perceive statement embeds statements:

---

permanent

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.

Facets

- **tabs** (boolean): Whether the displays should show their tab or not
- **toolbars** (boolean): Whether the displays should show their toolbar or not

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

Definition

define an action plan performed by an agent using the BDI engine

Facets

- **name** (an identifier), (omissible) :
- **emotion** (546706):
- **finished_when** (boolean):
- **instantaneous** (boolean):
- **intention** (546704):
- **priority** (float):
- **threshold** (float):
- **when** (boolean):

Usages

Embedments

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

---

put

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.

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

Usages

- The allowed parameters configurations are the following ones:

```
put expr at: expr in: expr_container;
put all: expr in: expr_container;
```

- In the case of a list, the position should an integer in the bound of the list. The facet all: is used to replace all the elements of the list by the given value.

```
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.

```
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.

```
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

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 BatchExperiments.

Facets

- **`name`** (an identifier), (omissible) :
- **`aggregation`** (a label), takes values in: {min, max}: the agregation method
- **`cycle_size_max`** (int): minimal size of the considered cycles
- **`cycle_size_min`** (int): maximal size of the considered cycles
- **`iter_max`** (int): number of iterations
- **`maximize`** (float): the value the algorithm tries to maximize
- **`minimize`** (float): the value the algorithm tries to minimize
- **`nb_tests_wthout_col_max`** (int): number of movements without collision before shortening the tabu list

- **tabu_list_size_init** (int): initial size of the tabu list
- **tabu_list_size_max** (int): maximal size of the tabu list
- **tabu_list_size_min** (int): minimal size of the tabu list

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

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.

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.

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

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. MultiLevelArchitecture#declaration-of-micro-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).

Facets

- **target** (any type in [agent, list, 27]), (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)

Usages

- We consider the following species. Agents of "C" species can be released from a "B" agent to become agents of "A" species. Agents of "D" species cannot be released from the "A" agent because species "D" has no parent species.

```
species A {
...
}
species B {
...
   species C parent: A {
   ...
   }
   species D {
   ...
   }
...
}
```

- To release all "C" agents from a "B" agent, agent "C" has to execute the following statement. The "C" agent will change to "A" agent. The won't consider "B" agent as their macro-agent (host) anymore. Their host (macro-agent) will the be the host (macro-agent) of the "B" agent.

```
release list(C);
```

- The modeler can specify the new host and the new species of the released agents:

```
release list (C) as: new_species in: new host;
```

- See also: capture,

Embedments

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

---

remove

Definition

Allows the agent to remove an element from a container (a list, matrix, map...).

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

Usages

- This statement should be used in the following ways, depending on the kind of container used and the expected action on it:

```
remove expr from: expr_container;
remove index: expr from: expr_container;
remove key: expr from: expr_container;
remove all: expr from: expr_container;
```

- In the case of list, the facet `item:` is used to remove the first occurence of a given expression, whereas `all` is used to remove all the occurrences of the given expression.

```
list<int> removeList <- [3,2,1,2,3];remove 2 from: removeList;//
    removeList equals [3,1,2,3]remove 3 all: true from: removeList;//
    removeList equals [1,2]remove index: 1 from: removeList;//removeList
     equals [1]
```

- In the case of map, the facet `key:` is used to remove the pair identified by the given key.

```
map<string,int> removeMap <- ["x"::5, "y"::7, "z"::7];remove key: "x"
    from: removeMap;//removeMap equals ["y"::7, "z"::7]remove 7 all:
    true from: removeMap;//removeMap equals map([])
```

- In addition, a map a be managed as a list with pair key as index. Given that, facets item:, all: and index: can be used in the same way:

```
map<string,int> removeMapList <- ["x"::5, "y"::7, "z"::7, "t"::5];
    remove 7 from: removeMapList;//removeMapList equals ["x"::5, "z"::7,
     "t"::5]remove [5,7] all: true from: removeMapList;//removeMapList
    equals ["t"::5]remove index: "t" from: removeMapList;//removeMapList
     equals map([])
```

- In the case of a graph, both edges and nodes can be removes using node: and edge facets. If a node is removed, all edges to and from this node are also removed.

```
graph removeGraph <- as_edge_graph([{1,2}::{3,4},{3,4}::{5,6}]);
remove node: {1,2} from: removeGraph;
remove node(1,2) from: removeGraph;
list var <- removeGraph.vertices; // var equals [{3,4},{5,6}]
list var <- removeGraph.edges; // var equals [polyline({3,4}::{5,6})]
remove edge: {3,4}::{5,6} from: removeGraph;
remove edge({3,4},{5,6}) from: removeGraph;
list var <- removeGraph.vertices; // var equals [{3,4},{5,6}]
list var <- removeGraph.edges; // var equals []
```

- In the case of an agent or a shape, remove allows to remove an attribute from the attributes map of the receiver. However, for agents, it will only remove attributes that have been added dynamically, not the ones defined in the species or in its built-in parent.

```
global {
   init {
      create speciesRemove;
      speciesRemove sR <- speciesRemove(0);     // sR.a now equals 100
      remove key:"a" from: sR;  // sR.a now equals nil
   }
}

species speciesRemove {
   int a <- 100;
}
```

- This statement can not be used on *matrix*.
- See also: add, put,

Embedments

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

return

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 DefiningActionsAndBehaviors#action.

Facets

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

Usages

- Example:

```
string foo {
     return "foo";
}

reflex {
    string foo_result <- foo();     // foos_result is now equals to "
   foo"
}
```

- In the specific case one wants an agent to ask another agent to execute a statement with a return, it can be done similarly to:

```
// In Species A:
string foo_different {
     return "foo_not_same";
}
/// ....
// In Species B:
reflex writing {
    string temp <- some_agent_A.foo_different [];   // temp is now
   equals to "foo_not_same"
}
```

Embedments

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

---

rule

Definition

A simple definition of a rule (set of statements which execution depend on a condition and a priority).

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

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

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.

Facets

- **name** (an identifier), (omissible) : The name of the rule
- **all** (boolean): add a desire for each belief
- **belief** (546704): The mandatory belief
- **beliefs** (list): The mandatory beliefs
- **desire** (546704): The mandatory desire
- **desires** (list): The mandatory desires
- **emotion** (546706): The mandatory emotion
- **emotions** (list): The mandatory emotions
- **ideal** (546704): The mandatory ideal
- **ideals** (list): The mandatory ideals
- **lifetime** (int): the lifetime value of the mental state created
- **new_belief** (546704): The belief that will be added
- **new_beliefs** (list): The belief that will be added
- **new_desire** (546704): The desire that will be added
- **new_desires** (list): The desire that will be added
- **new_emotion** (546706): The emotion that will be added
- **new_emotions** (list): The emotion that will be added
- **new_ideal** (546704): 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** (546704): The uncertainty that will be added
- **obligation** (546704): 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** (546704): The belief that will be removed

- **remove_beliefs** (list): The belief that will be removed
- **remove_desire** (546704): The desire that will be removed
- **remove_desires** (list): The desire that will be removed
- **remove_emotion** (546706): The emotion that will be removed
- **remove_emotions** (list): The emotion that will be removed
- **remove_ideal** (546704): The ideal that will be removed
- **remove_ideals** (list): The ideals that will be removed
- **remove_intention** (546704): The intention that will be removed
- **remove_obligation** (546704): 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** (546704): The uncertainty that will be removed
- **strength** (any type in [float, int]): The stregth of the mental state created
- **threshold** (float): Threshold linked to the emotion.
- **uncertainties** (list): The mandatory uncertainties
- **uncertainty** (546704): The mandatory uncertainty
- **when** (boolean):

Usages

- Other examples of use:

```
rule belief: new_predicate("test") when: flip(0.5) new_desire:
   new_predicate("test")
```

Embedments

- The `rule` statement is of type: **Single statement**
- The `rule` statement can be embedded into: simple_bdi, parallel_bdi, Species, Model,
- The `rule` statement embeds statements:

run

Facets

- **name** (string), (omissible) :

- **of** (string):
- **core** (int):
- **end_cycle** (int):
- **seed** (int):
- **with_output** (map):
- **with_param** (map):

Embedments

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

---

sanction

Definition

declare the actions an agent execute when enforcing norms of others during a perception

Facets

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

Usages

Embedments

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

---

save

Definition

Allows to save data in a file. The type of file can be “shp”, “asc”, “geotiff”, “text” or “csv”.

Facets

- `data` (any type), (omissible) : any expression, that will be saved in the file
- **`attributes`** (map): Allows to specify the attributes of a shape file where agents are saved. Must be expressed as a literal map. 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
- **`header`** (boolean): an expression that evaluates to a boolean, specifying whether the save will write a header if the file does not exist
- **`rewrite`** (boolean): an expression that evaluates to a boolean, specifying whether the save will ecrase the file or append data at the end of it. Default is true
- **`to`** (string): an expression that evaluates to an string, the path to the file, or directly to a file
- **`type`** (an identifier), takes values in: {shp, text, csv, asc, geotiff, image, kml, kmz, json}: an expression that evaluates to an string, the type of the output file (it can be only “shp”, “asc”, “geotiff”, “image”, “text” or “csv”)

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->"  + name + ":" + location) to: "save_data.txt
   " type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the values of all attributes of all the agents of a species into a csv (with optional attributes):

```
save species_of(self) to: "save_csvfile.csv" type: "csv" header: false;
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp" with: [name
   ::"nameAgent", location::"locationAgent"] crs: "EPSG:4326";
```

- To save the grid_value attributes of all the cells of a grid into an ESRI ASCII Raster file:

```
save grid to: "save_grid.asc" type: "asc";
```

- To save the grid_value attributes of all the cells of a grid into geotiff:

```
save grid to: "save_grid.tif" type: "geotiff";
```

- To save the grid_value attributes of all the cells of a grid into png (with a worldfile):

```
save grid to: "save_grid.png" type: "image";
```

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

Embedments

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

---

set

Definition

Allows to assign a value to the variable or attribute specified

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

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

Definition

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

Facets

Usages

- As every test should be independent from the others, the setup will mainly contain initialization of variables that will be used in each test.

```
species Tester {
    int val_to_test;

    setup {
        val_to_test <- 0;
    }

    test t1 {
       // [set of instructions, including asserts]
    }
}
```

- See also: test, assert,

Embedments

- The `setup` statement is of type: **Sequence of statements or action**
- The `setup` statement can be embedded into: Species, Experiment, Model,
- The `setup` statement embeds statements:

---

simulate

Definition

Allows an agent, the sender agent (that can be the GlobalSpecies), to ask another (or other) agent(s) to perform a set of statements. It obeys the following syntax, where the target attribute denotes the receiver agent(s):

Facets

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

Usages

- Other examples of use:

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

Embedments

- The `simulate` statement is of type: **Single statement**
- The `simulate` statement can be embedded into: chart, Experiment, Species, Behavior, Sequence of statements or action,
- The `simulate` statement embeds statements:

---

socialize

Definition

enables to directly add a social link from a perceived agent.

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

Usages

- Other examples of use:

```
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

Definition

Solves all equations which matched the given name, with all systems of agents that should solved simultaneously.

Facets

- **`equation`** (an identifier), (omissible) : the equation system identifier to be numerically solved
- **`integrated_times`** (list): time interval inside integration process
- **`integrated_values`** (list): list of variables's value inside integration process
- **`max_step`** (float): maximal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- **`method`** (string): integration method (can be one of "Euler", "ThreeEighthes", "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 (defaut 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 (defaut value: cycle*step, the time at the begining of the current cycle.)

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

Definition

The species statement allows modelers to define new species in the model. `global` and `grid` are speciel cases of species: `global` being the definition of the global agent (which has automatically one instance, world) and `grid` being a species with a grid topology.

Facets

- **name** (an identifier), (omissible) : the identifier of the species
- **cell_height** (float): (grid only), the height of the cells of the grid
- **cell_width** (float): (grid only), the width of the cells of the grid
- **compile** (boolean):
- **control** (22): defines the architecture of the species (e.g. fsm...)
- **edge_species** (species): In the case of a species defining a graph topology for its instances (nodes of the graph), specifies the species to use for representing the edges
- **file** (file): (grid only), a bitmap file that will be loaded at runtime so that the value of each pixel can be assigned to the attribute 'grid_value'
- **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. For instance,

'species a schedules: (10 among a)' will result in a population that schedules only 10 of its own agents every cycle. 'species b schedules: []' will prevent the agents of 'b' to be scheduled. Note that the scope of agents covered here can be larger than the population, which allows to build complex scheduling controls; for instance, defining 'global schedules: [] {...} species b schedules: []; species c schedules: b + world;' allows to simulate a model where the agents of b are scheduled first, followed by the world, without even having to create an instance of c.

- **skills** (list): The list of skills that will be made available to the instances of this species. Each new skill provides attributes and actions that will be added to the ones defined in this species
- **topology** (topology): The topology of the population of agents defined by this species. In case of nested species, it can for example be the shape of the macro-agent. In case of grid or graph species, the topology is automatically computed and cannot be redefined
- **torus** (boolean): is the topology toric (defaut: false). Needs to be defined on the global species.
- **use_individual_shapes** (boolean): (grid only),(true by default). Allows to specify whether or not the agents of the grid will have distinct geometries. If set to false, they will all have simpler proxy geometries
- **use_neighbors_cache** (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbors cache used for grids. Note that if a diffusion of variable occurs, GAMA will emit a warning and automatically switch to a caching version
- **use_regular_agents** (boolean): (grid only),(true by default). Allows to specify if the agents of the grid are regular agents (like those of any other species) or minimal ones (which can't have sub-populations, can't inherit from a regular species, etc.)
- **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)

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:

---

start_simulation

Facets

- **`name`** (string), (omissible) :

- **`of`** (string):
- **`seed`** (int):
- **`with_param`** (map):

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

Definition

A state, like a reflex, can contains several statements that can be executed at each time step by the agent.

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)

Usages

- Here is an exemple integrating 2 states and the statements in the FSM architecture:

```
state s_init initial: true {
    enter { write "Enter in" + state; }
        write "Enter in" + state;
    }

    write state;
```

```
        transition to: s1 when: (cycle > 2) {
            write "transition s_init -> s1";
        }

        exit {
            write "EXIT from "+state;
        }
    }
    state s1 {

    enter {write 'Enter in '+state;}

    write state;

    exit {write 'EXIT from '+state;}
}
```

- See also: enter, exit, transition,

Embedments

- The `state` statement is of type: **Behavior**
- The `state` statement can be embedded into: fsm, Species, Experiment, Model,
- The `state` statement embeds statements: enter, exit,

---

status

Definition

The statement makes the agent output an arbitrary message in the status box.

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

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:

---

switch

Definition

The "switch... match" statement is a powerful replacement for imbricated "if ... else ..." constructs. All the blocks that match are executed in the order they are defined. The block prefixed by default is executed only if none have matched (otherwise it is not).

Facets

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

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

Definition

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article and BatchExperiments.

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **aggregation** (a label), takes values in: {min, max}: the agregation method
- **iter_max** (int): number of iterations
- **maximize** (float): the value the algorithm tries to maximize
- **minimize** (float): the value the algorithm tries to minimize
- **tabu_list_size** (int): size of the tabu list

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

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.

Facets

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

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

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.

Facets

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

Usages

- An example of use:

```
species Tester {
    // set of attributes that will be used in test

    setup {
        // [set of instructions... in particular initializations]
    }

    test t1 {
       // [set of instructions, including asserts]
    }
}
```

- See also: setup, assert,

Embedments

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

---

trace

Definition

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

Facets

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

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.

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

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

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

Facets

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

Definition

enables to directly copy an emotion presents in the perceived specie.

Facets

- **`emotion`** (546706): the emotion that will be copied with the contagion
- **`name`** (an identifier), (omissible) : the identifier of the unconscious contagion
- **`charisma`** (float): The charisma value of the perceived agent (between 0 and 1)
- **`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

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

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.

Facets

- **name** (a label), (omissible) : the identifier of the user_command
- action (26): 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

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

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.

Facets

- **name** (an identifier), (omissible) : The name of the panel
- **initial** (boolean): Whether or not this panel will be the initial one

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

Definition

It allows to let the user define the value of a variable.

Facets

- **returns** (a new identifier): a new local variable containing the value given by the user
- **name** (a label), (omissible) : the displayed name
- **among** (list): the set of acceptable values for the variable
- **init** (any type): the init value
- **max** (float): the maximum value
- **min** (float): the minimum value
- **slider** (boolean): Whether to display a slider or not when applicable
- **type** (a datatype identifier): the variable type

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

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.

Facets

- **name** (an identifier), (omissible) : The name of the panel
- **initial** (boolean): Whether or not this panel will be the initial one

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

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.

Facets

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

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

Definition

Allows to declare an attribute of a species or an experiment

Facets

- **`name`** (a new identifier), (omissible) : The name of the attribute
- **`category`** (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- **`const`** (boolean): Indicates whether this attribute can be subsequently modified or not
- **`function`** (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- **`index`** (a datatype identifier): The type of the key used to retrieve the contents of this attribute
- **`init`** (any type): The initial value of the attribute
- **`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

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

Definition

Allows to declare an attribute of a species or experiment

Facets

- **`name`** (a new identifier), (omissible) : The name of the attribute
- **`among`** (list): A list of constant values among which the attribute can take its value
- **`category`** (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- **`const`** (boolean): Indicates whether this attribute can be subsequently modified or not
- **`function`** (any type in [int, float]): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- **`init`** (any type in [int, float]): The initial value of the attribute
- **`max`** (any type in [int, float]): The maximum value this attribute can take.
- **`min`** (any type in [int, float]): The minimum value this attribute can take
- **`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** (int): 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' or 'float'
- **update** (any type in [int, float]): An expression that will be evaluated each cycle to compute a new value for the attribute

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

Definition

Allows to declare an attribute of a species or an experiment

Facets

- **name** (a new identifier), (omissible) : The name of the attribute
- **among** (list): A list of constant values among which the attribute can take its value
- **category** (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- **const** (boolean): Indicates whether this attribute can be subsequently modified or not
- **function** (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:' and 'update:'
- **index** (a datatype identifier): The type of the index used to retrieve elements if the type of the attribute is a container type

- **init** (any type): The initial value of the attribute
- **of** (a datatype identifier): The type of the elements contained in the type of this attribute if it is a container type
- **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

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

Definition

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

Facets

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

Definition

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

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

Usages

- Outputting a message

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

Embedments

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

Chapter 94

Types

A variable's or expression's *type* (or *data type*) determines the values it can take, plus the operations that can be performed on or with it. GAML is a statically-typed language, which means that the type of an expression is always known at compile time, and is even enforced with casting operations. There are 4 categories of types:

- primitive types, declared as keyword in the language,
- complex types, also declared as keyword in the language,
- parametric types, a refinement of complex types (mainly children of container) that is dynamically constructed using an enclosing type, a contents type and a key type,
- species types, dynamically constructed from the species declarations made by the modeler (and the built-in species present).

The hierarchy of types in GAML (only primitive and complex types are displayed here, of course, as the other ones are model-dependent) is the following:

Table of contents

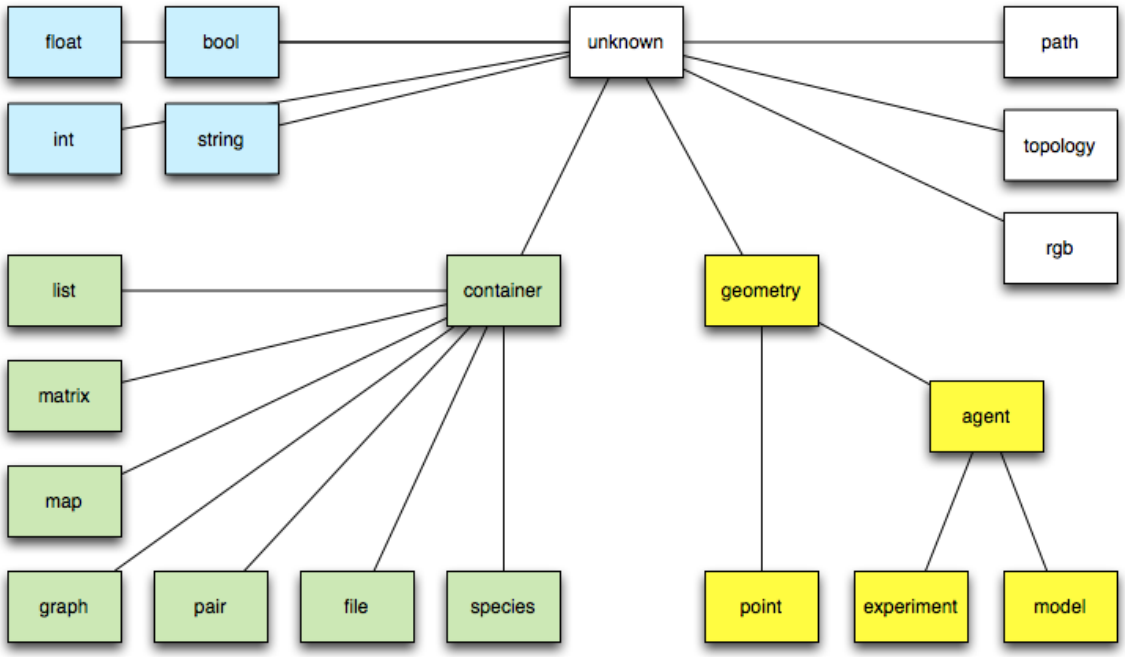


Figure 94.1: images/types_hierarchy.png

Primitive built-in types

bool

- **Definition:** primitive datatype providing two values: `true` or `false`.
- **Litteral declaration:** both `true` or `false` are interpreted as boolean constants.
- **Other declarations:** expressions that require a boolean operand often directly apply a casting to bool to their operand. It is a convenient way to directly obtain a bool value.

```
bool (0) -> false
```

Top of the page

float

- **Definition:** primitive datatype holding floating point values, its absolute value is comprised between 4.9E-324 and 1.8E308.
- **Comments:** this datatype is internally backed up by the Java double datatype.
- **Litteral declaration:** decimal notation 123.45 or exponential notation 123e45 are supported.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to float to their operand. Using it is a way to obtain a float constant.

```
float (12) -> 12.0
```

Top of the page

int

- **Definition:** primitive datatype holding integer values comprised between -2147483648 and 2147483647 (i.e. between `-2^31` and `2^31 - 1`.
- **Comments:** this datatype is internally backed up by the Java int datatype.
- **Litteral declaration:** decimal notation like 1, 256790 or hexadecimal notation like #1209FF are automatically interpreted.
- **Other declarations:** expressions that require an integer operand often directly apply a casting to int to their operand. Using it is a way to obtain an integer constant.

```
int (234.5) -> 234.
```

Top of the page

string

- **Definition:** a datatype holding a sequence of characters.
- **Comments:** this datatype is internally backed up by the Java String class. However, contrary to Java, strings are considered as a primitive type, which means they do not contain character objects. This can be seen when casting a string to a list using the list operator: the result is a list of one-character strings, not a list of characters.

- **Litteral declaration:** a sequence of characters enclosed in quotes, like 'this is a string' . If one wants to literally declare strings that contain quotes, one has to double these quotes in the declaration. Strings accept escape characters like `\n` (newline), `\r` (carriage return), `\t` (tabulation), as well as any Unicode character (`\uXXXX`).
- **Other declarations:** see string
- **Example:** see string operators.

Top of the page

Complex built-in types

Contrarily to primitive built-in types, complex types have often various attributes. They can be accessed in the same way as attributes of agents:

```
complex_type nom_var <- init_var;
ltype_attr attr_var <- nom_var.attr_name;
```

For example:

```
file fileText <- file("../data/cell.Data");
bool fileTextReadable <- fileText.readable;
```

agent

- **Definition:** a generic datatype that represents an agent whatever its actual species.
- **Comments:** This datatype is barely used, since species can be directly used as datatypes themselves.
- **Declaration:** the agent casting operator can be applied to an int (to get the agent with this unique index), a string (to get the agent with this name).

Top of the page

container

- **Definition:** a generic datatype that represents a collection of data.
- **Comments:** a container variable can be a list, a matrix, a map... Conversely, each list, matrix, and map is a kind of container. In consequence, every container can be used in container-related operators.
- **See also:** Container operators
- **Declaration:**

```
container c  <- [1,2,3];
container c  <- matrix [[1,2,3],[4,5,6]];
container c  <- map ["x"::5, "y"::12];
container c  <- list species1;
```

Top of the page

file

- **Definition:** a datatype that represents a file.
- **Built-in attributes:**
  - name (type = string): the name of the represented file (with its extension)
  - extension(type = string): the extension of the file
  - path (type = string): the absolute path of the file
    - readable (type = bool, read-only): a flag expressing whether the file is readable
    - writable (type = bool, read-only): a flag expressing whether the file is writable
    - exists (type = bool, read-only): a flag expressing whether the file exists
    - is_folder (type = bool, read-only): a flag expressing whether the file is folder
    - contents (type = container): a container storing the content of the file
- **Comments:** a variable with the `file` type can handle any kind of file (text, image or shape files...). The type of the `content` attribute will depend on the kind of file. Note that the allowed kinds of file are the followings:
  - text files: files with the extensions .txt, .data, .csv, .text, .tsv, .asc. The `content` is by default a list of string.

  - image files: files with the extensions .pgm, .tif, .tiff, .jpg, .jpeg, .png, .gif, .pict, .bmp. The `content` is by default a matrix of int.
  - shapefiles: files with the extension .shp. The `content` is by default a list of geometry.
  - properties files: files with the extension .properties. The `content` is by default a map of string::string.
  - folders. The `content` is by default a list of string.

- **Remark:** Files are also a particular kind of container and can thus be read, written or iterated using the container operators and commands.
- **See also:** File operators
- **Declaration:** a file can be created using the generic **`file`** (that opens a file in read only mode and tries to determine its contents), **`folder`** or the **`new_folder`** (to open an existing folder or create a new one) unary operators. But things can be specialized with the combination of the **`read`**/`write` and `image`/**`text`**/`shapefile`/`properties` unary operators.

```
folder(a_string)  // returns a file managing a existing folder
file(a_string) // returns any kind of file in read-only mode
read(text(a_string)) // returns a text file in read-only mode
read(image(a_string)) // does the same with an image file.
write(properties(a_string)) // returns a property file which is
   available for writing
                            // (if it exists, contents will be appended
    unless it is cleared
                            // using the standard container operations)
   .
```

Top of the page

geometry

- **Definition:** a datatype that represents a vector geometry, i.e. a list of georeferenced points.
- **Built-in attributes:**

  - location (type = point): the centroid of the geometry
  - area (type = float): the area of the geometry
  - perimeter (type = float): the perimeter of the geometry

  - holes (type = list of geometry): the list of the hole inside the given geometry
  - contour (type = geometry): the exterior ring of the given geometry and of his holes
  - envelope (type = geometry): the geometry bounding box
  - width (type = float): the width of the bounding box
  - height (type = float): the height of the bounding box
  - points (type = list of point): the set of the points composing the geometry

- **Comments:** a geometry can be either a point, a polyline or a polygon. Operators working on geometries handle transparently these three kinds of geometry. The envelope (a.k.a. the bounding box) of the geometry depends on the kind of geometry:

  - If this Geometry is the empty geometry, it is an empty point.
  - If the Geometry is a point, it is a non-empty point.
  - Otherwise, it is a Polygon whose points are (minx, miny), (maxx, miny), (maxx, maxy), (minx, maxy), (minx, miny).

- **See also:** Spatial operators
- **Declaration:** geometries can be built from a point, a list of points or by using specific operators (circle, square, triangle...).

```
geometry varGeom <- circle(5);
geometry polygonGeom <- polygon([{3,5}, {5,6},{1,4}]);
```

Top of the page

graph

- **Definition:** a datatype that represents a graph composed of vertices linked by edges.
- **Built-in attributes:**

  - edges(type = list of agent/geometry): the list of all edges
  - vertices(type = list of agent/geometry): the list of all vertices
  - circuit (type = path): an approximate minimal traveling salesman tour (hamiltonian cycle)

  - spanning_tree (type = list of agent/geometry): minimum spanning tree of the graph, i.e. a sub-graph such as every vertex lies in the tree, and as much edges lies in it but no cycles (or loops) are formed.
  - connected(type = bool): test whether the graph is connected

- **Remark:**
  - graphs are also a particular kind of container and can thus be manipulated using the container operators and commands.
  - This algorithm used to compute the circuit requires that the graph be complete and the triangle inequality exists (if x,y,z are vertices then d(x,y)+d(y,z)<d(x,z) for all x,y,z) then this algorithm will guarantee a hamiltonian cycle such that the total weight of the cycle is less than or equal to double the total weight of the optimal hamiltonian cycle.
  - The computation of the spanning tree uses an implementation of the Kruskal's minimum spanning tree algorithm. If the given graph is connected it computes the minimum spanning tree, otherwise it computes the minimum spanning forest.

- **See also:** Graph operators
- **Declaration:** graphs can be built from a list of vertices (agents or geometries) or from a list of edges (agents or geometries) by using specific operators. They are often used to deal with a road network and are built from a shapefile.

```
create road from: shape_file_road;
graph the_graph <- as_edge_graph(road);

graph([1,9,5])          --: ([1: in[] + out[], 5: in[] + out[], 9: in[] +
    out[]], [])
graph([node(0), node(1), node(2)]       // if node is a species
graph(['a'::345, 'b'::13])  --:  ([b: in[] + out[b::13], a: in[] + out[
   a::345], 13: in[b::13] + out[], 345: in[a::345] + out[]], [a::345=(a
   ,345), b::13=(b,13)])
graph(a_graph)  --: a_graph
graph(node1)    --: null
```

Top of the page

list

- **Definition:** a composite datatype holding an ordered collection of values.

- **Comments:** lists are more or less equivalent to instances of ArrayList in Java (although they are backed up by a specific class). They grow and shrink as needed, can be accessed via an index (see @ or index_of), support set operations (like union and difference), and provide the modeller with a number of utilities that make it easy to deal with collections of agents (see, for instance, shuffle, reverse,where,sort_by,...).
- **Remark:** lists can contain values of any datatypes, including other lists. Note, however, that due to limitations in the current parser, lists of lists cannot be declared litteraly; they have to be built using assignments. Lists are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a set of expressions separated by commas, enclosed in square brackets, like [12, 14, 'abc', self]. An empty list is noted `[]`.
- **Other declarations:** lists can be 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).
- **Comments:** Matrices are fixed-size structures that can be accessed by index (point for two-dimensions matrices, integer for vectors).
- **Litteral declaration:** Matrices cannot be defined literally. One-dimensions matrices can be built by using the matrix casting operator applied on a list. Two-dimensions matrices need to to be declared as variables first, before being filled.

```
//builds a one-dimension matrix, of size 5
matrix mat1 <- matrix ([10, 20, 30, 40, 50]);
//  builds a two-dimensions matrix with 10 columns and 5 rows, where
   each cell is initialized to 0.0
matrix mat2 <- 0.0 as_matrix({10,5});
// builds a two-dimensions matrix with 2 columns and 3 rows, with
   initialized cells
matrix mat3 <- matrix([["c11","c12","c13"],["c21","c22","c23"]]);
    -> c11;c21
       c12;c22
       c13;c23
```

Top of the page

pair

- **Definition:** a datatype holding a key and its associated value.
- **Built-in attributes:**
  - key (type = string): the key of the pair, i.e. the first element of the pair
  - value (type = string): the value of the pair, i.e. the second element of the pair
- **Remark:** pairs are also a particular kind of container and can thus be manipulated using the container operators and commands.
- **Litteral declaration:** a pair is defined by a key and a value separated by ::.
- **Other declarations:** a pair can also be built from:
  - a point,
  - a map (in this case the first element of the pair is the list of all the keys of the map and the second element is the list of all the values of the map),
  - a list (in this case the two first element of the list are used to build the pair)

```
pair testPair <- "key"::56;
pair testPairPoint <- {3,5};              // 3::5
pair testPairList2 <- [6,7,8];            // 6::7
pair testPairMap <- [2::6,5::8,12::45];   // [12,5,2]::[45,8,6]
```

Top of the page

path

- **Definition:** a datatype representing a path linking two agents or geometries in a graph.
- **Built-in attributes:**
  - source (type = point): the source point, i.e. the first point of the path
  - target (type = point): the target point, i.e. the last point of the path
  - graph (type = graph): the current topology (in the case it is a spatial graph), null otherwise
  - edges (type = list of agents/geometries): the edges of the graph composing the path

  - vertices (type = list of agents/geometries): the vertices of the graph composing the path
  - segments (type = list of geometries): the list of the geometries composing the path
  - shape (type = geometry) : the global geometry of the path (polyline)

- **Comments:** the path created between two agents/geometries or locations will strongly 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 topolopy
                               // of the current agent (i.e. a line in
    a& continuous topology,
                               // a path in a graph  in a graph
    topology )

a_topology path_between a_container_of_geometries // idem with an
    explicit topology and the possibility
                                                  // to have more than
    2 geometries
                                                  // (the path is then
    built incrementally)


path_between (a_graph, a_source, a_target) // idem with a the given
    graph as topology
```

Top of the page

point

- **Definition:** a datatype normally holding two positive float values. Represents the absolute coordinates of agents in the model.

- **Built-in attributes:**
  - x (type = float): coordinate of the point on the x-axis
  - y (type = float): coordinate of the point on the y-axis
- **Comments:** point coordinates should be positive, if a negative value is used in its declaration, the point is built with the absolute value.
- **Remark:** points are particular cases of geometries and containers. Thus they have also all the built-in attributes of both the geometry and the container datatypes and can be used with every kind of operator or command admitting geometry and container.
- **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 is which lives the population of agents

- Comments: this datatype is actually a “meta-type”. It allows to manipulate (in a rather limited fashion, however) the species themselves as any other values.
- Litteral declaration: the name of a declared species is already a 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};
    action fill {
```

```
            volume <- 1.0;
        }
}
```

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;
    }
}
```

Chapter 95

File Types

GAMA provides modelers with a generic type for files called **file**. It is possible to load a file using the *file* operator:

```
file my_file <- file("../includes/data.csv");
```

However, internally, GAMA makes the difference between the different types of files. Indeed, for instance:

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

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
0.2
Iris-setosa
```

```
4.9
3.0
1.4
0.2
Iris-setosa
...
```

Indeed, the content of CSV file is a matrix: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default ",").

In contrary:

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

will give:

```
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
Polygon
```

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile.

In order to know how to load a file, GAMA analyzes its extension. For instance for a file with a ".csv" extension, GAMA knows that the file is a **csv** one and will try to split each line with the , separator. However, if the modeler wants to split each line with a different separator (for instance **;**) or load it as a text file, he/she will have to use a specific file operator.

Indeed, GAMA integrates specific operators corresponding to different types of files.

Table of contents

Text File

Extensions

Here the list of possible extensions for text file: * “txt” * “data” * “csv” * “text” * “tsv” * “xml”

Note that when trying to define the type of a file with the default file loading operator (**file**), GAMA will first try to test the other type of file. For example, for files with “.csv” extension, GAMA will cast them as csv file and not as text file.

Content

The content of a text file is a list of string corresponding to each line of the text file. For example:

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

will give:

```
sepallength,sepalwidth,petallength,petalwidth,type
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
```

Operators

List of operators related to text files: * **text_file(string path)**: load a file (with an authorized extension) as a text file. * **text_file(string path, list content)**: load a file (with an authorized extension) as a text file and fill it with the given content. * **is_text(op)**: tests whether the operand is a text file

CSV File

Extensions

Here the list of possible extensions for csv file: * “csv” * “tsv”

Content

The content of a csv file is a matrix of string: each row of the matrix is a line of the file; each column of the matrix is value delimited by the separator (by default “,”). For example:

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

will give:

```
sepallength
sepalwidth
petallength
petalwidth
type
5.1
3.5
1.4
0.2
Iris-setosa
4.9
3.0
1.4
0.2
Iris-setosa
...
```

Operators

List of operators related to csv files: * **csv_file(string path)**: load a file (with an authorized extension) as a csv file with default separator (","). * **csv_file(string path, string separator)**: load a file (with an authorized extension) as a csv file with the given separator.

```
file my_file <- csv_file("../includes/data.csv", ";");
```

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

Shapefile

Shapefiles are classical GIS data files. A shapefile is not simple file, but a set of several files (source: wikipedia): * Mandatory files : * .shp — shape format; the feature geometry itself * .shx — shape index format; a positional index of the feature geometry to allow seeking forwards and backwards quickly * .dbf — attribute format; columnar attributes for each shape, in dBase IV format

- Optional files :
  - .prj — projection format; the coordinate system and projection information, a plain text file describing the projection using well-known text format
  - .sbn and .sbx — a spatial index of the features
  - .fbn and .fbx — a spatial index of the features for shapefiles that are read-only
  - .ain and .aih — an attribute index of the active fields in a table
  - .ixs — a geocoding index for read-write shapefiles
  - .mxs — a geocoding index for read-write shapefiles (ODB format)
  - .atx — an attribute index for the .dbf file in the form of shapefile.columnname.atx (ArcGIS 8 and later)
  - .shp.xml — geospatial metadata in XML format, such as ISO 19115 or other XML schema
  - .cpg — used to specify the code page (only for .dbf) for identifying the character encoding to be used

More details about shapefiles can be found here.

Extensions

Here the list of possible extension for shapefile: * “shp”

Content

The content of a shapefile is a list of geometries corresponding to the objects of the shapefile. For example:

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

will give:

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

Note that the attributes of each object of the shapefile is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of a corresponding attributes.

For example:

```
file my_file <- shape_file("../includes/data.shp");
write "my_file: " + my_file.contents;
loop el over: my_file {
    write (el get "TYPE");
}
```

Operators

List of operators related to shapefiles: * **shape_file(string path)**: load a file (with an authorized extension) as a shapefile with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). * **shape_file(string path, string code)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/) * **shape_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a shapefile with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/)

```
file my_file <- shape_file("../includes/data.shp", "EPSG:32601");
```

- **shape_file(string path, list content)**: load a file (with an authorized extension) as a shapefile and fill it with the given content.
  - **is_shape(op)**: tests whether the operand is a shapefile

OSM File

OSM (Open Street Map) is a collaborative project to create a free editable map of the world. The data produced in this project (OSM File) represent physical features on the ground (e.g., roads or buildings) using tags attached to its basic data structures (its nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way or relation (source: openstreetmap.org).

More details about OSM data can be found here.

Extensions

Here the list of possible extension for shapefile: * “osm” * “pbf” * “bz2” * “gz”

Content

The content of a OSM data is a list of geometries corresponding to the objects of the OSM file. For example:

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

will give:

```
Point
Point
Point
Point
Point
LineString
LineString
Polygon
Polygon
Polygon
...
```

Note that like for shapefiles, the attributes of each object of the osm file is stored in their corresponding GAMA geometry. The operator “get” (or “read”) allows to get the value of a corresponding attributes.

Operators

List of operators related to osm file: * **osm_file(string path)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, string code)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry. * **osm_file(string path, int EPSG_ID)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, all the nodes and ways of the OSM file will becomes a geometry.

```
file my_file <- osm_file("../includes/data.gz", "EPSG:32601");
```

- **osm_file(string path, map filter)**: load a file (with an authorized extension) as a osm file with default projection (if a prj file is defined, use it, otherwise use the default projection defined in the preference). In this case, only the elements with the defined values are loaded from the file.

```
//map used to filter the object to build from the OSM file according to
    attributes.
map filtering <- map(["highway"::["primary", "secondary", "tertiary", "
    motorway", "living_street","residential", "unclassified"], "building
    "::["yes"]]);

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

- **osm_file(string path, map filter, string code)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will

automatically decode the code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, only the elements with the defined values are loaded from the file.

- **osm_file(string path, map filter, int EPSG_ID)**: load a file (with an authorized extension) as a osm file with the given projection (GAMA will automatically decode the epsg code. For a list of the possible projections see: http://spatialreference.org/ref/). In this case, only the elements with the defined values are loaded from the file.
- **is_osm(op)**: tests whether the operand is a osm file

Grid File

Esri ASCII Grid files are classic text raster GIS data.

More details about Esri ASCII grid file can be found here.

Note that grid files can be used to initialize a grid species. The number of rows and columns will be read from the file. Similarly, the values of each cell contained in the grid file will be accessible through the **grid_value** attribute.

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

Extensions

Here the list of possible extension for grid file: * “asc”

Content

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

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

```
    }
}
```

will give:

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

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

For example:

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

Operators

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

```
file my_file <- grid_file("../includes/data.shp", "EPSG:32601");
```

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

Image File

Extensions

Here the list of possible extensions for image file: * “tif” * “tiff” * “jpg” * “jpeg” * “png” * “gif” * “pict” * “bmp”

Content

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

For example:

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

will give:

```
-9671572
-9671572
-9671572
-9671572
-9934744
-9934744
-9868951
-9868951
-10000537
-10000537
...
```

Operators

List of operators related to csv files: * **image_file(string path)**: load a file (with an authorized extension) as an image file. * **image_file(string path, matrix**

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

SVG File

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. Note that interactivity and animation features are not supported in GAMA.

More details about SVG file can be found here.

Extensions

Here the list of possible extension for SVG file: * “svg”

Content

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

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

will give:

```
Polygon
```

Operators

List of operators related to svg files: * **shape_file(string path)**: load a file (with an authorized extension) as a SVG file. * **shape_file(string path, point size)**: load a file (with an authorized extension) as a SVG file with the given size:

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

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

Property File

Extensions

Here the list of possible extensions for property file: * “properties”

Content

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

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

with the given property file:

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

will give:

```
3.0
4.0
5.0
Iris-setosa
2.5
```

Operators

List of operators related to text files: * **property_file(string path)**: load a file (with an authorized extension) as a property file. * **is_property(op)**: tests whether the operand is a property file

R File

R is a free software environment for statistical computing and graphics. GAMA allows to execute R script (if R is installed on the computer).

More details about R can be found here.

Note that GAMA also integrates some operators to manage R scripts: * R_compute * R_compute_param

Extensions

Here the list of possible extensions for R file: * “r”

Content

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

For example:

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

will give:

```
3.0
```

Operators

List of operators related to R files: * **R_file(string path)**: load a file (with an authorized extension) as a R file. * **is_R(op)**: tests whether the operand is a R file.

3DS File

3DS is one of the file formats used by the Autodesk 3ds Max 3D modeling, animation and rendering software. 3DS files can be used in GAMA to load 3D geometries.

More details about 3DS file can be found here.

Extensions

Here the list of possible extension for 3DS file: * “3ds” * “max”

Content

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

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

will give:

```
Polygon
```

Operators

List of operators related to 3ds files: * **threeds_file(string path)**: load a file (with an authorized extension) as a 3ds file. * **is_threeds(op)**: tests whether the operand is a 3DS file

OBJ File

OBJ file is a geometry definition file format first developed by Wavefront Technologies for its Advanced Visualizer animation package. The file format is open and has been adopted by other 3D graphics application vendors.

More details about Obj file can be found here.

Extensions

Here the list of possible extension for OBJ files: * “obj”

Content

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

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

will give:

```
Polygon
```

Operators

List of operators related to obj files: * **obj_file(string path)**: load a file (with an authorized extension) as a obj file. * **is_obj(op)**: tests whether the operand is a OBJ file

Chapter 96

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.

Chapter 97

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

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.

Chapter 98

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 ...
```

Table of Contents

Constants

- **#AdamsBashforth**, value= AdamsBashforth, Comment: AdamsBashforth solver
- **#AdamsMoulton**, value= AdamsMoulton, Comment: AdamsMoulton solver
- **#current_error**, value= , Comment: The text of the last error thrown during the current execution
- **#DormandPrince54**, value= DormandPrince54, Comment: DormandPrince54 solver
- **#dp853**, value= dp853, Comment: dp853 solver
- **#e**, value= 2.718281828459045, Comment: The e constant

- **#Euler**, value= Euler, Comment: Euler solver
- **#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= 2.147483647E9, 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= -2.147483648E9, Comment: A constant holding the minimum value an int can have (Java Integer.MIN_VALUE)
- **#nan**, value= NaN, Comment: A constant holding a Not-a-Number (NaN) value of type float (Java Double.POSITIVE_INFINITY)
- **#pi**, value= 3.141592653589793, Comment: The PI constant
- **#rk4**, value= rk4, Comment: rk4 solver
- **#ThreeEighthes**, value= ThreeEighthes, Comment: ThreeEighthes 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

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
- **#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= 0.025400000000000002, 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= 9.999999999999999E-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** (#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: defines an exact duration of 30 days. WARNING: this duration is of course not correct in terms of calendar
- **#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: defines an exact duration of 365 days. WARNING: this duration is of course not correct in terms of calendar

---

Volume units

- **#cl** (#centiliter,#centiliters), value= 1.0E-5, Comment: centiliter unit
- **#dl** (#deciliter,#deciliters), value= 1.0E-4, Comment: deciliter unit
- **#hl** (#hectoliter,#hectoliters), value= 0.1, Comment: hectoliter unit
- **#l** (#liter,#liters,#dm3), value= 0.001, Comment: liter unit
- **#m3**, value= 1.0, Comment: cube meter: the basic unit for volumes

---

Weight units

- **#gram** (#grams), value= 0.001, Comment: gram unit
- **#kg** (#kilo,#kilogram,#kilos), value= 1.0, Comment: second: the basic unit for weights
- **#longton** (#lton), value= 1016.0469088000001, Comment: short ton unit
- **#ounce** (#oz,#ounces), value= 0.028349523125, Comment: ounce unit
- **#pound** (#lb,#pounds,#lbm), value= 0.45359237, Comment: pound unit
- **#shortton** (#ston), value= 907.18474, Comment: short ton unit
- **#stone** (#st), value= 6.35029318, Comment: stone unit
- **#ton** (#tons), value= 1000.0, Comment: ton unit

Colors

In addition to the previous units, GAML provides a direct access to the 147 named colors defined in CSS (see http://www.cssportal.com/css3-color-names/). E.g,

```
rgb my_color <- °teal;
```

- **#aliceblue**, value= r=240, g=248, b=255, alpha=1
- **#antiquewhite**, value= r=250, g=235, b=215, alpha=1
- **#aqua**, value= r=0, g=255, b=255, alpha=1
- **#aquamarine**, value= r=127, g=255, b=212, alpha=1
- **#azure**, value= r=240, g=255, b=255, alpha=1
- **#beige**, value= r=245, g=245, b=220, alpha=1

- **#bisque**, value= r=255, g=228, b=196, alpha=1
- **#black**, value= r=0, g=0, b=0, alpha=1
- **#blanchedalmond**, value= r=255, g=235, b=205, alpha=1
- **#blue**, value= r=0, g=0, b=255, alpha=1
- **#blueviolet**, value= r=138, g=43, b=226, alpha=1
- **#brown**, value= r=165, g=42, b=42, alpha=1
- **#burlywood**, value= r=222, g=184, b=135, alpha=1
- **#cadetblue**, value= r=95, g=158, b=160, alpha=1
- **#chartreuse**, value= r=127, g=255, b=0, alpha=1
- **#chocolate**, value= r=210, g=105, b=30, alpha=1
- **#coral**, value= r=255, g=127, b=80, alpha=1
- **#cornflowerblue**, value= r=100, g=149, b=237, alpha=1
- **#cornsilk**, value= r=255, g=248, b=220, alpha=1
- **#crimson**, value= r=220, g=20, b=60, alpha=1
- **#cyan**, value= r=0, g=255, b=255, alpha=1
- **#darkblue**, value= r=0, g=0, b=139, alpha=1
- **#darkcyan**, value= r=0, g=139, b=139, alpha=1
- **#darkgoldenrod**, value= r=184, g=134, b=11, alpha=1
- **#darkgray**, value= r=169, g=169, b=169, alpha=1
- **#darkgreen**, value= r=0, g=100, b=0, alpha=1
- **#darkgrey**, value= r=169, g=169, b=169, alpha=1
- **#darkkhaki**, value= r=189, g=183, b=107, alpha=1
- **#darkmagenta**, value= r=139, g=0, b=139, alpha=1

- **#darkolivegreen**, value= r=85, g=107, b=47, alpha=1
- **#darkorange**, value= r=255, g=140, b=0, alpha=1
- **#darkorchid**, value= r=153, g=50, b=204, alpha=1
- **#darkred**, value= r=139, g=0, b=0, alpha=1
- **#darksalmon**, value= r=233, g=150, b=122, alpha=1
- **#darkseagreen**, value= r=143, g=188, b=143, alpha=1
- **#darkslateblue**, value= r=72, g=61, b=139, alpha=1
- **#darkslategray**, value= r=47, g=79, b=79, alpha=1
- **#darkslategrey**, value= r=47, g=79, b=79, alpha=1
- **#darkturquoise**, value= r=0, g=206, b=209, alpha=1
- **#darkviolet**, value= r=148, g=0, b=211, alpha=1
- **#deeppink**, value= r=255, g=20, b=147, alpha=1
- **#deepskyblue**, value= r=0, g=191, b=255, alpha=1
- **#dimgray**, value= r=105, g=105, b=105, alpha=1
- **#dimgrey**, value= r=105, g=105, b=105, alpha=1
- **#dodgerblue**, value= r=30, g=144, b=255, alpha=1
- **#firebrick**, value= r=178, g=34, b=34, alpha=1
- **#floralwhite**, value= r=255, g=250, b=240, alpha=1
- **#forestgreen**, value= r=34, g=139, b=34, alpha=1
- **#fuchsia**, value= r=255, g=0, b=255, alpha=1
- **#gainsboro**, value= r=220, g=220, b=220, alpha=1
- **#ghostwhite**, value= r=248, g=248, b=255, alpha=1
- **#gold**, value= r=255, g=215, b=0, alpha=1

- **#goldenrod**, value= r=218, g=165, b=32, alpha=1
- **#gray**, value= r=128, g=128, b=128, alpha=1
- **#green**, value= r=0, g=128, b=0, alpha=1
- **#greenyellow**, value= r=173, g=255, b=47, alpha=1
- **#grey**, value= r=128, g=128, b=128, alpha=1
- **#honeydew**, value= r=240, g=255, b=240, alpha=1
- **#hotpink**, value= r=255, g=105, b=180, alpha=1
- **#indianred**, value= r=205, g=92, b=92, alpha=1
- **#indigo**, value= r=75, g=0, b=130, alpha=1
- **#ivory**, value= r=255, g=255, b=240, alpha=1
- **#khaki**, value= r=240, g=230, b=140, alpha=1
- **#lavender**, value= r=230, g=230, b=250, alpha=1
- **#lavenderblush**, value= r=255, g=240, b=245, alpha=1
- **#lawngreen**, value= r=124, g=252, b=0, alpha=1
- **#lemonchiffon**, value= r=255, g=250, b=205, alpha=1
- **#lightblue**, value= r=173, g=216, b=230, alpha=1
- **#lightcoral**, value= r=240, g=128, b=128, alpha=1
- **#lightcyan**, value= r=224, g=255, b=255, alpha=1
- **#lightgoldenrodyellow**, value= r=250, g=250, b=210, alpha=1
- **#lightgray**, value= r=211, g=211, b=211, alpha=1
- **#lightgreen**, value= r=144, g=238, b=144, alpha=1
- **#lightgrey**, value= r=211, g=211, b=211, alpha=1
- **#lightpink**, value= r=255, g=182, b=193, alpha=1

- **#lightsalmon**, value= r=255, g=160, b=122, alpha=1
- **#lightseagreen**, value= r=32, g=178, b=170, alpha=1
- **#lightskyblue**, value= r=135, g=206, b=250, alpha=1
- **#lightslategray**, value= r=119, g=136, b=153, alpha=1
- **#lightslategrey**, value= r=119, g=136, b=153, alpha=1
- **#lightsteelblue**, value= r=176, g=196, b=222, alpha=1
- **#lightyellow**, value= r=255, g=255, b=224, alpha=1
- **#lime**, value= r=0, g=255, b=0, alpha=1
- **#limegreen**, value= r=50, g=205, b=50, alpha=1
- **#linen**, value= r=250, g=240, b=230, alpha=1
- **#magenta**, value= r=255, g=0, b=255, alpha=1
- **#maroon**, value= r=128, g=0, b=0, alpha=1
- **#mediumaquamarine**, value= r=102, g=205, b=170, alpha=1
- **#mediumblue**, value= r=0, g=0, b=205, alpha=1
- **#mediumorchid**, value= r=186, g=85, b=211, alpha=1
- **#mediumpurple**, value= r=147, g=112, b=219, alpha=1
- **#mediumseagreen**, value= r=60, g=179, b=113, alpha=1
- **#mediumslateblue**, value= r=123, g=104, b=238, alpha=1
- **#mediumspringgreen**, value= r=0, g=250, b=154, alpha=1
- **#mediumturquoise**, value= r=72, g=209, b=204, alpha=1
- **#mediumvioletred**, value= r=199, g=21, b=133, alpha=1
- **#midnightblue**, value= r=25, g=25, b=112, alpha=1
- **#mintcream**, value= r=245, g=255, b=250, alpha=1

- **#mistyrose**, value= r=255, g=228, b=225, alpha=1
- **#moccasin**, value= r=255, g=228, b=181, alpha=1
- **#navajowhite**, value= r=255, g=222, b=173, alpha=1
- **#navy**, value= r=0, g=0, b=128, alpha=1
- **#oldlace**, value= r=253, g=245, b=230, alpha=1
- **#olive**, value= r=128, g=128, b=0, alpha=1
- **#olivedrab**, value= r=107, g=142, b=35, alpha=1
- **#orange**, value= r=255, g=165, b=0, alpha=1
- **#orangered**, value= r=255, g=69, b=0, alpha=1
- **#orchid**, value= r=218, g=112, b=214, alpha=1
- **#palegoldenrod**, value= r=238, g=232, b=170, alpha=1
- **#palegreen**, value= r=152, g=251, b=152, alpha=1
- **#paleturquoise**, value= r=175, g=238, b=238, alpha=1
- **#palevioletred**, value= r=219, g=112, b=147, alpha=1
- **#papayawhip**, value= r=255, g=239, b=213, alpha=1
- **#peachpuff**, value= r=255, g=218, b=185, alpha=1
- **#peru**, value= r=205, g=133, b=63, alpha=1
- **#pink**, value= r=255, g=192, b=203, alpha=1
- **#plum**, value= r=221, g=160, b=221, alpha=1
- **#powderblue**, value= r=176, g=224, b=230, alpha=1
- **#purple**, value= r=128, g=0, b=128, alpha=1
- **#red**, value= r=255, g=0, b=0, alpha=1
- **#rosybrown**, value= r=188, g=143, b=143, alpha=1

- **#royalblue**, value= r=65, g=105, b=225, alpha=1
- **#saddlebrown**, value= r=139, g=69, b=19, alpha=1
- **#salmon**, value= r=250, g=128, b=114, alpha=1
- **#sandybrown**, value= r=244, g=164, b=96, alpha=1
- **#seagreen**, value= r=46, g=139, b=87, alpha=1
- **#seashell**, value= r=255, g=245, b=238, alpha=1
- **#sienna**, value= r=160, g=82, b=45, alpha=1
- **#silver**, value= r=192, g=192, b=192, alpha=1
- **#skyblue**, value= r=135, g=206, b=235, alpha=1
- **#slateblue**, value= r=106, g=90, b=205, alpha=1
- **#slategray**, value= r=112, g=128, b=144, alpha=1
- **#slategrey**, value= r=112, g=128, b=144, alpha=1
- **#snow**, value= r=255, g=250, b=250, alpha=1
- **#springgreen**, value= r=0, g=255, b=127, alpha=1
- **#steelblue**, value= r=70, g=130, b=180, alpha=1
- **#tan**, value= r=210, g=180, b=140, alpha=1
- **#teal**, value= r=0, g=128, b=128, alpha=1
- **#thistle**, value= r=216, g=191, b=216, alpha=1
- **#tomato**, value= r=255, g=99, b=71, alpha=1
- **#transparent**, value= r=0, g=0, b=0, alpha=0
- **#turquoise**, value= r=64, g=224, b=208, alpha=1
- **#violet**, value= r=238, g=130, b=238, alpha=1
- **#wheat**, value= r=245, g=222, b=179, alpha=1

- **#white**, value= r=255, g=255, b=255, alpha=1
- **#whitesmoke**, value= r=245, g=245, b=245, alpha=1
- **#yellow**, value= r=255, g=255, b=0, alpha=1
- **#yellowgreen**, value= r=154, g=205, b=50, alpha=1

Chapter 99

Pseudo-variables

The expressions known as **pseudo-variables** are special read-only variables that are not declared anywhere (at least not in a species), and which represent a value that changes depending on the context of execution.

Table of contents

self

The pseudo-variable `self` always holds a reference to the agent executing the current statement.

- Example (sets the `friend` attribute of another random agent of the same species to `self` and conversely):

```
friend potential_friend <- one_of (species(self) - self);
if potential_friend != nil {
    potential_friend.friend <- self;
    friend <- potential_friend;
}
```

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
```

Chapter 100

Variables and Attributes

Variables and attributes represent named data that can be used in an expression. They can be accessed depending on their *scope*:

- the scope of attributes declared in a species is itself, its child species and its micro-species.
- the scope of temporary variables is the one in which they have been declared, and all its sub-scopes. Outside its *scope* of validity, an expression cannot use a variable or an attribute directly. However, attributes can be used in a remote fashion by using a dotted notation on a given agent (see here).

Table of contents

Direct Access

When an agent wants to use either one of the variables declared locally, one of the attributes declared in its species (or parent species), one of the attributes declared in the macro-species of its species, it can directly invoke its name and the compiler will

do the rest (i.e. finding the variable or attribute in the right scope). For instance, we can have a look at the following example:

```
species animal {
   float energy <- 1000 min: 0 max: 2000 update: energy - 0.001;
   int age_in_years <- 1 update: age_in_years + int (time / 365);

   action eat (float amount <- 0) {
       float gain <- amount / age_in_years;
       energy <- energy + gain;
   }

   reflex feed {
      int food_found <- rnd(100);
      do eat (amount: food_found);
   }

}
```

Species declaration

Everywhere in the species declaration, we are able to directly name and use: * `time`, a global built-in variable, * `energy` and `age_in_years`, the two species attributes.

Nevertheless, in the species declaration, but outside of the action `eat` and the reflex `feed`, we **cannot** name the variables:

- `amount`, the argument of `eat` action,
- `gain`, a local variable defined into the `eat` action,
- `food_found`, the local variable defined into the `feed` reflex.

`Eat` action declaration

In the `eat` action declaration, we can directly name and use:

- `time`, a global built-in variable,
- `energy` and `age_in_years`, the two species attributes,
- `amount`, which is an argument to the action `eat`,
- `gain`, a temporary variable within the action.

We **cannot** name and use the variables:

- `food_found`, the local variable defined into the `feed` reflex.

`feed` reflex declaration

Similarly, in the `feed` reflex declaration, we can directly name and use: * `time`, a global built-in variable, * `energy` and `age_in_years`, the two species variables, * `food_found`, the local variable defined into the reflex.

But we **cannot** access to variables:

- `amount`, the argument of `eat` action,
- `gain`, a local variable defined into the `eat` action.

Remote Access

When an expression needs to get access to the attribute of an agent which does not belong to its scope of execution, a special notation (similar to that used in Java) has to be used:

```
remote_agent.variable
```

where remote_agent can be the name of an agent, an expression returning an agent, self, myself or each. For instance, if we modify the previous species by giving its agents the possibility to feed another agent found in its 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;
}
```

Chapter 101

Operators

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

Definition

Operators in the GAML language are used to compose complex expressions. An operator performs a function on one, two, or n operands (which are other expressions and thus may be themselves composed of operators) and returns the result of this function.

Most of them use a classical prefixed functional syntax (i.e. `operator_name(operand1, operand2, operand3)`, see below), with the exception of arithmetic (e.g. `+`, `/`), logical (**`and`**, **`or`**), comparison (e.g. `>`, `<`), access (`.`, `[..]`) and pair (`::`) operators, which require an infixed notation (i.e. `operand1 operator_symbol operand1`).

The ternary functional if-else operator, `? :`, uses a special infixed syntax composed with two symbols (e.g. `operand1 ? operand2 : operand3`). Two unary operators (`-` and `!`) use a traditional prefixed syntax that does not require parentheses unless the operand is itself a complex expression (e.g. `- 10`, `! (operand1 or operand2)`).

Finally, special constructor operators (`{...}` for constructing points, `[...]` for constructing lists and maps) will require their operands to be placed between their two

symbols (e.g. `{1,2,3}`, `[operand1, operand2, ..., operandn]` or `[key1::value1, key2::value2... keyn::valuen]`).

With the exception of these special cases above, the following rules apply to the syntax of operators:

- if they only have one operand, the functional prefixed syntax is mandatory (e.g. `operator_name(operand1)`)
- if they have two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2)`) or the infixed syntax (e.g. `operand1 operator_name operand2`) can be used.
- if they have more than two arguments, either the functional prefixed syntax (e.g. `operator_name(operand1, operand2, ..., operand)`) or a special infixed syntax with the first operand on the left-hand side of the operator name (e.g. `operand1 operator_name(operand2, ..., operand)`) can be used.

All of these alternative syntaxes are completely equivalent.

Operators in GAML are purely functional, i.e. they are guaranteed to not have any side effects on their operands. For instance, the **`shuffle`** operator, which randomizes the positions of elements in a list, does not modify its list operand but returns a new shuffled list.

Priority between operators

The priority of operators determines, in the case of complex expressions composed of several operators, which one(s) will be evaluated first.

GAML follows in general the traditional priorities attributed to arithmetic, boolean, comparison operators, with some twists. Namely:

- the constructor operators, like `::`, used to compose pairs of operands, have the lowest priority of all operators (e.g. `a > b :: b > c` will return a pair of boolean values, which means that the two comparisons are evaluated before the operator applies. Similarly, `[a > 10, b > 5]` will return a list of boolean values.

- it is followed by the `?:` operator, the functional if-else (e.g. `a > b ? a + 10 : a - 10` will return the result of the if-else).
- next are the logical operators, **`and`** and **`or`** (e.g. `a > b or b > c` will return the value of the test)
- next are the comparison operators (i.e. `>`, `<`, `<=`, `>=`, `=`, `!=`)
- next the arithmetic operators in their logical order (multiplicative operators have a higher priority than additive operators)
- next the unary operators - and `!`
- next the access operators `.` and `[]` (e.g. `{1,2,3}.x > 20 + {4,5,6}.y` will return the result of the comparison between the x and y ordinates of the two points)
- and finally the functional operators, which have the highest priority of all.

Using actions as operators

Actions defined in species can be used as operators, provided they are called on the correct agent. The syntax is that of normal functional operators, but the agent that will perform the action must be added as the first operand.

For instance, if the following species is defined:

```
species spec1 {
        int min(int x, int y) {
                return x > y ? x : y;
        }
}
```

Any agent instance of spec1 can use **`min`** as an operator (if the action conflicts with an existing operator, a warning will be emitted). For instance, in the same model, the following line is perfectly acceptable:

```
global {
        init {
                create spec1;
                spec1 my_agent <- spec1[0];
                int the_min <- my_agent min(10,20); // or min(my_agent,
    10, 20);
        }
}
```

If the action doesn't have any operands, the syntax to use is `my_agent the_action()`. Finally, if it does not return a value, it might still be used but is considering as returning a value of type **unknown** (e.g. **unknown** `result <- my_agent the_action( op1, op2);`).

Note that due to the fact that actions are written by modelers, the general functional contract is not respected in that case: actions might perfectly have side effects on their operands (including the agent).

Table of Contents

Operators by categories

3D

box, cone3D, cube, cylinder, dem, hexagon, pyramid, set_z, sphere, teapot,

---

Arithmetic operators

-, /, [](OperatorsAA#), *, +, 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, font, is, is_skill, list_with, matrix_with, species, to_gaml, topology,

---

Color-related operators

-, /, *, +, blend, brewer_colors, brewer_palettes, grayscale, hsb, mean, median, rgb, rnd_color, sum,

---

Comparison operators

!=, <, <=, =, >, >=, between,

Containers-related operators

-, ::, +, accumulate, all_match, among, at, collect, contains, contains_all, contains_any, contains_key, count, distinct, empty, every, first, first_with, get, group_by, in, index_by, inter, interleave, internal_at, internal_integrated_value, last, last_with, length, max, max_of, mean, mean_of, median, min, min_of, mul, none_matches, one_matches, one_of, product_of, range, 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,

Driving operators

as_driving_graph,

---

edge

edge_between, strahler,

EDP-related operators

diff, diff2,

Files-related operators

crs, csv_file, dxf_file, evaluate_sub_model, file, file_exists, folder, gaml_file, geojson_file, get, gif_file, gml_file, grid_file, image_file, is_csv, is_dxf, is_gaml, is_geojson, is_gif, is_gml, is_grid, is_image, is_json, is_obj, is_osm, is_pgm, is_property, is_R, 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, R_file, read, saved_simulation_file, shape_file, step_sub_model, svg_file, text_file, threeds_file, writable, xml_file,

---

FIPA-related operators

conversation, message,

GamaMetaType

type_of,

Graphs-related operators

add_edge, add_node, adjacency, agent_from_geometry, all_pairs_shortest_path, alpha_index, as_distance_graph, as_edge_graph, as_intersection_graph, as_path, beta_index, betweenness_centrality, biggest_cliques_of, connected_components_of, connectivity_index, contains_edge, contains_vertex, degree_of, directed, edge, edge_between, edge_betweenness, edges, gamma_index, generate_barabasi_albert, generate_complete_graph, generate_watts_strogatz, grid_cells_to_graph, in_degree_of, in_edges_of, layout_circle, layout_force, layout_grid, load_graph_from_file, 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_optimizer_type, with_weights,

---

Grid-related operators

as_4_grid, as_grid, as_hexagonal_grid, grid_at, path_between,

---

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, with_max_of, with_min_of,

---

List-related operators

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,

Material

material,

Matrix-related operators

-, /, ., *, +, append_horizontally, append_vertically, column_at, columns_list, determinant, eigenvalues, index_of, inverse, last_index_of, row_at, rows_list, shuffle, trace, transpose,

---

multicriteria operators

electre_DM, evidence_theory_DM, 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,

---

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, gauss, open_simplex_generator, poisson, rnd, rnd_choice, sample, shuffle, simplex_generator, skew_gauss, truncated_gauss,

---

ReverseOperators

restore_simulation, restore_simulation_from_file, save_agent, save_simulation, serialize, serialize_agent,

---

Shape

arc, box, circle, cone, cone3D, cross, cube, curve, cylinder, ellipse, envelope, geometry_collection, hexagon, line, link, plan, polygon, polyhedron, pyramid, rectangle, sphere, square, squircle, teapot, triangle,

---

Spatial operators

-, *, +, add_point, agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, angle_between, any_location_in, arc, around, as_4_grid, as_grid, as_hexagonal_grid, at_distance, at_location, box, centroid, circle, clean, clean_network, closest_points_with, closest_to, cone, cone3D, convex_hull, covers, cross, crosses, crs, CRS_transform, cube, curve, cylinder, dem, direction_between, disjoint_from, distance_between, distance_to, ellipse, envelope, farthest_point_to, farthest_to, geometry_collection, gini, hexagon, hierarchical_clustering, IDW, inside, inter, intersects, inverse_rotation, line, link, masked_by, moran, neighbors_at, neighbors_of, normalized_rotation, overlapping, overlaps, 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, towards, transformed_by, translated_by, triangle, triangulate, union, using, voronoi, with_precision, without_holes,

---

Spatial properties operators

covers, crosses, intersects, partially_overlaps, touches,

Spatial queries operators

agent_closest_to, agent_farthest_to, agents_at_distance, agents_inside, agents_overlapping, at_distance, closest_to, farthest_to, inside, neighbors_at, neighbors_of, overlapping,

---

Spatial relations operators

direction_between, distance_between, distance_to, path_between, path_to, towards,

---

Spatial statistical operators

hierarchical_clustering, simple_clustering_by_distance,

Spatial transformations operators

-, *, +, as_4_grid, as_grid, as_hexagonal_grid, at_location, clean, 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, corR, correlation, covariance, dbscan, distribution_of, distribution2d_of, dtw, durbin_watson, frequency_of, gamma, gamma_distribution, gamma_distribution_complemented, gamma_rnd, geometric_mean, gini, harmonic_mean, hierarchical_clustering, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, kmeans, kurtosis, kurtosis, log_gamma, max, mean, mean_deviation, meanR, median, min, moment, moran, mul, normal_area, normal_density, normal_inverse, predict, pValue_for_fStat, pValue_for_tStat, quantile, quantile_inverse, rank_interpolated, rms, simple_clustering_by_distance, skew, skewness, split, split_in, split_using, standard_deviation, student_area, student_t_inverse, sum, variance, variance,

---

Strings-related operators

+, <, <=, >, >=, at, 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, replace, replace_regex, reverse, sample, shuffle, split_with, string, upper_case,

---

SubModel

load_sub_model,

System

., command, copy, copy_to_clipboard, dead, eval_gaml, every, is_error, is_warning, user_input,

---

Time-related operators

date, string,

Types-related operators

action, agent, attributes, BDIPlan, bool, container, emotion, file, float, gaml_type, geometry, graph, int, kml, list, map, material, matrix, mental_state, Norm, pair, path, point, predicate, regression, rgb, Sanction, skill, social_link, topology, unknown,

---

User control operators

user_input,

Operators

-

Possible uses:

- **-** (**point**) —> **point**
- **-** (**float**) —> **float**
- **-** (**int**) —> **int**
- **float - matrix** —> **matrix**
- **-** (**float** , **matrix**) —> **matrix**
- **rgb - int** —> **rgb**

- - (rgb , int) —> rgb
- rgb - rgb —> rgb
- - (rgb , rgb) —> rgb
- geometry - float —> geometry
- - (geometry , float) —> geometry
- list - unknown —> list
- - (list , unknown) —> list
- geometry - container<unknown,geometry> —> geometry
- - (geometry , container<unknown,geometry>) —> geometry
- point - int —> point
- - (point , int) —> point
- float - float —> float
- - (float , float) —> float
- int - matrix —> matrix
- - (int , matrix) —> matrix
- map - map —> map
- - (map , map) —> map
- date - float —> date
- - (date , float) —> date
- int - float —> float
- - (int , float) —> float
- matrix<unknown> - matrix —> matrix
- - (matrix<unknown> , matrix) —> matrix
- date - date —> float
- - (date , date) —> float
- point - point —> point
- - (point , point) —> point
- float - int —> float
- - (float , int) —> float
- date - int —> date
- - (date , int) —> date
- matrix<unknown> - int —> matrix
- - (matrix<unknown> , int) —> matrix
- species - agent —> list
- - (species , agent) —> list
- geometry - geometry —> geometry
- - (geometry , geometry) —> geometry
- container - container —> list

- **-** (**container** , **container**) —> **list**
- **map** **-** **pair** —> **map**
- **-** (**map** , **pair**) —> **map**
- **int** **-** **int** —> **int**
- **-** (**int** , **int**) —> **int**
- **point** **-** **float** —> **point**
- **-** (**point** , **float**) —> **point**
- **matrix**<**unknown**> **-** **float** —> **matrix**
- **-** (**matrix**<**unknown**> , **float**) —> **matrix**

Result:

If it is used as an unary operator, it returns the opposite of the operand. Returns the difference of the two operands.

Comment:

The behavior of the operator depends on the type of the operands.

Special cases:

- if 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 and the right operand is empty, - returns the left operand

- 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 var18 <- rgb([255, 128, 32]) - 3; // var18 equals rgb([252,125,29])
```

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

```
rgb var19 <- rgb([255, 128, 32]) - rgb('red'); // var19 equals rgb
    ([0,128,32])
```

- 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 var20 <- shape - 5; // var20 equals a geometry corresponding
    to the geometry of the agent applying the operator reduced by a
    distance of 5
```

- if the left operand is a list and the right operand is an object of any type (except list), - returns a list containing the elements of the left operand minus all the occurrences of this object

```
list<int> var21 <- [1,2,3,4,5,6] - 2; // var21 equals [1,3,4,5,6]
list<int> var22 <- [1,2,3,4,5,6] - 0; // var22 equals [1,2,3,4,5,6]
```

- 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 var23 <- rectangle(10,10) - [circle(2), square(2)]; // var23
    equals rectangle(10,10) - (circle(2) + square(2))
```

- 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 var24 <- 3.5 - matrix([[2,5],[3,4]]); // var24 equals matrix
    ([[1.5,-1.5],[0.5,-0.5]])
```

- 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 var25 <- date('2000-01-02') - date('2000-01-01'); // var25 equals
    86400
```

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

```
point var26 <- {1, 2} - {4, 5}; // var26 equals {-3.0, -3.0}
```

- 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 var27 <- date('2000-01-01') - 86400; // var27 equals date
   ('1999-12-31')
```

- if both operands are a point, a geometry or an agent, returns the geometry resulting from the difference between both geometries

```
geometry var28 <- geom1 - geom2; // var28 equals a geometry
   corresponding to difference between geom1 and geom2
```

- 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> var29 <- [1,2,3,4,5,6] - [2,4,9]; // var29 equals [1,3,5,6]
list<int> var30 <- [1,2,3,4,5,6] - [0,8]; // var30 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 var31 <- 1 - 1; // var31 equals 0
```

- 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 var32 <- {1, 2} - 4.5; // var32 equals {-3.5, -2.5, -4.5}
point var33 <- {1, 2} - 4; // var33 equals {-3.0,-2.0,-4.0}
```

Examples:

```
matrix var0 <- (10.0 - (3.0 as_matrix({2,3}))); // var0 equals matrix
    ([[7.0,7.0,7.0],[7.0,7.0,7.0]])
point var1 <- -{3.0,5.0}; // var1 equals {-3.0,-5.0}
point var2 <- -{1.0,6.0,7.0}; // var2 equals {-1.0,-6.0,-7.0}
point var3 <- {2.0,3.0,4.0} - 1; // var3 equals {1.0,2.0,3.0}
float var4 <- 1.0 - 1.0; // var4 equals 0.0
float var5 <- 3.7 - 1.2; // var5 equals 2.5
float var6 <- 3.0 - 1.2; // var6 equals 1.8
map var7 <- ['a'::1,'b'::2] - ['b'::2]; // var7 equals ['a'::1]
map var8 <- ['a'::1,'b'::2] - ['b'::2,'c'::3]; // var8 equals ['a'::1]
date var9 <- date('2000-01-01') - 86400; // var9 equals date
    ('1999-12-31')
float var10 <- 1 - 1.0; // var10 equals 0.0
float var11 <- 3 - 1.2; // var11 equals 1.8
float var12 <- 1.0 - 1; // var12 equals 0.0
float var13 <- 3.7 - 1; // var13 equals 2.7
float var14 <- 3.0 - 1; // var14 equals 2.0
int var15 <- - (-56); // var15 equals 56
map var16 <- ['a'::1,'b'::2] - ('b'::2); // var16 equals ['a'::1]
map var17 <- ['a'::1,'b'::2] - ('c'::3); // var17 equals ['a'::1,'b
    '::2]
```

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:

- float != int —> bool
- != (float , int) —> bool
- date != date —> bool
- != (date , date) —> bool
- float != float —> bool
- != (float , float) —> bool
- unknown != unknown —> bool

- **!=** (**unknown** , **unknown**) —> **bool**
- **int != float** —> **bool**
- **!=** (**int** , **float**) —> **bool**

Result:

true if both operands are different, false otherwise

Examples:

```
bool var0 <- 3.0 != 3; // var0 equals false
bool var1 <- 4.7 != 4; // var1 equals true
bool var2 <- #now != #now minus_hours 1; // var2 equals true
bool var3 <- 3.0 != 3.0; // var3 equals false
bool var4 <- 4.0 != 4.7; // var4 equals true
bool var5 <- [2,3] != [2,3]; // var5 equals false
bool var6 <- [2,4] != [2,3]; // var6 equals true
bool var7 <- 3 != 3.0; // var7 equals false
bool var8 <- 4 != 4.7; // 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 / float** —> **float**
- **/ (int , float)** —> **float**
- **rgb / float** —> **rgb**
- **/ (rgb , float)** —> **rgb**
- **float / float** —> **float**
- **/ (float , float)** —> **float**
- **matrix<unknown> / int** —> **matrix**
- **/ (matrix<unknown> , int)** —> **matrix**
- **int / int** —> **float**
- **/ (int , int)** —> **float**
- **point / int** —> **point**
- **/ (point , int)** —> **point**
- **point / float** —> **point**
- **/ (point , float)** —> **point**
- **float / int** —> **float**

- **/** (**float** , **int**) —> **float**
- **matrix**<**unknown**> **/ matrix** —> **matrix**
- **/** (**matrix**<**unknown**> , **matrix**) —> **matrix**
- **matrix**<**unknown**> **/ float** —> **matrix**
- **/** (**matrix**<**unknown**> , **float**) —> **matrix**
- **rgb / int** —> **rgb**
- **/** (**rgb** , **int**) —> **rgb**

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 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 var0 <- rgb([255, 128, 32]) / 2.5; // var0 equals rgb([102,51,13])
```

- if both operands are numbers (float or int), performs a normal arithmetic division and returns a float.

```
float var1 <- 3 / 5.0; // var1 equals 0.6
```

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

```
point var2 <- {5, 7.5} / 2.5; // var2 equals {2, 3}
point var3 <- {2,5} / 4; // var3 equals {0.5,1.25}
```

- if one operand is a color and the other an integer, returns a new color resulting from the division of each component of the color by the right operand

```
rgb var4 <- rgb([255, 128, 32]) / 2; // var4 equals rgb([127,64,16])
```

See also:

*, +, -,

.

Possible uses:

- **agent** . **any expression** —> **unknown**
- . (**agent** , **any expression**) —> **unknown**
- **matrix**<**unknown**> . **matrix** —> **matrix**
- . (**matrix**<**unknown**> , **matrix**) —> **matrix**

Result:

It has two different uses: it can be the dot product between 2 matrices or return an evaluation of the expression (right-hand operand) in the scope the given agent.

Special cases:

- if the agent is nil or dead, throws an exception
- if the left operand is an agent, it evaluates of the expression (right-hand operand) in the scope the given agent

```
unknown var0 <- agent1.location; // var0 equals the location of the
    agent agent1
```

- if both operands are matrix, returns the dot product of them

```
matrix var1 <- matrix([[1,1],[1,2]]) . matrix([[1,1],[1,2]]); // var1
    equals matrix([[2,3],[3,5]])
```

---

^

Possible uses:

- **int ^ float** —> **float**
- **^** (**int** , **float**) —> **float**
- **float ^ int** —> **float**
- **^** (**float** , **int**) —> **float**
- **float ^ float** —> **float**
- **^** (**float** , **float**) —> **float**
- **int ^ int** —> **float**
- **^** (**int** , **int**) —> **float**

Result:

Returns the value (always a float) of the left operand raised to the power of the right operand.

Special cases:

- if the right-hand operand is equal to 0, returns 1
- if it is equal to 1, returns the left-hand operand.
- Various examples of power

```
float var1 <- 2 ^ 3; // var1 equals 8.0
```

Examples:

```
float var0 <- 4.84 ^ 0.5; // var0 equals 2.2
```

See also:

*, sqrt,

@

Same signification as at

---

*

Possible uses:

- matrix<unknown> * int —> matrix
- * (matrix<unknown> , int) —> matrix
- int * matrix —> matrix
- * (int , matrix) —> matrix
- int * float —> float
- * (int , float) —> float
- matrix<unknown> * float —> matrix
- * (matrix<unknown> , float) —> matrix
- point * float —> point
- * (point , float) —> point
- float * float —> float
- * (float , float) —> float
- float * int —> float
- * (float , int) —> float
- int * int —> int
- * (int , int) —> int
- float * matrix —> matrix
- * (float , matrix) —> matrix
- rgb * int —> rgb
- * (rgb , int) —> rgb
- geometry * point —> geometry
- * (geometry , point) —> geometry
- geometry * float —> geometry
- * (geometry , float) —> geometry
- point * point —> float
- * (point , point) —> float
- point * int —> point

- * (point , int) —> point
- matrix<unknown> * matrix —> matrix
- * (matrix<unknown> , matrix) —> matrix

Result:

Returns the product of the two operands.

Special cases:

- if one operand is a matrix and the other a number (float or int), performs a normal arithmetic product of the number with each element of the matrix (results are float if the number is a float.

```
matrix var1 <- 2 * matrix([[2,5],[3,4]]); // var1 equals matrix
    ([[4,10],[6,8]])
```

- 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 var2 <- 1 * 1; // var2 equals 1
```

- 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 var3 <- rgb([255, 128, 32]) * 2; // var3 equals rgb([255,255,64])
```

- 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 var4 <- shape * {0.5,0.5,2}; // var4 equals a geometry
    corresponding to the geometry of the agent applying the operator
    scaled by a coefficient of 0.5 in x, 0.5 in y and 2 in z
```

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

```
geometry var5 <- circle(10) * 2; // var5 equals circle(20)
geometry var6 <- (circle(10) * 2).location with_precision 9; // var6
   equals (circle(20)).location with_precision 9
float var7 <- (circle(10) * 2).height with_precision 9; // var7 equals
   (circle(20)).height with_precision 9
```

- if both operands are points, returns their scalar product

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

- if the left-hand operator is a point and the right-hand a number, returns a point with coordinates multiplied by the number

```
point var9 <- {2,5} * 4; // var9 equals {8.0, 20.0}
point var10 <- {2, 4} * 2.5; // var10 equals {5.0, 10.0}
```

Examples:

```
float var0 <- 2.5 * 2; // var0 equals 5.0
```

See also:

/, +, -,

+

Possible uses:

- map + map —> map
- + (map , map) —> map
- matrix<unknown> + int —> matrix
- + (matrix<unknown> , int) —> matrix
- matrix<unknown> + float —> matrix
- + (matrix<unknown> , float) —> matrix
- point + point —> point
- + (point , point) —> point
- container + container —> container
- + (container , container) —> container
- string + string —> string
- + (string , string) —> string
- int + float —> float
- + (int , float) —> float
- geometry + float —> geometry
- + (geometry , float) —> geometry
- container + unknown —> list
- + (container , unknown) —> list
- float + float —> float
- + (float , float) —> float
- point + float —> point
- + (point , float) —> point
- map + pair —> map
- + (map , pair) —> map
- geometry + geometry —> geometry
- + (geometry , geometry) —> geometry
- point + int —> point
- + (point , int) —> point
- string + unknown —> string
- + (string , unknown) —> string
- rgb + rgb —> rgb
- + (rgb , rgb) —> rgb
- float + matrix —> matrix
- + (float , matrix) —> matrix

- **date + float —> date**
- **+ (date , float) —> date**
- **matrix<unknown> + matrix —> matrix**
- **+ (matrix<unknown> , matrix) —> matrix**
- **int + int —> int**
- **+ (int , int) —> int**
- **rgb + int —> rgb**
- **+ (rgb , int) —> rgb**
- **date + int —> date**
- **+ (date , int) —> date**
- **int + matrix —> matrix**
- **+ (int , matrix) —> matrix**
- **float + int —> float**
- **+ (float , int) —> float**
- **date + string —> string**
- **+ (date , string) —> string**
- **+ (geometry, float, int) —> geometry**
- **+ (geometry, float, int, int) —> geometry**

Result:

Returns the sum, union or concatenation of the two operands.

Special cases:

- if one of the operands is nil, + throws an error
- if both operands are species, returns a special type of list called meta-population
- if both operands are points, returns their sum.

```
point var0 <- {1, 2} + {4, 5}; // var0 equals {5.0, 7.0}
```

- if the left-hand operand is a geometry and the right-hand operands a float, an integer and one of #round, #square or #flat, returns a geometry corresponding

to the left-hand operand (geometry, agent, point) enlarged by the first right-hand operand (distance), using a number of segments equal to the second right-hand operand and a flat, square or round end cap style

```
geometry var1 <- circle(5) + (5,32,#round); // var1 equals circle(10)
```

- if both operands are list, +returns the concatenation of both lists.

```
list<int> var2 <- [1,2,3,4,5,6] + [2,4,9]; // var2 equals
    [1,2,3,4,5,6,2,4,9]
list<int> var3 <- [1,2,3,4,5,6] + [0,8]; // var3 equals
    [1,2,3,4,5,6,0,8]
```

- if the left-hand and right-hand operand are a string, returns the concatenation of the two operands

```
string var4 <- "hello " + "World"; // var4 equals "hello World"
```

- 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 var5 <- circle(5) + 5; // var5 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> var6 <- [1,2,3,4,5,6] + 2; // var6 equals [1,2,3,4,5,6,2]
list<int> var7 <- [1,2,3,4,5,6] + 0; // var7 equals [1,2,3,4,5,6,0]
```

- if 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 var8 <- {1, 2} + 4.5; // var8 equals {5.5, 6.5,4.5}
```

- if the right-operand is a point, a geometry or an agent, returns the geometry resulting from the union between both geometries

```
geometry var9 <- geom1 + geom2; // var9 equals a geometry corresponding
    to union between geom1 and geom2
```

- if the left-hand operand is a string, returns the concatenation of the two operands (the left-hand one beind casted into a string)

```
string var10 <- "hello " + 12; // var10 equals "hello 12"
```

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

```
rgb var11 <- rgb([255, 128, 32]) + rgb('red'); // var11 equals rgb
   ([255,128,32])
```

- 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 var12 <- circle(5) + (5,32); // var12 equals circle(10)
```

- 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 var13 <- 1 + 1; // var13 equals 2
```

- 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 var14 <- rgb([255, 128, 32]) + 3; // var14 equals rgb([255,131,35])
```

- 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 var15 <- date('2000-01-01') + 86400; // var15 equals date
    ('2000-01-02')
```

- 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 var16 <- 3.5 + matrix([[2,5],[3,4]]); // var16 equals matrix
    ([[5.5,8.5],[6.5,7.5]])
```

Examples:

```
map var17 <- ['a'::1,'b'::2] + ['c'::3]; // var17 equals ['a'::1,'b
    '::2,'c'::3]
map var18 <- ['a'::1,'b'::2] + [5::3.0]; // var18 equals ['a'::1,'b
    '::2,5::3.0]
map var19 <- ['a'::1,'b'::2] + ('c'::3); // var19 equals ['a'::1,'b
    '::2,'c'::3]
map var20 <- ['a'::1,'b'::2] + ('c'::3); // var20 equals ['a'::1,'b
    '::2,'c'::3]
point var21 <- {1, 2} + 4; // var21 equals {5.0, 6.0,4.0}
date var22 <- date('2016-01-01 00:00:01') + 86400; // var22 equals date
    ('2016-01-02 00:00:01')
float var23 <- 1.0 + 1; // var23 equals 2.0
float var24 <- 1.0 + 2.5; // var24 equals 3.5
string var25 <- date('2000-01-01 00:00:00') + '_Test'; // var25 equals
    '2000-01-01 00:00:00_Test'
```

See also:

-, /, *,

<

Possible uses:

- **string** < **string** —> **bool**
- < (**string** , **string**) —> **bool**
- **float** < **int** —> **bool**
- < (**float** , **int**) —> **bool**
- **date** < **date** —> **bool**
- < (**date** , **date**) —> **bool**
- **int** < **int** —> **bool**
- < (**int** , **int**) —> **bool**
- **point** < **point** —> **bool**
- < (**point** , **point**) —> **bool**
- **int** < **float** —> **bool**
- < (**int** , **float**) —> **bool**
- **float** < **float** —> **bool**
- < (**float** , **float**) —> **bool**

Result:

true if the left-hand operand is less than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' < 'aeb'; // var0 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var1 <- {5,7} < {4,6}; // var1 equals false
bool var2 <- {5,7} < {4,8}; // var2 equals false
```

Examples:

```
bool var3 <- 3.5 < 7; // var3 equals true
bool var4 <- #now < #now minus_hours 1; // var4 equals false
bool var5 <- 3 < 7; // var5 equals true
bool var6 <- 3 < 2.5; // var6 equals false
bool var7 <- 3.5 < 7.6; // var7 equals true
```

See also:

>, >=, <=, =, !=,

<=

Possible uses:

- **int <= int** —> **bool**
- **<=** (**int** , **int**) —> **bool**
- **string <= string** —> **bool**
- **<=** (**string** , **string**) —> **bool**
- **float <= float** —> **bool**
- **<=** (**float** , **float**) —> **bool**
- **date <= date** —> **bool**
- **<=** (**date** , **date**) —> **bool**
- **float <= int** —> **bool**
- **<=** (**float** , **int**) —> **bool**
- **point <= point** —> **bool**
- **<=** (**point** , **point**) —> **bool**
- **int <= float** —> **bool**
- **<=** (**int** , **float**) —> **bool**

Result:

true if the left-hand operand is less or equal than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' <= 'aeb'; // var0 equals true
```

- if both operands are points, returns true if and only if the left component (x) of the left operand if less than or equal to x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var1 <- {5,7} <= {4,6}; // var1 equals false
bool var2 <- {5,7} <= {4,8}; // var2 equals false
```

Examples:

```
bool var3 <- 3 <= 7; // var3 equals true
bool var4 <- 3.5 <= 3.5; // var4 equals true
bool var5 <- (#now <= (#now minus_hours 1)); // var5 equals false
bool var6 <- 7.0 <= 7; // var6 equals true
bool var7 <- 3 <= 2.5; // var7 equals false
```

See also:

>, <, >=, =, !=,

=

Possible uses:

- **int = int —> bool**
- **= (int , int) —> bool**

- **float** = **float** —> **bool**
- = (**float** , **float**) —> **bool**
- **date** = **date** —> **bool**
- = (**date** , **date**) —> **bool**
- **float** = **int** —> **bool**
- = (**float** , **int**) —> **bool**
- **unknown** = **unknown** —> **bool**
- = (**unknown** , **unknown**) —> **bool**
- **int** = **float** —> **bool**
- = (**int** , **float**) —> **bool**

Result:

returns true if both operands are equal, false otherwise returns true if both operands are equal, false otherwise

Special cases:

- if both operands are any kind of objects, returns true if they are identical (i.e., the same object) or equal (comparisons between nil values are permitted)

```
bool var0 <- [2,3] = [2,3]; // var0 equals true
```

Examples:

```
bool var1 <- 4 = 5; // var1 equals false
bool var2 <- 4.5 = 4.7; // var2 equals false
bool var3 <- #now = #now minus_hours 1; // var3 equals false
bool var4 <- 4.7 = 4; // var4 equals false
bool var5 <- 3 = 3.0; // var5 equals true
bool var6 <- 4 = 4.7; // var6 equals false
```

See also:

!=, >, <, >=, <=,

>

Possible uses:

- **int** > **float** —> **bool**
- > (**int** , **float**) —> **bool**
- **string** > **string** —> **bool**
- > (**string** , **string**) —> **bool**
- **float** > **float** —> **bool**
- > (**float** , **float**) —> **bool**
- **date** > **date** —> **bool**
- > (**date** , **date**) —> **bool**
- **int** > **int** —> **bool**
- > (**int** , **int**) —> **bool**
- **point** > **point** —> **bool**
- > (**point** , **point**) —> **bool**
- **float** > **int** —> **bool**
- > (**float** , **int**) —> **bool**

Result:

true if the left-hand operand is greater than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are String, uses a lexicographic comparison of two strings

```
bool var0 <- 'abc' > 'aeb'; // var0 equals false
```

- if both operands are points, returns true if and only if the left component (x) of the left operand if greater than x of the right one and if the right component (y) of the left operand is greater than y of the right one.

```
bool var1 <- {5,7} > {4,6}; // var1 equals true
bool var2 <- {5,7} > {4,8}; // var2 equals false
```

Examples:

```
bool var3 <- 3 > 2.5; // var3 equals true
bool var4 <- 3.5 > 7.6; // var4 equals false
bool var5 <- (#now > (#now minus_hours 1)); // var5 equals true
bool var6 <- 13.0 > 7.0; // var6 equals true
bool var7 <- 3.5 > 7; // var7 equals false
```

See also:

<, >=, <=, =, !=,

>=

Possible uses:

- **point >= point** —> **bool**
- **>=** (**point** , **point**) —> **bool**
- **string >= string** —> **bool**
- **>=** (**string** , **string**) —> **bool**
- **int >= int** —> **bool**
- **>=** (**int** , **int**) —> **bool**
- **int >= float** —> **bool**
- **>=** (**int** , **float**) —> **bool**
- **date >= date** —> **bool**
- **>=** (**date** , **date**) —> **bool**
- **float >= float** —> **bool**
- **>=** (**float** , **float**) —> **bool**
- **float >= int** —> **bool**
- **>=** (**float** , **int**) —> **bool**

Result:

true if the left-hand operand is greater or equal than the right-hand operand, false otherwise.

Special cases:

- if one of the operands is nil, returns false
- if both operands are points, returns true if and only if the left component (x) of the left operand if greater or equal than x of the right one and if the right component (y) of the left operand is greater than or equal to y of the right one.

```
bool var0 <- {5,7} >= {4,6}; // var0 equals true
bool var1 <- {5,7} >= {4,8}; // var1 equals false
```

- if both operands are string, uses a lexicographic comparison of the two strings

```
bool var2 <- 'abc' >= 'aeb'; // var2 equals false
bool var3 <- 'abc' >= 'abc'; // var3 equals true
```

Examples:

```
bool var4 <- 3 >= 7; // var4 equals false
bool var5 <- 3 >= 2.5; // var5 equals true
bool var6 <- #now >= #now minus_hours 1; // var6 equals true
bool var7 <- 3.5 >= 3.5; // var7 equals true
bool var8 <- 3.5 >= 7; // var8 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]
list<int> var2 <- [1,2,4] accumulate (each * 2); // var2 equals [2,4,8]
```

See also:

collect,

---

acos

Possible uses:

- **acos** (**float**) —> **float**
- **acos** (**int**) —> **float**

Result:

Returns the value (in the interval [0,180], in decimal degrees) of the arccos of the operand (which should be in [-1,1]).

Special cases:

- if the right-hand operand is outside of the [-1,1] interval, returns NaN

Examples:

```
float var0 <- acos (0); // var0 equals 90.0
```

See also:

asin, atan, cos,

---

action

Possible uses:

- **action** (**any**) —> action

---

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: take the following argument: (kml, location (point),orientation (float), scale (float), file_path (string)) the kml export manager with new 3D model: take the following argument: (kml, location (point),orientation (float), scale (float), file_path (string), begin date, end date)

See also:

add_geometry, add_icon, add_label,

---

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:

the kml export manager with new geometry: take the following argument: (kml, geometry,linewidth, linecolor,fillcolor) the kml export manager with new geometry: take the following argument: (kml, geometry,linewidth, linecolor,fillcolor, begin date, end date) the kml export manager with new geometry: take the following argument: (kml, geometry,linewidth, color) the kml export manager with new geometry: take the following argument: (kml, geometry,linewidth, linecolor,fillcolor, end date) the kml export manager with new geometry: take the following argument: (kml, geometry, linecolor,fillcolor)

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:

the kml export manager with new icons: take the following argument: (kml, location (point),orientation (float), scale (float), file_path (string)) the kml export manager with new icons: take the following argument: (kml, location (point),orientation (float), scale (float), file_path (string), begin date, end date)

See also:

add_geometry, add_icon,

---

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_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

---

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,

---

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_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,

---

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: alpha = nb_cycles / (2*S-5) - planar graph)

Examples:

```
float var1 <- alpha_index(graphEpidemio); // var1 equals the alpha
    index of the graph
```

See also:

beta_index, gamma_index, nb_cycles, connectivity_index,

---

among

Possible 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
```

---

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**<**unknown**> **append_horizontally** **matrix** —> **matrix**
- **append_horizontally** (**matrix**<**unknown**> , **matrix**) —> **matrix**
- **matrix**<**unknown**> **append_horizontally** **matrix** —> **matrix**
- **append_horizontally** (**matrix**<**unknown**> , **matrix**) —> **matrix**

Result:

A matrix resulting from the concatenation of the rows of the two given matrices. A matrix resulting from the concatenation of the rows of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

Examples:

```
matrix var0 <- matrix([[1.0,2.0],[3.0,4.0]]) append_horizontally matrix
    ([[1,2],[3,4]]); // var0 equals matrix
    ([[1.0,2.0],[3.0,4.0],[1.0,2.0],[3.0,4.0]])
matrix var1 <- matrix([[1.0,2.0],[3.0,4.0]]) append_horizontally matrix
    ([[1,2],[3,4]]); // var1 equals matrix
    ([[1.0,2.0],[3.0,4.0],[1.0,2.0],[3.0,4.0]])
```

---

append_vertically

Possible uses:

- **matrix**<**unknown**> **append_vertically** **matrix** —> **matrix**
- **append_vertically** (**matrix**<**unknown**> , **matrix**) —> **matrix**
- **matrix**<**unknown**> **append_vertically** **matrix** —> **matrix**
- **append_vertically** (**matrix**<**unknown**> , **matrix**) —> **matrix**

Result:

A matrix resulting from the concatenation of the columns of the two given matrices. A matrix resulting from the concatenation of the columns of the two given matrices. If not both numerical or both object matrices, returns the first matrix.

Examples:

```
matrix var0 <- matrix([[1,2],[3,4]]) append_vertically matrix
    ([[1,2],[3,4]]); // var0 equals matrix([[1,2,1,2],[3,4,3,4]])
matrix var1 <- matrix([[1,2],[3,4]]) append_vertically matrix
    ([[1,2],[3,4]]); // var1 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 An arc, which radius is equal to the first operand, heading to the second and amplitude the third

Comment:

the center of the arc is by default the location of the current agent in which has been called this operator.the center of the arc is by default the location of the current agent in which has been called this operator. This operator returns a polygon by default.

Special cases:

- returns a point if the radius operand is lower or equal to 0.
- returns a point if the radius operand is lower or equal to 0.

Examples:

```
geometry var0 <- arc(4,45,90, false); // var0 equals a geometry as an
    arc of radius 4, in a direction of 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 map —> graph
- as_distance_graph (container , map) —> graph
- container as_distance_graph float —> graph
- as_distance_graph (container , float) —> graph
- as_distance_graph (container, float, species) —> graph

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices close enough (less than a distance, right-hand operand).

Comment:

as_distance_graph is more efficient for a list of points than as_intersection_graph.

Examples:

```
list(ant) as_distance_graph 3.0
```

See also:

as_intersection_graph, as_edge_graph,

---

as_driving_graph

Possible 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

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 map, the graph will be built by creating edges from pairs of the map

```
graph var1 <- as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]); // var1
    equals a graph with these three vertices and two edges
```

- 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 var2 <- as_edge_graph([line([{1,5},{12,45}]),line
   ([{13,45},{34,56}])],1); // var2 equals a graph with two edges and
   three vertices
```

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 tesselation of the first operand geometry

Examples:

```
list<geometry> var0 <- self as_hexagonal_grid {10, 5}; // var0 equals
    list of geometries (hexagonal) corresponding to the hexagonal
    tesselation of the first operand geometry
```

See also:

as_4_grid, as_grid,

---

as_int

Possible 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

Result:

creates a graph from a list of vertices (left-hand operand). An edge is created between each pair of vertices with an intersection (with a given tolerance).

Comment:

as_intersection_graph is more efficient for a list of geometries (but less accurate) than as_distance_graph.

Examples:

```
list(ant) as_intersection_graph 0.5
```

See also:

as_distance_graph, as_edge_graph,

---

as_map

Possible 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 to short, some elements will be omitted. Matrix remaining elements will be filled in by nil.

Special cases:

- if the right operand is nil, as_matrix is equivalent to the matrix operator

See also:

matrix,

---

as_path

Possible 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
```

---

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:

- string at int —> string
- at (string , int) —> string
- container<KeyType,ValueType> at KeyType —> ValueType
- at (container<KeyType,ValueType> , KeyType) —> ValueType

Result:

the element at the right operand index of the container

Comment:

The first element of the container is located at the index 0. In addition, if the user tries to get the element at an index higher or equals than the length of the container, he will get an IndexOutOfBoundException.The at operator behavior depends on the nature of the operand

Special cases:

- if it is a file, at returns the element of the file content at the index specified by the right operand
- if it is a population, at returns the agent at the index specified by the right operand
- if it is a graph and if the right operand is a node, at returns the in and out edges corresponding to that node
- if it is a graph and if the right operand is an edge, at returns the pair node_out::node_in of the edge
- if it is a graph and if the right operand is a pair node1::node2, at returns the edge from node1 to node2 in the graph
- if it is a list or a matrix, at returns the element at the index specified by the right operand

```
int var1 <- [1, 2, 3] at 2; // var1 equals 3
point var2 <- [{1,2}, {3,4}, {5,6}] at 0; // var2 equals {1.0,2.0}
```

Examples:

```
string var0 <- 'abcdef' at 0; // var0 equals 'a'
```

See also:

contains_all, contains_any,

---

at_distance

Possible 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 <= the right operand from the caller agent (in its topology)

Examples:

```
list<geometry> var0 <- [ag1, ag2, ag3] at_distance 20; // var0 equals
    the agents of the list located at a distance <= 20 from the caller
    agent (in the same order).
```

See also:

neighbors_at, neighbors_of, agent_closest_to, agents_inside, closest_to, inside, overlapping,

---

at_location

Possible 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** (**float**) —> **float**
- **atan** (**int**) —> **float**

Result:

Returns the value (in the interval [-90,90], in decimal degrees) of the arctan of the operand (which can be any real number).

Examples:

```
float var0 <- atan (1); // var0 equals 45.0
```

See also:

acos, asin, tan,

---

atan2

Possible 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`

---

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
```

BDIPlan

Possible uses:

- BDIPlan (any) —> BDIPlan

Result:

---

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) : 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** (**int**, **int**, **int**) —> **bool**
- **between** (**date**, **date**, **date**) —> **bool**
- **between** (**any expression**, **date**, **date**) —> **bool**
- **between** (**float**, **float**, **float**) —> **bool**

Result:

returns true the first integer operand is bigger than the second integer operand and smaller than the third integer operand

returns true if the first float operand is bigger than the second float operand and smaller than the third float operand

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')))
```

- returns true if the first operand is between the two dates passed in arguments (both exclusive). The version with 2 arguments compares the current_date with the 2 others

```
bool var3 <- (date('2016-01-01') between(date('2000-01-01'), date
    ('2020-02-02'))); // var3 equals true
// // will return true if the current_date of the model is in_between
    the 2
between(date('2000-01-01'), date('2020-02-02'))
```

Examples:

```
bool var6 <- between(5, 1, 10); // var6 equals true
bool var7 <- between(5.0, 1.0, 10.0); // var7 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:

poisson, gauss,

---

binomial_coeff

Possible uses:

- **int binomial_coeff int** —> **float**
- **binomial_coeff** (**int** , **int**) —> **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**) —> **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**) —> **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 var1 <- blend(#red, #blue); // var1 equals to a color very close to
    the purple
```

Examples:

```
rgb var3 <- blend(#red, #blue, 0.3); // var3 equals to a color between
    the purple and the blue
```

See also:

rgb, hsb,

---

bool

Possible uses:

- **bool** (**any**) —> **bool**

Result:

---

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.
geometry var1 <- box({10, 5 , 5}); // var1 equals a geometry as a
    rectangle with width = 10, height = 5 depth= 5.
float var2 <-  (box({10, 10 , 5}) at_location point(50,50,0)).location.
    y; // var2 equals 50.0
```

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/). The list of palettes can be obtained by calling brewer_palettes Build a list of colors of a given type (see website http://colorbrewer2.org/) with a given number of classes

Examples:

```
list<rgb> var0 <- list<rgb> colors <- brewer_colors("OrRd");; // var0
    equals a list of 6 blue colors
list<rgb> var1 <- list<rgb> colors <- brewer_colors("Pastel1", 5);; //
    var1 equals a list of 5 sequential colors in the palette named '
    Pastel1'. The list of palettes can be obtained by calling
    brewer_palettes
```

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) returns the list a palette with a given min number of classes and max number of classes)

Examples:

```
list<string> var0 <- list<string> palettes <- brewer_palettes(3);; //
    var0 equals a list of palettes that are composed of a min of 3
    colors
list<string> var1 <- list<string> palettes <- brewer_palettes(5,10);;
    // var1 equals a list of palettes that are composed of a min of 5
    colors and a max of 10 colors
```

See also:

brewer_colors,

---

buffer

Same signification as +

---

build

Possible uses:

- **build** (**matrix**) —> **regression**

Result:

returns the regression build from the matrix data (a row = an instance, the last value of each line is the y value) while using the given 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]]))
```

---

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
float var1 <- ceil(3.5); // var1 equals 4.0
float var2 <- ceil(-4.7); // var2 equals -4.0
```

See also:

floor, round,

---

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
```

---

circle

Possible uses:

- circle (float) —> geometry
- float circle point —> geometry
- circle (float , point) —> geometry

Result:

A circle geometry which radius is equal to the first operand, and the center has the location equal to the second operand. 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 operand is lower or equal to 0.
- returns a point if the operand is lower or equal to 0.

Examples:

```
geometry var0 <- circle(10,{80,30}); // var0 equals a geometry as a
    circle of radius 10, the center will be in the location {80,30}.
geometry var1 <- circle(10); // var1 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 operandis 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 polulines 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.
```

---

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. 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. command allows GAMA to issue a system command using the system terminal or shell and to receive a string containing the outcome of the command or script executed. By default, commands are blocking the agent calling them, unless the sequence ' &' is used at the end. In this case, the result of the operator is an empty string

---

cone

Possible 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. A cone geometry which min and max angles are given by the operands.

Comment:

the center of the cone is by default the location of the current agent in which has been called this operator.the center of the cone is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- cone({0, 45}); // var0 equals a geometry as a cone
    with min angle is 0 and max angle is 45.
geometry var1 <- cone(0, 45); // var1 equals a geometry as a cone with
    min angle is 0 and max angle is 45.
```

See also:

around, circle, line, link, norm, point, polygon, polyline, rectangle, square, triangle,

---

cone3D

Possible 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 edges (if the boolean is true) or vertices (if the boolean is false) that are in the connected components. 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.

Examples:

```
graph my_graph2 <- graph([]);
list<list> var1 <- connected_components_of (my_graph2, true); // var1
    equals the list of all the components as list
graph my_graph <- graph([]);
list<list> var3 <- connected_components_of (my_graph); // 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**

---

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 it the operand is contained in the file content
- if it is a population, contains returns true if the operand is an agent of the population, false otherwise
- if it is a graph, contains 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:

- **container** contains_all **container** —> **bool**
- contains_all (**container** , **container**) —> **bool**
- **string** contains_all **list** —> **bool**
- contains_all (**string** , **list**) —> **bool**

Result:

true if the left operand contains all the elements of the right operand, false otherwise

Comment:

the definition of contains depends on the container

Special cases:

- if the right operand is nil or empty, contains_all returns true
- if the left-operand is a string, test whether the string contains all the element of the list;

```
bool var4 <- "abcabcabc" contains_all ["ca","xy"]; // var4 equals false
```

Examples:

```
bool var0 <- [1,2,3,4,5,6] contains_all [2,4]; // var0 equals true
bool var1 <- [1,2,3,4,5,6] contains_all [2,8]; // var1 equals false
bool var2 <- [1::2, 3::4, 5::6] contains_all [1,3]; // var2 equals
    false
bool var3 <- [1::2, 3::4, 5::6] contains_all [2,4]; // var3 equals true
```

See also:

contains, contains_any,

---

contains_any

Possible uses:

- **container contains_any container** —> **bool**
- **contains_any** (**container** , **container**) —> **bool**
- **string contains_any list** —> **bool**
- **contains_any** (**string** , **list**) —> **bool**

Result:

true if the left operand contains one of the elements of the right operand, false otherwise

Comment:

the definition of contains depends on the container

Special cases:

- if the right operand is nil or empty, contains_any returns false

Examples:

```
bool var0 <- [1,2,3,4,5,6] contains_any [2,4]; // var0 equals true
bool var1 <- [1,2,3,4,5,6] contains_any [2,8]; // var1 equals true
bool var2 <- [1::2, 3::4, 5::6] contains_any [1,3]; // var2 equals
    false
bool var3 <- [1::2, 3::4, 5::6] contains_any [2,4]; // var3 equals true
bool var4 <- "abcabcabc" contains_any ["ca","xy"]; // 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 (righ-hand operand), false otherwise

Special cases:

- if the left-hand operand is nil, returns false
- if the right-hand operand is a pair, returns true if it exists an edge between the two elements of the pair in the graph

```
bool 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. >= {0,0} and < {rows, 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 < 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 (righ-hand operand), false otherwise

Special cases:

- if the left-hand operand is nil, returns false

Examples:

```
graph graphFromMap<-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}]);
bool var1 <- graphFromMap contains_vertex {1,5}; // var1 equals true
```

See also:

contains_edge,

---

conversation

Possible uses:

- **conversation** (**unknown**) —> **conversation**

---

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_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');
```

---

corR

Possible uses:

- container corR container —> unknown
- corR (container , container) —> unknown

Result:

returns the Pearson correlation coefficient of two given vectors (right-hand operands) in given variable (left-hand operand).

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
list X <- [1, 2, 3];
list Y <- [1, 2, 4];
unknown var2 <- corR(X, Y); // var2 equals 0.981980506061966
```

---

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 (int) —> float
- cos (float) —> float

Result:

Returns the value (in [-1,1]) of the cosinus of the operand (in decimal degrees). The argument is casted to an int before being evaluated.

Special cases:

- Operand values out of the range [0-359] are normalized.

Examples:

```
float var0 <- cos (0); // var0 equals 1.0
float var1 <- cos(360); // var1 equals 1.0
float var2 <- cos(-720); // var2 equals 1.0
float var3 <- cos (0.0); // var3 equals 1.0
float var4 <- cos(360.0); // var4 equals 1.0
float var5 <- cos(-720.0); // 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
```

---

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 operandsfor 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']
map<int,float> var1 <- create_map([0,1],[0.1,0.2,0.3]); // var1 equals
    [0::0.1,1::0.2]
map<string,float> var2 <- create_map(['a','b','c','d'],[1.0,2.0,3.0]);
    // var2 equals ['a'::1.0,'b'::2.0,'c'::3.0]
```

---

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 the width of the lines for the second A cross, which radius is equal to the first operand

Examples:

```
geometry var0 <- cross(10,2); // var0 equals a geometry as a cross of
    radius 10, and with a width of 2 for the lines
geometry var1 <- cross(10); // var1 equals a geometry as a cross of
    radius 10
```

See also:

around, cone, line, link, norm, point, polygon, polyline, super_ellipse, rectangle, square, circle, ellipse, triangle,

---

crosses

Possible 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,

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

Special cases:

- 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 var0 <- CRS_transform(shape); // var0 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 var1 <- shape CRS_transform("EPSG:4326"); // var1 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): 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**, **float**) —> **geometry**
- **curve** (**point**, **point**, **point**) —> **geometry**
- **curve** (**point**, **point**, **point**, **point**) —> **geometry**
- **curve** (**point**, **point**, **point**, **int**) —> **geometry**
- **curve** (**point**, **point**, **float**, **float**) —> **geometry**
- **curve** (**point**, **point**, **float**, **bool**) —> **geometry**
- **curve** (**point**, **point**, **float**, **int**, **float**) —> **geometry**

- **curve** (**point**, **point**, **point**, **point**, **int**) —> **geometry**
- **curve** (**point**, **point**, **float**, **bool**, **int**) —> **geometry**
- **curve** (**point**, **point**, **float**, **bool**, **int**, **float**) —> **geometry**
- **curve** (**point**, **point**, **float**, **int**, **float**, **float**) —> **geometry**

Result:

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). A cubic Bezier curve geometry built from the four given points composed of 10 points. A cubic Bezier curve geometry built from the four given points composed of a given number of points. 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. 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). A quadratic Bezier curve geometry built from the three given points composed of a given numnber of points. A cubic Bezier curve geometry built from the 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). A cubic Bezier curve geometry built from the two given points with the given coefficient for the radius and composed of 10 points. 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). 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. A quadratic Bezier curve geometry built from the three given points composed of 10 points.

Special cases:

- if the operand is nil, returns nil
- if the operand is nil, returns nil

- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the last operand (number of points) is inferior to 2, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil
- if the operand is nil, returns nil

Examples:

```
geometry var0 <- curve({0,0},{10,10}, 0.5, 100, 90); // var0 equals a
    cubic Bezier curve geometry composed of 100 points from p0 to p1 at
    the right side.
geometry var1 <- curve({0,0}, {0,10}, {10,10}); // var1 equals a cubic
    Bezier curve geometry composed of 10 points from p0 to p3.
geometry var2 <- curve({0,0}, {0,10}, {10,10}); // var2 equals a cubic
    Bezier curve geometry composed of 10 points from p0 to p3.
geometry var3 <- curve({0,0},{10,10}, 0.5, false, 100); // var3 equals
    a cubic Bezier curve geometry composed of 100 points from p0 to p1
    at the right side.
geometry var4 <- curve({0,0},{10,10}, 0.5, false, 100, 0.8); // var4
    equals a cubic Bezier curve geometry composed of 100 points from p0
    to p1 at the right side.
geometry var5 <- curve({0,0}, {0,10}, {10,10}, 20); // var5 equals a
    quadratic Bezier curve geometry composed of 20 points from p0 to p2.
geometry var6 <- curve({0,0},{10,10}, 0.5, 100, 0.8, 90); // var6
    equals a cubic Bezier curve geometry composed of 100 points from p0
```

```
    to p1 at the right side.
geometry var7 <- curve({0,0},{10,10}, 0.5); // var7 equals a cubic
    Bezier curve geometry composed of 10 points from p0 to p1.
geometry var8 <- curve({0,0},{10,10}, 0.5, 90); // var8 equals a cubic
    Bezier curve geometry composed of 100 points from p0 to p1 at the
    right side.
geometry var9 <- curve({0,0},{10,10}, 0.5, false); // var9 equals a
    cubic Bezier curve geometry composed of 10 points from p0 to p1 at
    the left side.
geometry var10 <- curve({0,0}, {0,10}, {10,10}); // var10 equals a
    quadratic Bezier curve geometry composed of 10 points from p0 to p2.
```

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,

---

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 and a specific locale (e.g. 'fr', 'en'...). The pattern can use "%Y %M %N %D %E %h %m %s %z" for parsing 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 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

Examples:

```
date d <- date("1999-january-30", 'yyyy-MMMM-dd', 'en');
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,

---

dem

Possible uses:

- **dem** (**file**) —> **geometry**
- **file dem float** —> **geometry**
- **dem** (**file** , **float**) —> **geometry**
- **file dem file** —> **geometry**
- **dem** (**file** , **file**) —> **geometry**
- **dem** (**file**, **file**, **float**) —> **geometry**

Result:

A polygon that is equivalent to the surface of the texture

Examples:

```
geometry var0 <- dem(dem); // var0 equals returns a geometry as a
    rectangle of width and height equal to the texture.
geometry var1 <- dem(dem,texture,z_factor); // var1 equals a geometry
    as a rectangle of width and height equal to the texture.
geometry var2 <- dem(dem,z_factor); // var2 equals a geometry as a
    rectangle of weight and height equal to the texture.
geometry var3 <- dem(dem,texture); // var3 equals a geometry as a
    rectangle of weight and height equal to the texture.
```

---

det

Same signification as determinant

---

determinant

Possible uses:

- **determinant** (**matrix**<**unknown**>) —> **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

---

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 disjoints from the right-geometry (or agent/point).

Special cases:

- if one of the operand is null, returns true.
- if one operand is a point, returns false if the point is included in the geometry.

Examples:

```
bool var0 <- polyline([{10,10},{20,20}]) disjoint_from polyline
    ([{15,15},{25,25}]); // var0 equals false
bool var1 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from
    polygon([{15,15},{15,25},{25,25},{25,15}]); // var1 equals false
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from
    {15,15}; // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from
    {25,25}; // var3 equals true
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) disjoint_from
    polygon([{35,35},{35,45},{45,45},{45,35}]); // var4 equals true
```

See also:

intersects, crosses, overlaps, partially_overlaps, touches,

---

distance_between

Possible 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:

- **point distance_to point** —> **float**
- **distance_to** (**point** , **point**) —> **float**
- **geometry distance_to geometry** —> **float**
- **distance_to** (**geometry** , **geometry**) —> **float**

Result:

A distance between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

Examples:

```
float var0 <- ag1 distance_to ag2; // var0 equals the distance between
   ag1 and ag2 considering the topology of the agent applying the
   operator
```

See also:

towards, direction_to, distance_between, direction_between, path_between, path_to,

---

distinct

Possible uses:

- **distinct** (**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 nil, remove_duplicates returns nil

```
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 elments with duplicated.

```
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]
```

---

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, nbbins) or (list,nbbins,valmin,valmax)

Examples:

```
map var0 <- distribution_of([1,1,2,12.5]); // var0 equals map(['values
    '::[2,1,0,0,0,0,1,0,0,0],'legend
    '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]','[12.0:1
    parlist'::[1,0]])
map var1 <- distribution_of([1,1,2,12.5],10); // var1 equals map(['
    values'::[2,1,0,0,0,0,1,0,0,0],'legend
    '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]','[12.0:1
    parlist'::[1,0]])
map var2 <- distribution_of([1,1,2,12.5]); // var2 equals map(['values
    '::[2,1,0,0,0,0,1,0,0,0],'legend
    '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]','[12.0:1
    parlist'::[1,0]])
```

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, nbbins) or (list,nbbins,valmin,valmax)

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]','[
    parlist'::[1,0]])
map var1 <- distribution2d_of([1,1,2,12.5]); // var1 equals map(['
    values'::[2,1,0,0,0,0,1,0,0,0],'legend
    '::['[0.0:2.0]','[2.0:4.0]','[4.0:6.0]','[6.0:8.0]','[8.0:10.0]','[10.0:12.0]','[
    parlist'::[1,0]])
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]','[
    parlist'::[1,0]])
```

See also:

as_map,

---

div

Possible uses:

- `int div float` —> `int`
- `div (int , float)` —> `int`
- `float div int` —> `int`
- `div (float , int)` —> `int`
- `float div float` —> `int`
- `div (float , float)` —> `int`
- `int div int` —> `int`
- `div (int , int)` —> `int`

Result:

Returns the truncation of the division of the left-hand operand by the right-hand operand.

Special cases:

- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.
- if the right-hand operand is equal to zero, raises an exception.

Examples:

```
int var0 <- 40 div 4.1; // var0 equals 9
int var1 <- 40.5 div 3; // var1 equals 13
int var2 <- 40.1 div 4.5; // var2 equals 8
int var3 <- 40 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: symetric1) returns the dynamic time warping between the two series of values (step pattern used: symetric1) 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
```

---

dxf_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**
- **pair edge int** —> **unknown**
- **edge** (**pair** , **int**) —> **unknown**
- **unknown edge unknown** —> **unknown**
- **edge** (**unknown** , **unknown**) —> **unknown**

- unknown edge int —> unknown
- edge (unknown , int) —> unknown
- edge (unknown, unknown, float) —> unknown
- edge (unknown, unknown, int) —> unknown
- edge (pair, unknown, int) —> unknown
- edge (unknown, unknown, unknown) —> unknown
- edge (pair, unknown, float) —> unknown
- edge (unknown, unknown, unknown, float) —> unknown
- edge (unknown, unknown, unknown, int) —> unknown

---

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**

---

eigenvalues

Possible uses:

- **eigenvalues** (**matrix**<**unknown**>) —> **list**<**float**>

Result:

The eigen values (matrix) of the given matrix

Examples:

```
list<float> var0 <- eigenvalues(matrix([[5,-3],[6,-4]])); // var0
    equals [2.0000000000000004,-0.9999999999999998]
```

---

electre_DM

Possible 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. An candidate outranks another if this one is at least as good as the other one. The ELECTRE methods are based on two concepts: the concordance and the discordance. The concordance characterizes the fact that, for an outranking relation to be validated, a sufficient majority of criteria should be in favor of this assertion. The discordance characterizes the fact that, for an outranking relation to be validated, none of the criteria in the minority should oppose too strongly this assertion. These two conditions must be true for validating the outranking assertion. More information about the ELECTRE methods can be found in [http://www.springerlink.com/content/g367r44322876223/ Figueira, J., Mousseau, V., Roy, B.: ELECTRE Methods. In: Figueira, J., Greco, S., and Ehrgott, M., (Eds.), Multiple Criteria Decision Analysis: State of the Art Surveys, Springer, New York, 133–162 (2005)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fives elements: a name, a weight, a preference value (p), an indifference value (q) and a veto value (v). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant. The veto value represents the threshold from which the difference between two criterion values disqualifies the candidate that obtained the smaller value; the last operand is the fuzzy cut.

Special cases:

- returns -1 is the list of candidates is nil or empty

Examples:

```
int var0 <- electre_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"
    utility", "weight" :: 2.0,"p"::0.5, "q"::0.0, "s"::1.0, "maximize"
    :: true],["name"::"price", "weight" :: 1.0,"p"::0.5, "q"::0.0, "s"
    ::1.0, "maximize" :: false]],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,

---

emotion

Possible uses:

- **emotion** (**any**) —> **emotion**

Result:

---

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 ('abced'); // var0 equals false
```

- if it is a list, empty returns true if there is no element in the list, and false otherwise

```
bool var1 <- empty([]); // var1 equals true
```

---

enlarged_by

Same signification as +

---

envelope

Possible 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 envolpe
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
- list every int —> list
- every (list , int) —> list
- list every any expression —> list<date>
- every (list , any expression) —> list<date>

Result:

true every operand * cycle, false 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 applies a step to an interval of dates defined by 'date1 to date2'

Comment:

the value of the every operator depends on the cycle. It can be used to do something every x cycle.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

Examples:

```
if every(2#cycle) {write "the cycle number is even";}
         else {write "the cycle number is odd";}
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
(date('2000-01-01') to date('2010-01-01')) every (#month) // builds an
   interval between these two dates which contains all the monthly
   dates starting from the beginning of the interval
```

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 ([http://www.glennshafer.com/books/amte.html Shafer G (1976) A mathematical theory of evidence, Princeton University Press]), is based on the work of Dempster ([http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/117 Dempster A (1967) Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, vol. 38, pp. 325–339]) on lower and upper probability distributions. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains seven elements: a name, a first threshold s1, a second threshold s2, a value for the assertion "this candidate is the best" at threshold s1 (v1p), a value for the assertion "this candidate is the best" at threshold s2 (v2p), a value for the assertion "this candidate is not the best" at threshold s1 (v1c), a value for the assertion "this candidate is not the best" at threshold s2 (v2c). v1p, v2p, v1c and v2c have to been defined in order that: v1p + v1c <= 1.0; v2p + v2c <= 1.0.; the last operand allows to use a simple version of this multi-criteria decision making method (simple if true)

Special cases:

- returns -1 is the list of candidates is nil or empty
- if the operator is used with only 2 operands (the candidates and the criteria), the last parameter (use simple method) is set to true

Examples:

```
int var0 <- evidence_theory_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["
    name"::"utility", "s1" :: 0.0,"s2"::1.0, "v1p"::0.0, "v2p"::1.0, "
    v1c"::0.0, "v2c"::0.0, "maximize" :: true],["name"::"price",  "s1"
    :: 0.0,"s2"::1.0, "v1p"::0.0, "v2p"::1.0, "v1c"::0.0, "v2c"::0.0, "
    maximize" :: true]], false); // 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]]); // 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.
- 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,

---

file

Possible uses:

- **file** (**string**) —> **file**
- **string file container** —> **file**
- **file** (**string** , **container**) —> **file**

Result:

Creates a file in read/write mode, setting its contents to the container passed in parameter opens a file in read only mode, creates a GAML file object, and tries to determine and store the file content in the contents attribute.

Comment:

The type of container to pass will depend on the type of file (see the management of files in the documentation). Can be used to copy files since files are considered as containers. For example: save file('image_copy.png', file('image.png')); will copy image.png to image_copy.pngThe file should have a supported extension, see file type definition for supported file extensions.

Special cases:

- If the specified string does not refer to an existing file, an exception is risen when the variable is used.

Examples:

```
let fileT type: file value: file("../includes/Stupid_Cell.Data");
            // fileT represents the file "../includes/Stupid_Cell.Data"
            // fileT.contents here contains a matrix storing all the
   data of the text file
```

See also:

folder, new_folder,

---

file

Possible uses:

- **file** (**any**) —> **file**

---

file_exists

Possible uses:

- **file_exists** (**string**) —> **bool**

Result:

Test whether the parameter is the path to an existing file.

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

```
unknown var4 <- [1::2, 3::4, 5::6] first_with (each >= 4); // var4
    equals 4
unknown var5 <- [1::2, 3::4, 5::6].pairs first_with (each.value >= 4);
    // var5 equals (3::4)
```

Examples:

```
unknown var0 <- [1,2,3,4,5,6,7,8] first_with (each > 3); // var0 equals
     4
unknown var2 <- g2 first_with (length(g2 out_edges_of each) = 0); //
    var2 equals node9
unknown var3 <- (list(node) first_with (round(node(each).location.x) >
    32); // var3 equals node2
```

See also:

group_by, last_with, where,

---

flip

Possible 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**

---

floor

Possible uses:

- **floor** (**float**) —> **float**

Result:

Maps the operand to the largest previous following integer, i.e. the largest integer not greater than x.

Examples:

```
float var0 <- floor(3); // var0 equals 3.0
float var1 <- floor(3.5); // var1 equals 3.0
float var2 <- floor(-4.7); // var2 equals -5.0
```

See also:

ceil, round,

---

folder

Possible 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,

---

font

Possible uses:

- **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 is 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). 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, weights). Reference: Visser, H., and T. de Nijs, 2006. The map comparison kit, Environmental Modelling & Software, 21

Examples:

```
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2],[cat2,
    cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,cat2,cat3
    ],[[1,0,0],[0,1,0],[0,0,1]], 2)
fuzzy_kappa([ag1, ag2, ag3, ag4, ag5],[cat1,cat1,cat2,cat3,cat2],[cat2,
    cat1,cat2,cat1,cat2], similarity_per_agents,[cat1,cat2,cat3
    ],[[1,0,0],[0,1,0],[0,0,1]], 2, [1.0,3.0,2.0,2.0,4.0])
```

---

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
     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
     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:

- gaml_file(string): This file constructor allows to read a gaml file (.gaml)

```
file f <- gaml_file("file.gaml");
```

- gaml_file(string,string,string): This file constructor allows to compile a gaml file and run an experiment

```
file f <- gaml_file("file.gaml", "my_experiment", "my_model");
```

See also:

is_gaml,

---

gaml_type

Possible uses:

- **gaml_type** (**any**) —> **gaml_type**

---

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_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**) —> **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) —> float

Result:

returns the gamma index of the graph (A measure of connectivity that considers the relationship between the number of observed links and the number of possible links: gamma = e/(3 * (v - 2)) - for planar graph.

Examples:

```
graph graphEpidemio <- graph([]);
float var1 <- gamma_index(graphEpidemio); // var1 equals the gamma
   index of the graph
```

See also:

alpha_index, beta_index, nb_cycles, connectivity_index,

---

gamma_rnd

Possible uses:

- float gamma_rnd float —> float
- gamma_rnd (float , float) —> float

Result:

returns a random value from a gamma distribution with specified values of the shape and scale parameters

Examples:

```
float var0 <- gamma_distribution_complemented(2,3,0.9) with_precision
    (3); // var0 equals 0.731
```

---

gauss

Possible uses:

- **gauss** (**point**) —> **float**
- **float gauss float** —> **float**
- **gauss** (**float** , **float**) —> **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 {meand,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:

skew_gauss, truncated_gauss, poisson,

---

generate_barabasi_albert

Possible uses:

- generate_barabasi_albert (container<unknown,agent>, species, int, bool) —> graph
- generate_barabasi_albert (species, species, int, int, bool) —> graph

Result:

returns a random scale-free network (following Barabasi-Albert (BA) model). returns a random scale-free network (following Barabasi-Albert (BA) model).

Comment:

The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:The Barabasi-Albert (BA) model is an algorithm for generating random scale-free networks using a preferential attachment mechanism. A scale-free network is a network whose degree distribution follows a power law, at least asymptotically.Such networks are widely observed in natural and human-made systems, including the Internet, the world wide web, citation networks, and some social networks. [From Wikipedia article]The map operand should includes following elements:

Special cases:

- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “size”: the graph will contain (size + 1) nodes
- “m”: the number of edges added per novel node
- “synchronized”: is the graph and the species of vertices and edges synchronized?
- “vertices_specy”: the species of vertices
- “edges_species”: the species of edges
- “size”: the graph will contain (size + 1) nodes
- “m”: the number of edges added per novel node
- “synchronized”: is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<yourNodeSpecy,yourEdgeSpecy> graphEpidemio <-
    generate_barabasi_albert(
         yourListOfNodes,
         yourEdgeSpecy,
         3,
         5,
         true);
graph<yourNodeSpecy,yourEdgeSpecy> graphEpidemio <-
    generate_barabasi_albert(
         yourNodeSpecy,
         yourEdgeSpecy,
         3,
         5,
         true);
```

See also:

generate_watts_strogatz,

---

generate_complete_graph

Possible uses:

- generate_complete_graph (container<unknown,agent>, species, bool) —> graph
- generate_complete_graph (species, species, int, bool) —> graph
- generate_complete_graph (container<unknown,agent>, species, float, bool) —> graph
- generate_complete_graph (species, species, int, float, bool) —> graph

Result:

returns a fully connected graph. returns a fully connected graph. returns a fully connected graph. returns a fully connected graph.

Comment:

Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:Arguments should include following elements:

Special cases:

- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “synchronized”: is the graph and the species of vertices and edges synchronized?

- “vertices_specy”: the species of vertices
- “edges_species”: the species of edges
- “size”: the graph will contain size nodes.
- “synchronized”: is the graph and the species of vertices and edges synchronized?
- “vertices_specy”: the species of vertices
- “edges_species”: the species of edges
- “size”: the graph will contain size nodes.
- “layoutRadius”: nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- “synchronized”: is the graph and the species of vertices and edges synchronized?
- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “layoutRadius”: nodes of the graph will be located on a circle with radius layoutRadius and centered in the environment.
- “synchronized”: is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(
            myListOfNodes,
            myEdgeSpecy,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(
            myVertexSpecy,
            myEdgeSpecy,
            10,
        true);
```

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(
            myVertexSpecy,
            myEdgeSpecy,
            10, 25,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_complete_graph(
            myListOfNodes,
            myEdgeSpecy,
            25,
        true);
```

See also:

generate_barabasi_albert, generate_watts_strogatz,

---

generate_watts_strogatz

Possible uses:

- **generate_watts_strogatz** (**container<unknown,agent>**, species, **float**, **int**, **bool**) —> **graph**
- **generate_watts_strogatz** (species, species, **int**, **float**, **int**, **bool**) —> **graph**

Result:

returns a random small-world network (following Watts-Strogatz model). returns a random small-world network (following Watts-Strogatz model).

Comment:

The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of

hops or steps. [From Wikipedia article]The map operand should includes following elements:The Watts-Strogatz model is a random graph generation model that produces graphs with small-world properties, including short average path lengths and high clustering.A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops or steps. [From Wikipedia article]The map operand should includes following elements:

Special cases:

- “vertices_specy”: the species of vertices
- “edges_species”: the species of edges
- “size”: the graph will contain (size + 1) nodes. Size must be greater than k.
- “p”: probability to “rewire” an edge. So it must be between 0 and 1. The parameter is often called beta in the literature.
- “k”: the base degree of each node. k must be greater than 2 and even.
- “synchronized”: is the graph and the species of vertices and edges synchronized?
- “agents”: list of existing node agents
- “edges_species”: the species of edges
- “p”: probability to “rewire” an edge. So it must be between 0 and 1. The parameter is often called beta in the literature.
- “k”: the base degree of each node. k must be greater than 2 and even.
- “synchronized”: is the graph and the species of vertices and edges synchronized?

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(
```

```
            myVertexSpecy,
            myEdgeSpecy,
            2,
            0.3,
            2,
        true);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- generate_watts_strogatz(
            myListOfNodes,
            myEdgeSpecy,
            0.3,
            2,
        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**

---

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 geometry (left operand). The attribute name is specified by the right operand. Reads an attribute of the specified agent (left operand). The attribute name is specified by the right operand.

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 (emotion) —> agent
- get_agent_cause (predicate) —> agent

Result:

get the agent cause value of the given emotion evaluate the agent_cause value of a predicate

Examples:

```
get_agent_cause(emotion)
get_agent_cause(pred1)
```

---

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**) —> **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** —> **list<mental_state>**
- **get_beliefs_with_name_op** (**agent** , **string**) —> **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`) —> `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`) —> `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): 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
```

---

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,

---

graph

Possible uses:

- **graph** (**any**) —> **graph**

---

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 * red + 0.587 * green + 0.114 * blue (Photoshop value)

Examples:

```
rgb var0 <- grayscale (rgb(255,0,0)); // var0 equals to a dark grey
```

See also:

rgb, hsb,

---

grid_at

Possible 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**

Result:

creates a graph from a list of cells (operand). An edge is created between neighbors.

Examples:

```
my_cell_graph<-grid_cells_to_graph(cells_list)
```

---

grid_file

Possible uses:

- **grid_file** (**string**) —> **file**
- **string grid_file int** —> **file**
- **grid_file** (**string** , **int**) —> **file**
- **string grid_file string** —> **file**
- **grid_file** (**string** , **string**) —> **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,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");
```

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 with and height

Comment:

the center of the hexagon is by default the location of the current agent in which has been called this operator.the center of the hexagon is by default the location of the current agent in which has been called this operator.the center of the hexagon is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- hexagon({10,5}); // var0 equals a geometry as a
   hexagon of width of 10 and height of 5.
geometry var1 <- hexagon(10); // var1 equals a geometry as a hexagon of
    width of 10 and height of 10.
geometry var2 <- hexagon(10,5); // var2 equals a geometry as a hexagon
   of width of 10 and height of 5.
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, rectangle, triangle,

---

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**>

---

hsb

Possible uses:

- **hsb** (**float**, **float**, **float**) —> **rgb**
- **hsb** (**float**, **float**, **float**, **float**) —> **rgb**
- **hsb** (**float**, **float**, **float**, **int**) —> **rgb**

Result:

Converts hsb (h=hue, s=saturation, b=brightness) value to Gama color

Comment:

h,s and b components should be floating-point values between 0.0 and 1.0 and when used alpha should be an integer (between 0 and 255) or a float (between 0 and 1) . Examples: Red=(0.0,1.0,1.0), Yellow=(0.16,1.0,1.0), Green=(0.33,1.0,1.0), Cyan=(0.5,1.0,1.0), Blue=(0.66,1.0,1.0), Magenta=(0.83,1.0,1.0)

Examples:

```
rgb var0 <- hsb (0.0,1.0,1.0); // var0 equals rgb("red")
rgb var1 <- hsb (0.5,1.0,1.0,0.0); // var1 equals rgb("cyan",0)
```

See also:

rgb,

---

hypot

Possible uses:

- **hypot** (**float**, **float**, **float**, **float**) —> **float**

Result:

Returns sqrt(x2 +y2) without intermediate overflow or underflow.

Special cases:

- If either argument is infinite, then the result is positive infinity. If either argument is NaN and neither argument is infinite, then the result is NaN.

Examples:

```
float var0 <- hypot(0,1,0,1); // var0 equals sqrt(2)
```

IDW

Possible 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_file

Possible uses:

- **image_file** (**string**) —> **file**
- **string image_file matrix**<**int**> —> **file**
- **image_file** (**string** , **matrix**<**int**>) —> **file**

Result:

Constructs a file of type image. Allowed extensions are limited to tiff, jpg, jpeg, png, pict, bmp

Special cases:

- 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,matrix): 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:

- `unknown in container` —> `bool`
- `in` (`unknown` , `container`) —> `bool`
- `string in string` —> `bool`
- `in` (`string` , `string`) —> `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 the right operand is nil or empty, in returns false
- if both operands are strings, returns true if the left-hand operand patterns is included in to the right-hand string;

Examples:

```
bool var0 <- 2 in [1,2,3,4,5,6]; // var0 equals true
bool var1 <- 7 in [1,2,3,4,5,6]; // var1 equals false
bool var2 <- 3 in [1::2, 3::4, 5::6]; // var2 equals false
bool var3 <- 6 in [1::2, 3::4, 5::6]; // var3 equals true
bool var4 <-  'bc' in 'abcded'; // 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**) —> **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` —> `float`
- `incomplete_gamma_complement` (`float` , `float`) —> `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` —> `string`
- `indented_by` (`string` , `int`) —> `string`

Result:

Converts a (possibly multiline) string by indenting it by a number – specified by the second operand – of tabulations to the right

Examples:

```
string var0 <- "my" + indented_by("text", 1); // var0 equals "my
    text"
```

---

index_by

Possible uses:

- container index_by any expression —> map
- index_by (container , any expression) —> map

Result:

produces a new map from the evaluation of the right-hand operand for each element of the left-hand operand

Special cases:

- if the left-hand operand is nil, index_by throws an error. 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:

- 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
- matrix index_of unknown —> point
- index_of (matrix , unknown) —> point

Result:

the index of the first occurence of the right operand in the left operand container the index of the first occurence of the right operand in the left operand container

Comment:

The definition of index_of and the type of the index depend on the container

Special cases:

- if the left operator is a species, returns the index of an agent in a species. If the argument is not an agent of this species, returns -1. Use int(agent) instead
- if the left operand is a map, index_of returns the index of a value or nil if the value is not mapped
- if both operands are strings, returns the index within the left-hand string of the first occurrence of the given right-hand string

```
int var1 <-  "abcabcabc" 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
```

- if the left operand is a matrix, index_of returns the index as a point

```
point var4 <- matrix([[1,2,3],[4,5,6]]) index_of 4; // var4 equals
    {1.0,0.0}
```

Examples:

```
unknown var0 <- [1::2, 3::4, 5::6] index_of 4; // var0 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**

---

inter

Possible uses:

- **container inter container** —> **list**
- **inter** (**container** , **container**) —> **list**
- **geometry inter geometry** —> **geometry**
- **inter** (**geometry** , **geometry**) —> **geometry**

Result:

the intersection of the two operands A geometry resulting from the intersection between the two geometries

Comment:

both containers are transformed into sets (so without duplicated element, cf. remove_duplicates operator) before the set intersection is computed.

Special cases:

- if an operand is a graph, it will be transformed into the set of its nodes
- returns nil if one of the operands is nil
- if an operand is a map, it will be transformed into the set of its values

```
list var0 <- [1::2, 3::4, 5::6] inter [2,4]; // var0 equals [2,4]
list var1 <- [1::2, 3::4, 5::6] inter [1,3]; // var1 equals []
```

- if an operand is a matrix, it will be transformed into the set of the lines

```
list var2 <- matrix([[3,2,1],[4,5,4]]) inter [3,4]; // var2 equals
    [3,4]
```

Examples:

```
list var3 <- [1,2,3,4,5,6] inter [2,4]; // var3 equals [2,4]
list var4 <- [1,2,3,4,5,6] inter [0,8]; // var4 equals []
geometry var5 <- square(10) inter circle(5); // var5 equals circle(5)
```

See also:

remove_duplicates, union, +, -,

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','e12','e13'],['e21','e22','e23'],['e31
    ','e32','e33']]); // var1 equals ['e11','e21','e31','e12','e22','e32
    ','e13','e23','e33']
```

---

internal_at

Possible uses:

- **agent internal_at list** —> **unknown**
- **internal_at** (**agent** , **list**) —> **unknown**
- **geometry internal_at list** —> **unknown**
- **internal_at** (**geometry** , **list**) —> **unknown**
- **container**<KeyType,ValueType> **internal_at list**<KeyType> —> ValueType
- **internal_at** (**container**<KeyType,ValueType> , **list**<KeyType>) —> ValueType

Result:

For internal use only. Corresponds to the implementation, for agents, of the access to containers with index For internal use only. Corresponds to the implementation, for geometries, of the access to containers with index For internal use only. Corresponds to the implementation of the access to containers with index

See also:

at,

---

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

---

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**<**unknown**>) —> **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>) —> 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,

---

is

Possible uses:

- `unknown is any expression` —> `bool`
- `is` (`unknown` , `any expression`) —> `bool`

Result:

returns true if the left operand is of the right operand type, false otherwise

Examples:

```
bool var0 <- 0 is int; // var0 equals true
bool var1 <- an_agent is node; // var1 equals true
bool var2 <- 1 is float; // var2 equals false
```

---

is_csv

Possible uses:

- `is_csv` (`any`) —> `bool`

Result:

Tests whether the operand is a csv file.

See also:

csv_file,

---

is_dxf

Possible uses:

- `is_dxf` (`any`) —> `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_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
bool var1 <- is_number(#infinity); // var1 equals true
bool var2 <- is_number(#nan); // var2 equals false
bool var3 <- is_number("test"); // var3 equals false
bool var4 <- is_number("123.56"); // var4 equals true
bool var5 <- is_number("-1.2e5"); // var5 equals true
bool var6 <- is_number("1,2"); // var6 equals false
bool var7 <- is_number("#12FA"); // var7 equals true
```

---

is_obj

Possible 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_R

Possible uses:

- `is_R` (`any`) —> `bool`

Result:

Tests whether the operand is a R file.

See also:

R_file,

---

is_saved_simulation

Possible uses:

- `is_saved_simulation` (`any`) —> `bool`

Result:

Tests whether the operand is a saved_simulation file.

See also:

saved_simulation_file,

---

is_shape

Possible uses:

- `is_shape` (`any`) —> `bool`

Result:

Tests whether the operand is a shape file.

See also:

shape_file,

---

is_skill

Possible uses:

- **unknown is_skill string** —> **bool**
- **is_skill** (**unknown** , **string**) —> **bool**

Result:

returns true if the left operand is an agent whose species implements the right-hand skill name

Examples:

```
bool var0 <- agentA is_skill 'moving'; // var0 equals true
```

---

is_svg

Possible 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,

---

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, weights). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8) kappa simulation indicator for 2 map comparisons: kappa(list_valsInits,list_valsObs,list_valsSim, categories). Reference: van Vliet, J., Bregt, A.K. & Hagen-Zanker, A. (2011). Revisiting Kappa to account for change in the accuracy assessment of land-use change models, Ecological Modelling 222(8).

Examples:

```
float var0 <- 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]); // var0 equals
    0.2702702702702703
float var1 <- kappa_sim(["cat1","cat1","cat2","cat2","cat2"],["cat1","
    cat3","cat2","cat1","cat3"],["cat1","cat3","cat2","cat3","cat1"],["
    cat1","cat2","cat3"]); // var1 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). Usage: kmeans(data,k) returns the list of clusters (list of instance indices) computed with the kmeans++ algorithm from the first operand data according to the number of clusters to split the data into (k) and the maximum number of iterations to run the algorithm for (If negative, no maximum will be used) (maxIt). Usage: kmeans(data,k,maxit)

Special cases:

- if the lengths of two vectors in the right-hand aren't equal, returns 0
- if the lengths of two vectors in the right-hand aren't equal, returns 0

Examples:

```
list<list> var0 <- kmeans ([[2,4,5], [3,8,2], [1,1,3], [4,3,4]],2); //
    var0 equals [[0,2,3],[1]]
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**

---

kurtosis

Possible uses:

- **kurtosis** (**list**) —> **float**

Result:

returns kurtosis value computed from the operand list of values (kurtosis = { [n(n+1) / (n -1)(n - 2)(n-3)] sum[(x_i - mean)^4] / 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 ([1,2,3,4,5]); // var0 equals -1.200000000000002
```

---

kurtosis

Possible uses:

- **kurtosis** (**container**) —> **float**
- **float kurtosis float** —> **float**
- **kurtosis** (**float** , **float**) —> **float**

Result:

Returns the kurtosis (aka excess) of a data sequence Returns the kurtosis (aka excess) of a data sequence

Examples:

```
float var0 <- kurtosis(3,12) with_precision(4); // var0 equals -2.9999
float var1 <- kurtosis([13,2,1,4,1,2]) with_precision(4); // var1
   equals 4.8083
```

---

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
- 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
- matrix last_index_of unknown —> point
- last_index_of (matrix , unknown) —> point
- list last_index_of unknown —> int
- last_index_of (list , unknown) —> int

Result:

the index of the last occurence of the right operand in the left operand container

Comment:

The definition of last_index_of and the type of the index depend on the container

Special cases:

- if the left operand is a species, the last index of an agent is the same as its index
- if the left operand is a map, last_index_of returns the index as an int (the key of the pair)

```
unknown var0 <- [1::2, 3::4, 5::4] last_index_of 4; // var0 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 var1 <- "abcabcabc" last_index_of "ca"; // 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 list, last_index_of returns the index as an integer

```
int var3 <- [1,2,3,4,5,6] last_index_of 4; // var3 equals 3
int var4 <- [4,2,3,4,5,4] last_index_of 4; // 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:

```
unknown var0 <- [1,2,3,4,5,6,7,8] last_with (each > 3); // var0 equals
    8
unknown var2 <- g2 last_with (length(g2 out_edges_of each) = 0 ); //
    var2 equals node11
unknown var3 <- (list(node) last_with (round(node(each).location.x) >
    32); // var3 equals node3
```

See also:

group_by, first_with, where,

layout_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 ! 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. usage: layoutForce(graph, bounds, coeff_force, cooling_rate, max_iteration, equilibirum 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 coefficien use 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; equilibirum criterion is the maximaldistance of displacement for a vertice to be considered as in equilibrium

layouts a GAMA graph using Force model. 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

---

layout_grid

Possible uses:

- **layout_grid** (**graph**, **geometry**, **float**) —> **graph**

Result:

layouts a Gama graph based on a grid latice. usage: layoutForce(graph, bounds, coeff_nb_cells). graph is the graph to which " + "applied the layout; 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>) —> geometry
- container<unknown,geometry> line float —> geometry
- line (container<unknown,geometry> , float) —> geometry

Result:

A polyline geometry from the given list of points. A polyline geometry from the given list of points represented as a cylinder of radius r.

Special cases:

- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.
- if the operand is nil, returns the point geometry {0,0}
- if the operand is composed of a single point, returns a point geometry.
- if a radius is added, the given list of points represented as a cylinder of radius r

```
geometry var3 <- polyline([{0,0}, {0,10}, {10,10}, {10,0}],0.2); //
    var3 equals a polyline geometry composed of the 4 points.
```

Examples:

```
geometry var0 <- polyline([{0,0}, {0,10}, {10,10}]); // var0 equals a
    polyline geometry composed of the 3 points.
geometry var1 <- line([{10,10}, {10,0}]); // var1 equals a line from 2
    points.
string var2 <- string(polyline([{0,0}, {0,10}, {10,10}])+line([{10,10},
     {10,0}])); // var2 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** —> **geometry**
- **link** (**geometry** , **geometry**) —> **geometry**

Result:

A dynamic line geometry between the location of the two operands

Comment:

The geometry of the link is a line between the locations of the two operands, which is built and maintained dynamically

Special cases:

- if one of the operands is nil, link returns a point geometry at the location of the other. If both are null, it returns a point geometry at {0,0}

Examples:

```
geometry var0 <- link (geom1,geom2); // var0 equals a link geometry
    between geom1 and geom2.
```

See also:

around, circle, cone, line, norm, point, polygon, polyline, rectangle, square, triangle,

---

list

Possible uses:

- `list` (`any`) —> `list`

---

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_graph_from_file

Possible uses:

- load_graph_from_file (string) —> graph
- string load_graph_from_file string —> graph
- load_graph_from_file (string , string) —> graph
- string load_graph_from_file file —> graph
- load_graph_from_file (string , file) —> graph
- load_graph_from_file (string, species, species) —> graph
- load_graph_from_file (string, string, species, species) —> graph
- load_graph_from_file (string, file, species, species) —> graph
- load_graph_from_file (string, string, species, species, bool) —> graph

Result:

loads a graph from a file returns a graph loaded from a given file encoded into a given format. The last boolean parameter indicates whether the resulting graph will be considered as spatial or not by GAMA

Comment:

Available formats: “pajek”: Pajek (Slovene word for Spider) is a program, for Windows, for analysis and visualization of large networks. See: http://pajek.imfm.si/doku.php?id=pajek for more details.“lgl”: LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details.“dot”: DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. See: http://en.wikipedia.org/wiki/DOT_language for more details.“edge”: This format is a simple text file with numeric vertex ids defining the edges.“gexf”: GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications. See: http://gexf.net/format/ for more details.“graphml”: GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details.“tlp” or “tulip”: TLP is

the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. “ncol”: This format is used by the Large Graph Layout progra. It is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge. See: http://bioinformatics.icmb.utexas.edu/lgl for more details.The map operand should includes following elements:Available formats: “pajek”: Pajek (Slovene word for Spider) is a program, for Windows, for analysis and visualization of large networks. See: http://pajek.imfm.si/doku.php?id=pajek for more details.“lgl”: LGL is a compendium of applications for making the visualization of large networks and trees tractable. See: http://lgl.sourceforge.net/ for more details.“dot”: DOT is a plain text graph description language. It is a simple way of describing graphs that both humans and computer programs can use. See: http://en.wikipedia.org/wiki/DOT_language for more details.“edge”: This format is a simple text file with numeric vertex ids defining the edges.“gexf”: GEXF (Graph Exchange XML Format) is a language for describing complex networks structures, their associated data and dynamics. Started in 2007 at Gephi project by different actors, deeply involved in graph exchange issues, the gexf specifications are mature enough to claim being both extensible and open, and suitable for real specific applications. See: http://gexf.net/format/ for more details.“graphml”: GraphML is a comprehensive and easy-to-use file format for graphs based on XML. See: http://graphml.graphdrawing.org/ for more details.“tlp” or “tulip”: TLP is the Tulip software graph format. See: http://tulip.labri.fr/TulipDrupal/?q=tlp-file-format for more details. “ncol”: This format is used by the Large Graph Layout progra. It is simply a symbolic weighted edge list. It is a simple text file with one edge per line. An edge is defined by two symbolic vertex names separated by whitespace. (The symbolic vertex names themselves cannot contain whitespace.) They might followed by an optional number, this will be the weight of the edge. See: http://bioinformatics.icmb.utexas.edu/lgl for more details.The map operand should includes following elements:

Special cases:

- “format”: the format of the file
- “filename”: the filename of the file containing the network

- "edges_species": the species of edges
- "vertices_specy": the species of vertices
- "format": the format of the file
- "filename": the filename of the file containing the network
- "edges_species": the species of edges
- "vertices_specy": the species of vertices
- "format": the format of the file, "filename": the filename of the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

- "filename": the filename of the file containing the network, "edges_species": the species of edges, "vertices_specy": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy );
```

- "format": the format of the file, "file": the file containing the network, "edges_species": the species of edges, "vertices_specy": the species of vertices

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy );
```

- "format": the format of the file, "file": the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

- "file": the file containing the network

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "example_of_Pajek_file");
```

Examples:

```
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy);
graph<myVertexSpecy,myEdgeSpecy> myGraph <- load_graph_from_file(
            "pajek",
            "./example_of_Pajek_file",
            myVertexSpecy,
            myEdgeSpecy , true);
```

---

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** (**int**) —> **float**
- **log** (**float**) —> **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(1); // var0 equals 0.0
float var1 <- log(10); // var1 equals 1.0
```

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
```

---

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**

---

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.
```

---

material

Possible uses:

- `float` `material` `float` —> `material`
- `material` (`float` , `float`) —> `material`

Result:

Returns

Examples:

See also:

,

---

material

Possible uses:

- `material` (`any`) —> `material`

---

matrix

Possible uses:

- `matrix` (`any`) —> `matrix`

---

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

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 of float, max returns the maximum of all the elements

```
unknown var0 <- max ([100, 23.2, 34.5]); // var0 equals 100.0
```

- if it is a list of points: max returns the maximum of all points as a point (i.e. the point with the greatest coordinate on the x-axis, in case of equality the point with the greatest coordinate on the y-axis is chosen. If all the points are equal, the first one is returned.)

```
unknown var1 <- max([{1.0,3.0},{3.0,5.0},{9.0,1.0},{7.0,8.0}]); // var1
    equals {9.0,1.0}
```

See also:

min,

---

max_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,

meanR

Possible uses:

- **meanR** (**container**) —> **unknown**

Result:

returns the mean value of given vector (right-hand operand) in given variable (left-hand operand).

Examples:

```
list<int> X <- [2, 3, 1];
int var1 <- meanR(X); // var1 equals 2
unknown var2 <- meanR([2, 3, 1]); // var2 equals 2
```

---

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:

---

message

Possible uses:

- **message** (**unknown**) —> msi.gama.extensions.messaging.GamaMessage

Result:

to be added

---

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**) —> **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** —> **date**
- **minus_weeks** (**date** , **int**) —> **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** —> **date**
- **minus_years** (**date** , **int**) —> **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** —> **int**
- **mod** (**int** , **int**) —> **int**

Result:

Returns the remainder of the integer division of the left-hand operand by the right-hand operand.

Special cases:

- if operands are float, they are truncated
- if the right-hand operand is equal to zero, raises an exception.

Examples:

```
int var0 <- 40 mod 3; // var0 equals 1
```

See also:

div,

---

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
```

---

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,

---

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**
- **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 agent —> emotion
- new_emotion (string , agent) —> emotion
- string new_emotion float —> emotion
- new_emotion (string , float) —> emotion
- string new_emotion predicate —> emotion
- new_emotion (string , predicate) —> emotion
- new_emotion (string, predicate, agent) —> emotion
- new_emotion (string, float, float) —> emotion
- new_emotion (string, float, agent) —> emotion
- new_emotion (string, float, predicate) —> emotion
- new_emotion (string, float, predicate, agent) —> emotion
- new_emotion (string, float, predicate, float) —> emotion
- new_emotion (string, float, float, agent) —> emotion
- new_emotion (string, float, predicate, float, agent) —> emotion

Result:

a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name, intensity) a new emotion with the given properties (name) a new emotion with the given properties (name,intensity,decay) a new emotion with the given properties (name) a new emotion with the given properties (name) a new emotion with the given properties (name,about) a new emotion with the given properties (name) a new emotion with the given properties (name,intensity,about) a new emotion with the given properties (name)

Examples:

```
emotion("joy",12.3,eatFood,4)
emotion("joy",12.3,eatFood,4)
emotion("joy",12.3,eatFood,4)
emotion("joy",12.3)
emotion("joy")
emotion("joy",12.3,4.0)
emotion("joy",12.3,eatFood,4)
emotion("joy",12.3,eatFood,4)
emotion("joy",eatFood)
emotion("joy",12.3,eatFood,4)
emotion("joy",12.3,eatFood)
emotion("joy",12.3,eatFood,4)
```

---

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,

---

new_mental_state

Possible uses:

- **new_mental_state** (**string**) —> **mental_state**
- **string new_mental_state predicate** —> **mental_state**
- **new_mental_state** (**string** , **predicate**) —> **mental_state**
- **string new_mental_state emotion** —> **mental_state**
- **new_mental_state** (**string** , **emotion**) —> **mental_state**
- **string new_mental_state mental_state** —> **mental_state**
- **new_mental_state** (**string** , **mental_state**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **int**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **float**) —> **mental_state**
- **new_mental_state** (**string**, **mental_state**, **int**) —> **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**, **mental_state**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **float**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **int**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **mental_state**, **float**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **mental_state**, **int**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **float**, **int**) —> **mental_state**
- **new_mental_state** (**string**, **mental_state**, **float**, **int**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **int**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **float**, **int**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **float**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **emotion**, **float**, **int**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **mental_state**, **float**, **int**, **agent**) —> **mental_state**
- **new_mental_state** (**string**, **predicate**, **float**, **int**, **agent**) —> **mental_state**

Result:

a new mental state a new mental state

Examples:

```
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
```

```
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
new_mental-state(belief)
```

---

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**, **agent**) —> **predicate**
- **new_predicate** (**string**, **map**, **bool**) —> **predicate**
- **new_predicate** (**string**, **map**, **bool**, **agent**) —> **predicate**

Result:

a new predicate with the given properties (name, values, agentCause) a new predicate with the given properties (name, values, is_true) a new predicate with the given properties (name, values) a new predicate with the given properties (name, values, is_true, agentCause) a new predicate with the given properties (name, values, lifetime)

a new predicate with the given properties (name) a new predicate with the given is_true (name, is_true)

Examples:

```
predicate("people to meet", ["time"::10], agentA)
predicate("people to meet", ["time"::10], true)
predicate("people to meet", people1 )
predicate("people to meet", ["time"::10], true, agentA)
predicate("people to meet", ["time"::10], true)
predicate("people to meet")
predicate("hasWater", true)
```

---

new_social_link

Possible uses:

- **new_social_link** (**agent**) —> **social_link**
- **new_social_link** (**agent**, **float**, **float**, **float**, **float**) —> **social_link**

Result:

a new social link a new social link

Examples:

```
new_social_link(agentA,0.0,-0.1,0.2,0.1)
new_social_link(agentA)
```

---

node

Possible uses:

- node (unknown) —> unknown
- unknown node float —> unknown
- node (unknown , float) —> unknown

---

nodes

Possible uses:

- nodes (container) —> container

---

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. Returns true if the left-hand operand is empty. '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

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:

---

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.Inverse returns the value in terms of standard deviations from the mean, so we need to adjust it for the given mean and standard deviation.

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,

---

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 rotationis 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 rotationis 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 it is a graph, one_of returns one of the lists of edges
- if it is a file, one_of returns one of the elements of the content of the file (that is also a container)
- if the operand is empty, one_of returns nil

- if it is a list or a matrix, one_of returns one of the values of the list or of the matrix

```
i <- any ([1,2,3]); //i equals 1, 2 or 3
string sMat <- one_of(matrix([["c11","c12","c13"],["c21","c22","c23"]])
   );   // sMat equals "c11","c12","c13", "c21","c22" or "c23"
```

- if it is a map, one_of returns one the value of a random pair of the map

```
int im <- one_of ([2::3, 4::5, 6::7]);  // im equals 3, 5 or 7
bool var6 <- [2::3, 4::5, 6::7].values contains im; // var6 equals true
```

- if it is a population, one_of returns one of the agents of the population

```
bug b <- one_of(bug);   // Given a previously defined species bug, b is
    one of the created bugs, e.g. bug3
```

See also:

contains,

one_verifies

Same signification as one_matches

---

open_simplex_generator

Possible uses:

- `open_simplex_generator` (`float`, `float`, `float`) —> `float`

Result:

take a x, y and a bias parameters and gives a value

Examples:

```
float var0 <- open_simplex_generator(2,3,253); // var0 equals 10.2
```

---

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
bool var1 <- false or false; // var1 equals false
bool var2 <- false or true; // var2 equals true
bool var3 <- true or true; // var3 equals true
 int a <-3 ; int b <- 4; int c <- 7;
bool var5 <- ((a+b) = c ) or ((a+b) > c ); // var5 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 attibute 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 attibute '
    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 'aminity' (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
    polygon([{15,15},{15,25},{25,25},{25,15}]); // var1 equals true
bool var2 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    {25,25}; // var2 equals false
bool var3 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    polygon([{35,35},{35,45},{45,45},{45,35}]); // var3 equals false
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    polyline([{10,10},{20,20}]); // var4 equals true
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    {15,15}; // var5 equals true
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    polygon([{0,0},{0,30},{30,30}, {30,0}]); // var6 equals true
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    polygon([{15,15},{15,25},{25,25},{25,15}]); // var7 equals true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) overlaps
    polygon([{10,20},{20,20},{20,30},{10,30}]); // var8 equals true
```

See also:

disjoint_from, crosses, intersects, partially_overlaps, touches,

---

pair

Possible uses:

- `pair` (`any`) —> `pair`

---

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 polygon([{35,35},{35,45},{45,45},{45,35}]); //
    var3 equals false
bool var4 <- polygon([{10,10},{10,20},{20,20},{20,10}])
    partially_overlaps polyline([{10,10},{20,20}]); // var4 equals false
bool var5 <- polygon([{10,10},{10,20},{20,20},{20,10}])
    partially_overlaps {15,15}; // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}])
    partially_overlaps polygon([{0,0},{0,30},{30,30}, {30,0}]); // var6
    equals false
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}])
    partially_overlaps polygon([{15,15},{15,25},{25,25},{25,15}]); //
    var7 equals true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}])
    partially_overlaps polygon([{10,20},{20,20},{20,30},{10,30}]); //
    var8 equals false
```

See also:

disjoint_from, crosses, overlaps, intersects, touches,

---

path

Possible uses:

- **path** (**any**) —> **path**

Result:

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:

- list<agent> path_between container<unknown,geometry> —> path
- path_between (list<agent> , container<unknown,geometry>) —> path
- 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
- path_between (map<agent,unknown>, geometry, geometry) —> path
- path_between (list<agent>, geometry, geometry) —> path
- path_between (topology, geometry, geometry) —> path
- path_between (graph, geometry, geometry) —> path

Result:

The shortest path between two objects according to set of cells with corresponding weights The shortest path between two objects according to set of cells The shortest

path between several objects according to set of cells The shortest path between a list of two objects in a graph The shortest path between several objects according to set of cells with corresponding weights

Examples:

```
path var0 <- path_between (cell_grid as_map (each::each.is_obstacle ?
    9999.0 : 1.0), ag1, ag2); // var0 equals A path between ag1 and ag2
    passing through the given cell_grid agents with a minimal cost
path var1 <- path_between (cell_grid where each.is_free, ag1, ag2); //
    var1 equals A path between ag1 and ag2 passing through the given
    cell_grid agents
path var2 <- my_topology path_between (ag1, ag2); // var2 equals A path
     between ag1 and ag2
path var3 <- path_between (cell_grid where each.is_free, [ag1, ag2, ag3
    ]); // var3 equals A path between ag1 and ag2 and ag3 passing
    through the given cell_grid agents
path var4 <- my_topology path_between [ag1, ag2]; // var4 equals A path
     between ag1 and ag2
path var5 <- path_between (my_graph, ag1, ag2); // var5 equals A path
    between ag1 and ag2
path var6 <- path_between (cell_grid as_map (each::each.is_obstacle ?
    9999.0 : 1.0), [ag1, ag2, ag3]); // var6 equals A path between ag1
    and ag2 and ag3 passing through the given cell_grid agents with
    minimal cost
```

See also:

towards, direction_to, distance_between, direction_between, path_to, distance_to,

---

path_to

Possible uses:

- **geometry path_to geometry** —> **path**
- **path_to** (**geometry** , **geometry**) —> **path**
- **point path_to point** —> **path**
- **path_to** (**point** , **point**) —> **path**

Result:

A path between two geometries (geometries, agents or points) considering the topology of the agent applying the operator.

Examples:

```
path var0 <- ag1 path_to ag2; // var0 equals the path between ag1 and
    ag2 considering the topology of the agent applying the operator
```

See also:

towards, direction_to, distance_between, direction_between, path_between, distance_to,

---

paths_between

Possible 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,

---

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**

---

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_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 occurence lambda as operand).

Comment:

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event, cf. Poisson distribution on Wikipedia.

Examples:

```
int var0 <- poisson(3.5); // var0 equals a random positive integer
```

See also:

binomial, gauss,

---

polygon

Possible 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:

---

predict

Possible uses:

- **regression predict list** —> **float**
- **predict** (**regression** , **list**) —> **float**

Result:

returns the value predict by the regression parameters for a given instance. Usage: predict(regression, instance)

Examples:

```
predict(my_regression, [1,2,3])
```

---

product

Same signification as mul

---

product_of

Possible 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 [http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6VCT-4VF56TV-1&_user=10&_coverDate=01%2F01%2F2010&_rdoc=1&_fmt=high&_orig=search&_sort=d&_docanchor=&view=c&_searchStrId=1389284642&_rerunOrigin=google&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=d334de2a4e0d6281199a39857648cd36 Behzadian, M., Kazemzadeh, R., Albadvi, A., M., A.: PROMETHEE: A comprehensive literature review on methodologies and applications. European Journal of Operational Research(2009)]. The first operand is the list of candidates (a candidate is a list of criterion values); the second operand the list of criterion: A criterion is a map that contains fours elements: a name, a weight, a preference value (p) and an indifference value (q). The preference value represents the threshold from which the difference between two criterion values allows to prefer one vector of values over another. The indifference value represents the threshold from which the difference between two criterion values is considered significant.

Special cases:

- returns -1 is the list of candidates is nil or empty

Examples:

```
int var0 <- promethee_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["name"::"
    utility", "weight" :: 2.0,"p"::0.5, "q"::0.0, "s"::1.0, "maximize"
    :: true],["name"::"price", "weight" :: 1.0,"p"::0.5, "q"::0.0, "s"
    ::1.0, "maximize" :: false]]); // var0 equals 1
```

See also:

weighted_means_DM, electre_DM, evidence_theory_DM,

---

property_file

Possible 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 degress 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 need not necessarily 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 <= 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
```

R_correlation

Same signification as corR

R_file

Possible uses:

- **R_file** (**string**) —> **file**
- **string R_file map** —> **file**
- **R_file** (**string** , **map**) —> **file**

Result:

Constructs a file of type R. Allowed extensions are limited to r

Special cases:

- R_file(string): This file constructor allows to read a R file

```
file f <-R_file("file.r");
```

- R_file(string,map): This file constructor allows to store a map in a R file (it does not save it - just store it in memory)

```
file f <-R_file("file.r",map(["param1"::1.0,"param2"::10.0 ]));
```

See also:

is_R,

R_mean

Same signification as meanR

---

range

Possible uses:

- **range** (**int**) —> **list**
- **int range int** —> **list**
- **range** (**int** , **int**) —> **list**
- **range** (**int**, **int**, **int**) —> **list**

Result:

Allows to build a list of int representing all contiguous values from the first to the second argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value Allows to build a list of int representing all contiguous values from the first to the second argument, using the step represented by the third argument. The range can be increasing or decreasing. Passing the same value for both will return a singleton list with this value. Passing a step of 0 will result in an exception. Attempting to build infinite ranges (e.g. end > start with a negative step) will similarly not be accepted and yield an exception Allows to build a list of int representing all contiguous values from zero to the argument. The range can be increasing or decreasing. Passing 0 will return a singleton list with 0

Examples:

```
list var0 <- range(0,2); // var0 equals [0,1,2]
```

---

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:

```
agent_name <- read ('name'); //agent_name equals reads the 'name'
    variable of agent then assigns the returned value to the 'agent_name
    ' variable.
```

---

rectangle

Possible uses:

- **rectangle** (**point**) —> **geometry**
- **float rectangle float** —> **geometry**
- **rectangle** (**float** , **float**) —> **geometry**
- **point rectangle point** —> **geometry**
- **rectangle** (**point** , **point**) —> **geometry**

Result:

A rectangle geometry which side sizes are given by the operands.

Comment:

the center of the rectangle is by default the location of the current agent in which has been called this operator.the center of the rectangle is by default the location of the current agent in which has been called this operator.

Special cases:

- returns nil if the operand is nil.
- returns nil if the operand is nil.
- returns nil if the operand is nil.

Examples:

```
geometry var0 <- rectangle(10, 5); // var0 equals a geometry as a
    rectangle with width = 10 and height = 5.
float var1 <- rectangle(10, 5).area; // var1 equals 50.0
geometry var2 <- rectangle({0.0,0.0}, {10.0,10.0}); // var2 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.
float var3 <- rectangle({0.0,0.0}, {10.0,10.0}).area; // var3 equals
    100.0
geometry var4 <- rectangle({10, 5}); // var4 equals a geometry as a
    rectangle with width = 10 and height = 5.
float var5 <- rectangle({10, 5}).area; // var5 equals 50.0
```

See also:

around, circle, cone, line, link, norm, point, polygon, polyline, square, triangle,

---

reduced_by

Same signification as -

---

regression

Possible uses:

- **regression** (**any**) —> **regression**

---

remove_duplicates

Same signification as distinct

---

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)
```

---

replace

Possible uses:

- **replace** (**string**, **string**, **string**) —> **string**

Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding the second operand by the third operand

Examples:

```
string var0 <- replace('to be or not to be,that is the question','to',
    'do'); // var0 equals 'do be or not do be,that is the question'
```

See also:

replace_regex,

---

replace_regex

Possible uses:

- **replace_regex** (**string**, **string**, **string**) —> **string**

Result:

Returns the String resulting by replacing for the first operand all the sub-strings corresponding to the regular expression given in the second operand by the third operand

Examples:

```
string var0 <- replace_regex("colour, color", "colou?r", "col"); //
    var0 equals 'col, col'
```

See also:

replace,

---

restore_simulation

Possible uses:

- **restore_simulation** (**string**) —> **int**

Result:

restore_simulation

---

restore_simulation_from_file

Possible uses:

- **restore_simulation_from_file** (ummisco.gama.serializer.gaml.GamaSavedSimulationFile) —> **int**

Result:

restoreSimulationFromFile

---

reverse

Possible uses:

- **reverse** (**container**<KeyType,ValueType>) —> **container**<**unknown**,**unknown**>
- **reverse** (**string**) —> **string**
- **reverse** (**map**<K,V>) —> **map**

Result:

the operand elements in the reversed order in a copy of the operand.

Comment:

the reverse operator behavior depends on the nature of the operand

Special cases:

- if it is a file, reverse returns a copy of the file with a reversed content
- if it is a population, reverse returns a copy of the population with elements in the reversed order
- if it is a graph, reverse returns a copy of the graph (with all edges and vertexes), with all of the edges reversed
- if it is a list, reverse returns a copy of the operand list with elements in the reversed order

```
list<int> var0 <- reverse ([10,12,14]); // var0 equals [14, 12, 10]
```

- if it is a map, reverse returns a copy of the operand map with each pair in the reversed order (i.e. all keys become values and values become keys)

```
map<int,string> var1 <- reverse (['k1'::44, 'k2'::32, 'k3'::12]); //
    var1 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> var2 <- reverse(matrix([["c11","c12","c13"],["c21","c22"
    ,"c23"]])); // var2 equals matrix([["c11","c21"],["c12","c22"],["c13
    ","c23"]])
```

- if it is a string, reverse returns a new string with characters in the reversed order

```
string var3 <- reverse ('abcd'); // var3 equals 'dcba'
```

Examples:

```
map<int,int> m <- [1::111,2::222, 3::333, 4::444];
map var5 <- reverse(m); // var5 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:

- **string rgb int** —> **rgb**
- **rgb** (**string** , **int**) —> **rgb**
- **rgb rgb float** —> **rgb**
- **rgb** (**rgb** , **float**) —> **rgb**
- **rgb rgb int** —> **rgb**
- **rgb** (**rgb** , **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 a name of color and alpha (between 0 and 255)
- It can be used with a color and an alpha between 0 and 1

- It can be used with r=red, g=green, b=blue (each between 0 and 255), a=alpha (between 0 and 255)
- 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 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)

Examples:

```
rgb var0 <- rgb ("red"); // var0 equals rgb(255,0,0)
rgb var1 <- rgb(rgb(255,0,0),0.5); // var1 equals a light red color
rgb var2 <- rgb (255,0,0,125); // var2 equals a light red color
rgb var4 <- rgb (255,0,0); // var4 equals #red
rgb var5 <- rgb(rgb(255,0,0),125); // var5 equals a light red color
rgb var6 <- rgb (255,0,0,0.5); // var6 equals a light red color
```

See also:

hsb,

---

rgb

Possible uses:

- **rgb** (**any**) —> **rgb**

---

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** (**float**) —> **float**
- **rnd** (**point**) —> **point**
- **rnd** (**int**) —> **int**
- **int rnd int** —> **int**
- **rnd** (**int** , **int**) —> **int**
- **float rnd float** —> **float**
- **rnd** (**float** , **float**) —> **float**
- **point rnd point** —> **point**
- **rnd** (**point** , **point**) —> **point**
- **rnd** (**point**, **point**, **float**) —> **point**
- **rnd** (**float**, **float**, **float**) —> **float**
- **rnd** (**int**, **int**, **int**) —> **int**

Result:

a random integer in the interval [0, operand]

Comment:

to obtain a probability between 0 and 1, use the expression (rnd n) / n, where n is used to indicate the precision

Special cases:

- if the operand is a float, returns an uniformly distributed float random number in [0.0, to]
- if the operand is a point, returns a point with three random float ordinates, each in the interval [0, ordinate of argument]

Examples:

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

See also:

flip,

---

rnd_choice

Possible uses:

- **rnd_choice** (**list**) —> **int**

Result:

returns an index of the given list with a probability following the (normalized) distribution described in the list (a form of lottery)

Examples:

```
int var0 <- rnd_choice([0.2,0.5,0.3]); // var0 equals 2/10 chances to
    return 0, 5/10 chances to return 1, 3/10 chances to return 2
```

See also:

rnd,

---

rnd_color

Possible uses:

- **rnd_color** (**int**) —> **rgb**
- **int rnd_color int** —> **rgb**
- **rnd_color** (**int** , **int**) —> **rgb**

Result:

Return a random color equivalent to rgb(rnd(first_op, last_op),rnd(first_op, last_op),rnd(first_op, last_op)) rgb color

Comment:

Return a random color equivalent to rgb(rnd(operand),rnd(operand),rnd(operand))

Examples:

```
rgb var0 <- rnd_color(100, 200); // var0 equals a random color,
   equivalent to rgb(rnd(100, 200),rnd(100, 200),rnd(100, 200))
rgb var1 <- rnd_color(255); // var1 equals a random color, equivalent
   to rgb(rnd(255),rnd(255),rnd(255))
```

See also:

rgb, hsb,

---

rotated_by

Possible uses:

- **geometry rotated_by int** —> **geometry**
- **rotated_by** (**geometry** , **int**) —> **geometry**
- **geometry rotated_by pair** —> **geometry**
- **rotated_by** (**geometry** , **pair**) —> **geometry**
- **point rotated_by pair** —> **point**
- **rotated_by** (**point** , **pair**) —> **point**
- **geometry rotated_by float** —> **geometry**
- **rotated_by** (**geometry** , **float**) —> **geometry**
- **rotated_by** (**geometry**, **float**, **point**) —> **geometry**

Result:

A geometry resulting from the application of the right-hand rotation operand (angles in 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 to the left-hand operand (geometry, agent, point) A geometry resulting from the application of a rotation by the right-hand operand angle (degree) to the left-hand operand (geometry, agent, point)

Comment:

the right-hand operand can be a float or a intreturn a vector that results from

Examples:

```
geometry var0 <- rotated_by(pyramid(10),45.0::{1,0,0}); // var0 equals
    the geometry resulting from a 45 degrees rotation along the {1,0,0}
    vector to the geometry of the agent applying the operator.
geometry var1 <- 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 <- self rotated_by 45; // var2 equals the geometry
    resulting from a 45 degrees rotation to the geometry of the agent
    applying the operator.
```

See also:

transformed_by, translated_by,

---

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** (**int**) —> **int**
- **round** (**point**) —> **point**
- **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,

---

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 with given weights takes a sample of the specified size from the elements of x using either with or without replacement

Examples:

```
list var0 <- sample([2,10,1],2,false,[0.1,0.7,0.2]); // var0 equals
    [10,2]
list var1 <- sample([2,10,1],2,false); // var1 equals [10,1]
```

Sanction

Possible uses:

- **Sanction** (**any**) —> Sanction

Result:

---

save_agent

Possible uses:

- **agent save_agent string** —> **int**
- **save_agent** (**agent** , **string**) —> **int**

---

save_simulation

Possible uses:

- **save_simulation** (**string**) —> **int**

---

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): 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,

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. transform it into a string.

---

serialize_agent

Possible uses:

- **serialize_agent** (**agent**) —> **string**

---

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
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

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,

---

shuffle

Possible uses:

- **shuffle** (**matrix**) —> **matrix**
- **shuffle** (**string**) —> **string**
- **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:

```
matrix var0 <- shuffle (matrix([["c11","c12","c13"],["c21","c22","c23"
    ]])); // var0 equals matrix([["c12","c21","c11"],["c13","c22","c23
    "]]) (for example)
string var1 <- shuffle ('abc'); // var1 equals 'bac' (for example)
list var2 <- shuffle ([12, 13, 14]); // var2 equals [14,12,13] (for
    example)
```

See also:

reverse,

---

signum

Possible uses:

- **signum** (**float**) —> **int**

Result:

Returns -1 if the argument is negative, +1 if it is positive, 0 if it is equal to zero or not a number

Examples:

```
int var0 <- signum(-12); // var0 equals -1
int var1 <- signum(14); // var1 equals 1
int var2 <- signum(0); // var2 equals 0
```

---

simple_clustering_by_distance

Possible 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

---

simplex_generator

Possible uses:

- **simplex_generator** (**float**, **float**, **float**) —> **float**

Result:

take a x, y and a bias parameters and gives a value

Examples:

```
float var0 <- simplex_generator(2,3,253); // var0 equals
    0.0976676931220678
```

---

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-Peuker 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
float var1 <- sin_rad(#pi/2); // var1 equals 1.0
```

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) with the given tolerance for the clipping and for the triangulation A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent) A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent) with the given tolerance for the clipping A list of geometries (polylines) corresponding to the skeleton of the operand geometry (geometry, agent) with the given tolerance for the clipping and for the triangulation

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.
list<geometry> var1 <- skeletonize(self); // var1 equals the list of
    geometries corresponding to the skeleton of the geometry of the
    agent applying the operator.
list<geometry> var2 <- skeletonize(self); // var2 equals the list of
    geometries corresponding to the skeleton of the geometry of the
    agent applying the operator.
list<geometry> var3 <- skeletonize(self); // var3 equals the list of
    geometries corresponding to the skeleton of the geometry of the
    agent applying the operator.
```

---

skew

Possible uses:

- **skew** (**container**) —> **float**
- **float skew float** —> **float**
- **skew** (**float** , **float**) —> **float**

Result:

Returns the skew of a data sequence when the 3rd moment has already been computed. Returns the skew of a data sequence, which is moment(data,3,mean) / standardDeviation3

Comment:

In R moment(c(1, 3, 5, 6, 9, 11, 12, 13), order=3,center=TRUE) is -10.125 and sd(c(1,3,5,6,9,11,12,13)) = 4.407785The value of the skewness tested here is different because there are different types of estimatorJoanes and Gill (1998) discuss three methods for estimating skewness:Type 1: g_1 = m_3 / m_2^(3/2). This is the typical definition used in many older textbooks.Type 2: G_1 = g_1 * sqrt(n(n-1)) / (n-2). Used in SAS and SPSS.Type 3: b_1 = m_3 / s^3 = g_1 ((n-1)/n)^(3/2). Used in MINITAB and BMDP.In R skewness(c(1, 3, 5, 6, 9, 11, 12, 13),type=3) is -0.1182316

Examples:

```
float var0 <- skew(-10.125,4.407785) with_precision(2); // var0 equals
    -0.12
float var1 <- skew([1,3,5,6,9,11,12,13]) with_precision(2); // var1
    equals -0.14
```

---

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:

gauss, truncated_gauss, poisson,

---

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

---

smooth

Possible uses:

- geometry smooth float —> geometry
- smooth (geometry , float) —> geometry

Result:

Returns a 'smoothed' geometry, where straight lines are replaces by polynomial (bicubic) curves. The first parameter is the original geometry, the second is the 'fit' parameter which can be in the range 0 (loose fit) to 1 (tightest fit).

Examples:

```
geometry var0 <- smooth(square(10), 0.0); // var0 equals a 'rounded'
    square
```

---

social_link

Possible uses:

- social_link (any) —> social_link

Result:

---

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 lef-hand operand (the graph) is nil, throws an Exception

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"::edge
    ,"vertices_specy"::node,"size"::3,"m"::5] );
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1
graph graphFromMap <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}])
    ;
point var3 <- graphFromMap source_of(link({1,5},{12,45})); // var3
    equals {1,5}
```

See also:

target_of,

---

spatial_graph

Possible 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** (**unknown**) —> species

Result:

casting of the operand to a species.

Special cases:

- if the operand is nil, returns nil;
- if the operand is an agent, returns its species;
- if the operand is a string, returns the species with this name (nil if not found);
- otherwise, returns nil

Examples:

```
species var0 <- species(self); // var0 equals the species of the
    current agent
species var1 <- species('node'); // var1 equals node
species var2 <- species([1,5,9,3]); // var2 equals nil
species var3 <- species(node1); // var3 equals node
```

---

species_of

Same signification as species

---

sphere

Possible 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*log10(elements)) bins. The splitting is strict (i.e. elements are in the ith bin if they are strictly smaller than the ith 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 according to a grid with the given number of rows and columns (geometry, nb_cols, nb_rows) A list of geometries that result from the decomposition of the geometry by square cells of the given side size (geometry, size) A list of geometries that result from the decomposition of the geometry by rectangle cells of the given dimension (geometry, {size_x, size_y})

Examples:

```
list<geometry> var0 <- to_rectangles(self, 10,20); // var0 equals the
    list of the geometries corresponding to the decomposition of the
    geometry of the agent applying the operator
list<geometry> var1 <- to_squares(self, 10.0); // var1 equals the list
    of the geometries corresponding to the decomposition of the geometry
     by squares of side size 10.0
list<geometry> var2 <- to_rectangles(self, {10.0, 15.0}); // var2
    equals the list of the geometries corresponding to the decomposition
     of the geometry by rectangles of size 10.0, 15.0
```

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 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) 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)

Examples:

```
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>> var1 <- split_in(l,3, true); // 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> 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>> var3 <- split_in(li,3); // 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<? extends java.lang.Comparable> —> list<list<unknown>>
- split_using (list<unknown> , list<? extends java.lang.Comparable>) —> list<list<unknown>>
- split_using (list<unknown>, list<? extends java.lang.Comparable>, 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 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 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

Examples:

```
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>> var1 <- split_using(l,[1.0,3.0,4.2], true); // 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> 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>> var3 <- split_using(li,[1.0,3.0,4.2]); // 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. 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).

Comment:

Delimiters themselves are excluded from the resulting list.Delimiters themselves are excluded from the resulting list.

Examples:

```
list var0 <- 'to be or not to be,that is the question' split_with ' ,';
    // var0 equals ['to','be','or','not','to','be','that','is','the','
   question']
list var1 <- 'aa::bb:cc' split_with ('::', true); // var1 equals ['aa
   ','bb:cc']
```

---

sqrt

Possible uses:

- **sqrt** (**int**) —> **float**
- **sqrt** (**float**) —> **float**

Result:

Returns the square root of the operand.

Special cases:

- if the operand is negative, an exception is raised

Examples:

```
float var0 <- sqrt(4); // var0 equals 2.0
float var1 <- sqrt(4); // var1 equals 2.0
```

---

square

Possible 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** —> **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**>

---

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:

retur 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 astring following a custom pattern and using a specific locale (e.g.: 'fr', 'en', etc.). 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 converts a date to astring 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')
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** (**container**) —> **unknown**
- **sum** (**graph**) —> **float**

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,

---

tan

Possible uses:

- **tan** (**float**) —> **float**
- **tan** (**int**) —> **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 * 180 (k an positive or negative integer). Nevertheless notice that tan(90) returns 1.633123935319537E16 (whereas we could except infinity).

Examples:

```
float var0 <- tan (0); // var0 equals 0.0
float var1 <- tan(90); // var1 equals 1.633123935319537E16
```

See also:

cos, sin,

---

tan_rad

Possible 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 lef-hand operand (the graph) is nil, returns nil

Examples:

```
graph graphEpidemio <- generate_barabasi_albert( ["edges_species"::edge
    ,"vertices_specy"::node,"size"::3,"m"::5] );
unknown var1 <- graphEpidemio source_of(edge(3)); // var1 equals node1
graph graphFromMap <-  as_edge_graph([{1,5}::{12,45},{12,45}::{34,56}])
    ;
unknown var3 <- graphFromMap target_of(link({1,5},{12,45})); // var3
    equals {12,45}
```

See also:

source_of,

---

teapot

Possible 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): 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,

---

to

Same signification as range

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 (#month) // builds an
    interval between these two dates which contains all the monthly
    dates starting from the beginning of the interval
```

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 given CRS

```
geometry var0 <- to_GAMA_CRS({121,14}, "EPSG:4326"); // 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 the current CRS, the one corresponding to the world's agent one

```
geometry var1 <- to_GAMA_CRS({121,14}); // 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(true); // var2 equals 'true'
string var3 <- to_gaml({23, 4.0}); // var3 equals '{23.0,4.0,0.0}'
string var4 <- to_gaml(5::34); // var4 equals '5::34'
string var5 <- to_gaml(rgb(255,0,125)); // var5 equals 'rgb (255, 0,
    125,255)'
string var6 <- to_gaml('hello'); // var6 equals "'hello'"
string var7 <- to_gaml([1,5,9,3]); // var7 equals '[1,5,9,3]'
string var8 <- to_gaml(['a'::345, 'b'::13, 'c'::12]); // var8 equals "
    map(['a'::345,'b'::13,'c'::12])"
string var9 <- to_gaml([[3,5,7,9],[2,4,6,8]]); // var9 equals
    '[[3,5,7,9],[2,4,6,8]]'
string var10 <- to_gaml(a_graph); // var10 equals ([((1 as node)::(3 as
     node))::(5 as edge),((0 as node)::(3 as node))::(3 as edge),((1 as
    node)::(2 as node))::(1 as edge),((0 as node)::(2 as node))::(2 as
    edge),((0 as node)::(1 as node))::(0 as edge),((2 as node)::(3 as
    node))::(4 as edge)] as map ) as graph
string var11 <- to_gaml(node1); // var11 equals  1 as node
```

---

to_rectangles

Same signification as split_geometry

Possible uses:

- **to_rectangles** (**geometry**, **point**, **bool**) —> **list**<**geometry**>
- **to_rectangles** (**geometry**, **int**, **int**, **bool**) —> **list**<**geometry**>

Result:

A list of rectangles of the size corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, dimension, overlaps), if overlaps = true, add the rectangles that overlap the border of the geometry A list of rectangles corresponding to the given dimension that result from the decomposition of the geometry into rectangles (geometry, nb_cols, nb_rows, overlaps) by a grid composed of the given number of columns and rows, if overlaps = true, add the rectangles that overlap the border of the geometry

Examples:

```
list<geometry> var0 <- to_rectangles(self, {10.0, 15.0}, true); // var0
    equals the list of rectangles of size {10.0, 15.0} corresponding to
    the discretization into rectangles of the geometry of the agent
   applying the operator. The rectangles overlapping the border of the
   geometry are kept
list<geometry> var1 <- to_rectangles(self, 5, 20, true); // var1 equals
    the list of rectangles corresponding to the discretization by a
   grid of 5 columns and 20 rows into rectangles of the geometry of the
    agent applying the operator. The rectangles overlapping the border
   of the geometry are kept
```

to_segments

Possible uses:

- to_segments (geometry) —> list<geometry>

Result:

A list of a segments resulting from the decomposition of the geometry (or its contours for polygons) into sgements

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, 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 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 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, 0.99); // var0 equals
    the list of 10 squares corresponding to the discretization into
   squares of the geometry of the agent applying the operator. The
   squares overlapping the border of the geometry are kept
list<geometry> var1 <- to_squares(self, 10, true); // 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 spliting the geometry into sub-geometries. A list of geometries resulting after spliting 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` (`unknown`) —> `topology`

Result:

casting of the operand to a topology.

Special cases:

- if the operand is a topology, returns the topology itself;
- if the operand is a spatial graph, returns the graph topology associated;
- if the operand is a population, returns the topology of the population;
- if the operand is a shape or a geometry, returns the continuous topology bounded by the geometry;

- if the operand is a matrix, returns the grid topology associated
- if the operand is another kind of container, returns the multiple topology associated to the container
- otherwise, casts the operand to a geometry and build a topology from it.

Examples:

```
topology var0 <- topology(0); // var0 equals nil
topology(a_graph)   --: Multiple topology in POLYGON
    ((24.712119771887785 7.867357373616512, 24.712119771887785
    61.283226839310565, 82.4013676510046  7.867357373616512)) at
    location[53.556743711446195;34.57529210646354]
```

See also:

geometry,

---

topology

Possible uses:

- **topology** (**any**) —> **topology**

---

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 <- polyline([{10,10},{20,20}]) touches {15,15}; // var0
    equals false
bool var1 <- polyline([{10,10},{20,20}]) touches {10,10}; // var1
    equals true
bool var2 <- {15,15} touches {15,15}; // var2 equals false
bool var3 <- polyline([{10,10},{20,20}]) touches polyline
    ([{10,10},{5,5}]); // var3 equals true
bool var4 <- polyline([{10,10},{20,20}]) touches polyline
    ([{5,5},{15,15}]); // var4 equals false
bool var5 <- polyline([{10,10},{20,20}]) touches polyline
    ([{15,15},{25,25}]); // var5 equals false
bool var6 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon
    ([{15,15},{15,25},{25,25},{25,15}]); // var6 equals false
bool var7 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon
    ([{10,20},{20,20},{20,30},{10,30}]); // var7 equals true
bool var8 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches polygon
    ([{10,10},{0,10},{0,0},{10,0}]); // var8 equals true
bool var9 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
    {15,15}; // var9 equals false
bool var10 <- polygon([{10,10},{10,20},{20,20},{20,10}]) touches
    {10,15}; // var10 equals true
```

See also:

disjoint_from, crosses, overlaps, partially_overlaps, intersects,

---

towards

Possible 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**<**unknown**>) —> **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**<**unknown**>) —> **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** (**geometry**) —> **list**<**geometry**>
- **triangulate** (**list**<**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 of the operand geometry (geometry, agent, point) 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 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

Examples:

```
list<geometry> var0 <- triangulate(self); // var0 equals the list of
    geometries (triangles) corresponding to the Delaunay triangulation
    of the geometry of the agent applying the operator.
list<geometry> var1 <- triangulate(self,0.1, 1.0); // 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([line([{0,50},{100,50}]), line
    ([{50,0},{50,100}])); // 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); // 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,0.1, 1.0); // 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:

gauss,

---

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 right-operand is a container of points, geometries or agents, returns the geometry resulting from the union all the geometries
- if the left or right operand is nil, union throws an error

Examples:

```
geometry var0 <- union([geom1, geom2, geom3]); // var0 equals a
    geometry corresponding to union between geom1, geom2 and geom3
list var1 <- [1,2,3,4,5,6] union [2,4,9]; // var1 equals
    [1,2,3,4,5,6,9]
list var2 <- [1,2,3,4,5,6] union [0,8]; // var2 equals
    [1,2,3,4,5,6,0,8]
list var3 <- [1,3,2,4,5,6,8,5,6] union [0,8]; // var3 equals
    [1,3,2,4,5,6,8,0]
```

See also:

inter, +,

---

unknown

Possible uses:

- **unknown** (**any**) —> **unknown**

---

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 equel 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
```

---

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_input

Possible uses:

- user_input (any expression) —> map<string,unknown>
- string user_input any expression —> map<string,unknown>
- user_input (string , any expression) —> map<string,unknown>

Result:

asks the user for some values (not defined as parameters). Takes a string (optional) and a map as arguments. The string is used to specify the message of the dialog

box. The map is to specify the parameters you want the user to change before the simulation starts, with the name of the parameter in string key, and the default value as value.

Comment:

This operator takes a map [string::value] as argument, displays a dialog asking the user for these values, and returns the same map with the modified values (if any). The dialog is modal and will interrupt the execution of the simulation until the user has either dismissed or accepted it. It can be used, for instance, in an init section to force the user to input new values instead of relying on the initial values of parameters :

Examples:

```
map<string,unknown> values <- user_input(["Number" :: 100, "Location"
    :: {10, 10}]);
create bug number: int(values at "Number") with: [location:: (point(
    values at "Location"))];
map<string,unknown> values2 <- user_input("Enter numer of agents and
    locations",["Number" :: 100, "Location" :: {10, 10}]);
create bug number: int(values2 at "Number") with: [location:: (point(
    values2 at "Location"))];
```

---

using

Possible uses:

- `any expression using topology` —> `unknown`
- `using` (`any expression` , `topology`) —> `unknown`

Result:

Allows to specify in which topology a spatial computation should take place.

Special cases:

- has no effect if the topology passed as a parameter is nil

Examples:

```
unknown var0 <- (agents closest_to self) using topology(world); // var0
    equals the closest agent to self (the caller) in the continuous
   topology of the world
```

---

variance

Possible uses:

- **variance** (**container**) —> **float**

Result:

the variance of the elements of the operand. See Variance for more details.

Comment:

The operator casts all the numerical element of the list into float. The elements that are not numerical are discarded.

Examples:

```
float var0 <- variance ([4.5, 3.5, 5.5, 7.0]); // var0 equals 1.671875
```

See also:

mean, median,

---

variance

Possible uses:

- **variance** (**float**) —> **float**
- **variance** (**int**, **float**, **float**) —> **float**

Result:

Returns the variance of a data sequence. That is (sumOfSquares - mean*sum) / size with mean = sum/size. Returns the variance from a standard deviation.

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.

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
```

---

variance_of

Possible uses:

- **container variance_of any expression** —> **unknown**
- **variance_of** (**container** , **any expression**) —> **unknown**

Result:

the variance of the right-hand expression evaluated on each of the elements of the left-hand operand

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

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**>

---

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
A list of geometries corresponding to the Voronoi diagram built from the list of points according to the given clip

Examples:

```
list<geometry> var0 <- voronoi
    ([{10,10},{50,50},{90,90},{10,90},{90,10}]); // var0 equals the list
     of geometries corresponding to the Voronoi Diagram built from the
    list of points.
list<geometry> var1 <- voronoi
    ([{10,10},{50,50},{90,90},{10,90},{90,10}], square(300)); // var1
    equals the list of geometries corresponding to the Voronoi Diagram
    built from the list of points with a square of 300m side size as
    clip.
```

---

weight_of

Possible 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 is the list of candidates is nil or empty

Examples:

```
int var0 <- weighted_means_DM([[1.0, 7.0],[4.0,2.0],[3.0, 3.0]], [["
    name"::"utility", "weight" :: 2.0],["name"::"price", "weight" ::
    1.0]]); // var0 equals 1
```

See also:

promethee_DM, electre_DM, evidence_theory_DM,

---

where

Possible uses:

- **container where any expression** —> **list**
- **where** (**container** , **any expression**) —> **list**

Result:

a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.

Comment:

in the right-hand operand, the keyword each can be used to represent, in turn, each of the right-hand operand elements.

Special cases:

- if the left-hand operand is a list nil, where returns a new empty list
- if the left-operand is a map, the keyword each will contain each value

```
list var4 <- [1::2, 3::4, 5::6] where (each >= 4); // var4 equals [4,
    6]
```

Examples:

```
list var0 <- [1,2,3,4,5,6,7,8] where (each > 3); // var0 equals [4, 5,
    6, 7, 8]
list var2 <- g2 where (length(g2 out_edges_of each) = 0 ); // var2
    equals [node9, node7, node10, node8, node11]
list var3 <- (list(node) where (round(node(each).location.x) > 32); //
    var3 equals [node2, node3]
```

See also:

first_with, last_with, where,

---

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
unknown var3 <- (list(node) with_max_of (round(node(each).location.x));
     // var3 equals node3
unknown var4 <- [1::2, 3::4, 5::6] with_max_of (each); // var4 equals 6
```

See also:

where, with_min_of,

---

with_min_of

Possible 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_optimizer_type

Possible uses:

- **graph with_optimizer_type string** —> **graph**
- **with_optimizer_type** (**graph** , **string**) —> **graph**

Result:

changes the shortest path computation method of the given graph

Comment:

the right-hand operand can be “Djikstra”, “Bellmann”, “Astar” to use the associated algorithm. Note that these methods are dynamic: the path is computed when needed. In contrarily, if the operand is another string, a static method will be used, i.e. all the shortest are previously computed.

Examples:

```
graphEpidemio <- graphEpidemio with_optimizer_type "static";
```

See also:

set_verbose,

---

with_precision

Possible uses:

- **float with_precision int** —> **float**
- **with_precision** (**float** , **int**) —> **float**
- **geometry with_precision int** —> **geometry**
- **with_precision** (**geometry** , **int**) —> **geometry**
- **point with_precision int** —> **point**
- **with_precision** (**point** , **int**) —> **point**

Result:

Rounds off the value of left-hand operand 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 precison. Rounds off the ordinates of the left-hand point to the precision given by the value of right-hand operand

Examples:

```
float var0 <- 12345.78943 with_precision 2; // var0 equals 12345.79
float var1 <- 123 with_precision 2; // var1 equals 123.00
geometry var2 <- self with_precision 2; // var2 equals the geometry
    resulting from the rounding of points of the geometry with a
    precision of 0.1.
point var3 <- {12345.78943, 12345.78943, 12345.78943} with_precision 2
    ; // var3 equals {12345.79, 12345.79, 12345.79}
```

See also:

round,

---

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)
```

---

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
```

---

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
```

Chapter 102

Exhaustive list of GAMA Keywords

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

Operators

-, :, ::, !, !=, ?, /, ., [](OperatorsAA#), [@](OperatorsAA#@), *, +, <, <=, =, >, >=, abs, accumulate, acos, action, add_3Dmodel, add_days, add_edge, add_geometry, add_hours, add_icon, add_minutes, add_months, add_ms, add_node, add_point, add_seconds, add_values, add_weeks, add_years, adjacency, after, agent, agent_closest_to, agent_farthest_to, agent_from_geometry, agents_at_distance, agents_inside, agents_overlapping, all_match, all_pairs_shortest_path, all_verify, alpha_index, among, and, and, angle_between, 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_map, as_matrix, as_path, 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, bool, box, brewer_colors, brewer_palettes, buffer, build, ceil, centroid, char, chi_square, chi_square_complemented, circle, clean, clean_network, 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_to_clipboard, corR, correlation, cos, cos_rad, count, covariance, covers, create_map, cross, crosses, crs, CRS_transform, csv_file, cube, curve, cylinder, date, dbscan, dead, degree_of, dem, det, determinant, diff, diff2, directed, direction_between, direction_to, 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, emotion, empty, enlarged_by, envelope, eval_gaml, eval_when, evaluate_sub_model, even, every, every_cycle, evidence_theory_DM, exp, fact, farthest_point_to, farthest_to, file, file, file_exists, first, first_of, first_with, flip, float, floor, folder, font, frequency_of, from, fuzzy_choquet_DM, fuzzy_kappa, fuzzy_kappa_sim, gaml_file, gaml_type, gamma, gamma_distribution, gamma_distribution_complemented, gamma_index, gamma_rnd, gauss, generate_barabasi_albert, generate_complete_graph, 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, gml_file, graph, 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, hsb, hypot, IDW, image_file, in, in_degree_of, in_edges_of, incomplete_beta, incomplete_gamma, incomplete_gamma_complement, indent-

ed_by, index_by, index_of, inside, int, inter, interleave, internal_at, internal_-integrated_value, 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_grid, is_image, is_json, is_number, is_obj, is_osm, is_pgm, is_property, is_R, is_saved_simulation, is_shape, is_skill, is_svg, is_text, is_threeds, is_warning, is_xml, json_file, kappa, kappa_sim, kmeans, kml, kurtosis, kurtosis, last, last_index_of, last_of, last_with, layout_circle, layout_force, layout_grid, length, lgamma, line, link, list, list_with, ln, load_graph_from_file, load_shortest_paths, load_sub_model, log, log_gamma, lower_case, main_connected_component, map, masked_by, material, material, matrix, matrix_with, max, max_flow_between, max_of, maximal_cliques_of, mean, mean_deviation, mean_of, meanR, 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, 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, open_simplex_-generator, or, or, osm_file, out_degree_of, out_edges_of, overlapping, overlaps, pair, partially_overlaps, path, path_between, path_to, paths_between, pbinom, pchisq, percent_absolute_deviation, percentile, pgamma, pgm_file, plan, plus_-days, plus_hours, plus_minutes, plus_months, plus_ms, plus_seconds, plus_weeks, plus_years, pnorm, point, points_along, points_at, 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, R_correlation, R_file, R_mean, range, rank_interpolated, read, rectangle, reduced_by, regression, remove_duplicates, remove_node_from, replace, replace_regex, restore_simulation, restore_simulation_from_file, reverse, rewire_n, rgb, rgb, rms, rnd, rnd_choice, rnd_color, rotated_by, rotation_composition, round, row_at, rows_list, sample, Sanction, save_agent, save_simulation, saved_simulation_file, 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, shuffle, signum, simple_clustering_by_distance, simple_clustering_by_envelope_distance, simplex_generator, simplification, sin, sin_rad, since, skeletonize, skew, skew_gauss, skewness, skill, smooth, 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, tan, tan_rad, tanh, target_of, teapot, text_file, TGauss, threeds_file, to, to_GAMA_CRS, to_gaml, to_rectangles, to_segments, to_squares, to_sub_geometries, to_triangles, tokenize, topology, topology, touches, towards, trace, transformed_by, translated_by, translated_to, transpose, triangle, triangulate, truncated_gauss, type_of, undirected, union, unknown, until, upper_case, use_cache, user_input, using, variance, variance, variance_of, vertical, voronoi, weight_of, weighted_means_DM, where, with_max_of, with_min_of, with_optimizer_type, with_precision, with_values, with_weights, without_holes, writable, xml_file, xor, years_between,

---

Statements

=, action, add, agents, annealing, ask, aspect, assert, benchmark, break, camera, capture, catch, chart, conscious_contagion, coping, create, data, datalist, default, diffuse, display, display_grid, display_population, do, draw, else, emotional_contagion, enforcement, enter, equation, error, event, exhaustive, exit, experiment, focus, focus_on, genetic, graphics, highlight, hill_climbing, if, image, inspect, law, layout, let, light, loop, match, migrate, monitor, norm, output, output_file, overlay, parameter, perceive, permanent, plan, put, reactive_tabu, reflex, release, remove, return, rule, rule, run, sanction, save, set, setup, simulate, socialize, solve, species, start_simulation, state, status, switch, tabu, task, test, 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, #aliceblue, #antiquewhite, #aqua, #aquamarine, #azure, #beige, #bisque, #black, #blanchedalmond, #blue, #blueviolet, #bold, #bottom_center, #bottom_left, #bottom_right, #brown, #burlywood, #cadetblue, #camera_location, #camera_orientation, #camera_target, #center, #chartreuse, #chocolate, #cl (#centiliter,#centiliters), #cm (#centimeter,#centimeters), #coral, #cornflowerblue, #cornsilk, #crimson, #current_error, #custom, #cyan, #cycle (#cycles), #darkblue, #darkcyan, #darkgoldenrod, #darkgray, #darkgreen, #darkgrey, #darkkhaki, #darkmagenta, #darkolivegreen, #darkorange, #darkorchid, #darkred, #darksalmon, #darkseagreen, #darkslateblue, #darkslategray, #darkslategrey, #darkturquoise, #darkviolet, #day (#days), #deeppink, #deepskyblue, #dimgray, #dimgrey, #display_height, #display_width, #dl (#deciliter,#deciliters), #dm (#decimeter,#decimeters), #dodgerblue, #DormandPrince54, #dp853, #e, #epoch, #Euler, #firebrick, #flat, #floralwhite, #foot (#feet,#ft), #forestgreen, #fuchsia, #gainsboro, #ghostwhite, #Gill, #gold, #goldenrod, #GraggBulirschStoer, #gram (#grams), #gray, #green, #greenyellow, #grey, #h (#hour,#hours), #HighamHall54, #hl (#hectoliter,#hectoliters), #honeydew, #horizontal, #hotpink, #inch (#inches), #indianred, #indigo, #infinity, #iso_local, #iso_offset, #iso_zoned, #italic, #ivory, #kg (#kilo,#kilogram,#kilos), #khaki, #km (#kilometer,#kilometers), #l (#liter,#liters,#dm3), #lavender, #lavenderblush, #lawngreen, #left_center, #lemonchiffon, #lightblue, #lightcoral, #lightcyan, #lightgoldenrodyellow, #lightgray, #lightgreen, #lightgrey, #lightpink, #lightsalmon, #lightseagreen, #lightskyblue, #lightslategray, #lightslategrey, #lightsteelblue, #lightyellow, #lime, #limegreen, #linen, #longton (#lton), #Luther, #m (#meter,#meters), #m2, #m3, #magenta, #maroon, #max_float, #max_int, #mediumaquamarine, #mediumblue, #mediumorchid, #mediumpurple, #mediumseagreen, #mediumslateblue, #mediumspringgreen, #mediumturquoise, #mediumvioletred, #midnightblue, #Midpoint, #mile (#miles), #min_float, #min_int, #mintcream, #minute (#minutes,#mn), #mistyrose, #mm (#milimeter,#milimeters), #moccasin, #month (#months), #msec (#millisecond,#milliseconds,#ms), #nan, #navajowhite, #navy, #nm (#nanometer,#nanometers), #none, #now, #oldlace, #olive, #olivedrab, #orange, #orangered, #orchid, #ounce (#oz,#ounces), #palegoldenrod, #palegreen, #paleturquoise, #palevioletred, #papayawhip, #peachpuff, #peru, #pi, #pink, #pixels (#px), #plain, #plum, #pound (#lb,#pounds,#lbm), #powderblue, #purple, #red, #right_center, #rk4, #rosybrown, #round, #royalblue, #saddlebrown, #salmon, #sandybrown, #seagreen, #seashell, #sec (#second,#seconds,#s),

#shortton (#ston), #sienna, #silver, #skyblue, #slateblue, #slategray, #slategrey, #snow, #split, #springgreen, #sqft (#square_foot,#square_feet), #sqin (#square_inch,#square_inches), #sqmi (#square_mile,#square_miles), #square, #stack, #steelblue, #stone (#st), #tan, #teal, #thistle, #ThreeEighthes, #to_deg, #to_rad, #tomato, #ton (#tons), #top_center, #top_left, #top_right, #transparent, #turquoise, #user_location, #vertical, #violet, #week (#weeks), #wheat, #white, #whitesmoke, #yard (#yards), #year (#years,#y), #yellow, #yellowgreen, #zoom,

---

Skills

advanced_driving, driving, fipa, MDXSKILL, messaging, moving, moving3D, network, physics, skill_road, skill_road_node, SQLSKILL,

---

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, compute_forces, advanced_follow_driving, is_ready_next_road, test_next_road, compute_path, path_from_nodes, drive_random, drive, external_factor_impact, speed_choice, lane_choice, follow_driving, goto_driving, start_conversation, send, reply, accept_proposal, agree, cancel, cfp, end_conversation,

failure, inform, propose, query, refuse, reject_proposal, request, subscribe, timeStamp, testConnection, select, send, wander, move, follow, goto, move, execute, connect, fetch_message, has_more_message, join_group, leave_group, simulate_step, register, unregister, timeStamp, getCurrentDateTime, getDateOffset, testConnection, executeUpdate, insert, select, list2Matrix,

---

Variables

speed, real_speed, current_path, final_target, current_target, current_index, targets, security_distance_coeff, safety_distance_coeff, min_security_distance, min_safety_distance, current_lane, vehicle_length, speed_coeff, max_acceleration, current_road, on_linked_road, proba_lane_change_up, proba_lane_change_down, proba_respect_priorities, proba_respect_stops, proba_block_node, proba_use_linked_road, right_side_driving, max_speed, distance_to_goal, segment_index_on_road, living_space, lanes_attribute, tolerance, obstacle_species, speed, 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, mass, friction, restitution, lin_damping, ang_damping, velocity, collisionBound, agents_on, all_agents, source_node, target_node, lanes, linked_road, maxspeed, roads_in, priority_roads, roads_out, stop, block,

---

Pseudo-Variables

self, myself, each

Types

action, agent, attributes, BDIPlan, bool, container, conversation, date, emotion, file, float, font, gaml_type, geometry, graph, int, kml, list, map, material, 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

Part IX

Tutorials

Chapter 103

Tutorials

We propose some tutorials that are designed to allow modelers to become progressively autonomous with the GAMA platform. These tutorials cover different aspects of GAMA (Grid environment, GIS integration, 3D, multi-level modeling, equation-based models...). It is a good idea to keep a copy of the reference of the GAML language around when undertaking one of these tutorials.

Predator Prey tutorial

This tutorial introduces the basic concepts of GAMA and the use of grids. It is based on the classic predator prey model (see for instance a formal definition here). It is particularly adapted to beginners that want to quickly learn how to build a simple model in GAMA.

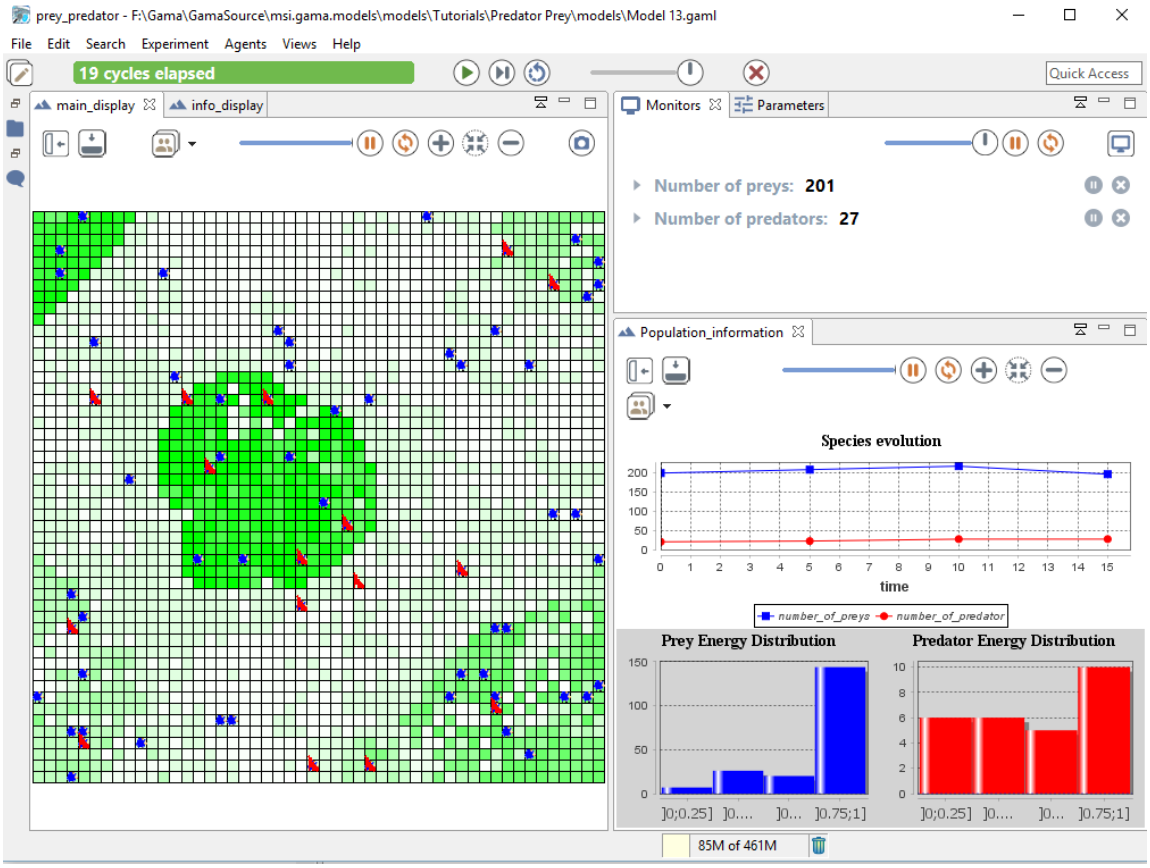


Figure 103.1: resources/images/tutorials/predator_prey.png

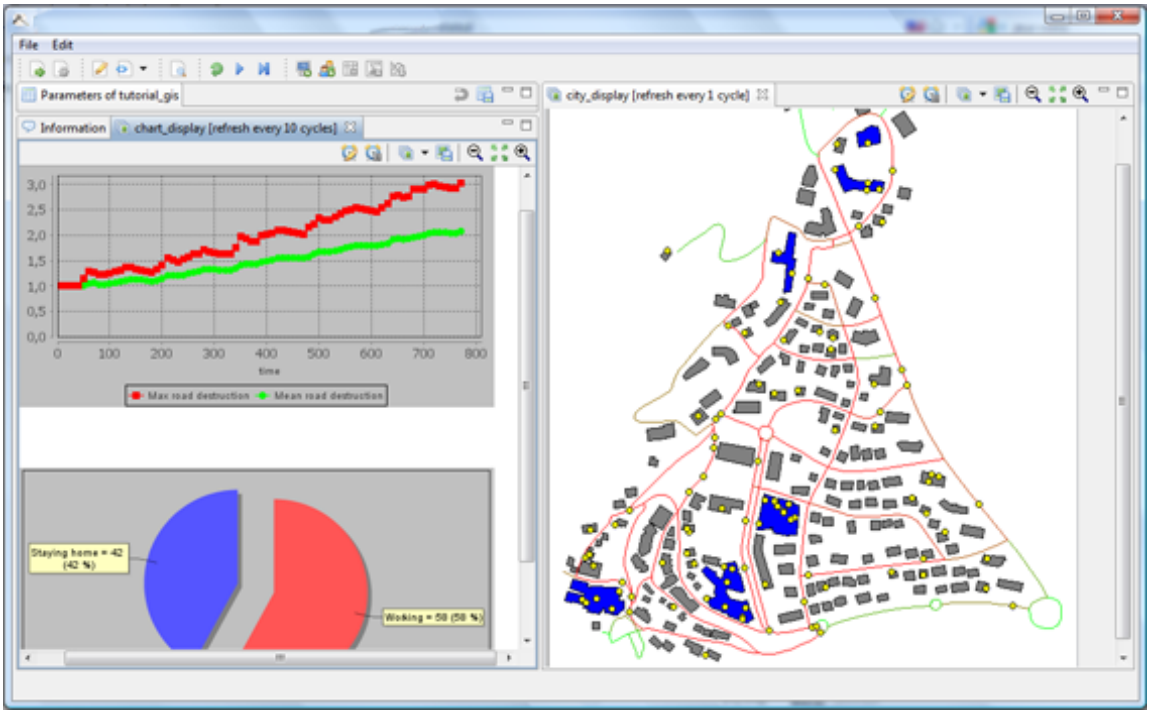


Figure 103.2: resources/images/tutorials/road_traffic.png

Figure 103.3: resources/images/tutorials/Luneray.jpg

Road Traffic

This tutorial introduces the use of GIS data. It is based on a mobility and daily activity model. It is particularly adapted to modelers that want to quickly learn how to integrate GIS data in their model and to use a road shapefile for the movement of their agents.

3D Tutorial

This tutorial introduces the use of 3D in GAMA. In particular, it offers a quick overview of the 3D capabilities of the platform and how to integrate 3D features in models.

Luneray's flu tutorial

This tutorial dedicated to beginners introduces the basic concepts of GAMA and proposes a brief overview of many features. It concerns a model of disease spreading in the small city of Luneray. In particular, it presents how to integrate GIS data and use GIS, to use a road shapefile for the movement of agents, and to define a 3D display.

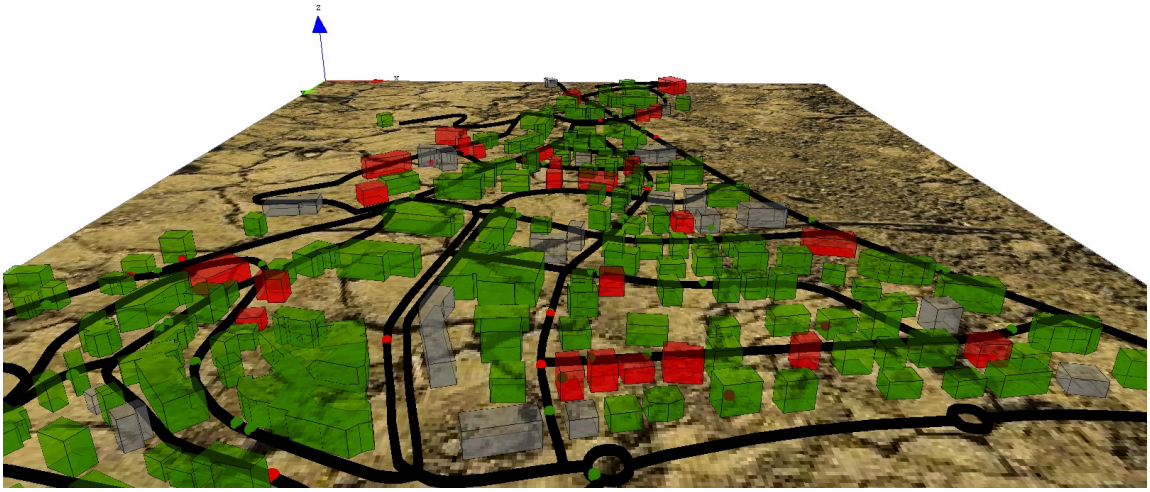


Figure 103.4: resources/images/tutorials/incremental_model.jpg

Incremental Model

This tutorial proposes is an advance version of the Luneray's tutorial. It concerns a model of disease spreading in a small city. In particular, it presents how to integrate GIS data and use GIS, to use a road shapefile for the movement of agents, to define a 3D display, to define a multi-level model and use differential equations.

BDI Architecture

This tutorial introduces the use of the BDI architecture (named BEN provided with the GAMA platform. It is particularly adapted for advanced users who want to integrate reasoning capabilities in theirs agents, taking into account their emotions and social relationships.

Chapter 104

Predator Prey

This tutorial presents the structure of a GAMA model as well as the use of a grid topology. In particular, this tutorial shows how to define a basic model, to define "grid agents" which are able to move within the constraints. It also introduce the displays and agents' aspect.

All the files related to this tutorial (images and models) are available in the Models Library (project Tutorials/Predator Prey).

Content

Model Overview

In this model, three types of entities are considered: preys, predators and vegetation cells. Preys eat grass on the vegetation cells and predators eat preys. At each simulation step, grass grows on the vegetation cells. Concerning the predators and preys, at each simulation step, they move (to a neighbor cell), eat, die if they do not have enough energy, and eventually reproduce.

Step List

This tutorial is composed of 12 incremental steps corresponding to 12 models. For each step we present its purpose, an explicit formulation and the corresponding

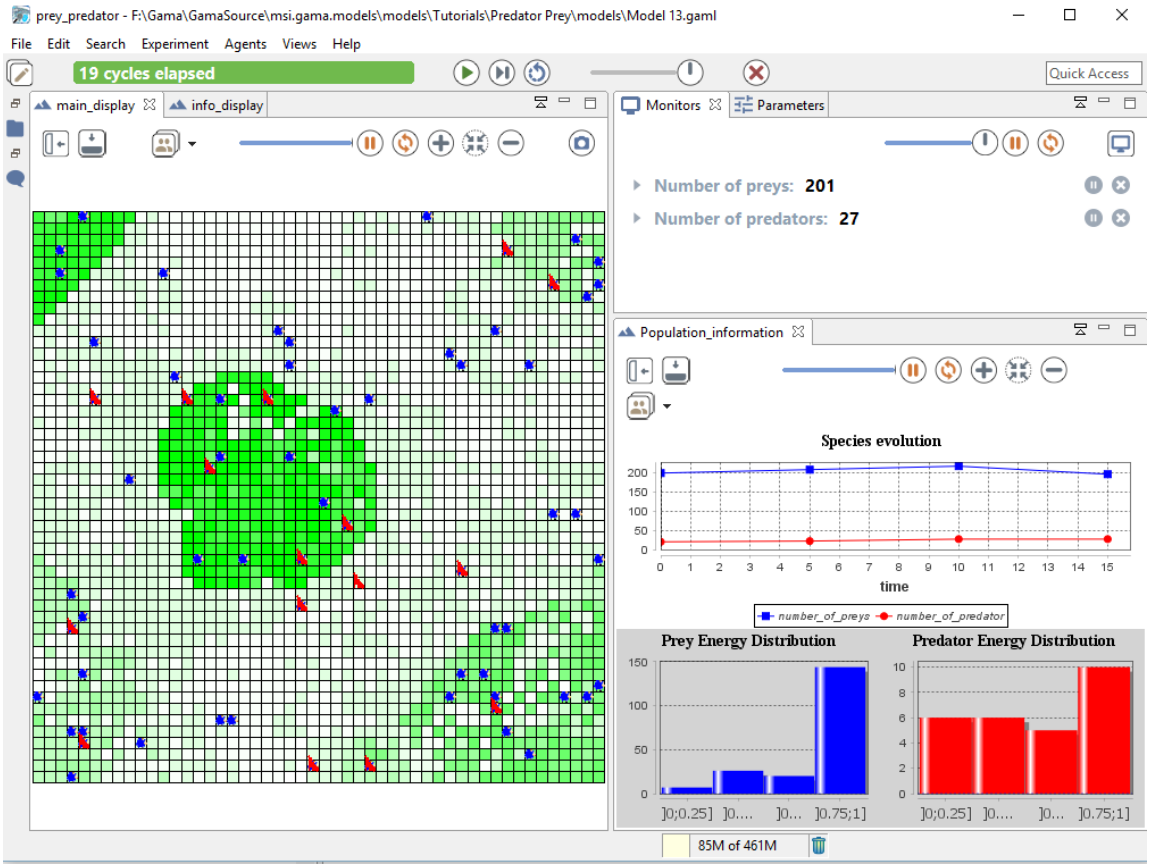


Figure 104.1: images/predator_prey.png

GAML code of the model.

1. Basic model (prey agents)
2. Dynamic of the vegetation (grid)
3. Behavior of the prey agent
4. Use of Inspectors/monitors
5. predator agents (parent species)
6. Breeding of prey and predator agents
7. Agent display (aspect)
8. Complex behaviors for the preys and predators
9. Adding of a stopping condition
10. Definition of charts
11. Writing files
12. Image loading (raster data)

Chapter 105

1. Basic Model

Content

This first step Illustrates how to write a model in GAMA. In particular, it describes how to structure a model and how to define species - that are the key components of GAMA models.

Formulation

- Definition of the **prey** species
- Definition of a **nb_prey_init** parameter
- Creation of **nb_prey_init prey** agents randomly located in the environment (size: 100x100)

Model Definition

Model structure

A GAMA model is composed of three types of section:

- **global** : this section, that is unique, defines the "world" agent, a special agent of a GAMA model. It represents all that is global to the model: dynamics, variables, actions. In addition, it allows to initialize the simulation (init block).
- **species** and **grid**: these sections define the species of agents composing the model. Grid is defined in the following model step "vegetation dynamic";
- **experiment** : these sections define a context of execution of the simulations. In particular, it defines the input (parameters) and output (displays, files. . .) of a model.

More details about the different sections of a GAMA model can be found here.

Species

A species represents a «prototype» of agents: it defines their common properties.

A species definition requires the definition of three different elements : * the internal state of its agents (attributes) * their behavior * how they are displayed (aspects)

Internal state

An attribute is defined as follows: type of the attribute and name. Numerous types of attributes are available: *int (integer), float (floating point number), string, bool (boolean, true or false), point (coordinates), list, pair, map, file, matrix, species of agents, rgb (color), graph, path. . .*

- Optional facets: <- (initial value), update (value recomputed at each step of the simulation), function:{..} (value computed each time the variable is used), min, max

In addition to the attributes the modeler explicitly defines, species "inherits" other attributes called "built-in" variables:

- A name (*name*): the identifier of the species
- A shape (*shape*): the default shape of the agents to be constructed after the species. It can be *a point, a polygon, etc.*
- A location (*location*): the centroid of its shape.

Behavior

In this first model, we define one species of agents: the **prey** agents. For the moment, these agents will not have a particular behavior, they will just exist and be displayed.

Display

An agent aspects have to be defined. An aspect is a way to display the agents of a species: aspect aspect_name {...} In the block of an aspect, it is possible to draw:

- A geometry: for instance, the shape of the agent (but it may be a different one, for instance a disk instead of a complex polygon)
- An image: to draw icons
- A text: to draw a text

In order to display our prey agents we define two attributes:

- **size** of type float, with for initial value: 1.0
- **color** of type *rgb*, with for initial value: "blue". It is possible to get a color value by using the symbol # + color name: e.g. #blue, #red, #white, #yellow, #magenta, #pink...

Prey species code

For the moment, we only define an aspect for this species. We want to display for each prey agent a circle of radius *size* and color *color*. We then use the keyword **draw** with a circle shape.

```
species prey {
    float size <- 1.0 ;
    rgb color <- #blue;

    aspect base {
        draw circle(size) color: color ;
    }
}
```

global section

The global section represents a specific agent, called world. Defining this agent follows the same principle as any agent and is, thus, defined after a species. The world agent represents everything that is global to the model: dynamics, variables... It allows to initialize simulations (init block): the world is always created and initialized first when a simulation is launched (before any other agents). The geometry (shape) of the world agent is by default a square with 100m for side size, but can be redefined if necessary (see the Road traffic tutorial).

global variable

In the current model, we will only have a certain number of preys thus we need to hold this number in a global or world's variable of type integer (*int*) which can be done as follows:

```
global {
   int nb_preys_init <- 200;
}
```

Model initialization

The init section of the global block allows to initialize the model which is executing certain commands, here we will create *nb_preys_init* number of prey agents. We use the statement *create* to create agents of a specific species: **create** species_name + :

- number: number of agents to create (int, 1 by default)
- from: GIS file to use to create the agents (optional, string or file)
- returns: list of created agents (list)

Definition of the init block in order to create *nb_preys_init prey* agents:

```
   init {
      create prey number: nb_preys_init ;
   }
```

experiment

An experiment block defines how a model can be simulated (executed). Several experiments can be defined for a given model. They are defined using : **experiment** exp_name type: gui/batch {`[input] [output]`}

- gui: experiment with a graphical interface, which displays its input parameters and outputs.
- batch: Allows to setup a series of simulations (w/o graphical interface).

In our model, we define a gui experiment called *prey_predator* :

```
experiment prey_predator type: gui {
}
```

input

Experiments can define (input) parameters. A parameter definition allows to make the value of a global variable definable by the user through the graphic interface.

A parameter is defined as follows:

- `parameter` title var: global_var category: cat;
- **`title`**: string to display
- **`var`**: reference to a global variable (defined in the global section)
- **`category`**: string used to «store» the operators on the UI - optional
- <-: init value - optional
- **`min`**: min value - optional
- **`max`**: min value - optional

Note that the init, min and max values can be defined in the global variable definition.

In the experiment, definition of a parameter from the global variable *nb_preys_init*:

```
experiment prey_predator type: gui {
   parameter "Initial number of preys: " var: nb_preys_init min: 1 max:
     1000 category: "Prey" ;
}
```

output

Output blocks are defined in an experiment and define how to visualize a simulation (with one or more display blocks that define separate windows). Each display can be refreshed independently by defining the facet **refresh_every:** nb (int) (the display will be refreshed every nb steps of the simulation).

Each display can include different layers (like in a GIS):

- Agents lists : **agents** layer_name value: agents_list aspect: my_aspect;
- Agents species : **species** my_species aspect: my_aspect
- Images: **image** layer_name file: image_file;
- Texts : **texte** layer_name value: my_text;
- Charts : see later.

Note that it is possible to define a opengl display (for 3D display) by using the facet **type: opengl**.

In our model, we define a display to draw the *prey* agents.

```
output {
     display main_display {
        species prey aspect: base ;
     }
  }
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    init {
        create prey number: nb_preys_init ;
    }
}

species prey {
    float size <- 1.0 ;
    rgb color <- #blue;
```

```
    aspect base {
        draw circle(size) color: color ;
    }
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init min: 1 max
    : 1000 category: "Prey" ;
    output {
        display main_display {
            species prey aspect: base ;
        }
    }
}
```

Chapter 106

2. Vegetation Dynamic

This second step presents the idea of environment or topological space. Defining a "vegetation" environment allows to define the movement of the preys through dynamic variables (use of the *update* facet). We will also discover more about displays.

Formulation

- Definition of a grid (for the vegetation)
- Definition of a dynamic for each cell (food production)
- Display of the cell color according to the quantity of food
- Localization of the prey agents on the cells (at its center)

Model Definition

grid

In GAMA, grids are specific agent species with a particular topology. First, a grid allows yet constrains the movement of other (moving) agents but they can have variables and behaviors.

A grid is defined as follows:

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

```
}
```

With:

- width : number of cells along x-axis
- height : number of cells along y-axis
- neighbors : neighborhood type (4 - Von Neumann, 6 - hexagon or 8 - Moore)

In our model, we define a grid species, called **vegetation_cell** composed of 50x50 cells and with a Von Neumann neighborhood. In order for each grid agents (or cell of the grid) to represent the vegetation, we provide them with four variables:

- *maxFood* : maximum food that a cell can contain -> type: *float* ; init value: 1.0
- *foodProd* : food produced at each simulation step -> type: *float* ; init value: random number between 0 and 0.01
- *food* : current quantity of food -> type: *float* ; init value: random number between 0 and 1.0; at each simulation step : food <- food + foodProd
- *color* : color of the cell -> type: *rgb* ; init value: color computed according to the food value: more the food value is close to 1.0, greener the color is, more the food value is close to 0, whiter the color is; update : computation of the new color depending on the current level of food (at each simulation step).

The **update** facet allows to give a behavior to the agents. Indeed, at each simulation step, each agent is activated (by default, in a random order) and first applies for each dynamic variable (in their definition order) its update expression. In the present case, it allows us to link the displayed color of the cell to its food level.

```
grid vegetation_cell width: 50 height: 50 neighbors: 4 {
   float maxFood <- 1.0 ;
   float foodProd <- (rnd(1000) / 1000) * 0.01 ;
   float food <- (rnd(1000) / 1000) update: food + foodProd max:
maxFood;
   rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)
)) update: rgb(int(255 * (1 - food)), 255, int(255 * (1 - food))) ;
}
```

There are several ways to define colors in GAML:

- the simplest way consists in using the symbol # + the color name (for a limited set of colors):

```
#blue
#red
```

- Another way consists in defining the 3 rgb integer values: rgb(red, green, blue) with red, green and blue between 0 and 255 (as we used in the current model.

```
rgb(0,0,0) : black ; rgb(255,255,255) : white
rgb(255,0,0) : red ;  rgb(0,255,0) : green ;  rgb(0,0,255) : blue
```

prey agents

In order to relate our prey agents to the vegetation cell grid, we add them with one new variable : **my_cell** of type vegetation_cell and for init value one of the vegetation_cell (chosen randomly).

```
species prey {
    ...
    vegetation_cell myCell <- one_of (vegetation_cell) ;
}
```

It is possible to obtain the list of all agents of a given species by using the name of the species while **one_of** to pick one element randomly from this list.

We linked each prey agent to a vegetation_cell but we need to locate them onto the cell. To do so, we set the prey location as equals to the location of the vegetation cell (i.e. its centroid **location**), we use in the init block the **<-** statement that allows to modify the value of a variable :

```
species prey {
    ...
    init {
        location <- myCell.location;
    }
}
```

display

In order to visualize the vegetation, we need to add it to the display. We use for that the statement **grid** with the optional facet **lines** to draw the border of the cells. Note that grid agents have built-in aspect thus it is not necessary to define one.

```
   output {
      display main_display {
         grid vegetation_cell lines: #black;
         species prey aspect: base ;
      }
   }
```

Note that the layers in a display work like layers in a GIS; the drawing order will be respected. In our model, the prey agents will be drawn above the vegetation_cell grid thus they need to be declared afterward.

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    init {
        create prey number: nb_preys_init ;
    }
}

species prey {
    float size <- 1.0 ;
    rgb color <- #blue;
    vegetation_cell myCell <- one_of (vegetation_cell) ;

    init {
        location <- myCell.location;
    }

    aspect base {
        draw circle(size) color: color ;
    }
}
```

```
grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init min: 1 max
   : 1000 category: "Prey" ;
    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: base ;
        }
    }
}
```

Chapter 107

3. Prey Agent Behavior

This third step Illustrates how to define the behaviors of prey agents and the concept of spatial topology.

Formulation

- Random movement of the prey agents to a distance of 2 cells (Von Neumann neighborhood)
- At each step, the prey agents loss energy
- At each step, the prey agents eat food if there is food on the cell on which they are localized (with a max of max_transfer) and gain energy
- If a prey agent has no more energy, it dies

Model Definition

Parameters

To define a behavior for the prey agents we add them three new parameters:

- The max energy of the prey agents
- The maximum energy that can a prey agent consume from vegetation per tick
- The energy used by a prey agent at each time step

As we consider these parameters to be global to all prey, we define them in the global section as follows:

```
float prey_max_energy <- 1.0;
float prey_max_transfer <- 0.1;
float prey_energy_consum <- 0.05;
```

Yet we may allow the user to change it from an experiment to another through the user interface. To do so we add the following definition of parameters within the experiment section :

```
parameter "Prey max energy: " var: prey_max_energy category: "Prey"
;
parameter "Prey max transfer: " var: prey_max_transfer  category: "
Prey" ;
parameter "Prey energy consumption: " var: prey_energy_consum
category: "Prey" ;
```

vegetation_cell grid

We add a new variable for the vegetation_cell grid called **neighbors**, that contains for each vegetation cell the list of the neighbor vegetation cells (distance of 2 - Von Neumann neighborhood). We will use these neighbors list for the movement of the prey.

```
grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    ...
    list<vegetation_cell> neighbors <- self neighbors_at 2;
}
```

Note that the result of the operator **neighbors_at dist** depends on the type of topology of the agent applying this operator:

- For a grid topology (grid species), the operator returns the neighbor cells (with a Von Neumann or Moore neighborhood).
- For a continuous topology, the operator returns the list of agents of which the shape is located at a distance equals or inferior *dist* meters to the agent shape.

Also, note the use of the self pseudo variable which is a reference to the agent currently executing the statement

Prey agents

We copy the values of the three global parameters into the prey species in order for it to be available for each agent and possibly modified locally.

```
species prey {
   ...
   float max_energy <- prey_max_energy ;
   float max_transfer <- prey_max_transfer ;
   float energy_consum <- prey_energy_consum ;
   ...
}
```

The energy used by each prey at each timestep is randomly computed initially (within]0;max_energy]).

```
species prey {
   ...
   float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;
   ...
}
```

In order to define the movement behaviour of a prey we will add a `reflex`. A reflex is a block of statements (that can be defined in global or any species) that will be automatically executed at each simulation step if its condition is true, it is defined as follows:

```
   reflex reflex_name when: condition {...}
```

The **when:** facet is optional: when it is omitted, the reflex is activated at each time step. Note that if several reflexes are defined for a species, the reflexes will be activated following their definition order.

We define a first reflex called **basic_move** that allows the prey agents to choose (randomly) a new vegetation_cell in the neighborhood of my_cell and to move to this cell.

```
species prey {
   ...
   reflex basic_move {
       myCell <- one_of (myCell.neighbors) ;
       location <- myCell.location ;
   }
```

```
}
```

We define a second reflex called **eat** that will only be activated when there is food in my_cell and that will allow the prey agents to eat food and gain energy. In order to store the energy gain by the eating (that is equals to the minimum between the **max_transfer** value and the quantity of food available in **myCell**), we define a local variable called **energy_transfer**. A local variable is a variable that will only exist within this block: once it has been executed, the variable is forgotten. To define it, we have to use the following statement:

```
var_type var_name <- value;
```

Thus, the reflex **eat** is defined by:

```
species prey {
   ...
   reflex eat when: myCell.food > 0 {
      float energy_transfer <- min([max_transfer, myCell.food]) ;
      myCell.food <- myCell.food - energy_transfer ;
      energy <- energy + energy_transfer ;
   }
}
```

We define a third reflex for the prey agent: when the agent has no more energy, it dies (application of the built-in **die** action):

```
species prey {
   ...
   reflex die when: energy <= 0 {
      do die ;
   }
}
```

Note that an action is a capability available to the agents of a species (what they can do). It is a block of statements that can be used and reused whenever needed. Some actions, called primitives, are directly coded in Java: for instance, the **die** action defined for all the agents.

- An action can accept arguments. For instance, write takes an argument called message.
- An action can return a result.

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

- for actions that do not return a result:

```
do action_name arg1: v1 arg2: v2;
```

- for actions that return a result:

```
my_var <- self action_name (arg1:v1, arg2:v2);
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    float prey_max_energy <- 1.0;
    float prey_max_transfer <- 0.1;
    float prey_energy_consum <- 0.05;

    init {
        create prey number: nb_preys_init ;
    }
}

species prey {
    float size <- 1.0 ;
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfer <- prey_max_transfer ;
    float energy_consum <- prey_energy_consum ;

    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }
```

```
    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }
    reflex eat when: myCell.food > 0 {
        float energy_transfer <- min([max_transfer, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfer ;
        energy <- energy + energy_transfer ;
    }
    reflex die when: energy <= 0 {
        do die ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init min: 1 max
   : 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfer: " var: prey_max_transfer  category: "
   Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: base ;
        }
    }
}
```

Chapter 108

4. Inspectors and Monitors

This fourth step Illustrates how to monitor more precisely the simulation. Practically, we will define monitors to follow the evolution of specific variables (or expression) whereas inspector allows to follow the state of a given agent (or a species).

Formulation

- Adding of a monitor to follow the evolution of the number of prey agents

Model Definition

global variable

We add a new global variable:

- **nb_preys** : returns, each time it is called, the current number of (live) prey agents

To do so we use the **->{expression}** facet which returns the value of **expression**, each time it is called.. We use as well the operator **length** that returns the number of elements in a list.

Thus, In the global section, we add the **nb_preys** global variable:

```
int nb_preys -> {length (prey)};
```

monitor

A monitor allows to follow the value of an arbitrary expression in GAML. It has to be defined in an output section. A monitor is defined as follows:

```
    monitor monitor_name value: an_expression refresh:every(nb_steps)
;
```

With:

- value: mandatory, its that value will be displayed in the monitor.
- refresh: bool, optional: if the expression is true, compute (default is true).

In this model, we define a monitor to follow the value of the variable **nb_preys**:

```
    monitor "number of preys" value: nb_preys;
```

inspector

Inspectors allow to obtain information about a species or an agent. There are two kinds of agent information features:

- Species browser: provides informations about all the agents of a species. Available in the Agents menu.

- Agent inspector: provides information about one specific agent. Also allows to change the values of its variables during the simulation. Available from the Agents menu, by right_clicking on a display, in the species inspector or when inspecting another agent. It provides also the possibility to «highlight» the inspected agent.

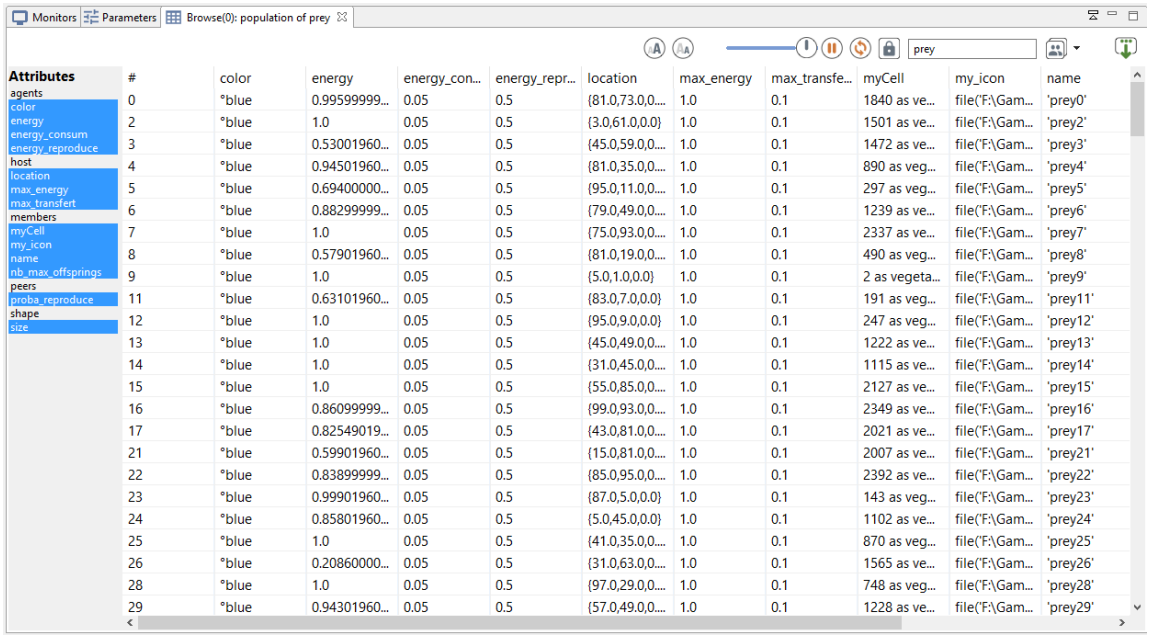


Figure 108.1: images/browser_table.png

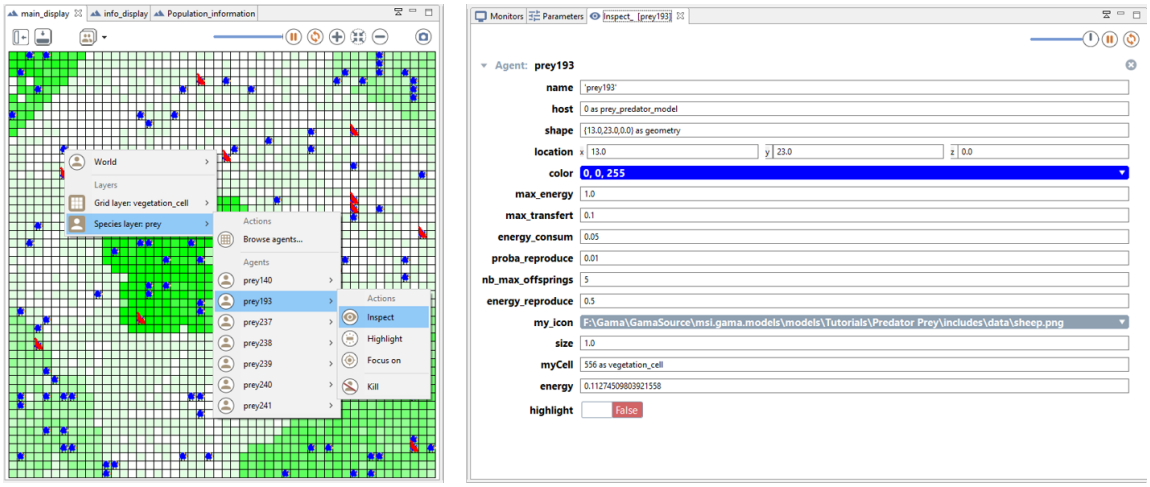


Figure 108.2: images/inspector.png

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200 ;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1;
    float prey_energy_consum <- 0.05;
    int nb_preys -> {length (prey)};

    init {
        create prey number: nb_preys_init ;
    }
}
species prey {
    float size <- 1.0 ;
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;

    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }
    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }
    reflex die when: energy <= 0 {
        do die ;
    }

    aspect base {
        draw circle(size) color: color ;
```

```
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init min: 1 max
   : 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: base ;
        }
        monitor "Number of preys" value: nb_preys ;
    }
}
```

Chapter 109

5. Predator Agent

This fifth step Illustrates how to use parent species. Indeed,prey and predators share a few common features thus we will define a generic species that will regroup all the common elements (variables, behaviors and aspects) between the prey and the predator species.

Formulation

- Definition of a new generic species: generic_species
- Definition of a new species: predator
- predator agents move randomly
- At each simulation step, a predator agent can eat a prey that is localized at its grid cell

Model Definition

parameters

We add four new parameters related to predator agents:

- The init number of predator agents
- The max energy of the predator agents

- The energy gained by a predator agent while eating a prey agent
- The energy consumed by a predator agent at each time step

We define four new global variables in the global section:

```
global {
    ...
    int nb_predators_init <- 20;
    float predator_max_energy <- 1.0;
    float predator_energy_transfer <- 0.5;
    float predator_energy_consum <- 0.02;
}
```

We define then the four corresponding parameters in the experiment:

```
    parameter "Initial number of predators: " var: nb_predators_init min
    : 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category:
      "Predator" ;
    parameter "Predator energy transfer: " var: predator_energy_transfer
       category: "Predator" ;
    parameter "Predator energy consumption: " var:
     predator_energy_consum  category: "Predator" ;
```

parent species

A species can have a parent species: it automatically get all the variables, skill and actions/reflex of the parent species

We define a species called **generic_species** that is the parent of the species **prey** and **predator**:

This species integrates all of the common feature between the **prey** and **predotor** species:

- the variables:
  - size
  - color
  - max_energy
  - max_transfer
  - energy_consum

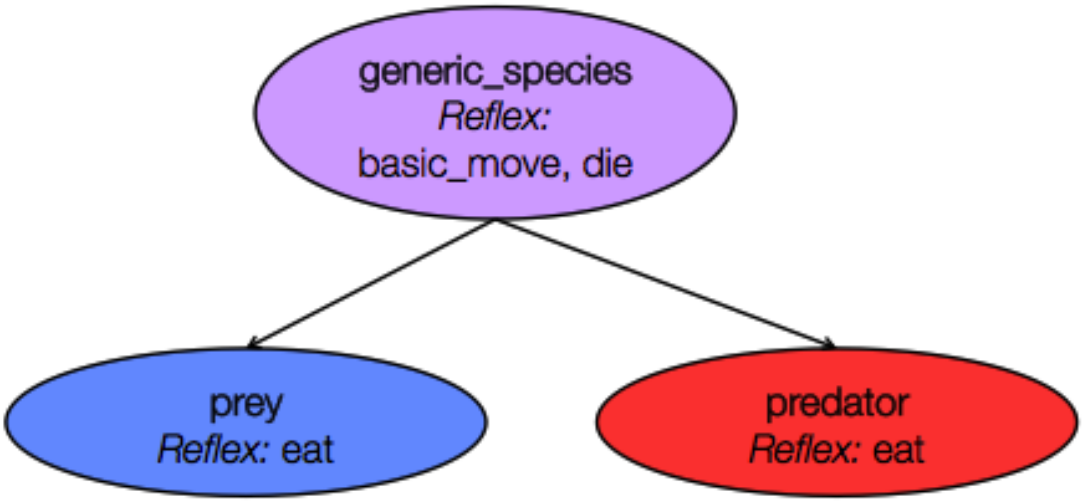


Figure 109.1: images/parent_species.png

  - myCell
  - energy

- the behaviors:
  - basic_move reflex
  - die reflex

- the aspect:
  - base aspect

```
species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfer;
    float energy_consum;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }
```

```
    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
}
```

prey species

We specialize the **prey** species from the **generic_species** species:

- definition of the initial value of the agent variables
- definition of the eat reflex

```
species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfer <- prey_max_transfer ;
    float energy_consum <- prey_energy_consum ;

    reflex eat when: myCell.food > 0 {
        float energy_transfer <- min([max_transfer, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfer ;
        energy <- energy + energy_transfer ;
    }
}
```

predator species

As done for the **prey** species, we specialize the **predator** species from the **generic_species** species:

- definition of the initial value of the agent variables
- definition of a new variable **reachable_preys** consisting in the list of prey agents contains in myCell; compute at each simulation step (when the agent is activated).
- definition of the eat reflex: activated when **reachable_preys** is not empty; choose one of the element of this list, ask it to die; and update the **predator** energy.

```
species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfer <- predator_energy_transfer ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfer ;
    }
}
```

Note that we used the **ask** statement. This statement allows to make a remote agent executes a list of statements. Use of the ask statement as follows:

```
ask one_agent or ask agents_list
```

We used as well the **species/agent list inside geometry/agent** operator. This operator returns all the agents of the specified species (or from the specified agent list) that are inside the given geometry or agent geometry.

global init block

Like in the previous model, in the init block of the model, we create nb_predators_init.

```
global {
   ...
   init {
      create prey number: nb_preys_init ;
      create predator number: nb_predators_init ;
   }
```

```
}
```

monitor

Like in the previous model, we define a monitor to display the number of predator agents.

Definition of a global variable **nb_predator** that returns the current number of **predator** agents:

```
global {
    ...
    int nb_predators -> {length (predator)};
    ...
}
```

Definition of the corresponding monitor:

```
monitor "number of predators" value: nb_predators ;
```

display

Also, do not forget to add the line to display predators in your simulation

```
display main_display {
    ...
    species predator aspect: icon ;
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfer <- 0.1 ;
```

```
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfer <- 0.5;
    float predator_energy_consum <- 0.02;
    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfer;
    float energy_consum;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfer <- prey_max_transfer ;
    float energy_consum <- prey_energy_consum ;
```

```
    reflex eat when: myCell.food > 0 {
        float energy_transfer <- min([max_transfer, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfer ;
        energy <- energy + energy_transfer ;
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfer <- predator_energy_transfer ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfer ;
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food)));
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfer: " var: prey_max_transfer  category: "
   Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
   : "Predator" ;
```

```
    parameter "Predator energy transfer: " var:
   predator_energy_transfer  category: "Predator" ;
    parameter "Predator energy consumption: " var:
   predator_energy_consum  category: "Predator" ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: base ;
            species predator aspect: base ;
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 110

6. Breeding

So far we created agents only during the initialisation of the simulation. In this sixth step, we Illustrate how to create new agents during a simulation of a dynamic species.

Formulation

- Adding of a reproduce action of the prey and predator agents:
  - When a agent has energy enough, it has a certain probability to have a certain number of offspring
  - The energy of the offspring is equal to the parent energy divided by the number of offspring
  - The parent get the same energy as its offspring

Model Definition

parameters

We add six new parameters related to breeding:

- The reproduction probability for prey agents
- The max number of offspring for prey agents
- The minimum energy to reproduce for prey agents

- The reproduction probability for predator agents
- The max number of offspring for predator agents
- The minimum energy to reproduce for predator agents

We define six new global variables in the global section:

```
global {
    ...
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;
}
```

We define then the six corresponding parameters in the experiment:

```
    parameter "Prey probability reproduce: " var: prey_proba_reproduce
     category: "Prey" ;
    parameter "Prey nb max offsprings: " var: prey_nb_max_offsprings
     category: "Prey" ;
    parameter "Prey energy reproduce: " var: prey_energy_reproduce
     category: "Prey" ;
    parameter "Predator probability reproduce: " var:
     predator_proba_reproduce category: "Predator" ;
    parameter "Predator nb max offsprings: " var:
     predator_nb_max_offsprings category: "Predator" ;
    parameter "Predator energy reproduce: " var:
     predator_energy_reproduce category: "Predator" ;
```

parent species

We add three new variables for the **generic_species**:

- proba_reproduce
- nb_max_offsprings
- energy_reproduce

We add as well a new reflex called **reproduce**:

- this reflex is activated only when:
  - the energy of the agent is greater or equals to energy_reproduce
  - AND according to the probability proba_reproduce: for this second condition, we use the **flip(proba)** operator that returns *true* according to the probability proba (*false* otherwise).
- this reflex creates **nb_offsprings** (random number between 1 and nb_max_offsprings) new agent of species the species of the agent using the create statement: we use a species casting operator on the current agent.
  - the created agents are initialized as follows:
    - myCell: myCell of the agent creating the agents
    - location: location of myCell
    - energy: energy of the agent creating the agents (use of keyword **myself**) divided by the number of offsprings.
- after the agent creation, the reflex updates the energy value of the current agent with the value: energy / nb_offsprings

```
species generic_species {
   ...
   float proba_reproduce ;
   int nb_max_offsprings;
   float energy_reproduce;
   ...
   reflex reproduce when: (energy >= energy_reproduce) and (flip(
proba_reproduce)) {
      int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
      create species(self) number: nb_offsprings {
         myCell <- myself.myCell ;
         location <- myCell.location ;
         energy <- myself.energy / nb_offsprings ;
      }
      energy <- energy / nb_offsprings ;
   }
}
```

Note that two keywords can be used to make explicit references to some agents:

- The agent that is currently executing the statements inside the block (for example a newly created agent): **self**
- The agent that is executing the statement that contains this block (for instance, the agent that has called the create statement): **myself**

prey species

We specialize the **prey** species from the **generic_species** species:

- definition of the initial value of the agent variables

```
species prey parent: generic_species {
    ...
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    ...
}
```

predator species

As done for the **prey** species, we specialize the **predator** species from the **generic_species** species:

- definition of the initial value of the agent variables

```
species predator parent: generic_species {
    ...
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    ...
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
```

```
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    int nb_max_offsprings;
    float energy_reproduce;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }

    reflex die when: energy <= 0 {
        do die ;
    }
```

```
    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
```

```
        }
        energy <- energy + energy_transfert ;
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food)));
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
   : "Predator" ;
    parameter "Predator energy transfert: " var:
   predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
   predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
   category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
   category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
   category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
   predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
   predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
   predator_energy_reproduce category: 'Predator' ;

    output {
```

```
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: base ;
            species predator aspect: base ;
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 111

7. Agent Aspect

In this seventh step we will focus on the display and more specifically the aspects of the agents: how they are represented. It can be a simple shape (circle, square, etc.), an icon, a polygon (see later GIS support).

Formulation

- Definition of two new aspects for the prey and predator agents:
  - A icon
  - A square with information about the agent energy
- Use of the **icon** aspect as default aspect for the prey and predator agents.

Model Definition

parent species

We add a new variable of type *image_file* (a particular kind of *file*) called **my_icon** to the **generic_species**. We define as well two new aspects:

- **icon** : draw the image given by the variable **icon**

- **info** : draw a square of side size **size** and color **color**; draw as a text the energy of the agent (with a precision of 2)

```
species generic_species {
   ...
   image_file my_icon;
   ...
   aspect base {
      draw circle(size) color: color ;
   }
   aspect icon {
      draw my_icon size: 2 * size ;
   }
   aspect info {
      draw square(size) color: color ;
      draw string(energy with_precision 2) size: 3 color: #black ;
   }
}
```

prey species

We specialize the **prey** species from the **generic_species** species:

- definition of the initial value of the agent variables

```
species prey parent: generic_species {
   ...
   file my_icon <- file("../images/predator_prey_sheep.png") ;
   ...
}
```

The image file is here: 

You have to copy it in your project folder: images/

predator species

As done for the **prey** species, we specialize the **predator** species from the **generic_species** species:

- definition of the initial value of the agent variables

```
species predator parent: generic_species {
   ...
   file my_icon <- file("../images/predator_prey_wolf.png") ;
   ...
}
```

The image file is here: 

You have to copy it in your project folder: images/

display

We change the default aspect of the prey and predator agents to **icon** aspect.

```
output {
   display main_display {
      grid vegetation_cell lines: #black ;
      species prey aspect: icon ;
      species predator aspect: icon ;
   }
}
```

We define a new display called info_display that displays the prey and predator agents with the **info** aspect.

```
output {
   display info_display {
      species prey aspect: info;
      species predator aspect: info;
   }
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }
```

```
    reflex basic_move {
        myCell <- one_of (myCell.neighbors) ;
        location <- myCell.location ;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
```

```
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
        file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfert ;
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
   : "Predator" ;
```

```
    parameter "Predator energy transfert: " var:
   predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
   predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
   category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
   category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
   category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
   predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
   predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
   predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: info ;
            species predator aspect: info ;
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 112

8. Complex Behavior

This eighth step Illustrates how to define and call actions and how to use conditional statements.

Formulation

- Definition of more complex behaviors for prey and predator agents:
  - The preys agents are moving to the cell containing the highest quantity of food
  - The predator agents are moving if possible to a cell that contains preys; otherwise random cell

Model Definition

parent species

We modify the **basic_move** reflex of the **generic_species** in order to give the **prey** and **predator** a more complex behaviors: instead of choose a random vegetation cell in the neighborhood, the agent will choose a vegetation cell (still in the neighborhood) thanks to a **choose_cell** action. This action will be specialized for each species.

```
species generic_species {
    ...
```

```
        reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
    }
        ...
}
```

We remind that an action is a capability available to the agents of a species (what they can do). It is a block of statements that can be used and reused whenever needed.

- An action can accept arguments.
- An action can return a result (statement return)

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

- for actions that do not return a result:

```
do action_name (arg1: v1 arg2: v2);
do action_name (v1, v2);
```

- for actions that return a result (which is stored in *my_var*):

```
my_var <- action_name (arg1:v1, arg2:v2);
my_var <- action_name (v1, v2);
```

prey species

We specialize the **choose_cell** species for the **prey** species: the agent will choose the vegetation cell of the neighborhood (list myCell.neighbors) that maximizes the quantity of food.

Note that GAMA offers numerous operators to manipulate lists and containers:

- Unary operators : min, max, sum...
- Binary operators :
  - where : returns a sub-list where all the elements verify the condition defined in the right operand.
  - first_with : returns the first element of the list that verifies the condition defined in the right operand.
  - ...

In the case of binary operators, each element (of the first operand list) can be accessed with the keyword **each**

Thus the **choose_cell** action of the **prey** species is defined by:

```
species prey parent: generic_species {
   ...
   vegetation_cell choose_cell {
   return (myCell.neighbors) with_max_of (each.food);
   }
   ...
}
```

predator species

We specialize the **choose_cell** species for the **predator** species: the agent will choose, if possible, a vegetation cell of the neighborhood (list myCell.neighbors) that contains at least a **prey** agent; otherwise it will choose a random cell.

We use for this action the **first_with** operator on the list neighbor vegetation cells (myCell.neighbors) with the following condition: the list of **prey** agents contained in the cell is not empty. Note that we use the **shuffle** operator to randomize the order of the list of neighbor cell.

If all the neighbor cells are empty (myCell_tmp = nil, **nil** is the null value), then the agent choosse a random cell in the neighborhood (one_of (myCell.neighbors)).

GAMA contains statements that allow to execute blocks depending on some conditions:

```
if condition1 {...}
else if condition2{...}
...
else {...}
```

This statement means that if condition1 = true then the first block is executed; otherwise if condition2 = true, then it is the second block, etc. When no conditions are satisfied and an else block is defined (it is optional), this latter is executed.

We then write the **choose_cell** action as follows:

```
   species predator parent: generic_species {
      ...
      vegetation_cell choose_cell {
      vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
    first_with (!(empty (prey inside (each))));
      if myCell_tmp != nil {
        return myCell_tmp;
      } else {
        return one_of (myCell.neighbors);
      }
      }
      ...
   }
```

Note there is ternary operator allowing to directly use a condition structure to evaluate a variable:

```
   condition ? value1 : value2
```

if condition is true, then returns value1; otherwise, returns value2.

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
```

```
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
```

```
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        return (myCell.neighbors) with_max_of (each.food);
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
```

```
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
   first_with (!(empty (prey inside (each))));
        if myCell_tmp != nil {
            return myCell_tmp;
        } else {
            return one_of (myCell.neighbors);
        }
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
```

```
    parameter "Predator max energy: " var: predator_max_energy category
    : "Predator" ;
    parameter "Predator energy transfert: " var:
    predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
    predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
    category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
    category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
    category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
    predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
    predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
    predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: info ;
            species predator aspect: info ;
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 113

9. Stopping condition

This 9th step Illustrates how to use the `pause` action to stop a simulation

Formulation

- Adding of a stopping condition for the simulation: when there is no more prey or predator agents, the simulation stops

Model Definition

We add a new reflex that stops the simulation if the number of preys or the number of predators is null.

```
global {
   ...
   reflex stop_simulation when: (nb_preys = 0) or (nb_predators = 0) {
      do pause ;
   }
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }

    reflex stop_simulation when: (nb_preys = 0) or (nb_predators = 0) {
        do pause ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;
```

```
    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
```

```
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        return (myCell.neighbors) with_max_of (each.food);
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
   first_with (!(empty (prey inside (each))));
        if myCell_tmp != nil {
            return myCell_tmp;
        } else {
            return one_of (myCell.neighbors);
        }
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
```

```
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
    update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
   : "Predator" ;
    parameter "Predator energy transfert: " var:
   predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
   predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
   category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
   category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
   category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
   predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
   predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
   predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
```

```
            species prey aspect: info ;
            species predator aspect: info ;
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 114

10. Charts

This 10th step Illustrates how to define charts.

Formulation

- Adding a new display to visualize:
  - One chart representing the evolution of the quantity of prey and predator agents over the time
  - Two histograms representing the energy distribution of the prey and predator agents

Model Definition

output

GAMA can display various chart types:

- Time series
- Pie charts
- Histograms

A chart must be defined in a display : it behaves exactly like any other layer.

Definition of a chart:

```
chart chart_name type: chart_type  {
     [data]
}
```

The data to draw are defined inside the chart block as follow:

```
    data data_legend value: data_value
```

We add a new display called **Population_information** that refreshes every 5 simulation steps. Inside this display, we define 3 charts: one of type *series* (i.e. time series chart), two of type *histogram* :

- "Species evolution"; background : white; size : {1, 0.5}; position : {0, 0}
  - data1: number_of_preys; color : blue
  - data2: number_of_predator; color : red
- "Prey Energy Distribution"; background : lightGray; size : {0.5, 0.5}; position : {0, 0.5}
  - data "]0;0.25]" : number of preys with (each.energy <= 0.25) ;
  - data "]0.25;0.5]" number of preys with ((each.energy > 0.25) and (each.energy <= 0.5)) ;
  - data "]0.5;0.75]" number of preys with ((each.energy > 0.5) and (each.energy <= 0.75)) ;
  - data "]0.75;1]" number of preys with (each.energy > 0.75) ;
- "Predator Energy Distribution"; background : lightGray; size : {0.5, 0.5}; position : {0.5, 0.5}
  - data "]0;0.25]" : number of predators with (each.energy <= 0.25) ;
  - data "]0.25;0.5]" number of predators with ((each.energy > 0.25) and (each.energy <= 0.5)) ;
  - data "]0.5;0.75]" number of predators with ((each.energy > 0.5) and (each.energy <= 0.75)) ;
  - data "]0.75;1]" number of predators with (each.energy > 0.75) ;

To evaluate the value of the data of the two histogram, we use the operator **list count condition" that returns the number of elements of**list**for which the condition is true.**

```
display Population_information refresh:every(5#cycles) {
    chart "Species evolution" type: series size: {1,0.5} position: {0,
    0} {
        data "number_of_preys" value: nb_preys color: #blue ;
        data "number_of_predator" value: nb_predators color: #red ;
    }
    chart "Prey Energy Distribution" type: histogram background: rgb("
    white") size: {0.5,0.5} position: {0, 0.5} {
        data "]0;0.25]" value: prey count (each.energy <= 0.25) color:#
    blue;
        data "]0.25;0.5]" value: prey count ((each.energy > 0.25) and (
    each.energy <= 0.5)) color:#blue;
        data "]0.5;0.75]" value: prey count ((each.energy > 0.5) and (
    each.energy <= 0.75)) color:#blue;
        data "]0.75;1]" value: prey count (each.energy > 0.75) color:#
    blue;
    }
    chart "Predator Energy Distribution" type: histogram background:
    rgb("white") size: {0.5,0.5} position: {0.5, 0.5} {
        data "]0;0.25]" value: predator count (each.energy <= 0.25)
    color: #red ;
        data "]0.25;0.5]" value: predator count ((each.energy > 0.25)
    and (each.energy <= 0.5)) color: #red ;
        data "]0.5;0.75]" value: predator count ((each.energy > 0.5)
    and (each.energy <= 0.75)) color: #red ;
        data "]0.75;1]" value: predator count (each.energy > 0.75)
    color: #red;
    }
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
```

```
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }

    reflex stop_simulation when: (nb_preys = 0) or (nb_predators = 0) {
        do pause ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
```

```
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
```

```
        return (myCell.neighbors) with_max_of (each.food);
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
   first_with (!(empty (prey inside (each))));
        if myCell_tmp != nil {
            return myCell_tmp;
        } else {
            return one_of (myCell.neighbors);
        }
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
   foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
     update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
   max: 1000 category: "Prey" ;
```

```
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
    ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
    "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
   category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
   min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
   : "Predator" ;
    parameter "Predator energy transfert: " var:
   predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
   predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
   category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
   category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
   category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
   predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
   predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
   predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: info ;
            species predator aspect: info ;
        }
        display Population_information refresh:every(5#cycles) {
            chart "Species evolution" type: series size: {1,0.5}
   position: {0, 0} {
                data "number_of_preys" value: nb_preys color: #blue ;
                data "number_of_predator" value: nb_predators color: #
   red ;
            }
```

```
            chart "Prey Energy Distribution" type: histogram background
    : rgb("lightGray") size: {0.5,0.5} position: {0, 0.5} {
                data "]0;0.25]" value: prey count (each.energy <= 0.25)
     color:#blue;
                data "]0.25;0.5]" value: prey count ((each.energy >
    0.25) and (each.energy <= 0.5)) color:#blue;
                data "]0.5;0.75]" value: prey count ((each.energy >
    0.5) and (each.energy <= 0.75)) color:#blue;
                data "]0.75;1]" value: prey count (each.energy > 0.75)
    color:#blue;
            }
            chart "Predator Energy Distribution" type: histogram
    background: rgb("lightGray") size: {0.5,0.5} position: {0.5, 0.5} {
                data "]0;0.25]" value: predator count (each.energy <=
    0.25) color: #red ;
                data "]0.25;0.5]" value: predator count ((each.energy >
     0.25) and (each.energy <= 0.5)) color: #red ;
                data "]0.5;0.75]" value: predator count ((each.energy >
     0.5) and (each.energy <= 0.75)) color: #red ;
                data "]0.75;1]" value: predator count (each.energy >
    0.75) color: #red;
            }
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 115

11. Writing Files

This 11th step Illustrates how to save data in a text file.

Formulation

- At each simulation step, write in a text file:
  - The time step
  - The number of prey and predator agents
  - The min and max energy of the prey and predator agents

Model Definition

global section

GAMA provides several ways to write a file.

A first ways consist of using the statement **file** in the output section: at each simulation step, the expression given is written in the given file.

```
file file_name type: file_type data: data_to_write;
```

With:

- file_name: string (by default the file in saved in the */models/* of your project directory)
- file_type: string

There are 2 possible types:

- `txt` (text) : in that case, my_data is treated as a string, which is written directly in the file
- `csv` : in that case, my_data is treated as a list of variables to write : ["var1", "var2", "var3"].

A second way to write file consists in using the save statement:

```
save my_data type: file_type to: file_name;
```

With:

- file_type : string
- file_name : string

There are 3 possible types:

- `shp` (shapefile - GIS data): in that case, my_data is treated as a list of agents: all their geometries are saved in the file (with some variables as attributes)
- `txt` (text): in that case, my_data is treated as a string, which is written directly in the file
- `csv`: in that case, my_data is treated as a list of values : [val1, val2, val3].

We use this statement (in a global reflex called **save_result**) to write:

- The cycle step: use of the **cycle** keyword that returns the current simulation step.
- The number of prey and predator agents: use of nb_preys and nb_predators variables
- The min and max energy of the prey and predator agents: use of **list min_of expression** and **list max_of expression** keywords. In addition, we verify with the tertiary operator (condition ? val_if : val_else).

```
reflex save_result when: (nb_preys > 0) and (nb_predators > 0){
    save ("cycle: "+ cycle + "; nbPreys: " + nb_preys
        + "; minEnergyPreys: " + (prey min_of each.energy)
        + "; maxSizePreys: " + (prey max_of each.energy)
        + "; nbPredators: " + nb_predators
        + "; minEnergyPredators: " + (predator min_of each.energy)
        + "; maxSizePredators: " + (predator max_of each.energy))
        to: "results.txt" type: "text" ;
}
```

Complete Model

```
model prey_predator

global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
    }

    reflex save_result when: (nb_preys > 0) and (nb_predators > 0){
        save ("cycle: "+ cycle + "; nbPreys: " + nb_preys
            + "; minEnergyPreys: " + (prey min_of each.energy)
            + "; maxSizePreys: " + (prey max_of each.energy)
```

```
            + "; nbPredators: " + nb_predators
            + "; minEnergyPredators: " + (predator min_of each.energy)
            + "; maxSizePredators: " + (predator max_of each.energy))
            to: "results.txt" type: "text" rewrite: (cycle = 0) ? true
    : false;
    }

    reflex stop_simulation when: (nb_preys = 0) or (nb_predators = 0) {
        do pause ;
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
```

```
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        return (myCell.neighbors) with_max_of (each.food);
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
```

```
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
        }
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
    first_with (!(empty (prey inside (each))));
        if myCell_tmp != nil {
            return myCell_tmp;
        } else {
            return one_of (myCell.neighbors);
        }
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
    foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
     update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
    max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
     ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
     "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
    category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
    min: 0 max: 200 category: "Predator" ;
```

```
    parameter "Predator max energy: " var: predator_max_energy category
    : "Predator" ;
    parameter "Predator energy transfert: " var:
    predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
    predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
    category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
    category: 'Prey' ;
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
    category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
    predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
    predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
    predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: info ;
            species predator aspect: info ;
        }
        display Population_information refresh:every(5#cycles) {
            chart "Species evolution" type: series size: {1,0.5}
    position: {0, 0} {
                data "number_of_preys" value: nb_preys color: #blue ;
                data "number_of_predator" value: nb_predators color: #
    red ;
            }
            chart "Prey Energy Distribution" type: histogram background
    : rgb("lightGray") size: {0.5,0.5} position: {0, 0.5} {
                data "]0;0.25]" value: prey count (each.energy <= 0.25)
     color:#blue;
                data "]0.25;0.5]" value: prey count ((each.energy >
    0.25) and (each.energy <= 0.5)) color:#blue;
                data "]0.5;0.75]" value: prey count ((each.energy >
    0.5) and (each.energy <= 0.75)) color:#blue;
```

```
                data "]0.75;1]" value: prey count (each.energy > 0.75)
    color:#blue;
            }
            chart "Predator Energy Distribution" type: histogram
    background: rgb("lightGray") size: {0.5,0.5} position: {0.5, 0.5} {
                data "]0;0.25]" value: predator count (each.energy <=
    0.25) color: #red ;
                data "]0.25;0.5]" value: predator count ((each.energy >
     0.25) and (each.energy <= 0.5)) color: #red ;
                data "]0.5;0.75]" value: predator count ((each.energy >
     0.5) and (each.energy <= 0.75)) color: #red ;
                data "]0.75;1]" value: predator count (each.energy >
    0.75) color: #red;
            }
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 116

12. Image loading

This 12th step illustrates how to load an image file and to use it to initialize a grid.

Formulation

- Building of the initial environment (food and foodProd of the cells) from a image file

Model Definition

global variable

We add a new global variable: the image file:

```
file map_init <- image_file("../images/predator_prey_raster_map.png");
```

The image file is here: 

You have to copy it in your project folder: images/

model initialization

In order to have a more complex environment, we want to use this image as the initialization of the environment. The food level available in a vegetation_cell will be based on the green level of the corresponding pixel in the image. You will be able to use such process to represent existing real environment in your model. We modify the global init of the model in order to cast the image file in a matrix. We use for that the **file as_matrix {nb_cols, nb_lines}** operator that allows to convert a file (image, csv) to a matrix composed of **nb_cols** columns and **nb_lines** lines.

Concerning the manipulation of matrix, it is possible to obtain the element [i,j] of a matrix by using **my_matrix [i,j]**.

A grid can be view as spatial matrix: each cell of a grid has two built-in variables **grid_x** and **grid_y** that represent the column and line indexes of the cell.

```
init {
    create prey number: nb_preys_init ;
    create predator number: nb_predators_init ;
    matrix init_data <- map_init as_matrix {50,50};
    ask vegetation_cell {
       color <- rgb (init_data[grid_x,grid_y]) ;
       food <- 1 - ((color as list)[0] / 255) ;
       foodProd <- food / 100 ;
    }
 }
```

Conclusion

Congratulations, you have complete your first GAMA models! Now, you have enough knowledge to create many models that includes: dynamic grid-based environment, moving and interacting agents and the needed visualization to make good use of your simulation. Feel free to use this knowledge to create your very own models! Or perhaps you want to continue your study with the more advanced tutorials?

Complete Model

```
model prey_predator
```

```
global {
    int nb_preys_init <- 200;
    int nb_predators_init <- 20;
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1 ;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.01;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.01;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;
    file map_init <- image_file("../images/predator_prey_raster_map.png
   ");

    int nb_preys -> {length (prey)};
    int nb_predators -> {length (predator)};

    init {
        create prey number: nb_preys_init ;
        create predator number: nb_predators_init ;
        ask vegetation_cell {
            color <- rgb (map_init at {grid_x,grid_y}) ;
            food <- 1 - (((color as list)[0]) / 255) ;
            foodProd <- food / 100 ;
        }
    }

    reflex save_result when: (nb_preys > 0) and (nb_predators > 0){
        save ("cycle: "+ cycle + "; nbPreys: " + nb_preys
            + "; minEnergyPreys: " + ((prey as list) min_of each.energy
   )
            + "; maxSizePreys: " + ((prey as list) max_of each.energy)
            + "; nbPredators: " + nb_predators
            + "; minEnergyPredators: " + ((predator as list) min_of
   each.energy)
            + "; maxSizePredators: " + ((predator as list) max_of each.
   energy))
            to: "results.txt" type: "text" ;
    }

    reflex stop_simulation when: (nb_preys = 0) or (nb_predators = 0) {
        do pause ;
```

```
    }
}

species generic_species {
    float size <- 1.0;
    rgb color  ;
    float max_energy;
    float max_transfert;
    float energy_consum;
    float proba_reproduce ;
    float nb_max_offsprings;
    float energy_reproduce;
    image_file my_icon;
    vegetation_cell myCell <- one_of (vegetation_cell) ;
    float energy <- (rnd(1000) / 1000) * max_energy  update: energy -
   energy_consum max: max_energy ;

    init {
        location <- myCell.location;
    }

    reflex basic_move {
        myCell <- choose_cell();
        location <- myCell.location;
    }

    vegetation_cell choose_cell {
        return nil;
    }

    reflex die when: energy <= 0 {
        do die ;
    }

    reflex reproduce when: (energy >= energy_reproduce) and (flip(
   proba_reproduce)) {
        int nb_offsprings <- 1 + rnd(nb_max_offsprings -1);
        create species(self) number: nb_offsprings {
            myCell <- myself.myCell ;
            location <- myCell.location ;
            energy <- myself.energy / nb_offsprings ;
        }
        energy <- energy / nb_offsprings ;
    }

    aspect base {
```

```
        draw circle(size) color: color ;
    }
    aspect icon {
        draw my_icon size: 2 * size ;
    }
    aspect info {
        draw square(size) color: color ;
        draw string(energy with_precision 2) size: 3 color: #black ;
    }
}

species prey parent: generic_species {
    rgb color <- #blue;
    float max_energy <- prey_max_energy ;
    float max_transfert <- prey_max_transfert ;
    float energy_consum <- prey_energy_consum ;
    float proba_reproduce <- prey_proba_reproduce ;
    int nb_max_offsprings <- prey_nb_max_offsprings ;
    float energy_reproduce <- prey_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_sheep.png") ;

    reflex eat when: myCell.food > 0 {
        float energy_transfert <- min([max_transfert, myCell.food]) ;
        myCell.food <- myCell.food - energy_transfert ;
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        return (myCell.neighbors) with_max_of (each.food);
    }
}

species predator parent: generic_species {
    rgb color <- #red ;
    float max_energy <- predator_max_energy ;
    float energy_transfert <- predator_energy_transfert ;
    float energy_consum <- predator_energy_consum ;
    list<prey> reachable_preys update: prey inside (myCell);
    float proba_reproduce <- predator_proba_reproduce ;
    int nb_max_offsprings <- predator_nb_max_offsprings ;
    float energy_reproduce <- predator_energy_reproduce ;
    file my_icon <- file("../images/predator_prey_wolf.png") ;

    reflex eat when: ! empty(reachable_preys) {
        ask one_of (reachable_preys) {
            do die ;
```

```
        }
        energy <- energy + energy_transfert ;
    }

    vegetation_cell choose_cell {
        vegetation_cell myCell_tmp <- shuffle(myCell.neighbors)
    first_with (!(empty (prey inside (each))));
        if myCell_tmp != nil {
            return myCell_tmp;
        } else {
            return one_of (myCell.neighbors);
        }
    }
}

grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float maxFood <- 1.0 ;
    float foodProd <- (rnd(1000) / 1000) * 0.01 ;
    float food <- (rnd(1000) / 1000) max: maxFood update: food +
    foodProd ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food)))
     update: rgb(int(255 * (1 - food)), 255, int(255 *(1 - food))) ;
    list<vegetation_cell> neighbors  <- (self neighbors_at 2);
}

experiment prey_predator type: gui {
    parameter "Initial number of preys: " var: nb_preys_init  min: 0
    max: 1000 category: "Prey" ;
    parameter "Prey max energy: " var: prey_max_energy category: "Prey"
     ;
    parameter "Prey max transfert: " var: prey_max_transfert  category:
     "Prey" ;
    parameter "Prey energy consumption: " var: prey_energy_consum
    category: "Prey" ;
    parameter "Initial number of predators: " var: nb_predators_init
    min: 0 max: 200 category: "Predator" ;
    parameter "Predator max energy: " var: predator_max_energy category
    : "Predator" ;
    parameter "Predator energy transfert: " var:
    predator_energy_transfert  category: "Predator" ;
    parameter "Predator energy consumption: " var:
    predator_energy_consum  category: "Predator" ;
    parameter 'Prey probability reproduce: ' var: prey_proba_reproduce
    category: 'Prey' ;
    parameter 'Prey nb max offsprings: ' var: prey_nb_max_offsprings
    category: 'Prey' ;
```

```
    parameter 'Prey energy reproduce: ' var: prey_energy_reproduce
category: 'Prey' ;
    parameter 'Predator probability reproduce: ' var:
predator_proba_reproduce category: 'Predator' ;
    parameter 'Predator nb max offsprings: ' var:
predator_nb_max_offsprings category: 'Predator' ;
    parameter 'Predator energy reproduce: ' var:
predator_energy_reproduce category: 'Predator' ;

    output {
        display main_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: icon ;
            species predator aspect: icon ;
        }
        display info_display {
            grid vegetation_cell lines: #black ;
            species prey aspect: info ;
            species predator aspect: info ;
        }
        display Population_information refresh:every(5#cycles) {
            chart "Species evolution" type: series size: {1,0.5}
position: {0, 0} {
                data "number_of_preys" value: nb_preys color: #blue ;
                data "number_of_predator" value: nb_predators color: #
red ;
            }
            chart "Prey Energy Distribution" type: histogram background
: rgb("lightGray") size: {0.5,0.5} position: {0, 0.5} {
                data "]0;0.25]" value: prey count (each.energy <= 0.25)
 color:#blue;
                data "]0.25;0.5]" value: prey count ((each.energy >
0.25) and (each.energy <= 0.5)) color:#blue;
                data "]0.5;0.75]" value: prey count ((each.energy >
0.5) and (each.energy <= 0.75)) color:#blue;
                data "]0.75;1]" value: prey count (each.energy > 0.75)
color:#blue;
            }
            chart "Predator Energy Distribution" type: histogram
background: rgb("lightGray") size: {0.5,0.5} position: {0.5, 0.5} {
                data "]0;0.25]" value: predator count (each.energy <=
0.25) color: #red ;
                data "]0.25;0.5]" value: predator count ((each.energy >
 0.25) and (each.energy <= 0.5)) color: #red ;
                data "]0.5;0.75]" value: predator count ((each.energy >
 0.5) and (each.energy <= 0.75)) color: #red ;
```

```
                data "]0.75;1]" value: predator count (each.energy >
    0.75) color: #red;
            }
        }
        monitor "Number of preys" value: nb_preys;
        monitor "Number of predators" value: nb_predators;
    }
}
```

Chapter 117

Road Traffic

This tutorial has for goal to present the use of GIS data and complex geometries. In particular, this tutorial shows how to load gis data, to agentify them and to use a network of polylines to constraint the movement of agents. All the files related to this tutorial (shapefiles and models) are available in the Models Library (project road_traffic_tutorial).

If you are not familiar with agent-based models or GAMA we advice you to have a look at the prey-predator model first.

Model Overview

The model built in this tutorial concerns the study of the road traffic in a small city. Two layers of GIS data are used: a road layer (polylines) and a building layer (polygons). The building GIS data contain an attribute: the 'NATURE' of each building: a building can be either 'Residential' or 'Industrial'. In this model, people agents are moving along the road network. Each morning, they are going to an industrial building to work, and each night they are coming back home. Each time a people agent takes a road, it wears it out. More a road is worn out, more a people agent takes time to go all over it. The town council is able to repair some roads.

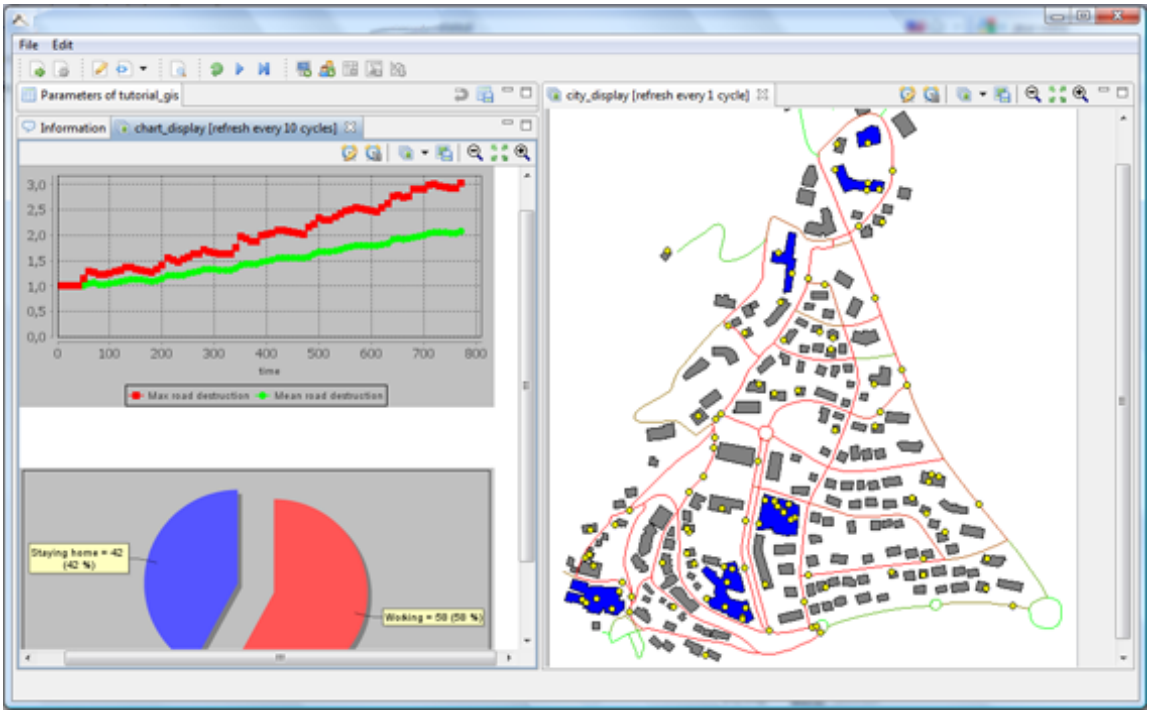


Figure 117.1: images/road_traffic.png

Step List

This tutorial is composed of 7 steps corresponding to 7 models. For each step we present its purpose, an explicit formulation and the corresponding GAML code.

1. Loading of GIS data (buildings and roads)
2. Definition of people agents
3. Movement of the people agents
4. Definition of weight for the road network
5. Dynamic update of the road network
6. Definition of a chart display
7. Automatic repair of roads

Chapter 118

1. Loading of GIS Data

This first step Illustrates how to load GIS data (shapefiles) and to read attributes from GIS data.

Formulation

- Set the time duration of a time step to 10 minutes
- Load, agentify and display two layers of GIS data (building and road). Agentifying a GIS element will allow us to give it a behavior later on (thus not being simply a static/passive object).
- Read the 'NATURE' attribute of the building data: the buildings of 'Residential' type will be colored in gray, the buildings of 'Industrial' type will be color in blue.

Model Definition

species

In this first model, we have to define two species of agents: the **building** agents and the **road** ones. These agents will not have a particular behavior, they will just be displayed. For each of these species, we define a new attribute: **color** of type *rgb*, with for initial value: "black" for the **road** agent and "gray" (by default) for

the **building** agent. Concerning the **building** agent, we define a second attribute named **type** representing the type of the building ("Residential" or "Industrial"). At last, we define an aspect for these species. In this model, we want to represent the geometry of the agent, we then use the keyword **draw** that allow to draw a given geometry. In order to draw the geometry of the agent we use the attribute **shape** (which is a built-in attribute of all agents).

```
species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
    }
}

species road  {
    rgb color <- #black ;

        aspect base {
        draw shape color: color ;
    }
}
```

parameters

GAMA allows to automatically read GIS data that are formatted as shapefiles. In order to let the user chooses his/her shapefiles, we define three parameters. One allowing the user to choose the road shapefile, one allowing him/her to choose the building shapefile, and, at last, one allowing him/her to choose the bounds shapefile. We will come back later on the notion of "bounds" in GAMA.

Definition of the three global variables of type *file* concerning the GIS files:

```
global {
   file shape_file_buildings <- file("../includes/building.shp");
   file shape_file_roads <- file("../includes/road.shp");
   file shape_file_bounds <- file("../includes/bounds.shp");
}
```

All shapefiles are available in the model library (under Library models -> Tutorials -> Road Traffic) or you can download them by following this GitHub link.

In the experiment section, we add three parameters to allow the user to change the shapefile used directly through the UI:

```
experiment road_traffic type: gui {
   parameter "Shapefile for the buildings:" var: shape_file_buildings
    category: "GIS" ;
   parameter "Shapefile for the roads:" var: shape_file_roads category:
     "GIS" ;
   parameter "Shapefile for the bounds:" var: shape_file_bounds
    category: "GIS" ;
}
```

agentification of GIS data

In GAMA, the agentification of GIS data is very straightforward: it only requires to use the **create** command with the **from** facet to pass the shapefile. Each object of the shapefile will be directly used to instantiate an agent of the specified species. The reading of an attribute in a shapefile is also very simple. It only requires to use the **with** facet: the argument of this facet is a dictionary of which the keys are the names of the agent attributes and the value the **read** command followed by the name of the shapefile attribute ("NATURE" in our case).

Init section of the global block: creation of the road and building agents from the shape files. Concerning the **building** agents, reading of the "NATURE" attribute of the shapefile to initiate the value of the **type** attribute. If the **type** attribute is equal to "Industrial" set the **color** attribute to "blue".

```
global {
  ...
  init {
   create building from: shape_file_buildings with: [type::read ("
   NATURE")] {
      if type="Industrial" {
         color <- #blue ;
      }
   }
   create road from: shape_file_roads ;
  }
```

time step

In GAMA, by default, a time step represents 1 second. It is possible to redefine this value by overriding the **step** global variable. This value of the time step is used by the moving primitives of GAMA.

In our model, we define that a step represent 10 minutes. Note that it is possible to define the unit of a value by using # + unit name. For instance, #mn or #km for kilometers.

```
global {
  ...
  float step <- 10 #mn;
  ...
}
```

environment

Building a GIS environment in GAMA requires nothing special, just to define the bounds of the environment, i.e. the geometry of the world agent. It is possible to use a shapefile to automatically define it by computing its envelope. In this model, we use a specific shapefile to define it. However, it would been possible to use the road shapefile to define it and let GAMA computes it enveloppe automatically.

```
global {
  ...
  geometry shape <- envelope(shape_file_bounds);
  ...
}
```

display

We define a display to visualize the road and building agents. We use for that the classic **species** keyword. In order to optimize the display we use an opengl display (facet **type: opengl**).

In the **experiment** block:

```
output {
   display city_display type:opengl {
```

```
        species building aspect: base ;
        species road aspect: base ;
    }
}
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
    }
}

species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
    }
}

species road  {
    rgb color <- #black ;
    aspect base {
        draw shape color: color ;
    }
}
```

```
experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
   category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
   : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
   category: "GIS" ;

    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
        }
    }
}
```

Chapter 119

2. People Agents

This second step Illustrates how to obtain a random point inside a geometry. We will also define some moving agent called *people*.

Formulation

- Define a new species of agents: the **people** agents. The **people** agents have a point for geometry and are represented by a yellow circle of radius 10m.
- At initialization, 100 **people** agents are created. Each **people** agent is placed inside a building of type 'Residential' (randomly selected).

Model Definition

species

We define a new species of agents: the **people** agents. In this model, these agents will not have a specific behavior yet. They will be just displayed. Thus, we just have to define an aspect for the agents. We want to represent the **people** agents by a yellow circle of radius 10m. We then use the **circle** value for the **shape** facet of the **draw** command, with the expected color and radius size (defined by the facet **size**).

```
species people {
    rgb color <- #yellow ;
```

```
    aspect base {
        draw circle(10) color: color;
    }
}
```

parameter

We have to add a new parameter: the number of **people** agents created

In the global section, definition of the **nb_people** variable:

```
    int nb_people <- 100;
```

In the experiment section, definition of the parameter:

```
    parameter "Number of people agents" var: nb_people category: "People
    " ;
```

creation and placement of the people agents

We have to create **nb_people people** agents. Each **people** is placed in a buildings of type 'Residential' randomly selected. In order to simplify the GAML code, we defined a local variable **residential_buildings** that represent the list of buildings of type 'Residential'. To filter the list of **building** agents (obtained by **building**), we use the **where** operator. We use the operator **one_of** to randomly select one agent of this list. There are several ways to place a **people** agent inside this building. In this tutorial, we choose to use the **any_location_in** operator. This operator returns a random point situated inside the operand geometry.

```
global {
    ...
    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
```

```
        list<building> residential_buildings <- building where (each.
    type="Residential");
        create people number: nb_people {
            location <- any_location_in (one_of (residential_buildings)
    );
        }
     }
}
```

display

We add the **people** agent in the defined display.

```
    output {
       display city_display {
          species building aspect: base ;
          species road aspect: base ;
          species people aspect: base ;
       }
    }
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;
    int nb_people <- 100;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
```

```
        list<building> residential_buildings <- building where (each.
    type="Residential");
        create people number: nb_people {
            location <- any_location_in (one_of (residential_buildings)
    );
        }
    }
}

species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
    }
}

species road  {
    rgb color <- #black ;
    aspect base {
        draw shape color: color ;
    }
}

species people {
    rgb color <- #yellow ;

    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
    category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
    : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
    category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
    People" ;

    output {
        display city_display type:opengl {
            species building aspect: base ;
```

```
            species road aspect: base ;
            species people aspect: base ;
        }
    }
}
```

Chapter 120

3. Movement of People

This third step presents how to create a road system from GIS data. More precisely, it shows how to build a graph from a list of polylines and to constrain the movement of an agent according to this graph.

Formulation

- Definition of day_time global variable that will indicate, according to the simulation step, the time of the day: each simulation step will represent 10 minutes, then the day_time variable will be ranged between 0 and 144.
- For each **people** agent: define a living_place(building of type 'Residential') and working place (building of type 'Industrial').
- For each **people** agent: define start_work and end_work hours that respectively represent when the agent leaves its house to go to work and when it leaves its working_place to go back home. These hours will be randomly define between 36 (6 a.m;) and 60 (10 a.m.) for the start_work and 84 (2p.m.) and 132 (10p.m.) for the end_work.
- For each **people** agent: define a objective variable: this one can 'go home' or 'working'.
- For each **people** agent: define a speed. The speed will be randomly define between 50 and 100.
- The **people** agents move along the road, taking the shortest path.

Model Definition

people agents

First, we have to change the skill of the **people** agents: as we want to use an action of the **moving** skill (**goto**), we will provide the **people** agents with this skill. A skill is a built-in module that provide the modeler a self-contain and relevant set of actions and variables.

```
species people skills: [moving]{
    ...
}
```

Then, we have to add new variables to the people agents: living_place, working_place, start_work, end_work, objective. In addition, we will add a "the_target" variable that will represents the point toward which the agent will be currently moving.

```
species people skills: [moving]{
 rgb color <- #yellow ;
 building living_place <- nil ;
 building working_place <- nil ;
 int start_work ;
 int end_work  ;
 string objective ;
 point the_target <- nil ;

    ...
}
```

We define two reflex methods that allow to change the objective (and the_target) of the agent at the start_work and en_work hours. Concerning the *target value*, we choose a random point in the objective building (working_place or living_place) by using the **any_location_in** operator.

```
species people skills: [moving]{
   ...
   reflex time_to_work when: current_hour = start_work and objective
 = "resting"{
        objective <- "working" ;
    the_target <- any_location_in (working_place);
   }

   reflex time_to_go_home when: current_hour = end_work and
objective = "working"{
```

```
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
      }
      ...
  }
```

At last, we define a reflex method that allows the agent to move. If a target point is defined (the_target != nil), the agent moves in direction to its target using the **goto** action (provided by the moving skill). Note that we specified a graph to constraint the movement of the agents on the road network with the facet **on**. We will see later how this graph is built. The agent uses the shortest path (according to the graph) to go to the target point. When the agent arrives at destination (the_target = location), the target is set to nil (the agent will stop moving).

```
  species people skills: [moving]{
      ...
      reflex move when: the_target != nil {
      do goto target: the_target on: the_graph ;
      if the_target = location {
        the_target <- nil ;
      }
     }
  }
```

parameters

We add several parameters (**min_work_start**, **max_work_start**, **min_work_end**, **max_work_end**, **min_speed** and **max_speed**) and two global variables: **the_graph** (graph computed from the road network) and **current_hour** (current hour of the day). The value of the **current_hour** variable is automatically computed at each simulation step and is equals to "(time(the simulation step **step) / 1 hour) modulo 24".**

In the global section:

```
global {
        ...
        int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
    int max_work_end <- 20;
```

```
    float min_speed <- 1.0 #km / #h;
    float max_speed <- 5.0 #km / #h;
    graph the_graph;
        ...
}
```

In the experiment section:

```
experiment road_traffic type: gui {
        ...
    parameter "Earliest hour to start work" var: min_work_start
   category: "People" min: 2 max: 8;
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
   People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
   People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
   0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;
        ...
}
```

initialization

First, we need to compute from the **road** agents, a graph for the moving of the **people** agents. The operator **as_edge_graph** allows to do that. It automatically builds from a set of agents or geometries a graph where the agents are the edges of the graph, a node represent the extremities of the agent geometry.

```
   init {
      ...
      create road from: shape_file_roads ;
      the_graph <- as_edge_graph(road);
      ...
   }
```

We randomly assign one working place and one house to each **people** agent. To simplify the GAML code, we define two temporary variables: the list of buildings of type 'Residential' and the list of buildings of type 'Industrial' (by using the **where**

command). At the creation of each **people** agent, we define a speed, a start_work and end_work to each **people** agent (according to the min and max define in the parameters). We define as well an initial objective ("resting"). Concerning the definition of the living_place and working_place, these ones are randomly chosen in the residential_buildings and industrial_buildings lists.

```
   init {
      ...
      list<building> residential_buildings <- building where (each.type
   ="Residential");
      list<building>  industrial_buildings <- building  where (each.
   type="Industrial") ;
      create people number: nb_people {
          speed <- min_speed + rnd (max_speed - min_speed) ;
          start_work <- min_work_start + rnd (max_work_start -
   min_work_start) ;
          end_work <- min_work_end + rnd (max_work_end - min_work_end)
   ;
          living_place <- one_of(residential_buildings) ;
          working_place <- one_of(industrial_buildings) ;
          objective <- "resting";
          location <- any_location_in (living_place);
      }
      ...
  }
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;
    int nb_people <- 100;
    int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
    int max_work_end <- 20;
    float min_speed <- 1.0 #km / #h;
```

```
    float max_speed <- 5.0 #km / #h;
    graph the_graph;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
        the_graph <- as_edge_graph(road);

        list<building> residential_buildings <- building where (each.
    type="Residential");
        list<building>  industrial_buildings <- building  where (each.
    type="Industrial") ;
        create people number: nb_people {
            speed <- min_speed + rnd (max_speed - min_speed) ;
            start_work <- min_work_start + rnd (max_work_start -
    min_work_start) ;
            end_work <- min_work_end + rnd (max_work_end - min_work_end
    ) ;
            living_place <- one_of(residential_buildings) ;
            working_place <- one_of(industrial_buildings) ;
            objective <- "resting";
            location <- any_location_in (living_place);
        }
    }
}

species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
    }
}

species road  {
    rgb color <- #black ;
    aspect base {
        draw shape color: color ;
    }
}
```

```
species people skills:[moving] {
    rgb color <- #yellow ;
    building living_place <- nil ;
    building working_place <- nil ;
    int start_work ;
    int end_work  ;
    string objective ;
    point the_target <- nil ;

    reflex time_to_work when: current_hour = start_work and objective =
    "resting"{
        objective <- "working" ;
        the_target <- any_location_in (working_place);
    }

    reflex time_to_go_home when: current_hour = end_work and objective
   = "working"{
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
    }

    reflex move when: the_target != nil {
        do goto target: the_target on: the_graph ;
        if the_target = location {
            the_target <- nil ;
        }
    }

    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
   category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
   : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
   category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
   People" ;
    parameter "Earliest hour to start work" var: min_work_start
   category: "People" min: 2 max: 8;
```

```
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
   People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
   People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
   0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;

    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
            species people aspect: base ;
        }
    }
}
```

Chapter 121

4. Weight for Road Network

The present model will introduce how to design a road system, or graph, based on the road GIS data and provide each edge a `weight`representing the destruction level of the road.

Formulation

- Add a **destruction_coeff** variable to the **road** agent. The value of this variable is higher or equal to 1 or lower or equal to 2. At initialization, the value of this variable is randomly defined between 1 and 2.
- In the road network graph, more a road is worn out (destruction_coeff high), more a **people** agent takes time to go all over it. Then the value of the arc representing the road in the graph is equal to "length of the road $\star$ destruction_coeff".
- The color of the road depends of the **destruction_coeff**. If "destruction_coeff = 1", the road is green, if "destruction_coeff = 2", the road is red.

Model Definition

road agent

We add a **destruction_coeff** variable which initial value is randomly defined between 1 and 2 and which have a max of 2. The color of the agent will depend of this variable.

In order to simplify the GAML code, we define a new variable **colorValue** that represents the value of red color and that will be defined between 0 and 255.

```
species road  {
 float destruction_coeff <- 1 + ((rnd(100))/ 100.0) max: 2.0;
 int colorValue <- int(255*(destruction_coeff - 1)) update: int
(255*(destruction_coeff - 1));
 rgb color <- rgb(min([255, colorValue]),max ([0, 255 - colorValue])
,0)  update: rgb(min([255, colorValue]),max ([0, 255 - colorValue])
,0) ;

     ...
}
```

weigthed road network

In GAMA, adding a weight for a graph is very simple, we use the **as_edge_graph** operator with the graph for left-operand and a weight map for the right-operand. A weight contains the weight of each edge: [edge1::weight1, edge2:: weight2,...]. In this model, the weight will be equal to the length of the road (perimeter of the polyline) **its destruction coefficient**.

```
 init {
   ...
   create road from: shape_file_roads ;
   map<road,float> weights_map <- road as_map (each:: (each.
destruction_coeff * each.shape.perimeter));
   the_graph <- as_edge_graph(road) with_weights weights_map;
   ...
}
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
```

```
    float step <- 10 #mn;
    int nb_people <- 100;
    int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
    int max_work_end <- 20;
    float min_speed <- 1.0 #km / #h;
    float max_speed <- 5.0 #km / #h;
    graph the_graph;

    init {
        create building from: shape_file_buildings with: [type::string(
   read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
        map<road,float> weights_map <- road as_map (each:: (each.
   destruction_coeff * each.shape.perimeter));
        the_graph <- as_edge_graph(road) with_weights weights_map;


        list<building> residential_buildings <- building where (each.
   type="Residential");
        list<building>  industrial_buildings <- building  where (each.
   type="Industrial") ;
        create people number: nb_people {
            speed <- min_speed + rnd (max_speed - min_speed) ;
            start_work <- min_work_start + rnd (max_work_start -
   min_work_start) ;
            end_work <- min_work_end + rnd (max_work_end - min_work_end
   ) ;
            living_place <- one_of(residential_buildings) ;
            working_place <- one_of(industrial_buildings) ;
            objective <- "resting";
            location <- any_location_in (living_place);
        }
    }
}

species building {
    string type;
    rgb color <- #gray  ;
```

```
    aspect base {
        draw shape color: color ;
    }
}

species road  {
    float destruction_coeff <- 1 + ((rnd(100))/ 100.0) max: 2.0;
    int colorValue <- int(255*(destruction_coeff - 1)) update: int
    (255*(destruction_coeff - 1));
    rgb color <- rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0)  update: rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0) ;

    aspect base {
        draw shape color: color ;
    }
}

species people skills:[moving] {
    rgb color <- #yellow ;
    building living_place <- nil ;
    building working_place <- nil ;
    int start_work ;
    int end_work  ;
    string objective ;
    point the_target <- nil ;

    reflex time_to_work when: current_hour = start_work and objective =
    "resting"{
        objective <- "working" ;
        the_target <- any_location_in (working_place);
    }

    reflex time_to_go_home when: current_hour = end_work and objective
   = "working"{
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
    }

    reflex move when: the_target != nil {
        do goto target: the_target on: the_graph ;
        if the_target = location {
            the_target <- nil ;
        }
    }
```

```
    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
    category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
    : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
    category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
    People" ;
    parameter "Earliest hour to start work" var: min_work_start
    category: "People" min: 2 max: 8;
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
    People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
    People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
    0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;

    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
            species people aspect: base ;
        }
    }
}
```

Chapter 122

5. Dynamic weights

This 5th step illustrates how to obtain a shortest path from a point to another and to update the weights of an existing graph.

Formulation

- At initialization, the value of the **destruction_coeff** of the **road** agents will be equal to 1.
- Add a new parameter: the **destroy** parameter that represents the value of destruction when a people agent takes a road. By default, it is equal to 0.02.
- When an people arrive at its destination (home or work), it updates the **destruction_coeff** of the **road** agents it took to reach its destination: "destruction_coeff = destruction_coeff - destroy". Then, the graph is updated.

Model Definition

global section

We add the **destroy** parameter.

In the global section, definition of the **destroy** and **update_roads** variables:

```
float destroy <- 0.02;
```

In the experiment section, definition of the parameter:

```
parameter "Value of destruction when a people agent takes a road"
var: destroy category: "Road" ;
```

We define a new reflex that updates the graph at each simulation step. For that, we use the **with_weights** operator. This operator allows to update the weights of an existing graph.

```
global {
   ...
   reflex update_graph{
      map<road,float> weights_map <- road as_map (each:: (each.
destruction_coeff * each.shape.perimeter));
      the_graph <- the_graph with_weights weights_map;
   }
}
```

people agents

At each time-step, after a **people** agent have moved over one or multiple segments, it updates the value of the destruction coefficient of **road** agents crossed (i.e. roads belonging to the path followed). We have for that to set the argument **return_path** to *true* in the **goto** action to obtain the path followed, then to compute the list of agents concerned by this path with the operator **agent_from_geometry**.

```
species people skills: [moving]{
   ...
   reflex move when: the_target != nil {
     path path_followed <- self goto [target::the_target, on::
the_graph, return_path:: true];
     list<geometry> segments <- path_followed.segments;
     loop line over: segments {
         float dist <- line.perimeter;
         ask road(path_followed agent_from_geometry line) {
             destruction_coeff <- destruction_coeff + (destroy *
dist / shape.perimeter);
         }
     }
     if the_target = location {
         the_target <- nil ;
     }
 }
```

```
    ...
    }
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;
    int nb_people <- 100;
    int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
    int max_work_end <- 20;
    float min_speed <- 1.0 #km / #h;
    float max_speed <- 5.0 #km / #h;
    float destroy <- 0.02;
    graph the_graph;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- as_edge_graph(road) with_weights weights_map;


        list<building> residential_buildings <- building where (each.
    type="Residential");
        list<building>  industrial_buildings <- building  where (each.
    type="Industrial") ;
        create people number: nb_people {
            speed <- min_speed + rnd (max_speed - min_speed) ;
```

```
            start_work <- min_work_start + rnd (max_work_start -
    min_work_start) ;
            end_work <- min_work_end + rnd (max_work_end - min_work_end
    ) ;
            living_place <- one_of(residential_buildings) ;
            working_place <- one_of(industrial_buildings) ;
            objective <- "resting";
            location <- any_location_in (living_place);
        }
    }

    reflex update_graph{
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- the_graph with_weights weights_map;
    }
}

species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
    }
}

species road  {
    float destruction_coeff <- 1 + ((rnd(100))/ 100.0) max: 2.0;
    int colorValue <- int(255*(destruction_coeff - 1)) update: int
    (255*(destruction_coeff - 1));
    rgb color <- rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0)  update: rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0) ;

    aspect base {
        draw shape color: color ;
    }
}

species people skills:[moving] {
    rgb color <- #yellow ;
    building living_place <- nil ;
    building working_place <- nil ;
    int start_work ;
    int end_work  ;
```

```
    string objective ;
    point the_target <- nil ;

    reflex time_to_work when: current_hour = start_work and objective =
    "resting"{
        objective <- "working" ;
        the_target <- any_location_in (working_place);
    }

    reflex time_to_go_home when: current_hour = end_work and objective
   = "working"{
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
    }

    reflex move when: the_target != nil {
        path path_followed <- self goto [target::the_target, on::
   the_graph, return_path:: true];
        list<geometry> segments <- path_followed.segments;
        loop line over: segments {
            float dist <- line.perimeter;
            ask road(path_followed agent_from_geometry line) {
                destruction_coeff <- destruction_coeff + (destroy *
   dist / shape.perimeter);
            }
        }
        if the_target = location {
            the_target <- nil ;
        }
    }

    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
   category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
   : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
   category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
   People" ;
```

```
    parameter "Earliest hour to start work" var: min_work_start
   category: "People" min: 2 max: 8;
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
   People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
   People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
   0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;
    parameter "Value of destruction when a people agent takes a road"
   var: destroy category: "Road" ;
    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
            species people aspect: base ;
        }
    }
}
```

Chapter 123

6. Charts

This 6th step illustrates how to display charts.

Formulation

- Add a chart to display the evolution of the road destruction: the mean value of the **destruction_coeff** of the **road** agents, and its max value (refreshed every 10 simulation steps).
- Add a chart to display the activity of the **people** agent (working or staying home, refreshed every 10 simulation steps).

Model Definition

chart display

First we add a chart of type **series** to display the road destruction evolution. To compute the mean of the **destruction_coeff**, we use the **mean** operator. For the max, we use the **max_of** operator.

```
    output {
      display chart_display refresh:every(10#cycles) {
            chart "Road Status" type: series size: {1, 0.5} position:
  {0, 0} {
```

```
                data "Mean road destruction" value: mean (road
    collect each.destruction_coeff) style: line color: #green ;
           data "Max road destruction" value: road max_of each.
    destruction_coeff style: line color: #red ;
         }
          ...
       }
    }
```

Second, we add a chart of type **pie** to display the activity of the **people** agents. We use for that the **objective** variable of the **people** agents and the **count** operator that allows to compute the number of elements of a list that verify a condition.

```
    output {
      ...
      display chart_display refresh:every(10#cycles) {
          ...
          chart "People Objectif" type: pie style: exploded size: {1,
    0.5} position: {0, 0.5}{
           data "Working" value: people count (each.objective="working"
    ) color: #magenta ;
           data "Resting" value: people count (each.objective="resting"
    ) color: #blue ;
       }
      }
    }
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;
    int nb_people <- 100;
    int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
```

```
    int max_work_end <- 20;
    float min_speed <- 1.0 #km / #h;
    float max_speed <- 5.0 #km / #h;
    float destroy <- 0.02;
    graph the_graph;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- as_edge_graph(road) with_weights weights_map;


        list<building> residential_buildings <- building where (each.
    type="Residential");
        list<building>  industrial_buildings <- building  where (each.
    type="Industrial") ;
        create people number: nb_people {
            speed <- min_speed + rnd (max_speed - min_speed) ;
            start_work <- min_work_start + rnd (max_work_start -
    min_work_start) ;
            end_work <- min_work_end + rnd (max_work_end - min_work_end
    ) ;
            living_place <- one_of(residential_buildings) ;
            working_place <- one_of(industrial_buildings) ;
            objective <- "resting";
            location <- any_location_in (living_place);
        }
    }

    reflex update_graph{
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- the_graph with_weights weights_map;
    }
}

species building {
    string type;
    rgb color <- #gray  ;
```

```
    aspect base {
        draw shape color: color ;
    }
}

species road  {
    float destruction_coeff <- 1 + ((rnd(100))/ 100.0) max: 2.0;
    int colorValue <- int(255*(destruction_coeff - 1)) update: int
    (255*(destruction_coeff - 1));
    rgb color <- rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0)  update: rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0) ;

    aspect base {
        draw shape color: color ;
    }
}

species people skills:[moving] {
    rgb color <- #yellow ;
    building living_place <- nil ;
    building working_place <- nil ;
    int start_work ;
    int end_work  ;
    string objective ;
    point the_target <- nil ;

    reflex time_to_work when: current_hour = start_work and objective =
    "resting"{
        objective <- "working" ;
        the_target <- any_location_in (working_place);
    }

    reflex time_to_go_home when: current_hour = end_work and objective
    = "working"{
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
    }

    reflex move when: the_target != nil {
        path path_followed <- self goto [target::the_target, on::
    the_graph, return_path:: true];
        list<geometry> segments <- path_followed.segments;
        loop line over: segments {
            float dist <- line.perimeter;
```

```
            ask road(path_followed agent_from_geometry line) {
                destruction_coeff <- destruction_coeff + (destroy *
    dist / shape.perimeter);
            }
        }
        if the_target = location {
            the_target <- nil ;
        }
    }

    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
    category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
    : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
    category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
    People" ;
    parameter "Earliest hour to start work" var: min_work_start
    category: "People" min: 2 max: 8;
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
    People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
    People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
    0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;
    parameter "Value of destruction when a people agent takes a road"
    var: destroy category: "Road" ;
    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
            species people aspect: base ;
        }
        display chart_display refresh:every(10#cycles) {
```

```
            chart "Road Status" type: series size: {1, 0.5} position:
    {0, 0} {
                data "Mean road destruction" value: mean (road collect
    each.destruction_coeff) style: line color: #green ;
                data "Max road destruction" value: road max_of each.
    destruction_coeff style: line color: #red ;
            }
            chart "People Objectif" type: pie style: exploded size: {1,
     0.5} position: {0, 0.5}{
                data "Working" value: people count (each.objective="
    working") color: #magenta ;
                data "Resting" value: people count (each.objective="
    resting") color: #blue ;
            }
        }
    }
}
```

Chapter 124

7. Automatic Road Repair

This 7th step illustrates how to select in a list an element that optimize a given function.

Formulation

- Add a new parameter, **repair_time**, that is equal to 2.
- Every **repair_time**, the **road** with the highest **destruction_coeff** value is repaired (set its **destruction_coeff** to 1).

Model Definition

parameters

We add a new parameter: the **repair_time**.

In the global section, definition of the **repair_time** variable:

```
int repair_time <- 2 ;
```

In the experiment section, definition of the parameter:

```
parameter "Number of steps between two road repairs" var:
 repair_time category: "Road" ;
```

road repairing

We have to add a reflex method in the global section that is triggered every **repair_-time** hours / step. This method selects, thanks to the **with_max_of** operation the **road** agent with the highest **destruction_coeff** value, then sets this value at 1.

```
global {
    ...
    reflex repair_road when: every(repair_time #hour / step) {
     road the_road_to_repair <- road with_max_of (each.destruction_coeff
    ) ;
     ask the_road_to_repair {
         destruction_coeff <- 1.0 ;
     }
    }
}
```

Complete Model

```
model tutorial_gis_city_traffic

global {
    file shape_file_buildings <- file("../includes/building.shp");
    file shape_file_roads <- file("../includes/road.shp");
    file shape_file_bounds <- file("../includes/bounds.shp");
    geometry shape <- envelope(shape_file_bounds);
    float step <- 10 #mn;
    int nb_people <- 100;
    int current_hour update: (time / #hour) mod 24;
    int min_work_start <- 6;
    int max_work_start <- 8;
    int min_work_end <- 16;
    int max_work_end <- 20;
    float min_speed <- 1.0 #km / #h;
    float max_speed <- 5.0 #km / #h;
    float destroy <- 0.02;
    int repair_time <- 2 ;
    graph the_graph;

    init {
        create building from: shape_file_buildings with: [type::string(
    read ("NATURE"))] {
            if type="Industrial" {
```

```
                color <- #blue ;
            }
        }
        create road from: shape_file_roads ;
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- as_edge_graph(road) with_weights weights_map;


        list<building> residential_buildings <- building where (each.
    type="Residential");
        list<building>  industrial_buildings <- building  where (each.
    type="Industrial") ;
        create people number: nb_people {
            speed <- min_speed + rnd (max_speed - min_speed) ;
            start_work <- min_work_start + rnd (max_work_start -
    min_work_start) ;
            end_work <- min_work_end + rnd (max_work_end - min_work_end
    ) ;
            living_place <- one_of(residential_buildings) ;
            working_place <- one_of(industrial_buildings) ;
            objective <- "resting";
            location <- any_location_in (living_place);
        }
     }

     reflex update_graph{
        map<road,float> weights_map <- road as_map (each:: (each.
    destruction_coeff * each.shape.perimeter));
        the_graph <- the_graph with_weights weights_map;
     }
     reflex repair_road when: every(repair_time #hour / step) {
        road the_road_to_repair <- road with_max_of (each.
    destruction_coeff) ;
        ask the_road_to_repair {
            destruction_coeff <- 1.0 ;
        }
     }
}

species building {
    string type;
    rgb color <- #gray  ;

    aspect base {
        draw shape color: color ;
```

```
    }
}

species road  {
    float destruction_coeff <- 1 + ((rnd(100))/ 100.0) max: 2.0;
    int colorValue <- int(255*(destruction_coeff - 1)) update: int
    (255*(destruction_coeff - 1));
    rgb color <- rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0)  update: rgb(min([255, colorValue]),max ([0, 255 - colorValue])
    ,0) ;

    aspect base {
        draw shape color: color ;
    }
}

species people skills:[moving] {
    rgb color <- #yellow ;
    building living_place <- nil ;
    building working_place <- nil ;
    int start_work ;
    int end_work  ;
    string objective ;
    point the_target <- nil ;

    reflex time_to_work when: current_hour = start_work and objective =
     "resting"{
        objective <- "working" ;
        the_target <- any_location_in (working_place);
    }

    reflex time_to_go_home when: current_hour = end_work and objective
    = "working"{
        objective <- "resting" ;
        the_target <- any_location_in (living_place);
    }

    reflex move when: the_target != nil {
        path path_followed <- self goto [target::the_target, on::
    the_graph, return_path:: true];
        list<geometry> segments <- path_followed.segments;
        loop line over: segments {
            float dist <- line.perimeter;
            ask road(path_followed agent_from_geometry line) {
                destruction_coeff <- destruction_coeff + (destroy *
    dist / shape.perimeter);
```

```
            }
        }
        if the_target = location {
            the_target <- nil ;
        }
    }

    aspect base {
        draw circle(10) color: color;
    }
}

experiment road_traffic type: gui {
    parameter "Shapefile for the buildings:" var: shape_file_buildings
    category: "GIS" ;
    parameter "Shapefile for the roads:" var: shape_file_roads category
    : "GIS" ;
    parameter "Shapefile for the bounds:" var: shape_file_bounds
    category: "GIS" ;
    parameter "Number of people agents" var: nb_people category: "
    People" ;
    parameter "Earliest hour to start work" var: min_work_start
    category: "People" min: 2 max: 8;
    parameter "Latest hour to start work" var: max_work_start category:
     "People" min: 8 max: 12;
    parameter "Earliest hour to end work" var: min_work_end category: "
    People" min: 12 max: 16;
    parameter "Latest hour to end work" var: max_work_end category: "
    People" min: 16 max: 23;
    parameter "minimal speed" var: min_speed category: "People" min:
    0.1 #km/#h ;
    parameter "maximal speed" var: max_speed category: "People" max: 10
     #km/#h;
    parameter "Value of destruction when a people agent takes a road"
    var: destroy category: "Road" ;
    parameter "Number of hours between two road repairs" var:
    repair_time category: "Road" ;

    output {
        display city_display type:opengl {
            species building aspect: base ;
            species road aspect: base ;
            species people aspect: base ;
        }
        display chart_display refresh:every(10#cycles) {
```

```
            chart "Road Status" type: series size: {1, 0.5} position:
    {0, 0} {
                data "Mean road destruction" value: mean (road collect
    each.destruction_coeff) style: line color: #green ;
                data "Max road destruction" value: road max_of each.
    destruction_coeff style: line color: #red ;
            }
            chart "People Objectif" type: pie style: exploded size: {1,
     0.5} position: {0, 0.5}{
                data "Working" value: people count (each.objective="
    working") color: #magenta ;
                data "Resting" value: people count (each.objective="
    resting") color: #blue ;
            }
        }
    }
}
```

Chapter 125

3D Tutorial

This tutorial introduces the 3D features offered by GAMA.

Model Overview

Step List

This tutorial is composed of 3 steps corresponding to 3 models. For each step we present its purpose, an explicit formulation and the corresponding GAML code.

1. Basic model
2. Moving cells
3. Moving cells with neighbors

Chapter 126

1. Basic Model

In this first step, we will see how to define a 3D environment and populate it.

Formulation

Initialize a 3D world with a population of cells placed randomly in a 3D 100x100x100 cube.

- Definition of the **cells** species
- Definition of the **nb_cells** parameter
- Creation of **nb_cells cells** agents randomly located in the 3D environment size: 100x100x100.

Model Definition

In this model we define one species of agent: the **cells** agents. The agents will be just displayed as a blue sphere of radius 1.

```
species cells{
  aspect default {
    draw sphere(1) color:#blue;
  }
}
```

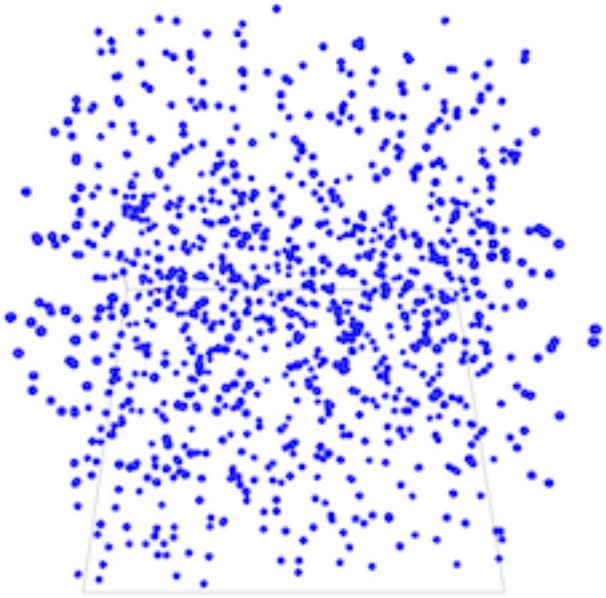


Figure 126.1: images/3D_model_LQ.png

Global Section

Global variable

Definition of a global variable **nb_cells** of type *int* representing the number of **cells** agents.

```
global {
   int nb_cells <-100;
}
```

Model initialization

Definition of the init block in order to create *nb_cells* **cells** agents. By default an agent is created with a random location in x and y, and a z value equal to 0. In our case we want to place the **cells** randomly in the 3D environment so we set a random value for x, y and z.

```
create cells number: nb_cells {
  location <- {rnd(100), rnd(100), rnd(100)};
}
```

Experiment

In our model, we define a basic gui experiment called *Tuto3D* :

```
experiment Tuto3D type: gui {
}
```

Input

Definition of a parameter from the the global variable *nb_cells* :

```
experiment Tuto3D type: gui {
   parameter "Initial number of cells: " var: nb_cells min: 1 max: 1000
    category: "Cells";
}
```

Output

In our model, we define a display to draw the **cells** agents in a 3D environment.

```
output {
  display View1 type:opengl {
    species cells;
  }
}
```

Complete Model

This model is available in the model library (under Tutorials > 3D) and the GIT version of the model can be found here Model 01.gaml

```
model Tuto3D

global {
  int nb_cells <-100;
  init {
    create cells number: nb_cells {
      location <- {rnd(100), rnd(100), rnd(100)};
    }
  }
}

species cells{
  aspect default {
    draw sphere(1) color:#blue;
  }
}

experiment Tuto3D  type: gui {
  parameter "Initial number of cells: " var: nb_cells min: 1 max: 1000
   category: "Cells" ;
  output {
    display View1 type:opengl {
      species cells;
    }
  }
}
```

Chapter 127

2. Moving Cells

This second step model adds the **moving3D** skills to the **cell** agents and simply makes move the **cells** agent by defining a reflex that will call the action **move**. We will also add additional visual information to the display.

Formulation

- Redefining the shape of the world with a 3D Shape.
- Attaching new skills (**moving3D**) to **cells** agent.
- Modify **cells** aspect
- Add a graphics layer

Model Definition

Global Section

Global variable

We use a new global variable called _environmentSize_that to define the size of our 3D environment. In the global section we define the new variable

```
int environmentSize <-100;
```

Then we redefine the shape of the world (by default the shape of the world is a 100x100 square) as cube that will have the size defined by the *environment_size* variable. To do so we change the shape of the world in the **global** section:

```
geometry shape <- cube(environmentSize);
```

Model initialization

When we create the **cells** agent we want to place them randomly in the 3D environement. To do so we set the location with a random value for x, y and z between 0 and *environmentSize*.

```
create cells number: nb_cells {
  location <- {rnd(environmentSize), rnd(environmentSize), rnd(
   environmentSize)};
}
```

Moving3D skills

In the previous example, we only created **cells** agent that did not have any behavior. In this step we want to make move the **cells** agent. To do so we add a **moving3D** skills to the cells.

More information on built-in skills proposed by Gama can be found here

```
species cells skills:[moving3D]{
...
}
```

Then we define a new reflex for the species **cells** that consists in calling the action *move* bundled in **moving3D** skills.

```
reflex move{
  do move;
}
```

Finally we modify a bit the aspect of the sphere to set its size according to the *environmentSize* global variable previously defined.

```
aspect default {
  draw sphere(environmentSize*0.01) color:#blue;
}
```

Experiment

The experiment is the same as the previous one except that we will display the bounds of the environment by using the **graphics** layer.

```
graphics "env"{
  draw cube(environmentSize) color: #black empty:true;
}
```

Output

```
output {
  display View1 type:opengl{
    graphics "env"{
      draw cube(environmentSize) color: #black empty:true;
    }
    species cells;
  }
}
```

Complete Model

The GIT version of the model can be found here Model 02.gaml

```
model Tuto3D

global {
  int nb_cells <-100;
  int environmentSize <-100;
  geometry shape <- cube(environmentSize);
  init {
    create cells number: nb_cells {
      location <- {rnd(environmentSize), rnd(environmentSize), rnd(
   environmentSize)};
    }
  }
}

species cells skills:[moving3D]{

  reflex move{
```

```
    do move;
  }
  aspect default {
    draw sphere(environmentSize*0.01) color:#blue;
  }
}

experiment Tuto3D  type: gui {
  parameter "Initial number of cells: " var: nb_cells min: 1 max: 1000
   category: "Cells" ;
  output {
    display View1 type:opengl{
      graphics "env"{
        draw cube(environmentSize) color: #black empty:true;
      }
      species cells;
    }
  }
}
```

Chapter 128

3. Connections

Formulation

- Mapping the network of connection

Model Definition

In this final step we will display edges between cells that are within a given distance.

Cells update

We add a new reflex to collect the neighbors of the cell that are within a certain distance :

```
species cells skills:[moving3D]{
...
reflex computeNeighbors {
                neighbors <- cells select ((each distance_to self) <
    10);
        }
}
```

Then we update the cell aspect as follows. For each elements (cells) of the **neighbors** list we draw a line between this neighbor's location and the current cell's location.

```
aspect default {
  draw sphere(environmentSize*0.01) color:#orange;
  loop pp over: neighbors {
    draw line([self.location,pp.location]);
  }
}
```

Complete Model

The GIT version of the model can be found here Model 03.gaml

```
model Tuto3D

global {
  int nb_cells <-100;
  int environmentSize <-100;
  geometry shape <- cube(environmentSize);
  init {
    create cells number: nb_cells {
      location <- {rnd(environmentSize), rnd(environmentSize), rnd(
   environmentSize)};
    }
  }
}

species cells skills: [moving3D] {
    rgb color;
    list<cells> neighbors;
    int offset;

    reflex move {
      do wander;
    }

    reflex computeNeighbors {
      neighbors <- cells select ((each distance_to self) < 10);
    }

    aspect default {
        draw sphere(environmentSize*0.01) color:#orange;
        loop pp over: neighbors {
            draw line([self.location,pp.location]);
        }
```

```
    }
}


experiment Tuto3D  type: gui {
  parameter "Initial number of cells: " var: nb_cells min: 1 max: 1000
   category: "Cells" ;
  output {
    display View1 type:opengl background:rgb(10,40,55){
      graphics "env"{
        draw cube(environmentSize) color: #black empty:true;
      }
      species cells;
    }
  }
}
```

Chapter 129

Incremental Model

This tutorial has for goal to give an overview all most of the capabilities of GAMA. In particular, it presents how to build a simple model and the use of GIS data, graphs, 3D visualization, multi-level modeling and differential equations. All the files related to this tutorial (images and models) are available in the Models Library (project Tutorials/Incremental Model).

Model Overview

The model built in this tutorial concerns the study of the spreading of a disease in a small city. Three type of entities are taken into account: the people, the buildings and the roads.

We made the following modeling choice:

- Simulation step: 1 minute
- People are moving on the roads from building to building
- People use the shortest path to move between buildings
- All people have the same speed and move at constant speed
- Each time, people arrived at a building they are staying a certain time
- The staying time depends on the current hour (lower at 9h - go to work - at 12h go to lunch - at 18h - go back home)
- Infected people are never cured

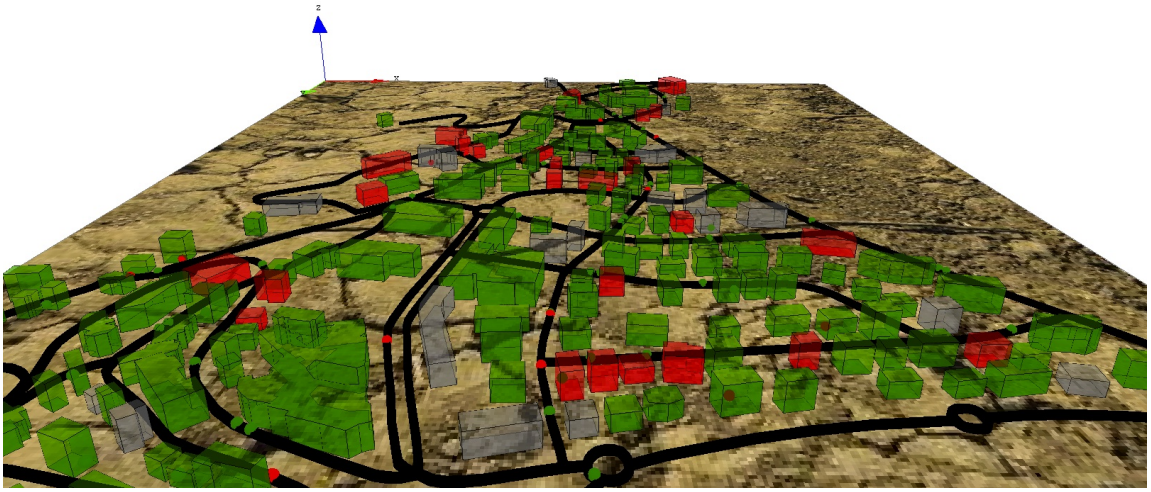


Figure 129.1: images/incremental_model.jpg

Step List

This tutorial is composed of 7 steps corresponding to 7 models. For each step we present its purpose, an explicit formulation and the corresponding GAML code.

1. Simple SI Model
2. Charts
3. Integration of GIS Data
4. Movement on Graph
5. Visualizing in 3D
6. Multi-Level
7. Differential Equations

Chapter 130

1. Simple SI Model

This first step Illustrates how to write a model in GAMA. In particular, it describes how to structure a model and how to define species - that are the key components of GAMA models.

Formulation

- Definition of the **people** species with a variable (is_infected) , an aspect (base) and two behaviors (move and infect)
- Definition of **nb_infected_init**, **distance_infection** and **proba_infection** parameters
- Creation of **500 people** agents randomly located in the environment (size: 500x500)
- Definition a display to visualize the people agents.

Model Definition

model structure

A GAMA model is composed of three types of sections:

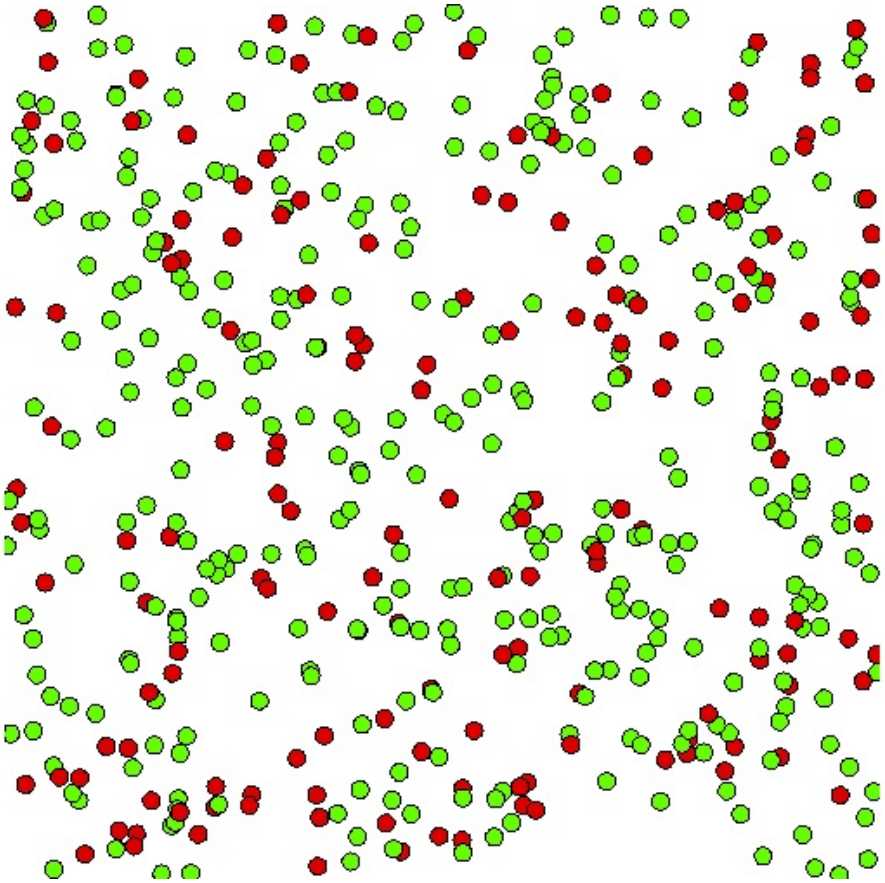


Figure 130.1: images/Incremental_model1.jpg

- **global** : this section, that is unique, defines the "world" agent, a special agent of a GAMA model. It represents all that is global to the model: dynamics, variables, actions. In addition, it allows to initialize the simulation (init block).
- **species** : these sections define the species of agents composing the model.
- **experiment** : these sections define a context of execution of the simulations. In particular, it defines the input (parameters) and output (displays, files...) of a model.

More details about the different sections of a GAMA model can be found here.

species

A species represents a «prototype» of agents: it defines their common properties.

A species includes several sub-definitions:

- the internal state of its agents (attributes)
- their behavior
- how they are displayed (aspects)

GAMA provides as well the possibility to give **skills** to species of agents. A skill is a module integrating variables and actions coded in Java.

Concerning our model, we will give the **moving** skill to the **people** agents: it will give to the people agents supplementary variables (speed, heading, destination) and actions (follow, goto, move, wander).

```
species people skills:[moving]{
}
```

Concerning the internal state, a attribute is defined as follows: type of the attribute (int (integer), float (floating point number), string, bool (boolean, true or false), point (coordinates), list, pair, map, file, matrix, agent species, rgb (color), graph, path...) + name

- Optional facets: <- (initial value), update (value recomputed at each step of the simulation), function:{..} (value computed each time the variable is used), min, max

Note that all the species inherit from predefined built-in variables:

- A name (*name*)
- A shape (*shape*)
- A location (*location*): the centroid of its shape.

We will give a variable to **people** agents: **is_infected** (bool):

```
species people skills:[moving]{
    bool is_infected <- false;
}
```

Concerning the display of an agent, aspects have to be defined. An aspect represents a possible way to display the agents of a species : aspect aspect_name {...} In the block of an aspect, it is possible to draw:

- A geometry: for instance, the shape of the agent
- An image: to draw icons
- A text: to draw a text

We define an aspect for this species. In this model, we want to display for each people agent a circle of radius 5 and red or green according to the value of is_infected (if infected: red, green otherwise). We then use the keyword **draw** with a circle shape. To define the color that depends on is_infected, we use the tertiary operator **condition ? val1 : val2**. If the condition is true, this operator will return **val1**, otherwise **val2**. Note that it is possible to get a color value by using the symbol # + color name: e.g. #blue, #red, #white, #yellow, #magenta, #pink...

```
species people skills:[moving]{
    ...
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
}
```

Concerning the behavior of agents, the simplest way to define it is through reflexes. A reflex is a block of statements (that can be defined in global or any species) that will be automatically executed at each simulation step if its condition is true. A reflex is defined as follows:

```
reflex reflex_name when: condition {...}
```

The **when** facet is optional: when it is omitted, the reflex is activated at each time step. Note that if several reflexes are defined for a species, the reflexes will be activated following their definition order.

We define a first reflex called **move** that allows the people agents to move using the **wander** action (provided by the **moving** skill) that allows to randomly move (with taking into account the agent **speed**)

```
species people {
        ...
        reflex move{
        do wander;
    }
}
```

Note that an action is a capability available to the agents of a species (what they can do). It is a block of statements that can be used and reused whenever needed. Some actions, called primitives, are directly coded in Java: for instance, the **wander** action defined in the **moving** skill.

- An action can accept arguments. For instance, write takes an argument called message.
- An action can return a result.

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

- for actions that do not return a result:

```
do action_name arg1: v1 arg2: v2;
```

- for actions that return a result:

```
my_var <- self action_name (arg1:v1, arg2:v2);
```

The second reflex we have to define is the **infect** one. This reflex will be activated only if **is_infected** is true. This reflex consists is asking all the people agents at a distance lower or equal to **infection_distance** to become infected with a probability **proba_infection**.

```
species people skills:[moving]{
    ...
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    ...
}
```

Note that we used the **ask** statement. This statement allows to make a remote agent executes a list of statements. We used as well the **flip** operator that allows to test a probability.

global section

The global section represents the definition of the species of a specific agent (called world). The world agent represents everything that is global to the model: dynamics, variables... It allows to init simulations (init block): the world is always created and initialized first when a simulation is launched. The geometry (shape) of the world agent is by default a square with 100m for side size, but can be redefined if necessary. In the same way, the modeler can redefine the **step** variable that represents the duration of a simulation step and that is by default 1 second.

global variable

For our model, we define 4 global variables: **nb_people** (int, init value: 500), **infection distance** (float value, init value: 2 meters), **proba_infection** (float, init value: 0.05) and nb_infected_init (int, init value: 5). In addition, we redefine the geometry of the world by a square of 500 meters size and a simulation step of 1 minute.

```
global{
    int nb_people <- 500;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    float step <- 1 #minutes;
```

```
    geometry shape<-square(500 #m);
}
```

Model initialization

The init section of the global block allows to initialize the model. The statement *create* allows to create agents of a specific species: **create** species_name + :

- number: number of agents to create (int, 1 by default)
- from: GIS file to use to create the agents (string or file)
- returns: list of created agents (list)

For our model, the definition of the init block in order to create **nb_people** people agents. We set the init value of the **speed** variable (given by the **moving** skill) to 5km/h. > In addition we ask **nb_infected_init** people to become infected (use of the **nb among list** to randomly draw nb elements of the list).

```
global{
    ...
    init{
        create people number:nb_people {
            speed <- 5.0 #km/#h;
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}
```

experiment

An experiment block defines how a model can be simulated (executed). Several experiments can be defined for a model. They are defined using : **experiment** exp_name type: gui/batch {`[input] [output]`}

- gui: experiment with a graphical interface, which displays its input parameters and outputs.
- batch: Allows to setup a series of simulations (w/o graphical interface).

In our model, we define a gui experiment called main_experiment :

```
experiment main_experiment type: gui {
}
```

input

Experiments can define (input) parameters. A parameter definition allows to make the value of a global variable definable by the user through the graphic interface.

A parameter is defined as follows:

parameter title var: global_var category: cat;

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

Note that the init, min and max values can be defined in the global variable definition.

In the experiment, definition of three parameters from the the global variable **infection_distance**, **proba_infection** and **nb_infected_init** :

```
experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    ...
}
```

output

Output blocks are defined in an experiment and define how to visualize a simulation (with one or more display blocks that define separate windows). Each display can be refreshed independently by defining the facet **refresh_every:** nb (int) (the display will be refreshed every nb steps of the simulation).

Each display can include different layers (like in a GIS):

- Agents lists: **agents** layer_name value: agents_list aspect: my_aspect;
- Agents species: **species** my_species aspect: my_aspect
- Images: **image** layer_name file: image_file;
- Texts: **texte** layer_name value: my_text;
- Charts: see later.

Note that it is possible to define a opengl display (for 3D display) by using the facet **type: opengl**.

In our model, we define a display to draw the **people** agents with their **circle** aspect.

```
experiment main_experiment type:gui{
    ...
        output {
        display map {
            species people aspect:circle;
        }
    }
}
```

Complete Model

```
model SI_city

global{
    int nb_people <- 500;
    float agent_speed <- 5.0 #km/#h;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    float step <- 1 #minutes;
    geometry shape<-square(500 #m);

    init{
        create people number:nb_people;
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }

}
```

```
species people skills:[moving]{
    float speed <- agent_speed;
    bool is_infected <- false;
    reflex move{
        do wander;
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        display map {
            species people aspect:circle;
        }
    }
}
```

Chapter 131

2. Charts

This step Illustrates how to define monitors and charts in GAMA. In addition, it illustrates how to define a stopping condition for the simulation.

Formulation

- Definition of new global variables: current_hour, nb_people_infected, nb_-people_not_infected, infected_rate
- Definition of a monitor to follow the current hour and the nb of people infected
- Definition of a series chart to follow the number of people infected and not infected
- Definition of a stopping condition (when infected rate = 1)

Model Definition

global variables

In order to define dynamic variable able to update itself, we use the **update** facet of variable definition. Indeed, at each simulation step, all the agents (and the world agent) apply for each dynamic variable (in their definition order) its update expression. We define 4 new variables:

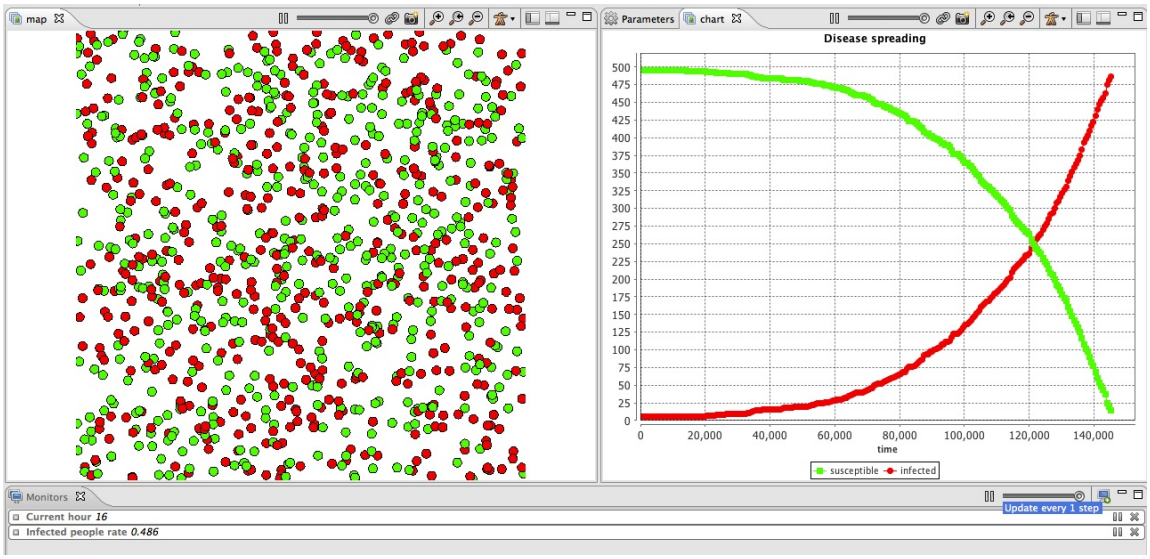


Figure 131.1: images/Incremental_model2.jpg

- **current hour** (int) : current simulation step (**cycle**) / 60 mod 24
- **nb_people_infected** (int): nb of people with is_infected = true (use of the **list count condition** operator that count the number of elements of the list for which the condition is true)
- **nb_people_not_infected** (int): nb_people - nb of people infected
- **infected_rate** (float): nb of people infected / nb of people

```
global{
    ...
        int current_hour update: (cycle / 60) mod 24;
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;
    ...
}
```

stopping condition

We add a new reflex that stops the simulation if the **infected_rate** is equal to 1. To stop the simulation, we use the `pause` action.

```
global {
    ...
        reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}
```

monitor

A monitor allows to follow the value of an arbitrary expression in GAML. It has to be defined in an output section. A monitor is defined as follows:

```
      monitor monitor_name value: an_expression refresh:every(nb_steps)
    ;
```

With:

- value: mandatory, its value will be displayed in the monitor.
- refresh: bool, optional: if the expression is true, compute (default is true).

In this model, we define 2 monitors to follow the value of the variable **current_hour** and **infected_rate**:

```
experiment main_experiment type:gui{
    ...
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        ...
    }
}
```

chart

GAMA can display various chart types:

- Time series
- Pie charts
- Histograms

A chart must be defined in a display: it behaves exactly like any other layer. Definition of a chart :

```
chart chart_name type: chart_type  {
     [data]
}
```

The data to draw are defined inside the chart block:

```
data data_legend value: data_value
```

We add a new display called **chart** refresh every 10 simulation steps. Inside this display, we define a chart of type *series*:

- "Species evolution"; background : white; size : {1, 0.5}; position : {0, 0}
  - data1: susceptible; color : green
  - data2: infected; color : red

```
experiment main_experiment type:gui{
    ...
    output {
        ...
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Complete Model

```
model SI_city

global{
    int nb_people <- 500;
    float step <- 1 #minutes;
    geometry shape<-envelope(square(500 #m));
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    int current_hour update: (cycle / 60) mod 24;
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/length(people);

    init{
        create people number:nb_people {
            speed <- 5.0 #km/#h;
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;
    reflex move{
        do wander;
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
```

```
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map {
            species people aspect:circle;
        }
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Chapter 132

3. Integration of GIS Data

This step Illustrates how load and agentry GIS data.

Formulation

- Load, agentify and display two layers of GIS data (building and road)
- Modify the initialization of the people agents to put them inside buildings

Model Definition

species

We have to define two species of agents: the **building** agents and the **road** ones. These agents will not have a particular behavior, they will just be displayed. We define an aspect for these species. In this model, we want to represent the geometry of the agent, we then use the keyword **draw** that allow to draw a given geometry. In order to draw the geometry of the agent we use the attribute **shape** (which is a built-in attribute of all agents).

```
species building {
    aspect geom {
        draw shape color: #gray;
    }
```

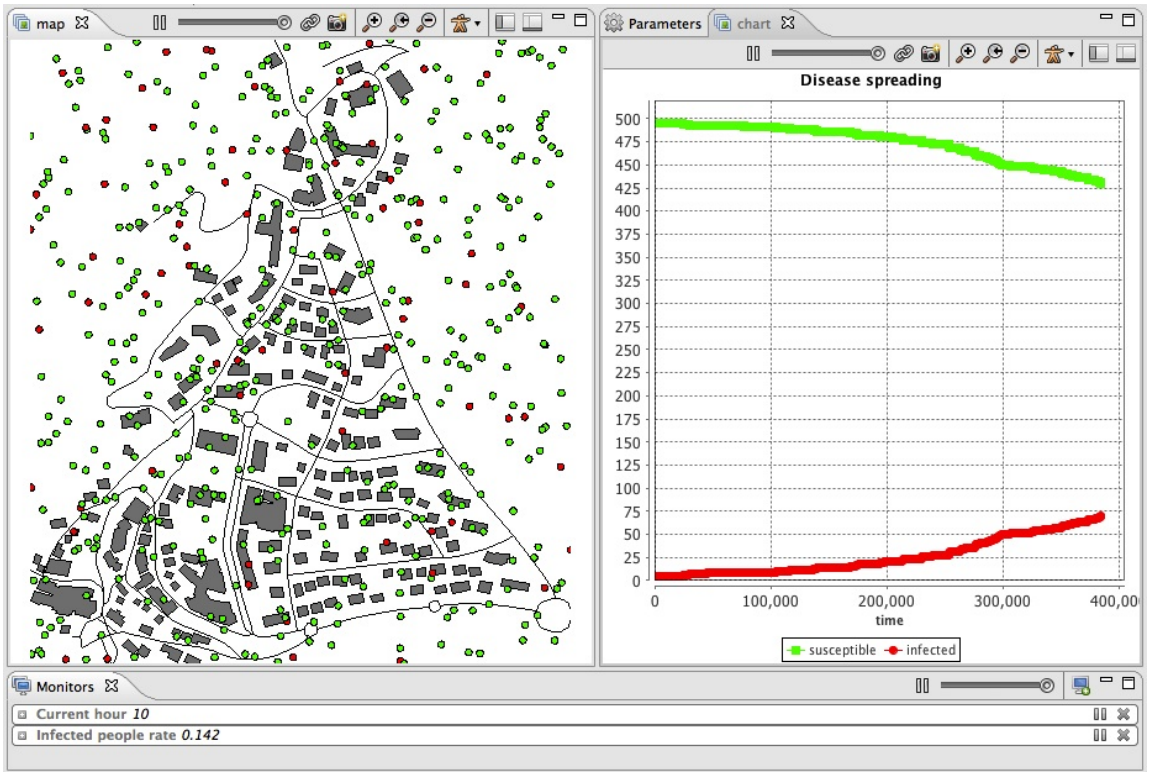


Figure 132.1: resources/images/tutorials/Incremental_model3.jpg

```
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}
```

parameters

GAMA allows to automatically read GIS data that are formatted as shapefiles. In order to let the user chooses his/her shapefiles, we define two parameters. One allowing the user to choose the road shapefile, one allowing him/her to choose the building shapefile.

Definition of the two global variables of type *file* concerning the GIS files:

```
global {
   file shape_file_buildings <- file("../includes/building.shp");
   file shape_file_roads <- file("../includes/road.shp");
}
```

agentification of GIS data

In GAMA, the agentification of GIS data is very straightforward: it only requires to use the **create** command with the **from** facet to pass the shapefile. Each object of the shapefile will be directly used to instantiate an agent of the specified species.

We modify the init section of the global block in order to create road and building agents from the shape files. Then, we define the initial location of people as a point inside one of the building.

```
global {
  ...
       init {
        create road from: roads_shapefile;
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
            building bd <- one_of(building);
            location <- any_location_in(bd);
```

```
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
        }
}
```

We defined here a local variable called **bd** of type building that is a one of the building (randomly chosen). Note that the name of a species can be used to obtain all the agents of this species (here **building** returns the list of all the buildings). The **any_location_in** returns a random point inside a geometry or an agent geometry.

environment

Building a GIS environment in GAMA requires nothing special, just to define the bounds of the environment, i.e. the geometry of the world agent. It is possible to use a shapefile to automatically define it by computing its envelope. In this model, we use the road shapefile to define it.

```
global {
  ...
  geometry shape <- envelope(shape_file_roads);
  ...
}
```

display

We add to the **map** display the road and building agents.

In the **experiment** block:

```
output {
   display map {
    species road aspect:geom;
    species building aspect:geom;
    species people aspect:circle;
    }
    ...
}
```

Complete Model

```
model model3

global {
    int nb_people <- 500;
    float step <- 1 #minutes;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    file roads_shapefile <- file("../includes/road.shp");
    file buildings_shapefile <- file("../includes/building.shp");
    geometry shape <- envelope(roads_shapefile);
    int current_hour update: (cycle / 60) mod 24;
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;

    float infected_rate update: nb_people_infected/length(people);
    init {
        create road from: roads_shapefile;
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
            building bd <- one_of(building);
            location <- any_location_in(bd);
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;

    reflex move{
        do wander;
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
```

```
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map {
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Chapter 133

4. Movement on Graph

This step illustrates how load a graph and use it for the displacement of our species.

Formulation

- definition of a global graph to represent the road network
- definition of a new global variable: **staying_coeff** to represent the fact that people move more near 9h, 12h and 18h
- definition of two new variables for the people agents: **target** and **staying_counter**
- definition of a new reflex for people agents: **stay**
- modification of the **move** reflex of the people agents

Model Definition

global variables

We define two new global variables:

- **road_network** (graph) : represents the graph that will be built from the road network

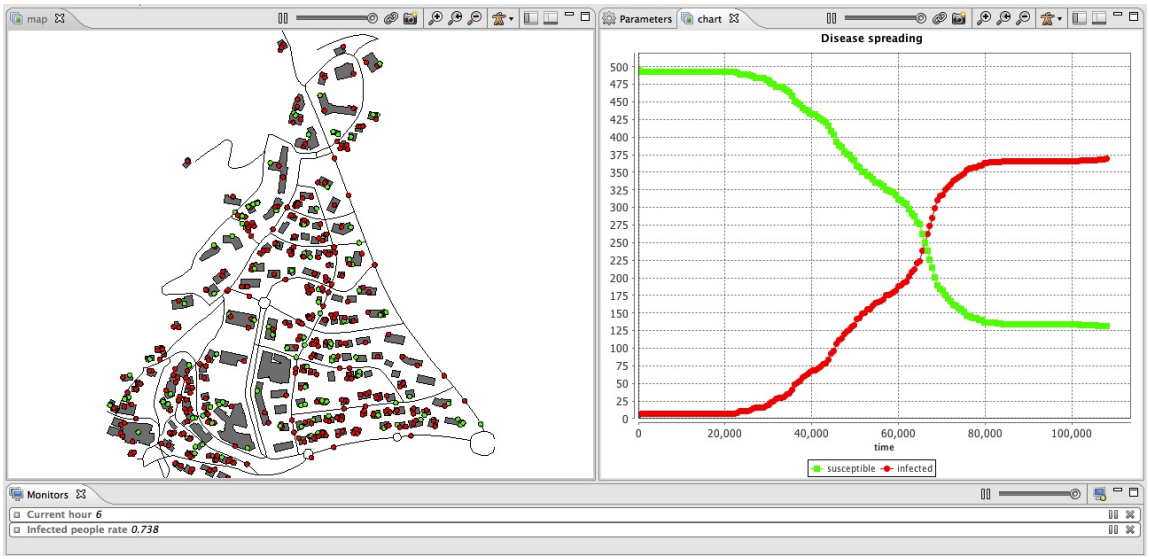


Figure 133.1: images/Incremental_model4.jpg

- **staying_coeff** (float) : represents the fact that people have more chance to move from their current building near 9h (go to work), 12h (lunch time) and 18h (go home). This variable is updated at each simulation step (use of the **update** facet).

```
 global{
    ....
        graph road_network;
    float staying_coeff update: 10.0 ^ (1 + min([abs(current_hour - 9),
    abs(current_hour - 12),
abs(current_hour - 18)]));
    ....
}
```

initialization

We need to compute from the **road** agents, a graph for the moving of the **people** agents. The operator **as_edge_graph** allows to do that. It automatically builds from a set of agents or geometries a graph where the agents are the edges of the graph, a node represent the extremities of the agent geometry. The weight of each edge corresponds to the length of the road.

```
global {
    ...
    init {
        ...
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        ...
    }
}
```

people agent

First, we add two new variables for the people agents:

- **target** (point): the target location that the people want to reach (a point inside a building)
- **staying_counter** (int): the number of cycles since the agent arrived at its building

We define a new reflex called **stay** that is activated when the agent has no target (target = nil), i.e. when the agent is inside a building. This reflex increments the **staying_counter**, then it test the probability to leave that is computed from the **staying_counter** (longer the agent is inside the building, more it has a chance to leave) and the staying_coeff (closer to 9h, 12h and 18h, more the agent has a chance to leave). If the agents decide to leave, it computes a new target as a random point inside one of the buildings (randomly chosen).

```
species people skills:[moving]{
    ...
    reflex stay when: target = nil {
        staying_counter <- staying_counter + 1;
        if flip(staying_counter / staying_coeff) {
            target <- any_location_in (one_of(building));
        }
    }
    ...
}
```

We modify the **move** reflex. Now, this reflex is activated only when the agent has a target (target != nil). In this case the agent moves in direction to its target using

the **goto** action. Note that we specified a graph (**road_network**) to constraint the moving of the agents on the road network with the facet **on**. The agent uses the shortest path (according to the graph) to go to the target point. When the agent arrives at destination (location = location), the target is set to nil (the agent will stop moving) and the staying_counter is set to 0.

```
species people skills:[moving]{
    ...
    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- nil;
            staying_counter <- 0;
        }
    }
}
```

Complete Model

```
model model4

global {
    int nb_people <- 500;
    float step <- 1 #minutes;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    file roads_shapefile <- file("../includes/road.shp");
    file buildings_shapefile <- file("../includes/building.shp");
    geometry shape <- envelope(roads_shapefile);
    int current_hour update: (cycle / 60) mod 24;
    graph road_network;
    float staying_coeff update: 10.0 ^ (1 + min([abs(current_hour - 9),
     abs(current_hour - 12), abs(current_hour - 18)]));
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;

    float infected_rate update: nb_people_infected/length(people);
    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
```

```
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
            building bd <- one_of(building);
            location <- any_location_in(bd);
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;
    point target;
    int staying_counter;
    reflex staying when: target = nil {
        staying_counter <- staying_counter + 1;
        if flip(staying_counter / staying_coeff) {
            target <- any_location_in (one_of(building));
        }
    }

    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- nil;
            staying_counter <- 0;
        }
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
}

species road {
```

```
    aspect geom {
        draw shape color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map {
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Chapter 134

5. Visualizing in 3D

This step Illustrates how to define a 3D display.

Formulation

- add a variable (height: int from 10m to 20m) and modify the aspect of buildings to display them in 3D
- add a variable (display_shape: geometry; shape with a buffer of 2m) and modify the aspect of the roads to display them with this new shape.
- add a new global variable that indicate if it is night or not (bool: night before 7h and after 20h).
- define a new aspect (sphere3D) for people to display them as sphere.
- modify the display to use this new aspect.

Model Definition

building

First, we add a new variable for buildings (**height**) of type float from 10m to 20m. Then we modify the aspect in order to specify a depth for the geometry (using the **depth** facet).

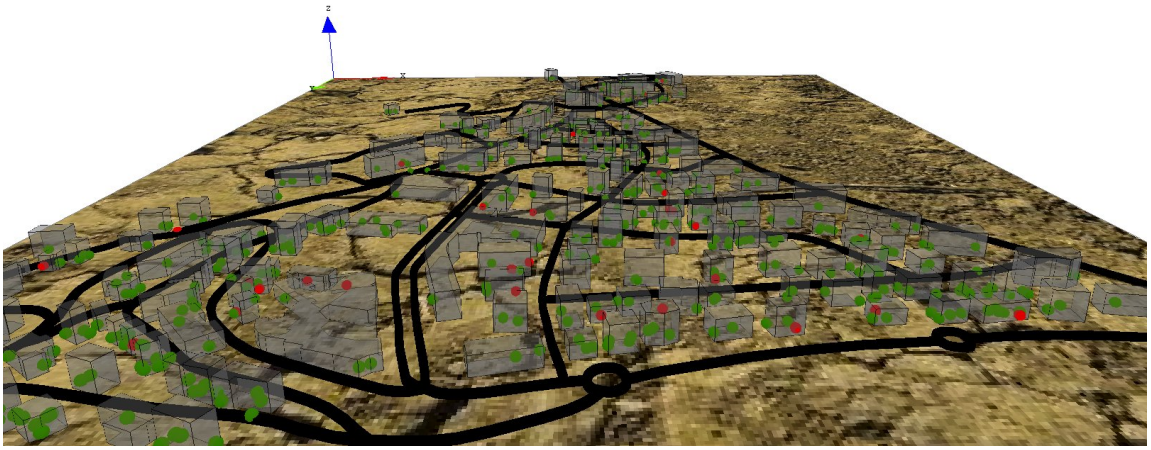


Figure 134.1: images/Incremental_model5.jpg

```
species building {
    float height <- 10#m + rnd(10) #m;
    aspect geom {
        draw shape color: #gray depth: height;
    }
}
```

road

Concerning the road, we add a new variable (**display_shape**) of type geometry that correspond to the shape of the road with a buffer of 2 meters. Then we modify the aspect in order to display this geometry instead of the shape of the agent. In order to avoid "z-fighting" problems, we add a depth to the geometry (of 3 meters).

```
species road {
    geometry display_shape <- shape + 2.0;
    aspect geom {
        draw display_shape color: #black depth: 3.0;
    }
}
```

global variable

We define a new global variable called **is_night** of type *bool* to indicate if it is night or not. This variable will be update at each simulation step and will be *true* if the **current_hour** is lower than 7h or higher than 20h.

```
global{
    ...
        bool is_night <- true update: current_hour < 7 or current_hour
   > 20;
       ...
}
```

people

We define a new aspect for the people agent called **sphere3D**. This aspect draw people agent as a 3m sphere. In order to avoid to cut the sphere in half, we translate the centroid of the drawn sphere to 3m along the z axis.

```
species people skills:[moving]{
    ...
    aspect sphere3D{
        draw sphere(3) at: {location.x,location.y,location.z + 3} color
   :is_infected ? #red : #green;
    }
}
```

display

The element that we have to modify is the display. We change its name to **map_3D** to better reflect its visual aspect.

In order to get a 3D aspect, we specify that this display will be an opengl one. For that, we just have to add the facet **type: opengl**. In addition, to get a different light between night and day : The statement `light` allows us to declare a light. We can change up to 7 lights, called through the facet "id". The default light is a white light, directional, with the id=1. You can set the intensity of the light through the facet "color" (you can pass a color, or an integer between 0 and 255). To have a nice effect night / day, we will set the intensity of the light to 50 during the night, and 255 for the day. To learn more about light, please read this page.

Then, we add a new layer that consists in an image (soil.jpg) by using the **image** statement. In order to see the people inside the building, we add transparency to the building (0.5). The transparency of a layer is a float value between 0 (solid) and 1 (totally transparent). In order to be able to manage this transparency aspect, opengl has to draw the people agents before the building, thus we modify the order of drawing of the different layers (people agents before building agents). At last, we modify the aspect of the people agents by the new one: **sphere3D**.

```
experiment main_experiment type:gui{
    ...
    output {
        ...
        display map_3D type: opengl {
            light 1 color:(is_night ? 50 : 255);
            image "../includes/soil.jpg";
            species road aspect:geom;
            species people aspect:sphere3D;
            species building aspect:geom transparency: 0.5;
        }
        ...
    }
}
```

Complete Model

```
model model5

global {
    int nb_people <- 500;
    float step <- 1 #minutes;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    file roads_shapefile <- file("../includes/road.shp");
    file buildings_shapefile <- file("../includes/building.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;
    int current_hour update: (cycle / 60) mod 24;
    float staying_coeff update: 10.0 ^ (1 + min([abs(current_hour - 9),
     abs(current_hour - 12), abs(current_hour - 18)]));
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
```

```
    int nb_people_not_infected <- nb_people - nb_infected_init update:
    nb_people - nb_people_infected;
    bool is_night <- true update: current_hour < 7 or current_hour >
    20;

    float infected_rate update: nb_people_infected/length(people);
    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
            building bd <- one_of(building);
            location <- any_location_in(bd);
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;
    point target;
    int staying_counter;
    reflex stay when: target = nil {
        staying_counter <- staying_counter + 1;
        if flip(staying_counter / staying_coeff) {
            target <- any_location_in (one_of(building));
        }
    }

    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- nil;
            staying_counter <- 0;
        }
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
```

```
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
    aspect sphere3D{
        draw sphere(3) at: {location.x,location.y,location.z + 3} color
   :is_infected ? #red : #green;
    }
}

species road {
    geometry display_shape <- shape + 2.0;
    aspect geom {
        draw display_shape color: #black depth: 3.0;
    }
}

species building {
    float height <- 10#m + rnd(10) #m;
    aspect geom {
        draw shape color: #gray depth: height;
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map_3D type: opengl {
            light 1 color:(is_night ? 50 : 255);
            image "../includes/soil.jpg";
            species road aspect:geom;
            species people aspect:sphere3D;
            species building aspect:geom transparency: 0.5;
        }
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
```

```
            }
        }
}
```

Chapter 135

6. Multi-Level

This step Illustrates how to define a multi-level model.

Formulation

We propose to let the buildings manage what happens when the people are inside buildings. In this context, we will use the multi-level properties of GAMA: when a people agent will be inside a building, it will be captured by it and its species will be modified. It will be not anymore the people agent that will decide when to leave the building, but the building itself.

We will need to:

- define a micro-species of people inside the building species (**people_in_buildings**)
- define a new variable for the building agent (_people_in**building**)
- define two new behaviors for building: **let_people_leave** and **let_people_enter**
- modify the aspect of the building
- modify some global variables for counting the number of infected people_

Figure 135.1: images/Incremental_model6.jpg

Model Definition

building

First, we define a new species called **people_in_building** inside the **building** species. Thus, a building could have agents of this species as **members** and control them. The **people_in_building** species has for parent the **people** species, which means that a **people_in_building** agent has all the properties, aspect and behaviors of a **people** agent. In our case, we want the once a people agent inside a building, this people agent does nothing. Then, we use the **schedules** facet of the species to remove the **people_in_building** from the scheduler.

```
species building {
    ...
        species people_in_building parent: people schedules: [] {
    }
    ...
}
```

We define a new dynamic variable for building agent: **people_inside** that will correspond to the list of **people_in_building** agents inside the building. Note that

we use the syntax **-> {...}** to make the variable dynamic. However, instead of **update** that allows a variable to be recomputed at each simulation step, the syntax **-> {...}** allows the variable to be recomputed each time it is called (and thus avoid outdated problems). To compute this variable, we use the **members** built-in variable that corresponds to the list of micro-agents captured by the macro-agent.

```
species building {
    ...
        list<people_in_building> people_inside -> {members collect
   people_in_building(each)};
    ...
}
```

We define a first reflex for the buildings that will be activated at each simulation step and that will allow the building to capture all the people that are inside its geometry and that are not moving (target = nil). Capturing agents means putting them inside its **members** list and changing their species: here the **people** agents become **people_in_building** agents.

```
species building {
    ...
      reflex let_people_enter {
        list<people> entering_people <- people inside self where (each.
   target = nil);
        if not (empty (entering_people)) {
            capture entering_people as: people_in_building ;
        }
    }
        ....
}
```

We define a second reflex for the buildings that will be activated at each simulation step and that will allow the building to release some of the **people_in_building** agents. First, it increments the staying counter of all the **people_in_building** agents. Then it builds the list of leaving people by testing the same probability as before for all the **people_in_building** agents. Finally, if this list is not empty, it releases them as people agents (and gives them a new target point).

```
species building {
    ...
       reflex let_people_leave  {
        ask members as: people_in_building{
            staying_counter <- staying_counter + 1;
        }
```

```
        list<people_in_building> leaving_people <- list<
    people_in_building>(members where (flip(people_in_building(each).
    staying_counter / staying_coeff)));
        if not (empty (leaving_people)) {
            release leaving_people as: people in: world returns:
    released_people;
            ask released_people {
                target <- any_location_in (one_of(building));
            }
        }
    }
        ....
}
```

At last, we refine the aspect of the buildings: if there is not people inside the building, we draw it with gray color. If the number of **people_in_building** infected is higher than the number of **people_in_building** not infected, we draw it in red; otherwise in green.

```
species building {
    ...
        aspect geom {
        int nbI <- members count people_in_building(each).is_infected;
        int nbT <- length(members);
        draw shape color:nbT = 0 ? #gray : (float(nbI)/nbT > 0.5 ? #red
    : #green) depth: height;
    }
}
```

global variables

In order to take into account the people that are inside the buildings for the computation of **nb_people_infected**, we first build the list of **people_in_building**. As **people_in_building** is a macro species of **building**, we cannot compute it directly like for the other species, we then aggregate all the list **people_inside** of all building in a single list (**list_people_in_buildings**). Then, we compute the number of infected people as the number of people infected outside the building + the number of people infected inside them.

```
global  {
    ...
```

```
    list<people_in_building> list_people_in_buildings update: (building
     accumulate each.people_inside) where (not dead(each));
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected) + (empty(list_people_in_buildings) ? 0 :
   list_people_in_buildings count (each.is_infected));
        ...
}
```

Complete Model

```
global  {
    int nb_people <- 500;
    float step <- 1 #minutes;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    file roads_shapefile <- file("../includes/road.shp");
    file buildings_shapefile <- file("../includes/building.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;
    int current_hour update: (cycle / 60) mod 24;
    float staying_coeff update: 10.0 ^ (1 + min([abs(current_hour - 9),
     abs(current_hour - 12), abs(current_hour - 18)]));

    list<people_in_building> list_people_in_buildings update: (building
     accumulate each.people_inside) where (not dead(each));
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected) + (empty(list_people_in_buildings) ? 0 :
   list_people_in_buildings count (each.is_infected));

    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    bool is_night <- true update: current_hour < 7 or current_hour >
   20;

    float infected_rate update: nb_people_infected/nb_people;
    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
```

```
            building bd <- one_of(building);
            location <- any_location_in(bd);
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }

    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;
    point target;
    int staying_counter;

    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- any_location_in (one_of(building));
            target <- nil;
            staying_counter <- 0;
        }
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
    aspect sphere3D{
        draw sphere(3) at: {location.x,location.y,location.z + 3} color
    :is_infected ? #red : #green;
    }
}

species road {
    geometry display_shape <- shape + 2.0;
    aspect geom {
        draw display_shape color: #black depth: 3.0;
```

```
    }
}

species building {
    float height <- 10#m + rnd(10) #m;
    list<people_in_building> people_inside -> {members collect
   people_in_building(each)};

    aspect geom {
        int nbI <- members count people_in_building(each).is_infected;
        int nbT <- length(members);
        draw shape color:nbT = 0 ? #gray : (float(nbI)/nbT > 0.5 ? #red
    : #green) depth: height;
    }

    species people_in_building parent: people schedules: [] {
        aspect circle{}
        aspect sphere3D{}
    }

    species people_in_2 parent: people schedules: [] {
        aspect circle{}
        aspect sphere3D{}
    }

    reflex let_people_leave  {
        ask members as: people_in_building{
            staying_counter <- staying_counter + 1;
        }
        list<people_in_building> leaving_people <- list<
   people_in_building>(members where (flip(people_in_building(each).
   staying_counter / staying_coeff)));
        if not (empty (leaving_people)) {
            release leaving_people as: people in: world returns:
   released_people;
            ask released_people {
                target <- any_location_in (one_of(building));
            }
        }
    }
    reflex let_people_enter {
        list<people> entering_people <- people inside self where (each.
   target = nil);
        if not (empty (entering_people)) {
            capture entering_people as: people_in_building ;
        }
```

```
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map_3D type: opengl {
            light 1 color:(is_night ? 50 : 255);
            image "../includes/soil.jpg";
            species road aspect:geom;
            species people aspect:sphere3D;
            species building aspect:geom transparency: 0.5;
        }
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Chapter 136

7. Differential Equations

This step Illustrates how to use differential equations.

Formulation

We are interested by the spreading of the disease inside the buildings. In order to model it, we will use differential equations. So, we will need to:

- add two global variables to define the building epidemic properties (**beta** and **h**).
- add new variables for the buildings (**I**, **S**, **T**, **t**, **I_to_1**);
- define differential equations for disease spreading inside buildings
- add a behavior for buildings for the spreading of the disease.

Model Definition

global variables

We define two new global variables that will be used disease spreading dynamic inside the buildings.

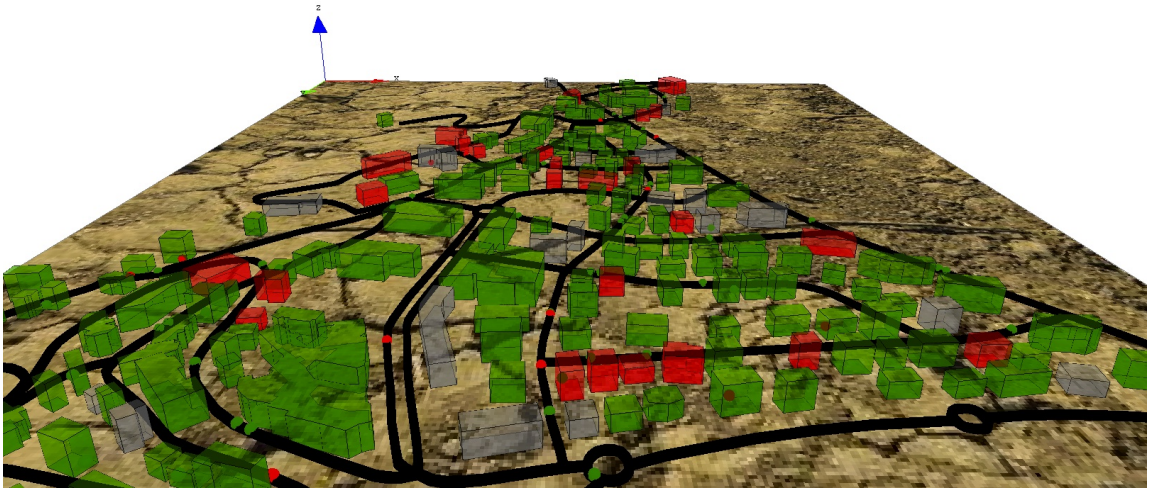


Figure 136.1: images/incremental_model.jpg

```
global  {
    ...
    float beta <- 0.01;
    float h<-0.1;
        ...
}
```

building

In order to define the disease spreading dynamic, we define several variables that will be used by the differential equations:

- **I** : float, number of people infected in the building
- **S** : float, number of people not infected in the building
- **T** : float, total number of people in the building
- **t** : float, current time
- **I_to1** : float, the remaining number of people infected (float number lower between 0 and 1 according to the differential equations).

```
species building {
    ....
    float I;
    float S;
    float T;
    float t;
    float I_to1;
        ...
}
```

Then, we define the differential equations that will use for the disease spreading dynamic. Note that to define a differential equation system we use the block **equation** + name. These equations are the classic ones used by SI mathematical models.

```
species building {
    ....
    equation SI{
        diff(S,t) = (- beta * S * I / T) ;
        diff(I,t) = (  beta * S * I / T) ;
    }
       ...
}
```

At last, we define a new reflex for the building called **epidemic** that will be activated only when there is someone inside the building. This reflex first computes the number of people inside the building (**T**), then the number of not infected people (**S**) and finally the number of infected ones (**I**).

If there is at least one people infected and one people not infected, the differential equations is integrated (according to the discretisation step value **h**) with the method Runge-Kutta 4 to compute the new value of infected people. We then sum the old value of **I_to_1** with the number of people newly infected (this value is a float and not an integer). Finally, we cast this value as an integer, ask the corresponding number of not infected people to become infected, and decrement this integer value to **I_to1**.

```
species building {
    ....
    reflex epidemic when: not empty(members){
        T <- float(length(members));
        list<people_in_building> S_members <- list<people_in_building>(
   members where not (people_in_building(each).is_infected));
                S <- float(length(S_members));
```

```
                I <- T-S;
                float I0 <- I;
                if (I > 0 and S > 0) {
                    solve SI method: "rk4" step: h;
                    I_to1 <- I_to1 + (I - I0);
                    int I_int <- min([int(S),int(I_to1)]);
                    I_to1 <- I_to1 - I_int;
                    ask(I_int among S_members){
                      is_infected <- true;
                    }
                }
        }
        ...
}
```

Complete Model

```
model model7
global  {
    int nb_people <- 500;
    float step <- 1 #minutes;
    float infection_distance <- 2.0 #m;
    float proba_infection <- 0.05;
    int nb_infected_init <- 5;
    file roads_shapefile <- file("../includes/road.shp");
    file buildings_shapefile <- file("../includes/building.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;
    int current_hour update: (cycle / 60) mod 24;
    float staying_coeff update: 10.0 ^ (1 + min([abs(current_hour - 9),
    abs(current_hour - 12), abs(current_hour - 18)]));
    float beta <- 0.01;
    float h<-0.1;
    list<people_in_building> list_people_in_buildings update: (building
    accumulate each.people_inside) where (not dead(each));
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected) + (empty(list_people_in_buildings) ? 0 :
   list_people_in_buildings count (each.is_infected));

    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    bool is_night <- true update: current_hour < 7 or current_hour >
   20;
```

```
    float infected_rate update: nb_people_infected/nb_people;
    init {
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            speed <- 5.0 #km/#h;
            building bd <- one_of(building);
            location <- any_location_in(bd);
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }

    }
    reflex end_simulation when: infected_rate = 1.0 {
        do pause;
    }
}

species people skills:[moving]{
    bool is_infected <- false;
    point target;
    int staying_counter;

    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- any_location_in (one_of(building));
            target <- nil;
            staying_counter <- 0;
        }
    }
    reflex infect when: is_infected{
        ask people at_distance infection_distance {
            if flip(proba_infection) {
                is_infected <- true;
            }
        }
    }
    aspect circle{
        draw circle(5) color:is_infected ? #red : #green;
    }
    aspect sphere3D{
```

```
        draw sphere(3) at: {location.x,location.y,location.z + 3} color
    :is_infected ? #red : #green;
    }
}

species road {
    geometry display_shape <- shape + 2.0;
    aspect geom {
        draw display_shape color: #black depth: 3.0;
    }
}

species building {
    float height <- 10#m + rnd(10) #m;
    list<people_in_building> people_inside -> {members collect
   people_in_building(each)};
    float I;
    float S;
    float T;
    float t;
    float I_to1;

    aspect geom {
        int nbI <- members count people_in_building(each).is_infected;
        int nbT <- length(members);
        draw shape color:nbT = 0 ? #gray : (float(nbI)/nbT > 0.5 ? #red
    : #green) depth: height;
    }

    species people_in_building parent: people schedules: [] {
    }

    reflex let_people_leave  {
        ask members as: people_in_building{
            staying_counter <- staying_counter + 1;
        }
        list<people_in_building> leaving_people <- list<
   people_in_building>(members where (flip(people_in_building(each).
   staying_counter / staying_coeff)));
        if not (empty (leaving_people)) {
            release leaving_people as: people in: world returns:
   released_people;
            ask released_people {
                target <- any_location_in (one_of(building));
            }
        }
```

```
    }
    reflex let_people_enter {
        list<people> entering_people <- people inside self where (each.
   target = nil);
        if not (empty (entering_people)) {
            capture entering_people as: people_in_building ;
        }
    }
    equation SI{
        diff(S,t) = (- beta * S * I / T) ;
        diff(I,t) = (  beta * S * I / T) ;
    }

    reflex epidemic when: not empty(members){
        T <- float(length(members));
        list<people_in_building> S_members <- list<people_in_building>(
   members where not (people_in_building(each).is_infected));
        S <- float(length(S_members));
        I <- T-S;
        float I0 <- I;
        if (I > 0 and S > 0) {
            solve SI method: "rk4" step: h;
            I_to1 <- I_to1 + (I - I0);
            int I_int <- min([int(S),int(I_to1)]);
            I_to1 <- I_to1 - I_int;
            ask(I_int among S_members){
                is_infected <- true;
            }
        }
    }
}

experiment main_experiment type:gui{
    parameter "Infection distance" var: infection_distance;
    parameter "Proba infection" var: proba_infection min: 0.0 max: 1.0;
    parameter "Nb people infected at init" var: nb_infected_init ;
    output {
        monitor "Current hour" value: current_hour;
        monitor "Infected people rate" value: infected_rate;
        display map_3D type: opengl {
            light 1 color:(is_night ? 50 : 255);
            image "../includes/soil.jpg";
            species road aspect:geom;
            species people aspect:sphere3D;
            species building aspect:geom transparency: 0.5;
        }
```

```
        display chart refresh:every(10) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Chapter 137

Luneray's flu

This tutorial has for goal to introduce how to build a model with GAMA and to use GIS data and graphs. In particular, this tutorial shows how to write a simple GAMA model (the structure of a model, the notion of species...) load gis data, to agentify them and to use a network of polylines to constraint the movement of agents. All the files related to this tutorial (shapefiles and models) are available here.

The importation of models is described here

Model Overview

The model built in this tutorial concerns the spreading of a flu in the city of Luneray (Normandie, France).

Two layers of GIS data are used: a road layer (polylines) and a building layer (polygons). In this model, people agents are moving from building to building using the road network. Each infected people can infect the neighbor people.

Some data collected concerning Luneray and the disease:

- Number of inhabitants: 2147 (source : wikipedia)
- Mean speed of the inhabitants (while moving on the road) : 2-5 km/h
- The disease - non lethal - is spreading (by air) from people to people
- Time to cure the disease: more than 100 days
- Infection distance: 10 meters

Figure 137.1: images/Luneray.jpg

- Infection probability (when two people are at infection distance) : 0.05/ 5 minutes

From the data collected, we made some modeling choice:

- Simulation step: 5 minutes
- People are moving on the roads from building to building
- People use the shortest path to move between buildings
- All people move at constant speed
- Each time, people arrived at a building they are staying a certain time
- Infected people are never cured

Step List

This tutorial is composed of 5 steps corresponding to 5 models. For each step we present its purpose, an explicit formulation and the corresponding GAML code.

1. Creation of a first basic disease spreading model
2. Definition of monitors and chart outputs
3. Importation of GIS data
4. Use of a graph to constraint the movements of people
5. Definition of 3D displays

Chapter 138

1. Creation of a first basic disease spreading model

This first step illustrates how to create simple agents and make them move in their environment.

Formulation

- Set the time duration of a time step to 5 minutes
- Define the people species with a moving skill
- Define the move reflex that allow the people agent to move randomly and the infect reflex that allows them to infect other people agents.
- Define the aspect of the people species
- Add the people species to a display

Model Definition

Project and model

The first step of this tutorial consists in launching GAMA and choosing a workspace, then to define a new project or to import the existing one. For people that do not want to re-write all the models but just to follow the model construction, they can

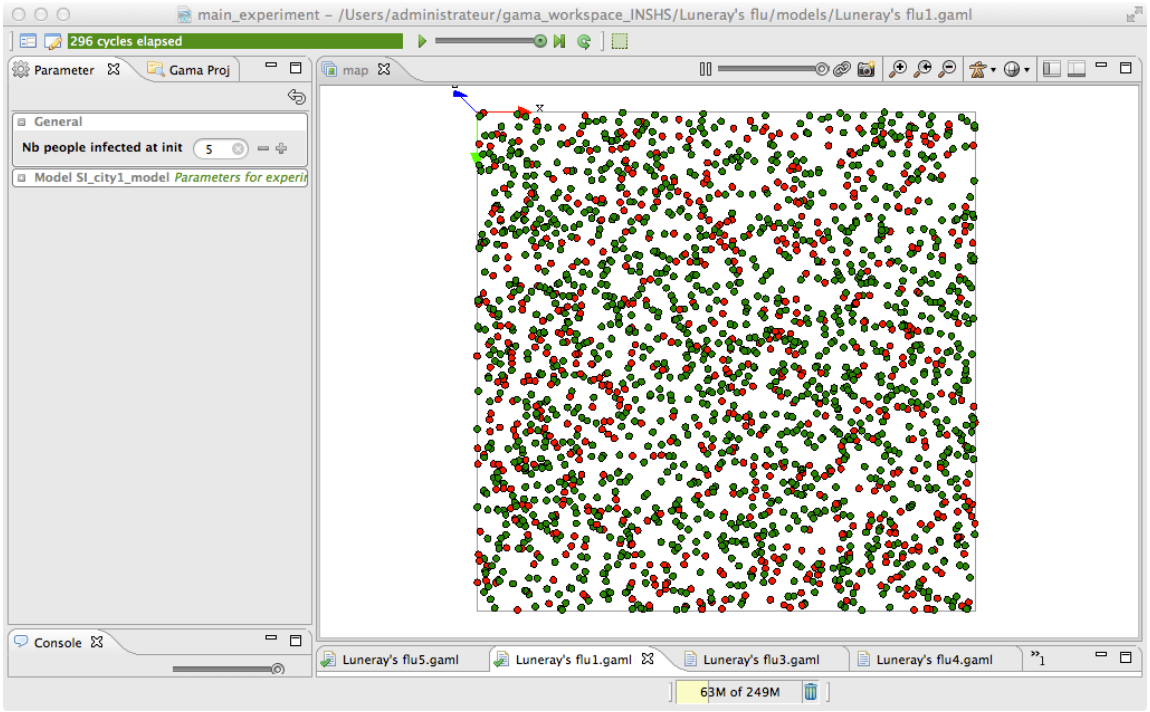


Figure 138.1: images/Luneray.jpg

just download the model project here and the follow this procedure to import it into GAMA. For the other, the project and model creation procedures are detailed here.

Note that the concepts of workspace and projects are explained here.

model structure

A GAMA model is composed of three type of sections:

- **global** : this section, that is unique, defines the "world" agent, a special agent of a GAMA model. It represents all that is global to the model: dynamics, variables, actions. In addition, it allows to initialize the simulation (init block).
- **species** and **grid**: these sections define the species of agents composing the model. Grid are defined in the following model step "vegetation dynamic";
- **experiment** : these sections define a context of execution of the simulations. In particular, it defines the input (parameters) and output (displays, files. . .) of a model.

More details about the different sections of a GAMA model can be found here.

species

A species represents a «prototype» of agents: it defines their common properties.

Three different elements can be defined in a species:

- the internal state of its agents (attributes)
- their behavior
- how they are displayed (aspects)

In our model, we define a people species:

```
species people {

}
```

In addition, we want to add a new capability to our agent: the possibility to move randomly. For that, we add a specific skill to our people agents. A skill is a built-in module that provide the modeler a self-contain and relevant set of actions and variables. The moving provides the agents with several attributes and actions related to movement.

```
species people skills: [moving]{
    ...
}
```

Internal state

An attribute is defined as follows: type of the attribute and name. Numerous types of attributes are available: *int (integer), float (floating point number), string, bool (boolean, true or false), point (coordinates), list, pair, map, file, matrix, espèce d'agents, rgb (color), graph, path...*

- Optional facets: <- (initial value), update (value recomputed at each step of the simulation), function:{..} (value computed each time the variable is used), min, max

In addition to the attributes the modeler explicitly defines, species "inherits" other attributes called "built-in" variables:

- A name (*name*): the identifier of the species
- A shape (*shape*): the default shape of the agents to be construct after the species. It can be *a point, a polygon, etc.*
- A location (*location*) : the centroid of its shape.

In our model, we define 2 new attribute to our people agents:

- **speed** of type float, with for initial value: a random value between 2 and 5 km/h
- **is_infected** of type bool, with for initial value: false

```
species people skills:[moving]{
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;
}
```

Note we use the rnd operator to define a random value between 2 and 5 for the speed. In addition, we precise a unit for the speed value by using the # symbol. For more details about units, see here.

Behavior

GAMA proposes several ways to define the behavior of a species: dynamic variables (update facet), reflexes. . . .

A reflex is a block of statements (that can be defined in global or any species) that will be automatically executed at each simulation step if its condition is true, it is defined as follows:

```
   reflex reflex_name when: condition {...}
```

The **when** facet is optional: when it is omitted, the reflex is activated at each time step. Note that if several reflexes are defined for a species, the reflexes will be activated following their definition order.

We define a first reflex called **move** that is activated at each simulation step (no condition) and that makes the people move randomly using the wander action from the moving skill.

```
species people skills:[moving]{
    //variable definition
    reflex move{
        do wander;
    }
}
```

We define a second reflex called **infect** that is activated only when the agent is infected (is_infected = true) and that ask all the people at a distance of 10m to test a probability to be infected.

```
species people skills:[moving]{
    //variable definition and move reflex
```

```
    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }
}
```

The ask allows an agent to ask another agents to do something (i.e. to execute a sequence of statements). The at_distance operator allows to get the list of agents (here of people agents) that are located at a distance lower or equal to the given distance (here 10m). The flip operator allows to test a probability.

Display

An agent aspects have to be defined. An aspect is a way to display the agents of a species : aspect aspect_name {...} In the block of an aspect, it is possible to draw:

- A geometry : for instance, the shape of the agent (but it may be a different one, for instance a disk instead of a complex polygon)
- An image : to draw icons
- A text : to draw a text

In our model, we define an aspect for the people agent called **circle** that draw the agents as a circle of 10m radius with a color that depends on their **is_infected** attribute. If the people agent is infected, it will be draw in red, in green otherwise.

```
species people {
    ...//variable and reflex definition

    aspect circle {
            draw circle(10) color:is_infected ? #red : #green;
        }
    }
}
```

The **?** structure allows to return a different value (here red or green) according to a condition (here is_infected = true).

global section

The global section represents a specific agent, called world. Defining this agent follows the same principle as any agent and is, thus, defined after a species. The world agent represents everything that is global to the model : dynamics, variables... It allows to initialize simulations (init block): the world is always created and initialized first when a simulation is launched (before any other agents). The geometry (shape) of the world agent is by default a square with 100m for side size, but can be redefined if necessary. The *step* attribute of the world agent allows to specify the duration of one simulation step (by default, 1 step = 1 seconde).

global variable

In the current model, we define 4 global attributes:

- nb_people: the number of people that we want to create (init value: 2147)
- nb_infected_init: the number of people infected at the initialization of the simulation (init value: 5)
- step: redefine in order to set the duration of a simulation step to 5 minutes.
- shape: redefine in order to set the geometry of the world to a square of 1500 meters side size.

```
global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    geometry shape<-square(1500 #m);
}
```

Model initialization

The init section of the global block allows to initialize the define what will happen at the initialization of a simulation, for instance to create agents. We use the statement *create* to create agents of a specific species: **create** species_name + :

- number : number of agents to create (int, 1 by default)
- from : GIS file to use to create the agents (optional, string or file)

- returns: list of created agents (list)

For our model, we define the init block in order to create *nb_people people* agents and ask *nb_infected_init* of them to be infected:

```
global {
    // world variable definition
    init{
        create people number:nb_people;
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}
```

experiment

An experiment block defines how a model can be simulated (executed). Several experiments can be defined for a given model. They are defined using : **experiment** exp_name type: gui/batch {`[input] [output]`}

- gui : experiment with a graphical interface, which displays its input parameters and outputs.
- batch : Allows to setup a series of simulations (w/o graphical interface).

In our model, we define a gui experiment called *main_experiment* :

```
experiment main_experiment type: gui {
}
```

input

Experiments can define (input) parameters. A parameter definition allows to make the value of a global variable definable by the user through the graphic interface.

A parameter is defined as follows:

parameter title var: global_var category: cat;

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

Note that the init, min and max values can be defined in the global variable definition.

In our model, we define one parameter:

- "Nb people infected at init" that will define the value of the global variable *nb_infected_init* with a min value of 1 and a max value of 2147 (the number of people living in Luneray).

```
experiment main_experiment type:gui{
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
    }
}
```

output

Output blocks are defined in an experiment and define how to visualize a simulation (with one or more display blocks that define separate windows). Each display can be refreshed independently by defining the facet **refresh_every:** nb (int) (the display will be refreshed every nb steps of the simulation).

Each display can include different layers (like in a GIS):

- Agents lists : **agents** layer_name value: agents_list aspect: my_aspect;
- Agents species : **species** my_species aspect: my_aspect
- Images: **image** layer_name file: image_file;
- Texts : **texte** layer_name value: my_text;
- Charts : see later.

Note that it is possible to define a opengl display (for 3D display or just to optimize the display) by using the facet **type: opengl**.

```
output {
    display map {
        species people aspect:circle;
    }
}
```

Complete Model

```
model model1

global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    geometry shape<-square(1500 #m);

    init{
        create people number:nb_people;
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}

species people skills:[moving]{
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;

    reflex move{
        do wander;
    }
    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }

    aspect circle {
```

```
        draw circle(10) color:is_infected ? #red : #green;
    }
}

experiment main type: gui {
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
        display map {
            species people aspect:circle;
        }
    }
}
```

Next step: Definition of monitors and chart outputs

Chapter 139

2. Definition of monitors and chart outputs

This second step illustrates how to create monitors and charts to follows the evolution of variables and to add an ending condition to the simulation.

Formulation

- Add three new global dynamic variables to follow the evolution of the number of infected people agents, of not infected people agents and of the rate of infected people.
- Define an ending condition for the simulation
- Define a monitor to follow the rate of infected people agents
- Define a chart to follow the rate of infected people agents

Model Definition

global section

global variables

GAMA offers the possibility to define dynamic variable that will be recomputed at each simulation step by using the *update* facet when defining a variable. When an

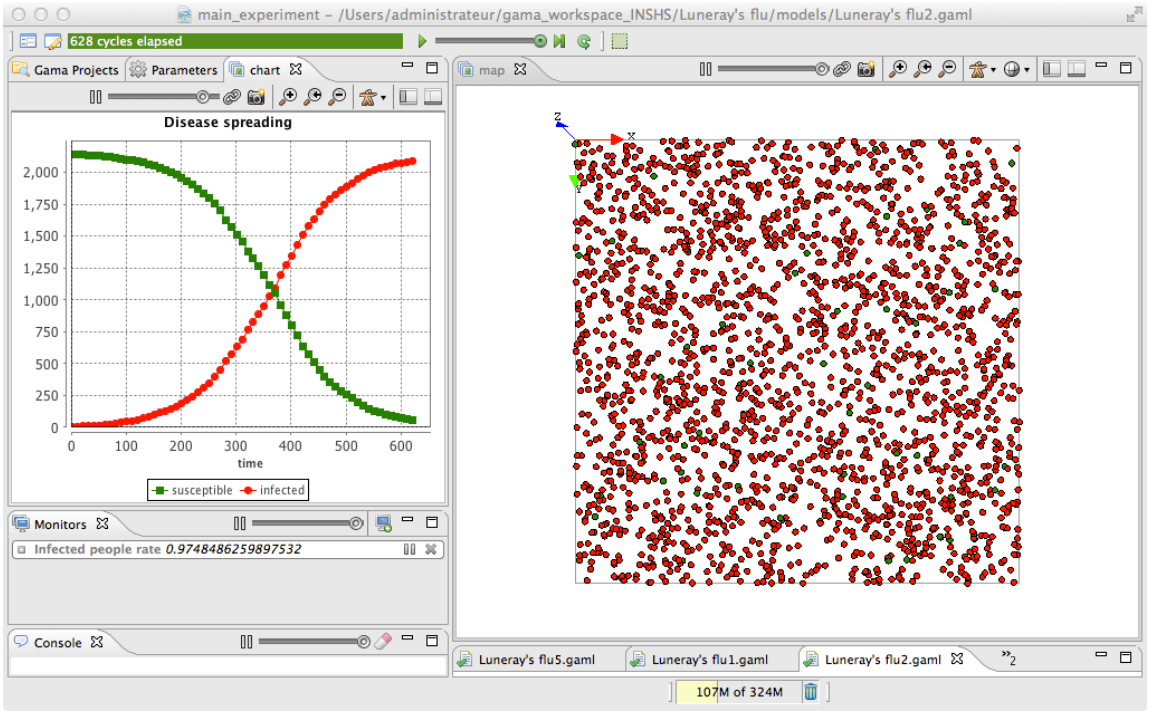


Figure 139.1: images/Luneray.jpg

agent is activated, first, it recomputes each of its variables with a update facet (in their definition order), then it activates each of its reflexes (in their definition order).

To better follow the evolution of sick people, we add three new global variables to the model:

- nb_people_infected of type *int* with *nb_infected_init* as init value and updated at each simulation step by the number of infected people
- nb_people_not_infected of type *int* with *(nb_people - nb_infected_init)* as init value and updated at each simulation step by the number of not infected people
- infected_rate of type *float* updated at each simulation step by the number of infected people divided by the number of people.

```
global{
    //... other attributes
    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;
    //... init
}
```

We used the count operator that allows to count the number of elements of a list for which the left condition is true. The keyword *each* represents each element of the list.

ending condition

The simplest way to add an ending condition to a model is to add a global reflex that is activated at the end of the simulation that will pause the simulation (use of the *pause* global action).

In our model, we add a new reflex called *end_simulation* that will be activated when the infected rate is 1.0 (i.e. all the people agents are infected) and that will apply the *pause* action.

```
global {
    //.. variable and init definition

    reflex end_simulation when: infected_rate = 1.0 {
```

```
        do pause;
    }
}
```

experiment

monitor

GAMA provides modelers with the possibility to define monitors. A monitor allows to follow the value of an arbitrary expression in GAML. It will appear, in the User Interface, in a small window on its own and be recomputed every time step (or according to its 'refresh' facet).

Definition of a monitor:

- *value*: mandatory, the expression whose value will be displayed by the monitor.
- *refresh*: bool, optional : if the expression is true, compute (default is true).

For our model, we define a monitor to follow the value of the *infected_rate* variable:

```
experiment main_experiment type:gui{
    //...parameters
    output {
        monitor "Infected people rate" value: infected_rate;

                //...display
    }
}
```

Chart

In GAMA, charts are considered as a display layer. GAMA can display 3 main types of charts using the *type* facet:

- `histogram`
- `pie`
- `series`/`xy`/`scatter`: both display series with lines or dots, with 3 subtypes :
  - `series`: to display the evolution of one/several variable (vs time or not).

- `xy`: to specify both x and y value. To allow stacked plots, only one y value for each x value.
- `scatter`: free x and y values for each serie.

In our model, we define a new display called _ chart_display_ that will be refresh every 10 simulation steps. In this display, we add a series charts with 2 layers of data:

- susceptible: the number of people that are not infected (in green)
- infected: the number of people that are infected (in red)

```
experiment main_experiment type:gui{
    //...parameters
    output {
        //...display and monitors

        display chart_display refresh:every(10 #cycle) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Complete Model

```
model model2

global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    geometry shape<-square(1500 #m);

    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;
```

```
    init{
        create people number:nb_people;
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}

species people skills:[moving]{
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;

    reflex move{
        do wander;
    }
    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }

    aspect circle {
        draw circle(10) color:is_infected ? #red : #green;
    }
}

experiment main type: gui {
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
        monitor "Infected people rate" value: infected_rate;

        display map {
            species people aspect:circle;
        }

        display chart_display refresh: every(10 #cycle) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
```

```
            }
        }
}
```

Next step: Importation of GIS data

Chapter 140

3. Importation of GIS data

This third step illustrates how load GIS data and to agentify them.

Formulation

- Define 2 new species that will just be displayed: *road* and *building*.
- Define new global attributes to load GIS data (shape file).
- Use the GIS data to create the *road* and *building* agents.
- Add the *road* and *building* agents to the display.

Model Definition

For this step, you will need to add the shapefiles of the roads and buildings inside the *includes* folder of the project. The shapefiles (and all the other files) can be found here.

species

In this model, we have to define two species of agents: the **road** agents and the **building** ones. These agents will not have a particular behavior, they will just be displayed. For each of this species we define an aspect called *geom*. As we want to represent the geometry of the agent, we then use the keyword **draw** that allow to draw

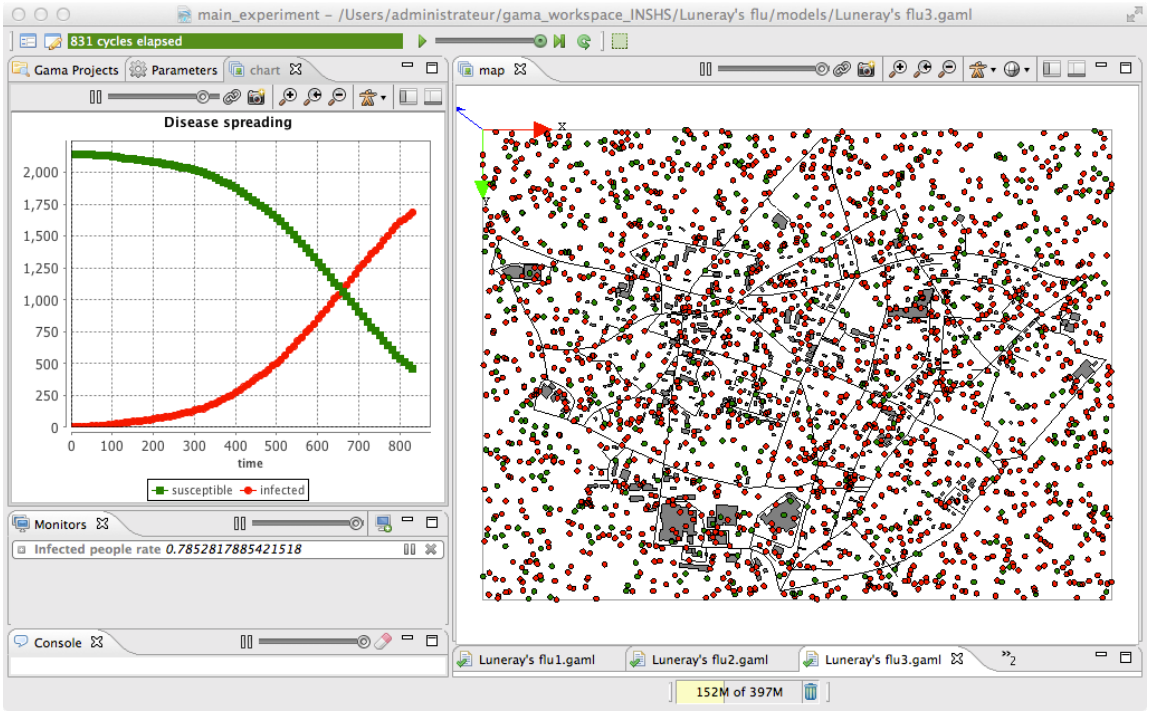


Figure 140.1: images/luneray3.png

a given geometry. In order to draw the geometry of the agent we use the attribute **shape** (which is a built-in attribute of all agents). The road will be displayed in black and the building in gray.

```
species road {
    aspect geom {
        draw shape color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
}
```

global section

global variables

GAMA allows to automatically read GIS data that are formatted as shape files (or as OSM file). In our model, we define 2 shapefiles: one corresponding to the roads and the other ones to the buildings. Note that GAMA is able to manage the projection of the GIS data. In order to set the right size (and position) of the world geometry, we define its value as the envelope of the road shapefile (and no more a square of 1500 meters).

```
global{
    //... other attributes
    file roads_shapefile <- file("../includes/routes.shp");
    file buildings_shapefile <- file("../includes/batiments.shp");
    geometry shape <- envelope(roads_shapefile);
    //... init
}
```

agentification of GIS data

In GAMA, the agentification of GIS data is very straightforward: it only requires to use the **create** command with the **from** facet to pass the shapefile. Each object of the shapefile will be directly used to instantiate an agent of the specified species.

The reading of an attribute in a shapefile is also very simple. It only requires to use the **with** facet: the argument of this facet is a dictionary of which the keys are the names of the agent attributes and the value the **read** command followed by the name of the shapefile attribute.

In our model, we modify the init section in order to first create the *road* agents from the road shapefile, and the *building* agents from the building shapefile. Then, when creating people agents, we choose for them a random location inside a random building. Note that it is possible to execute a sequence of statements at the creation of agents by using a block ({...}) rather than a simple line (;) when using the *create* statement.

```
global {
    // world variable definition

    init{
        create road from: roads_shapefile;
        create building from: buildings_shapefile;
        create people number:nb_people {
            location <- any_location_in(one_of(building));      }
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}
```

We used here the one_of operator that returns a random element from a list and the any_location_in operator that returns a random location inside a geometry.

experiment

Output

In the *map* display, we add the *road* and *building* species with their *geom* aspect just before the *people* species (in order to draw the people agents on the top of the roads and buildings).

```
experiment main_experiment type: gui {
    ... //parameter definition

    output {
```

```
    ... //monitor definition

        display map type: opengl{
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }
        ... //chart display definition
    }
}
```

Complete Model

```
model model3

global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    file roads_shapefile <- file("../includes/roads.shp");
    file buildings_shapefile <- file("../includes/buildings.shp");
    geometry shape <- envelope(roads_shapefile);

    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;


    init{
        create road from: roads_shapefile;
        create building from: buildings_shapefile;
        create people number:nb_people {
            location <- any_location_in(one_of(building));
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}

species people skills:[moving]{
```

```
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;

    reflex move{
        do wander;
    }

    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }

    aspect circle {
        draw circle(10) color:is_infected ? #red : #green;
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
}

experiment main type: gui {
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
        monitor "Infected people rate" value: infected_rate;

        display map {
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }

        display chart_display refresh: every(10 #cycle) {
```

```
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Next step: Use of a graph to constraint the movements of people

Chapter 141

4. Use of a graph to constraint the movements of people

This fourth step illustrates how to use a graph to constraint the movements of agents

Formulation

- Define a new global variable: the road network (graph).
- Build the road network graph from the road agents
- Add new attribute to the people agents (target)
- Define a new reflex for people agents: stay.
- Modify the move reflex of the people agents.

Model Definition

global section

global variables

In this model, we want that people agents move from buildings to buildings by using the shortest path in the road network. In order to compute this shortest path, we need to use a graph structure.

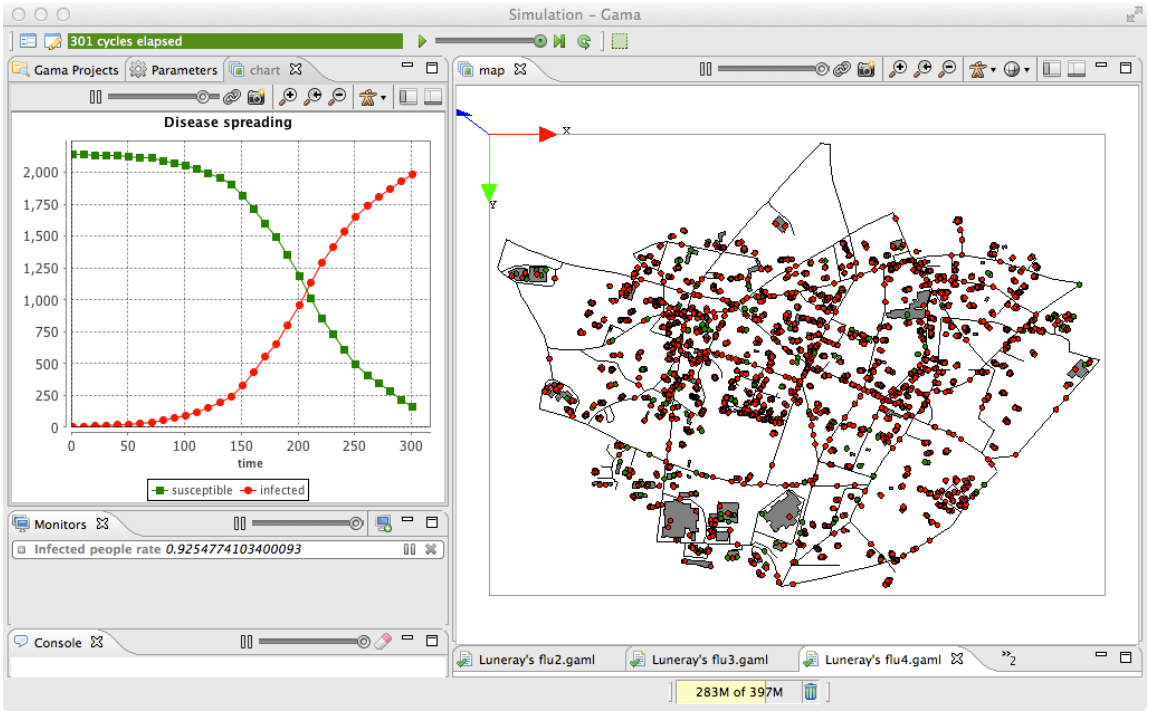


Figure 141.1: images/luneray4.png

We thus define a new global variable called *road_network* of type *graph* that will represent the road network.

```
global{
    //... other attributes
    graph road_network;

    //... init
}
```

In order to compute the graph from the road network, we use, just after having creating the road agents, the as_edge_graph operator. This operator automatically built a graph from a set of polylines. Each extremity point of the lines will become a node in the graph, and each polyline an edge. By default, the graph is not oriented and the weights of the edges are the perimeters of the polylines. It is of course possible to change through the use of some operators.

```
global {
    // world variable definition

    init{
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            location <- any_location_in(one_of(building));
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}
```

people species

We want to modify the behavior of the people agents in order to make them move from buildings to buildings by using the shortest path in the road network.

variables

In order to implement this behavior, we will add two variables to our people species:

- *target* of type *point* that will be the location where the agent wants to go

```
species people skills:[moving]{
    //...the other attributes
    point target;
    //....
}
```

behavior

First, we add a new reflex called *stay* that will be activated when the agent is in a house (i.e. its target is null) and that will define with a probability of 0.05 if the agent has to go or not. If the agent has to go, it will randomly choose a new target (a random location inside one of the building).

```
reflex stay when: target = nil {
    if flip(0.05) {
        target <- any_location_in (one_of(building));
    }
}
```

Then, we modify the *move* reflex. This one will be only activated when the agent will have to move (target not null). Instead of using the *wander* action of the *moving* skill, we use the *goto* one that allows to make an agent moves toward a given target. In addition, it is possible to add a facet *on* to precise on which topology the agent will have to move on. In our case, the topology is the road network. When the agent reach its destination (location = target), it sets its target to null.

```
reflex move when: target != nil{
    do goto target:target on: road_network;
    if (location = target) {
        target <- nil;
    }
}
```

Complete Model

```
model model4

global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    file roads_shapefile <- file("../includes/roads.shp");
    file buildings_shapefile <- file("../includes/buildings.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;


    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;


    init{
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            location <- any_location_in(one_of(building));
        }
        ask nb_infected_init among people {
            is_infected <- true;
        }
    }
}

species people skills:[moving]{
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;
    point target;

    reflex stay when: target = nil {
        if flip(0.05) {
            target <- any_location_in (one_of(building));
        }
    }

    reflex move when: target != nil{
        do goto target:target on: road_network;
```

```
        if (location = target) {
            target <- nil;
        }
    }

    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }

    aspect circle {
        draw circle(10) color:is_infected ? #red : #green;
    }
}

species road {
    aspect geom {
        draw shape color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
}

experiment main type: gui {
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
        monitor "Infected people rate" value: infected_rate;

        display map {
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }

        display chart_display refresh: every(10 #cycle) {
            chart "Disease spreading" type: series {
```

```
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
    }
}
```

Next step: Definition of 3D displays

Chapter 142

5. Definition of 3D displays

This fifth step illustrates how to define 3D displays

Formulation

- Define a new 3D aspect for roads.
- Define a new 3D aspect for buildings
- Define a new 3D aspect for people
- Define a new 3D display

Model Definition

species

We define a new aspect for the road species called *geom3D* that draw the road agent that as a black tube of 2m radius built from its geometry. Note that it is possible to get the list of points composing a geometry by using the *points* variable of the geometry.

```
species road {
    //....
    aspect geom3D {
        draw line(shape.points, 2.0) color: #black;
    }
```

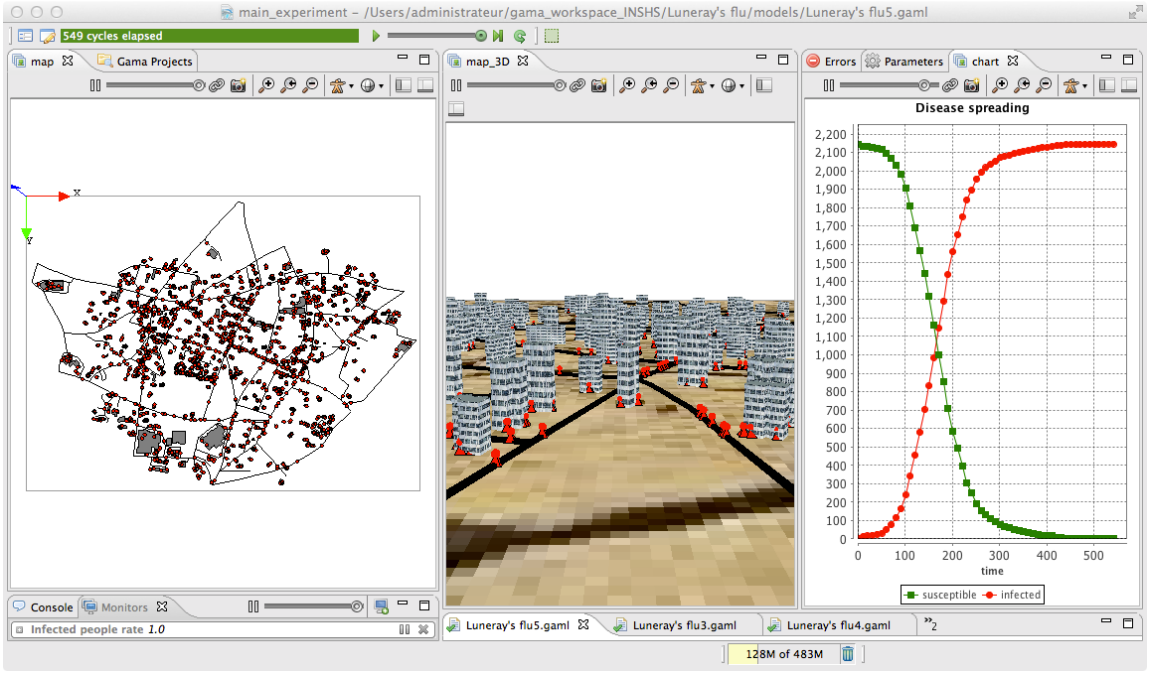


Figure 142.1: images/luneray5.png

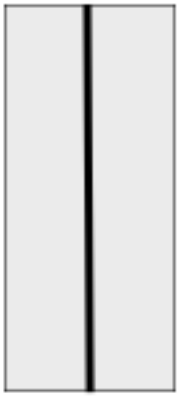


Figure 142.2: images/roads_display.tiff

Figure 142.3: images/buildings_display.tiff

```
}
```

Concerning the building species, we define an aspect called *geom3D* that draws the shape of the building with a depth of 20 meters and with using a texture "texture.jpg" for the face and a texture for the roof "roof_top.png".

```
species building {
    //....
    aspect geom3D {
        draw shape depth: 20 #m border: #black texture:["../includes/
    roof_top.png","../includes/texture.jpg"];
    }
}
```

At last, we define a new aspect called *geom3D* for the people species that displays the agent only if it is on a road (target != nil). In this aspect, we use an obj file that contains a 3D object. The use of the *obj_file* operator allows to apply an initial rotation to an obj file. In our case, we add a rotation of -90° along the x axis. We specify with the *size* facet that we want to draw the 3D object inside a bounding box of 5m. As the location of the 3D object is its centroid and as we want to draw the 3D object on the top of the ground, we use the *at* facet to put an offset along the z axis. We use also the rotate facet to change the orientation of the 3D object according to the heading of the agent. At last, we choose to draw the 3D object in green if the agent is not infected; in red otherwise.

```
species people skills:[moving]{
    //....
    aspect geom3D {
        if target != nil {
```

Figure 142.4: images/people_display.tiff

```
            draw obj_file("../includes/people.obj", 90::{-1,0,0}) size:
    5
            at: location + {0,0,7} rotate: heading - 90 color:
   is_infected ? #red : #green;
        }
    }
}
```

output

We define a new display called *view3D* of type *opengl* with an *ambient_light* of 80. Inside this display, we first draw a background image representing the satellite image of the Luneray. Note that GAMA is able to manage world files to georeferenced images. In our case, as a pgw file exists in the includes folder, the satellite image will be well localized in the display. After drawing the background image, we display first the building species with their geom3D aspect, then the road species with their geom3D aspect and finally the people species with their geom3D aspect. Only the people agents will be redrawn at each simulation step.

```
experiment main_experiment type: gui {
    output {
    // monitor and other displays
        display view3D type: opengl ambient_light: 80 {
            image "../includes/luneray.png" refresh:false;
            species building aspect:geom3D refresh: false;
            species road aspect: geom3D refresh: false;
            species people aspect: geom3D ;
        }
    }
}
```

Complete Model

```
model model4

global {
    int nb_people <- 2147;
    int nb_infected_init <- 5;
    float step <- 5 #mn;
    file roads_shapefile <- file("../includes/roads.shp");
    file buildings_shapefile <- file("../includes/buildings.shp");
    geometry shape <- envelope(roads_shapefile);
    graph road_network;


    int nb_people_infected <- nb_infected_init update: people count (
   each.is_infected);
    int nb_people_not_infected <- nb_people - nb_infected_init update:
   nb_people - nb_people_infected;
    float infected_rate update: nb_people_infected/nb_people;


    init{
        create road from: roads_shapefile;
        road_network <- as_edge_graph(road);
        create building from: buildings_shapefile;
        create people number:nb_people {
            location <- any_location_in(one_of(building));
        }
        ask nb_infected_init among people {
            is_infected <- true;
```

```
        }
    }
}

species people skills:[moving]{
    float speed <- (2 + rnd(3)) #km/#h;
    bool is_infected <- false;
    point target;

    reflex stay when: target = nil {
        if flip(0.05) {
            target <- any_location_in (one_of(building));
        }
    }

    reflex move when: target != nil{
        do goto target:target on: road_network;
        if (location = target) {
            target <- nil;
        }
    }

    reflex infect when: is_infected{
        ask people at_distance 10 #m {
            if flip(0.05) {
                is_infected <- true;
            }
        }
    }

    aspect circle {
        draw circle(10) color:is_infected ? #red : #green;
    }

    aspect geom3D {
        if target != nil {
            draw obj_file("../includes/people.obj", 90::{-1,0,0}) size:
    5
            at: location + {0,0,7} rotate: heading - 90 color:
   is_infected ? #red : #green;
        }
    }

}

species road {
```

```
    aspect geom {
        draw shape color: #black;
    }
    aspect geom3D {
        draw line(shape.points, 2.0) color: #black;
    }
}

species building {
    aspect geom {
        draw shape color: #gray;
    }
    aspect geom3D {
        draw shape depth: 20 #m border: #black texture:["../includes/
   roof_top.png","../includes/texture.jpg"];
    }
}

experiment main type: gui {
    parameter "Nb people infected at init" var: nb_infected_init min: 1
    max: 2147;

    output {
        monitor "Infected people rate" value: infected_rate;

        display map {
            species road aspect:geom;
            species building aspect:geom;
            species people aspect:circle;
        }



        display chart_display refresh: every(10 #cycle) {
            chart "Disease spreading" type: series {
                data "susceptible" value: nb_people_not_infected color:
    #green;
                data "infected" value: nb_people_infected color: #red;
            }
        }
        display view3D type: opengl ambient_light: 80 {
            image "../includes/luneray.png" refresh:false;
            species building aspect:geom3D refresh: false;
            species road aspect: geom3D refresh: false;
            species people aspect: geom3D ;
        }
```

```
    }
}
```

Chapter 143

Co-modeling tutorial

This tutorial introduces the co-modeling feature offered by GAMA.

Model Overview

Step List

This tutorial is composed of 3 steps corresponding to 3 models. For each step we present its purpose, an explicit formulation and the corresponding GAML code.

0. Presentation of the weather and plant growth models
1. Instanciate a weather model and step it
2. Display agents and indicators of the weather model
3. Run 2 co-models in a single model
4. Run 4 plantgrow model coupled with a weather model

Chapter 144

BDI Agents

This tutorial aims at presenting the use of BDI agents in GAMA. In particular, this tutorial shows how to define a BDI agents, then to add social relation between BDI agents, to add emotions and a personality to the agents and finally social norms, obligations and enforcements. These notions come from the BEN architecture, described in details in the page Using BEN architecture.

If you are not familiar with agent-based models or GAMA we advise you to have a look at the prey-predator model first.

Model Overview

The model built in this tutorial concerns gold miners that try to find and sell gold nuggets. More precisely, we consider that several gold mines containing a certain amount of gold nuggets are located in the environment. In the same way, a market where the miners can sell their gold nuggets is located in the environment. The gold miners try to find gold mines, to extract gold nuggets from them and to sell the gold extracted nuggets at the market.

Step List

This tutorial is composed of 5 steps corresponding to 5 models. For each step, we present its purpose, an explicit formulation, and the corresponding GAML code.

1. Creation of the basic model: gold mines and market
2. Definition of the BDI miners
3. Definition of social relations between miners
4. Use of emotions and personality for the miners
5. Adding norms, obligations and enforcement

Chapter 145

1. Skeleton model

This first step consists in defining the skeleton model with the gold mines and the gold market.

Formulation

- Definition of the gold mine species
- Definition of the market species
- Creation of the gold mine and market agents
- Definition of a display with the gold mines and the market

Model Definition

species

In this first model, we have to define two species of agents: the **goldmine** agents and the **market** ones. These agents will not have a particular behavior, they will just be displayed. For the goldmine species, we define a new attribute: **quantity** of type *int*, with for initial value a random integer between 0 and 20. We also define aspect called **default** that displays the goldmine as a triangle with a gray color if the gold mine is empty, yellow otherwise. The size of the triangle depends on the quantity of gold nuggets in the mine. Concerning the market species, we define a

new attribute: **golds** of type *int*. We define as well an aspect called **default** that displays the market as a blue square.

```
species goldmine {
    int quantity <- rnd(1,20);
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
            draw triangle(200 + quantity*50) color: #yellow border: #
   black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}
```

global variables

We define two global variables for the model: one called **nb_mines** that will be used to define the number of mines and that will be set to 10. One call **the_market** that will represent the market agent (that will be unique).

In addition, we define the duration of a simulation step to 10 minutes, and we define the shape of the environnement by a square with a side size of 20 kilometers.

```
global {
    int nb_mines <- 10;
    market the_market;
    float step <- 10#mn;
    geometry shape <- square(20 #km);
}
```

global init

At the initialisation of the model, we create a market agent and *nb_mines* goldmine agents. For the market agent, we set the value of the *the_market* agent with the created agent.

```
global {
   ...
   init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
    }
}
```

display

We define a display to visualize the market and goldmine agents. We use for that the classic **species** keyword. In order to optimize the display we use an opengl display (facet **type: opengl**).

In the **experiment** block:

```
output {
   display map type: opengl {
    species market ;
    species goldmine ;
   }
}
```

Complete Model

```
model GoldBdi

global {
    int nb_mines <- 10;
    market the_market;
    float step <- 10#mn;
```

```
    geometry shape <- square(20 #km);

    init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
    }
}

species goldmine {
    int quantity <- rnd(1,20);
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
            draw triangle(200 + quantity*50) color: #yellow border: #
   black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}

experiment GoldBdi type: gui {

    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
        }
    }
}
```

Back to the start of the tutorial 1. Definition of the BDI miners 1. Definition of

social relations between miners 1. Use of emotions and personality for the miners 1. Adding norms, obligations and enforcement

Chapter 146

2. BDI Agents

This second step consists in defining the gold miner agents using the GAMA BDI architecture.

Formulation

- Definition of global predicates
- Definition of the gold miner species
- Definition of the gold miner perceptions
- Definition of the gold miner rules
- Definition of the gold miner plans
- Creation and display of the gold miners

BDI agents

A classic paradigm to formalize the internal architecture of cognitive agents in Agent-Oriented Software Engineering is the BDI (Belief-Desire-Intention) paradigm. This paradigm, based on the philosophy of action (Bratman, 1987), allows to design expressive and realistic agents.

The concepts of Belief-Desire-Intention can be summarized as follow for the Gold Miner: the Miner agent has a general desire to find gold. As it is the only thing it wants at the beginning, it is its initial intention (what it is currently doing). To find

gold, it wanders around (its plan is to wander). When it perceives some gold nuggets, it stores this information (it has a new belief about the existence and location of this gold nugget), and it adopts a new desire (it wants to extract the gold). When it perceives a gold nugget, the intention to find gold is put on hold and a new intention is selected (to extract gold). To achieve this intention, the plan has two steps, i.e. two new (sub)intentions: to choose a gold nugget to extract (among its known gold nuggets) and to go and take it. And so on.

In GAMA, we propose a control architecture for agents based on this paradigm. This control architecture provides the agents with 3 databases linked to the agent cognition:

- **belief_base** (what it knows): the internal knowledge the agent has about the world or about its internal state, updated during the simulation. A belief can concern any type of information (a quantity, a location, a boolean value, etc).
- **desire_base** (what it wants): objectives that the agent would like to accomplish, also updated during the simulation. Desires can have hierarchical links (sub/super desires) when a desire is created as an intermediary objective.
- **intention_base** (what it is doing): what the agent has chosen to do. The current intention will determine the selected plan. Intentions can be put on hold (for example when they require a sub-intention to be achieved).

In addition, the BDI architecture provides agents with three types of behavior structures

- **Perception**: a perception is a function executed at each iteration to update the agent's Belief base, to know the changes in its environment (the world, the other agents and itself). The agent can perceive other agents up to a fixed distance or inside a specific geometry.
- **Rule**: a rule is a function executed at each iteration to infer new desires or beliefs from the agent's current beliefs and desires, i.e. a new desire or belief can emerge from the existing ones.
- **Plan**: the agent has a set of plans, which are behaviors defined to accomplish specific intentions. Plans can be instantaneous and/or persistent, and may have a priority value (that can be dynamic), used to select a plan when several possible plans are available to accomplish the same intention.

To be more precise on the behavior of BDI agents (what the agent is going to do when activated), this one is composed of 10 steps (see (Caillou et al., 2017) and (Taillandier et al., 2016) for more details):

1. *Perceive*: Perceptions are executed.
2. *Rule*: Rules are executed.
3. *Is one of my intentions achieved?*: If one of my intentions is achieved, sets the current plan to nil and removes the intention from the intention base. If the achieved intention's super-intention is on hold, it is reactivated (its sub-intention just got completed).
4. *Do I keep the current intention?*: To take into account the environment instability, an intention-persistence coefficient is applied: with this probability, the current intention is removed from the intention stack.
5. *Do I have a current plan?*: If I have a current plan, just execute it. Similarly to intentions, a plan-persistence coefficient is defined: with this probability, the current plan is just dropped.
6. *Choose a desire as new current intention*: If the current intention is on hold (or the intention base is empty), choose a desire as new current intention. The new selected intention is the desire with higher priority.
7. *Choose a plan as a new current plan*: The new current plan is selected among the plans compatible with the current intention (and if their activation condition is checked) and with the highest priority.
8. *Execute the plan*: The current plan is executed.
9. *Is my plan finished?*: To allow persistent plans, a plan may have a termination condition. If it is not reached, the same plan will be kept for the next iteration.
10. *Was my plan instantaneous?*: Most agent-based simulation frameworks (GAMA included) are synchronous frameworks using steps. One consequence is that it may be useful to apply several plans during one single step. For example, if a step represents a day or a year, it would be unrealistic for an agent to spend one step to apply a plan like "choose a destination". This kind of plans (mostly reasoning plans) can be defined as instantaneous: in this case a new thinking loop is applied during the same agent step.

The architecture introduces two new main types of variables related to cognition:

- **predicate**: a predicate unifies the representation of the information about the world. It can represent a situation, an event or an action.

- **mental_state**: it represents the element (belief, desire, intention) manipulated by the agent and the architecture to take a decision. A mental state is composed of a modality, a predicate or another mental state, a real value and a lifetime. The modality indicates the type of the mental state (e.g. a belief or a desire),

the predicate indicates the fact about which is this mental state (a mental state can also be about another mental state like a belief about a belief, etc), the value has a different interpretation depending on the modality and finally, the lifetime indicates the duration of the mental state (it can be infinite).

Model Definition

predicates

As a first step of the integration of the BDI agents in our model, we define a set of global predicate that will represent all the information that will be manipulated by the miner agents:

- *mine_location*: represents the information about the location of a gold mine.
- *choose_goldmine*: represents the information that the miner wants to choose a gold mine.
- *has_gold*: represents the information that the miner has a gold nugget.
- *find_gold*: represents the information that the miner wants to find gold.
- *sell_gold*: represents the information that the miner wants to sell gold.

We define as well two global string (*mine_at_location* and *empty_mine_location*) for simplification purpose and to avoid misspellings.

```
global {
        ...
    string mine_at_location <- "mine_at_location";
    string empty_mine_location <- "empty_mine_location";

    predicate mine_location <- new_predicate(mine_at_location) ;
    predicate choose_goldmine <- new_predicate("choose a gold mine");
    predicate has_gold <- new_predicate("extract gold");
    predicate find_gold <- new_predicate("find gold") ;
    predicate sell_gold <- new_predicate("sell gold") ;
        ...
}
```

skeleton of the miner species

We then define a miner species with the *moving* skill and the *simple_bdi* control architecture. The miner agents have 5 variables:

- *viewdist*: distance of perception of the miner agent
- *speed*: speed of the agent
- *mycolor*: the color of the agent (random color)
- *target*: where the agent wants to go
- *gold_sold*: the number of gold nuggets sold by the agent

We define the init block of the species such as to add at the creation of the agent the desire to find gold nuggets (*find_gold* predicate). we use for that the *add_desire* action provides with the BDI architecture.

At last, we define an aspect in which we draw the agent with its *mycolor* color and with a depth that depends on the number of gold nuggets collected.

```
species miner skills: [moving] control:simple_bdi {
    float viewdist<-1000.0;
    float speed <- 2#km/#h;
    rgb mycolor<-rnd_color(255);
    point target;
    int gold_sold;

    init
    {
        do add_desire(find_gold);
    }
    aspect default {
            draw circle(200) color: mycolor border: #black depth:
   gold_sold;
    }
}
```

perception

We add a *perceive* statement for the miner agents. This perceive will allow to detect the gold mine that are not empty (i.e. the quantity of gold is higher than 0) at a distance lower or equal to “viewdist”. The use of the *focus* statement allows to add for

each detected goldmine a belief corresponding to the location of this goldmine. The name of the belief will be "mine_at_location" and the location value of the goldmine will be stored in the *values* (a map) variable of the belief at the key "location_value". In addition, we ask the miner agent to remove the intention to find gold, allowing the agent to choose a new intention. The boolean value of the *remove_intention* action is used to specify if the agent should or not remove the given intention from the desire base as well. In our case, we choose to keep the desire to find golds.

```
species miner skills: [moving] control:simple_bdi {
    ...
    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus id:mine_at_location var:location;
        ask myself {
            do remove_intention(find_gold, false);
        }
    }
}
```

Note that the perceive statement works as the ask statement: the instructions written in the statement are executed in the context of the perceive agents. It is for that that we have to use the *myself* keyword to ask the miner agent to execute the *remove_intention* action.

rules

We define two rules for the miner agents:

- if the agent believes that there is somewhere at least one gold mine with gold nuggets, the agent gets the new desire to has a gold nugget with a strength of 2.
- if the agent believes that it has a gold nugget, the agent gets the new desire to sell the gold nugget with a strength of 3.

```
species miner skills: [moving] control:simple_bdi {
    ...
    rule belief: mine_location new_desire: has_gold strength: 2.0;
    rule belief: has_gold new_desire: sell_gold strength: 3.0;
}
```

The strength of a desire will be used when selecting a desire as a new intention: the agent will choose as new intention the one with the highest strength. In our model, if

the agent has the desires to find gold, to has gold and to sell gold, it will choose as intention to sell gold as it is the one with the highest strength. It is possible to replace this deterministic choice by a probabilistic one by setting the *probabilistic_choice* built-in varibale of the BDI agent to true (false by default).

plans

The last (and most important) part of the definition of BDI agents consist in defining the plans that the agents can carry out to acheive its intention.

The first plan called *letsWander* is defined to achieve the *find_gold* intention. This plan will just consists in executing the *wander* action of the *moving* skill (random move).

```
species miner skills: [moving] control:simple_bdi {
        ...
        plan letsWander intention:find_gold
    {
        do wander;
    }
       ...
}
```

The second plan called *getGold* is defined to achieve the *has_gold* intention. if the agent has no target (it does not know where to go), it adds a new sub-intention to choose a goldmine and put the current intention on hold (the agent will wait to select a gold mine to go before executing again this plan). The *add_subintention* has 3 arguments: the sub-intention (choose_goldmine), the super intention (extract_gold) and a boolean that defines if the sub-intention should or not be added as well as a desire. If the agent has already a target, it moves toward this target using the *goto* action of the *moving* skill. If the agent reaches its target - goldmine - (target = location), the agent tries to extract gold nuggets from it. If the corresponding goldmine (that one located at the target location) is not empty, the agent extract a gold nugget from it: the agent adds the belief that it has a gold nugget, then the quantity of golds in the gold mine is reduced. Otherwise, if the gold mine is empty, the agent adds the belief that this gold mine is empty. then the target is set to nil. get_current_intention() returns the current intention. As plan getGold is executed only when intention is about the has_gold predicate, the current intention is about this predicate.

```
species miner skills: [moving] control:simple_bdi {
```

```
        ...
        plan getGold intention:has_gold
    {
        if (target = nil) {
            do add_subintention(get_current_intention(),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 {
                    do add_belief(has_gold);
                    ask current_mine {quantity <- quantity - 1;}
                } else {
                    do add_belief(new_predicate(empty_mine_location, ["
   location_value"::target]));
                }
                target <- nil;
            }
        }
    }
       ...
}
```

The third plan called *choose_closest_goldmine* is defined to achieve the *choose_goldmine* intention that is instantaneous. First, the agent defines the list of all the gold mines it knows (*mine_at_location* beliefs), then removes the gold mines that it knows that they are empty (*empty_mine_location* beliefs). If the list of the possible mines is empty, the agent removes the desire and the intention to *extract_gold*. We use for that the *remove_intention* action, that removes an intention from the intention base; the second argument allows to define if the intention should be removed as well from the desire base. If the agent knows at least one gold mine that is not empty, it defines as its new target the closest gold mine.

```
species miner skills: [moving] control:simple_bdi {
        ...
        plan choose_closest_goldmine intention: choose_goldmine
   instantaneous: true{
        list<point> possible_mines <- get_beliefs_with_name(
   mine_at_location) collect (point(get_predicate(mental_state (each)).
   values["location_value"]));
        list<point> empty_mines <- get_beliefs_with_name(
```

```
    empty_mine_location) collect (point(get_predicate(mental_state (each
    )).values["location_value"]));
         possible_mines <- possible_mines - empty_mines;
         if (empty(possible_mines)) {
             do remove_intention(extract_gold, true);
         } else {
             target <- (possible_mines with_min_of (each distance_to
    self)).location;
         }
         do remove_intention(choose_goldmine, true);
     }
        ...
}
```

The last plan called *return_to_base* is defined to achieve the *sell_gold* intention. The agent moves in direction of the market using the *goto* action. if the agent reaches the market, it sells its gold nugget to it: first, it removes the belief that it has a gold nugget, then it removes the intention and the desire to sell golds, at last it increments its *gold_sold* variable.

```
species miner skills: [moving] control:simple_bdi {
        ...
        plan return_to_base intention: sell_gold {
        do goto target: the_market ;
        if (the_market.location = location)  {
            do remove_belief(has_gold);
            do remove_intention(sell_gold, true);
            gold_sold <- gold_sold + 1;
        }
    }
        ...
}
```

Gobal section

We define two new global variables:

- *nbminer*: number of gold miners
- *inequality*: recomputed at each simulation step: standard deviation of the number of gold nuggets extracted per miners.

In the global init, after creating the gold mines and the market, we create the gold miner agents.

At last, we define a global reflex *end_simulation* that is activated when all the gold mines are empty and no more miner has a gold nuggets and that pauses the simulation.

```
global {
    ...
    int nbminer<-5;
    float inequality <- 0.0 update:standard_deviation(miner collect
   each.gold_sold);
    ...
        init
    {
        ...
        create miner number:nbminer;
    }

    reflex end_simulation when: sum(goldmine collect each.quantity) = 0
    and empty(miner where each.has_belief(has_gold)){
        do pause;
    }
}
```

Map display

We add to the map display the miner species.

```
experiment GoldBdi type: gui {

    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
            species miner;
        }
    }
}
```

Complete Model

```
model GoldBdi

global {
    int nb_mines <- 10;
    int nbminer<-5;
    market the_market;

    string mine_at_location <- "mine_at_location";
    string empty_mine_location <- "empty_mine_location";

    float step <- 10#mn;

    //possible predicates concerning miners
    predicate mine_location <- new_predicate(mine_at_location) ;
    predicate choose_goldmine <- new_predicate("choose a gold mine");
    predicate has_gold <- new_predicate("extract gold");
    predicate find_gold <- new_predicate("find gold") ;
    predicate sell_gold <- new_predicate("sell gold") ;



    float inequality <- 0.0 update:standard_deviation(miner collect
   each.gold_sold);

    geometry shape <- square(20 #km);

    init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
        create miner number:nbminer;
    }

    reflex end_simulation when: sum(goldmine collect each.quantity) = 0
    and empty(miner where each.has_belief(has_gold)){
        do pause;
    }
}

species goldmine {
    int quantity <- rnd(1,20);
```

```
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
            draw triangle(200 + quantity*50) color: #yellow border: #
   black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}

species miner skills: [moving] control:simple_bdi {

    float viewdist<-1000.0;
    float speed <- 2#km/#h;
    rgb mycolor<-rnd_color(255);
    point target;
    int gold_sold;

    init
    {
        do add_desire(find_gold);
    }

    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus id:mine_at_location var:location;
        ask myself {
            do remove_intention(find_gold, false);
        }
    }
    rule belief: mine_location new_desire: has_gold strength: 2.0;
    rule belief: has_gold new_desire: sell_gold strength: 3.0;


    plan letsWander intention:find_gold
    {
        do wander;
```

```
    }

    plan getGold intention:has_gold
    {
        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 {
                    do add_belief(has_gold);
                    ask current_mine {quantity <- quantity - 1;}
                } else {
                    do add_belief(new_predicate(empty_mine_location, ["
    location_value"::target]));
                }
                target <- nil;
            }
        }
    }

    plan choose_closest_goldmine intention: choose_goldmine
    instantaneous: true{
        list<point> possible_mines <- get_beliefs_with_name(
    mine_at_location) collect (point(get_predicate(mental_state (each)).
    values["location_value"]));
        list<point> empty_mines <- get_beliefs_with_name(
    empty_mine_location) collect (point(get_predicate(mental_state (each
    )).values["location_value"]));
        possible_mines <- possible_mines - empty_mines;
        if (empty(possible_mines)) {
            do remove_intention(has_gold, true);
        } else {
            target <- (possible_mines with_min_of (each distance_to
    self)).location;
        }
        do remove_intention(choose_goldmine, true);
    }

    plan return_to_base intention: sell_gold {
        do goto target: the_market ;
        if (the_market.location = location)  {
            do remove_belief(has_gold);
```

```
            do remove_intention(sell_gold, true);
            gold_sold <- gold_sold + 1;
        }
    }

    aspect default {
      draw circle(200) color: mycolor border: #black depth: gold_sold;
    }
}

experiment GoldBdi type: gui {

    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
            species miner;
        }
    }
}
```

Back to the start of the tutorial 1. Creation of the basic model: gold mines and market 3. Definition of social relations between miners 4. Use of emotions and personality for the miners 5. Adding norms, obligations and enforcement

Chapter 147

3. Social relation

This third step consists in adding social relation between agents and the possibility to share information about the known gold mines.

Formulation

- Definition of global predicates
- Definition of the gold miner species
- Definition of the gold miner perceptions with socialization
- Definition of a new gold miner plan to share information

Social relation

The BDI architecture of GAMA allows to define explicit social relations between agents. Based on the work of Svennevig, a social link with another agent is defined as a tuple <agent, liking, dominance, solidarity, familiarity, trust> with the following elements:

- Agent: the agent concerned by the link, identified by its name.
- Liking: 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 the concerned agent is hated, a value of 1 indicates that the concerned agent is liked.

- Dominance: 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 the concerned agent is dominating, a value of 1 indicates that the concerned agent is dominated.
- Solidarity: a real value between 0 and 1 representing the degree of solidarity with the agent concerned by the link. A value of 0 indicates no solidarity with the concerned agent, a value of 1 indicates a complete solidarity with the concerned agent.
- Familiarity: a real value between 0 and 1 representing the degree of familiarity with the agent concerned by the link. A value of 0 indicates no familiarity with the concerned agent, a value of 1 indicates a complete familiarity with the concerned agent.
- Trust: a real value between -1 and +1 representing the degree of trust with the agent concerned by th link. A value of -1 indicates a doubt about the agent concerned, a value of 1 indicates a complete trust with the concerned agent.

With this definition, a social relation is not necessarily symmetric. For example, let's take two agents, Alice and Bob, with a social link towards each other. The agent Bob may have a social link <Alice,1,-0.5,0.6,0.8,-0.2> (Bob likes Alice with a value of 1, he thinks he is dominated by Alice, he is solidary with Alice with a value of 0.6, he is familiar with Alice with a value of 0.8 and he doubts about her with a value 0.2) and Alice may have a social link <Bob,-0.2,0.2,0.4,0.5,0.8> (Alice dislikes Bob with a value of 0.2, she thinks she is dominating Bob, she is solidary with Bob with a value of 0.4, she is familiar with Bob with a value of 0.5 and she trusts Bob with a value of 0.5).

Model Definition

predicates

We add a new global predicate called *share_information* that represents the information that the miner wants to share information.

```
global {
    ...
    predicate share_information <- new_predicate("share information") ;
        ...
}
```

perception

We add a new perceive statement for the miner agents. This perceive will allow to create a social relation with the miners that are located at a distance lower or equal to "viewdist" to the agent. For each of these miner agents, the agents create a new social relation using the *socialize* statement with a liking value that depends on the color of the agents: more the agents are close, higher will be the liking value.

```
species miner skills: [moving] control:simple_bdi {
    ...
        perceive target:miner in:viewdist {
        socialize liking: 1 -  point(mycolor.red, mycolor.green,
   mycolor.blue) distance_to point(myself.mycolor.red, myself.mycolor.
   green, myself.mycolor.blue) / ( 255);
    }
}
```

We also modify the perceive statement previously defined in order to add the desire to share information with a strength of 5 if the agent finds a gold mine.

```
species miner skills: [moving] control:simple_bdi {
    ...
    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus mine_at_location var:location;
        ask myself {
            do add_desire(predicate:share_information, strength: 5.0);
            do remove_intention(find_gold, false);
        }
    }
```

plan

At last, we add a new plan for the miner agents called *share_information_to_friends* to achieve the intention *share_information* that is instantaneous. In this plan, the miner agent first defines its list of friends, i.e. the miners with which it has a social link and that it likes (linking higher than 0). then for each friend, it shares its list of known mines (beliefs about their location), then its knowledge about the mines that are empty (beliefs about their location). At last, it removes the desire and intention to *share_information*.

```
species miner skills: [moving] control:simple_bdi {
    ...
```

```
    plan share_information_to_friends intention: share_information
   instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
   each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
   mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        loop known_empty_goldmine over: get_beliefs_with_name(
   empty_mine_location) {
            ask my_friends {
                do add_belief(known_empty_goldmine);
            }
        }

        do remove_intention(share_information, true);
    }
}
```

Complete Model

```
model GoldBdi

global {
    int nb_mines <- 10;
    int nbminer<-5;
    market the_market;

    string mine_at_location <- "mine_at_location";
    string empty_mine_location <- "empty_mine_location";

    float step <- 10#mn;

    //possible predicates concerning miners
    predicate mine_location <- new_predicate(mine_at_location) ;
    predicate choose_goldmine <- new_predicate("choose a gold mine");
    predicate has_gold <- new_predicate("extract gold");
    predicate find_gold <- new_predicate("find gold") ;
    predicate sell_gold <- new_predicate("sell gold") ;
    predicate share_information <- new_predicate("share information") ;
```

```
    float inequality <- 0.0 update:standard_deviation(miner collect
   each.gold_sold);

    geometry shape <- square(20 #km);
    init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
        create miner number:nbminer;
    }

    reflex end_simulation when: sum(goldmine collect each.quantity) = 0
    and empty(miner where each.has_belief(has_gold)){
        do pause;
    }
}

species goldmine {
    int quantity <- rnd(1,20);
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
            draw triangle(200 + quantity*50) color: #yellow border: #
   black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}

species miner skills: [moving] control:simple_bdi {

    float viewdist<-1000.0;
    float speed <- 2#km/#h;
    rgb mycolor<-rnd_color(255);
```

```
    point target;
    int gold_sold;

        bool use_social_architecture <- true;

    init
    {
        do add_desire(find_gold);
    }

    perceive target:miner in:viewdist {
        socialize liking: 1 -  point(mycolor.red, mycolor.green,
    mycolor.blue) distance_to point(myself.mycolor.red, myself.mycolor.
    green, myself.mycolor.blue) / ( 255);
    }

    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus id: mine_at_location var:location;
        ask myself {
            do add_desire(predicate:share_information, strength: 5.0);
            do remove_intention(find_gold, false);
        }
    }

    rule belief: mine_location new_desire: has_gold strength: 2.0;
    rule belief: has_gold new_desire: sell_gold strength: 3.0;


    plan letsWander intention:find_gold
    {
        do wander;
    }

    plan getGold intention:has_gold
    {
        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 {
                    do add_belief(has_gold);
                    ask current_mine {quantity <- quantity - 1;}
```

```
                } else {
                    do add_belief(new_predicate(empty_mine_location, ["
location_value"::target]));
                }
                target <- nil;
            }
        }
    }

    plan choose_closest_goldmine intention: choose_goldmine
instantaneous: true{
        list<point> possible_mines <- get_beliefs_with_name(
mine_at_location) collect (point(get_predicate(mental_state (each)).
values["location_value"]));
        list<point> empty_mines <- get_beliefs_with_name(
empty_mine_location) collect (point(get_predicate(mental_state (each
)).values["location_value"]));
        possible_mines <- possible_mines - empty_mines;
        if (empty(possible_mines)) {
            do remove_intention(has_gold, true);
        } else {
            target <- (possible_mines with_min_of (each distance_to
self)).location;
        }
        do remove_intention(choose_goldmine, true);
    }

    plan return_to_base intention: sell_gold {
        do goto target: the_market ;
        if (the_market.location = location)  {
            do remove_belief(has_gold);
            do remove_intention(sell_gold, true);
            gold_sold <- gold_sold + 1;
        }
    }
    plan share_information_to_friends intention: share_information
instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
```

```
        loop known_empty_goldmine over: get_beliefs_with_name(
    empty_mine_location) {
            ask my_friends {
                do add_belief(known_empty_goldmine);
            }
        }

        do remove_intention(share_information, true);
    }

    aspect default {
      draw circle(200) color: mycolor border: #black depth: gold_sold;
    }
}


experiment GoldBdi type: gui {
    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
            species miner;
        }
    }
}
```

Back to the start of the tutorial 1. Creation of the basic model: gold mines and market 2. Definition of the BDI miners 4. Use of emotions and personality for the miners 5. Adding norms, obligations and enforcement

Chapter 148

4. Emotions and Personality

This fourth step consists in adding emotions that will impact the gold miner agent behavior and defining the personality of the agents.

Formulation

- Definition of global emotions
- Modification of the miner species to integrate emotions and personality

Emotions

The BDI architecture of GAMA gives the possibility to generate emotions and to use them in the cognition. The definition of emotions in GAMA is based on the OCC theory of emotions. According to this theory, an emotion is a valued answer to the appraisal of a situation. In GAMA an emotion is represented by a set of 5 elements:

- E: the name of the emotion felt by agent i.
- P: the predicate that represents the fact about which the emotion is expressed.
- A: the agent causing the emotion.
- I: the intensity of the emotion.
- D: the decay of the emotion's intensity.

The BDI architecture of GAMA integrates a dynamic creation of emotions process that will create emotions according to the mental states of the agent. More precisely, twenty emotions can be created: eight emotions related to events, four emotions related to other agents and eight emotions related to actions.

The complete description of these emotions and their creation rules can be found in (Bourgais et at., 2017).

Personality

In order to facilitate the parametrization of the BDI agents, we add the possibility to define all the parameters related to the BDI architecture through the OCEAN model, which proposes to represent the personality of a person according to five factors (corresponding to the 5 variables of the BDI agents):

- O: represents the openness of someone (open-minded/narrow-minded).
- C: represents the consciousness of someone (act with preparations/impulsive).
- E: represents the extroversion of someone (extrovert/shy).
- A: represents the agreeableness of someone (friendly/hostile).
- N: represent the degree of control someone has on its emotions (calm/neurotic) \end{itemize}

Each of these variables has a value between 0 and 1. 0.5 represents the neutral value, below 0.5, the value is considered negatively and above 0.5, it is considered positively. For example, someone with a value of 1 for N is considered as calm and someone with a value of 0 for A is considered as hostile.

Model Definition

emotions

We add a new global emotion called *joy* that represents the joy emotion.

```
global {
    ...
    emotion joy <- new_emotion("joy");
        ...
}
```

emotions and personality

To use emotions (and to activate the automatic emotion generation process), we just have to set the value of the built-in variable *use_emotions_architecture* to true (false by default). In our case, one of the possible desires concerns the predicate *has_gold*, and when an agent fulfill this desire and find a gold nugget (plan *getGold*), it gets the belief *has_gold*, and the emotion engine automatically creates a *joy* emotion.

To be able to define the parameter of a BDI agent through the OCEAN model, we have to set the value of the built-in variable *use_personality* to true (false by default). In this model, we chose to use the default value of the O, C, E, A and N variables (default value: 0.5). The interest of using the personality in our case is to allow the emotion engine to give a lifetime to the created emotions (otherwise, the emotions would have an infinite lifetime).

In this model, we only use the emotions to define if the miner agents are going to share or not its knowledge about the gold mines. We consider that the miner only shares information if it has a joy emotion.

```
species miner skills: [moving] control:simple_bdi {
        ...
        bool use_emotions_architecture <- true;
        bool use_personality <- true;

    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus mine_at_location var:location;
        ask myself {
            if (has_emotion(joy)) {do add_desire(predicate:
   share_information, strength: 5.0);}
            do remove_intention(find_gold, false);
        }
    }
        ...
}
```

Complete Model

```
model GoldBdi

global {
    int nb_mines <- 10;
```

```
    int nbminer<-5;
    market the_market;

    string mine_at_location <- "mine_at_location";
    string empty_mine_location <- "empty_mine_location";

    float step <- 10#mn;

    //possible predicates concerning miners
    predicate mine_location <- new_predicate(mine_at_location) ;
    predicate choose_goldmine <- new_predicate("choose a gold mine");
    predicate has_gold <- new_predicate("extract gold");
    predicate find_gold <- new_predicate("find gold") ;
    predicate sell_gold <- new_predicate("sell gold") ;
    predicate share_information <- new_predicate("share information") ;


    emotion joy <- new_emotion("joy");

    float inequality <- 0.0 update:standard_deviation(miner collect
   each.gold_sold);

    geometry shape <- square(20 #km);
    init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
        create miner number:nbminer;
    }

    reflex end_simulation when: sum(goldmine collect each.quantity) = 0
    and empty(miner where each.has_belief(has_gold)){
        do pause;
    }
}

species goldmine {
    int quantity <- rnd(1,20);
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
```

```
            draw triangle(200 + quantity*50) color: #yellow border: #
    black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}

species miner skills: [moving] control:simple_bdi {

    float viewdist<-1000.0;
    float speed <- 2#km/#h;
    rgb mycolor<-rnd_color(255);
    point target;
    int gold_sold;

        bool use_social_architecture <- true;
    bool use_emotions_architecture <- true;
    bool use_personality <- true;

    init
    {
        do add_desire(find_gold);
    }

    perceive target:miner in:viewdist {
        socialize liking: 1 -  point(mycolor.red, mycolor.green,
    mycolor.blue) distance_to point(myself.mycolor.red, myself.mycolor.
    green, myself.mycolor.blue) / ( 255);
    }

    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus id:mine_at_location var:location;
        ask myself {
            if (has_emotion(joy)) {do add_desire(predicate:
    share_information, strength: 5.0);}
            do remove_intention(find_gold, false);
        }
    }
```

```
    rule belief: mine_location new_desire: has_gold strength: 2.0;
    rule belief: has_gold new_desire: sell_gold strength: 3.0;

    plan letsWander intention:find_gold
    {
        do wander;
    }

    plan getGold intention:has_gold
    {
        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 {
                    do add_belief(has_gold);
                    ask current_mine {quantity <- quantity - 1;}
                } else {
                    do add_belief(new_predicate(empty_mine_location, ["
    location_value"::target]));
                }
                target <- nil;
            }
        }
    }

    plan choose_closest_goldmine intention: choose_goldmine
    instantaneous: true{
        list<point> possible_mines <- get_beliefs_with_name(
    mine_at_location) collect (point(get_predicate(mental_state (each)).
    values["location_value"]));
        list<point> empty_mines <- get_beliefs_with_name(
    empty_mine_location) collect (point(get_predicate(mental_state (each
    )).values["location_value"]));
        possible_mines <- possible_mines - empty_mines;
        if (empty(possible_mines)) {
            do remove_intention(has_gold, true);
        } else {
            target <- (possible_mines with_min_of (each distance_to
    self)).location;
        }
```

```
        do remove_intention(choose_goldmine, true);
    }

    plan return_to_base intention: sell_gold {
        do goto target: the_market ;
        if (the_market.location = location)  {
            do remove_belief(has_gold);
            do remove_intention(sell_gold, true);
            gold_sold <- gold_sold + 1;
        }
    }
    plan share_information_to_friends intention: share_information
   instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
   each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
   mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        loop known_empty_goldmine over: get_beliefs_with_name(
   empty_mine_location) {
            ask my_friends {
                do add_belief(known_empty_goldmine);
            }
        }

        do remove_intention(share_information, true);
    }

    aspect default {
      draw circle(200) color: mycolor border: #black depth: gold_sold;
    }
}


experiment GoldBdi type: gui {
    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
            species miner;
        }
    }
```

```
}
```

Back to the start of the tutorial 1. Creation of the basic model: gold mines and market 2. Definition of the BDI miners 3. Definition of social relations between miners 5. Adding norms, obligations and enforcement

Chapter 149

5. Norms, obligation, and enforcement

This last step consists of adding social norms, obligations, and enforcement into the agents 'behavior.

Formulation

- Definition of global predicates
- Definition of the policeman species
- Definition of the enforcement done by policeman species
- Definition of the law agents have to follow
- Definition of a gold miner norm to fulfill its obligation and its social norms
- Definition of the enforcement done by gold miners

Norms, obligations, and enforcement

The BDI architecture of GAMA allows defining explicit social norms, laws that lead to obligations and an enforcement process to sanction or reward the other agent depending on their behavior toward norms. A social norm is a set of actions executed under certain conditions which are known by the people has the right things to do in that conditions. As it is, it can be assimilated into a plan. However, a norm can be

violated which mean an agent chose to disobey and do not execute it while it should. To do this, each agent has an obedient value, between 0 and 1 ad computed from its personality and each norm has a threshold. If the obedient value of the agent is above the threshold, the norm is executed. An obligation is a mental state that can be assimilated with a desire. It is created by a law that indicates under which conditions the agent has to follow a particular obligation. Once again, the law can have a threshold value to be activated or not depending on the obedient value of the agent. If an agent has an obligation, it will give up its current intention and current plan to get this obligation as its intention. Then, it will choose a specific norm to answer this obligation, once again with a threshold on the obedient value. Finally, an enforcement mechanism can be defined during the perception process. Norms, laws, and obligation can be enforced. If a violation is detected, a sanction can be executed. If the norm/law/obligation is fulfilled, a reward can be executed.

Model Definition

law

We add a law to the gold miner species that will create the obligation to get a gold if a gold nugget is perceived. This law replaces a rule and expresses the fact that miners are working or not, depending on their obedience value.

```
species miner skills: [moving] control:simple_bdi {
    ...
    law working belief: mine_location new_obligation: has_gold
   when:not has_obligation(has_gold) and not has_belief(has_gold)
    strength: 2.0 threshold:thresholdLaw;

        ...
}
```

norms

The miners will have two norms. A first one to answer the obligation to collect gold. This norms replaces the previous plan created for this purpose. However, a new plan is declared to get 3 pieces of gold at each time. This plan will be considered illegal by the policeman species.

```
species miner skills: [moving] control:simple_bdi {
    ...
        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;
            }
        }
    }
}
```

The second norm is a social norm to communicate the list of known mines to one's friends. It replaces the previous plan that did this action, while a new plan is added to give a wrong list of mines to one's friend.

```
species miner skills: [moving] control:simple_bdi {
    ...
    norm share_information intention:share_information threshold:
   thresholdNorm instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
   each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
   mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        loop known_empty_goldmine over: get_beliefs_with_name(
```

```
    empty_mine_location) {
              ask my_friends {
                  do add_belief(known_empty_goldmine);
              }
          }

          do remove_intention(share_information, true);
      }
```

enforcement of the social norm

Finally, for the gold-miner agent, an enforcement is defined about the social norm to communicate the location of mines to other agents. A sanction and a reward are declared to change the liking value with the agent controlled, depending on if the norm is violated or fulfilled.

```
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);
    }
}
```

Definition of policeman species

Finally, we define a policeman species that will wander through the map and enforce the miners about the law and the obligation. The sanctions will be a fine collected by policemen.

```
species policeman skills: [moving] control:simple_bdi {
    predicate patroling <- new_predicate("patrolling");
```

```
    float viewdist <- 1000.0;
    miner agent_perceived <- nil;

    init {
        do add_desire(patroling);
    }

    perceive target:miner in: viewdist{
        myself.agent_perceived <- self;
        enforcement law:"working" sanction:"sanctionToLaw";
        enforcement obligation:has_gold sanction: "sanctionToObligation
    " reward:"rewardToObligation";
    }

    sanction sanctionToLaw{
        ask agent_perceived{
            thresholdLaw <- 0.0;
            gold_sold <- gold_sold-5;
        }
        fine <- fine +5;
    }

    sanction sanctionToObligation {
        ask agent_perceived{
            gold_sold <- gold_sold-3;
            do remove_intention(sell_gold,true);
            thresholdObligation <- self.thresholdObligation - 0.1;
        }
        fine <- fine + 3;
    }

    sanction rewardToObligation{
        ask agent_perceived{
            gold_sold <- gold_sold+2;
        }
        fine <- fine -2;
    }

    plan patrol intention: patroling{
        do wander;
    }

    aspect base{
        draw circle(viewdist) color: #blue depth:0.0;
    }
}
```

Complete Model

```
model GoldBdi

global {
    int nb_mines <- 10;
    int nbminer<-5;
    int nb_police <- 1;
    int fine <-0;
    market the_market;

    string mine_at_location <- "mine_at_location";
    string empty_mine_location <- "empty_mine_location";

    float step <- 10#mn;

    //possible predicates concerning miners
    predicate mine_location <- new_predicate(mine_at_location) ;
    predicate choose_goldmine <- new_predicate("choose a gold mine");
    predicate has_gold <- new_predicate("extract gold");
    predicate find_gold <- new_predicate("find gold") ;
    predicate sell_gold <- new_predicate("sell gold") ;
    predicate share_information <- new_predicate("share information") ;


    emotion joy <- new_emotion("joy");

    float inequality <- 0.0 update:standard_deviation(miner collect
   each.gold_sold);

    geometry shape <- square(20 #km);
    init
    {
        create market {
            the_market <- self;
        }
        create goldmine number:nb_mines;
        create miner number:nbminer;
        create policeman number:nb_police;
    }

    reflex end_simulation when: sum(goldmine collect each.quantity) = 0
    and empty(miner where each.has_belief(has_gold)){
        do pause;
        ask miner{
```

```
            write name + " : " +gold_sold;
        }
        write "**********************";
        write "fine : " + fine;
    }
}

species goldmine {
    int quantity <- rnd(1,20);
    aspect default
    {
        if (quantity = 0) {
            draw triangle(200) color: #gray border: #black;
        } else {
            draw triangle(200 + quantity*50) color: #yellow border: #
   black;
        }

    }
}

species market {
    int golds;
    aspect default
    {
      draw square(1000) color: #black ;
    }
}

species policeman skills: [moving] control:simple_bdi {
    predicate patroling <- new_predicate("patrolling");
    float viewdist <- 1000.0;
    miner agent_perceived <- nil;

    init {
        do add_desire(patroling);
    }

    perceive target:miner in: viewdist{
        enforcement law:"working" sanction:"sanctionToLaw";
        enforcement obligation:has_gold /*when:has_belief(has_gold)*/
   sanction: "sanctionToObligation" reward:"rewardToObligation";
    }

    sanction sanctionToLaw{
        ask agent_perceived{
```

```
            thresholdLaw <- 0.0;
            gold_sold <- gold_sold-5;
        }
        fine <- fine +5;
    }

    sanction sanctionToObligation {
        ask agent_perceived{
            gold_sold <- gold_sold-3;
            do remove_intention(sell_gold,true);
            thresholdObligation <- self.thresholdObligation - 0.1;
        }
        fine <- fine + 3;
    }

    sanction rewardToObligation{
        ask agent_perceived{
            gold_sold <- gold_sold+2;
        }
        fine <- fine -2;
    }

    plan patrol intention: patroling{
        do wander;
    }

    aspect base{
        draw circle(viewdist) color: #blue depth:0.0;
    }
}

species miner skills: [moving] control:simple_bdi {

    float viewdist<-1000.0;
    float speed <- 2#km/#h;
    rgb mycolor<-rnd_color(255);
    point target;
    int gold_sold;
    int gold_transported<-0;
    agent agent_perceived<-nil;

    bool use_social_architecture <- true;
    bool use_emotions_architecture <- true;
    bool use_personality <- true;

    float openness <- gauss(0.5,0.12);
```

```
    float conscientiousness <- gauss(0.5,0.12);
    float extraversion <- gauss(0.5,0.12);
    float agreeableness <- gauss(0.5,0.12);
    float neurotism <- gauss(0.5,0.12);

    float plan_persistence <- 1.0;
    float intention_persistence <- 1.0;

    float thresholdLaw <- 1.0;
    float thresholdObligation <- 1.0;
    float thresholdNorm <- 0.5;

    init
    {
        do add_desire(find_gold);
    }

    perceive target:self{
        if(gold_transported>0){
            do add_belief(has_gold);
        } else {
            do remove_belief(has_gold);
        }
    }

    perceive target:miner in:viewdist {
        myself.agent_perceived<-self;
        socialize liking: point(mycolor.red, mycolor.green, mycolor.
    blue) distance_to point(myself.mycolor.red, myself.mycolor.green,
    myself.mycolor.blue) / ( 255) - 1;
        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);
    }

    perceive target:goldmine where (each.quantity > 0) in:viewdist {
        focus id:mine_at_location var:location;
        ask myself {
```

```
            if (has_emotion(joy)) {do add_desire(predicate:
    share_information, strength: 5.0);}
            do remove_intention(find_gold, false);
        }
    }

    rule belief: has_gold new_desire: sell_gold strength: 3.0;

    law working belief: mine_location new_obligation: has_gold when:not
     has_obligation(has_gold) and not has_belief(has_gold) strength: 2.0
     threshold:thresholdLaw;

    plan letsWander intention:find_gold
    {
        do wander;
    }

    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;
            }
        }
    }

    plan getMoreGold intention:has_gold
    {
        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 <- 3;
                    do add_belief(has_gold);
                    ask current_mine {if(quantity>=3) {
                        quantity <- quantity - 3;
                    }else {
                        quantity <- 0;
                    }
                    }
                } 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;
            }
        }
     }

     plan choose_closest_goldmine intention: choose_goldmine
    instantaneous: true{
        list<point> possible_mines <- get_beliefs_with_name(
    mine_at_location) collect (point(get_predicate(mental_state (each)).
    values["location_value"]));
        list<point> empty_mines <- get_beliefs_with_name(
    empty_mine_location) collect (point(get_predicate(mental_state (each
    )).values["location_value"]));
        possible_mines <- possible_mines - empty_mines;
        if (empty(possible_mines)) {
            do remove_intention(has_gold, true);
        } else {
            target <- (possible_mines with_min_of (each distance_to
    self)).location;
        }
        do remove_intention(choose_goldmine, true);
     }

     plan return_to_base intention: sell_gold when: has_belief(has_gold)
    {
```

```
        do goto target: the_market ;
        if (the_market.location = location)  {
            do remove_belief(has_gold);
            do remove_intention(sell_gold, true);
            gold_sold <- gold_sold + gold_transported;
            gold_transported <- 0;
        }
    }

    norm share_information intention:share_information threshold:
    thresholdNorm instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
    each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
    mine_at_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        loop known_empty_goldmine over: get_beliefs_with_name(
    empty_mine_location) {
            ask my_friends {
                do add_belief(known_empty_goldmine);
            }
        }

        do remove_intention(share_information, true);
    }

    plan share_information_to_friends intention: share_information
    instantaneous: true{
        list<miner> my_friends <- list<miner>((social_link_base where (
    each.liking > 0)) collect each.agent);
        loop known_goldmine over: get_beliefs_with_name(
    empty_mine_location) {
            ask my_friends {
                do add_belief(known_goldmine);
            }
        }
        do remove_intention(share_information, true);
    }

    aspect default {
      draw circle(200) color: mycolor border: #black depth: gold_sold;
    }
}
```

```
experiment GoldBdi type: gui {
    output {
        display map type: opengl
        {
            species market ;
            species goldmine ;
            species miner;
            species policeman aspect:base;
        }

    }
}
```

Back to the start of the tutorial 1. Creation of the basic model: gold mines and market 2. Definition of the BDI miners 3. Definition of social relations between miners 4. Use of emotions and personality for the miners

Part X

Pedagogical materials

Chapter 150

Some pedagogical materials

Initiation to algorithms with Scratch

A set of exercices for your first step to algorithms using the graphical tool Scratch: PDF.

Memo GAML

A summary of the organization of a GAML model, its main parts and the main keywords, statements: PDF.

Exercice (*): Firefly synchronization From UML diagram, implement a GAMA model.

- **Keywords:** grid, displays, plot, synchronization.
- **Subject:** PDF, Keynote, PPTX
- **UML diagrams:** asta version
- **A model:** gaml file

Exercice (*): Firefighter model Implement the model given in the model description file. The guide file helps you to separate the implementation of the structure of the model, its initialization, its dynamics and ways to visualize it.

- **Keywords:** grid, inheritance, displays, plot, 3D.
- **Model description**: PDF
- **Guide:** PDF, Keynote, PPTX
- **UML diagrams:** asta version
- **A model:** gaml file

Exercice (**): Wolves, Goats, Cabbages model Implement the model given in the model description file. The detailed subject divides the model in 3 steps and contains helps and advices to implement the model.

- **Keywords:** grid, inheritance, displays, plot, prey-predator model.
- **Model description**: PDF
- **Detailed subject:** PDF, Word
- **Guide:** PDF, PPTX
- **A model:** gaml file

Exercice (**): Schelling model

- **Keywords:** grid, GIS data, displays, plot, Graphical modeling, Schelling model.
- **Subject:** PDF, Keynote, PPTX
- **A model:** gaml file

Exercice (**): Traffic model

- **Keywords:** GIS data, graph, skills, moving skill, displays, plot, mobillity model.
- **Subject:** PDF, Keynote, PPTX
- **A model:** gaml file

Exercice (***): Shortest path on grid by distance diffusion

- **Keywords:** grid, move, displays, diffusion model, algorithm.
- **Subject:** PDF, Word
- **A model:** gaml file

Part XI

Extensions

Chapter 151

Extensions

Here's the community list of existing extensions. Feel free to add yours.

If you want to add your extension, please follow this structure : - A title (the name of your extension) - A short description (explain what your extension could be used for) - An image (optional) - A link (to your website, a download page or whatever relevant link)

Thanks!

RAMA

Chapter 152

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.

Chapter 153

Installing the GIT version

Tested on MacOS X (10.14.4)

Important note: *the current Git version is* ***not*** *compatible with the* ***GAMA 1.6.1 release*** *and* ***neither*** *with the* ***GAMA 1.7RC2 release****.*

Install Eclipse 2019-03

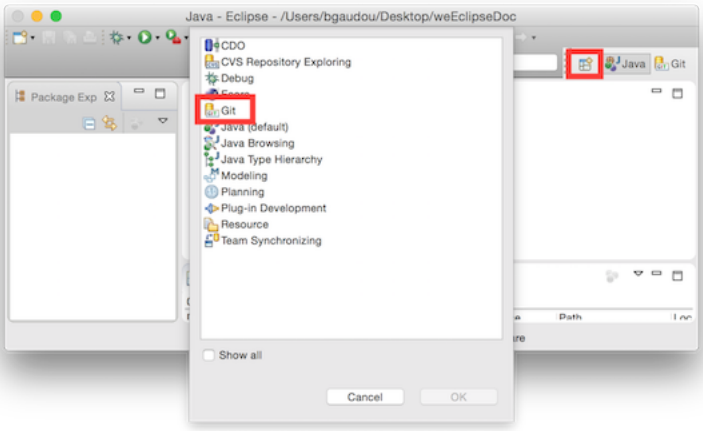
Download the "Installer of 2019-03" and choose to install the **Eclipse DSL TOOLS** version. This is the latest version under which GAMA is certified to work. Alternatively, you can directly download the "Eclipse IDE for Java and DSL developers" package. Regarding Java, we **strongly** recommend to install the Java Oracle 1.8 JDK that can be downloaded here.

Install GAMA source code

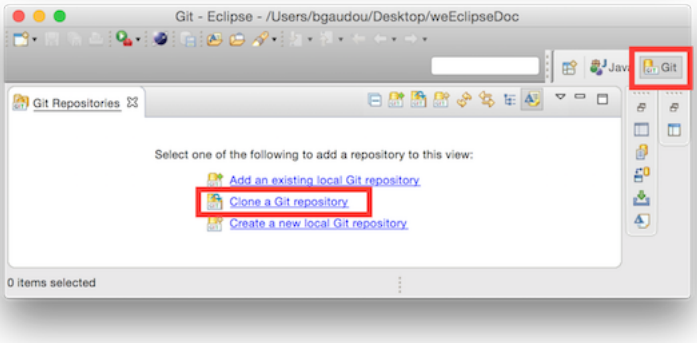
The source is to be downloaded from GitHub in two steps: by creating a local clone of the GitHub repository and then importing the different projects that constitute GAMA into the Eclipse workspace.

1. Open the Git perspective:

- Windows > Perspective > Open Perspective > Other...



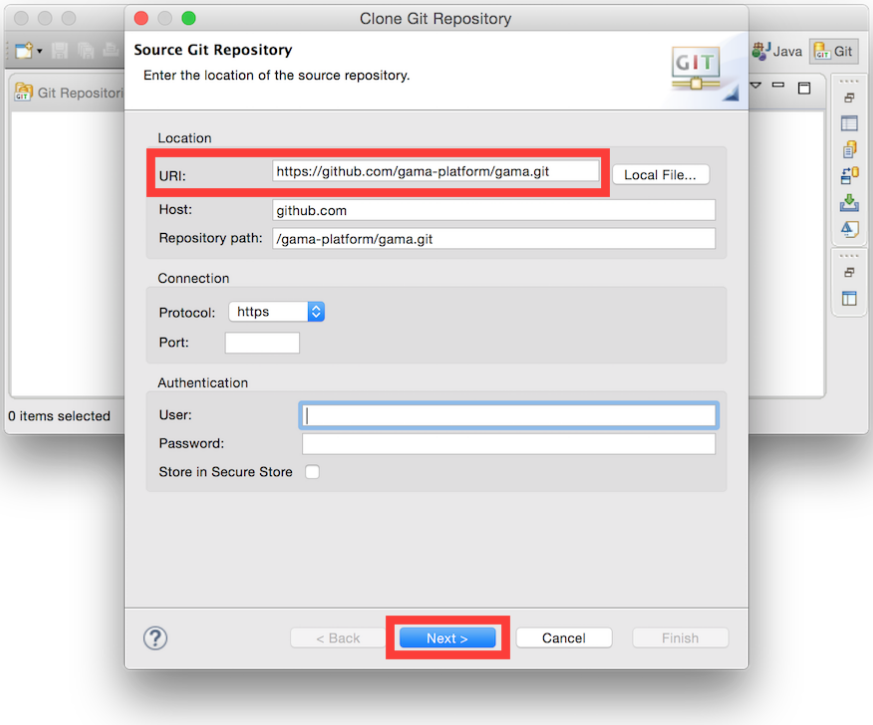
– Choose `Git`



2. Click on “Clone a Git repository”

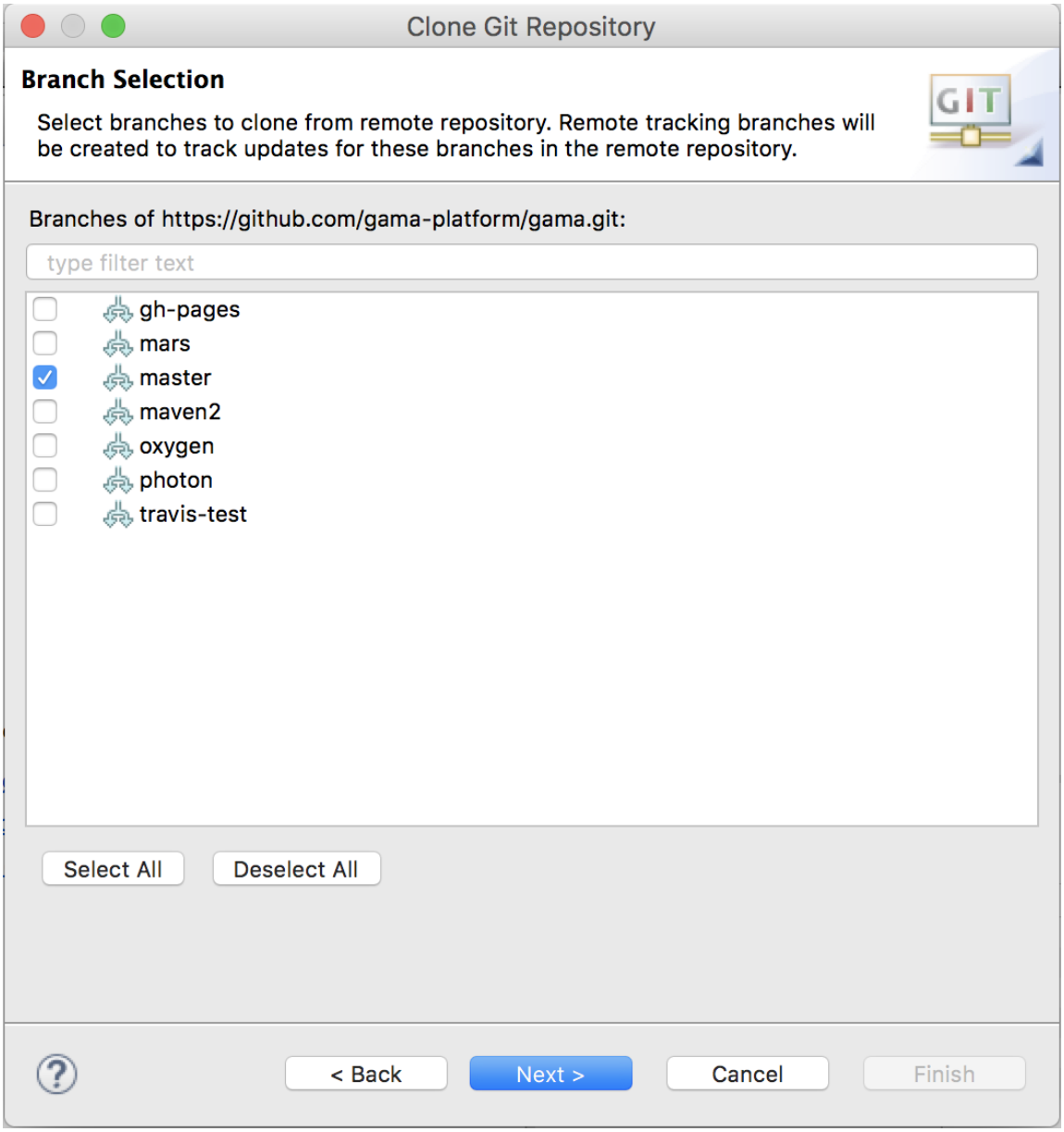
- In **Source Git repository** window:

  – 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 master branch
* 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...

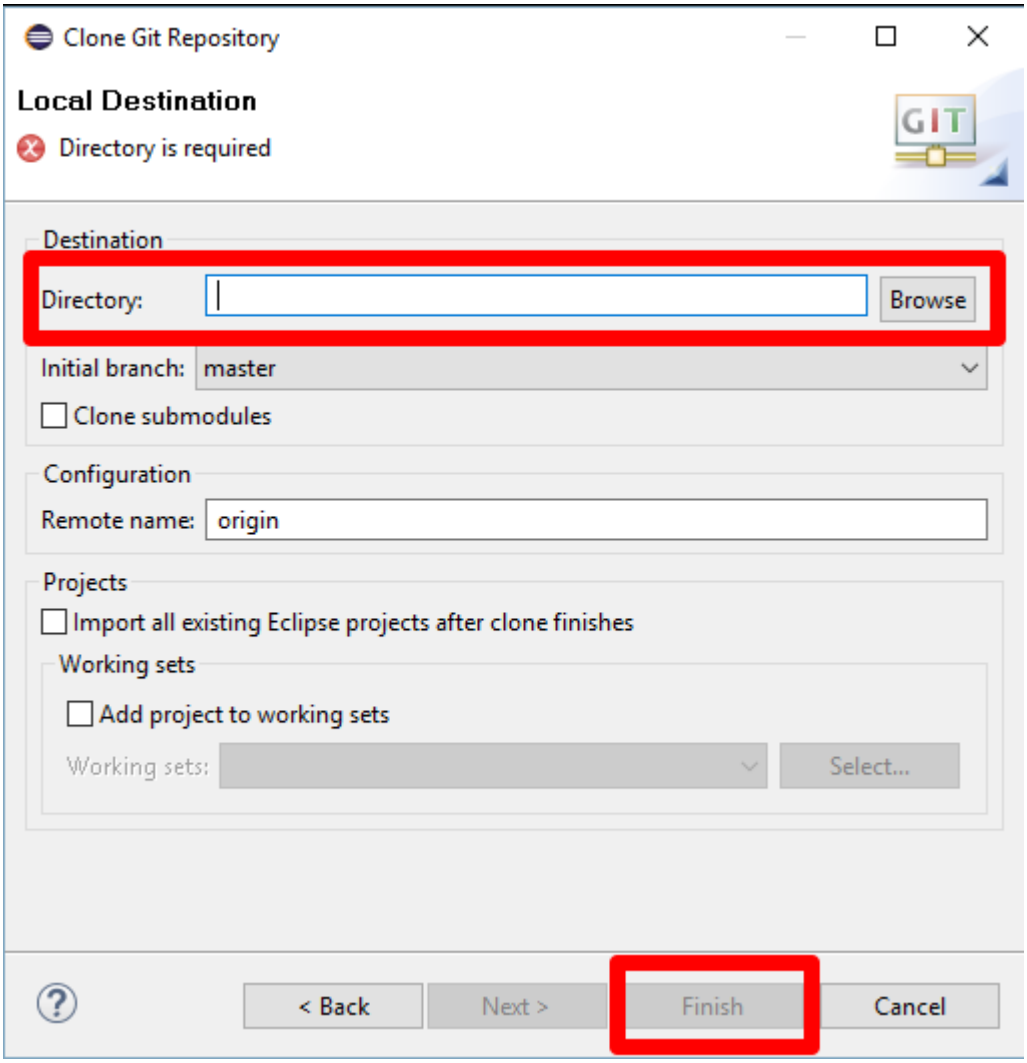


Figure 153.1: Local destination

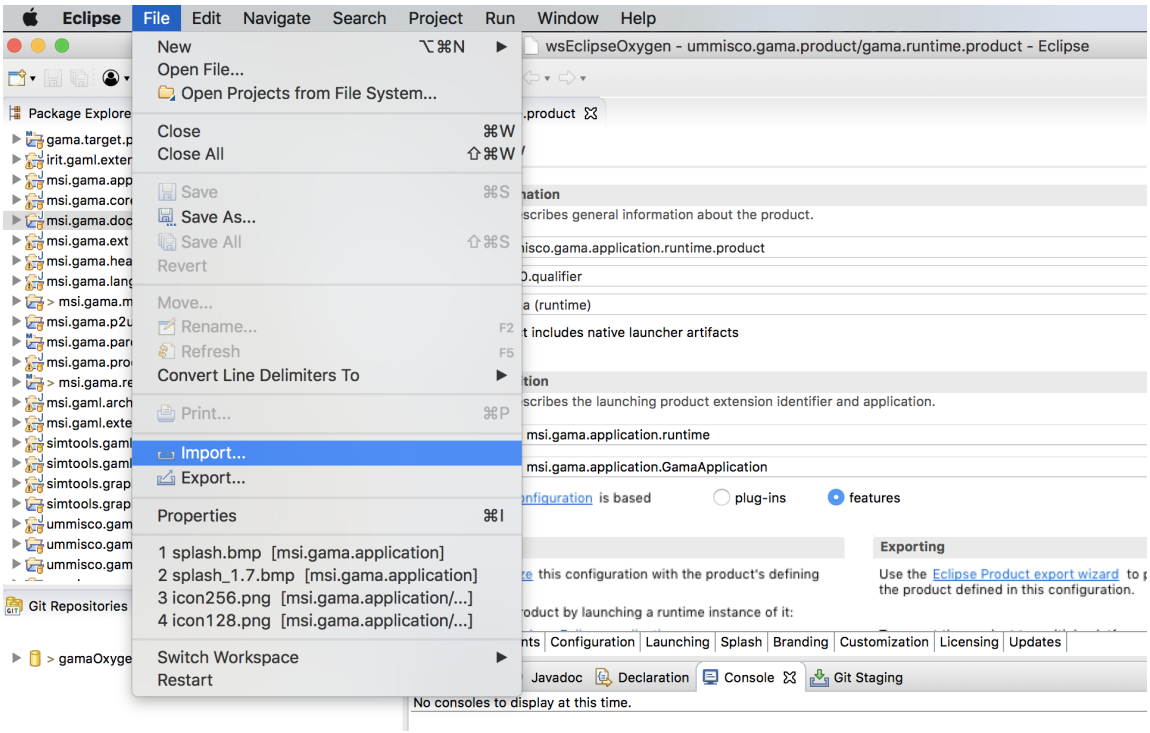


Figure 153.2: Context Working tree

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.`,

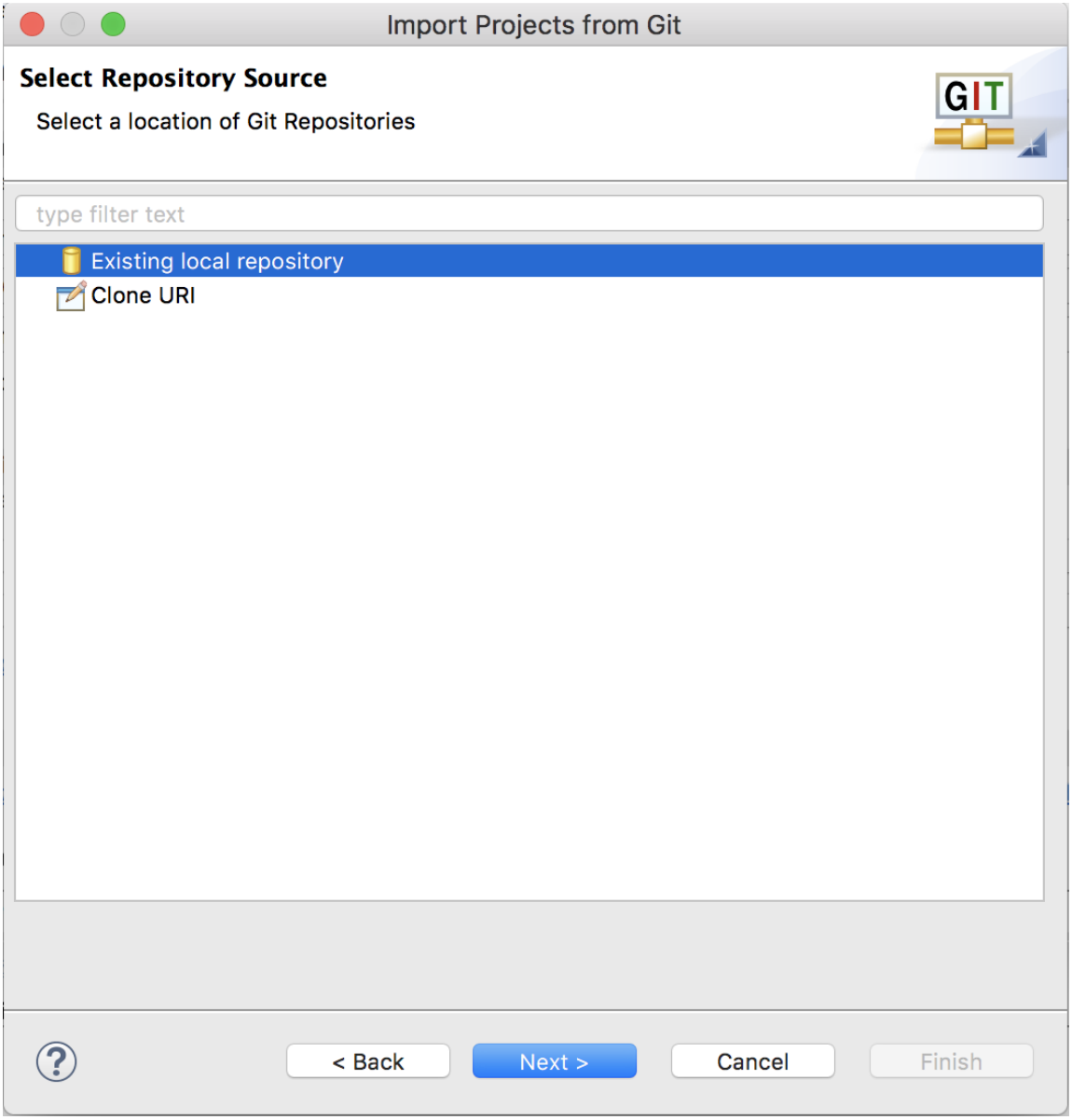
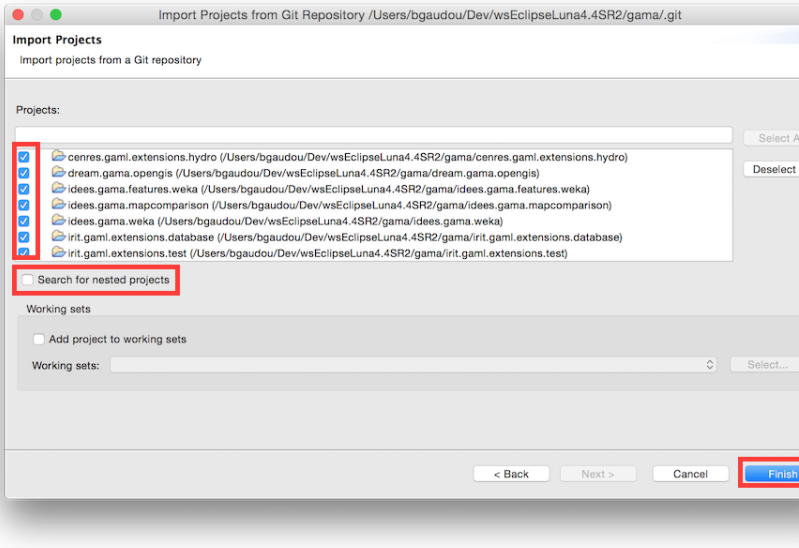


Figure 153.3: Context Local Repository

- 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 is targeting JDK 1.8, and Eclipse will produce errors if it not found in your environment. So, either you set the compatibility to

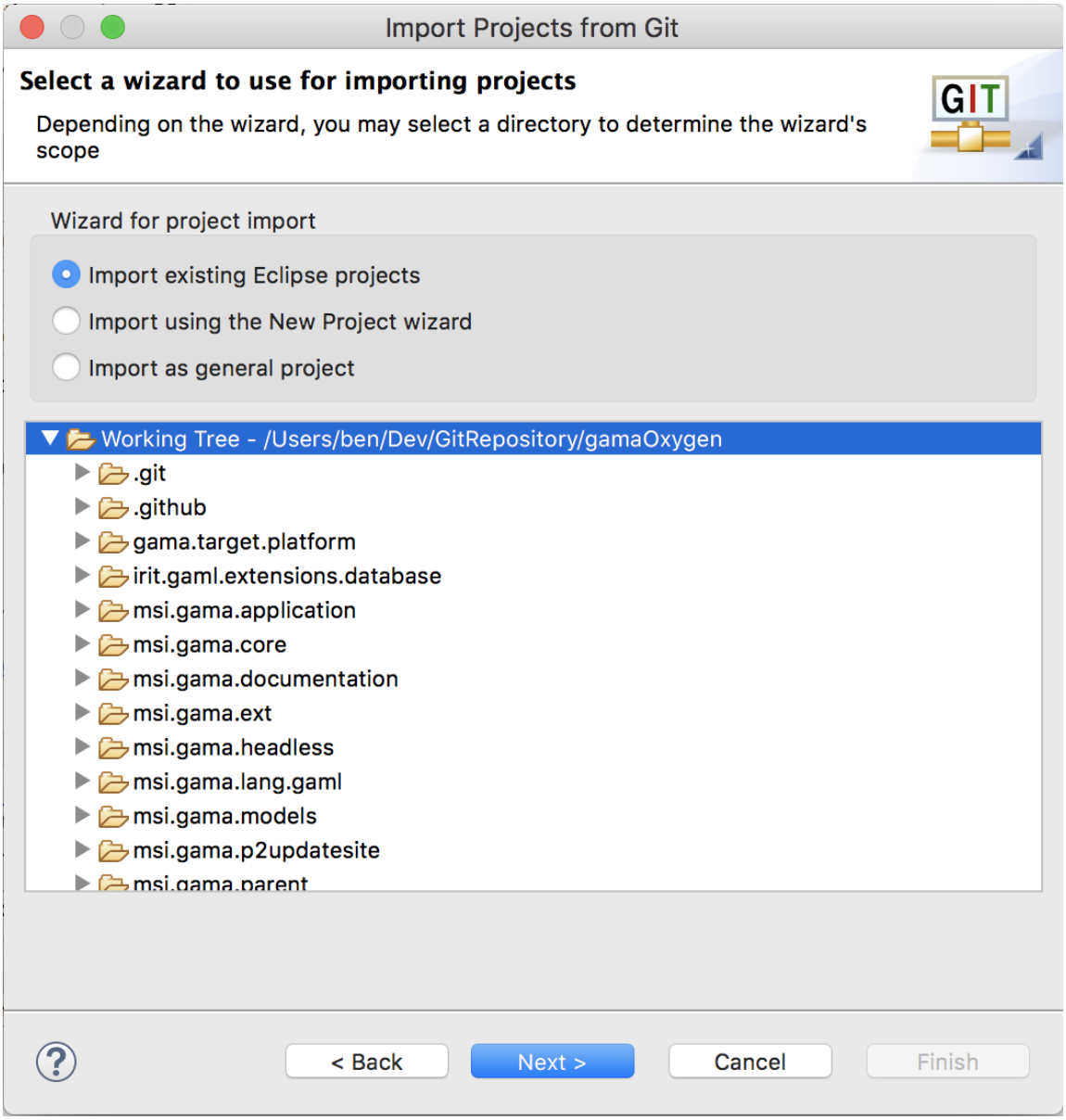


Figure 153.4: GIT Import projects

1.8 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).

Run GAMA

1. In the `ummisco.gama.product` plugin, open the `gama.runtime.product` file (`gama.product` is used to produce the release).
2. Go to "Overview" tab and click on Synchronize
3. Click on Launch an Eclipse Application

GIT Tutorials

For those who want 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=PL1F56EA413018EEE

Chapter 154

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

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`, wich 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

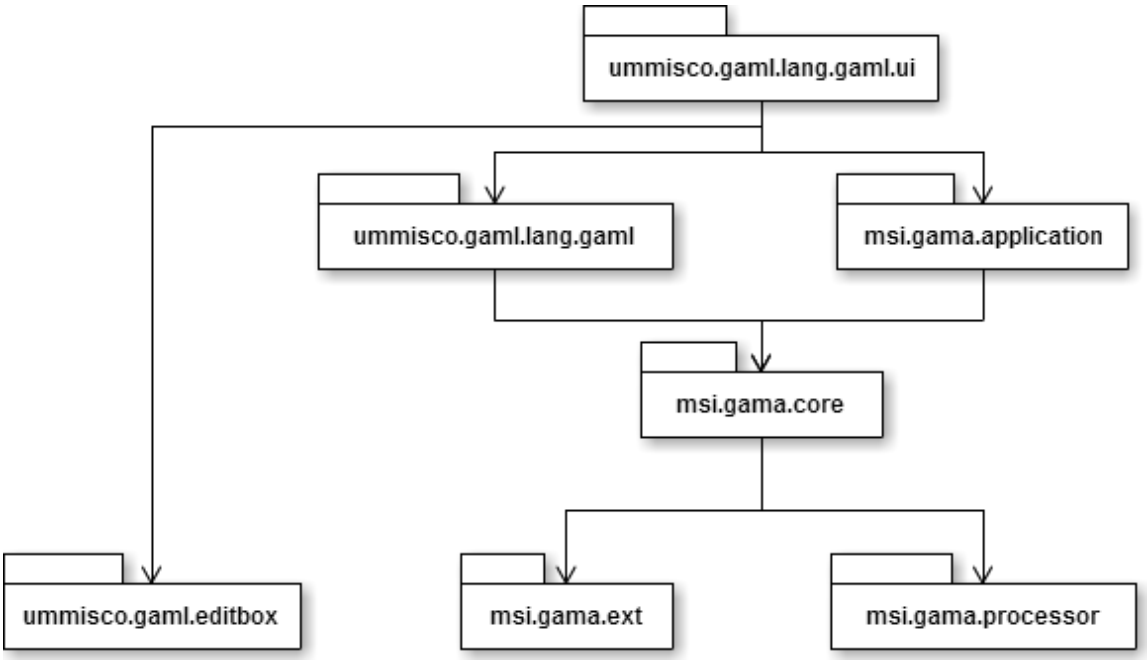


Figure 154.1: Minimal configuration projects dependencies

    * `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 (depreciated, 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 concluant, 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 graph-analysis 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

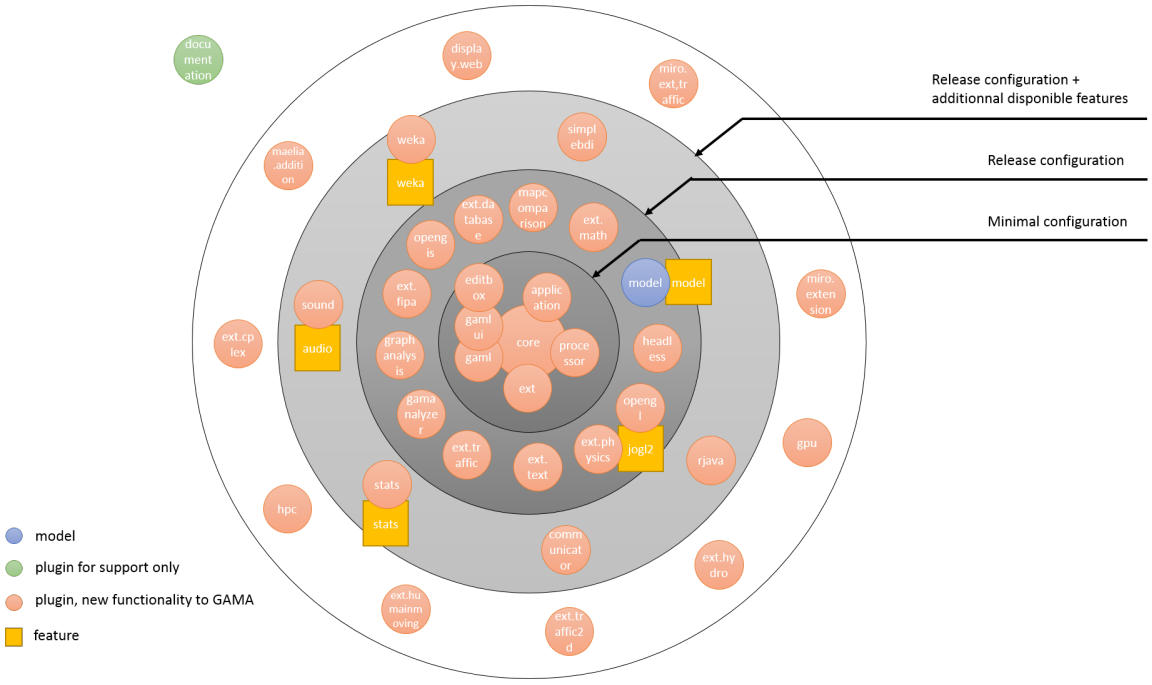


Figure 154.2: Global architecture of GAMA

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)

Chapter 155

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 detailled in this page).

Creation of a plug-in

Here are detailled steps to create and configure a new GAMA plug-in.

- File > New > Project > plug-in project
  - In the "New plug-in Project" / "Plug-in project" window:
    - Choose as **name** « name_of_the_plugin » (or anything else)
    - Check "Use défaut location"
    - Check "Create a Java Project"
    - The project should be targeted to run with Eclipse
    - working set is unchecked
    - Click on "Next"
  - In the "New plug-in Project" / "Content" window:
    - Id : could contain the name of your institution and/or your project, e.g. « irit.maelia.gaml.additions »

        * version 1.0.0.qualifier (this latter mention is important if you plan on distributing the plugin on GAMA update site)
        * Name « Additions to GAML from Maelia project »
        * Uncheck “Generate an activator, a Java class that controls the plug-in’s life cycle” ,
        * Uncheck “This plug-in will make contributions to the UI”
        * Check “No” when it asks “Would you like to create a rich client application ?”
        * Click on “Next”
    - In the “New plug-in Project” / “Templates” window:
        * Uncheck “Create a plug-in using one of the templates”
        * Click on “Finish”

Your plug-in has been created.

- Edit the file “Manifest.MF”:
    - Overview pane:
        * check « This plug-in is a singleton »
    - Dependencies pane:
        * 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”.
    - Runtime pane:
        * In exported Packages: nothing (but when you will have implemented new packages in the plug-in you should add them there)
        * Add in the classpath all the additional libraries (.jar files) used in the project.
    - Extensions pane:
        * “Add” “gaml.extension”
    - Save the file. This should create a “plugin.xml” file.
    - Select the project and in menu Project > Properties:

    * Java Compiler > Annotation Processing: check "Enable project specific settings", then in "Generated Source Directory", change ".apt_generated" to "gaml",
    * Java Compiler > Annotation Processing > Factory path: check "Enable project specific settings", then "Add Jars" and choose "msi.gama.processor/processor/plugins/msi.gama.processor.1.4.0.jar"
    * Close the menu. It should compile the project and create the `gaml` directory.
    * Return in the Project > Properties Menu.
    * In Java Buildpath > Source, check that the gaml directory has been added. Otherwise click on Add Folder and select the gaml directory

The plug-in is ready to accept any addition to the GAML language, e.g. skills, actions, operators.

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 detailled 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

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.

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”.

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 an 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
        - Richt-click on the project, Team > Update
        - Richt-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.

Chapter 156

Developing a New Skill

A skill adds new features (attributes) and new capabilities (actions) to the instances of the species that use it.

Defining the class

A Skill is basically a **singleton** and **stateless** Java class that: * extends the abstract class `Skill`, * begins by the annotation [@skill](DevelopingIndexAnnotations#@skill): `@skill(name = "name_of_the_skill_in_gaml")`.

Note: GAMA annotations are classes defined into the `msi.gama.precompiler.GamlAnnotations` class.

Defining new attributes

To add new attributes to the species that declares this skill, developers have to define them before the class using the [@vars](DevelopingIndexAnnotations#@vars) and `@variable` annotations. The `@vars` annotation contains a set of `@variable` elements.

In a [@variable](DevelopingIndexAnnotations#@variable) element, one has to define the name, the type and the default value of the attribute. For example in MovingSkill:

```
@vars({
  @variable(name = IKeyword.SPEED, type = IType.FLOAT, init = "1.0"),
```

```
  @variable(name = IKeyword.HEADING, type = IType.INT, init = "rnd(359)
    ")
})
```

In order to detail how to access these new attributes (if needed), developers have to define a getter (using `@getter`) and a setter (using `@setter`) methods. If no getter (and setter) is defined, the attribute can nevertheless be set and get, using implicit by default getter and setter. But as soon as a getter and/or a setter is defined, they replace the implicit default ones. For example:

```
@getter(IKeyword.SPEED)
public double getSpeed(final IAgent agent) {
    return (Double) agent.getAttribute(IKeyword.SPEED);
}

@setter(IKeyword.SPEED)
public void setSpeed(final IAgent agent, final double s) {
    agent.setAttribute(IKeyword.SPEED, s);
}
```

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.

The developer can also define parameters for this action using the annotation `@arg` will a set of parameters names. For example, the action `goto` of the MovingSkill is defined as follows:

```
@action(name="goto", args={
    @arg(name = "target", type = { IType.AGENT, IType.POINT,IType.
   GEOMETRY }, optional = false),
    @arg(name = IKeyword.SPEED, type = IType.FLOAT, optional = true),
    @arg(name = "on", type = { IType.GRAPH }, optional = true)
    }
)

public IPath primGoto(final IScope scope) throws GamaRuntimeException {
...
}
```

It is called in GAMA models with:

```
do goto (target: the_target, on: the_graph);
```

or

```
path path_followed <- self goto (target: the_target, on: the_graph,
    return_path: true);
```

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 associated with an action has to return a non-void object. * the method should have only one parameter: the scope (type IScope). * the method can only throw `GamaRuntimeException`s. Other exceptions should be caught in the method and wrapped in a `GamaRuntimeException` before being thrown.

Annotations

@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.*

@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.*

@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.*

@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.

@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.*

@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.*

All these annotations are defined in the `GamlAnnotations.java` file of the `msi.gama.processor` plug-in. # 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.

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](DevelopingIndexAnnotations#@symbol): `@symbol(name = "name_of_the_statement_gaml", kind = "kind_of_statement", with_sequence = true/false),`
    * [@inside](DevelopingIndexAnnotations#@inside): `@symbol(kinds = {"kind_of_statement_1","kind_of_statement_2","..."}`

In addition the 4 following optional annotations can be added:

- [@facets](DevelopingIndexAnnotations#@facets): to describe the set of [@facet](DevelopingIndexAnnotations#@facet) annotations,
  - [@doc](DevelopingIndexAnnotations#@doc): to document the statement.
  - [@serializer](DevelopingIndexAnnotations#@serializer): in addition, statements can benefit from a custom serializer, by declaring `@serializer(CustomSerializer.class)`, with a class extending `SymbolSerializer`.
  - [@validator](DevelopingIndexAnnotations#@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 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 omissible.

```
@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)
   }, omissible = 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 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,
    internal = true),
    @facet(name = IKeyword.INDEX, type = IType.TYPE_ID, optional = true
    , internal = true),
    @facet(name = IKeyword.VIRTUAL, type = IType.BOOL, optional = true)
    }, omissible = IKeyword.NAME)
@validator(ActionValidator.class)
@serializer(ActionSerializer.class)
public class ActionStatement extends AbstractStatementSequenceWithArgs
    {
```

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)
    }, omissible = IKeyword.NAME)
public class SingleStatementExample extends AbstractStatement {
```

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 = stack.getAgentScope();
    String nameStr = null;
    if ( agent != null && !agent.dead() ) {
        nameStr = Cast.asString(stack, name.value(stack));
        if ( nameStr == null ) {
            nameStr = "nil";
```

```
        }
        GuiUtils.informConsole(nameStr);
    }
    return nameStr;
}
```

The variable `scope` of type `IScope` can be used to:

- get the current agent with: `scope.getAgentScope()`
  - 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 wether or not a sequence can or should follow the symbol denoted by this class.*
  - **with_args** (boolean, false by default): *indicates wether 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 serial-

izer, 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.

Chapter 157

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](DevelopingIndexAnnotations#@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.

Chapter 158

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 requiers 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](DevelopingIndexAnnotations#@vars) annotation to describe the attributes of a complex type. The `@vars` annotation contains a set of `@var` elements.

```
@vars({ @var(name = IKeyword.COLOR_RED, type = IType.INT), @var(name =
    IKeyword.COLOR_GREEN, type = IType.INT),
     @var(name = IKeyword.COLOR_BLUE, type = IType.INT), @var(name =
    IKeyword.ALPHA, type = IType.INT),
     @var(name = IKeyword.BRIGHTER, type = IType.COLOR), @var(name =
    IKeyword.DARKER, type = IType.COLOR) })
public class GamaColor extends Color implements IValue {
```

It can contain setter and/or getter for each of its attributes. Setters and getters are methods annotated by the [@getter](DevelopingIndexAnnotations#@getter) or [@setter](DevelopingIndexAnnotations#@setter) annotations.

```
@getter(IKeyword.COLOR_RED)
public Integer red() {
    return super.getRed();
}
```

In addition 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. To this purpose the following method needs to be implemented:

```
public abstract String stringValue(IScope scope) throws
    GamaRuntimeException;
```

The type file

The class representing the type is a Java class such that:

- the class should be annotated by the [@type](DevelopingIndexAnnotations#@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)
```

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.

Chapter 159

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. mulit-criteria decision-making, 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](DevelopingIndexAnnotations#@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](DevelopingIndexAnnotations#@vars) (and one embedded [@var](DevelopingIndexAnnotations#@var) per additional attribute) annotation on top of the class,
  - to define [@setter](DevelopingIndexAnnotations#@setter) and [@getter](DevelopingIndexAnnotations#@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](DevelopingIndexAnnotations#@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.

Chapter 160

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](DevelopingIndexAnnotations#@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...).

Chapter 161

Index of annotations

Annotations are used to link Java methods and classes to GAML language.

@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](#species) or [@skill](#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](#arg), empty by default): *the list of arguments passed to this action. Each argument is an instance of arg.*
  - **doc** (set of [@doc](#doc), empty by default): *the documentation associated to the action.*

@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](#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](#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](#example), empty by default): *an array of String representing some examples or use-cases about how to use this element.*
  - **usages** (set of [@usage](#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 into 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](#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](#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 files. 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](#doc), empty by default): *the documentation attached to this operator.*

@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 shoud also be used as an initializer

@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).*

@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 belong (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](#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 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.

@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](#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](#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 wether or not a sequence can or should follow the symbol denoted by this class.*
  - **with_args** (boolean, false by default): *indicates wether 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](#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_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](#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 unecessary 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 initialze 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 be 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](#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 descried using a @pattern (rather than an array of [@example](#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 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](#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.*

Part XII

Developing GAMA

Chapter 162

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.

Chapter 163

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

Chapter 164

Concepts

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 unconsistency 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 Jstor 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
```

Chapter 165

Packages

Core

The main plugin of the GAMA Platform that defines the core functionalities: most Gaml operators, statements, skills, types, etc.

Metamodel

`IAgent`, `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

Chapter 166

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();
```

Chapter 167

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"

Table of contents

Chapter 168

Documentation

The GAMA documentation comes in 2 formats: a set of wiki files availaible 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.documetation` plugin) to organize the wiki files in a pdf file.

- `MainGeneratePDF.java` file parsers 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 generate 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_deplicates 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 annoted 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 classified 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 annoted 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 annoted 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

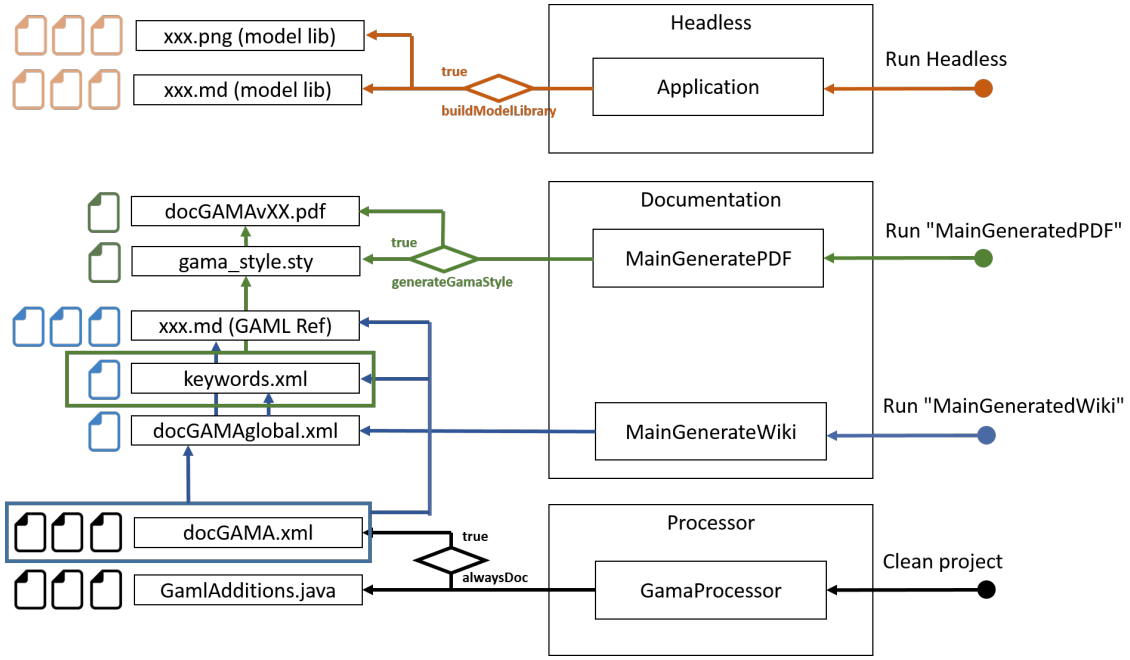
If you make some modifications in the plugin processor, you have to rebuild the .jar file associated to the processor to take into account the changes. Here are the several steps you have to do:

- In the "processor" plugin, open the *plugin.xml*.
- In exporting (from the *Overview* tab), click on *Export Wizard*.
- In the tab "Destination", choose processor/plugins for the directory. In the tab "Options", delete the field *Qualifier replacement*. Click "finish".
- Right click on the folder "processor" to refresh. –> It's ok !

Chapter 169

General workflow of file generation

This following diagram explains roughly the workflow for the generation of the different files:



Chapter 170

How to write the Website Content

In this page, we will explain all about the convention we use to write and generate the website content and the wiki content. Since the release of GAMA 1.7, with the new GAMA website, we have two contents:

- The *wiki* content is hosted in github, witch directly interpret the markdown format of the files to display them in a proper way. This wiki, since it is a wiki, can be edited by any user. It is then, in constant changes.
- The *website* content is the content of the real GAMA website. It is a verified and fixed version of the documentation (usually a re-generation of the website content is done when there is a new release of the software)

Index

  - Website Database
  - Loading the Database

- Manage concepts keywords

Requirements

To generate automatically the documentation, the GAMA Git version is required. See Install Git version for more details.

Among all the GAMA plugins, only one is related to documentation generation:

- `msi.gama.documentation`: it contains some useful java scripts to help you to write a correct documentation.

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.

gama.wiki tree structure

The "gama.wiki" plugin contains all the wiki content, and almost all the website content. It contains a folder content which contains the following folders:

- Tutorials
  - LearnGAMLStepByStep: contains the linear documentation to learn about the concepts of GAML
  - Recipes: contains short pieces of documentation answering a specific problematic
  - Tutorials: contains applicative tutorials
- References
  - ModelLibrary: contains the model library (only present in the website)
  - PlatformDocumentation: contains the documentation dealing with how to use the platform
  - GAMLReferences: contains GAML references

  - PluginDocumentation: contains the documentation of the additional plugins

- Community

  - Projects: contains a presentation of the projects where GAMA is involved (only present in the website)
  - Training: contains a presentation of the training sessions organized by the GAMA team (only present in the website)

- WikiOnly: contains the content only present in the wiki, and not in the website

  - DevelopingExtensions: contains explanations about how to extend the platform

- resources: contains all the additional resources needed (images, pdf...)

For the rest of this document, the highest level of tree structure ("Tutorials"/"References"/"Community"/"WikiOnly") will be named as **tabs**. The level just under will be named as **sections**, and the level under will be named as **sub-section**. All this content is written using the markdown format. All the images resources are included in the *resources/images* folder. They are actually under different subfolders. From the markdown page, you can call the resource with the relative path `resource/images/sub_folder/image_name.png`. If a *section/sub-section* contains one of several sub-division, then those sub-divisions will be stored in a folder with the name of the corresponding *section/sub-section*, and this *section/sub-section* folder will be associated with a markdown file with the same name (indeed, a *section/sub-section* has its own page). If a *section/sub-section* has no sub-division, then this *section/sub-section* is simply defined with a markdown file containing the content wanted.

Notice that there is some content which is present only in the wiki (the "WikiOnly" content), some content present only in the website (the model library, most of the community content...). In fact, the wiki tree structure is determined by the file _Sidebar, while the website tree structure is determined by the file *WebsiteTreeStructure*.

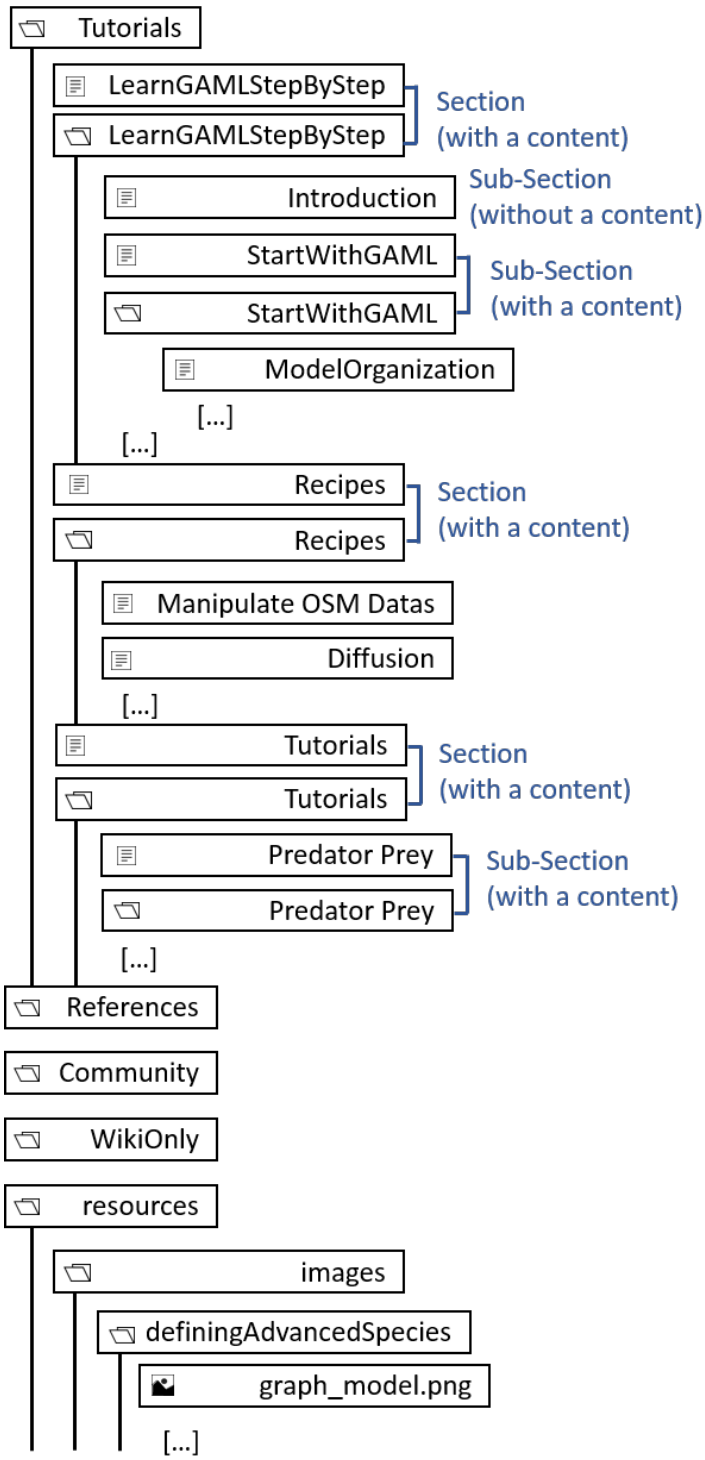


Figure 170.1: resources/images/developpingExtension/tree_structure.png

Good practices when writing markdown files

Title

Each markdown files **has to** start with a title in the markdown format (like `# title`). This is this title which will be displayed in the tree structure of the website.

Hypertext Links

Even if the repository how have a more complexe tree structure, you don't have to (and you must not !) speficy the relative or absolute path to the target page, just naming the page will work : `[text_displayed](the_name_of_the_md_file)`

Images Links

As already said in the previous paragraph, images have to be in an "resources/images/folder_name" folder next to your md file, so that you can write the relative path more easily.

Insert Metadatas

Metadatas in content files are written as comments, with the following syntax:

```
[//]: # (name_of_the_medatada|value_of_the_metadata)
```

Medatadas are not displayed in the wiki and the website content. For the website generation, metadatas are used in order to build the database, most of all to manage the search engine, and the learning graph.

Here is the list of metadata we use in the content files:

- **keyword** : will write an invisible anchor in this exact place in the website. When the user will do a research about this word, he can access directly to this anchor.
- **startConcept/endConcept** : used to delimit a concept. The value of those two metadatas is the name of the concept. All the concepts are listed in the file "DependencyGraph", in the content folder in your wiki repository.

keyword

The value of the keyword has to have this structure : keyword_category_keyword_name (indeed, several keywords can have the same name ! The type of the keyword has to be specified). Here is the list of the several keyword categories : concept, operator, statement, species, architecture, type, constant and skill. Example of metadata : `[//]: # (keyword|concept_3D)`, or `[//]: # (keyword|operator_int)`.

startConcept/endConcept

The value of the keyword have to be one of the values defined in the file learningConcept.xml.

Notice that a *concept* in the meaning of keyword is not the same as a *concept* (or *learning concept*) in the learning graph! Please read the part concerning the database to learn more about it.

Website generation workflow

This part is not implemented yet, it is under construction.

The gama.documentation plugin

This plugin is used to generate GAML documentation automatically in the markdown format, and copy paste the content to the wiki folder. The plugin is also used to generate the model library in the markdown format, with the source code, a quick description, and an image (screenshot). In the same time, the plugin generates a html page (an "abstract") and put it directly in the model folder (in order to be loaded directly from GAMA).

The documentation plugin contains also 2 other scripts which helps to create content:

learningGraphDatabaseGenerator

The **learningGraphDatabaseGenerator** script is used to generate the "*nodesDatabase.js*" file, which is read to visualize the learning graph. This script needs the

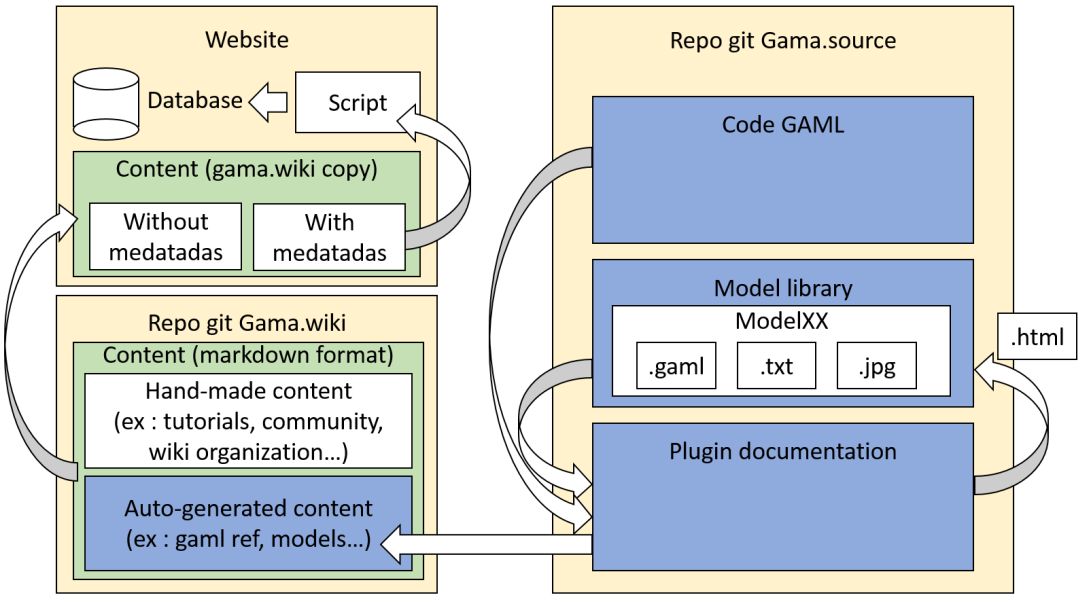


Figure 170.2: resources/images/developpingExtension/website_generation_workflow.png

"learningGraph.xml" file as input. Each learning concepts has an id, a name, a very short description, a position (position in x and y in %, between 0 and 1. This value is then multiplied by the **coeff** in the learningGraphDatabaseGenerator), and a list of prerequisite learning concepts. A category of learning concept (also defined in the learningGraph.xml file) has an id, a position (position in x and y), a position for the hallow (position in x and y of the big circle displayed when a category is selected), a size for the hallow, a color, a name, and the list of learningConcept associated.

modelLibraryGenerator

The **modelLibraryGenerator** script is used to generate all the markdown files of the model library. This script

- Parse all the models of the model library, and build an "input" xml file for a headless execution (this file is deleted at the end of the execution).
  - By default, this will ask to execute all the experiments for each model, and take a screenshot of the 10th cycle for each display.

  - You can change this default behavior by changing the file "**modelScreenshot.xml**", in the wiki repo (see description below)

- Execute the headless
- Copy-paste all the generated images in the write folder, with the write names.
- Browse a second time all the models, build the md file, including the screenshot computed from the headless execution, and analyzing the header of each model to extract the title, author and tags. Each md files respects the following format : path_from_model_with_underscore_instead_of_slash_without_strange_char + ".md". (ex : "Features/3D/3D Visualization/models/3D camera and trajectories.gaml" becomes "Features_3D_Visualization_models_3D_camera_and_trajectories.md".)

Format of the xml file to "tune up" the screenshot generation :

```
<xmlFile>
  <experiment id="name_of_the_file_without_extention"+" "+"model_name"+
   " "+"experiment_name">
    <display name="display_name_1" cycle_number="
   number_of_the_cycle_for_the_screenshot"/>
    <display name="display_name_2" cycle_number="
   number_of_the_cycle_for_the_screenshot"/>
  </experiment>
</xmlFile>
```

TODO

The gama.wiki repository

This repository contains in on hand the content auto generated by the documentation plugin, and in the other hand a handmade content. All the content is in the markdown format, organized through a specific tree structure, sometime containing metadatas.

The website repository

This repository contains:

- A copy of the content of the wiki repo (copy/pasted manually to freeze a specific commit of the wiki)

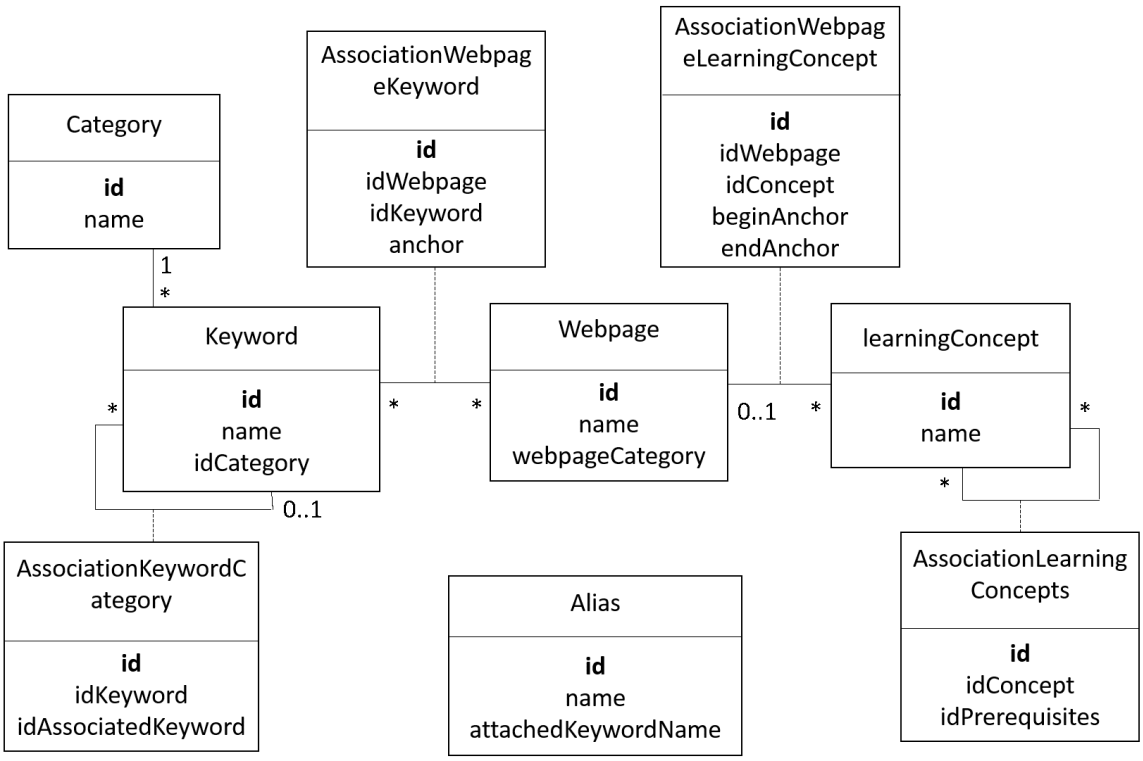


Figure 170.3: resources/images/developpingExtension/global_database.png

- A Database management system
- A script used to interpret the metadatas from the content, in order to load the database
- Some pages which are not in the wiki repo, and some heavy resources (such as videos)

Website database

Keyword

A **keyword** is a keyword that can be used for search, either manually (the user enters the wanted keyword in the searchbar of the website) or automatically (through the search tab in the navigation panel) A keyword in attached with a category (among the following names : concept, type, operator, statement, species, architecture, constant, action, attribute, skill, facet).

- A keyword that is a ***concept*** can be linked with other keywords (ex : the keyword "BDI" will be linked with the keywords "eval_when", "get_priority"...)
- A keyword that is a ***facet*** is linked to a ***statement*** or a ***species*** keyword (ex : the keyword "torus" will be linked with the keyword "global").
- A keyword that is an ***action*** or an ***attribute*** is linked either to a ***skill*** keyword (if it is actually an action or an attribute of a skill), an ***architecture*** keyword (if it is an action or a behavior of an architecture), or a ***species*** keyword (if it is a built-in action or attribute).
- A keyword that is a ***statement*** can be linked to an ***architecture***.

A keyword is composed of:

- **id** (unique id)
- **name** (the word which is searched by the user)
- **idCategory** (id of the category)

A cagegory is composed of:

- **id** (unique id)
- **name** (the name of the category)

Alias

Another database is used to join an **alias** to an existing keyword. Ex: the word "alias" will be changed as "die".

An alias is composed of:

- **id** (unique id)
- **name** (name of the alias. ex : "kill")
- **attachedKeywordName** (name of the keyword attached. ex : "die")

Note that an alias does not know the id of the keyword, but only the name of the attached keyword(s). Indeed, the alias "integer" will give the keyword name "int", but several keywords correspond to the keyword name "int" (it can be the type "int", or the cast operator "int")

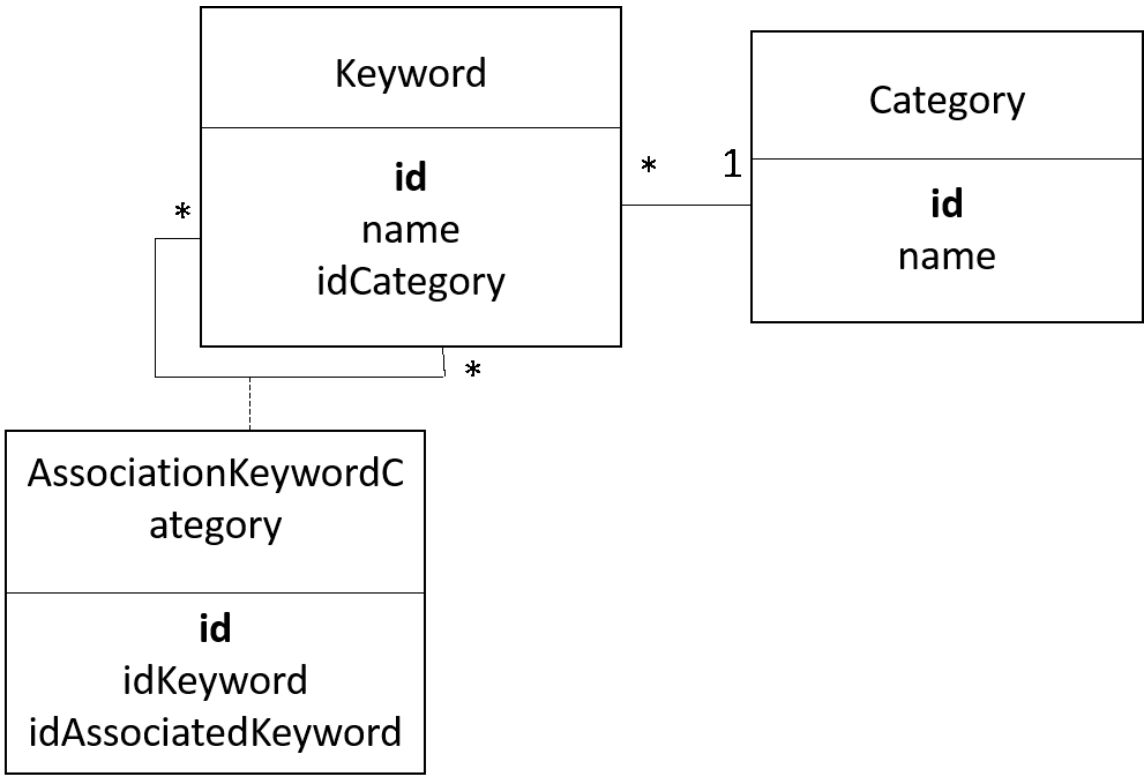


Figure 170.4: resources/images/developpingExtension/keyword_table.png

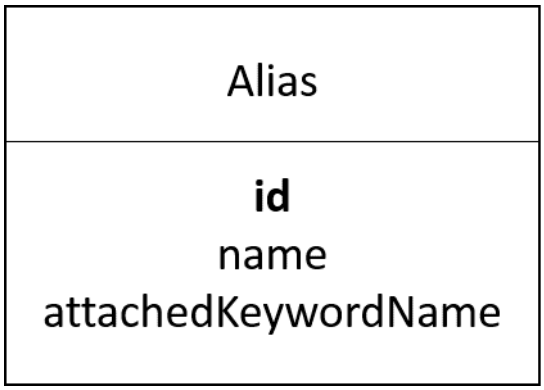


Figure 170.5: resources/images/developpingExtension/alias_table.png

Webpage

A **webpage** can be either a page of the model library, a page of the gaml reference, or an other page of the documentation.

A webpage is composed of:

- **id** (unique id)
- **name** (name of the webpage)
- **webpageCategory** (the name of the category of the webpage, a value among *modelPage*, *docPage*, gamlRefPage_).

The tables **webpage** and **keyword** are linked through an association table. This association table contains also an anchor (an anchor has a unique value) to the wanted page.

Note that only the keywords which have the category *concept*, *species*, *type*, *operator*, *skill* and *constant* can be attached to a webpage.

The keywords which have the category *action*, *attribute* and *facet* forward to the attached keyword.

The keywords which have the category *statement* are attached to a webpage only if they are not attached to another keyword. If they are attached to another keyword (an *architecture* keyword), then the *statement* keyword forward to the attached keyword.

LearningConcept

LearningConcept is used to build the learning graph (notice that a "learning concept" and a "keyword concept" is not the same thing !)

A LearningConcept is composed with:

- **id** (unique id)
- **name** (name of the learning concept)

A LearningConcept is linked to a webpage through an association table. This table is composed also with two anchors that are used to delimit the position of the learning concept in a page (the beginning position and the ending position).

A LearningConcept can be associated to other LearningConcepts through an association table, used to spefify the "prerequisite concepts".

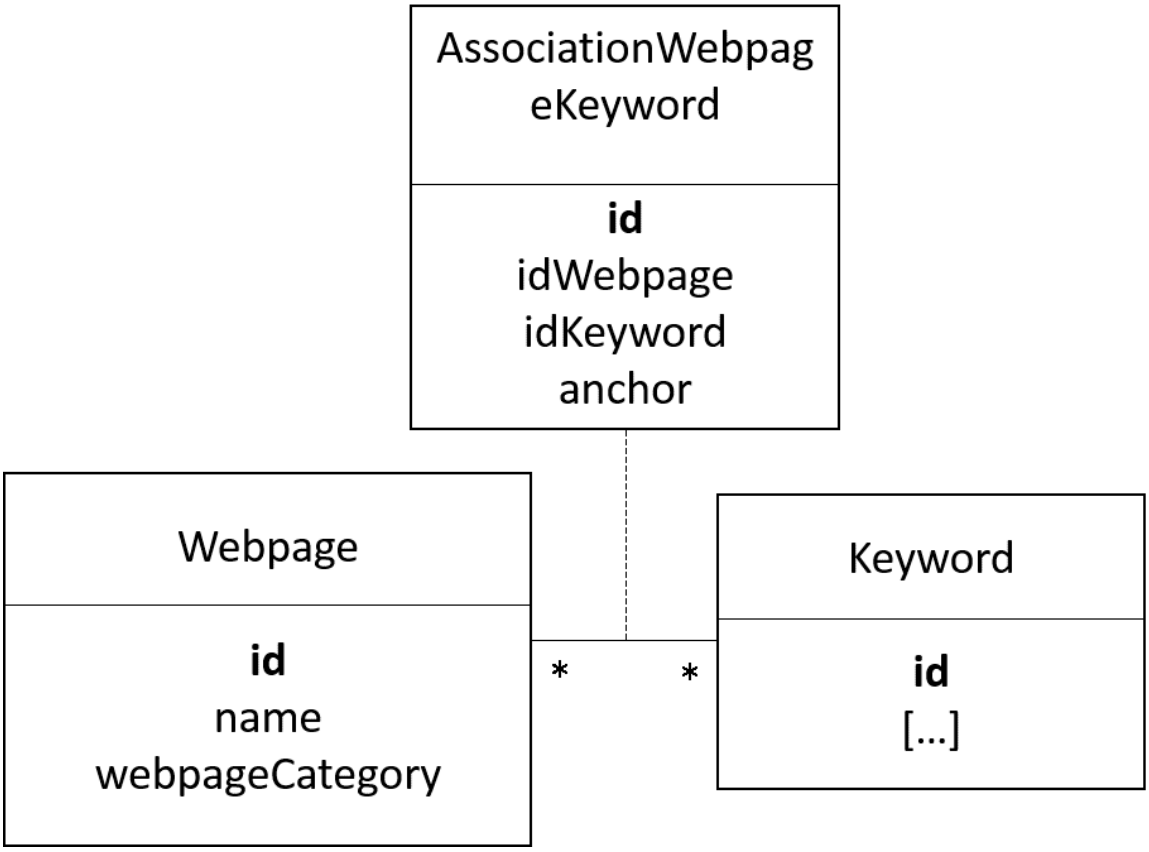


Figure 170.6: resources/images/developpingExtension/webpage_table.png

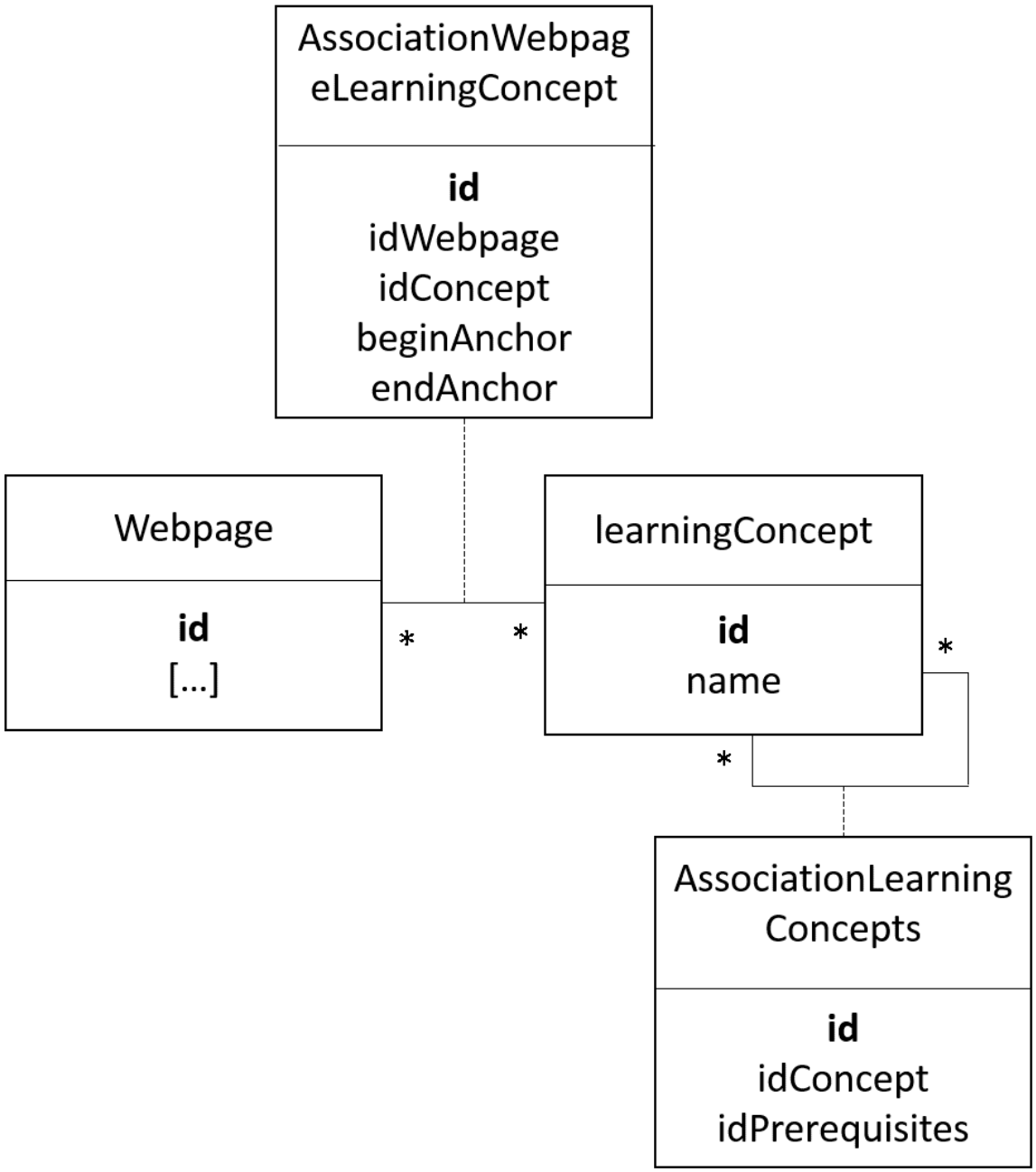


Figure 170.7: resources/images/developpingExtension/learningConcept_table.png

Loading the Database

The database is loaded from a gathering of independent files. Some of those files are handmade written, other are generated automatically.

Role of the documentation generation script in the construction of the database

As explained in the explication of the documentation generation pages, the documentation generation script is used to generate the gaml references and the model library pages (in the markdown format with metadatas), but also to build two files **category.txt** and **keyword.xml**.

The file **category.txt** is a very simple file, listing the different keyword categories. This file will be used to build the **Category** table.

Format of the file:

```
concept, type, statement, species, architecture, operator, skill,
   constant, action, attribute, facet
```

The file **keyword.xml** is an xml file that contains all the possible keywords (all except some keywords written manually directly in the documentation pages). The GAML words can be found directly using the code of GAMA. The concept words can be found using the code of GAMA (thanks to the tag "category") and also by using the tags in the header of the model files. This xml file will be used to build the **Keyword** and the **AssociationKeywordCategory** tables.

Format of the file:

```
<keyword id:keywordname_keywordcategory>
  <name>keywordname</name>
  <category>keywordcategory</category>
  <associatedKeywordList>
    <associatedKeyword>keywordId1</associatedKeyword>
    <associatedKeyword>keywordId2</associatedKeyword>
  </associatedKeywordList>
</keyword>
```

Note that:

- The list associatedKeywordList contains only one element for the *facet* keywords, one of no element for the *action* or *attribute* keywords (none when the

action/attribute is a built-in), several or no elements for the *concept* keywords, and none for the other type of keywords.

- The id is built with the value of the attribute "name" and with the value of the attribute "category" for every keywords except the *statement*, *facet*, *action* and *attribute* keywords, which need also the value of the associatedKeyword linked. Ex : the id of the facet "torus" will be "facet_torus_statement_global".

Preparation of the repository before the generation of the database

After the generation of the markdown content in the wiki repository, two other files have to be built manually: the files **alias.txt** and **learningConcept.xml**.

The **alias.txt** file contains all the connexions between alias and keyword name. It will be used to build the **Alias** table.

Format of the file :

```
aliasName1:remplacedWord1
aliasName2:remplacedWord2
kill:die
```

The **learningConcept.xml** file is used to list the learning concepts, and to connect them to their prerequisite concepts. It will be used to build the **LearningConcept** and the **AssociationLearningConcept** tables.

Format of the file :

```
<learningConcept id:learningConceptName>
  <name>learningConceptName</name>
  <prerequisiteList>
    <prerequisite>learningConcept1</prerequisite>
    <prerequisite>learningConcept2...</prerequisite>
  </prerequisiteList>
</learningConcept>
```

Note that the value of the attribute "name" can be used as an unique id.

Role of the website content generation script in the construction of the database

After copy-paste the content to the website folder, a script is used to build the database and to generate website content.

The **Category**, **Alias**, **LearningConcept** and **AssociationLearningConcept** tables are loaded easily from the files **category.txt**, **alias.txt**, and **learningConcept.xml**.

The **Keyword** and **AssociationKeywordCategory** tables are loaded from the **keyword.xml** file. Note that those two tables are not entirely loaded yet, because other keywords can be presents in the header of other files.

The markdown files are converted one by one into html format.

- When a metadata **startConcept/endConcept** is found (syntax : //: # (beginAnchor|name_of_learning_concept)), the metadata is replaced with an anchor in the page (with an unique id), and the **AssociationWebpageConcept** table is updated.
- When a metadata **keyword** is found (syntax : //: # (keyword|name_of_keyword_category_name_of_keyword)), the metadata is replaced with an anchor in the page (with an unique id), and the **AssociationWebpageKeyword** table is updated (the **Keyword** and **AssociationKeywordCategory** are updated if the keyword does not exist yet in the table).

Manage concepts keywords

ALL the concepts must be declared in the "IConcept" java class. If you want to add a new concept, please check before if your the concept you want to add cannot be remplaced by one of the existing concept. If it is the case, you can add your word as if it was an **alias**, pointing to the existing concept. Note that all the alias are listed in the alias.txt file. If you realy think that the concept has to be added, please add it to the IConcept file, and also to (one or several) of the lists *CONCEPTS_NOT_FOR_GAML_REF*, *CONCEPTS_NOT_FOR_MODEL_LIBRARY*, *CONCEPTS_DEDICATED_TO_SYNTAX* in the ConceptManager class if needed.

Most of the keywords of the website (used for the search) are managed automatically. But the "concepts" keywords have to be (partially) hand-made managed. You can add the concepts with 3 differents methods :

In the Documentation

"Documentation" here designs all the content manually written in the wiki. All those pages can contain "concepts" through the metadata format :

```
[//]: # (keyword|concept_name_of_concept)
```

You can either : - place those metadatas anywhere in the page if you want to point directly in this part of the page if the user makes a search - place those metadatas **above the title** : doing this, the "automatic search" (left navigation panel) will be made with this concept. You have to be really sure this concept is the **main** concept of the page to place it there.

In the Model library

Directly from the gaml files of the model library, you can add the concept you want through the "Tags" in the header of the model.

Exemple :

```
/**
* Name: 3D Display model of differents shapes and a special Object
* Author:
* Description: Model presenting a 3D display of differents shapes (
   pyramid, cone, cylinder, sphere and a teapot object) to represent
   the same agents but with
*       different aspects. Five experiments are possible, one for each
   of the shapes presented previously. In each experiment, the agents
   move to create a big circle but flee
*       from their closest neighbor.
* Tags: 3d, shape, neighbors
*/
```

Note that if you don't want this model to have a page in the website, you can name it starting with the character _.

In the GAML References

You can add a concept to a gaml word by using the syntax `concept = { IConcept.MY_CONCEPT }`.

Exemple :

```
@operator(value = "flip",
    concept = { IConcept.RANDOM }
```

The following text has been automatically generated from "mainCheckConcepts"

last update : 2019/08/12 15:37:28

List of concepts to use for model library (except Syntax):

3d, agent_location, agent_movement, algorithm, architecture, asc, batch, bdi, camera, chart, clustering, color, communication, comodel, comparison, csv, database, date, dem, dgs, diffusion, dxf, edge, elevation, equation, fipa, fsm, geometry, gis, gml, graph, graph_weight, graphic, grid, gui, headless, hydrology, image, inheritance, inspector, light, load_file, math, mirror, monitor, multi_criteria, multi_level, multi_simulation, neighbors, network, nil, node, obj, obstacle, osm, overlay, physics_engine, r, raster, regression, save_file, scheduler, serialize, shape, shapefile, shortest_path, skill, sound, spatial_computation, spatial_relation, spatial_transformation, sport, statistic, svg, system, task_based, test, text, texture, tif, topology, transport, txt, xml

List of concepts to use exclusively in Syntax models:

arithmetic, attribute, cast, condition, container, filter, list, logical, loop, map, matrix, string, ternary

List of concepts to use for GAML worlds:

3d, action, agent_location, agent_movement, algorithm, architecture, arithmetic, asc, attribute, batch, bdi, behavior, camera, cast, chart, clustering, color, communication, comodel, comparison, condition, constant, container, csv, cycle, database, date, dem, dgs, diffusion, dimension, display, dxf, edge, elevation, equation, experiment, file, filter, fipa, fsm, geometry, gis, gml, graph, graph_weight, graphic, graphic_unit, grid, gui, headless, hydrology, image, inspector, length_unit, light, list, load_file, logical, loop, map, math, matrix, mirror, monitor, multi_criteria, multi_level, multi_simulation, neighbors, network, nil, node, obj, obstacle, optimization, osm, overlay, parameter, physics_engine, point, r, random, random_operator, raster, regression, save_file, scheduler, serialize, shape, shapefile, shortest_path, skill, sound, spatial_computation, spatial_relation, spatial_transformation, species, statistic, string, surface_unit, svg, system, task_based, ternary, test, text, texture, tif, time, time_unit, topology, transport, txt, type, volume_unit, weight_unit, xml

Concept name	in Doc	in GAML Ref	in Model Lib	TOTAL
3d	2	16	39	57
action	3	8	_	12
agent_location	1	21	0	22
agent_movement	0	1	23	24
algorithm	1	8	1	10
architecture	2	9	1	12
arithmetic	0	33	0	33
asc	0	2	2	4
attribute	2	6	_	9
autosave	2	_	_	2
background	1	_	_	1
batch	2	7	3	12
bdi	1	0	0	1
behavior	3	7	_	10
camera	1	1	2	4
cast	1	17	1	19
chart	0	5	5	10
clustering	0	5	3	8
color	1	17	4	22
communication	0	1	0	1
comodel	0	0	8	8
comparison	0	6	2	8
condition	1	7	0	8
constant	0	13	_	13
container	1	61	1	63
csv	2	4	3	9
cycle	2	2	_	4
database	1	0	17	18
date	1	48	2	51
dem	0	0	1	1
dgs	0	0	1	1
diffusion	1	0	14	15
dimension	2	35	_	37
display	2	25	_	29
distribution	1	_	_	1
dxf	0	2	1	3

Concept name	in Doc	in GAML Ref	in Model Lib	TOTAL
edge	1	16	1	18
elevation	0	0	7	7
enumeration	1	_	_	1
equation	2	0	15	17
experiment	2	2	_	4
facet	1	_	_	1
file	1	42	_	45
filter	1	12	0	13
fipa	1	1	11	13
fsm	1	0	0	1
geometry	2	98	0	100
gis	0	5	24	29
global	1	_	_	1
gml	0	2	0	2
graph	1	53	32	86
graph_weight	1	3	0	4
graphic	1	10	1	12
graphic_unit	1	6	_	7
grid	4	9	33	46
gui	3	9	24	36
halt	1	_	_	1
headless	0	0	0	0
hydrology	0	0	1	1
image	0	4	0	4
import	1	_	_	1
inheritance	1	_	1	2
init	3	_	_	3
inspector	1	2	1	4
layer	1	_	_	1
length_unit	0	9	_	9
light	2	1	4	7
list	0	9	2	11
load_file	4	1	18	23
logical	1	8	0	9
loop	1	2	2	5
map	0	16	1	17

Concept name	in Doc	in GAML Ref	in Model Lib	TOTAL
math	1	35	20	56
matrix	2	22	8	32
mirror	1	0	2	3
model	1	_	_	1
monitor	1	1	1	3
multi_criteria	0	5	1	6
multi_level	1	3	9	13
multi_simulation	1	0	2	3
neighbors	1	5	1	7
network	0	0	0	0
nil	1	0	0	1
node	1	21	1	23
obj	0	0	2	2
obstacle	0	1	3	4
opengl	3	_	_	3
operator	1	_	_	1
optimization	3	1	_	4
osm	1	2	2	5
output	1	_	_	6
overlay	0	0	1	1
parameter	2	1	_	3
pause	1	_	_	1
permanent	0	_	_	0
physics_engine	0	0	4	4
point	0	29	_	29
probability	1	_	_	1
pseudo_variable	1	_	_	1
r	1	2	0	3
random	3	10	_	13
random_operator	0	0	_	0
raster	0	0	4	4
reflex	1	_	_	1
refresh	2	_	_	2
regression	0	2	1	3
save_file	0	2	7	9
scheduler	2	2	0	4

Concept name	in Doc	in GAML Ref	in Model Lib	TOTAL
serialize	0	0	0	0
shape	3	21	1	25
shapefile	2	2	34	38
shortest_path	1	6	4	11
skill	2	5	42	49
sound	0	0	0	0
spatial_computation	0	75	7	82
spatial_relation	0	32	0	32
spatial_transformation	0	26	2	28
species	1	20	_	21
sport	0	_	2	2
statistic	0	31	5	36
string	0	27	0	27
surface_unit	0	4	_	4
svg	0	2	0	2
system	1	11	0	12
task_based	1	4	0	5
ternary	1	2	1	4
test	0	6	2	8
text	2	10	0	12
texture	0	1	3	4
tif	0	2	1	3
time	2	20	_	22
time_unit	0	13	_	13
topology	3	9	4	16
torus	1	_	_	1
transport	1	0	6	7
txt	0	0	1	1
type	0	44	_	44
update	1	_	_	1
volume_unit	0	5	_	5
weight_unit	0	8	_	8
world	1	_	_	1
write	1	_	_	1
xml	0	4	0	4

Part XIII

Scientific References

Chapter 171

References

This page contains a subset of the scientific papers that have been written either about GAMA or using the platform as an experimental/modeling support.

If you happen to publish a paper that uses or discusses GAMA, please let us know, so that we can include it in this list.

As stated in the first page, if you need to cite GAMA in a paper, we kindly ask you to use this reference:

- Taillandier, P., Gaudou, P. Grignard, A. Huynh, Q.N., Marilleau, N., Caillou, P., Philippon, D., Drogoul, A. (2018) Building, Composing and Experimenting Complex Spatial Models with the GAMA Platform. GeoInformatica, Dec. 2018. https://doi.org/10.1007/s10707-018-00339-6.

Papers about GAMA

- Taillandier, P., Grignard, A., Marilleau, N., Philippon, D., Huynh Q.N., Gaudou, B., Drogoul, A. (2019) "Participatory Modeling and Simulation with the GAMA Platform". Journal of Artificial Societies and Social Simulation 22 (2), 1-3. DOI: 10.18564/jasss.3964
- Caillou, P., Gaudou, B., Grignard, A., Truong, C.Q., Taillandier, P. (2017) A Simple-to-Use BDI Architecture for Agent-Based Modeling and Simulation, in: Advances in Social Simulation 2015. Springer, Cham, pp. 15–28. doi:10.1007/978-3-319-47253-9_2

- Chapuis, K., Taillandier, P., Renaud, M., Drogoul, A. (2018) "Gen*: a generic toolkit to generate spatially explicit synthetic populations". International Journal of Geographical Information Science 32 (6), 1194-1210
- Taillandier, Patrick, Arnaud Grignard, Benoit Gaudou, and Alexis Drogoul. "Des données géographiques à la simulation à base d'agents: application de la plate-forme GAMA." Cybergeo: European Journal of Geography (2014).
- Grignard, A., Taillandier, P., Gaudou, B., Vo, D-A., Huynh, Q.N., Drogoul, A. (2013) GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.
- Grignard, A., Drogoul, A., Zucker, J.D. (2013) Online analysis and visualization of agent based models, Computational Science and Its Applications–ICCSA 2013. Springer Berlin Heidelberg, 2013. 662-672.
- Taillandier, P., Drogoul, A., Vo, D.A. and Amouroux, E. (2012) GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control, in 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp 242-258.
- Taillandier, P., Drogoul, A. (2011) From Grid Environment to Geographic Vector Agents, Modeling with the GAMA simulation platform. In '25th Conference of the International Cartographic Association', Paris, France.
- Taillandier, P., Drogoul A., Vo D.A., Amouroux, E. (2010) GAMA : bringing GIS and multi-level capabilities to multi-agent simulation, in 'the 8th European Workshop on Multi-Agent Systems', Paris, France.
- Amouroux, E., Taillandier, P. & Drogoul, A. (2010) Complex environment representation in epidemiology ABM: application on `H5N1` propagation. In 'the 3rd International Conference on Theories and Applications of Computer Science' (ICTACS'10).
- Amouroux, E., Chu, T.Q., Boucher, A. and Drogoul, A. (2007) GAMA: an environment for implementing and running spatially explicit multi-agent simulations. In 'Pacific Rim International Workshop on Multi-Agents', Bangkok, Thailand, pp. 359–371.

HDR theses

- **Patrick Taillandier**, "Vers une meilleure intégration des dimensions spatiales, comportementales et participatives en simulation à base d'agents", University

Toulouse 1 Capitole, France 2019.
- **Benoit Gaudou**, "Toward complex models of complex systems - One step further in the art of Agent-Based Modelling", University Toulouse 1 Capitole, France 2016.
- **Nicolas Marilleau**, "Distributed Approaches based on Agent Based Systems to model and simulate complex systems with a space", Pierre and Marie Curie University, Paris, France 2016.

PhD theses

- **Mathieu Bourgais**, "Vers des agents cognitifs, affectifs et sociaux dans la simulation", Normandie Université, defended November 30th, 2018.
- **Huynh Quang Nghi**, "CoModels, engineering dynamic compositions of coupled models to support the simulation of complex systems", University of Paris 6, defended December 5th, 2016.
- **Truong Chi Quang**, "Integrating cognitive models of human decision-makingin agent-based models : an application to land useplanning under climate change in the Mekong river delta", University of Paris 6 & Can Tho University, defended December 7th, 2016.
- **Arnaud Grignard**, "Modèles de visualisation à base d'agents", University of Paris 6, defended October 2nd, 2015.
- **Truong Minh Thai**, "To Develop a Database Management Tool for Multi-Agent Simulation Platform", 'Université Toulouse 1 Capitole, defended February 11th, 2015.
- **Truong Xuan Viet**, "Optimization by Simulation of an Environmental Surveillance Network: Application to the Fight against Rice Pests in the Mekong Delta (Vietnam)", University of Paris 6 & Ho Chi Minh University of Technology, defended June 24th, 2014.
- **Nguyen Nhi Gia Vinh**, "Designing multi-scale models to support environmental decision: application to the control of Brown Plant Hopper invasions in the Mekong Delta (Vietnam)", University of Paris 6, defended Oct. 31st, 2013.
- **Vo Duc An**, "An operational architecture to handle multiple levels of representation in agent-based models", University of Paris 6, defended Nov. 30th 2012.
- **Amouroux Edouard**, "KIMONO: a descriptive agent-based modeling methodology for the exploration of complex systems: an application to epidemiology", University of Paris 6, defended Sept. 30th, 2011.

- **Chu Thanh Quang**, "Using agent-based models and machine learning to enhance spatial decision support systems: Application to resource allocation in situations of urban catastrophes", University of Paris 6, defended July 1st, 2011.
- **Nguyen Ngoc Doanh**, "Coupling Equation-Based and Individual-Based Models in the Study of Complex Systems: A Case Study in Theoretical Population Ecology", University of Paris 6, defended Dec. 14th, 2010.

Research papers that use GAMA as modeling/simulation support

2019

- Adam, C., Taillandier, F.. Games ready to use: A serious game for teaching natural risk management. Simulation and Gaming, SAGE Publications, 2018.
- Mancheva, L., Adam, C., & Dugdale, J., 2019. Multi-agent geospatial simulation of human interactions and behaviour in bushfires. In International Conference on Information Systems for Crisis Response and Management. In ISCRAM 2019 conference, Valencia, Spain.
- Daudé, E., Chapuis, K., Taillandier, P., Tranouez, P., Caron, C., Drogoul, A., Gaudou, B., Rey-Coyrehourq, S., Saval, A., Zucker, J. D., 2019. ESCAPE: Exploring by Simulation Cities Awareness on Population Evacuation. In ISCRAM 2019 conference, Valencia, Spain.
- Farias, G. P., Leitzke, B. S., Born, M. B., de Aguiar, M. S., Adamatti, D. F., 2019. Modelagem Baseada em Agentes para Analise de Recursos Hidricos. In the Workshop-School on Agents, Environments, and Applications (WESAAC), Florianopolis – Santa Catarina (Brazil).
- Marrocco, L., Ferrer, E. C., Bucchiarone, A., Grignard, A., Alonso, L., Larson, K., 2019. BASIC: Towards a Blockchained Agent-Based SImulator for Cities. In International Workshop on Massively Multiagent Systems (pp. 144-162). Springer, Cham.
- Ruiz-Chavez, Z., Salvador-Meneses, J., Mejía-Astudillo, C., Diaz-Quilachamin, S., 2019. Analysis of Dogs's Abandonment Problem Using Georeferenced Multi-agent Systems. International Work-Conference on the Interplay Between Natural and Artificial Computation (pp. 297-306). Springer, Cham. https://doi.org/10.1007/978-3-030-19651-6_29

- Rodrique, K., Tuong, H., Manh, N., 2019. An Agent-based Simulation for Studying Air Pollution from Traffic in Urban Areas: The Case of Hanoi City. Int. J. Adv. Comput. Sci. Appl. 10. https://doi.org/10.14569/IJACSA.2019.0100376
- Micolier, A., Taillandier, F., Taillandier, P., Bos, F., 2019. Li-BIM, an agent-based approach to simulate occupant-building interaction from the Building-Information Modelling. Eng. Appl. Artif. Intell. 82, 44–59. https://doi.org/10.1016/j.engappai.2019.03.008
- Houssou, N.L.J., Cordero, J.D., Bouadjio-Boulic, A., Morin, L., Maestripieri, N., Ferrant, S., Belem, M., Pelaez Sanchez, J.I., Saenz, M., Lerigoleur, E., Elger, A., Gaudou, B., Maurice, L., Saqalli, M., 2019. Synchronizing Histories of Exposure and Demography: The Construction of an Agent-Based Model of the Ecuadorian Amazon Colonization and Exposure to Oil Pollution Hazards. J. Artif. Soc. Soc. Simul. 22, 1. https://doi.org/10.18564/jasss.3957
- Knapps, V., Zimmermann, K.-H., 2019. Distributed Monitoring of Topological Events via Homology. ArXiv190104146 Cs Math.
- Galimberti, A., Alyokhin, A., Qu, H., Rose, J., 2019. Simulation modelling of Potato virus Y spread in relation to initial inoculum and vector activity. Journal of Integrative Agriculture

2018

- Marilleau, N., Lang, C., Giraudoux, P., 2018. Coupling agent-based with equation-based models to study spatially explicit megapopulation dynamics. Ecol. Model. 384, 34–42. https://doi.org/10.1016/j.ecolmodel.2018.06.011
- Alfeo, A.L., Ferrer, E.C., Carrillo, Y.L., Grignard, A., Pastor, L.A., Sleeper, D.T., Cimino, M.G.C.A., Lepri, B., Vaglini, G., Larson, K., Dorigo, M., Pentland, A. 'Sandy', 2018. Urban Swarms: A new approach for autonomous waste management. ArXiv181007910 Cs.
- Qu, H., Drummond, F., 2018. Simulation-based modeling of wild blueberry pollination. Comput. Electron. Agric. 144, 94–101. https://doi.org/10.1016/j.compag.2017.11.003
- Shaham, Y., Benenson, I., 2018. Modeling fire spread in cities with non-flammable construction. Int. J. Disaster Risk Reduct. 31, 1337–1353. https://doi.org/10.1016/j.ijdrr.2018.03.010
- Mewes, B., Schumann, A.H., 2018. IPA (v1): a framework for agent-based modelling of soil water movement. Geosci. Model Dev. 11, 2175–2187. https://doi.org/10.5194/gmd-11-2175-2018

- Grignard, A., Macià, N., Alonso Pastor, L., Noyman, A., Zhang, Y., Larson, K., 2018. CityScope Andorra: A Multi-level Interactive and Tangible Agent-based Visualization, in: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '18. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1939–1940.
- Zhang, Y., Grignard, A., Lyons, K., Aubuchon, A., Larson, K., 2018. Real-time Machine Learning Prediction of an Agent-Based Model for Urban Decision-making (Extended Abstract) 3.
- Bandyopadhyay, M., Singh, V., 2018. Agent-based geosimulation for assessment of urban emergency response plans. Arab. J. Geosci. 11, 165. https://doi.org/10.1007/s12517-018-3523-5
- Samad, T., Iqbal, S., Malik, A.W., Arif, O., Bloodsworth, P., 2018. A multi-agent framework for cloud-based management of collaborative robots. Int. J. Adv. Robot. Syst. 15, 172988141878507. https://doi.org/10.1177/1729881418785073
- Humann, J., Spero, E., 2018. Modeling and simulation of multi-UAV, multi-operator surveillance systems, in: 2018 Annual IEEE International Systems Conference (SysCon). Presented at the 2018 Annual IEEE International Systems Conference (SysCon), pp. 1–8. https://doi.org/10.1109/SYSCON.2018.8369546
- Mazzoli, M., Re, T., Bertilone, R., Maggiora, M., Pellegrino, J., 2018. Agent Based Rumor Spreading in a scale-free network. ArXiv180505999 Cs.
- Grignard, A., Alonso, L., Taillandier, P., Gaudou, B., Nguyen-Huu, T., Gruel, W., Larson, K., 2018a. The Impact of New Mobility Modes on a City: A Generic Approach Using ABM, in: Morales, A.J., Gershenson, C., Braha, D., Minai, A.A., Bar-Yam, Y. (Eds.), Unifying Themes in Complex Systems IX, Springer Proceedings in Complexity. Springer International Publishing, pp. 272–280.
- Touhbi, S., Babram, M.A., Nguyen-Huu, T., Marilleau, N., Hbid, M.L., Cambier, C., Stinckwich, S., 2018. Time Headway analysis on urban roads of the city of Marrakesh. Procedia Comput. Sci. 130, 111–118. https://doi.org/10.1016/j.procs.2018.04.019
- Laatabi, A., Marilleau, N., Nguyen-Huu, T., Hbid, H., Ait Babram, M., 2018. ODD+2D: An ODD Based Protocol for Mapping Data to Empirical ABMs. J. Artif. Soc. Soc. Simul. 21, 9. https://doi.org/10.18564/jasss.3646
- Chapuis K., Taillandier P., Gaudou B., Drogoul A., Daudé E. (2018) A Multi-modal Urban Traffic Agent-Based Framework to Study Individual Response to Catastrophic Events. In: Miller T., Oren N., Sakurai Y., Noda I., Savarimuthu B., Cao Son T. (eds) PRIMA 2018: Principles and Practice of Multi-Agent

Systems. PRIMA 2018. Lecture Notes in Computer Science, vol 11224. Springer, Cham

- Bourgais, M., Taillandier, P., Vercouter, L., Adam, C., 2018. Emotion Modeling in Social Simulation: A Survey. J. Artif. Soc. Soc. Simul. 21, 5.
- Grillot, M., Guerrin, F., Gaudou, B., Masse, D., Vayssières, J., 2018. Multi-level analysis of nutrient cycling within agro-sylvo-pastoral landscapes in West Africa using an agent-based model. Environ. Model. Softw. 107, 267–280. https://doi.org/10.1016/j.envsoft.2018.05.003
- Valette, M., Gaudou, B., Longin, D., Taillandier, P., 2018. Modeling a Real-Case Situation of Egress Using BDI Agents with Emotions and Social Skills, in: Miller, T., Oren, N., Sakurai, Y., Noda, I., Savarimuthu, B.T.R., Cao Son, T. (Eds.), PRIMA 2018: Principles and Practice of Multi-Agent Systems. Springer International Publishing, Cham, pp. 3–18. https://doi.org/10.1007/978-3-030-03098-8_1
- Humann, J., Spero, E. (2018) Modeling and Simulation of multi-UAV, multi-Operator Surveillance Systems, 2018 Annual IEEE International Systems Conference (SysCon), Vancouver, BC.
- Lammoglia, A., Leturcq, S., Delay, E., 2018. Le modèle VitiTerroir pour simuler la dynamique spatiale des vignobles sur le temps long (1836-2014). Exemple d'application au département d'Indre-et-Loire. Cybergeo Eur. J. Geogr. https://doi.org/10.4000/cybergeo.29324
- Amores, D., Vasardani, M., Tanin, E., 2018. Early Detection of Herding Behaviour during Emergency Evacuations 15 pages. https://doi.org/10.4230/lipics.giscience.2018.1
- Rakotoarisoa, M.M., Fleurant, C., Taibi, A.N., Rouan, M., Caillault, S., Razakamanana, T., Ballouche, A., 2018. Un modèle multi-agents pour évaluer la vulnérabilité aux inondations : le cas des villages aux alentours du Fleuve Fiherenana (Madagascar). Cybergeo Eur. J. Geogr. https://doi.org/10.4000/cybergeo.29144

2017

- Cura, R., Tannier, C., Leturcq, S., Zadora-Rio, E., Lorans, E., & Rodier, X. (2017). Transition 8: 800-1100. Fixation, polarisation et hiérarchisation de l'habitat rural en Europe du Nord-Ouest (chap. 11). (https://simfeodal.github.io/)
- Becu, N., Amalric, M., Anselme, B., Beck, E., Bertin, X., Delay, E., Long,

N., Marilleau, N., Pignon-Mussaud, C., Rousseaux, F., 2017. Participatory simulation to foster social learning on coastal flooding prevention. Environ. Model. Softw. 98, 1–11. https://doi.org/10.1016/j.envsoft.2017.09.003

- Adam, C., Gaudou, B., 2017. Modelling Human Behaviours in Disasters from Interviews: Application to Melbourne Bushfires. J. Artif. Soc. Soc. Simul. 20, 12. https://doi.org/10.18564/jasss.3395
- Adam, C., Taillandier, P., Dugdale, J., Gaudou, B., 2017. BDI vs FSM Agents in Social Simulations for Raising Awareness in Disasters: A Case Study in Melbourne Bushfires. Int. J. Inf. Syst. Crisis Response Manag. 9, 27–44. https://doi.org/10.4018/IJISCRAM.2017010103
- Amalric, M., Anselme, B., Bécu, N., Delay, E., Marilleau, N., Pignon, C., Rousseaux, F., 2017. Sensibiliser au risque de submersion marine par le jeu ou faut-il qu'un jeu soit spatialement réaliste pour être efficace ? Sci. Jeu. https://doi.org/10.4000/sdj.859
- Emery, J., Marilleau, N., Martiny, N., Thévenin, T., Badram, M.A., Grignard, A., Hbdid, H., 2017. MARRAKAIR : UNE SIMULATION PARTICIPATIVE POUR OBSERVER LES ÉMISSIONS ATMOSPHÉRIQUES DU TRA- FIC ROUTIER EN MILIEU URBAIN 5.
- Martiny, N., Emery, J., Ceamanos, X., Briottet, X., Marilleau, N., Thevenin, T., Léon, J.-F., 2017. La Qualité de l'air en ville à Très haute Résolution (Quali_ThR): Apport des images Pléiades dans la démarche SCAUP ?, in: FUTURMOB : Préparer La Transition Vers La Mobilité Autonome. Montbéliard, France.
- Ta, X.-H., Gaudou, B., Longin, D., Ho, T.V., 2017. Emotional contagion model for group evacuation simulation. Informatica 41.
- Huynh, N.Q., Nguyen-Huu, T., Grignard, A., Huynh, H.X., Drogoul, A., 2017. Coupling equation based models and agent-based models: example of a multi-strains and switch SIR toy model. EAI Endorsed Trans. Context-Aware Syst. Appl. 4, 152334. https://doi.org/10.4108/eai.6-3-2017.152334
- Taillandier, P., Bourgais, M., Drogoul, A., Vercouter, L. Using parallel computing to improve the scalability of models with BDI agents. Social Simulation Conference, Sep 2017, Dublin, Ireland.
- Philippon, D., Choisy, M., Drogoul, A., Gaudou, B., Marilleau, N., Taillandier, P., Truong, Q.C. (2017) Exploring Trade and Health Policies Influence on Dengue Spread with an Agent-Based Model, in: Nardin, L.G., Antunes, L. (Eds.), Multi-Agent Based Simulation XVII. Springer International Publishing, Cham, pp. 111–127.doi:10.1007/978-3-319-67477-3_6
- Marilleau, N., Giraudoux, P., Lang, C., 2017. Multi-agent simulation as a tool

to study risk in a spatial context, in: International Forum on Disaster Risk Management. Kunming, China.

2016

- Fosset, P., Banos, A., Beck, E., Chardonnel, S., Lang, C., Marilleau, N., Piombini, A., Leysens, T., Conesa, A., Andre-Poyaud, I., Thevenin, T., 2016. Exploring Intra-Urban Accessibility and Impacts of Pollution Policies with an Agent-Based Simulation Platform: GaMiroD. Systems 4, 5. https://doi.org/10.3390/systems4010005
- Grignard, A., Fantino, G., Lauer, J.W., Verpeaux, A., Drogoul, A., 2016. Agent-Based Visualization: A Simulation Tool for the Analysis of River Morphosedimentary Adjustments, in: Gaudou, B., Sichman, J.S. (Eds.), Multi-Agent Based Simulation XVI. Springer International Publishing, Cham, pp. 109–120. https://doi.org/10.1007/978-3-319-31447-1_7
- Lucien, L., Lang, C., Marilleau, N., Philippe, L., 2016. Multiagent Hybrid Architecture for Collaborative Exchanges between Communicating Vehicles in an Urban Context. Procedia Comput. Sci. 83, 695–699. https://doi.org/10.1016/j.procs.2016.04.154
- Laatabi, A., Marilleau, N., Nguyen-Huu, T., Hbid, H., Babram, M.A., 2016. Formalizing Data to Agent Model Mapping Using MOF: Application to a Model of Residential Mobility in Marrakesh, in: Jezic, G., Chen-Burger, Y.-H.J., Howlett, R.J., Jain, L.C. (Eds.), Agent and Multi-Agent Systems: Technology and Applications. Springer International Publishing, Cham, pp. 107–117. https://doi.org/10.1007/978-3-319-39883-9_9
- Taillandier, P., Banos, A., Drogoul, A., Gaudou, B., Marilleau, N., Truong, Q.C. (2016) Simulating Urban Growth with Raster and Vector models: A case study for the city of Can Tho, Vietnam, in: Osman, N., Sierra, C. (Eds.), Autonomous Agents and Multiagent Systems, Lecture Notes in Computer Science. Springer International Publishing, pp. 154–171. Doi: 10.1007/978-3-319-46840-2_10.
- Nghi, H.Q, Nguyen-Huu, T., Grignard, A., Huynh, X.H., Drogoul, A. (2016) Toward an Agent-Based and Equation-Based Coupling Framework. International Conference on Nature of Computation and Communication, 311-324
- Bhamidipati, S., van der Lei, T., & Herder, P. (2016) A layered approach to model interconnected infrastructure and its significance for asset management. EJTIR, 16(1), 254-272.
- Drogoul A., Huynh N.Q. and Truong Q.C. (2016) Coupling environmental,

social and economic models to understand land-use change dynamics in the Mekong Delta. Front. Environ. Sci. 4:19. doi:10.3389/fenvs.2016.00019.

- Grignard, A., Fantino, G., Lauer, J.W., Verpeaux, A., Drogoul, A., 2016. Agent-Based Visualization: A Simulation Tool for the Analysis of River Morphosedimentary Adjustments, in: Gaudou, B., Sichman, J.S. (Eds.), Multi-Agent Based Simulation XVI. Springer International Publishing, Cham, pp. 109–120. https://doi.org/10.1007/978-3-319-31447-1_7
- Truong, Q.C., Taillandier, P., Gaudou, B., Vo, M.Q., Nguyen, T.H., Drogoul, A. (2016) Exploring Agent Architectures for Farmer Behavior in Land-Use Change. A Case Study in Coastal Area of the Vietnamese Mekong Delta, in: Gaudou, B., Sichman, J.S. (Eds.), Multi-Agent Based Simulation XVI, Lecture Notes in Computer Science. Springer International Publishing, pp. 146–158. doi: 10.1007/978-3-319-31447-1_10.
- Lang, C., Marilleau, N., Giraudoux, P., 2016. Couplage de SMA avec des EDO pour simuler les phénomènes écologiques à grande échelle, in: 2ème Rencontre "Informatique Scientifique à Besançon". Besançon, France.
- Giraudoux, P., Lang, C., Marilleau, N., 2016. Coupling agent based with equation based models for studying explicitly spatial population dynamics.
- Lucien, L., Lang, C., Marilleau, N., Philippe, L., 2016. A Proposition of Data Organization and Exchanges to Collaborate in an Autonomous Agent Context, in: 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES). Presented at the 2016 19th IEEE Intl Conference on Computational Science and Engineering (CSE), IEEE 14th Intl Conference on Embedded and Ubiquitous Computing (EUC), and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), IEEE, Paris, pp. 561–568. https://doi.org/10.1109/CSE-EUC-DCABES.2016.242

2015

- Gasmi, N., Grignard, A., Drogoul, A., Gaudou, B., Taillandier, P., Tessier, O., An, V.D., 2015. Reproducing and Exploring Past Events Using Agent-Based Geo-Historical Models, in: Grimaldo, F., Norling, E. (Eds.), Multi-Agent-Based Simulation XV. Springer International Publishing, Cham, pp. 151–163.
- Le, V.-M., Chevaleyre, Y., Ho Tuong Vinh, Zucker, J.-D., 2015. Hybrid of

linear programming and genetic algorithm for optimizing agent-based simulation. Application to optimization of sign placement for tsunami evacuation, in: The 2015 IEEE RIVF International Conference on Computing & Communication Technologies - Research, Innovation, and Vision for Future (RIVF). Presented at the 2015 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), IEEE, Can Tho, Vietnam, pp. 138–143. https://doi.org/10.1109/RIVF.2015.7049889

- Emery, J., Marilleau, N., Martiny, N., Thévenin, T., Villery, J., 2015. L'apport de la simulation multi-agent du trafic routier pour l'estimation des pollutions atmosphériques automobiles, in: Douzièmes Rencontres de Théo Quant. Besançon, France.

2014

- Macatulad, E. G., Blanco, A. C. (2014) 3DGIS-BASED MULTI-AGENT GEOSIMULATION AND VISUALIZATION OF BUILDING EVACUATION USING GAMA PLATFORM. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-2, 2014. ISPRS Technical Commission II Symposium, 6 – 8 October 2014, Toronto, Canada.
- Bhamidipati, S. (2014) A simulation framework for asset management in climate-change adaptation of transportation infrastructure. In: Proceedings of 42nd European Transport Conference. Frankfurt, Germany.
- Gaudou, B., Sibertin-Blanc, C., Thérond, O., Amblard, F., Auda, Y., Arcangeli, J.-P., Balestrat, M., Charron-Moirez, M.-H., Gondet, E., Hong, Y., Lardy, R., Louail, T., Mayor, E., Panzoli, D., Sauvage, S., Sanchez-Perez, J., Taillandier, P., Nguyen, V. B., Vavasseur, M., Mazzega, P. (2014). The MAELIA multi-agent platform for integrated assessment of low-water management issues. In: International Workshop on Multi-Agent-Based Simulation (MABS 2013), Saint-Paul, MN, USA, 06/05/2013-07/05/2013, Vol. 8235, Shah Jamal Alam, H. Van Dyke Parunak, (Eds.), Springer, Lecture Notes in Computer Science, p. 85-110.
- Gaudou, B., Lorini, E., Mayor, E. (2014.) Moral Guilt: An Agent-Based Model Analysis. In: Conference of the European Social Simulation Association (ESSA 2013), Warsaw, 16/09/2013-20/09/2013, Vol. 229, Springer, Advances in Intelligent Systems and Computing, p. 95-106.
- Le, V.-M., Chevaleyre, Y., Zucker, J.-D., Tuong Vinh, H., 2014. Approaches to Optimize Local Evacuation Maps for Helping Evacuation in Case of Tsunami,

in: Hanachi, C., Bénaben, F., Charoy, F. (Eds.), Information Systems for Crisis Response and Management in Mediterranean Countries. Springer International Publishing, Cham, pp. 21–31. https://doi.org/10.1007/978-3-319-11818-5_3
- Emery, J., Marilleau, N., Thévenin, T., Martiny, N., 2014. Du comptage ponctuel à l'affectation par simulation multi-agents : application à la circulation routière de la ville de Dijon, in: Conférence Internationale de Géomatique et d'analyse Spatiale (SAGEO). Grenoble, France, p. CD-ROM.

2013

- Drogoul, A., Gaudou, B., Grignard, A., Taillandier, P., & Vo, D. A. (2013). Practical Approach To Agent-Based Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, p. 277-300, Regional Social Sciences Summer University.
- Drogoul, A., Gaudou, B. (2013) Methods for Agent-Based Computer Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, 1.6, p. 130-154, Regional Social Sciences Summer University.
- Truong, M.-T., Amblard, F., Gaudou, B., Sibertin-Blanc, C., Truong, V. X., Drogoul, A., Hyunh, X. H., Le, M. N. (2013). An implementation of framework of business intelligence for agent-based simulation. In: Symposium on Information and Communication Technology (SoICT 2013), Da Nang, Viet Nam, 05/12/2013-06/12/2013, Quyet Thang Huynh, Thanh Binh Nguyen, Van Tien Do, Marc Bui, Hong Son Ngo (Eds.), ACM, p. 35-44.
- Le, V. M., Gaudou, B., Taillandier, P., Vo, D. A (2013). A New BDI Architecture To Formalize Cognitive Agent Behaviors Into Simulations. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems (KES-AMSTA 2013), Hue, Vietnam, 27/05/2013-29/05/2013, Vol. 252, Dariusz Barbucha, Manh Thanh Le, Robert J. Howlett, C. Jain Lakhmi (Eds.), IOS Press, Frontiers in Artificial Intelligence and Applications, p. 395-403.
- Emery, J., Boyard-Micheau, J., Marilleau, N., Martiny, N., Thévenin, T., 2013. Exploitation of traffic counting data for traffic study in urban areas: from traffic assignment to simulation model validation, in: 18th European Colloquium in Theoretical and Quantitative Geography (ECTQG). Dourdan, France.
- Banos, A., Marilleau, N., 2013. Improving Individual Accessibility to the City, in: Gilbert, T., Kirkilionis, M., Nicolis, G. (Eds.), Proceedings of the European Conference on Complex Systems 2012, Springer Proceedings in Complexity.

Springer International Publishing, pp. 989–992.

2012

- Taillandier, P., Therond, O., Gaudou B. (2012), A new BDI agent architecture based on the belief theory. Application to the modelling of cropping plan decision-making. In 'International Environmental Modelling and Software Society', Germany, pp. 107-116.
- NGUYEN, Q.T., BOUJU, A., ESTRAILLIER, P. (2012) Multi-agent architecture with space-time components for the simulation of urban transportation systems.
- Cisse, A., Bah, A., Drogoul, A., Cisse, A.T., Ndione, J.A., Kebe, C.M.F. & Taillandier P. (2012), Un modèle à base d'agents sur la transmission et la diffusion de la fièvre de la Vallée du Rift à Barkédji (Ferlo, Sénégal), Studia Informatica Universalis 10 (1), pp. 77-97.
- Taillandier, P., Amouroux, E., Vo, D.A. and Olteanu-Raimond A.M. (2012), Using Belief Theory to formalize the agent behavior: application to the simulation of avian flu propagation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 575-587.
- Le, V.M., Adam, C., Canal, R., Gaudou, B., Ho, T.V. and Taillandier, P. (2012), Simulation of the emotion dynamics in a group of agents in an evacuation situation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 604-619.
- Nguyen Vu, Q. A., Canal, R., Gaudou, B., Hassas, S., Armetta, F. (2012), TrustSets - Using trust to detect deceitful agents in a distributed information collecting system. In: Journal of Ambient Intelligence and Humanized Computing, Springer-Verlag, Vol. 3 N. 4, p. 251-263.

2011

- Taillandier, P., Therond, O. (2011), Use of the Belief Theory to formalize Agent Decision Making Processes : Application to cropping Plan Decision Making. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 138-142.

- Taillandier, P. & Amblard, F. (2011), Cartography of Multi-Agent Model Parameter Space through a reactive Dicotomous Approach. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 38-42.
- Taillandier, P. & Stinckwich, S. (2011), Using the PROMETHEE Multi-Criteria Decision Making Method to Define New Exploration Strategies for Rescue Robots', IEEE International Symposium on Safety, Security, and Rescue Robotics, Kyoto, Japon, pp. 321 - 326.

2010

- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), A cluster-based approach for disturbed, spatialized, distributed information gathering systems, in 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, pp. 588-603.
- Nguyen, N.D., Taillandier, P., Drogoul, A. and Augier, P. (2010), Inferring Equation-Based Models from Agent-Based Models: A Case Study in Competition Dynamics.In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp. 413-427.
- Amouroux, E., Gaudou, B. Desvaux, S. and Drogoul, A. (2010), O.D.D.: a Promising but Incomplete Formalism For Individual-Based Model Specification. in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), pp. 1-4.
- Nguyen, N.D., Phan, T.H.D., Nguyen, T.N.A., Drogoul, A., Zucker, J-D. (2010), Disk Graph-Based Model for Competition Dynamic, in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen, T.K., Marilleau, N., Ho T.V., El Fallah Seghrouchni, A. (2010), A meta-model for specifying collaborative simulation, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), `TrustSets` - Using trust to detect deceitful agents in a distributed information collecting system, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), the best student paper award.
- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S., Armetta, F. and Stinckwich, S. (2010), Using trust and cluster organisation to improve robot swarm

mapping, Paper to appear in 'Workshop on Robots and Sensors integration in future rescue INformation system' (ROSIN 2010).

2009

- Taillandier, P. and Buard, E. (2009), Designing Agent Behaviour in Agent-Based Simulation through participatory method. In 'The 12th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Nagoya, Japan, pp. 571–578.
- Taillandier, P. and Chu, T.Q. (2009), Using Participatory Paradigm to Learn Human Behaviour. In 'International Conference on Knowledge and Systems Engineering', Ha noi, Viet Nam, pp. 55–60.
- Gaudou, B., Ho, T.V. and Marilleau, N. (2009), Introduce collaboration in methodologies of modeling and simulation of Complex Systems. In 'International Conference on Intelligent Networking and Collaborative Systems (INCOS '09)'. Barcelona, pp. 1–8.
- Nguyen, T.K., Gaudou B., Ho T.V. and Marilleau N. (2009), Application of PAMS Collaboration Platform to Simulation-Based Researches in Soil Science: The Case of the MIcro-ORganism Project. In 'IEEE International Conference on Computing and Telecommunication Technologies (IEEE-RIVF 09)'. Da Nang, Viet Nam, pp. 296–303.
- Nguyen, V.Q., Gaudou B., Canal R., Hassas S. and Armetta F. (2009), Stratégie de communication dans un système de collecte d'information à base d'agents perturbés. In 'Journées Francophones sur les Systèmes Multi-Agents (JFSMA'09)'.

2008

- Chu, T.Q., Boucher, A., Drogoul, A., Vo, D.A., Nguyen, H.P. and Zucker, J.D. (2008). Interactive Learning of Expert Criteria for Rescue Simulations. In Pacific Rim International Workshop on Multi-Agents, Ha Noi, Viet Nam, pp. 127–138.
- Amouroux, E., Desvaux, S. and Drogoul, A. (2008), Towards Virtual Epidemiology: An Agent-Based Approach to the Modeling of H5N1 Propagation and Persistence in North-Vietnam. In Pacific Rim International Workshop on Multi-Agents, Ha Noi, Viet Nam, pp. 26–33.

Part XIV

Projects using GAMA

Chapter 172

Projects

References

This page contains a subset of the scientific papers that have been written either about GAMA or using the platform as an experimental/modeling support.

If you happen to publish a paper that uses or discusses GAMA, please let us know, so that we can include it in this list.

If you need to cite GAMA in a paper, we kindly ask you to use this reference:

- A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

Papers about GAMA

- Taillandier, Patrick, Arnaud Grignard, Benoit Gaudou, and Alexis Drogoul. "Des données géographiques à la simulation à base d'agents: application de la plate-forme GAMA." Cybergeo: European Journal of Geography (2014).
  - A. Grignard, P. Taillandier, B. Gaudou, D-A. Vo, N-Q. Huynh, A. Drogoul (2013), GAMA 1.6: Advancing the Art of Complex Agent-Based Modeling and Simulation. In 'PRIMA 2013: Principles and Practice of Multi-Agent

Systems', Lecture Notes in Computer Science, Vol. 8291, Springer, pp. 117-131.

- Grignard, Arnaud, Alexis Drogoul, and Jean-Daniel Zucker. "Online analysis and visualization of agent-based models." Computational Science and Its Applications–ICCSA 2013. Springer Berlin Heidelberg, 2013. 662-672.
- Taillandier, P., Drogoul, A., Vo, D.A. and Amouroux, E. (2012), GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp 242-258.
- Taillandier, P. & Drogoul, A. (2011), From Grid Environment to Geographic Vector Agents, Modeling with the GAMA simulation platform. In '25th Conference of the International Cartographic Association', Paris, France.
- Taillandier, P. ; Drogoul A. ; Vo D.A. & Amouroux, E. (2010), GAMA : bringing GIS and multi-level capabilities to multi-agent simulation, in 'the 8th European Workshop on Multi-Agent Systems', Paris, France.
- Amouroux, E., Taillandier, P. & Drogoul, A. (2010), Complex environment representation in epidemiology ABM: application on H5N1 propagation. In 'the 3rd International Conference on Theories and Applications of Computer Science' (ICTACS'10).
- Amouroux, E., Chu, T.Q., Boucher, A. and Drogoul, A. (2007), GAMA: an environment for implementing and running spatially explicit multi-agent simulations. In 'Pacific Rim International Workshop on Multi-Agents', Bangkoku, Thailand, pp. 359–371.

PhD theses

- **Truong Xuan Viet**, "Optimization by Simulation of an Environmental Surveillance Network: Application to the Fight against Rice Pests in the Mekong Delta (Vietnam)", University of Paris 6 & Ho Chi Minh University of Technology, defended June 24th, 2014.
  - **Nguyen Nhi Gia Vinh**, "Designing multi-scale models to support environmental decision: application to the control of Brown Plant Hopper

invasions in the Mekong Delta (Vietnam)", University of Paris 6, defended Oct. 31st, 2013.

- **Vo Duc An**, "An operational architecture to handle multiple levels of representation in agent-based models", University of Paris 6, defended Nov. 30th 2012.
- **Amouroux Edouard**, "KIMONO: a descriptive agent-based modeling methodology for the exploration of complex systems: an application to epidemiology", University of Paris 6, defended Sept. 30th, 2011.
- **Chu Thanh Quang**, "Using agent-based models and machine learning to enhance spatial decision support systems: Application to resource allocation in situations of urban catastrophes", University of Paris 6, defended July 1st, 2011.
- **Nguyen Ngoc Doanh**, "Coupling Equation-Based and Individual-Based Models in the Study of Complex Systems: A Case Study in Theoretical Population Ecology", University of Paris 6, defended Dec. 14th, 2010.

Research papers that use GAMA as modeling/simulation support

2014

- E. G. Macatulad , A. C. Blanco (2014) 3DGIS-BASED MULTI-AGENT GEOSIMULATION AND VISUALIZATION OF BUILDING EVACUATION USING GAMA PLATFORM. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XL-2, 2014. ISPRS Technical Commission II Symposium, 6 – 8 October 2014, Toronto, Canada. Retrieved from http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-2/87/2014/isprsarchives-XL-2-87-2014.pdf

  - S. Bhamidipati (2014) A simulation framework for asset management in climate-change adaptation of transportation infrastructure. In: Proceedings of 42nd European Transport Conference. Frankfurt, Germany. Retrieved from http://abstracts.aetransport.org/paper/download/id/4317
  - Gaudou, B., Sibertin-Blanc, C., Thérond, O., Amblard, F., Auda, Y., Arcangeli, J.-P., Balestrat, M., Charron-Moirez, M.-H., Gondet, E., Hong,

Y., Lardy, R., Louail, T., Mayor, E., Panzoli, D., Sauvage, S., Sanchez-Perez, J., Taillandier, P., Nguyen, V. B., Vavasseur, M., Mazzega, P. (2014). The MAELIA multi-agent platform for integrated assessment of low-water management issues. In: International Workshop on Multi-Agent-Based Simulation (MABS 2013), Saint-Paul, MN, USA, 06/05/2013-07/05/2013, Vol. 8235, Shah Jamal Alam, H. Van Dyke Parunak, (Eds.), Springer, Lecture Notes in Computer Science, p. 85-110.

- Gaudou, B., Lorini, E., Mayor, E. (2014.) Moral Guilt: An Agent-Based Model Analysis. In: Conference of the European Social Simulation Association (ESSA 2013), Warsaw, 16/09/2013-20/09/2013, Vol. 229, Springer, Advances in Intelligent Systems and Computing, p. 95-106.

2013

- Drogoul, A., Gaudou, B., Grignard, A., Taillandier, P., & Vo, D. A. (2013). Practical Approach To Agent-Based Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, p. 277-300, Regional Social Sciences Summer University.

  - Drogoul, A., Gaudou, B. (2013) Methods for Agent-Based Computer Modelling. In: Water and its Many Issues. Methods and Cross-cutting Analysis. Stéphane Lagrée (Eds.), Journées de Tam Dao, 1.6, p. 130-154, Regional Social Sciences Summer University.
  - Truong, M.-T., Amblard, F., Gaudou, B., Sibertin-Blanc, C., Truong, V. X., Drogoul, A., Hyunh, X. H., Le, M. N. (2013). An implementation of framework of business intelligence for agent-based simulation. In: Symposium on Information and Communication Technology (SoICT 2013), Da Nang, Viet Nam, 05/12/2013-06/12/2013, Quyet Thang Huynh, Thanh Binh Nguyen, Van Tien Do, Marc Bui, Hong Son Ngo (Eds.), ACM, p. 35-44.
  - Le, V. M., Gaudou, B., Taillandier, P., Vo, D. A (2013). A New BDI Architecture To Formalize Cognitive Agent Behaviors Into Simulations. In: Advanced Methods and Technologies for Agent and Multi-Agent Systems (KES-AMSTA 2013), Hue, Vietnam, 27/05/2013-29/05/2013, Vol. 252, Dariusz Barbucha, Manh Thanh Le, Robert J. Howlett, C. Jain Lakhmi (Eds.), IOS Press, Frontiers in Artificial Intelligence and Applications, p. 395-403.

2012

- Taillandier, P., Therond, O., Gaudou B. (2012), A new BDI agent architecture based on the belief theory. Application to the modelling of cropping plan decision-making. In 'International Environmental Modelling and Software Society', Germany, pp. 107-116.
  - Taillandier, P., Therond, O., Gaudou B. (2012), Une architecture d'agent BDI basée sur la théorie des fonctions de croyance: application à la simulation du comportement des agriculteurs. In 'Journées Francophones sur les Systèmes Multi-Agents', France, pp. 107-116.
  - NGUYEN, Quoc Tuan, Alain BOUJU, and Pascal ESTRAILLIER. "Multi-agent architecture with space-time components for the simulation of urban transportation systems." (2012).
  - Cisse, A., Bah, A., Drogoul, A., Cisse, A.T., Ndione, J.A., Kebe, C.M.F. & Taillandier P. (2012), Un modèle à base d'agents sur la transmission et la diffusion de la fièvre de la Vallée du Rift à Barkédji (Ferlo, Sénégal), Studia Informatica Universalis 10 (1), pp. 77-97.
  - Taillandier, P., Amouroux, E., Vo, D.A. and Olteanu-Raimond A.M. (2012), Using Belief Theory to formalize the agent behavior: application to the simulation of avian flu propagation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 575-587.
  - Le, V.M., Adam, C., Canal, R., Gaudou, B., Ho, T.V. and Taillandier, P. (2012), Simulation of the emotion dynamics in a group of agents in an evacuation situation. In 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, Volume 7057/2012, pp. 604-619.
  - Nguyen Vu, Q. A., Canal, R., Gaudou, B., Hassas, S., Armetta, F. (2012), TrustSets - Using trust to detect deceitful agents in a distributed information collecting system. In: Journal of Ambient Intelligence and Humanized Computing, Springer-Verlag, Vol. 3 N. 4, p. 251-263.

2011

- Taillandier, P. & Therond, O. (2011), Use of the Belief Theory to formalize Agent Decision Making Processes : Application to cropping Plan Decision

Making. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 138-142.

  - Taillandier, P. & Amblard, F. (2011), Cartography of Multi-Agent Model Parameter Space through a reactive Dicotomous Approach. In '25th European Simulation and Modelling Conference', Guimaraes, Portugal, pp. 38-42.
  - Taillandier, P. & Stinckwich, S. (2011), Using the PROMETHEE Multi-Criteria Decision Making Method to Define New Exploration Strategies for Rescue Robots', IEEE International Symposium on Safety, Security, and Rescue Robotics, Kyoto, Japon, pp. 321 - 326.

2010

- Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), A cluster-based approach for disturbed, spatialized, distributed information gathering systems, in 'The first Pacific Rim workshop on Agent-based modeling and simulation of Complex Systems (PRACSYS)', India, pp. 588-603.

  - Nguyen, N.D., Taillandier, P., Drogoul, A. and Augier, P. (2010), Inferring Equation-Based Models from Agent-Based Models: A Case Study in Competition Dynamics.In 'The 13th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', India, Volume 7057/2012, pp. 413-427.
  - Amouroux, E., Gaudou, B. Desvaux, S. and Drogoul, A. (2010), O.D.D.: a Promising but Incomplete Formalism For Individual-Based Model Specification. in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), pp. 1-4.
  - Nguyen, N.D., Phan, T.H.D., Nguyen, T.N.A., Drogoul, A. and Zucker, J-D. (2010), Disk Graph-Based Model for Competition Dynamic, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').
  - Nguyen, T.K., Marilleau, N., Ho T.V. and El Fallah Seghrouchni, A. (2010), A meta-model for specifying collaborative simulation, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF').

  - Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S. and Armetta, F. (2010), `TrustSets` - Using trust to detect deceitful agents in a distributed information collecting system, Paper to appear in 'IEEE International Conference on Computing and Telecommunication Technologies'(2010 IEEE RIVF'), the best student paper award.
  - Nguyen Vu, Q.A. , Gaudou, B., Canal, R., Hassas, S., Armetta, F. and Stinckwich, S. (2010), Using trust and cluster organisation to improve robot swarm mapping, Paper to appear in 'Workshop on Robots and Sensors integration in future rescue INformation system' (ROSIN 2010).

2009

- Taillandier, P. and Buard, E. (2009), Designing Agent Behaviour in Agent-Based Simulation through participatory method. In 'The 12th International Conference on Principles and Practices in Multi-Agent Systems (PRIMA)', Nagoya, Japan, pp. 571–578.
  - Taillandier, P. and Chu, T.Q. (2009), Using Participatory Paradigm to Learn Human Behaviour. In 'International Conference on Knowledge and Systems Engineering', Ha noi, Viet Nam, pp. 55–60.
  - Gaudou, B., Ho, T.V. and Marilleau, N. (2009), Introduce collaboration in methodologies of modeling and simulation of Complex Systems. In 'International Conference on Intelligent Networking and Collaborative Systems (INCOS '09)'. Barcelona, pp. 1–8.
  - Nguyen, T.K., Gaudou B., Ho T.V. and Marilleau N. (2009), Application of PAMS Collaboration Platform to Simulation-Based Researches in Soil Science: The Case of the MIcro-ORganism Project. In 'IEEE International Conference on Computing and Telecommunication Technologies (IEEE-RIVF 09)'. Da Nang, Viet Nam, pp. 296–303.
  - Nguyen, V.Q., Gaudou B., Canal R., Hassas S. and Armetta F. (2009), Stratégie de communication dans un système de collecte d'information à base d'agents perturbés. In 'Journées Francophones sur les Systèmes Multi-Agents (JFSMA'09)'.

2008

- Chu, T.Q., Boucher, A., Drogoul, A., Vo, D.A., Nguyen, H.P. and Zucker, J.D. (2008). Interactive Learning of Expert Criteria for Rescue Simulations. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 127–138.

  - Amouroux, E., Desvaux, S. and Drogoul, A. (2008), Towards Virtual Epidemiology: An Agent-Based Approach to the Modeling of H5N1 Propagation and Persistence in North-Vietnam. In 'Pacific Rim International Workshop on Multi-Agents', Ha Noi, Viet Nam, pp. 26–33.

Part XV

Training Session

1883

Chapter 173

Training Session

Modeling for supporting decision in urban management issues

7-11 December 2015 - Siem Reap (Cambodia)

This training session took place at the Apsara Authorities, where we introduced how to build a model with agent-based approach, using GAMA. In a new and very fast-growing city such as Siem Reap, some measures have to be taken to anticipate the future of the city, and modeling is a science that can give some solutions to face those problems.

The training session was divided into 2 parts:

- A theoretical part (3 days) dealing with the following subjects :
  - Urban issues and introduction to Agent-Based Modeling
  - Presentation of the modeling methodology
  - Introduction to GAMA with a model on urban segregation
  - GIS datas and graphs to model urban mobility
  - GIS, Raster datas and graphs to model urban growth
  - Use of experiments to calibrate and explore models
- A practical part (2 days) to build a model about urban mobility in Siem Reap (by groups of 4/5 people)

Figure 173.1: resources/other/trainingSession/SiemReap2015/photos/group.JPG


Trainers: Drogoul Alexis, Gaudou Benoit, Trung Quang, Philippon Damien, Mazars Julien.

A Glance at Sustainable Urban Development (JTD)

July 2014 - Da lat (Vietnam)

The JTD (Journées de Tam Dao) is an annual gathering of french-talkers researchers during the summer for one week, dealing with a specific subject related to sustainable development. For this 8th JTD, the topic was about sustainable urban development, and a workshop has been made especially about how to use tools as GAMA to build models in order to explore and understand urban spatial dynamics.

Trainers: Drogoul Alexis, Banos Arnaud, Huynh Quang Nghi, Truong Chi Quang, Vo Duc An.

Here is the link to download the pdf report of the JTD 2014: https://drive.google.com/file/d/0B2Go6pohIhQcbERhczZRd253UUU/view.

The perception and Management of Risk (JTD)

July 2013 - Da lat (Vietnam)

The JTD (Journées de Tam Dao) is an annual gathering of french-talkers researchers during the summer for one week, dealing with a specific subject related to sustainable

development. For this 7th JTD, the topic was about the perception and management of risks, and a workshop has been made especially about how to use tools as GAMA to build models in order to understand past crises to better understand the present.

Trainers: Alexis Drogoul, Benoit Gaudou, Nasser Gasmi, Arnaud Grignard, Patrick Taillandier, Olivier Tessier, Vo Duc An

Here is the link to download the pdf report of the JTD 2013: https://drive.google.com/file/d/0B2Go6pohIhQcNXFwVllHd2pFdlk/view.

Water and its many Issues (JTD)

July 2012 - Vietnam

The JTD (Journées de Tam Dao) is an annual gathering of french-talkers researchers during the summer for one week, dealing with a specific subject related to sustainable development. For this 6th JTD, the topic was about the perception and management of risks, and a workshop has been made especially about how to use tools as GAMA to build models with an agent-based approach.

Trainers : Alexis Drogoul, Benoit Gaudou, Arnaud Grignard, Patrick Taillandier, Vo Duc An

Here is the link to download the pdf report of the JTD 2012: https://docs.google.com/file/d/0B2Go6pohIhQcUWRKU2hPelNqQmc/view.

Chapter 174

Introduction

Le but est de proposer une réorganisation des plug-ins de Gama

Cf. page Google Doc: https://docs.google.com/document/d/1gd4nlJH8ns4_iKqiId-w3LSa__Sk9yUccX9TEbzr2PW0/edit?usp=sharing

Part XVI

Events

1891

Chapter 175

References

This page references the events that are linked to GAMA.

If you happen to participate to an event linked to GAMA, please let us know, so that we can include it in this list.

Events linked to GAMA

List of GAMA Coding Camps : * Coding Camp March 2014 (photos) * Coding Camp March 2012 * Fall Coding Camp 2012 * Programme doctoral internationale 2012

Part XVII

Older versions

1895

Chapter 176

Versions of GAMA

GAMA exists since 2007 and has undergone a number of changes since its first release, materialized by different versions of the platform. Although we do not maintain these versions anymore, some are still used in lectures or specific projects. This page provides a list of these versions and, for each of them, a summary of its features and a link to its documentation in PDF format.

GAMA 1.8 (July 2019)

Version 1.8 is the current version of GAMA. It improves many features over version 1.7.

- Documentation: Documentation in PDF

GAMA 1.7 RC 2 (April 2017)

This version introduced many features over 1.6.1:

GAMA 1.6.1 (June 2014)

GAMA 1.6.1 improves many features over version 1.6:

Figure 176.1: images/splash_1_8.png

Figure 176.2: images/splash_1_7.png

- OpenGL displays improvements
- Various enhancements for displays (overlay, focus, light, trace, DEM, colors with alpha, addition of CSS colors...)
- Compilation of models (errors tracked, memory, report of errors, etc.)
- Validation of stochastic models (random now extended to every areas of a model, incl. the operations on HashSets)
- GIS file handling (esp. with the possibility to pass a custom CRS)
- Handling of OSM data
- Comparison of raster and vectorial maps
- Traffic moving skill improvement
- Handling of various other file types (uniform constructors, ...)
- Documentation itself (both online and on the website)
- Testing framework and debugging tools (trace, sample)
- Speed of computation of several spatial operators
- Experiments (permanent output, access to the duration of cycles)
- Type system (complete parametric definitions of types)
- Addition of several 3D operations
- Auto-update mechanism of plugins
- Automatic importation of files/projects when double-clicked in the OS (or transmitted by command line arguments)
- Definition of charts (dynamic data lists, marker type, range, etc.)

Documentation:

- Documentation in PDF

GAMA 1.6 (July 2013)

This version introduced many features over 1.5.1:

- correction of bugs (in particular, freeze, memory consumption)
- performance improvement (in particular for "big" models)
- further simplification of the GAML language (assignments, manipulation of containers)
- integration of an agent browser

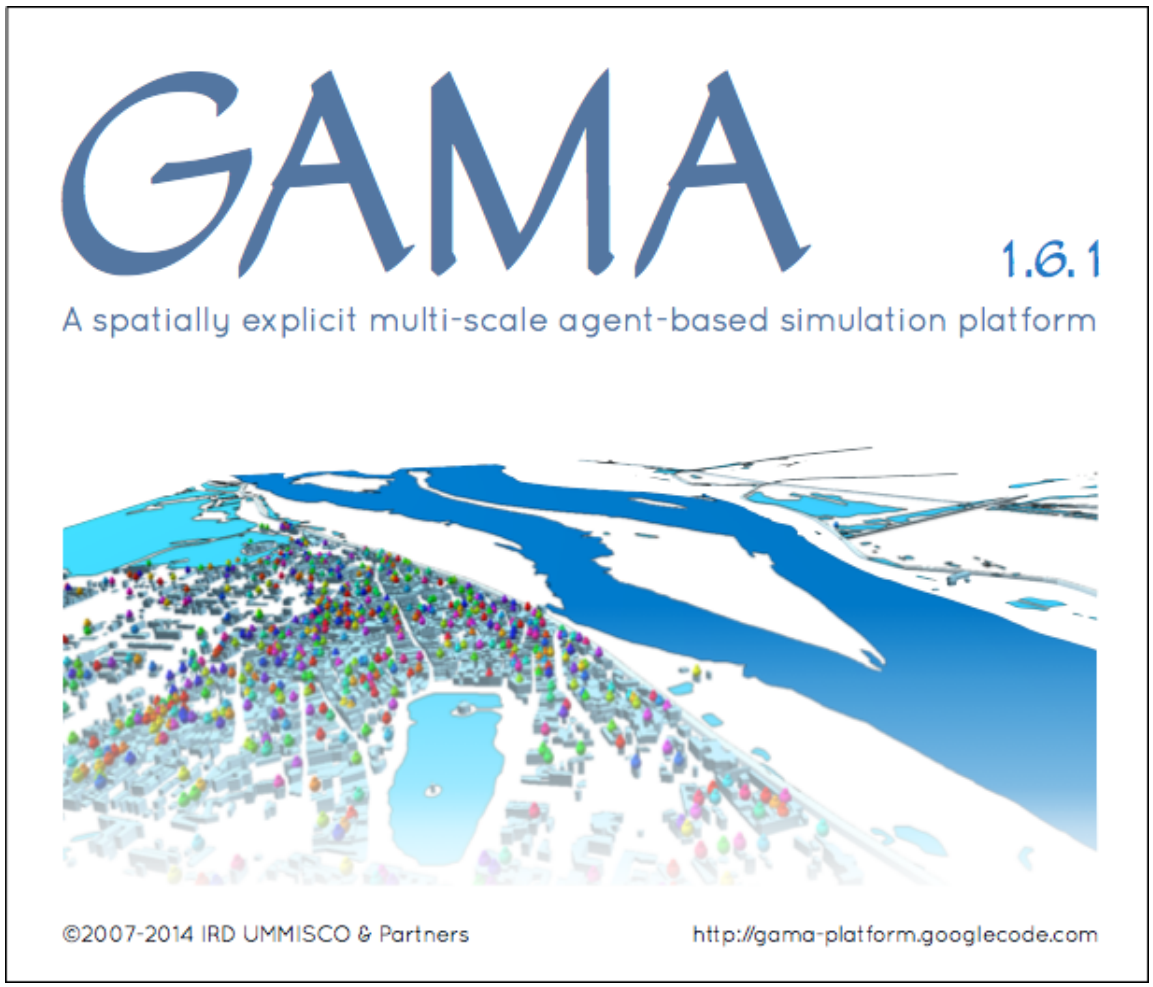


Figure 176.3: images/splash_1_61.png

- improvement of the 3D integration (new operators, new display facet bug corrections. . .)
- removing of the environment block
- more user/simulation interaction (event statement)

Documentation:

- Documentation in PDF

GAMA 1.5 / 1.5.1 (July & November 2012)

Key points:

- Improvement of the performance and stability
- Simplification of the modeling language (omissible facets, types as declarations, etc.)
- Integration of non-spatial graphs
- Introduction of OpenGL displays
- Improvement of the user interaction in the simulation perspective
- Generalization of the notion of experiment

Version 1.5.1 improved some features of 1.5:

- correction of bugs (in particular, no more freezes when reloading an experiment)
- performance improvement (in particular for “big” models)
- improvement of the 3D integration (new operators to add a “z” to geometries, bug corrections. . .)
- new models (driving_traffic, Vote, 3D models)

Documentation:

- Documentation in PDF

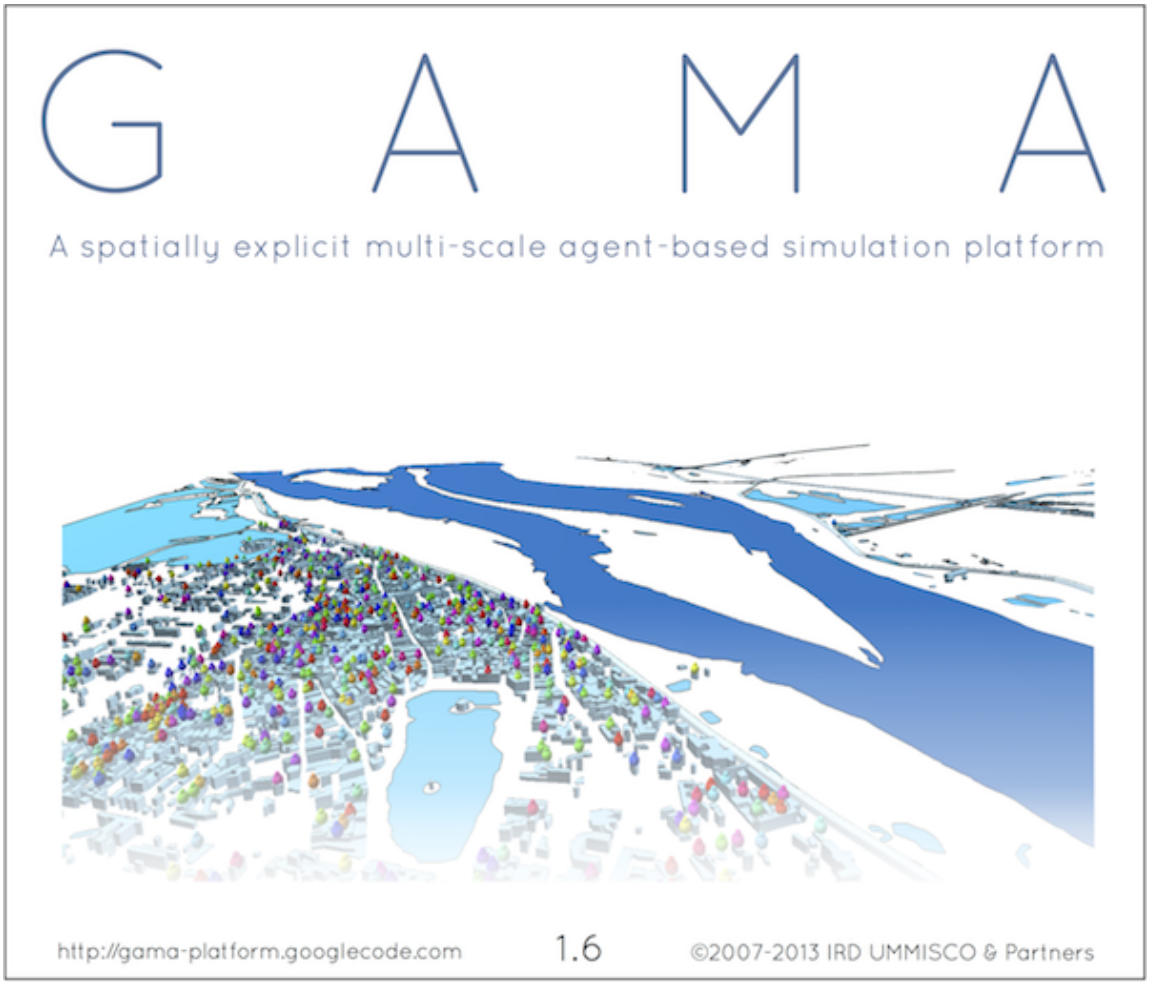


Figure 176.4: images/splash_1_6.png

Figure 176.5: images/splash_1_5.png

GAMA 1.4 (December 2011)

This versions introduced the new integrated development environment based on XText, as well as a completely revamped modeling language, not based on XML anymore, and much easier to read and write. It also introduced new important types such as geometry, graph, path, and topology in order to ease the spatial manipulation of agents.

Key points:

- Deep refactoring of the source code
- New programming language (based on XText)
- Integration of a true IDE based on Eclipse/Xtext
- Deep refactoring of the meta-model
- Better integration of multi-level species
- New important notion: topology
- New variable types: geometry, graph, path, topology
- Many more novelties/improvements/enrichments...

Documentation:

- Documentation in PDF

GAMA 1.3 (August 2010)

Version 1.3 added numerous operators to manipulate GIS data. Moreover, it integrated new features like the possibility to define custom displays and to define multi-level models. It allowed to use clustering and decision-making methods. And it greatly improved the performances of the platform.

Key points:

- Important improvement of the performance of the platform
- Improvement of the simulation display
- Enrichment of the spatial operators/actions
- Integration of multi-level models

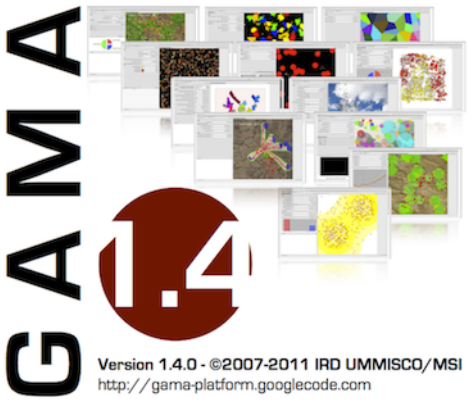


Figure 176.6: images/splash_1_4.png

- Integration of clustering algorithms
- Integration of decision-making algorithms

Documentation:

- Documentation in PDF

GAMA 1.1 (October 2009)

The first "real" release of GAMA, which incorporated several changes, including an XML editor coupled with the simulator, the possibility to take snapshots of every graphical window, save parameters and monitors for future reuse, save charts as CSV files, and definitely fixed the memory leaks observed in previous internal versions.

Key points:

- Transformation into a more complete and versatile platform
- Addition of batch controllers
- Integration of a true modeling language (based on XML)

Figure 176.7: images/splash_1_3.png

- First release as an open-source project

Documentation:

- Guidebook of GAMA 1.1
- Training session on GAMA 1.1 (Hanoi)

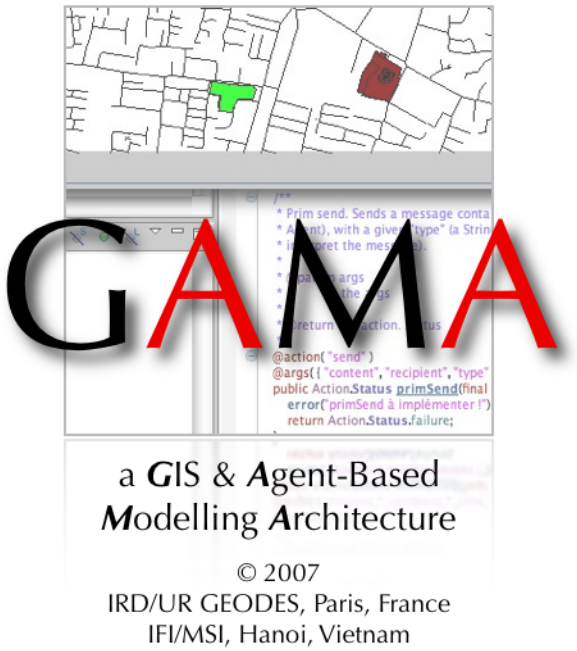


Figure 176.8: images/splash_1_1.png