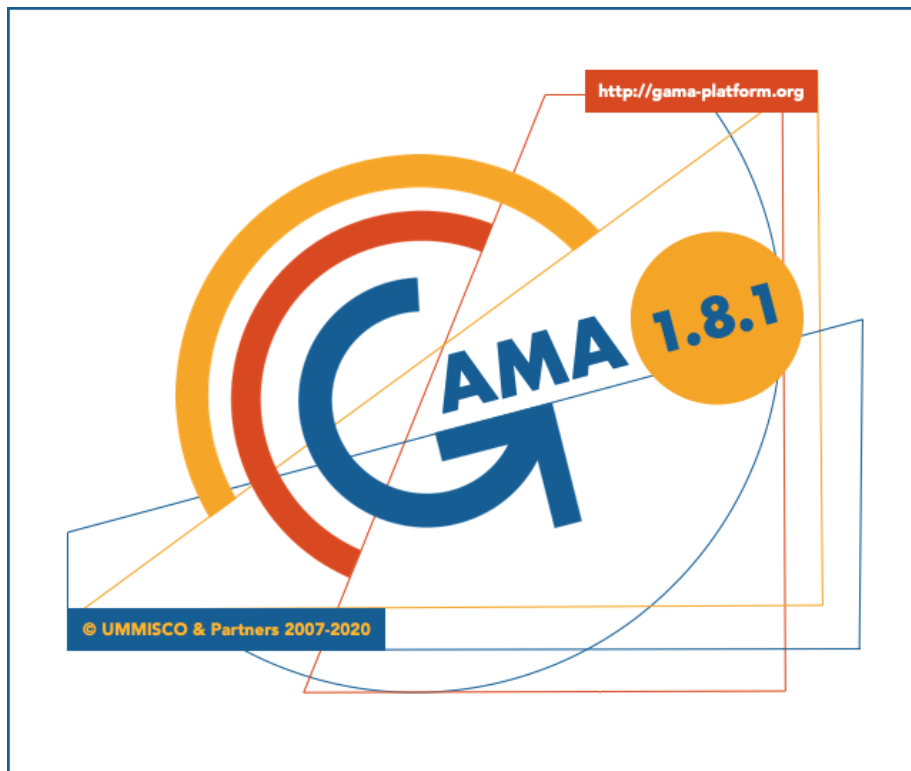


GAMA v1.8.2 documentation

by GAMA team

<http://gama-platform.org>



Contents

I	Home	13
1	GAMA	15
	Multiple application domains	15
	Training sessions	16
	High-level and intuitive agent-based language	16
	GIS and Data-Driven models	17
	Declarative user interface	17
	Development Team	19
	Citing GAMA	19
	Acknowledgement	20
2	Introduction	21
	Documentation	23
	Source Code	23
	Copyright Information	24
	Developers	24
	Citing GAMA	25
	Contact Us	26

II Platform	27
3 Platform	29
4 Installation and Launching	31
5 Workspace, Projects and Models	33
6 Editing models	35
7 Running Experiments	37
8 Preferences	39
Table of contents	39
Opening Preferences	40
Interface	40
Editors	43
Execution	45
Displays	47
Data and Operators	50
Manage preferences in GAML	54
Advanced Preferences	55
9 Troubleshooting	57
Table of contents	57
On Ubuntu (& Linux Systems)	58
On macOS	58
Memory problems	58
Submitting an Issue	59

III Learn GAML step by step	65
10 Learn GAML Step by Step	67
How to proceed to learn better?	67
11 Introduction	69
Table of contents	70
Lexical semantics of GAML	70
Translation into a concrete syntax	72
Vocabulary correspondence with the object-oriented paradigm as in Java .	74
Vocabulary correspondence with the agent-based paradigm as in NetLogo .	74
12 Manipulate basic species	77
13 The global species	79
Index	79
Declaration	79
Environment size	81
Built-in attributes	81
Built-in Actions	84
The <code>init</code> statement	85
14 Defining advanced species	87
15 Defining GUI Experiment	89
Types of experiments	89
Experiment attributes	90
Experiment facets	90
Defining displays layout	91
Defining elements of the GUI experiment	91

16 Exploring Models	93
17 Optimizing Models	95
18 Multi-Paradigm Modeling	97
IV Recipes	101
19 Recipes	103
20 Manipulate OSM Datas	105
21 Implementing diffusion	121
Index	121
Diffuse statement	122
Diffusion with matrix	124
Diffusion with parameters	129
Computation methods	132
Using a mask	133
Pseudo-code	139
22 Using Database Access	141
Description	142
Supported DBMS	142
SQLSKILL	143
MDXSKILL	149
AgentDB	154
Using database features to define environment or create species	159

23 Calling R	163
Introduction	163
Table of contents	163
Configuration in GAMA	164
Calling R from GAML	164
24 Using FIPA ACL	171
Table of Contents	171
Main steps to create a conversation using FIPA Communication Acts and Interaction Protocols	172
Attach the <code>fipa</code> skill to a species	172
Initiate a conversation	173
Receive messages	173
Reply to a received message	174
End a conversation	175
The <code>message</code> type	175
The <code>conversation</code> data type	175
25 Using GAMAnalyzer	177
Install	177
Built-in Variable	177
Example	178
26 Using BEN (simple_bdi)	181
Introduction to BEN	181
The BEN architecture	181
Predicates, knowledge and personality	183
Perception	191
Managing knowledge bases	194
Making Decision	201

27 Known issues	275
Crash when using openGL on Windows	275
Grid not displayed right using openGL	275
V GAML References	279
28 GAML References	281
Index of keywords	281
29 Built-in Species	283
Table of Contents	283
agent	284
AgentDB	284
base_edge	288
experiment	288
graph_edge	290
graph_node	291
physical_world	291
30 Built-in Skills	295
Introduction	295
Table of Contents	296
advanced_driving	296
driving	306
dynamic_body	308
fipa	310
MDXSKILL	317
messaging	318

moving	319
moving3D	322
network	323
public_transport	327
public_transport_scheduler	330
skill_road	331
skill_road_node	333
SQLSKILL	333
static_body	337
31 Built-in Architectures	339
INTRODUCTION	339
Table of Contents	339
fsm	340
parallel_bdi	340
probabilistic_tasks	340
reflex	341
rules	341
simple_bdi	341
sorted_tasks	370
user_first	370
user_last	371
user_only	371
weighted_tasks	371
32 Statements	373
Table of Contents	373
Statements by kinds	374

Statements by embedment	377
General syntax	380
33 Types	523
Table of contents	523
Primitive built-in types	525
Complex built-in types	527
How to change the processor	593
34 General workflow of file generation	595
VI Projects using GAMA	597
35 Projects	599
Publications	599
Projects	599
36 Scientific References	611
Table of Contents	611
Papers about GAMA	612
HDR theses	613
PhD theses	613
PhD theses that use GAMA as modeling/simulation support	614
Master theses that use GAMA as modeling/simulation support	615
Research papers that use GAMA as modeling/simulation support	616
37 Training Session	635
SEARCA Phillippines 2021 (Online)	635
AWP 2021(Online)	635

USTH Training session 2020	635
SMAC Toulouse 2020	635
Application to disaster management and evacuation	636
Training session TLU 2019	636
AWP Phnom Penh 2019	636
Formation Toulouse 2019	636
Training session Brasilia 2019	636
Application to disaster management and evacuation	637
AWP Can Tho 2018	637
SCEMSITE 2018	637
Formation Toulouse 2018	637
GAMA 1.7RC1 training session - Pays-Bas	637
Analysis of land use dynamics (JTD 2017)	637
Master TRIAD 2017	638
EDSS USTH Master 2016	638
Design urban energy transition policies (JTD 2016)	638
Modeling for supporting decisions in urban management issues	638
Epidemiological risks and the integration of regional health policies (JTD 2015)	641
MAPS 8 2015	641
Nex Days 2015 (GAMA 1.6.1)	641
MISS ABMS 2014	641
MAPS epidemic city tutorial 2014	641
GAMA training session Phillipines	641
A Glance at Sustainable Urban Development (JTD)	642
AUF 2013	642
MISS ABM 2013	642
The perception and Management of Risk (JTD)	642

Can Tho training session 2012	643
ESSA Tutorial 2012	643
Water and its many Issues (JTD)	643
Introduction of GAMA 1.4	643
Formation à IRD Bondy	644
Introduction to the GAMA and PAMS platforms (IFI 2009)	644
38 Events	645
Events linked to GAMA	645

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, **1.8.2**, 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 particular needs. The source code is available from the dedicated [Github repository](#).

Example of application domains where GAMA is mostly present:

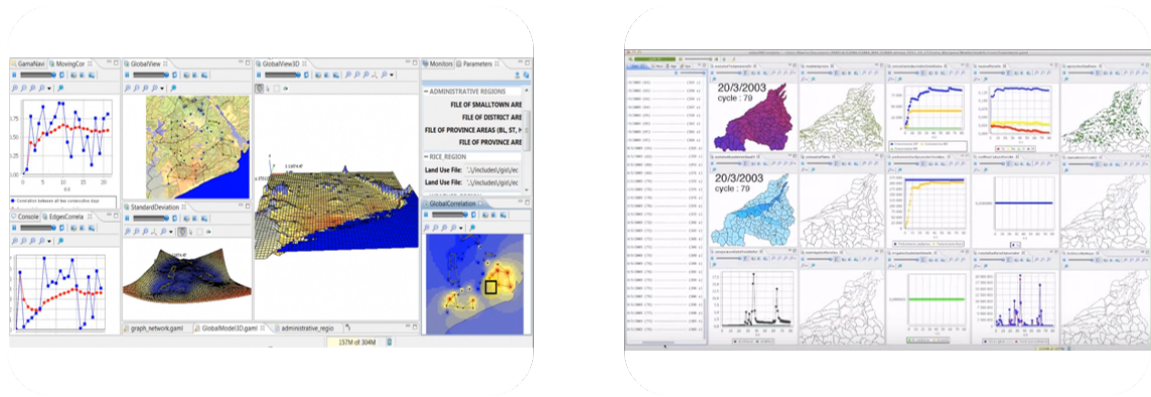


Figure 1.1: Multiple application domains

- 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

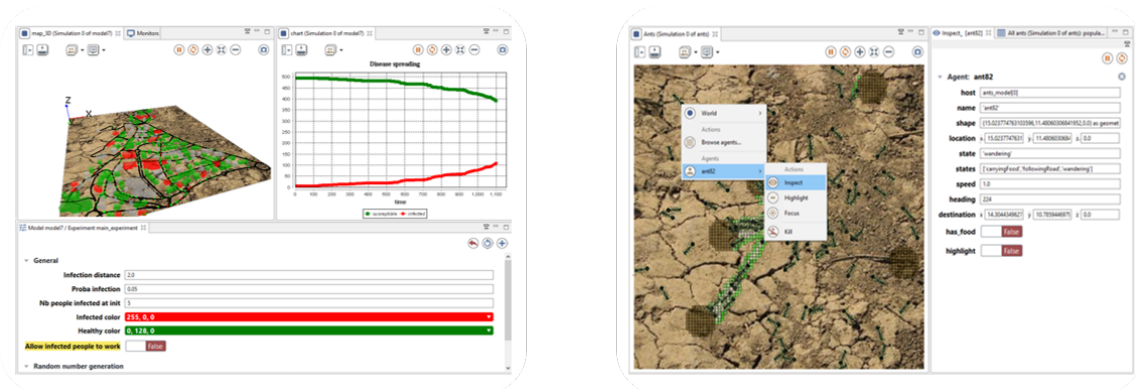


Figure 1.4: Declarative User Interface

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/SU, 32 Avenue Henri Varagnat, 93143 Bondy Cedex, France.
- [ACROSS International Joint Lab](#), Thuyloi University, Hanoi, Vietnam (since 2021)
- [DREAM Research Team](#), University of Can Tho, Vietnam (since 2011).
- [UMR 5505 IRIT](#), CNRS/University of Toulouse 1, France (since 2010).
- [UR MIAT](#), INRAE, 24 Chemin de Borde Rouge, 31326 Castanet Tolosan Cedex, France (since 2016).
- [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.

This page is licensed under a Creative Commons Attribution 4.0 International License.

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

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

Developers

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

- **Alexis Drogoul**, Senior Researcher at the [IRD](#), member of the [UMMISCO](#) International Research Unit. Mostly working on agent-based modeling and simulation. Has contributed and still contributes to the original design of the platform, including the GAML language (from the meta-model to the editor) and simulation facilities like Java2D displays.
- **Patrick Taillandier**, 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-MS](#)) 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.

This page is licensed under a Creative Commons Attribution 4.0 International License.

Part II
Platform

Chapter 3

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 4

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 patches](#). In this part, we will also present you briefly an other way to launch GAMA without any GUI : the [headless mode](#).

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

Chapter 5

Workspace, Projects and Models

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

Except when running in [headless mode](#), **GAMA cannot function without a valid workspace**.

The workspace is organized in 4 [categories](#), which are themselves organized into **projects**.

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

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

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

1. [Navigating in the Workspace](#)

2. [Changing Workspace](#)
3. [Importing Models](#)

Chapter 6

Editing models

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

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

This section covers the following sub-sections:

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

Chapter 7

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 8

Preferences

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

Please note that the preferences specific to GAMA will be shared, on the same machine, and for the 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

- [Preferences](#)
 - [Opening Preferences](#)
 - [Interface](#)
 - [Editors](#)
 - [Execution](#)
 - [Displays](#)
 - [Data and Operators](#)
 - [Manage preferences in GAML](#)
 - [Advanced Preferences](#)

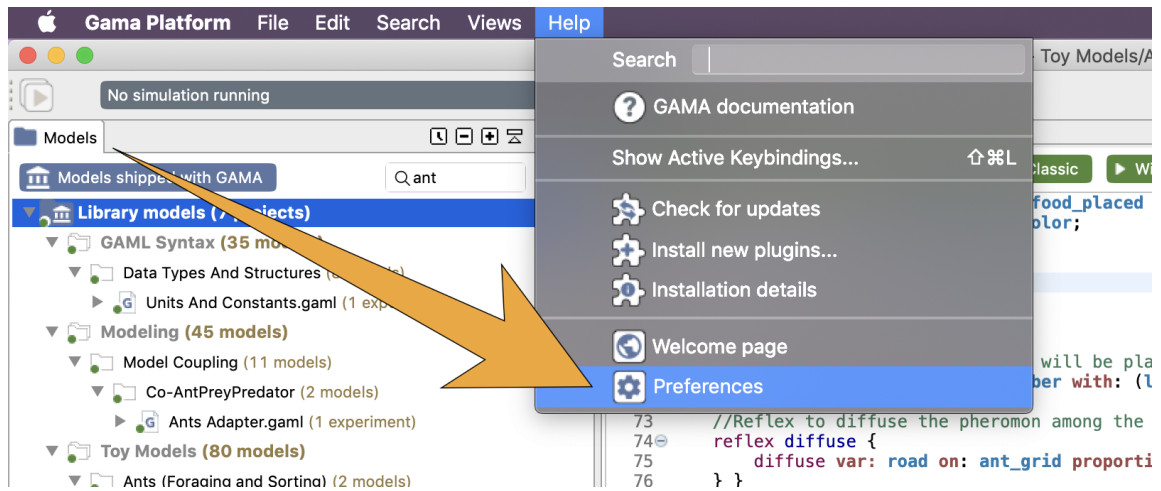


Figure 8.1: Open the Preferences from the “Help” menu of the interface.

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.

Interface

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

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

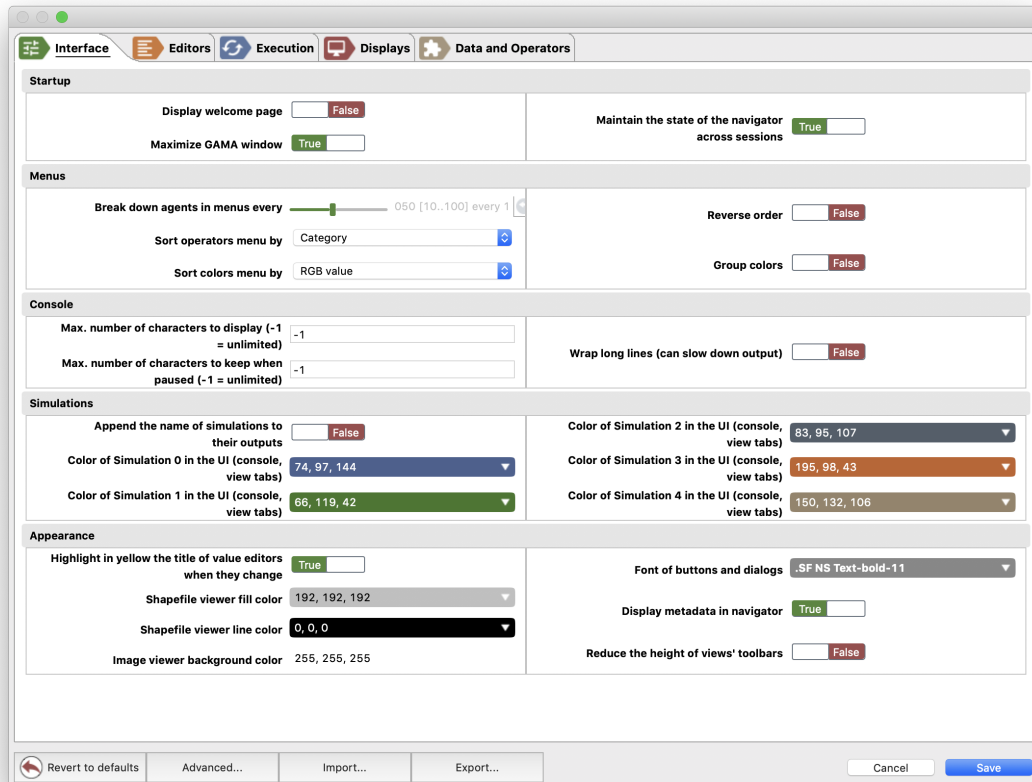


Figure 8.2: Interface pane in Preferences.

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

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

Editors

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

- **Options**

- **Show warning markers in the editor:** if false, the warning will only be available from the Validation View.
- **Show information markers in the editor:** if false, the information will only be available from the Validation View.
- **Save all editors when switching perspectives**
- **Hide editors when switching to simulation perspectives (can be overridden in the ‘layout’ statement)**
- **Applying formatting on save:** if true, every time a model file is saved, its code is formatted.
- **Save all model files before launching an experiment**
- **Drag files and resources as references in GAML files:** a GAML model file is dropped in another file as an import and other resources as the definition of a variable accessing to this resource.
- **Ask before saving each file**

- **Edition**

- **Close curly brackets ({})**
- **Close square brackets ([])**
- **Close parentheses**
- **Turn on colorization of code sections:** if true, it activates the colorization of code blocks in order to improve the visual understanding of the code structure.
- **Font of editors**

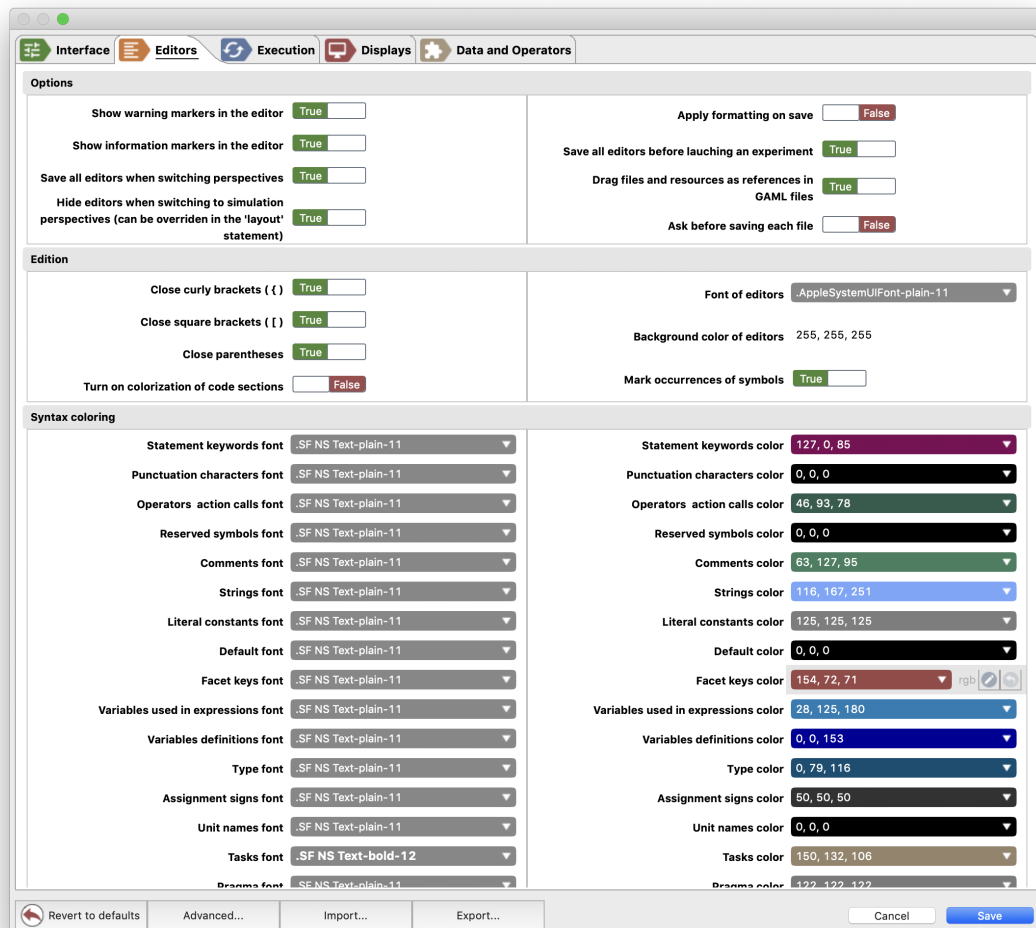


Figure 8.3: Editors pane in Preferences.

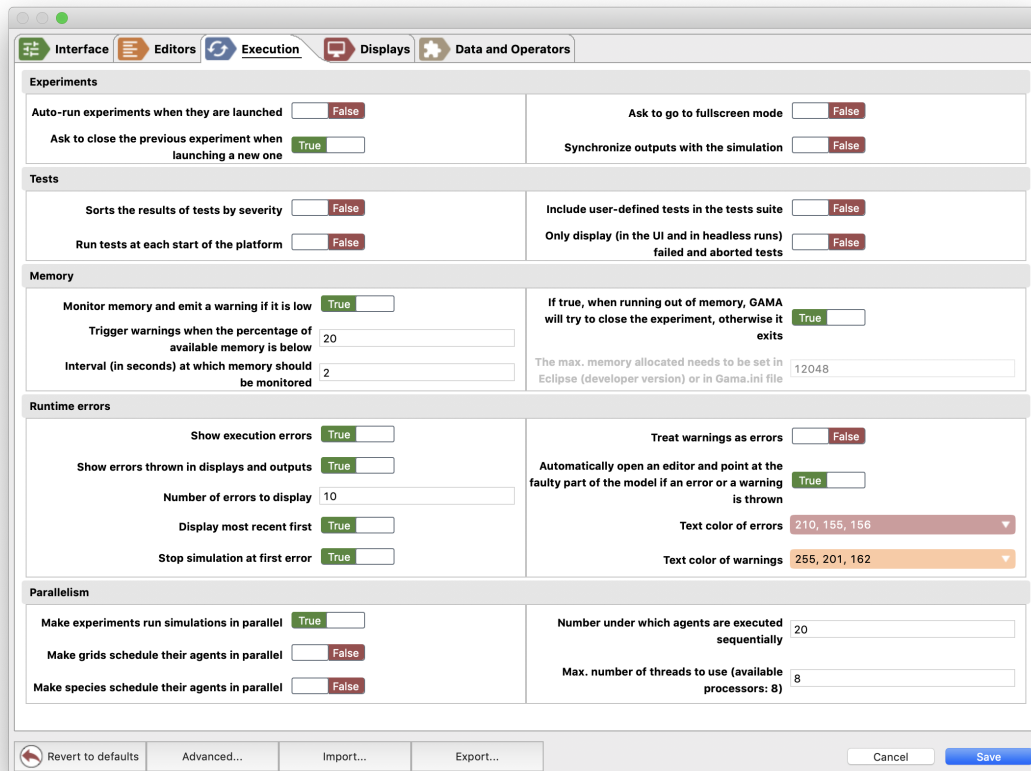


Figure 8.4: Execution pane in Preferences.

- **Background color of editors**
- **Mark occurrences of symbols:** if true, when a symbol is selected, all its other occurrences are also highlighted.
- **Syntax coloring:** this section allows the modeler to set the font and color of each GAML keyword kind in the syntax coloring (in any GAMA editor).

Execution

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

- **Experiments:** various settings regarding the execution of experiments.
 - **Auto-run experiments when they are launched:** see [this page](#).
 - **Ask to close the previous simulation before launching a new one:** if false, previous simulations (if any) will be closed without warning.
 - **Ask to go to fullscreen mode:** if true, ask the modeler before switching to the fullscreen mode.
 - **Synchronize outputs with the simulation:** if true, simulation cycles will wait for the displays to have finished their rendering before passing to the next cycle (this setting can be changed on an individual basis dynamically [here](#)).
- **Tests**
 - **Sorts the results of tests by severity**
 - **Run tests at each start of the platform**
 - **Include user-defined tests in the tests suite**
 - **Only display (in the UI and in headless runs) failed and aborted tests**
- **Memory:** a given amount of memory (RAM) is allocated to the execution of GAMA (it has to be set in the `Gama.ini` file). The allocated memory size should be chosen in accordance with the requirements of the model that is developed and the other applications running in your OS.
 - **Monitor memory and emit a warning if it is low:** a warning will appear during an experiment run when the memory is low.
 - **Trigger warnings when the percentage of available memory is below**
 - **Interval (in seconds) at which memory should be monitored**
 - **If true, when running out of memory, GAMA will try to close the experiment, otherwise it exits**
- **Runtime errors:** how to manage and consider simulation errors.
 - **Show execution errors:** whether errors should be displayed or not.
 - **Show errors thrown in displays and outputs:** the code defined inside the `aspect` block of a species will be executed each time the agents are repainted in a display. In particular, when the displays are not synchronized, some errors can occur due to some inconsistency between the model and the display (e.g. drawing a dead agent). As a consequence, the code executed inside an aspect should be limited as much as possible.

- **Number of errors to display:** how many errors should be displayed at once
- **Display most recent first:** errors will be sorted in the inverse chronological order if true.
- **Stop simulation at first error:** if false, the simulations will display the errors and continue (or try to).
- **Treat warnings as errors:** if true, no more distinction is made between warnings (which do not stop the simulation) and errors (which can potentially stop it).
- **Automatically open an editor and point at the faulty part of the model if an error or a warning is thrown**
- **Text color of errors**
- **Text color of warnings**
- **Parallelism:** various settings regarding the parallel execution of experiments.
 - **Make experiments run simulations in parallel:** if true, in the case of a multi-simulations experiment, the simulation will be executed in parallel (note that the number of simulations that can be executed in parallel will depend on the number of threads to use).
 - **Make grids schedule their agents in parallel:** the agents of grid species will be executed in parallel. Depending on the model, this could increase the simulation speed, but the modeler cannot have any control over the execution order of the agents.
 - **Make species schedule their agents in parallel**
 - **Number under which agents are executed sequentially**
 - **Max. number of threads to use (available processors: 8)**

Displays

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

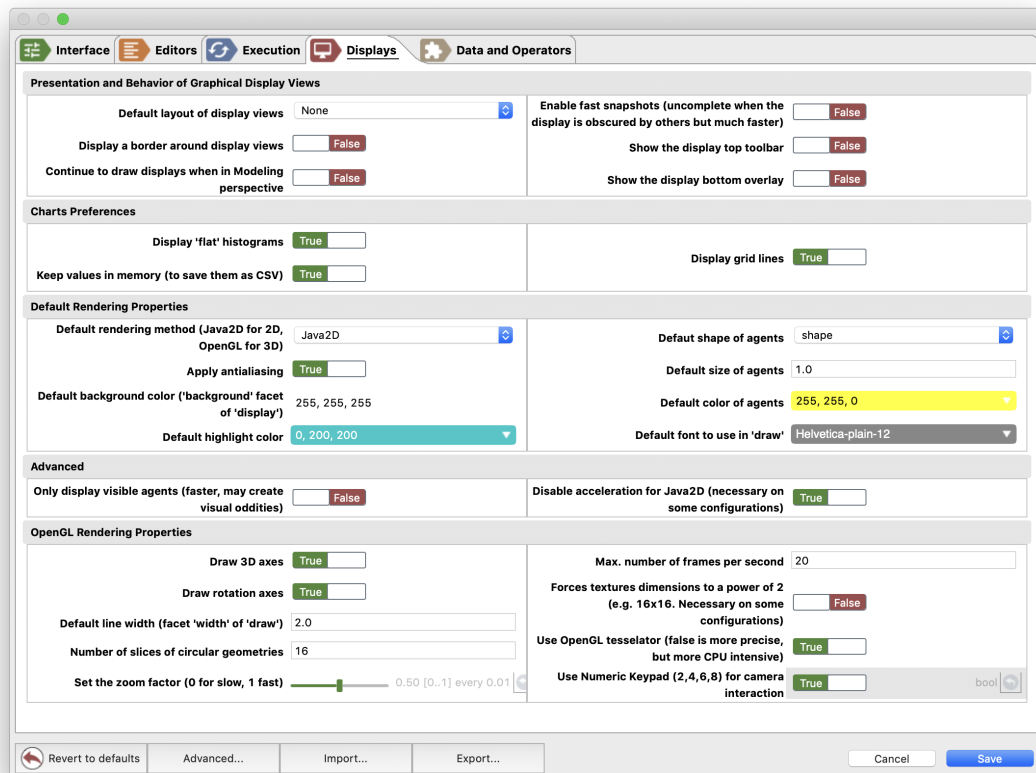


Figure 8.5: Displays pane in Preferences.

- **Continue to draw displays when in Modeling perspective:** if true, when the simulation is running and the modeler chooses to switch to the Modeling perspective the displays are still updated. This is particularly relevant for displays showing plots of data over time.
- **Enable fast snapshots (uncomplete when the display is obscured but much faster)**
- **Show the display top toolbar:** this could also be configured manually for each display (cf [displays related page](#)).
- **Show the display bottom overlay:** this could also be configured manually for each display (cf [displays related page](#)).
- **Charts Preferences**
 - **Display ‘flat’ histograms:** if false, the histograms are displayed in a 3D style.
 - **Keep values in memory (to save them as csv)**
 - **Display grid lines:** in charts (and in particular [series](#)), if true, a grid is displayed in background.
- **Default Rendering Properties:** various properties of displays
 - **Default rendering method (JavaED fro 2D, OpenGL for 3D):** use either ‘Java2D’ or ‘OpenGL’ if nothing is specified in the [declaration of a display](#).
 - **Apply antialiasing:** if true, displays are drawn using antialiasing, which is slower but renders a better quality of image and text (this setting can be changed on an individual basis dynamically [here](#)).
 - **Default background color:** indicates which color to use when none is specified in the [declaration of a display](#).
 - **Default highlight color:** indicates which color to use for highlighting agents in the displays.
 - **Default shape of agents:** a choice between `shape` (which represents the actual geometrical shape of the agent) and geometrical operators (`circle`, `square`, `triangle`, `point`, `cube`, `sphere` etc.) as default shape to display agents when no `aspect` is defined.
 - **Default size of agents:** what size to use. This expression must be constant.
 - **Default color of agents:** what color to use.
 - **Default font to use in ‘draw’**
- **Advanced:**

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

Data and Operators

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

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

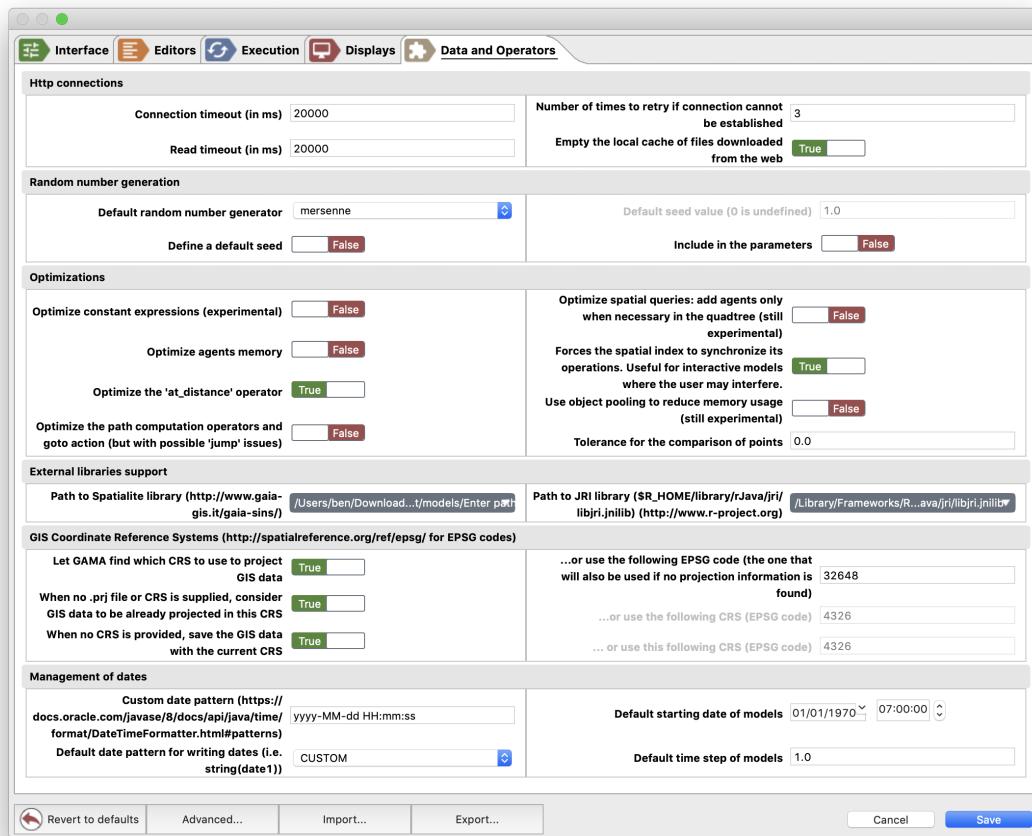


Figure 8.6: The Data and Operators pane in Preferences.

- **Empty the local cache of files downloaded from the web:** if true, after having downloaded the files and used them in the model, the files will be deleted.
- **Random Number Generation:** all the options pertaining to generating random numbers in simulations
 - **Default random number generator:** the name of the generator to use by default (if none is specified in the model).
 - **Define a default seed:** whether or not a default seed should be used if none is specified in the model (otherwise it is chosen randomly by GAMA)
 - **Default seed value (0 is undefined):** the value of this default seed
 - **Include in the parameter:** whether the choice of generator and seed is included by default in the [parameters views](#) of experiments or not.
- **Optimizations**
 - **Optimize constant expressions (experimental):** whether expressions considered as constants should be computed and replaced by their value when compiling models. Allows to save memory and speed, but may cause some problems with complex expressions.
 - **Optimize agents memory:** whether the memory used by agents is reduced (or not) when their structure appears to be simple: no sub-agents, for instance, because no sub-species is defined.
 - **Optimize the ‘at_distance’ operator:** an optimisation that considers the number of elements on each side and changes the loop to consider the fastest case.
 - **Optimize the path computation operators and goto action (but with possible ‘jump’ issues):** when an agent is not already on a path, simplifies its choice of the closest segment to choose and makes it jump directly on it rather than letting it move towards the segment.
 - **Optimize spatial queries: add agents only when necessary in the quadtree (still experimental):** if no queries is conducted against a species of agents, then it is not necessary to maintain them in the global quad tree.
 - **Forces the spatial index to synchronize its operations. Useful for interactive models where the user may interfere.:** when true, forces the quadtree to use concurrent data structures and to synchronize reads and writes, allowing users to interact with the simulation without raising concurrent modification errors.

- **Use object pooling to reduce memory usage (still experimental):** when true, tries to reuse the same common objects (lists, maps, etc.) over and over rather than creating new ones.
- **Tolerance for the comparison of points:** depending on the way they are computed, 2 points who should be the same, could not be equal. This preference allows to be more tolerant in the way points are compared.
- **External libraries support**
 - **Path to Spatialite (<http://www.gaia-gis.it/gaia-sins/>):** the path toward the spatial extension for the SQLite database.
 - **Path to JRI library (`$R_HOME/library/rJava/jri/libjri.jnilib`) (<http://www.r-project.org>):** when we need to couple GAMA and R, we need to set properly the path toward this file.
- **GIS Coordinate Reference Systems (<http://spatialreference.org/ref/epsg/> for EPSG codes):** settings about CRS to use when loading or saving GIS files
 - **Let GAMA decide which CRS to use to project GIS data:** if true, GAMA will decide which CRS, based on input, should be used to project GIS data. Default is `true` (i.e. GAMA will always try to find the relevant CRS, and, if none can be found, will fall back one the one provided below)
 - **...or use the following CRS (EPSG code):** choose a CRS that will be applied to all GIS data when projected in the models. Please refer to <http://spatialreference.org/ref/epsg/> for a list of EPSG codes. If the option above is `false`, then the use of this CRS will be enforced in all models. Otherwise, GAMA will first try to find the most relevant CRS and then fall back on this one.
 - **When no .prj file or CRS is supplied, consider GIS data to be already projected in the CRS:** if true, GIS data that is not accompanied by a CRS information will be considered as projected using the above code.
 - **...or use the following CRS (EPSG code):** choose a CRS that will represent the default code for loading uninformed GIS data.
 - **When no CRS is provided, save the GIS data with the current CRS:** if true, saving GIS data will use the projected CRS unless a CRS is provided.
 - **...or use the following CRS (EPSG code):** otherwise, you might enter a CRS to use to save files.

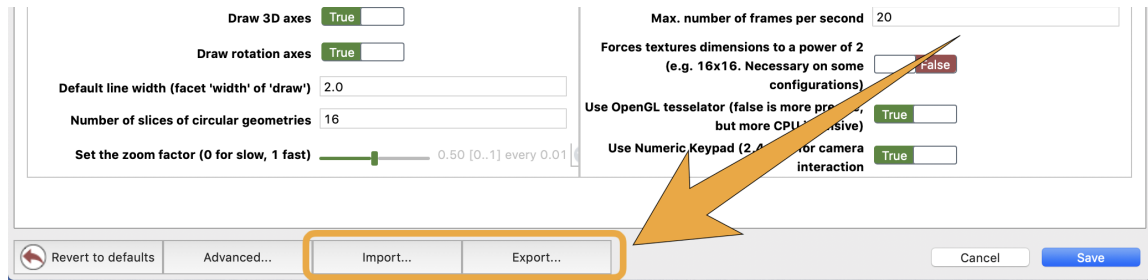


Figure 8.7: Import or export the preferences from/in a GAML model.

- **Management of dates:** some preferences for default values related to [the dates in GAMA](#).
 - Custom date pattern (<https://docs.oracle.com/javase/8/docs/api/java/time/>)
 - Default date pattern for writing dates (i.e. `string(date1)`)
 - Default starting date of models: set the default value of the [global variable `starting_date`](#).
 - Default time step of models: define the default duration of a simulation step, i.e. the value of the variable `step` (by default, it is set to 1s).

Manage preferences in GAML

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

When you export your preferences, the GAML file will look like the following code. It contains 2 experiments: one to display all the preferences in the console and the other one to set your preferences with the values written in the model.

```
model preferences

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

  //Display grid lines
}
```

```
write sample(gama.pref_chart_display_gridlines);

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

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

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

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

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

// ...
```

Advanced Preferences

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

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

Depending on what is installed, the second view that appears will contain a tree of options on the left and preference pages on the right. **Contrary to the first set of preferences, please note that these preferences will be saved in the current workspace**, which means that changing workspace will revert them to their default values. It is, however, possible to import them in the new workspace using of the wizards provided in the standard “Import...” command (see [here](#)).

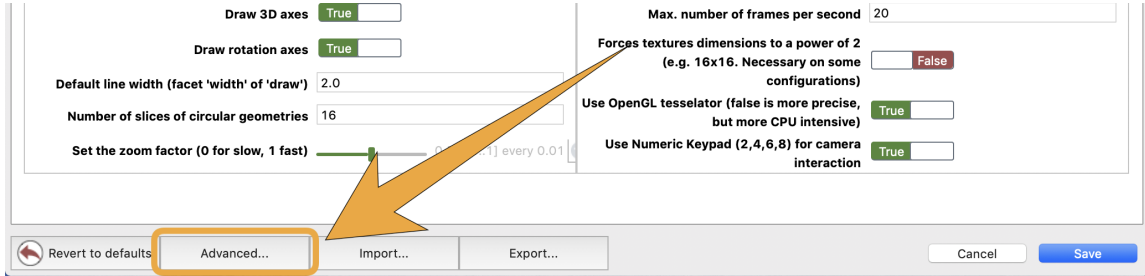


Figure 8.8: Open the advanced preferences.

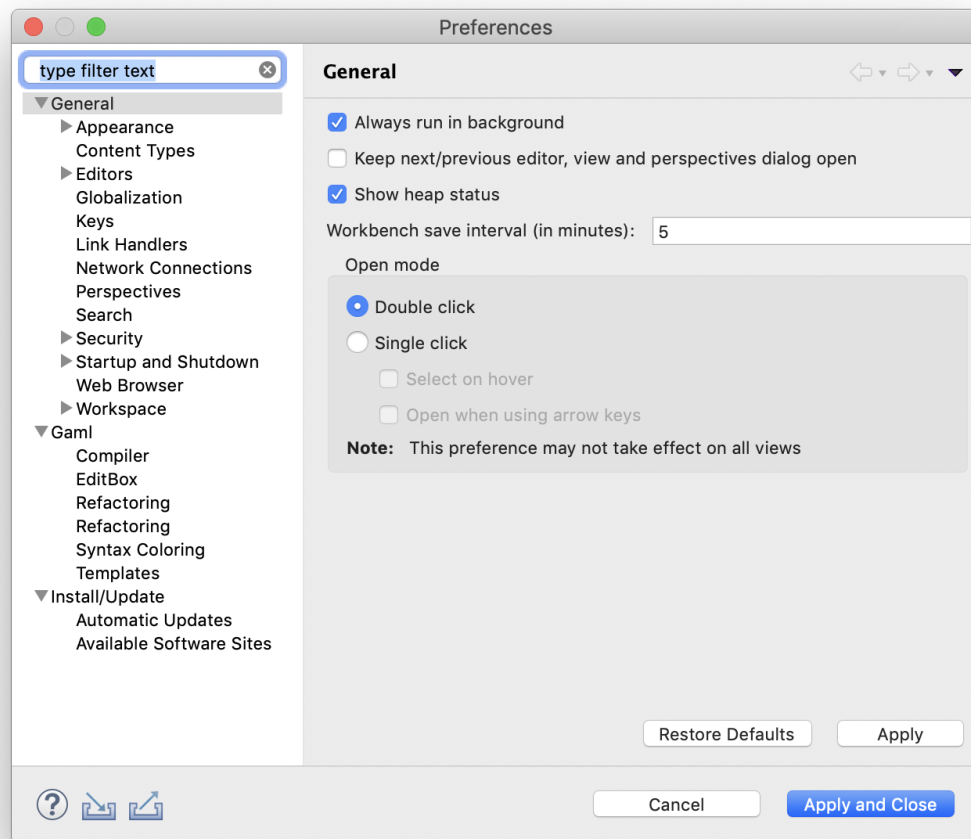


Figure 8.9: The advanced preferences available from the Preferences window.

Chapter 9

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

- [Troubleshooting](#)
 - [Table of contents](#)
 - [On Ubuntu \(& Linux Systems\)](#)
 - * [Workaround if OpenGL display crash GAMA](#)
 - [On macOS](#)
 - * [First launch of GAMA should be in GUI mode](#)
 - [Memory problems](#)
 - [Submitting an Issue](#)

On Ubuntu (& Linux Systems)

Workaround if OpenGL display crash GAMA

In case GAMA crashes whenever trying to display an OpenGL display or a Java2D, and you are running Ubuntu 21.10 (or earlier), it probably means that you're using **Wayland** as Display backend. You can fix it by running in a terminal `export GDK_BACKEND=x11` and launch GAMA from this same terminal. This workaround is described here: https://bugs.eclipse.org/bugs/show_bug.cgi?id=577515 and in [Issue 3373](#).

On macOS

First launch of GAMA should be in GUI mode

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

Memory problems

The most common causes of problems when running GAMA are memory problems. Depending on your activities, on the size of the models you are editing, on the size of the experiments you are running, etc., you have a chance to require more memory than what is currently allocated to GAMA. A typical GAMA installation will need between 2 and 4GB of memory to run “normally” and launch small models. Memory problems are easy to detect: in the bottom-right corner of its window, GAMA will always display the status of the current memory. The first number represents the memory currently used (in MB), the second (always larger) the memory currently allocated by the JVM. And the little trash icon allows to “garbage collect” the memory still used by agents that are not used anymore (if any). If GAMA appears to hang or crash and if you can see that the two numbers are very close, it means that the memory required by GAMA exceeds the memory allocated.

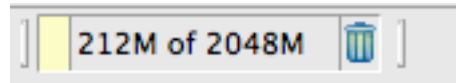


Figure 9.1: Memory bar status in GAMA.

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

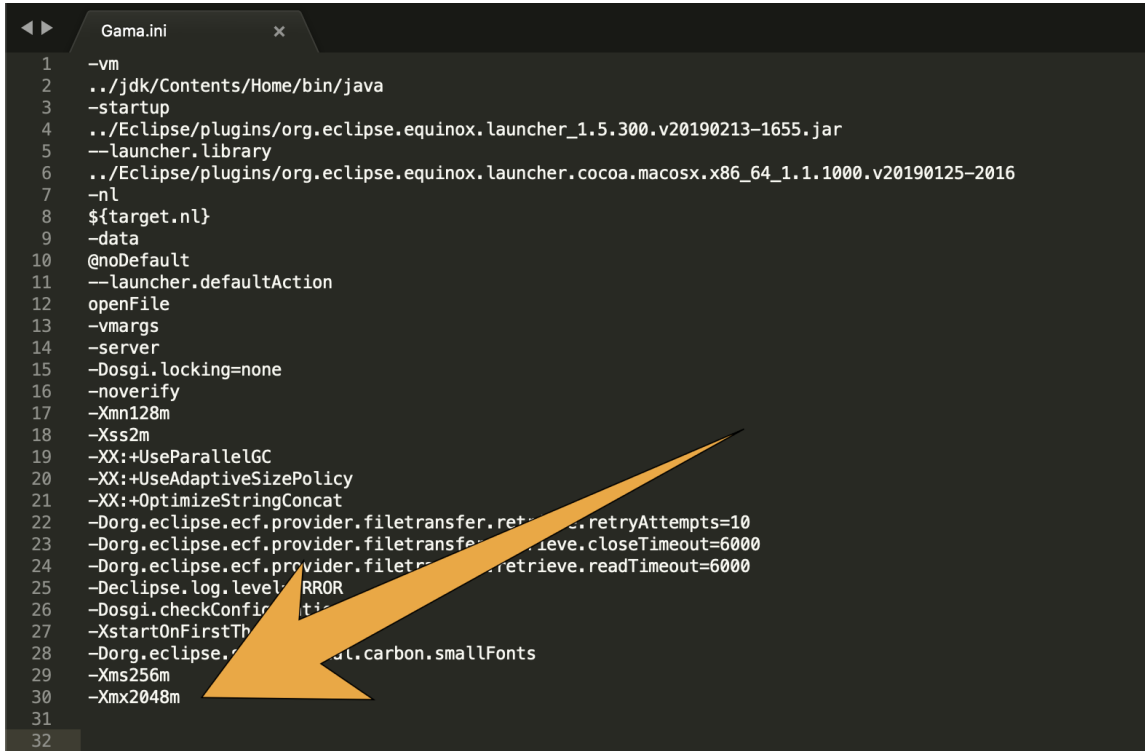
`-Xms` supplies the minimal amount of memory the JVM should allocate to GAMA, `-Xmx` the maximal amount. By changing these values (esp. the second one, of course, for example to 4096M, or 4g, or more!), saving the file and relaunching GAMA, you can probably solve your problem. Note that 32 bits versions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

Submitting an Issue

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

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

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



```
1  -vm
2  ../jdk/Contents/Home/bin/java
3  -startup
4  ../Eclipse/plugins/org.eclipse.equinox.launcher_1.5.300.v20190213-1655.jar
5  --launcher.library
6  ../Eclipse/plugins/org.eclipse.equinox.launcher.cocoa.macosx.x86_64_1.1.1000.v20190125-2016
7  -nl
8  ${target.nl}
9  -data
10 @noDefault
11 --launcher.defaultAction
12 openFile
13 -vmargs
14 -server
15 -Dosgi.locking=none
16 -noverify
17 -Xmn128m
18 -Xss2m
19 -XX:+UseParallelGC
20 -XX:+UseAdaptiveSizePolicy
21 -XX:+OptimizeStringConcat
22 -Dorg.eclipse.ecf.provider.filetransfer.retryAttempts=10
23 -Dorg.eclipse.ecf.provider.filetransfer.level.closeTimeout=6000
24 -Dorg.eclipse.ecf.provider.filetransfer.retrieve.readTimeout=6000
25 -Declipse.log.level=ERROR
26 -Dosgi.checkConfig=true
27 -XstartOnFirstThread
28 -Dorg.eclipse.equinox.preferences.internal.carbon.smallFonts
29 -Xms256m
30 -Xmx2048m
31
32
```

Figure 9.2: Gama.ini file: the place to allocate more memory to GAMA to deal with big projects.

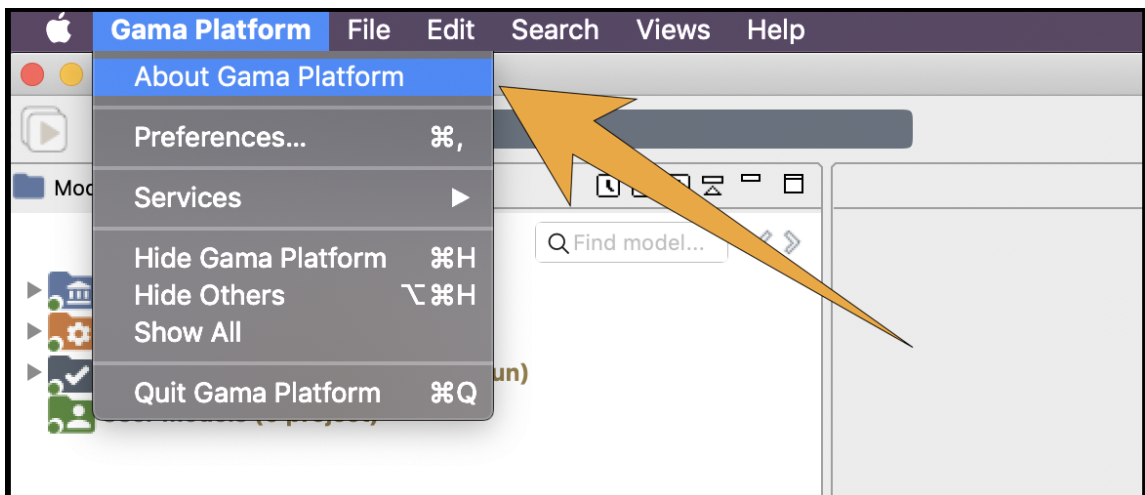


Figure 9.3: Open information about GAMA windows.

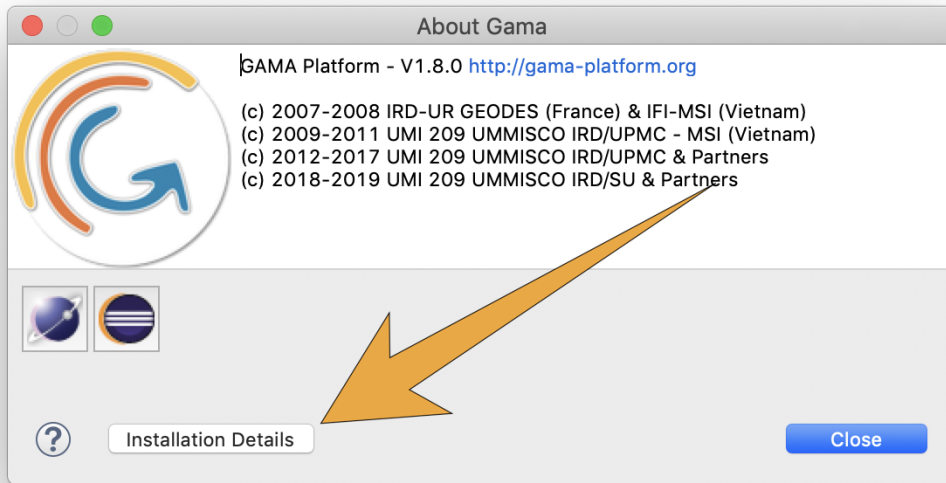


Figure 9.4: images/dialog_about_gama.png

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 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.

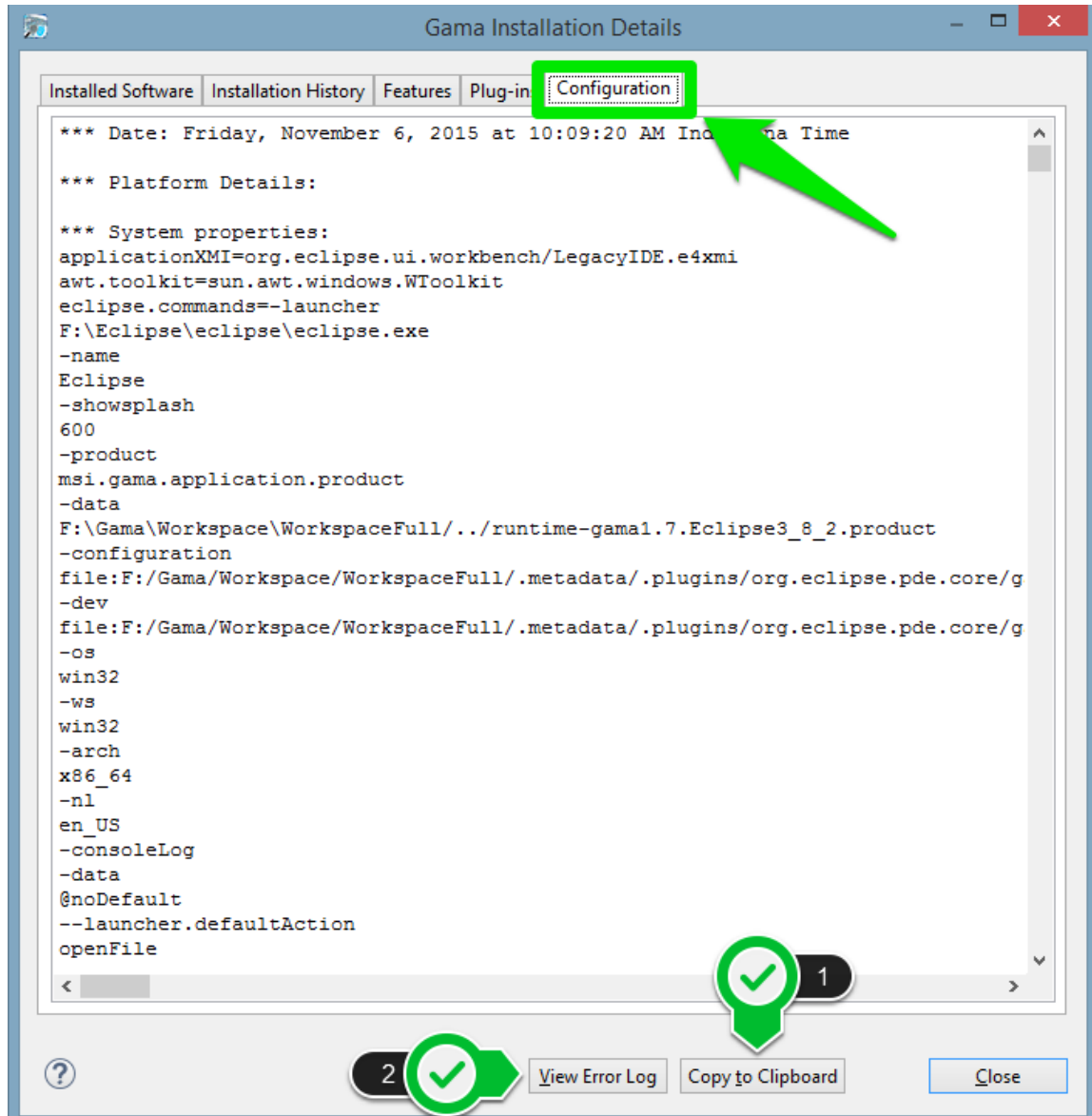


Figure 9.5: images/dialog_configuration.png

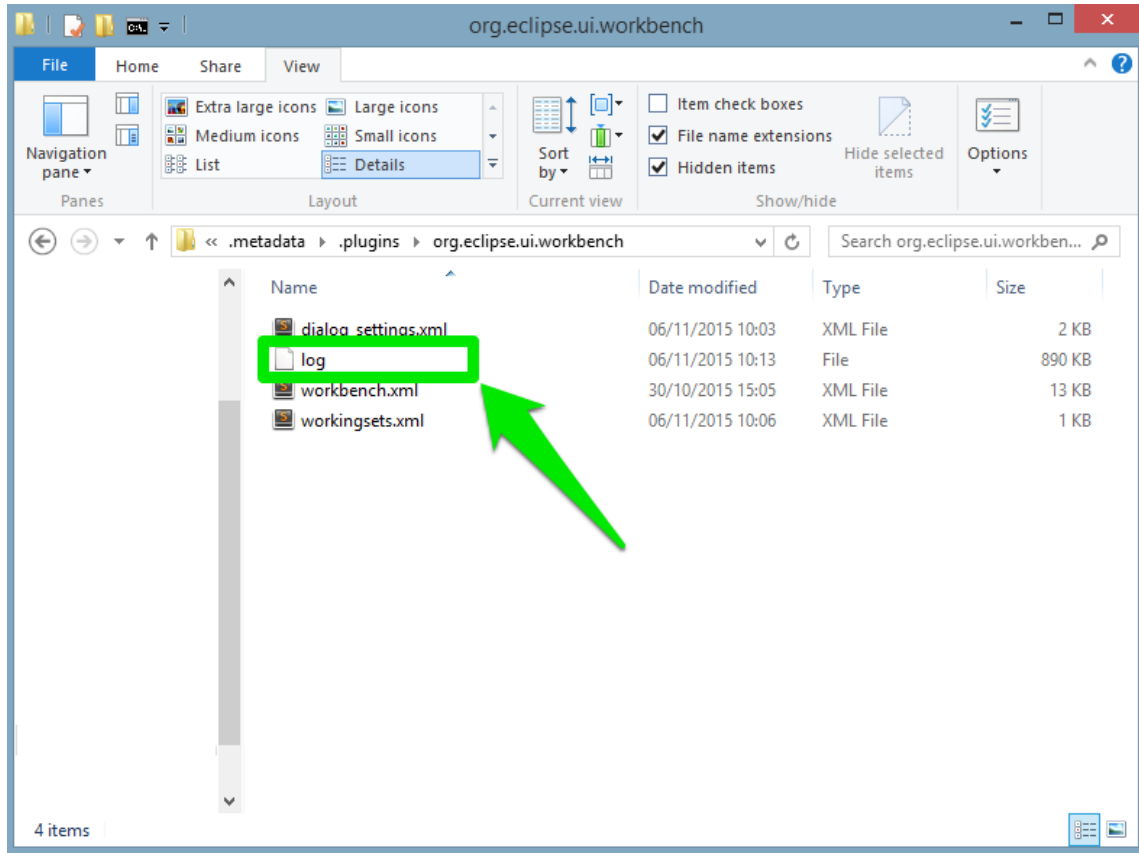


Figure 9.6: images/log_file.png

Part III

Learn GAML step by step

Chapter 10

Learn GAML Step by Step

This large progressive tutorial has been designed to help you to learn **GAML** (**G**Ama **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 11

Introduction

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

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

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

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

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

Table of contents

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

Lexical semantics of GAML

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

1. The role of GAML is to support modelers in writing **models**, which are specifications of **simulations** that can be executed and controlled during **experiments**, themselves specified by **experiment plans**.
2. The **agent-oriented** modeling paradigm means that everything “active” (entities of a model, systems, processes, activities, like simulations and experiments) can be represented in GAML as an **agent** (which can be thought of as a computational component owning its own data and executing its own behavior, alone or in interaction with other agents).
3. Like in the object-oriented paradigm, where the notion of *class* is used to supply a specification for *objects*, agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they know*), **actions** (*what they can do*), **behaviors** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology** (*how they are connected*) or **schedule** (*in which order and when they should execute*).

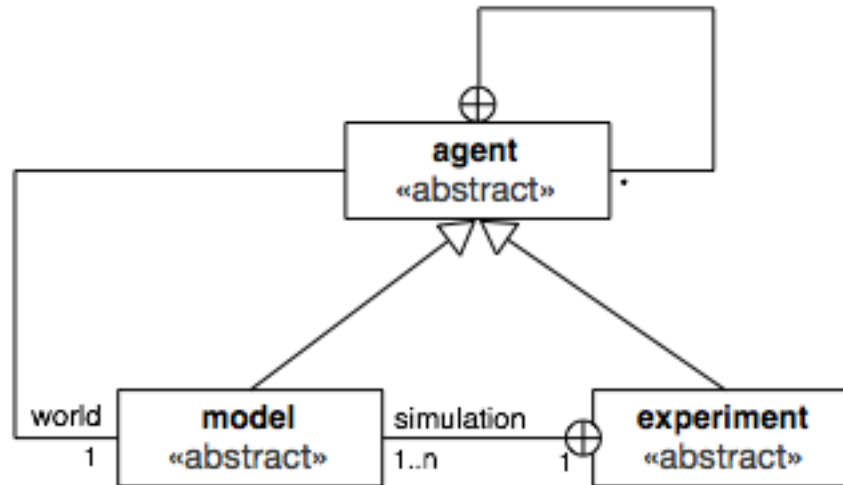


Figure 11.1: GAML meta-model.

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

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

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

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

Translation into a concrete syntax

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

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

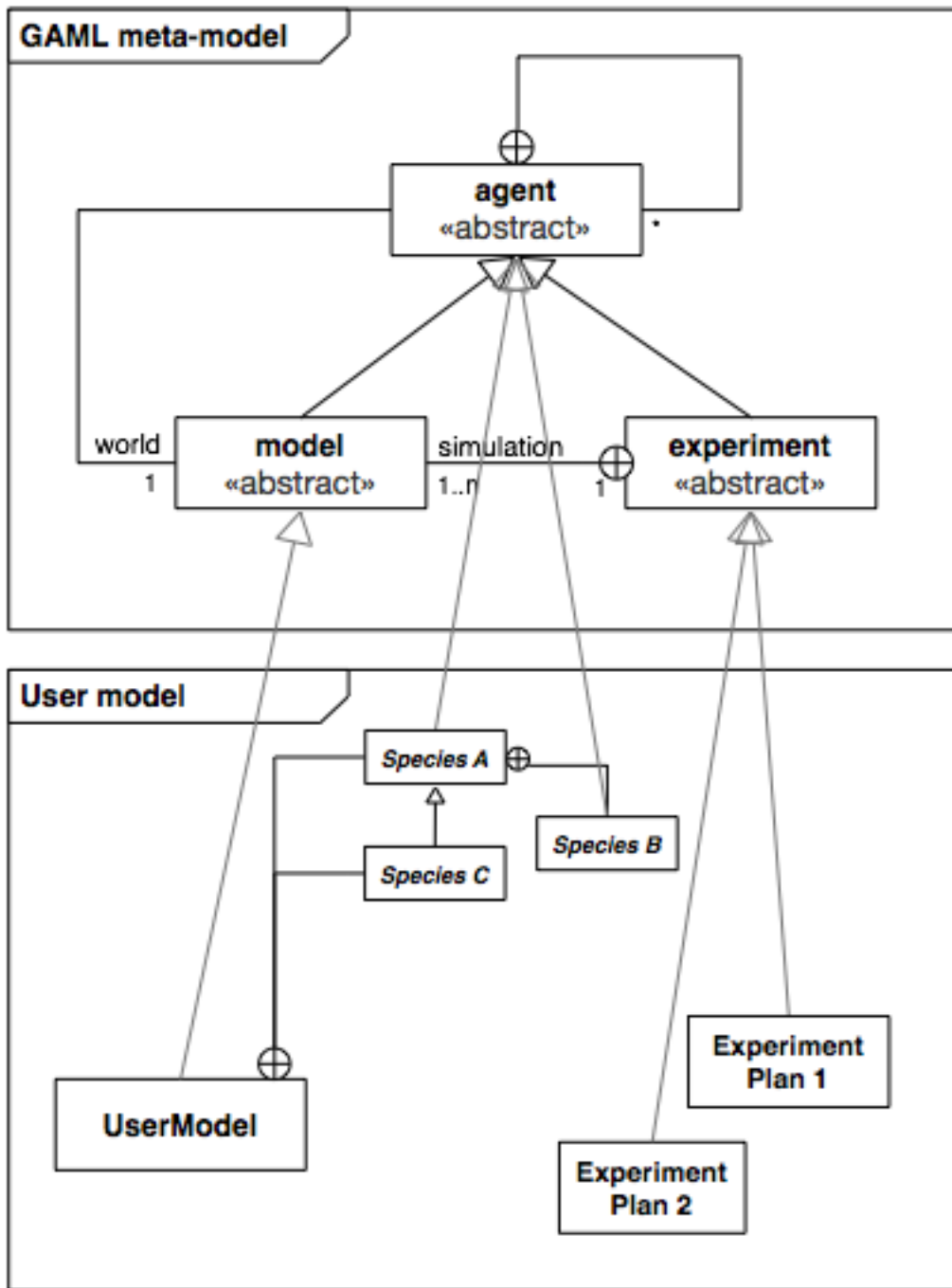


Figure 11.2: Instanciation of the GAML meta-model in a User model.

attributes, actions, behaviors, aspects, variables, parameters and outputs of experiments.

7. **imperative statements** that execute something or control the flow of execution of **actions, behaviors and aspects** are called **commands**.
8. A **species** declaration (**global, species** or **grid** keywords) can only include 6 types of declarative statements : **attributes, actions, behaviors, aspects, equations** and (nested) **species**. In addition, **experiment** species allow to declare **parameters, outputs** and batch **methods**.

Vocabulary correspondence with the object-oriented paradigm as in Java

GAML	Java
species	class
micro-species	nested class
parent species	superclass
child species	subclass
model	program
experiment	(main) class
agent	object
attribute	member
action	method
behavior	collection of methods
aspect	collection of methods, mixed with the behavior
skill	interface (on steroids)
statement	statement
type	type
parametric type	generics

Vocabulary correspondence with the agent-based paradigm as in NetLogo

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

Chapter 12

Manipulate basic species

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

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

Chapter 13

The global species

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

Index

- [Declaration](#)
- [Environment Size](#)
- [Built-in Attributes](#)
- [Built-in Actions](#)
- [The init statement](#)

Declaration

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

```
global {
```

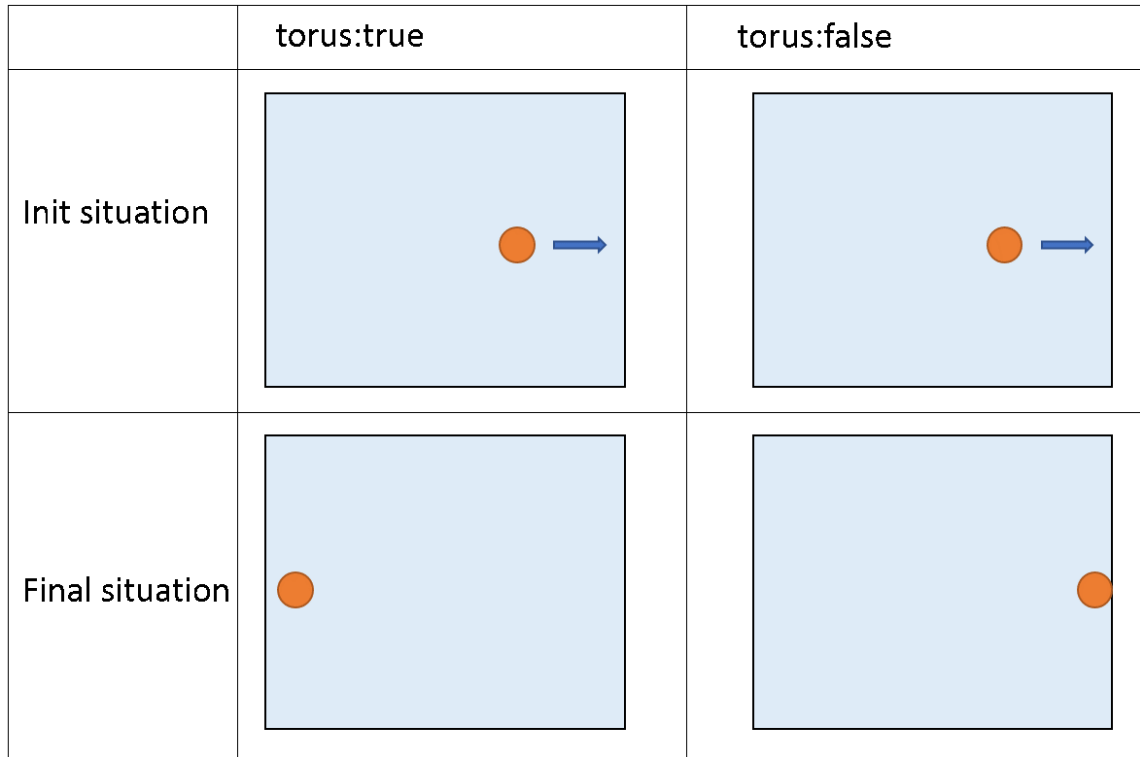


Figure 13.1: images/torus.png

```

// 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 on the other side. If it's not, the agents won't be able to go out of the environment). By default, the environment is not a torus.

```

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 specified, it will be meter by default.

Built-in attributes

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

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

world

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

experiment

- contains the `experiment` agent that has created this simulation agent.

cycle

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

To learn more about time, please read the [recipe about dates](#).

step

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

To learn more about time, please read the [recipe about dates](#).

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

time

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

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

To learn more about time, please read the [recipe about dates](#).

starting_date and current_date

- date, represent the starting date (resp. the current date) of the simulation. The `current_date` is updated from the `starting_date` by the value `step` at each simulation step.

To learn more about time, please read the [recipe about dates](#).

duration

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

total_duration

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

average_duration

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

machine_time

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

seed

- float, the seed of the random number generator. It will influence the set of random numbers that will be generated all over the simulation. 2 simulations of a model with the same parameters' values should behave identically when the seed is set to the same value. If it is not redefined by the modeler, it will be chosen randomly.

agents

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

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

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

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

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

Built-in Actions

The global species is provided with two specific actions.

pause

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

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

die

- stops the simulation (in fact it kills the simulation).

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

Other built-in actions are defined for the model species, just as in [any other regular species](#).

The init statement

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

Chapter 14

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 15

Defining GUI Experiment

When you execute your simulation, you will often need to display some information. For each simulation, you can define some inputs, outputs and behaviors:

- The inputs will be composed of parameters manipulated by the user for each simulation.
- The behaviors will be used to define behavior executed at each step of the **experiment**.
- The outputs will be composed of displays, monitors. They will be defined inside the scope `output`. The definition of their layout can also be set with the `layout` statement.

```
experiment exp_name type: gui {  
  [input]  
  [behaviors]  
  output {  
    layout [layout_option]  
    [display statements]  
    [monitor statements]  
  }  
}
```

Types of experiments

You can define four types of experiments (through the facet `type`):

- **gui** experiments (the default type) are used to play an experiment and displays its outputs. It is also used when the user wants to interact with the simulations.
- **batch** experiments are used to play an experiment several times (usually with other input values), used for model exploration. We will [come back to this notion a bit further in the tutorial](#).
- **test** experiments are used to [write unit tests](#) on a model (used to ensure its quality).
- **memorize** experiments are GUI experiments in which [the simulation state is kept in memory and the user can backtrack to any previous step](#).

Experiment attributes

Inside experiment scope, you can access to some built-in attributes which can be useful, such as `minimum_cycle_duration`, to force the duration of one cycle.

```
experiment my_experiment type: gui {  
  float minimum_cycle_duration <- 2.0#minute;  
}
```

In addition, the attributes `simulations` (resp. ‘simulation’) contain the list of all the simulation agents that are running in the current experiment (resp. a single simulation, the last element of the simulation list).

Experiment facets

Finally, in the case of a GUI experiment, the facet `autorun` and `benchmark` can be used such as:

```
experiment name type: gui autorun: true benchmark: true { }
```

When `autorun` is set to `true` the launch of the experiment will be followed automatically by its run. When `benchmark` is set to `true`, GAMA records the number of invocations and running time of the statements and operators of the simulations launched in this experiment. The results are automatically saved in a csv file in a folder called ‘benchmarks’ when the experiment is closed.

Other built-ins are available, to learn more about, go to the page [experiment built-in](#).

Defining displays layout

The `layout` can be added to `output` to specify the layout of the various displays defined below (e.g. `#nonce`, `#split`, `#stack`, `#vertical` or `#horizontal`). It will also define which elements of the interface are displayed: `parameters`, `navigator`, `editors`, `consoles`, `toolbars`, `tray`, or `tabs` facets (expecting a boolean value).

Defining elements of the GUI experiment

In this part, we will focus on the **gui experiments**. We will start with learning how to **define input parameters**, then we will study the outputs, such as **displays**, **monitors and inspectors**, and **export files**. We will finish this part with how to define **user commands**.

Chapter 16

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 17

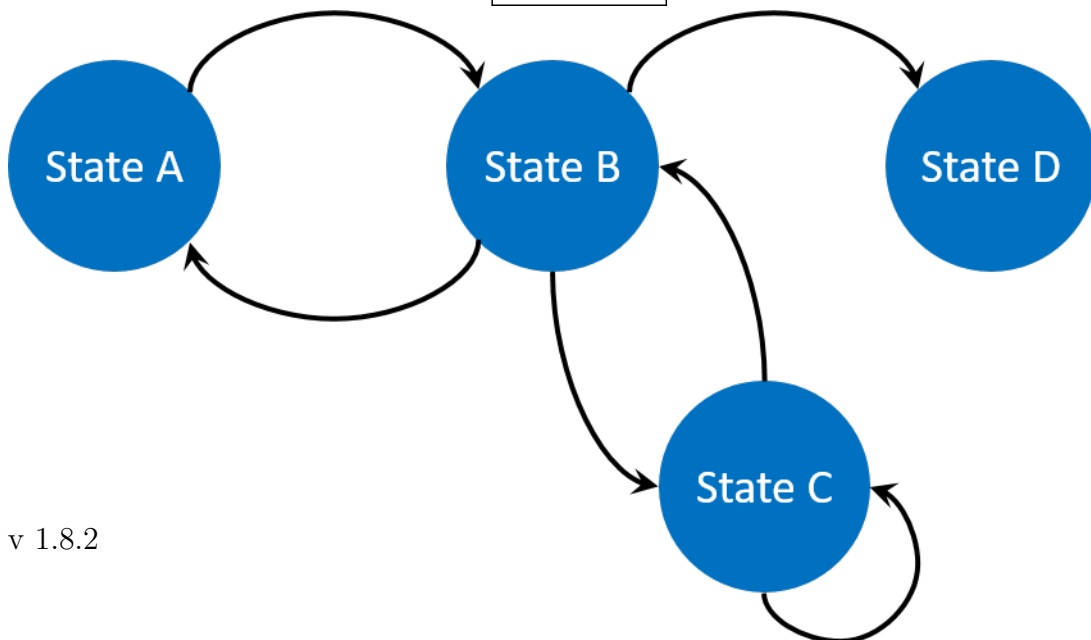
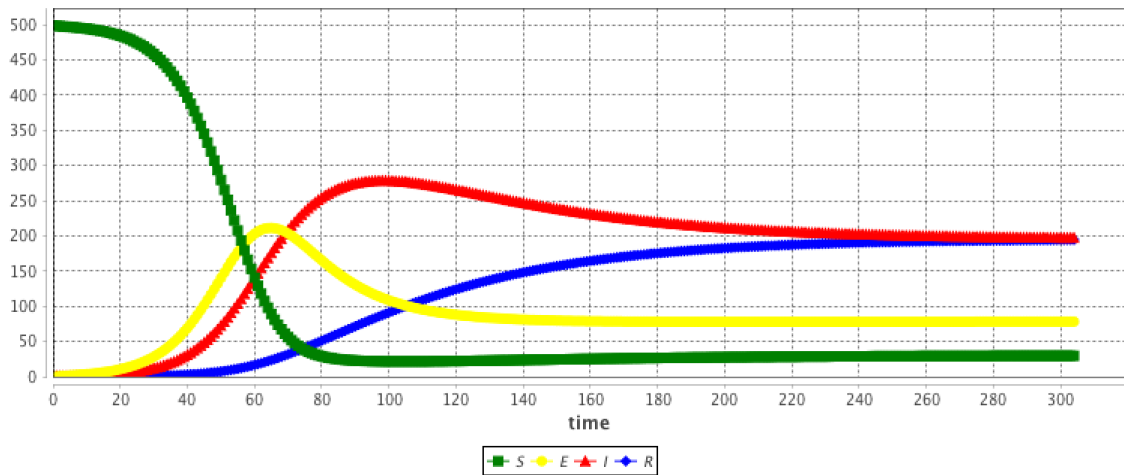
Optimizing Models

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 18

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 another approach, a math based approach, through use of [equations](#).

Part IV

Recipes

Chapter 19

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 20

Manipulate OSM Datas

This section will be presented as a quick tutorial, showing how to proceed to manipulate OSM (Open street map) data, clean them and load them into GAMA. We will use the software [QGIS](#) to change the attributes of the OSM file.

Note that GAMA can read and import OpenStreetMap data natively and create agents from them. An example model is provided in the Model Library (Data Importation / OSM File Import.gaml). In this case, you will have to write a model to import, select data from OpenStreetMap before creating agents and then could export them into shapefiles, much easier to use in GAMA.

From the website openstreetmap.org, we will choose a place (in this example, we will take a neighborhood in New York City). Directly from the website, you can export the chosen area in the osm format.

We have now to manipulate the attributes for the exported osm file. Several softwares can be used, but we will focus on [QGIS](#), which is totally free and provides a lot of possibilities in term of manipulation of data.

Once you have installed correctly QGIS, launch QGIS Desktop, and start to import the topology from the osm file.

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.

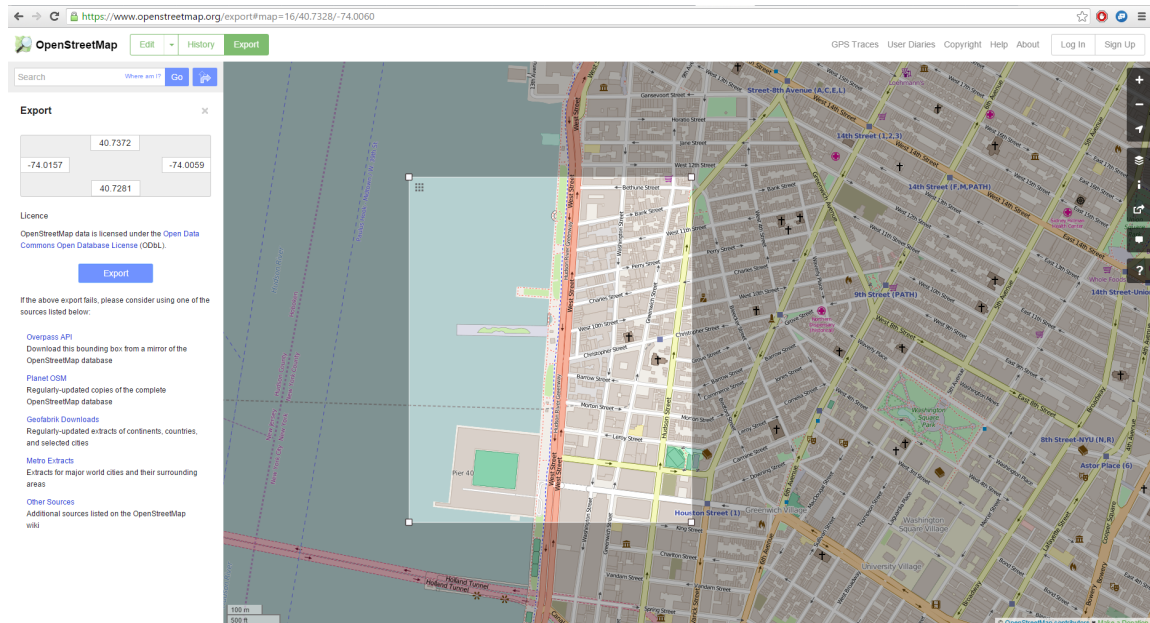


Figure 20.1: OpenStreetMap website to select a place.

A message indicates that the export was successful, and you have now a new layer created.

We will now manipulate the attributes of your datafile. Right-click on the layer, and select “Open Attribute Table”.

The table of attribute appears. Select the little pencil on the top-left corner of the window to modify the table.

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 finish, 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
```

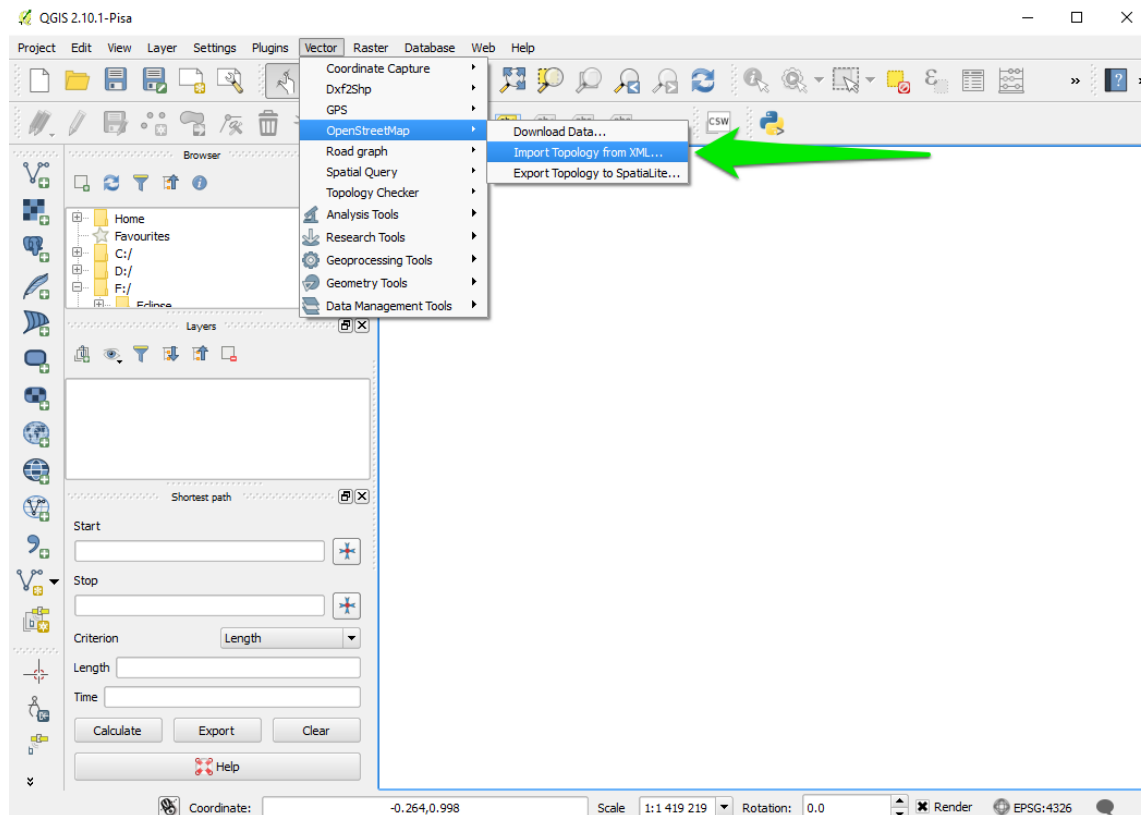


Figure 20.2: Import OpenStreetMap data into QGIS.

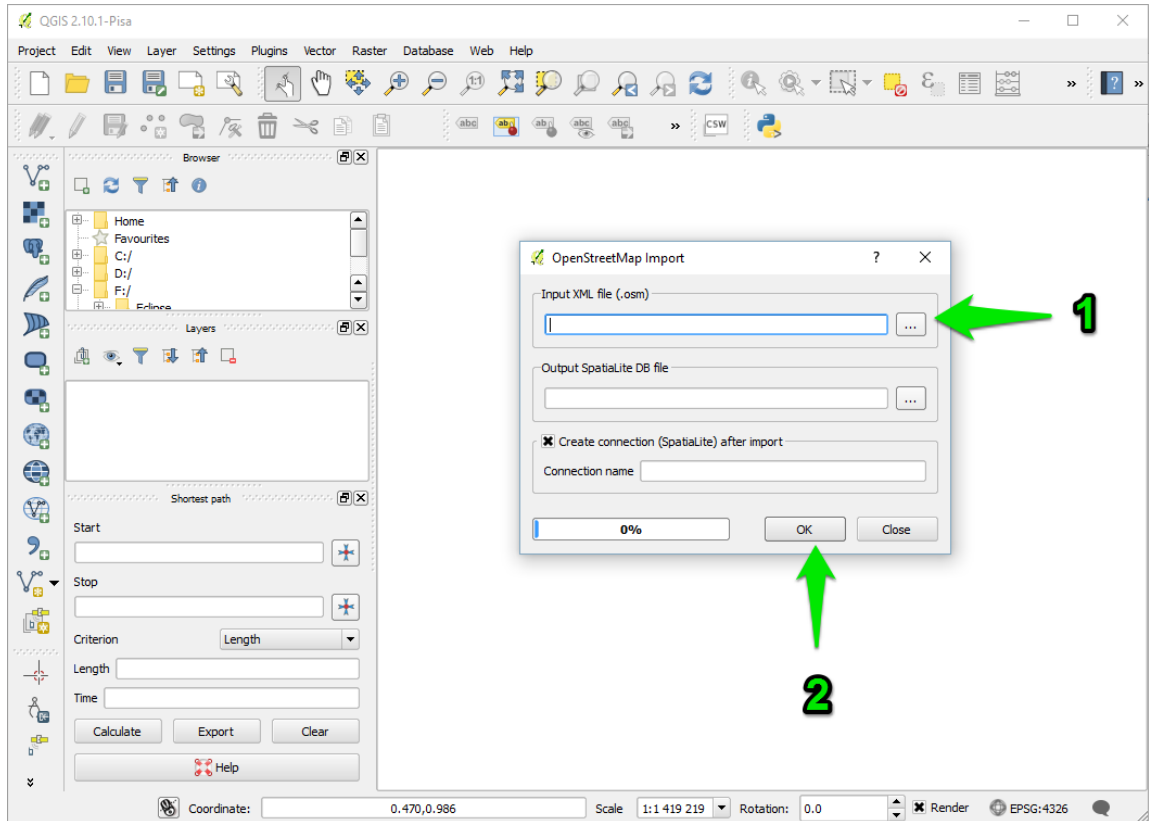


Figure 20.3: Import OpenStreetMap data into QGIS 2.

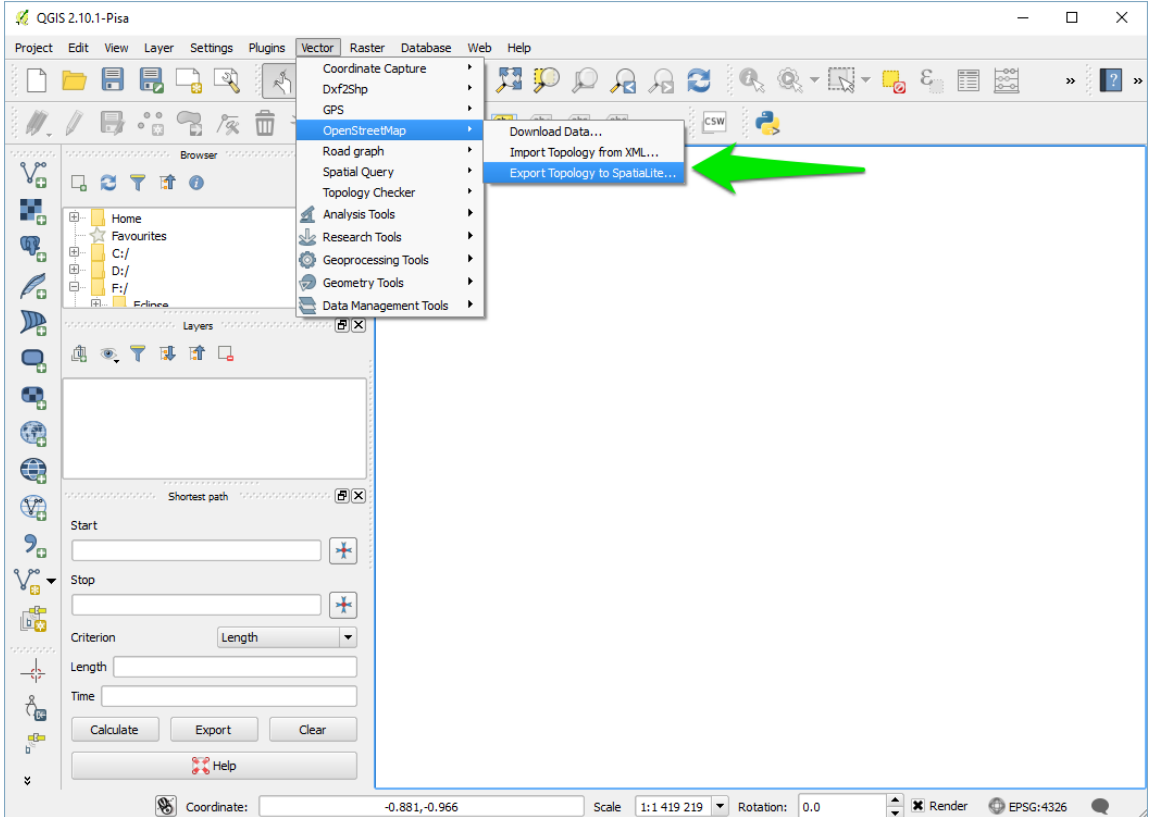


Figure 20.4: Export the data to SpatiaLite (through QGIS).

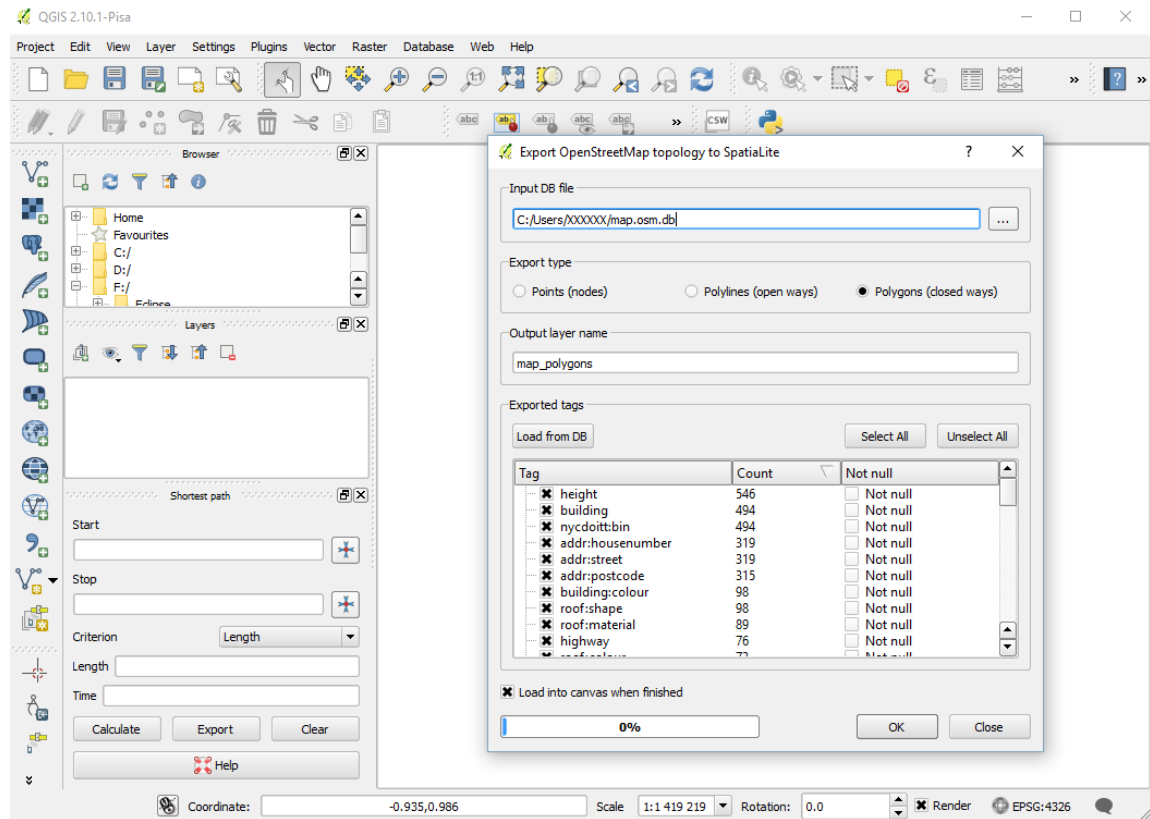


Figure 20.5: Select the OSM attribute before importation.

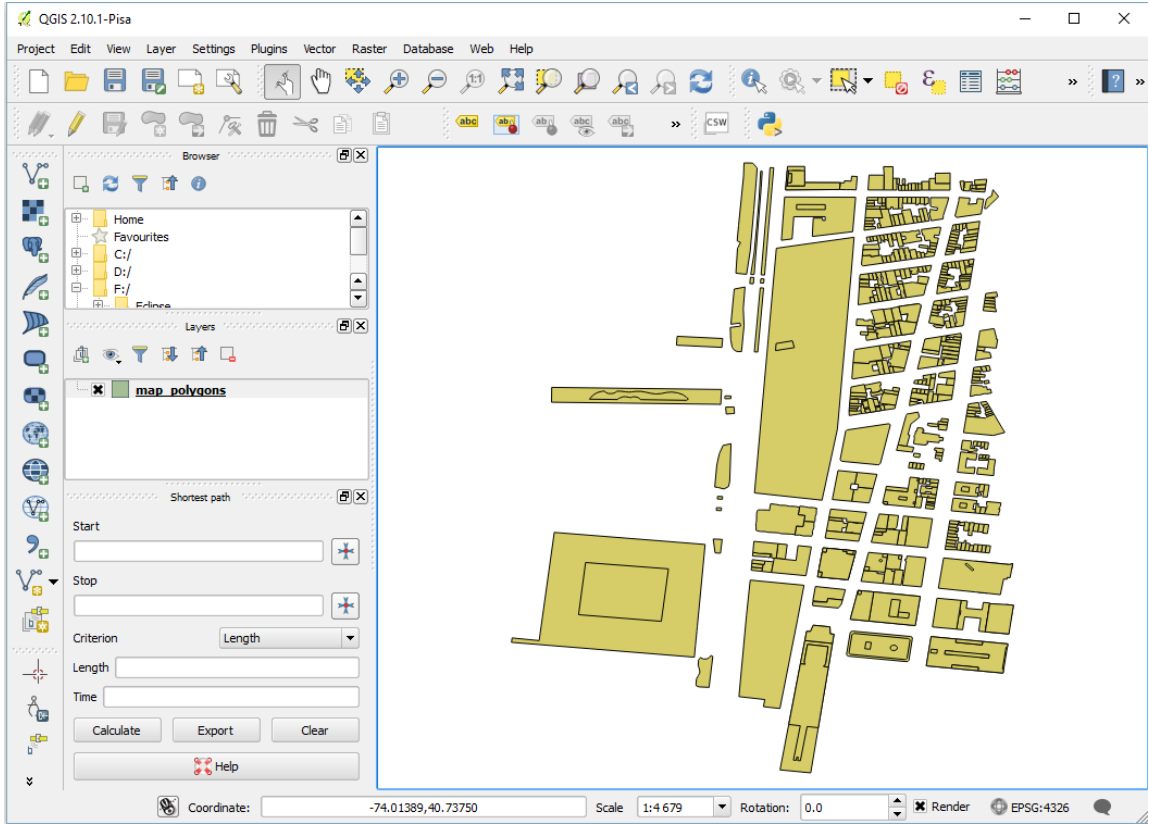


Figure 20.6: Display of OSM data imported in QGIS.

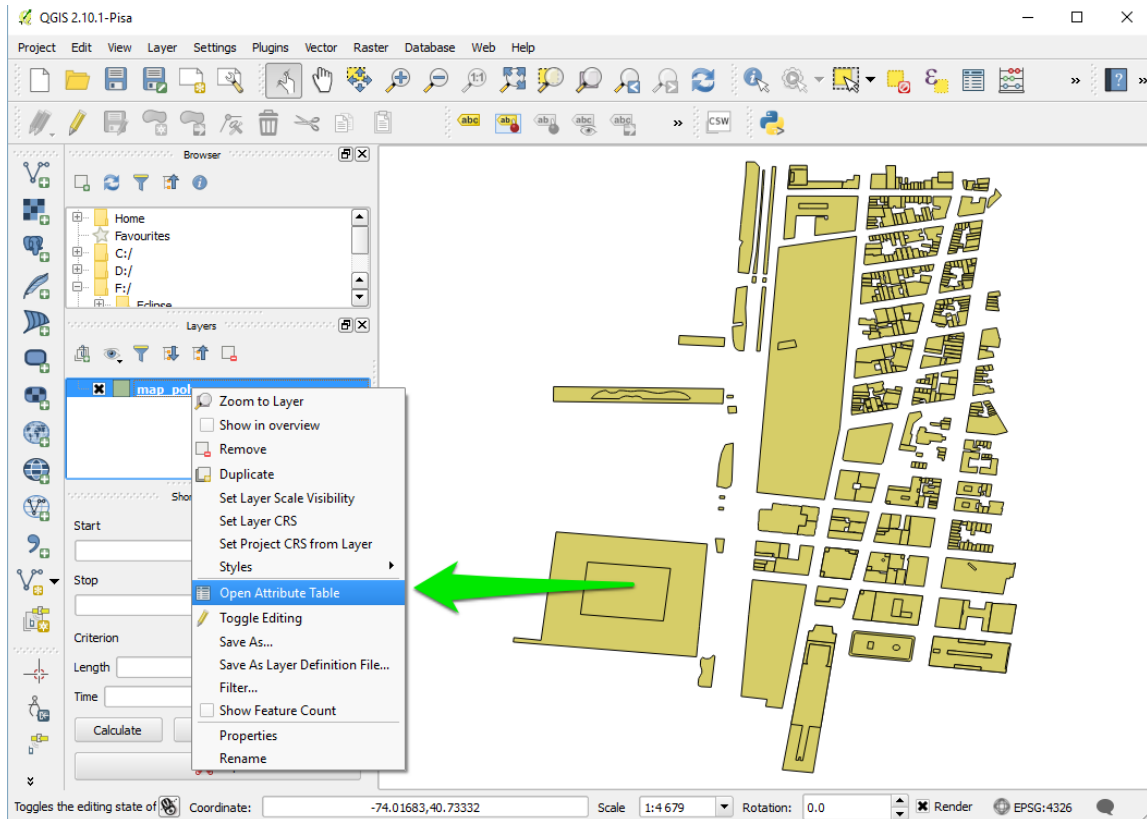


Figure 20.7: Open attribute table to display all the agents.

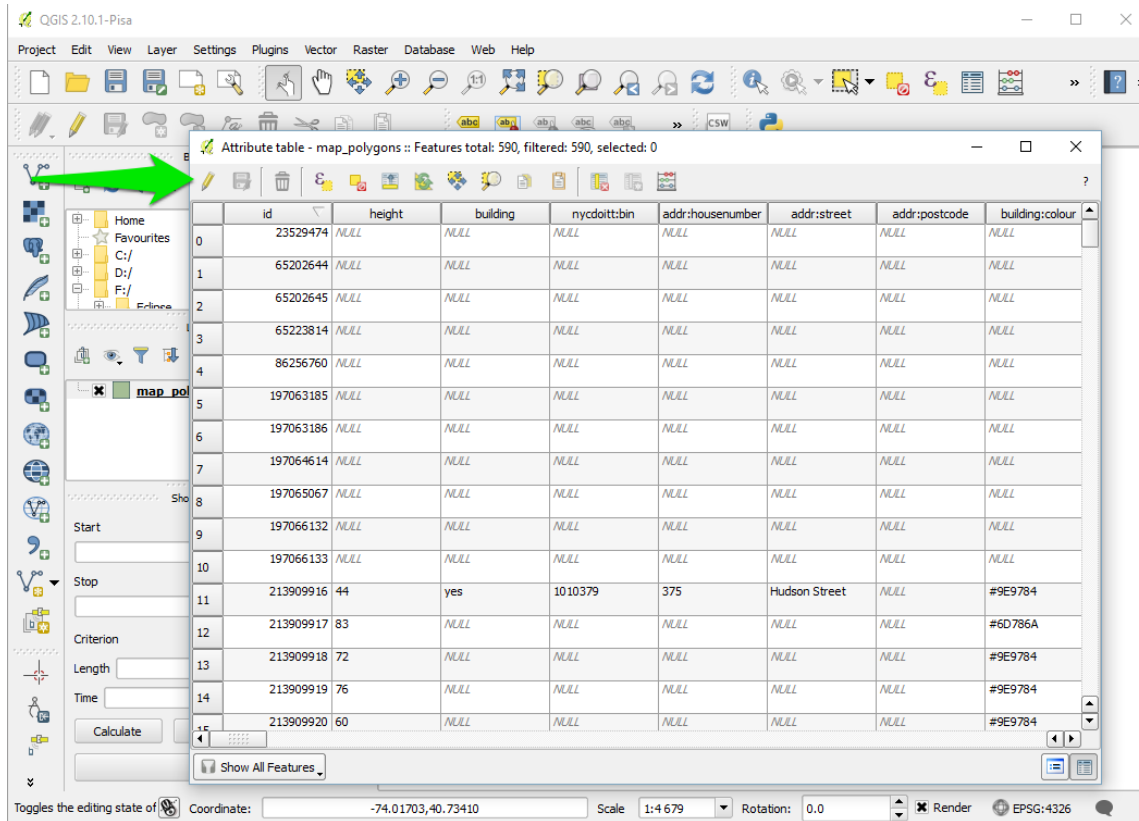


Figure 20.8: Attribute tables.

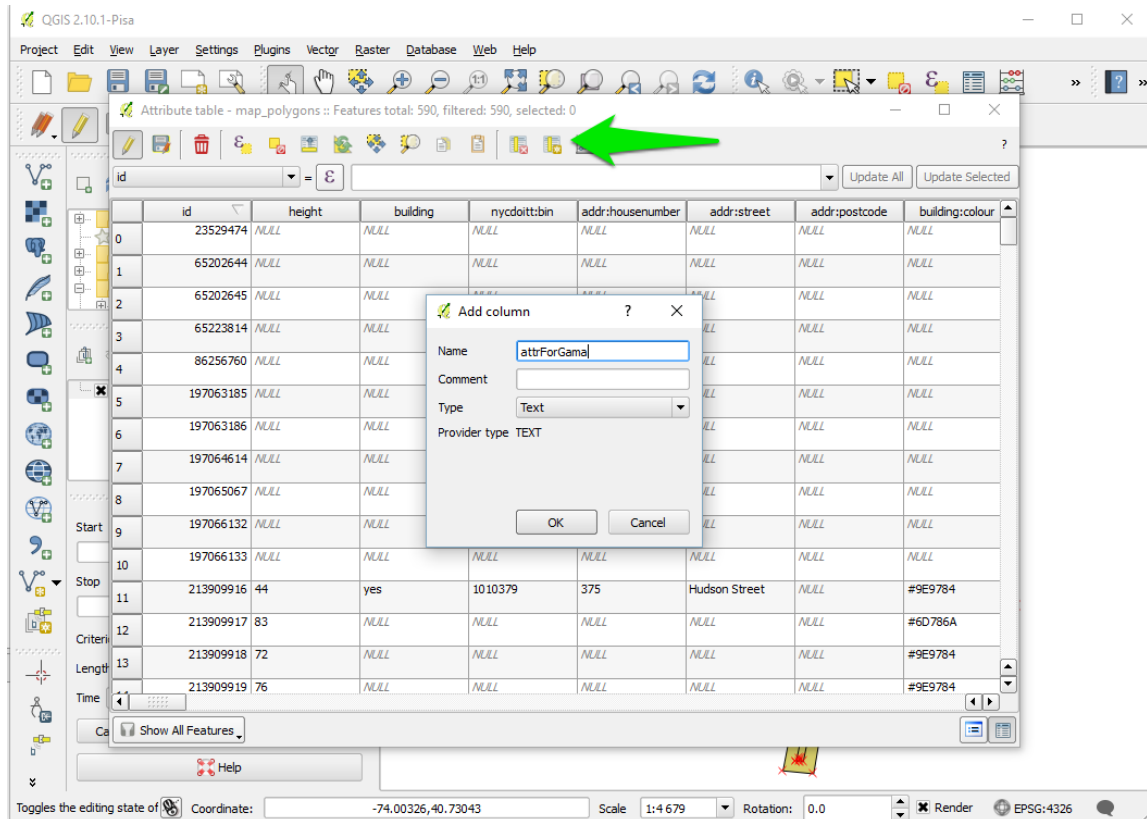


Figure 20.9: Add an attribute to the attribute table.

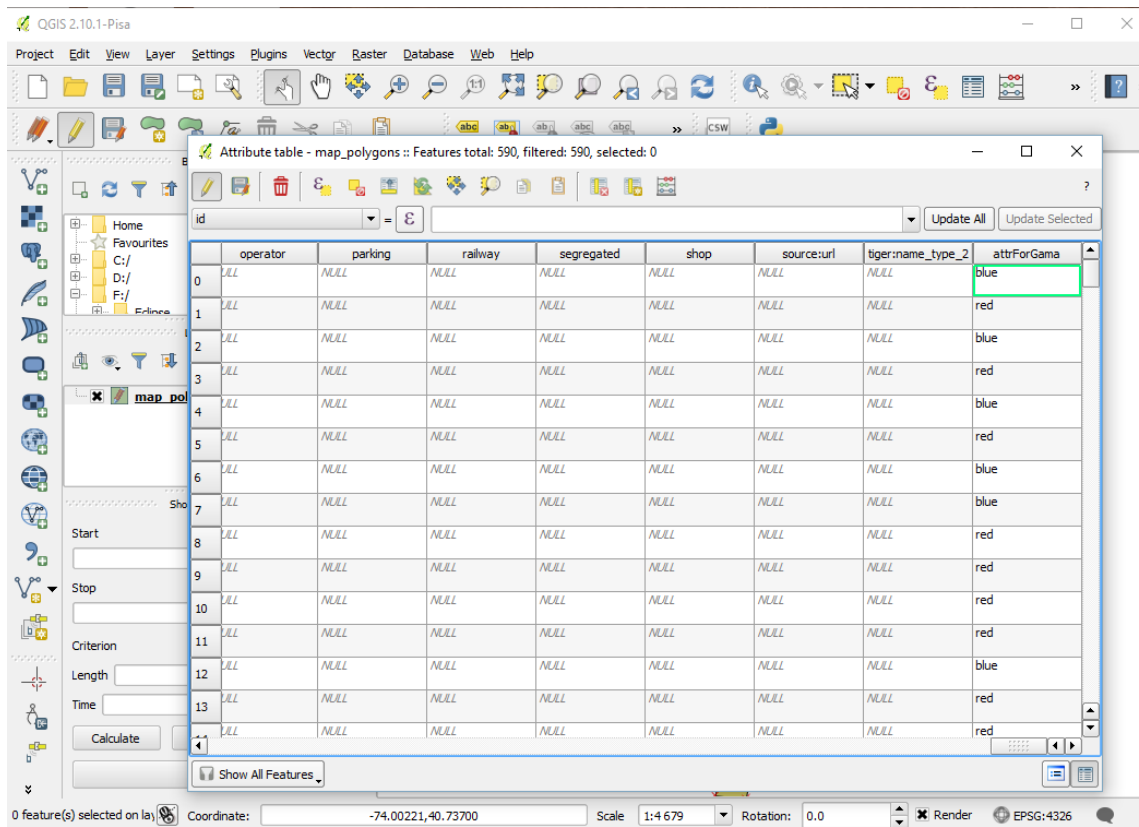


Figure 20.10: Fill the new attribute with values.

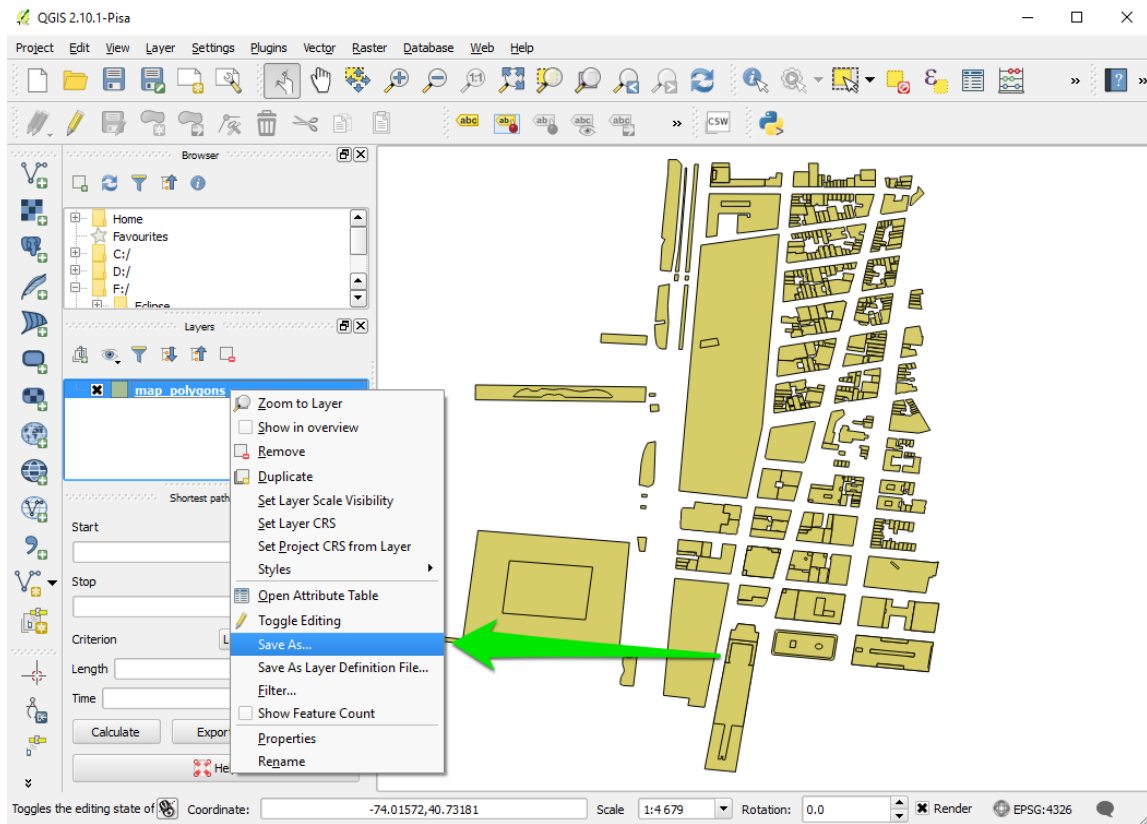


Figure 20.11: Open the save data window.

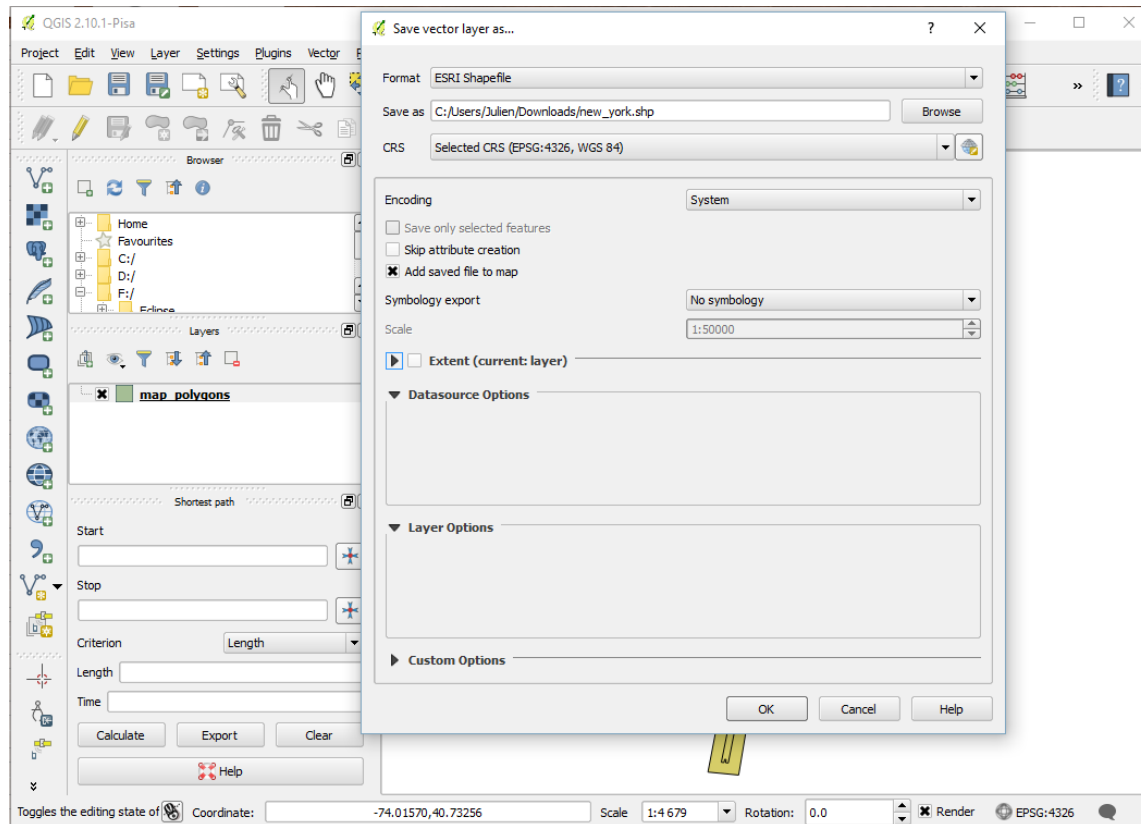


Figure 20.12: Save the data in a shapefile.

```

global {
  // Global variables related to the Management units
  file shapeFile <- file('../includes/new_york.shp');

  //definition of the environment size from the shapefile.
  //Note that is possible to define it from several files by
  using: geometry shape <- envelope(envelope(file1) +
  envelope(file2) + ...);
  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 the

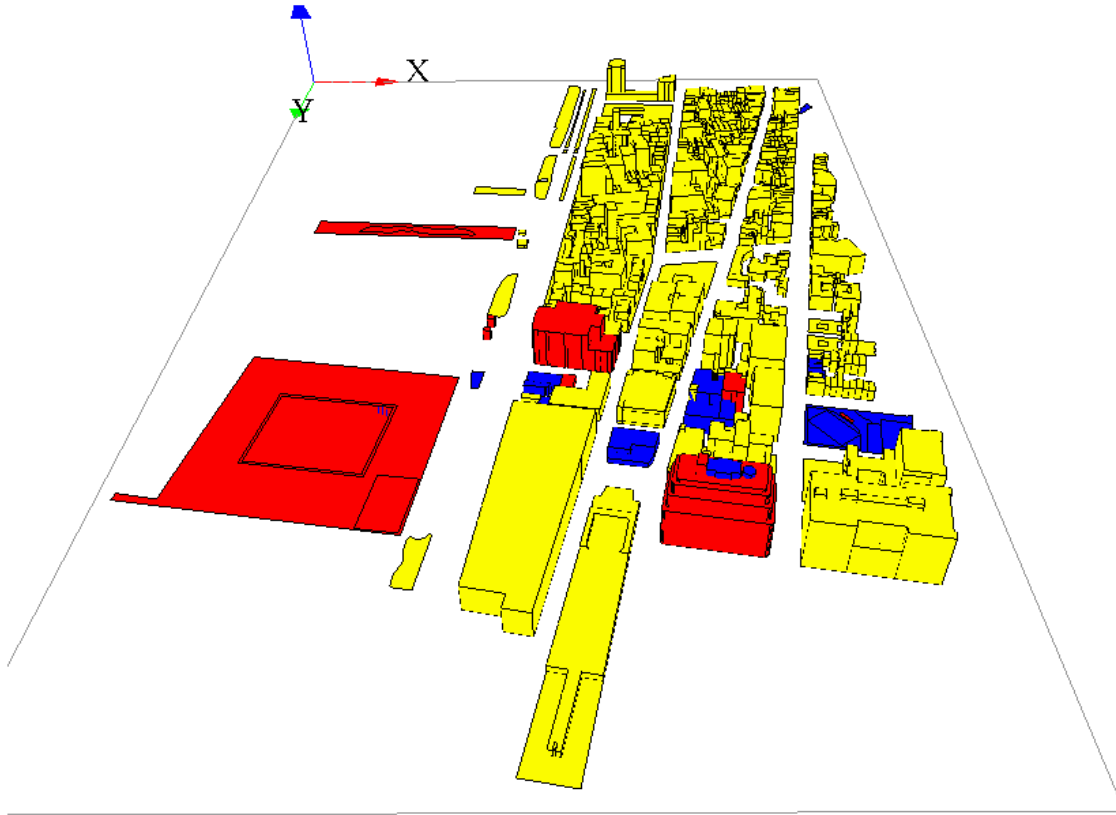


Figure 20.13: images/manipulate_OSM_file_13.png

value of the attribute “attrForGama”, and an elevation regarding the value of the attribute “height”.

Chapter 21

Implementing diffusion

GAMA provides you the possibility to represent and simulate the diffusion of a variable through a grid topology.

Index

- Diffuse statement
- Diffusion with matrix
 - Diffusion matrix
 - Gradient matrix
 - Compute multiple propagations at the same step
 - Executing several diffusion matrix
- Diffusion with parameters
- Computation methods
 - Convolution
 - Dot Product
- Use mask
 - Generalities
 - Tips
- Pseudo code

Diffuse statement

The statement to use for the diffusion is `diffuse`. It has to be used in a `grid` species. The `diffuse` uses the following facets:

- `var` (an identifier), (omissible) : the variable to be diffused
- `on` (any type in [container, species]): the list of agents (in general cells of a grid), on which the diffusion will occur
- `avoid_mask` (boolean): if true, the value will not be diffused in the masked cells, but will be restituted to the neighboring cells, multiplied by the variation value (no signal loss). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (loss of signal). (default value : false)
- `cycle_length` (int): the number of diffusion operation applied in one simulation step
- `mask` (matrix): a matrix masking the diffusion (matrix created from an image for example). The cells corresponding to the values smaller than “-1” in the mask matrix will not diffuse, and the other will diffuse.
- `matrix` (matrix): the diffusion matrix (“kernel” or “filter” in image processing). Can have any size, as long as dimensions are odd values.
- `method` (an identifier), takes values in: {convolution, dot_product}: the diffusion method
- `min_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 propagations of the same signal occurring at once from different places. If propagation equals ‘diffusion’, the intensity of a signal is shared between its neighbors with respect to ‘proportion’, ‘variation’ and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is $S' = (S / N / \text{proportion}) - \text{variation}$. The intensity of S is then diminished by $S * \text{proportion}$ on P. In diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals ‘gradient’, the original intensity is not modified, and each neighbor receives the intensity: $S / \text{proportion} - \text{variation}$. If multiple propagations occur at once, only the maximum intensity is kept on each place. If ‘propagation’ is not defined, it is assumed that it is equal to

‘diffusion’.

- **proportion** (float): a diffusion rate
- **radius** (int): a diffusion radius (in number of cells from the center)
- **variation** (float): an absolute value to decrease at each neighbor

To write a diffusion, you first have to declare a grid and declare a special attribute for the diffusion. You will then have to write the **diffuse** statement in another scope (such as the **global** scope for instance), which will permit the values to be diffused at each step. There, you will specify which variable you want to diffuse (through the **var** facet), on which species or list of agents you want the diffusion (through the **on** facet), and how you want this value to be diffused (through all the other facets, we will see how it works **with matrix** and **with special parameters** just after).

Here is the template of code we will use for the next following part of this page:

```
global {
  int size <- 64; // the size has to be a power of 2.
  cells selected_cells;

  // Initialize the emitter cell as the cell at the center
  of the world
  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 **odd values**. The most often, diffusion matrices are square matrices, but you can also declare a rectangular matrix.

Example of matrix:

```

matrix<float> mat_diff <- matrix([
  [1/9,1/9,1/9],
  [1/9,1/9,1/9],
  [1/9,1/9,1/9]]);

```

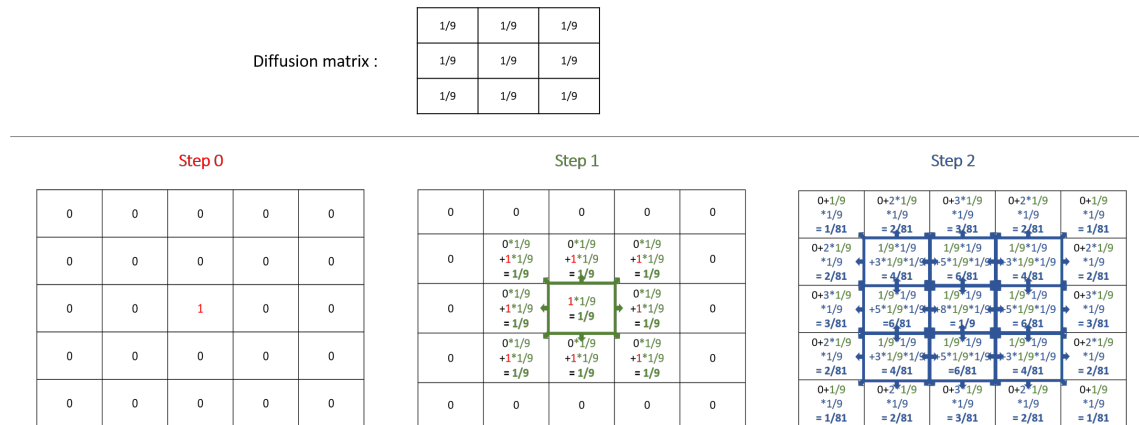


Figure 21.1: Illustration of the computation under a diffusion propagation.

In the `diffuse` statement, you then have to specify the matrix of diffusion you want in the facet `matrix`.

```
diffuse var: phero on: cells matrix:mat_diff;
```

Using the facet `propagation`, you can specify if you want the value to be propagated as a *diffusion* or as a *gradient*.

Diffusion matrix

A *diffusion* (the default value of the facet `propagation`) will spread the values to the neighbors' cells according to the diffusion matrix, and all those values will be added together, as it is the case in the following example:

Note that the sum of all the values diffused at the next step is equal to the sum of the values that will be diffused multiply by the sum of the values of the diffusion matrix. That means that if the sum of the values of your diffusion matrix is larger than 1, the values will increase exponentially at each step. The sum of the value of a diffusion matrix is usually equal to 1.

Here are some matrix examples you can use, played with the template model:

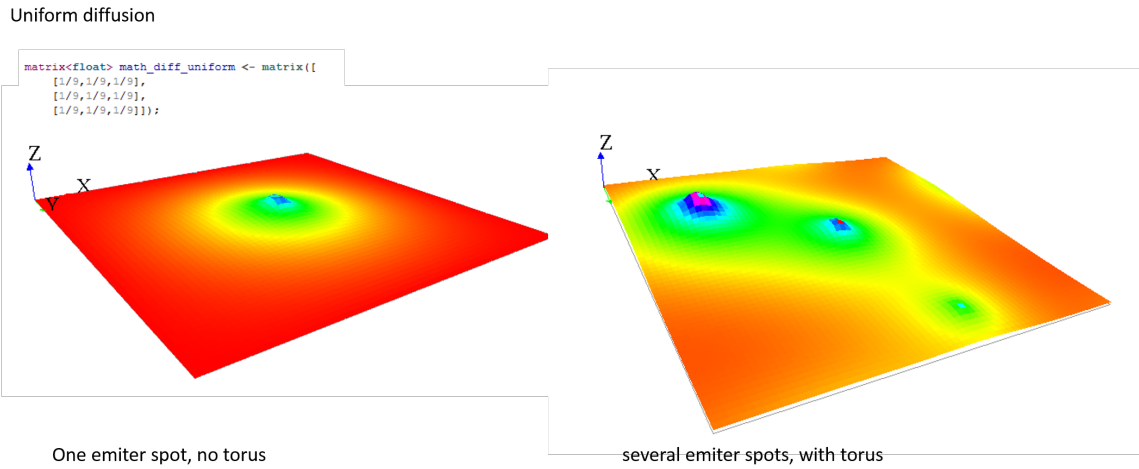


Figure 21.2: Examples of uniform diffusions with one and several sources.

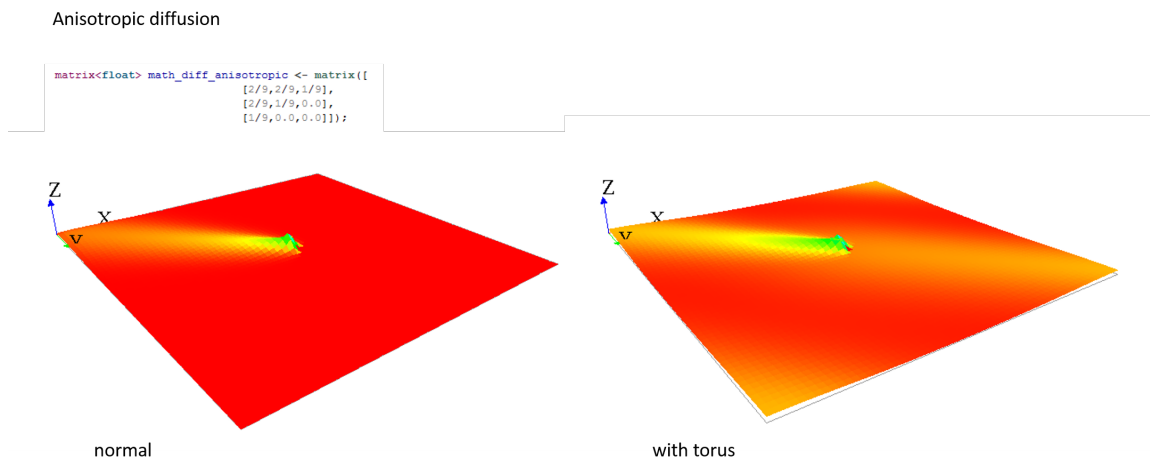


Figure 21.3: Examples of anisotropic diffusions (with and with torus environment).

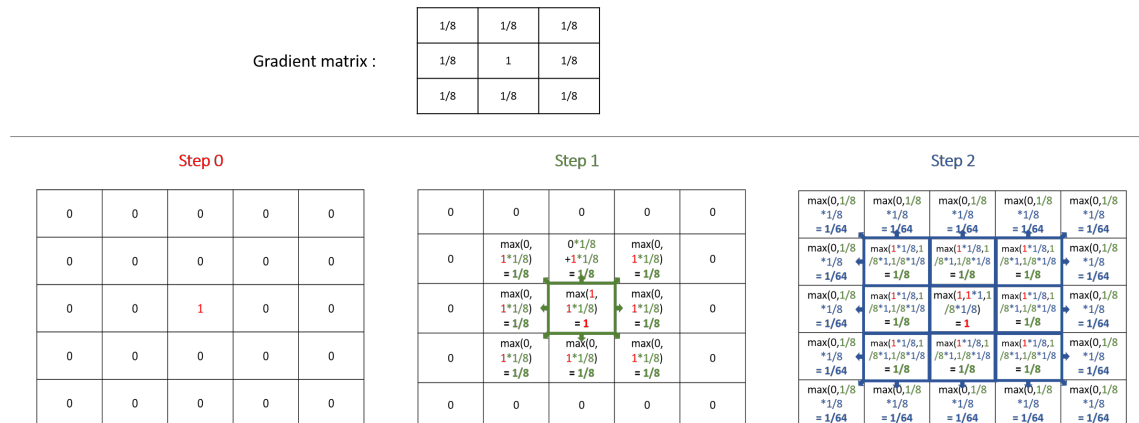


Figure 21.4: Illustration of the computation under a gradient propagation.

Gradient matrix

A **gradient** (use facet : `propagation:gradient`) is another type of propagation. This time, only the larger value diffused will be chosen as the new one.

Note that unlike the *diffusion* propagation, the sum of your matrix can be greater than 1 (and it is the case, most often !).

Here are some matrix examples with gradient propagation:

Compute multiple propagations at the same step

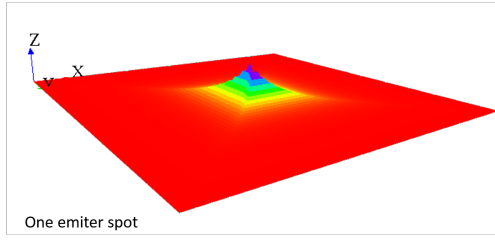
You can compute several times the propagation you want by using the facet `cycle_length`. GAMA will compute for you the corresponding new matrix and will apply it.

Writing those two things are exactly equivalent (for diffusion):

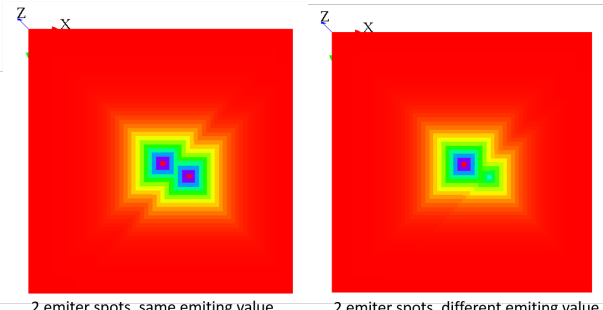
```
matrix<float> mat_diff <- matrix([
  [1/81,2/81,3/81,2/81,1/81] ,
  [2/81,4/81,6/81,4/81,2/81] ,
  [3/81,6/81,1/9,6/81,3/81] ,
  [2/81,4/81,6/81,4/81,2/81] ,
  [1/81,2/81,3/81,2/81,1/81]]);
reflex diff {
  diffuse var: phero on: cells matrix:mat_diff;
```

Uniform gradient

```
matrix<float> math_grad <- matrix([
  [3/4, 3/4, 3/4],
  [3/4, 1, 3/4],
  [3/4, 3/4, 3/4]]);
```



One emitter spot



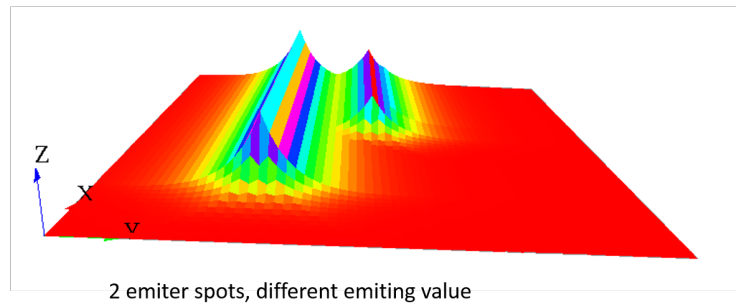
2 emitter spots, same emitting value

2 emitter spots, different emitting value

Figure 21.5: Examples of gradient diffusions with one and several sources.

Irregular gradient

```
matrix<float> math_grad <- matrix([
  [2/4, 3/4, 3/4],
  [1/4, 1, 1],
  [2/4, 3/4, 3/4]]);
```



2 emitter spots, different emitting value

Figure 21.6: Examples of irregular gradient diffusions.

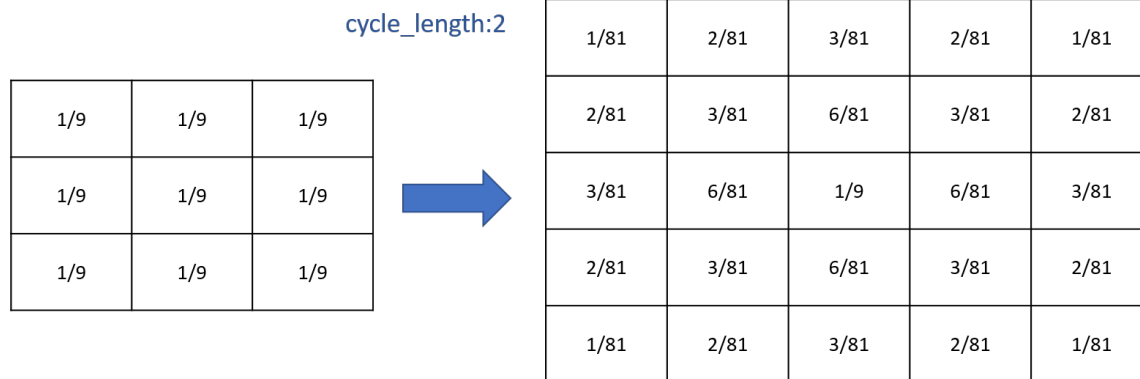


Figure 21.7: Example of computation with a cycle length of 2.

and

```
matrix<float> mat_diff <- matrix([
  [1/9,1/9,1/9],
  [1/9,1/9,1/9],
  [1/9,1/9,1/9]]);
reflex diff {
  diffuse var: phero on: cells matrix:mat_diff cycle_length
:2;
```

Executing several diffusion matrix

If you execute several times the statement `diffuse` with different matrix on the same variable, their values will be added (and centered if their dimensions are not equal).

Thus, the following 3 matrices will be combined to create one unique matrix:

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`.

$$\begin{array}{|c|c|c|} \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array} + \begin{array}{|c|} \hline 1/9 \\ \hline 0 \\ \hline 1/9 \\ \hline \end{array} + \begin{array}{|c|} \hline 1/9 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 1/9 & 0 & 1/9 \\ \hline 0 & 0 & 0 \\ \hline 1/9 & 0 & 1/9 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline 1/9 & 1/9 & 1/9 \\ \hline \end{array}$$

Figure 21.8: Example of matrix combinations.

Depending on which **propagation** you choose, and how many neighbors your grid has, the propagation matrix will be computed differently. The propagation matrix will have the size: $\text{range} * 2 + 1$.

Let's note **P** for the propagation value, **V** for the variation, **R** for the range and **N** for the number of neighbors.

- **With diffusion propagation**

For diffusion propagation, we compute following the following steps:

- (1) We determine the “minimale” matrix according to N (if $N = 8$, the matrix will be $[[P/9, P/9, P/9] [P/9, 1/9, P/9] [P/9, P/9, P/9]]$. if $N = 4$, the matrix will be $[[0, P/5, 0] [P/5, 1/5, P/5] [0, P/5, 0]]$).
- (2) If $R \neq 1$, we propagate the matrix R times to obtain a $[2 * R + 1] [2 * R + 1]$ matrix (same computation as for **cycle_length**).
- (3) If $V \neq 0$, we subtract each value by $V * \text{DistanceFromCenter}$ (DistanceFromCenter depends on N).

Ex with the default values ($P=1, R=1, V=0, N=8$):

size = 2*R + 1 = 5

$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64
$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,2)-0^*0$ = 1/64
$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,0)-0^*0$ = 1	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,2)-0^*0$ = 1/64
$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,1)-0^*0$ = 1/8	$1/\text{pow}(8,2)-0^*0$ = 1/64
$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64	$1/\text{pow}(8,2)-0^*0$ = 1/64

Figure 21.9: resources/images/recipes/gradient_computation_from_parameters.png

- **With gradient propagation**

The value of each cell will be equal to $**P/\text{POW}(N,\text{DistanceFromCenter})-\text{DistanceFromCenter}*V**$. (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 a uniform matrix.

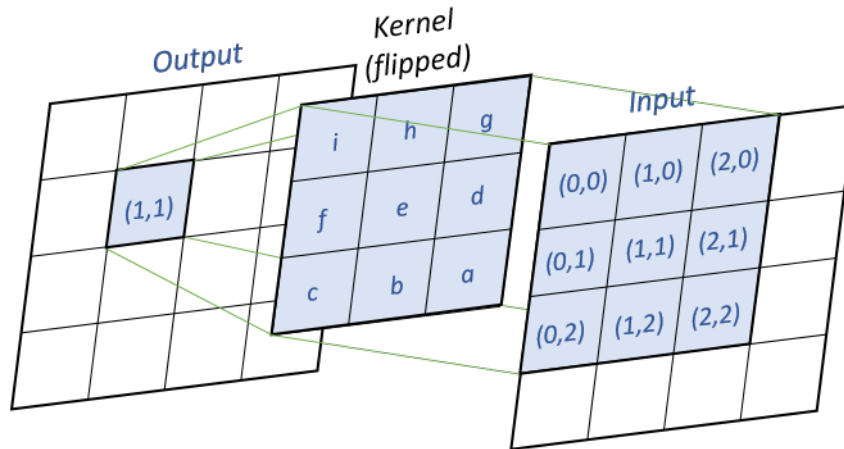


Figure 21.10: Illustration of convolution product computation.

Computation methods

You can compute the output matrix using two computation methods by using the facet `method`: the dot product and the convolution. Note that the result of those two methods is exactly the same (except if you use the `avoid_mask` facet, the results can be slightly different between the two computations).

Convolution

`convolution` is the default computation method for diffusion. For every output cells, we will multiply the input values and the flipped kernel together, as shown in the following image :

Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
  for (j = 0 ; j < y.nbCols ; j++)
    for (m = 0 ; m < k.nbRows ; m++)
      for (n = 0 ; n < k.nbCols ; n++)
        y[i,j] += k[k.nbRows - m - 1, k.nbCols - n - 1]
                * x[i - k.nbRows/2 + m, j - k.nbCols/2 + n]
```

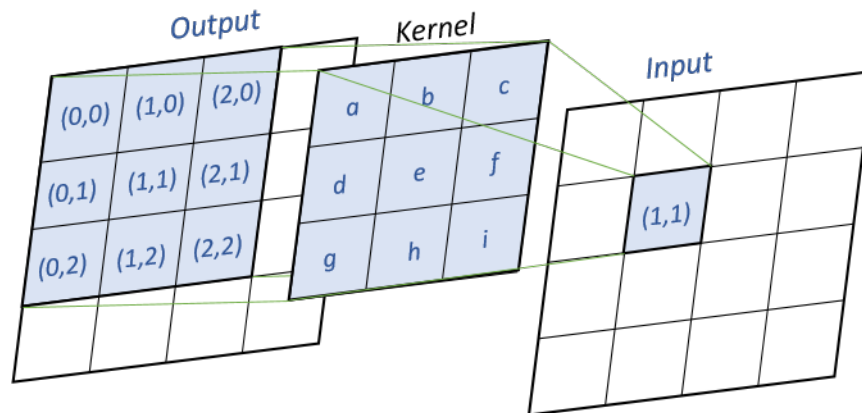


Figure 21.11: Illustration of dot product computation.

Dot Product

`dot_product` method will compute the matrix using a simple dot product between the matrix. For every input cells, we multiply the cell by the kernel matrix, as shown in the following image :

Pseudo-code (`k` the kernel, `x` the input matrix, `y` the output matrix) :

```
for (i = 0 ; i < y.nbRows ; i++)
  for (j = 0 ; j < y.nbCols ; j++)
    for (m = 0 ; m < k.nbRows ; m++)
      for (n = 0 ; n < k.nbCols ; n++)
        y[i - k.nbRows/2 + m, j - k.nbCols/2 + n] += k[m, n] *
          x[i, j]
```

Using a mask

Generalities

If you want to propagate some values in a heterogeneous grid, you can use some mask to forbid some cells to propagate their values.

You can pass a matrix to the facet `mask`. All the values smaller than `-1` will not propagate, and all the values greater or equal to `-1` will propagate.

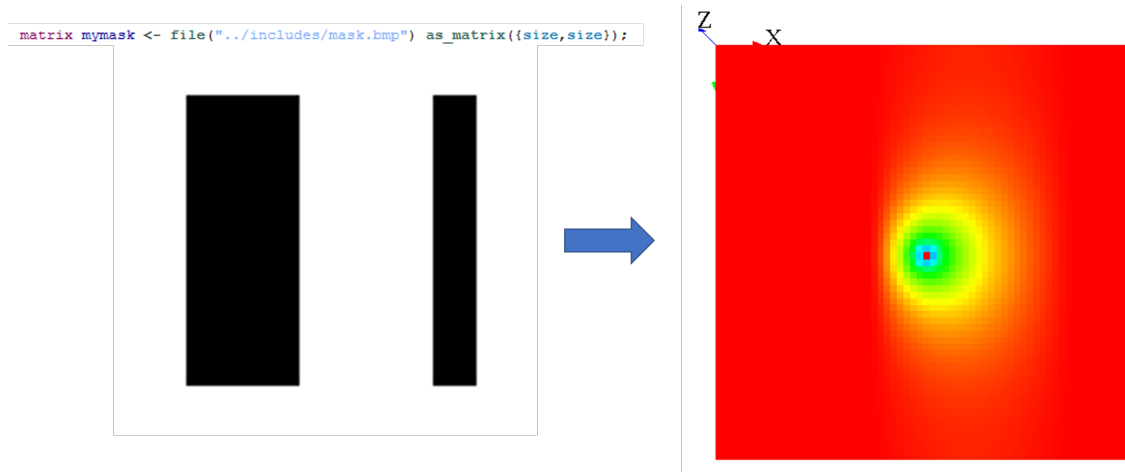


Figure 21.12: Use of a mask to constrain the diffusion.

A simple way to use mask is by loading an image :

Note that when you use the `on` facet for the `diffuse` statement, you can choose only some cells, and not every cell. In fact, when you restrain the values to be diffuse, it is exactly the same process as if you were defining a mask.

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) :

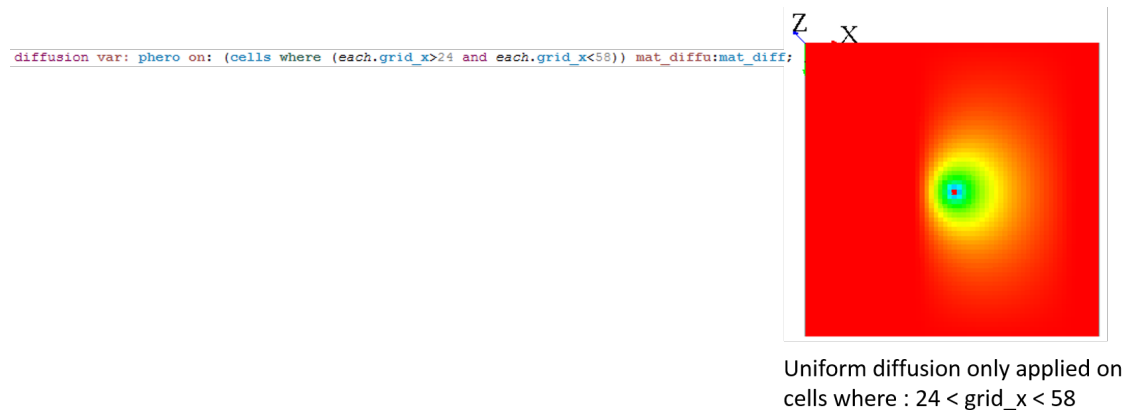


Figure 21.13: Constraint on the diffusion using filtering on cells.

```
matrix<float> mat_diff <- matrix([
  [1/9,1/9,1/9],
  [1/9,1/9,1/9],
  [1/9,1/9,1/9]]);

matrix<float> mat_diff_left_wall <- matrix([
  [0.0,0.0,2/9],
  [0.0,0.0,4/9],
  [0.0,0.0,2/9]]);

reflex diff {
  diffuse var: phero on: (cells where(each.grid_x>30))
  matrix:mat_diff;
  diffuse var: phero on: (cells where(each.grid_x=30))
  matrix:mat_diff_left_wall;
}
```

Note that almost the same result can be obtained by using the facet `avoid_mask`: the value of all masked cells will remain at 0, and the value which was supposed to be affected to the masked cell will be distributed to the neighboring cells. Notice that the results can be slightly different if you are using the `convolution` or the `dot_product` method: the algorithm of redistribution of the value to the neighboring cells is a bit different. We advise you to use the `dot_product` with the `avoid_mask` facet, the results are more accurate.

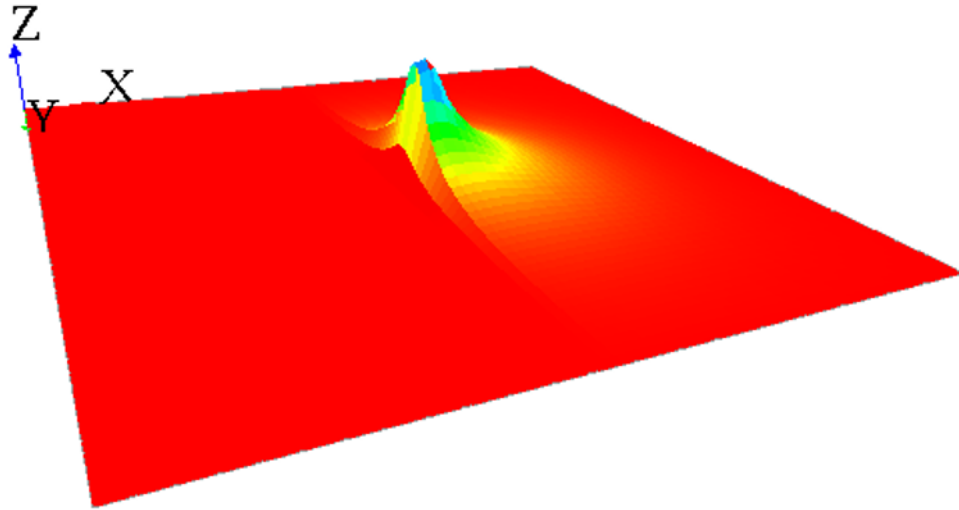


Figure 21.14: Diffusion limited by a wall, using a mask.

Wind pushing the diffusion

Let's simulate a uniform diffusion that is pushed by a wind from "north" everywhere in the grid. A wind from "west" as blowing at the top side of the grid. We will here have to build 2 matrices: one for the uniform diffusion, one for the "north" wind and one for the "west" wind. The sum of the values for the 2 matrices meant to simulate the wind will be equal to 0 (as it will be added to the diffusion matrix).

```
matrix<float> mat_diff <- matrix([
  [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],
```

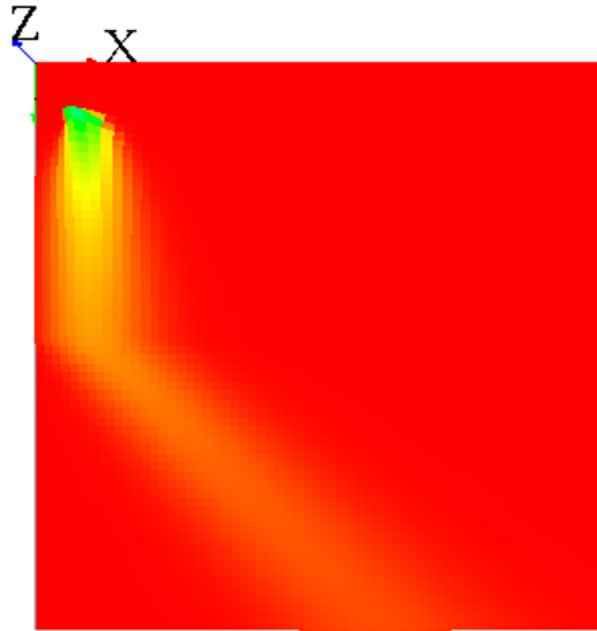



Figure 21.15: Diffusion impacted with a wind.

```

    [1/9,1/9,1/9]]);
reflex diff {
  diffuse var: phero on: cells matrix:mat_diff;
  diffuse var: phero on: cells matrix:mat_wind_from_north;
  diffuse var: phero on: (cells where (each.grid_y>=32))
  matrix:mat_wind_from_west;
}

```

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([
```

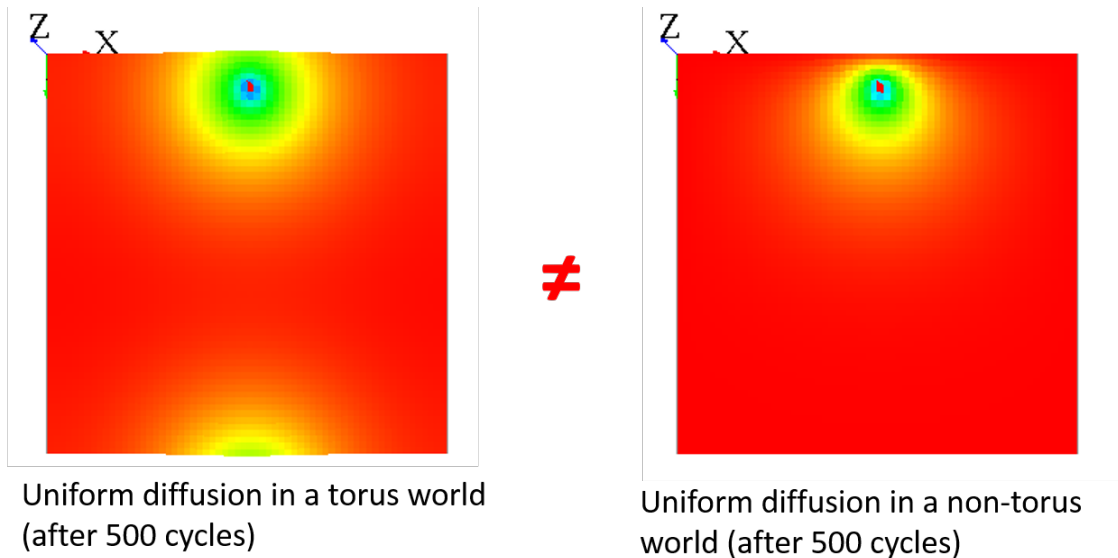


Figure 21.16: Comparison of diffusion with and without torus environment.

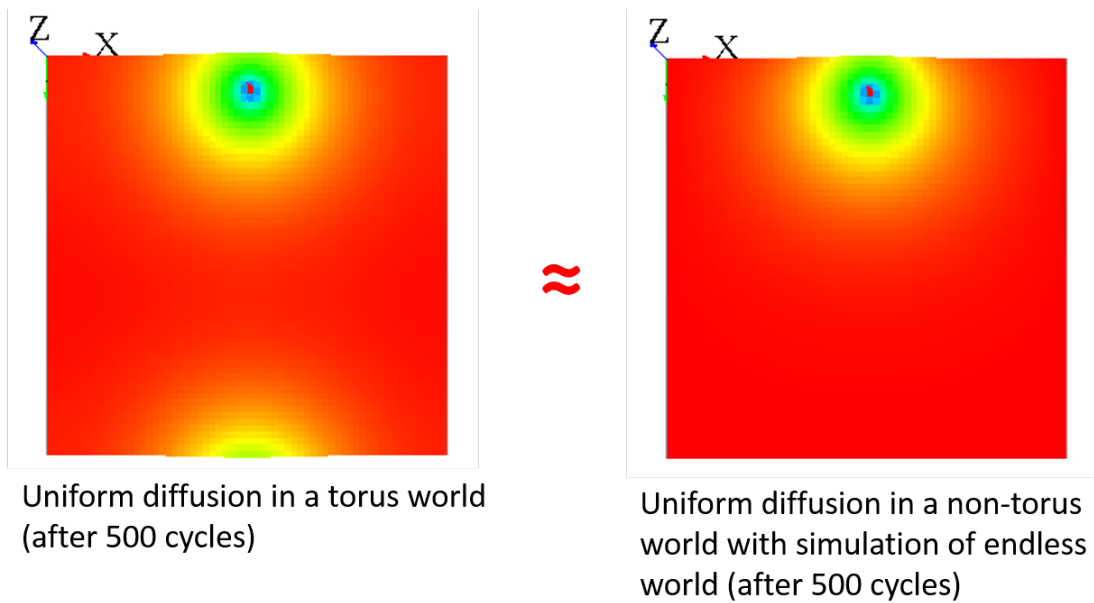


Figure 21.17: Attempt to fake torus environment with different matrices.

```

        [1/9,1/9,1/9],
        [1/9,1/9,1/9],
        [1/9,1/9,1/9]]);

matrix<float> mat_diff_upper_edge <- matrix([
        [0.0,0.0,0.0],
        [1/9+7/81,2/9+1/81,1/9+7/81],
        [1/9,1/9,1/9]]);

reflex diff {
    diffuse var: phero on: (cells where(each.grid_y>0)) matrix
    :mat_diff;
    diffuse var: phero on: (cells where(each.grid_y=0)) matrix
    :mat_diff_upper_edge;
}

```

Pseudo-code

This section is more for a better understanding of the source code.

Here is the pseudo-code for the computation of diffusion :

- 1) : Execute the statement `diffuse`, store the diffusions in a map (from class `DiffusionStatement` to class `GridDiffuser`) :

```

- Get all the facet values
- Compute the "real" mask, from the facet "mask:" and the
  facet "on:".
  - If no value for "mask:" and "on:" all the grid, the mask
    is equal to null.
- Compute the matrix of diffusion
  - If no value for "matrix:", compute with "nb_neighbors", "
    is_gradient", "proportion", "propagation", "variation", "
    range".
  - Then, compute the matrix of diffusion with "cycle_length".
- 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`, replace it by `input[idx] * matDiffu[i][j]`.
 - Else, do the computation (gradient or diffusion).
 - Finish the diffusion :
 - If `output[idx] > -Double.MAX_VALUE`, write the new `value` in the cell.

Chapter 22

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
<i>dbtype</i>	No	DBMS type value. Its value is a string. We must use “mysql” when we want to connect to a MySQL. That is the same for “postgres”, “sqlite” or “sqlserver” (ignore case sensitive)
<i>host</i>	Yes	Host name or IP address of data server. It is absent when we work with SQLite.
<i>port</i>	Yes	Port of connection. It is not required when we work with SQLite.
<i>database</i>	No	Name of database. It is the file name including the path when we work with SQLite.
<i>user</i>	Yes	Username. It is not required when we work with SQLite.
<i>passwd</i>	Yes	Password. It is not required when we work with SQLite.

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 <i>Preferences->External</i> 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
<i>olaptype</i>	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)
<i>dbtype</i>	No	DBMS type value. Its value is a string. We must use “mysql” when we want to connect to a MySQL. That is the same for “postgres” or “sqlserver” (ignore case sensitive)
<i>host</i>	No	Host name or IP address of data server.
<i>port</i>	No	Port of connection. It is no required when we work with SQLite.
<i>database</i>	No	Name of database. It is file name include path when we work with SQLite.
<i>catalog</i>	Yes	Name of catalog. It is an optional parameter. We do not need to use it when we connect to SSAS via XMLA and its file name includes the path when we connect a ROLAP database directly by using Mondrian API (see Example as below)
<i>user</i>	No	Username.
<i>passwd</i>	No	Password.

Table 2: OLAP Connection parameter description

Example: Definitions of OLAP connection parameter

```
//Connect SQL Server Analysis Services via XMLA
map<string,string> SSAS <- [
    'olaptype'::'SSAS/XMLA',
    'dbtype'::'sqlserver',
    'host'::'172.17.88.166',
    'port'::'80',
    'database'::'olap',
    'user'::'test',
```

```
        'passwd'::'abc'];  
  
//Connect 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 condition_ - 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

AgentDB 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
<i>dbtype</i>	No	DBMS type value. Its value is a string. We must use “mysql” when we want to connect to a MySQL. That is the same for “postgres”, “sqlite” or “sqlserver” (ignore case sensitive)
<i>host</i>	Yes	Host name or IP address of data server. It is absent when we work with SQLite.
<i>port</i>	Yes	Port of connection. It is not required when we work with SQLite.
<i>database</i>	No	Name of database. It is the file name including the path when we work with SQLite.
<i>user</i>	Yes	Username. It is not required when we work with SQLite.
<i>passwd</i>	Yes	Password. It is not required when we work with SQLite.
<i>srid</i>	Yes	srid (Spatial Reference Identifier) corresponds to a spatial reference system. This value is specified when GAMA connects to spatial database. If it is absent then GAMA uses spatial reference system defined in Preferences->External configuration.
<i>select</i>	No	Selection string

Table 3: Select boundary parameter description

Example:

```
map<string,string> BOUNDS <- [
  //'srid'::'32648',
  'host'::'localhost',
  'dbtype'::'postgres',
  'database'::'spatial_DB',
  'port'::'5433',
  'user'::'postgres',
  'passwd'::'tmt',
  'select'::'SELECT ST_AsBinary(geom) as geom FROM bounds;'
];
```

- **Step 2:** define boundary of environment by using the map object in first step.

```
geometry shape <- envelope(BOUNDS);
```

Note: We can do the same way if we work with MySQL, SQLite, or SQLServer and we must convert Geometry format in GIS database to binary format.

Create agents from the result of a select action

If we are familiar with how to create agents from a shapefile then it becomes very simple to create agents from select result. We can do as below:

- **Step 1:** Define a species with SQLSKILL or AgentDB

```
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 23

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

- Introduction
 - Configuration in GAMA
 - Calling R from GAML
 - * Calling the built-in operators
 - Example 1
 - * Calling R codes from a text file (.txt) WITHOUT the parameters
 - Example 2
 - Correlation.R file
 - * Output

- Example 3
- RandomForest.R file

- Load the package:
- Read data from iris:
- Build the decision tree:
- Build the random forest of 50 decision trees:
- Predict the acceptance of test set:
- Calculate the accuracy:
 - Output
 - Calling R codes from a text file (.R, .txt) WITH the parameters
 - * Example 4
 - * Mean.R file
 - Output
 - * Example 5
 - * AddParam.R file
 - * Output

Configuration in GAMA

- 1) Install R language into your computer.
- 2) In GAMA, select menu option: **Edit/Preferences**.
- 3) In “**Config RScript’s path**”, browse to your “**Rscript**” file (R language installed in your system).

Notes: Ensure that `install.packages(“Runiversal”)` is already applied in R environment.

Calling R from GAML

Calling the built-in operators

Example 1

```
model CallingR

global {
  list X <- [2, 3, 1];
  list Y <- [2, 12, 4];

  list result;

  init{
    write corR(X, Y); // -> 0.755928946018454
    write meanR(X); // -> 2.0
  }
}
```

Calling R codes from a text file (.R,.txt) WITHOUT the parameters

Using `R_compute(String RFile)` operator. This operator DOESN'T ALLOW to add any parameters from 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 24

Using FIPA ACL

GAMA allows modelers to provide agents the capability to communicate with other agents using [FIPA](#) Communication Acts (such as inform, request, call for proposal. . .) and [Interaction Protocols](#) (such [Contract Net Interaction Protocol](#), [Request Interaction Protocol](#)).

To add these capabilities to the chosen species, the modeler needs to attach the [fipa skill](#): it adds to agents of the species some additional attributes (e.g. the list of messages received) and available actions (e.g. the possibility to send messages given the chosen Communication Act).

The exhaustive list of available Communication Acts and Interaction Protocols is available from the technical description of the [fipa skill page](#). Examples can be found in the model library bundled with GAMA ([Plugin models / FIPA Skill](#)).

Table of Contents

- [Main steps to create a conversation using FIPA Communication Acts and Interaction Protocols](#)
- [Attach the fipa skill to a species](#)
- [Initiate a conversation](#)
- [Receive messages](#)
- [Reply to a received message](#)
- [The message data type](#)
- [The conversation data type](#)

Main steps to create a conversation using FIPA Communication Acts and Interaction Protocols

1. Attach the skill `fipa` to the agents' species that need to use Communication Acts
2. An initiator agent starts a conversation with some agents: it chooses the Interaction Protocol and starts it by sending the first Communication Acts of the protocol
3. Each agent involved in the conversation needs to check its received messages and respond to them by choosing the appropriate Communication Act.

Attach the `fipa` skill to a species

To attach the `fipa` skill to a species, the modeler has to add it in the `skills` facet of the `species` statement (in a way similar to any other skill).

```
species any_species skills: [fipa] {  
    ...  
}
```

Agents of any species can communicate in the same conversation. The only constraint is that they need to have the capabilities to receive and send messages, i.e. to have the skill `fipa`.

Species can have several attached skills: a single species can be provided with both the `moving` and `fipa` skills (and any other ones).

This skill adds to every agent of the species: * some additional attributes: * `conversations` is the list of the agent's current conversations, * `mailbox` is the list of messages of all types of performatives, * `requests`, `informs`, `proposes...` are respectively the list of the 'request', 'inform', 'propose' performative messages. * some additional actions, such as: * `inform`, `accept_proposal...` that replies a message with an 'inform' (respectively 'accept_proposal' performative message). * `start_conversation` that starts a conversation with a chosen interaction protocol. * `end_conversation` that replies a message with an 'end_conversation' performative message. This message marks the end of a conversation. In a 'no-protocol' conversation, it is the responsibility of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol. * `reply` that replies a message. This

action should be only used to reply a message in a ‘no-protocol’ conversation and with a ‘user-defined performative’. For performatives supported by GAMA, please use the ‘action’ with the same name as the ‘performative’. For example, to reply a message with a ‘request’ performative message, the modeler should use the ‘request’ action.

Initiate a conversation

An interaction using an Interaction Protocol starts with the creation of a conversation by an agent, using the `start_conversation` action.

The modeler specifies the chosen **protocol** (facet `protocol`), **list of participants** (facet `to`), **communication act** (facet `performative`) and **message** (facet `contents`).

```
species Initiator skills: [fipa] {
  reflex send_propose_message when: (time = 1) {
    do start_conversation to: [p] protocol: 'fipa-propose'
    performative: 'propose' contents: ['Go swimming?'] ;
  }
}
```

Receive messages

Each agent (with the `fipa` skill) is provided with several “mailbox” attributes filtering the various received messages by communication act: e.g. `proposes` contains the list of the received messages with the “Propose” communication act.

Receiving a message consists thus in looking at each message from the mailbox, and acting in accordance with its contents, participants...

Important remark: once the `contents` field of a received message has been read, it is removed from all the lists it appears in.

```
species Initiator skills: [fipa] {
  reflex read_accept_proposals when: !(empty(
  accept_proposals)) {
    write name + ' receives accept_proposal messages';
    loop i over: accept_proposals {
      write 'accept_proposal message with content: ' +
      string(i.contents);
    }
  }
}
```

```

    }
  }
}

species Participant skills: [fipa] {
  reflex accept_proposal when: !(empty(proposes)) {
    message proposalFromInitiator <- proposes at 0;

    do accept_proposal message: proposalFromInitiator
    contents: ['OK! It \'s hot today!'] ;
  }
}

```

Remark: * To test that the agent has received a new message is simply done by testing whether the dedicated mailing box contains messages. * To get a message, the modeler can either loop over the message list to get all the messages or get a message by its index in the message box.

Reply to a received message

Given the message it has received, an agent can reply using the appropriate Communication Act (using the appropriate action). It simply has to specify the message to which it replies and the content of the reply.

Note that it does not need to specify the receiver as it is contained in the message.

```

species Participant skills: [fipa] {
  reflex accept_proposal when: !(empty(proposes)) {
    message proposalFromInitiator <- proposes at 0;

    do accept_proposal message: proposalFromInitiator
    contents: ['OK! It \'s hot today!'] ;
  }
}

```

End a conversation

When a conversation is made in the scope of an Interaction Protocol, it is ended automatically when the last Communicative Act has been sent.

In the case of a ‘no-protocol conversation’, it is the responsibility of the modeler to explicitly send the `end_conversation` message to mark the end of a conversation/interaction protocol.

When a conversation ends, it is automatically removed from the list `conversations`.

The message type

The agents’ mailbox is defined as a list of messages. Each message is a GAML object of type `message`. An exhaustive description of this type is provided in the dedicated [GAML Data Types page](#).

A `message` object is defined by a set of several fields, such as: * `contents` (type `unknown`): the content of the message * `sender` (type `unknown`): the sender of the message. In the case where the sender is an agent, it is possible to get the corresponding agent with `agent(m.sender)` (where `m` is the considered message). * `unread` (type `bool`): specify whether the message has been read. * `emission_timestamp` (type `int`) * `reception_timestamp` (type `int`)

The conversation data type

The agents’ `conversations` contain the list of the conversations in which the agent takes part. Each conversation is a GAML object of type `conversation` that contains the list of messages exchanged, the protocol, initiator. . . An exhaustive description of this type is provided in the dedicated [GAML Data Types page](#).

A `conversation` object is defined by a set of several fields, such as: * `messages` (type = list of messages): the list of messages that compose this conversation * `protocol` (type = string): the name of the protocol followed by the conversation * `initiator` (type = agent): the agent that has initiated this conversation * `participants` (type = list of agents): the list of agents that participate in this conversation * `ended` (type = bool): whether this conversation has ended or not

Chapter 25

Using GAMAnalyzer

Install

Go to Git View -> Click on Import Projects Add the dependencies in `umisco.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.

- **metadatabasehistory:** 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_metadatabasehistory:** Aggregate each metadatabasehistory 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 <-"singlobal";
  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 26

Using BEN (simple_bdi)

Introduction to BEN

BEN (Behavior with Emotions and Norms) is an agent architecture providing social agents with cognition, emotions, emotional contagion, personality, social relations, and norms. This work has been done during the Ph.D. of Mathieu Bourgeois, funded by the ANR ACTEUR.

The BEN architecture is accessible in GAMA through the use of the `simple_bdi` architecture when defining agents. This page indicates the theoretical running of BEN as well as the practical way it has been implemented in GAMA.

This page features all the descriptions for the running of the BEN architecture. This page is updated with the version of BEN implemented in GAMA. To get more details on its implementation in GAMA, see [operators related to BDI](#), [BDI tutorial](#) or [BDI built-in architecture reference](#).

The BEN architecture

The BEN Architecture used by agents to make a decision at each time step is represented by the image right below:

Each social agent has its own instance of the BEN architecture to make a decision. The architecture is composed of 4 main parts connected to the agent's knowledge bases, seated on the agent's personality. Each part is made up of processes that

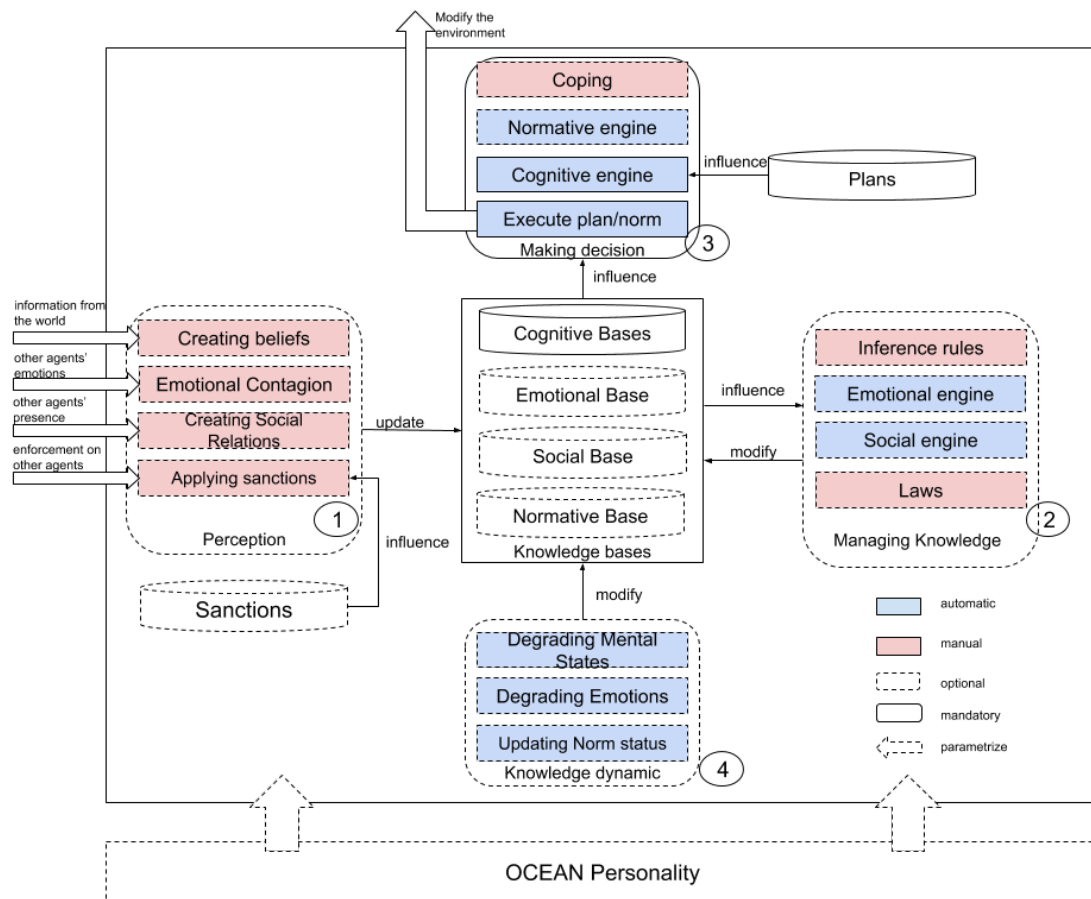


Figure 26.1: Architecture of the BEN architecture.

are automatically computed (in blue) or which need to be manually defined by the modeler (in pink). Some of these processes are mandatory (in solid line) and some others are optional (in dotted line). This modularity enables each modeler to only use components that seem pertinent to the studied situation without creating heavy and useless computations.

The Activity diagram bellow shows the order in which each module and each process is activated. The rest of this page explains in details how each process from each module works and what is the difference between the theoretical architecture and its implementation.

Predicates, knowledge and personality

In BEN, an agent represents its environment through the concept of predicates.

A predicate represents information about the world. This means it may represent a situation, an event or an action, depending on the context. As the goal is to create behaviors for agents in a social environment, that is to say taking actions performed by other agents into account with facts from the environment in the decision making process, an information P caused by an agent j with an associated list of value V is represented by $\mathbf{P}_j(\mathbf{V})$. A predicate \mathbf{P} represents an information caused by any or none agent, with no particular value associated. The opposite of a predicate P is defined as **not P**.

In GAML, the 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_values** changes all the map of values while **add_values** enables to add a new value without changing the rest of the map). These values can be accessed with operators **get_agent_cause**, **get_truth**, **get_values**. An operator **not** is also defined for predicates.

Below is an example of how to define predicates in GAML:

```
predicate a <- new_predicate("test");
predicate b <- new_predicate("test",["value1"::10]);
predicate c <- new_predicate("test",agentBob);
predicate d <- new_predicate("test",false);
predicate e <- new_predicate("test",agenBob,false);
```

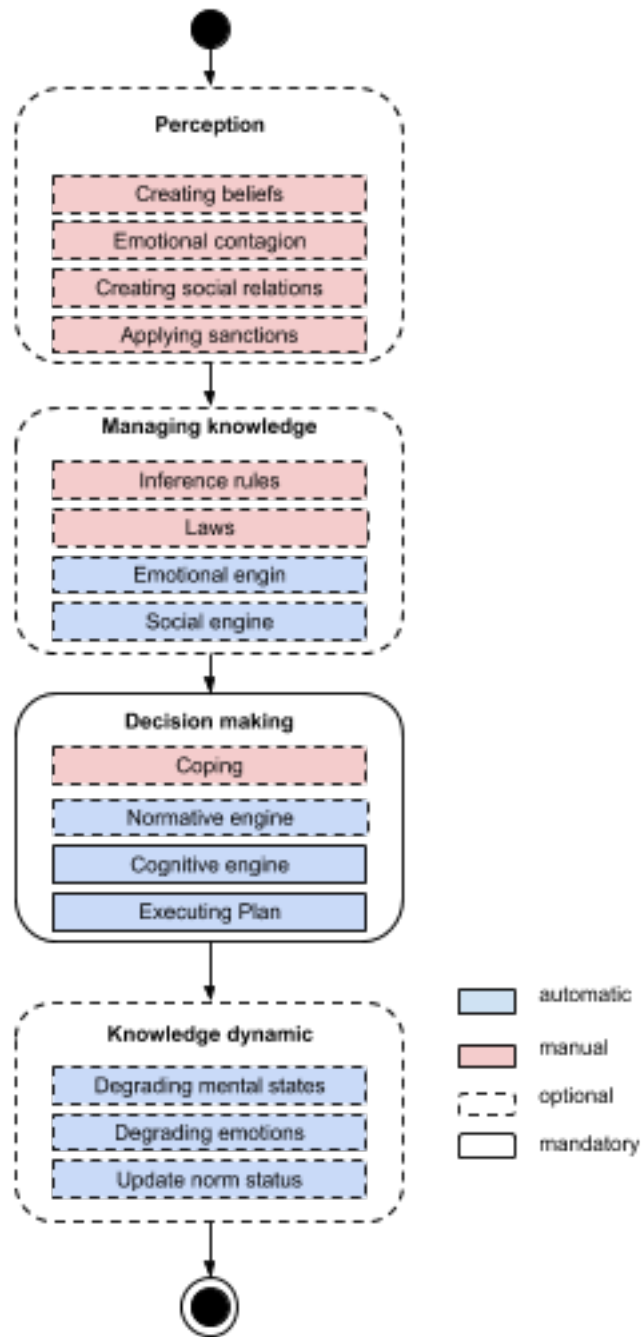


Figure 26.2: Activity diagram illustrating the activation order of the BEN architecture.

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 (Bratman, 1987) in which agents have a belief base, a desire base and an intention base to store the cognitive mental states about the world. In order to connect cognition with other social features, the architecture outlines a total of 6 different modalities which are defined as follows:

- **Belief**: represents what the agent knows about the world. The value attached to this mental state indicates the strength of the belief.
- **Uncertainty**: represents an uncertain information about the world. The value attached to this mental state indicates the importance of the uncertainty.
- **Desire**: represents a state of the world the agent wants to achieve. The value attached to this mental state indicates the priority of the desire.
- **Intention**: represents a state of the world the agent is committed to achieve. The value attached to this mental state indicates the priority of the intention.
- **Ideal**: represents an information socially judged by the agent. The value attached to this mental state indicates the praiseworthiness value of the ideal about P. It can be positive (the ideal about P is praiseworthy) or negative (the ideal about P is blameworthy).
- **Obligation**: represents a state of the world the agent has to achieve. The value attached to this mental state indicates the priority of the obligation.

In GAML, mental states are manipulated thanks to add, remove and get actions related to each modality: **add_belief**, **remove_belief**, **get_belief**, **add_desire**,

`remove_desire` ... Then, operators enables to acces or modify each attribute of a given mental state: `get_predicate`, `set_predicate`, `get_strength`, `set_strength`, `get_lifetime`, `set_lifetime`, etc.

Below is an exemple of code in GAML concerning cognitive mental states:

```
reflex testCognition{
  predicate a <- new_predicate("test");
  do add_belief(a,strength1,lifetime1);
  mental_state b <- get_uncertainty(a);
  int c <- get_lifetime(b);
}
```

Emotions

In BEN, the definition of emotions is based on the OCC theory of emotions (Ortony, 90). According to this theory, an emotion is a valued answer to the appraisal of a situation. Once again, as the agents are taken into consideration in the context of a society and should act depending on it, the definition of an emotion needs to contain the agent causing it. Thus, an emotion is represented by **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 **Emi(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 emotions:

```
reflex testEmotion{
  predicate a <- new_predicate("test");
  do add_emotion(new_emotion("hope",a));
  do add_emotion(new_emotion("joy",intensity1,a, decay1));
  float c <- get_intensity(get_emotion(new_emotion("joy",a))
);
}
```

Social relations

As people create social relations when living with other people and change their behavior based on these relationships, BEN architecture makes it possible to describe social relations in order to use them in agents' behavior. Based on the research carried out by (Svennevig, 2000), a social relation is described by using a finite set of variables. Svennevig identifies a minimal set of four variables: liking, dominance, solidarity, and familiarity. A trust variable is added to interact with the enforcement of social norms. Therefore, in BEN, a social relation between agent i and agent j is expressed as $\mathbf{R}_{i,j}(\mathbf{L},\mathbf{D},\mathbf{S},\mathbf{F},\mathbf{T})$ with the following elements:

- **R**: the identifier of the social relation.
- **L**: a real value between -1 and 1 representing the degree of liking with the agent concerned by the link. A value of -1 indicates that agent j is hated, a value of 1 indicates that agent j is liked.
- **D**: a real value between -1 and 1 representing the degree of power exerted on the agent concerned by the link. A value of -1 indicates that agent j is dominating, a value of 1 indicates that agent j is dominated.
- **S**: a real value between 0 and 1 representing the degree of solidarity with the agent concerned by the link. A value of 0 indicates that there is no solidarity with agent j , a value of 1 indicates a complete solidarity with agent j .
- **F**: a real value between 0 and 1 representing the degree of familiarity with the agent concerned by the link. A value of 0 indicates that there is no familiarity with agent j , a value of 1 indicates a complete familiarity with agent j .
- **T**: a real value between -1 and 1 representing the degree of trust with the agent j . A value of -1 indicates doubts about agent j while a value of 1 indicates complete trust with agent j . The trust value does not evolve automatically in accordance with emotions.

With this definition, a social relation is not necessarily symmetric, which means $R_{i,j}(L,D,S,F,T)$ is not equal by definition to $R_{j,i}(L,D,S,F,T)$. $\mathbf{L}[R_{i,j}]$ stands for the liking value of the social relation between agent i and agent j , $\mathbf{D}[i,j]$ stands for its dominance value, $\mathbf{S}[R_{i,j}]$ for its solidarity value, $\mathbf{F}[R_{i,j}]$ represents its familiarity value and $\mathbf{T}[R_{i,j}]$ its trust value.

In GAML, social relations are manipulated with `add_social_link`, `remove_social_link` and `get_social_link` actions. Each feature of a social link is accessible with `set` and `gt` operators (`set_agent`, `get_agent`, `set_liking`, `get_liking`, `set_dominance`, etc.)

Below is an exemple of code to manipulates social relations in GAML:

```

reflex testSocialRelations{
  do add_social_link(new_social_link(agentAlice));
  do add_social_link(new_social_link(agentBob
    ,0.5,-0.3,0.2,0.1));
  float val <- get_liking(get_social_link(new_social_link(
    agentBob)));
  social_link sl <- set_dominance(get_social_link(
    new_social_link(agentBob)),0.3);
}

```

Personality and additional variables

In order to define personality traits, BEN relies on the OCEAN model (McCrae, 1992), also known as the big five factors model. In the BEN architecture, this model is represented through a vector of five values between 0 and 1, with 0.5 as the neutral value. The five personality traits are:

- **O**: represents the openness of someone. A value of 0 stands for someone narrow-minded, a value of 1 stands for someone open-minded.
- **C**: represents the consciousness of someone. A value of 0 stands for someone impulsive, a value of 1 stands for someone who acts with preparations.
- **E**: represents the extroversion of someone. A value of 0 stands for someone shy, a value of 1 stands for someone extrovert.
- **A**: represents the agreeableness of someone. A value of 0 stands for someone hostile, a value of 1 stands for someone friendly.
- **N**: represents the degree of control someone has on his/her emotions, called neurotism. A value of 0 stands for someones neurotic, a value of 1 stands for someone calm.

In GAML, these variables are build-in attributes of agents using the `simple_bdi` control architecture. They are called *openness*, *conscientiousness*, *extroversion*, *agreeableness* and *neurotism*. To use this personality to automatically parametrize the other modules, a modeler needs to indicate it as shown in the GAML example below:

```
species miner control:simple_bdi {  
  ...  
  bool use_personality <- true;  
  float openness <- 0.1;  
  float conscientiousness <- 0.2;  
  float extroversion <- 0.3;  
  float agreeableness <- 0.4;  
  float neurotism <- 0.5;  
  ...  
}
```

With BEN, the agent has variables related to some of the social features. The idea behind the BEN architecture is to connect these variables to the personality module and in particular to the five dimensions of the OCEAN model in order to reduce the number of parameters which need to be entered by the user. These additional variables are:

- The probability to keep the current plan.
- The probability to keep the current intention.
- A charisma value linked to the emotional contagion process.
- An emotional receptivity value linked to the emotional contagion.
- An obedience value used by the normative engine.

With the cognition, the agent has two parameters representing the probability to randomly remove the current plan or the current intention in order to check whether there could be a better plan or a better intention in the current context. These two values are connected to the consciousness components of the OCEAN model as it describes the tendency of the agent to prepare its actions (with a high value) or act impulsively (with a low value).

- Probability Keeping Plans = $C1/2$
- 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 manually. For the probability to keep the current plan and the probability to keep the current intention, he/she also has to indicates it with a particular boolean value, as shown in the GAML example below:

```
species miner control: simple_bdi {
  ...
  bool use_personality <- true;
  bool use_persistence <- true;
  float plan_persistence <- 0.3;
  float intention_persistence <- 0.4;
  float obedience <- 0.2;
  float charisma <- 0.3;
  float receptivity <- 0.6;
  ...
}
```

Perception

The first step of BEN is the perception of the environment. This module is used to connect the environment to the knowledge of the agent, transforming information from the world into cognitive mental states, emotions or social links but also used to apply sanctions during the enforcement of norms from other agents.

Below is an example of code to define a perception in GAML:

```
perceive target: fireArea in: 10{
    ...
}
```

The first process in this perception consists of **adding beliefs** about the world. During this phase, information from the environment is transformed into predicates which are included in beliefs or uncertainties and then added to the agent's knowledge bases. This process enables the agent to update its knowledge about the world. From the modeler's point of view, it is only necessary to specify which information is transformed into which predicate. The addition of a belief $BeliefA(X)$ 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 $Chi \times Rj$ 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 automatically computed.

- If $Em_j(P)$ already exists:
 - $I[Em_j(P)] = I[Em_j(P)] + I[Em_i(P)] \times Chi \times Rj$
 - if $pEm_i(P) > I[Em_j(P)]$:
 - * $De[Em_j(P)] = De[Em_i(P)]$
 - if $I[Em_j(P)] > I[Em_i(P)]$:
 - * $De[Em_j(P)] = De[Em_j(P)]$
- If $Em_j(P)$ does not already exist:
 - $I[Em_j(P)] = I[Em_i(P)] \times Chi \times Rj$
 - $De[Em_j(P)] = De[Em_i(P)]$.

In GAML, *emotional_contagion* statement helps to define an emotional contagion during a perception, as shown below:

```
perceive target: otherHumanAgents in: 10{
  emotional_contagion emotion_detected:fearFire threshold:
  contagionThreshold;
  //creates the detected emotion, if detected, in the agent
  doing the perception.
  emotional_contagion emotion_detected:joyDance
  emotion_created:joyPartying;
  //creates the emotion "joyPartying", if emotion "joyDance"
  is detected in the perceived agent.
}
```


During the perception, the agent has the possibility of **creating social relations** with other perceived agents. The modeler indicates the initial value for each component of the social link, as explained previously. By default, a neutral relation is created, with each value of the link at 0.0. Social relations can also be defined before the start of the simulation, to indicate that an agent has links with other agents at the start of the simulation, like links with friends or family members.

In GAML, the *socialize* statement help creating dynamicaly new social relations, as shown below:

```
perceive target:otherHumanAgents in: 10{
  socialize;
  //creates a neutral relation
  socialize dominance: -0.8 familiarity:0.2 when: isBoss;
  //example of a social link with precise values for some of
  its dimensions in a certain context
}
```

Finally, the agent may **apply sanctions** through the norm enforcement of other agents perceived. The modeler needs to indicate which modality is enforced and the sanction and reward used in the process. Then, the agent checks if the norm, the obligation, or the law, is violated, applied or not activated by the perceived agent. Notions of norms laws and obligations and how they work are explained later in this 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 knowledge bases

The second step of the architecture, corresponding to the module number 2, consists of managing the agent's knowledge. This means updating the knowledge bases according to the latest perceptions, adding new desires, new obligations, new emotions or updating social relations, for example.

Modelers have to use **inference rules** for this purpose. These rules are triggered by a new belief, a new uncertainty or a new emotion, in a certain context, and may add or remove any cognitive mental state or emotion indicated by the user. Using multiple inference rules helps the agent to adapt its mind to the situation perceived without removing all its older cognitive mental states or emotions, thus enabling the creation of a cognitive behavior. These inference rules enable to link manually the various dimensions of an agent, for example creating desires depending on emotions, social relations and personality.

In GAML, the *rule* statement enables to define inference rules:

```

species miner skills: [moving] control: simple_bdi {
...
  perceive target: miner in: viewdist {
...
  }
...
  rule belief: new_predicate("testA") new_desire:
new_predicate("testB");
}

```

Using the same idea, modelers can define **laws**. These laws enable the creation of obligations in a given context based on the newest beliefs created by the agent through its perception or its inference rules. The modeler also needs to indicate an

obedience threshold and if the agent's obedience value is below that threshold, the law is violated. If the law is activated, the obligation is added to the agent's cognitive mental state bases. The definition of laws makes it possible to create a behavior based on obligations imposed upon the agent.

Below is an example of the definition of a *law* statement in GAML:

```
law belief: new_predicate("testA") new_obligation:
  new_predicate("testB") threshold:thresholdLaw;
```

Emotional engine

BEN enables the agent to get emotions about its cognitive mental states. This **addition of emotions** is based on the OCC model (Ortony, 1990) and its logical formalism (Adam, 2007), which has been proposed to integrate the OCC model in a BDI formalism.

According to the OCC theory, emotions can be split into three groups: emotions linked to events, emotions linked to people and actions performed by people, and emotions linked to objects. In BEN, as the focus is on relations between social agents, only the first two groups of emotions (emotions linked to events and people) are considered.

The twenty emotions defined in this paper can be divided into seven groups depending on their relations with mental states: emotions about beliefs, emotions about uncertainties, combined emotions about uncertainties, emotions about other agents with a positive liking value, emotions about other agents with a negative liking value, emotions about ideals and combined emotions about ideals. All the initial intensities and decay value are computed using the OCEAN model and the value attached to the concerned mental states.

The emotions about beliefs are joy and sadness and are expressed this way:

- $Joyi(Pj,j) = Beliefi(Pj) \ \& \ Desirei(P)$
- $Sadnessi(Pj,j) = Beliefi(Pj) \ \& \ Desirei(not \ P)$

Their initial intensity is computed according to the following equation with N the neurotism component from the OCEAN model:

- $I[Emi(P)] = V[Beliefi(P)] \times V[Desirei(P)] \times (1+(0,5-N))$

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:

- $I[Emi(P)] = V[Uncertaintyi(P)] \times V[Desirei(P)] \times (1+(0,5-N))$

Combined emotions about uncertainties are emotions built upon fear and hope. They appear when an uncertainty is replaced by a belief, transforming fear and hope into satisfaction, disappointment, relief or fear confirmed and they are defined this way:

- **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.

- $I[Emi(P)] = V[Beliefi(P)] \times I[Em'i(P)]$

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 for i (P,j)** = $L[Ri,j]>0$ & Joyj(P)
- **Sorry for i (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.

- $I[Emi(P)] = I[Emj(P)] \times L[Ri,j] \times (1-(0,5-A))$

Emotions about other agents with a negative liking value are close to the previous definitions, however, they are related to the emotions of other agents which are in the social relation base with a negative liking value. These emotions are resentment and gloating and have the following definition:

- **Resentment i (P,j)** = $L[Ri,j]<0$ & Joyj(P)
- **Gloating i (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”.

- $I[Emi(P)] = I[Emj(P)] \times |L[Ri,j]| \times (1+(0,5-A))$

Emotions about ideals are related to the agent’s ideal base which contains, at the start of the simulation, all the actions about which the agent has a praiseworthiness value to give. These ideals can be praiseworthy (their praiseworthiness value is positive) or blameworthy (their praiseworthiness value is negative). The emotions coming from these ideals are pride, shame, admiration and reproach and have the following definition:

- **Pride i (Pi,i)** = Beliefi(Pi) & Ideali(Pi) & $V[Ideali(Pi)]>0$

- **Shamei(P_i,i)** = Beliefi(P_i) & Ideali(P_i) & V[Ideali(P_i)]<0
- **Admirationi(P_j,j)** = Beliefi(P_j) & Ideali(P_j) & V[Ideali(P_j)]>0
- **Reproachi(P_j,j)** = Beliefi(P_j) & Ideali(P_j) & V[Ideali(P_j)]<0

Their initial intensity is computed according to the following equation with O the openness value from the OCEAN model:

- $I[Em_i(P)] = V[Belief_i(P)] \times |V[Ideali(P)]| \times (1+(0,5-O))$

Finally, combined emotions about ideals are emotions built upon pride, shame, admiration and reproach. They appear when joy or sadness appear with an emotion about ideals. They are gratification, remorse, gratitude and anger which are defined as follows:

- **Gratificationi(P_i,i)** = Pridei(P_i,i) & Joyi(P_i)
- **Remorsei(P_i,i)** = Shamei(P_i,i) & Sadnessi(P_i)
- **Gratitudei(P_j,j)** = Admirationi(P_j,j) & Joyi(P_j)
- **Angeri(P_j,j)** = Reproachi(P_j,j) & Sadnessi(P_j)

Their initial intensity is computed according to the following equation with Em^{'i}(P) the emotion about ideals and Em^{"i}(P) the emotion about beliefs.

- $I[Em_i(P)] = I[Em^{'i}(P)] \times I[Em^{''i}(P)]$

In order to keep the initial intensity of each emotion between 0 and 1, each equation is truncated between 0 and 1 if necessary.

The initial decay value for each of these twenty emotions is computed according to the same equation with Deltat a time step which enables to define that an emotion does not last more than a given time:

- $De[Em_i(P)] = N \times I[Em_i(P)] \times Deltat$

To use this automatic computation of emotion, a modeler need to activate it as shown in the GAML example below :

```
species miner control:simple_bdi {
  ...
  bool use_emotions_architecture <- true;
  ...
}
```

Social Engine

When an agent already known is perceived (i.e. there is already a social link with it), the social relationship with this agent is updated automatically by BEN. This update is based on the work of (Ochs, 2009) and takes the agent's cognitive mental states and emotions into account. In this section, the **automatic update of each variable of a social link** $R_{i,j}(L,D,S,F,T)$ by the architecture is described in details; the trust variable of the link is however not updated automatically.

- **Liking:** according to (Ortony, 1991), the degree of liking between two agents depends on the valence (positive or negative) of the emotions induced by the corresponding agent. In the emotional model of the architecture, *joy* and *hope* are considered as positive emotions (*satisfaction* and *relief* automatically raise *joy* with the emotional engine) while *sadness* and *fear* are considered as negative emotions (*fear confirmed* and *disappointment* automatically raise *sadness* with the emotional engine). So, if an agent i has a positive (resp. negative) emotion caused by an agent j , this will increase (resp. decrease) the value of appreciation in the social link from i concerning j .

Moreover, research has shown that the degree of liking is influenced by the solidarity value [?]. 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.

- $L[R_{i,j}] = L[R_{i,j}] + |L[R_{i,j}](1 - |L[R_{i,j}])| S[R_{i,j}] + aL (1 - |L[R_{i,j}])|(mPos - mNeg)$
- $aL = 1 - N$
- **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.

- $S[Ri,j]=S[Ri,j] + S[Ri,j] \times (1-S[Ri,j]) \times (aS1 (sim-opp)/(NbKnow) - aS2 mNeg)$
- $aS1 = 1-O$
- $aS2 = 1-N$
- **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.
- $F[Ri,j]=F[Ri,j] \times (1+L[Ri,j])$

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.

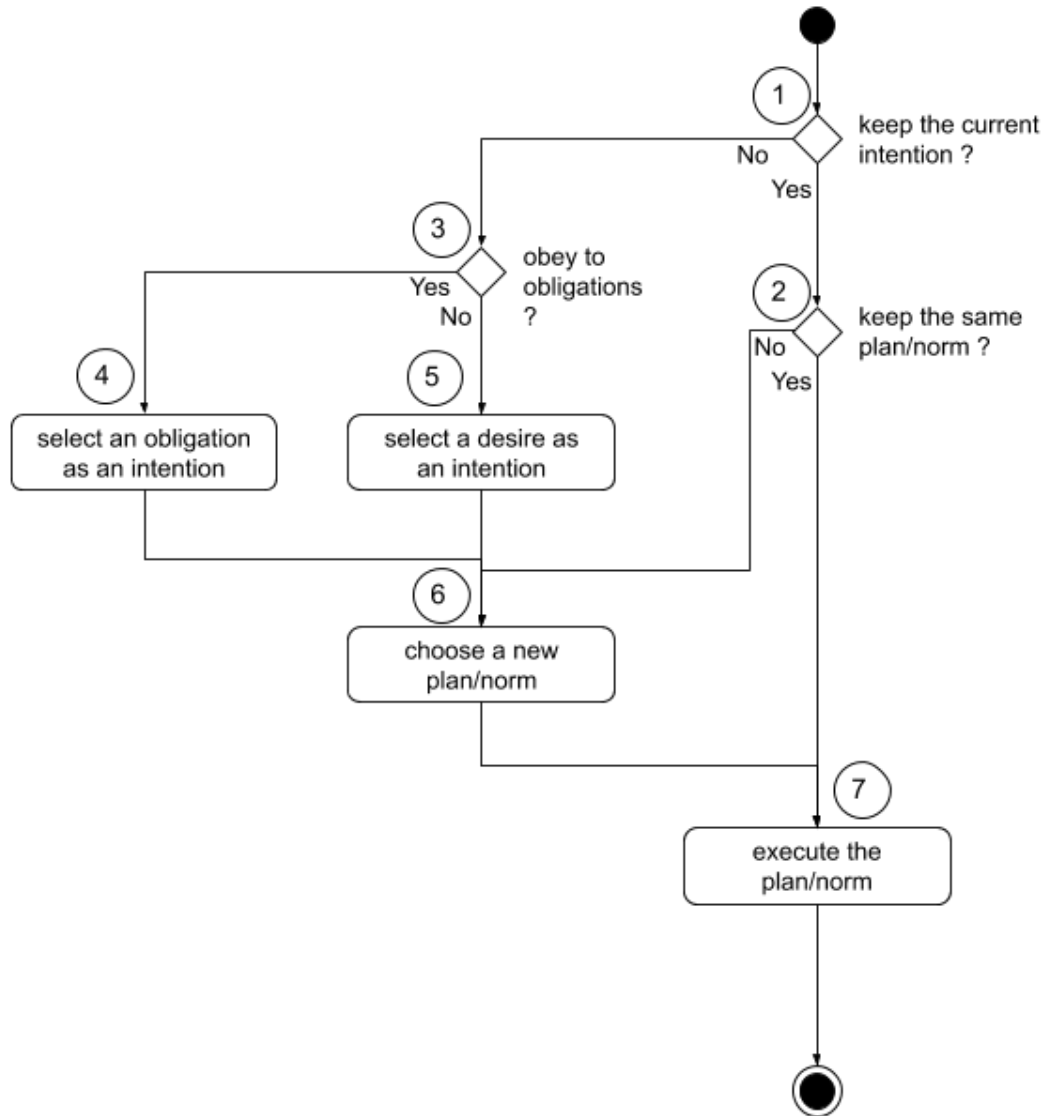


Figure 26.3: Diagram activity of cognitive engine (decision-making process) of the BEN architecture.

Steps 4, 5 and 6 do not have to be deterministic; they may be probabilistic. In this case, the priority value associated with obligations, desires, plans, and norms serves as a probability.

In GAML, a modeler may indicate the use of a probabilistic or deterministic cognitive engine with the variable `probabilistic_choice`, as shown in the example code below:

```
species miner control: simple_bdi {
  ...
  bool probabilistic_choice <- true;
  ...
}
```

Defining plans

The modeler needs to define action plans which are used by the cognitive engine, as explained earlier. These plans are a set of behaviors executed in a certain context in response to an intention. In BEN, a plan owned by agent i is represented by $\mathbf{Pli}(\mathbf{Int}, \mathbf{Cont}, \mathbf{Pr}, \mathbf{B})$ with:

- **Pi**: the name of the plan.
- **Int**: the intention triggering this plan.
- **Cont**: the context in which this plan may be applied.
- **Pr**: a priority value used to choose between multiple plans relevant at the same time. If two plans are relevant to the same priority, one is chosen at random.
- **B**: the behavior, as a sequence of instructions, to execute if the plan is chosen by the agent.

The context of a plan is a particular state of the world in which this plan should be considered by the agent making a decision. This feature enables to define multiple plans answering the same intention but activated in various contexts.

Below is an example for the definition of two plans answering the same intention in different contexts in GAML:

```
species miner control: simple_bdi skills: [moving]{
  ...
  plan evacuationFast intention: in_shelter emotion:
  fearConfirmed priority:2 {
```

```

    color <- #yellow;
    speed <- 60 #km/#h;
    if (target = nil or noTarget) {
        target <- (shelter with_min_of (each.location
distance_to location)).location;
        noTarget <- false;
    } else {
        do goto target: target on: road_network move_weights:
current_weights recompute_path: false;
        if (target = location) {
            do die;
        }
    }
}

plan evacuation intention: in_shelter finished_when:
has_emotion(fearConfirmed){
    color <-#darkred;
    if (target = nil or noTarget) {
        target <- (shelter with_min_of (each.location
distance_to location)).location;
        noTarget <- false;
    } else {
        do goto target: target on: road_network move_weights:
current_weights recompute_path: false;
        if (target = location) {
            do die;
        }
    }
}
...
}

```

Defining norms

A normative engine may be used within the cognitive engine, as it has been explained above. This normative engine means choosing an obligation as the current in ... 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.

`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 behavior which may be disobeyed is formally represented by a `norm` possessed by `agent _i_ **No`
`_i(Int,Cont,Ob,Pr,B,Vi)** with:`

```

* **No**: the name of the norm.
* **Int**: the intention which triggers this norm.
* **Cont**: the context in which this norm can be applied.
* **Ob**: an obedience value that serves as a threshold to
  determine whether or not the norm is applied depending on
  the agent's obedience value (if the agent's value is above
  the threshold, the norm may be executed).
* **Pr**: a priority value used to choose between multiple
  norms applicable at the same time.
* **B**: the behavior, as a sequence of instructions, to
  execute if the norm is followed by the agent.
* **Vi**: a violation time indicating how long the norm is
  considered violated once it has been violated.

```

In GAML, a norm is defined as follows:

```

species miner control: simple_bdi { ... //this first norm answer an intention
coming from an obligation norm doingJob obligation:has_gold finished_when: has_
belief(has_gold) threshold:thresholdObligation{ if (target = nil) { do add_subin-
tention(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_informa-
tion threshold:thresholdNorm instantaneous: true{ list my_friends <- list((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 of reducing their lifetime. When the lifetime is null, the mental **state** is removed from its base. The ****degradation of emotions**** consists of reducing the intensity of each **emotion** stored by its decay **value**. When the intensity of an **emotion** is null, the **emotion** is removed from the emotional base.

In GAML, **if** a mental **state** has a lifetime **value** or **if** an **emotion** has an intensity and a decay **value**, this degradation process is done automatically.

Finally, ****the status of each norm is updated**** to indicate **if** the norm was activated or not (**if** the context was **right** or wrong) and **if** it was violated or not (the norm was activated but the **agent** disobeyed it). Also, a norm can be violated for a certain time which is updated and **if** it becomes null, the norm is not violated anymore.

These last steps enable the **agent's** behavior's components to automatically evolve through time, leading the **agents** to forget a piece of knowledge after a certain amount of time, creating dynamics in their behavior.

Conclusion

The BEN architecture is already implemented in GAMA and may be accessed by adding the **simple_bdi control** architecture to

```

    the definition of a species.

A tutorial may be found with the [BDI Tutorial](BDIAgents).

# Advanced Driving Skill
[//]: # (keyword|concept_transport)
[//]: # (keyword|concept_skill)
[//]: # (keyword|skill_driving)

This page aims at presenting how to use the advanced driving
skill in models.

The use of the advanced driving skill requires to use 3 skills
:

* **Advanced driving skill**: dedicated to the definition of
  the driver species. It provides the driver agents with
  variables and actions allowing to move an agent on a graph
  network and to tune its behavior.
* **Road skill**: dedicated to the definition of roads. It
  provides the road agents with variables and actions
  allowing to registers agents on the road.
* **Road node skill**: dedicated to the definition of nodes.
  It provides the node agents with variables allowing to take
  into account the intersection of roads and the traffic
  signals.

## Table of contents

* [Advanced Driving Skill](#advanced-driving-skill)
  * [Structure of the network: road and road node skills](#
    structure-of-the-network-road-and-road-node-skills)
  * [Advanced driving skill](#advanced-driving-skill)
  * [Application example](#application-example)

## Structure of the network: road and road_node skills

```


The advanced driving `skill` is versatile enough to be usable with most of classic road GIS data, in particular, OSM data. We use a classic format for the roads and nodes. Each road is a polyline composed of road sections (segments). Each road has a `target` node and a `source` node. Each node knows all its input and output roads. A road is considered as directed. For bidirectional roads, 2 roads have to be defined corresponding to both directions. Each road will be the `**`linked_road`**` of the other. Note that for some GIS data, only one road is defined for bidirectional roads, and the nodes are not explicitly defined. In this case, it is very easy, using the GAML language, to create the reverse roads and the corresponding nodes (it only requires a few lines of GAML).

```
![Road structure in the Driving Skill](resources/images/
  recipes/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).

```
![Roads representation in the driving skill.](resources/images/
  recipes/roads.PNG)
```

The `**road skill**` (``skill_road``) 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 read-only variables:

```
* **`agents_on`**: list of list (of driver agents); for each
lane, the list of driver agents on the road.
* **`all_agents`**: list (of driver agents): the list of
agents on the road.
```

The `road node skill` (`skill_road_node``) provides the road node agents with several variables that will define the road node properties:

```
* **`roads_in`**: list of road agents; the list of road agents
that have this node for target node.
* **`roads_out`**: list of road agents; the list of road
agents that have this node for source node.
* **`stop`**: list of list of road agents; list of stop
signals, and for each stop signal, the list of concerned
roads.
* **`priority_roads`**: list of road agents: the list of
priority roads.
```

It provides as well the road agents with one read-only variable:

```
* **`block`**: map: key: driver agent, value: list of road
agents; the list of driver agents blocking the node, and
for each agent, the list of concerned roads.
```

Advanced driving skill

Each driver agent has a planned trajectory that consists of 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**` (``advanced_driving``) provides the driver agents with several variables that will define the car properties and the personality of the driver:

```
* **`final_target`**: point; final location that the agent
  wants to reach (its goal).
* **`vehicle_length`**: float; length of the vehicle.
* **`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_coef`**: float; coefficient for the
  security distance. The security distance will depend on the
  driver speed and on this coefficient.
* **`proba_lane_change_up`**: float; probability to change
  lane to an upper lane if necessary (and if possible).
* **`proba_lane_change_down`**: float; probability to change
  lane to a lower lane if necessary (and if possible).
* **`proba_use_linked_road`**: float; probability to take the
  reverse road if necessary (if there is a reverse road).
* **`proba_respect_priorities`**: float; probability to
  respect left/right (according to the driving side) priority
  at intersections.
* **`proba_respect_stops`**: list of float; probabilities to
  respect each type of stop signals (traffic light, stop sign
  ...).
* **`proba_block_node`**: float; probability to accept to
  block the intersecting roads to enter a new road.
```

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 stuck for several minutes behind a slow vehicle.

In addition, the advanced driving skill provides driver agents with several actions:

- * `**`compute_path`**`: arguments: a graph and a target node. This action computes from a graph the shortest path to reach a given node.
- * `**`drive`**`: no argument. This action moves the driver on its current path according to the traffic condition and the driver properties (vehicle properties and driver personality).

The ``drive`` 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](#) below).

```
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.

![Activity diagram describing the driver behavior.](resources/images/recipes/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).

```
![Activity diagram of driver behavior when stopped at an
  intersection.](resources/images/recipes/
  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 move (that will depend on the remaining time). For each road section, the agent first computes the maximal distance it can travel according to the remaining time and its speed. Then, the agent computes its security distance according to its speed and its `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.

```
![Activity diagram of the following action of the advanced
  driving skill.](resources/images/recipes/follow_driving.png
)
```

The computation of the maximal distance an agent can move on a road section consists of computing for each possible lane the maximal distance the agent can move. First, if there is a lower lane, the agent tests the probability to change its lane to a lower one. If it decides to test the lower lane, the agent computes the distance to the next vehicle on this lane and memorizes it. If this distance corresponds to the maximal distance it can travel, it chooses this lane; otherwise, it computes the distance to the next

vehicle on its current lane and memorizes it if it is higher than the current memorized maximal distance. Then if the memorized distance is lower than the maximal distance the agent can travel and if there is an upper lane, the agents test the probability to change its lane to an upper one. If it decides to test the upper lane, the agent computes the distance to the next vehicle on this lane and memorizes it if it is higher than the current memorized maximal distance. At last, if the memorized distance is still lower than the maximal distance it can travel if the agent is on the highest lane and if there is a reverse road, the agent tests the probability to use the reverse road (linked road). If it decides to use the reverse road, the agent computes the distance to the next vehicle on the lane 0 of this road and memorizes the distance if it is higher than the current memorized maximal distance.

```
![Activity diagram of the driver behavior to define its
  maximum distance to others.](resources/images/recipes/
  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.

```
![[Simple example of the driving skill.]](resources/images/
  recipes/sim_snapshot.png)
```

The following code shows the definition of `species` to represent the road infrastructure:

```
species road skills: [skill_road] { string oneway; }

species road_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 `road_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 `road_node`` `species` a `reflex` (behavior) called `dynamic`` that will be activated only for traffic `light` nodes and that will increment the `counter``


```

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 are 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 concerned 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(road_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` skill 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, 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 an 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;

[The complete code of the model with the data can be found here](resources/images/recipes/Rouentraffic.zip).

```

[//]: # (keyword|concept_date)
# Manipulate Dates

[//]: # (keyword|type_date)
[//]: # (keyword|concept_time)
## Managing Time in Models

If some models are based on an abstract time - only the number
of cycles is important - others are based on a real time.
To this purpose, GAMA provides some tools to manage time.

First, GAMA allows the modeler to define the duration of a
simulation step. It provides access to different time
variables. At last, since GAMA 1.7, it provides a date
variable type and some global variables allowing to use a
real calendar to manage time.

## Definition of the step and use of temporal unity values

GAMA provides three important [global variables to manage time
](GlobalSpecies#cycle):

* `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 - months - days
- hours - minutes - seconds format. In this presentation,
every month is considered as being composed of 30 days (the
different number of days of months are not taken into
account).

```

Concerning `step global` variable, the variable can be modified by the modeler. A classic way of doing it consists of reediting the variable in the `global` section:

```
global { float step <- 1 #hour; }
```

In this example, each simulation `step` will represent 1 hour. This time will be taken into account for `all` actions based on time (e.g. moving actions).

Note that the `value` of the ``step`` variable should be given in seconds. To facilitate the definition of the `step value` and of `all` expressions based on time, GAMA provides [different built-in constant variables accessible with the `"`#`"` symbol](UnitsAndConstants#time-units):

```
* `#s` : second - 1 second
* `#mn` : minute - 60 seconds
* `#hour` : hour - 60 minutes - 3600 seconds
* `#day` : day - 24 hours - 86400 seconds
* `#week` : week - 7 days - 604800 seconds
* `#month` : month - 30 days - 2592000 seconds
* `#year` : year - 12 month - 3.1104E7 seconds
```

The `date` variable `type` and the use of a real calendar
Since GAMA 1.7, it is possible to use a real calendar to manage the time. For that, the modeler has only to define the starting `date` of the simulation. This variable is of `type`date`` which allows him/her to represent a `date` and time.

A `date` variable has several `attributes`:

```
* `year` (int): the year component of the date
* `month` (int): the month component of the date
* `day` (int): the day component of the date
* `hour` (int): the hour component of the date
* `minute` (int): the minute component of the date
* `second` (int): the second component of the date
* `day_of_week` (int): the day of the week
```

```
* `week_of_year` (int): the week of the year
```

Several ways can be used to define a `date`. The simplest one consists in `using` a `list` of `int` values: [year, month of the year, day of the month, hour of the day, minute of the hour, second of the minute]

```
date my_date <- date([2010,3,23,17,30,10]); // the 23th of March 2010, at 17:30:10
```

Another way consists in `using` a `string` with the good format. The following one is perhaps the most complete, with year, month, day, hour, minute, second and also the time zone.

```
date my_date <- date("2010-3-23T17:30:10+07:00");
```

But the following ones can also be used:

```
// without time zone: my_date3 <- date("2010-03-23 17:30:10"); //Dates (without
time) my_date3 <- date("20100323"); my_date3 <- date("2010-03-23"); // Dates
using some patterns: my_date3 <- date("03 23 2010","MM dd yyyy"); my_date3 <-
date("01 23 20","HH mm ss");
```

Note that the current (real) `date` can be accessed through the ``#now`` built-in variable (variable of `type date`).

In addition, GAMA provides different useful operators working on dates. For instance, it is possible to compute the duration in seconds between 2 dates using the `"`-`"` operator. The result is given in seconds:

```
float d <- starting_date - my_date;
```

It is also possible to `add` or subtract a duration (in seconds) to a `date`:

```
write "my_date + 10:" + (my_date + 10); write "my_date - 10:" + (my_date -
10);
```

At last, it is possible to `add` or subtract a duration (in years, months, weeks, days, hours, minutes, seconds) to a `date`:

```
write "my_date add_years 1:" + (my_date add_years 1); write "my_date add_ -
months 1:" + (my_date add_months 1); write "my_date add_weeks 1:" + (my_date
add_weeks 1); write "my_date add_days 1:" + (my_date add_days 1); write "my_ -
date add_hours 1:" + (my_date add_hours 1); write "my_date add_minutes
1:" + (my_date add_minutes 1); write "my_date add_seconds 1:" + (my_date
add_seconds 1);
```

```
write "my_date subtract_years 1:" + (my_date subtract_years 1); write "my_date
subtract_months 1:" + (my_date subtract_months 1); write "my_date subtract_ -
weeks 1:" + (my_date subtract_weeks 1); write "my_date subtract_days 1:" +
(my_date subtract_days 1); write "my_date subtract_hours 1:" + (my_date sub-
tract_hours 1); write "my_date subtract_minutes 1:" + (my_date subtract_minutes
1); write "my_date subtract_seconds 1:" + (my_date subtract_seconds 1);
```

```
## Date variables in the model
```

For the modelers, two **global date** variables are available:

- * ``starting_date``: **date** considered as the beginning of the simulation (by **default** the starting **date** is ``1970-01-01 07:00:00``).
- * ``current_date``: **current date** of the simulation.

Defining a **value** of the `starting_date` allows to change the normal time management of the simulation by a more realistic one (**using** a calendar):

```
global { date starting_date <- date([1979,12,17,19,45,10]); }
```

When a **value** is set to this variable, the ``current_date`` variable is automatically initialized **with** the same **value**. However, at each simulation **step**, the ``current_date`` variable is incremented by the ``step`` variable. The **value** of the ``current_date`` will **replace** the **value** of the time variable in the top **left** green panel.

Note that you have to be careful **when** a real calendar is used, the built-in constants ``#month`` and ``#year`` should not be used as there are not consistent **with** the calendar (where month can be composed of 28, 29, 30 or 31 days).

```
[//]: # (startConcept|light)
[//]: # (keyword|concept_3d)
[//]: # (keyword|concept_light)
# 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](#light-generalities)
* [Default light](#default-light)
* [Custom lights](#custom-lights)
```

```
## Light generalities
```

Before going deep into the code, here is a quick explanation about how light works in OpenGL.

First of all, you need to know that there are 3 types of lights you can manipulate: the **ambient light**, the **diffuse light** and the **specular light**. Each "light" in OpenGL is in fact composed of those 3 types of lights.

```
### Ambient light
```


The **ambient light** is the light of your world without any lighting. If a face of a cube is not stricken by the light rays, for instance, this face will appear totally black if there is no ambient light. To make your world more realistic, it is better to have ambient light. Ambient light has then no position or direction. It is equally distributed to all the objects of your scene.

Here is an example of our GAML scene using only ambient light (color red) (see below [how to define ambient light in GAML](ManipulateLight#ambient-light-1)):

```
![Example of a scene with a red ambient light.](resources/
images/lightRecipes/ambient_light.png)
```

Diffuse light

The **diffuse light** can be seen as the light rays: if a face of a cube is stricken by the diffuse light, it will take the color of this diffuse light. You have to know that the more perpendicular the face of your object will be to the light ray, the more lightened the face will be.

A diffuse light has then a direction. It can have also a position.

You have 2 categories of diffuse light: the **positional lights**, and the **directional lights**.

Positional lights

Those lights have a position in your world. It is the case of **point lights** and **spot lights**.

* **Point lights***

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) :

```
![[Scene with only a red point light.]](resources/images/
  lightRecipes/point_light.png)

* **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) :

![[Scene with only a red spot light.]](resources/images/
  lightRecipes/spot_light.png)
```

Positional lights, as they have a position, can also have an attenuation according to the distance between the light source and the object. The value of positional lights are computed with the following formula:

$$\text{diffuse_light} = \text{diffuse_light} * (1 / (1 + \text{constante_attenuation} + \text{linear_attenuation} * d + \text{quadratic_attenuation} * d^2))$$

By changing those 3 values (constante_attenuation, linear_attenuation and quadratic_attenuation), you can control the way light is diffused over your world (if your world is "foggy" for instance, you may turn your linear and quadratic attenuation on). Note that by default, all those attenuations are equal to 0.

```
Here is an example of our GAML scene using only diffusion
  light, with a point light with linear attenuation (color
  red, the light source is displayed as a red sphere):

![[Scene with only diffusion light and red point light with
  linear attenuation.]](resources/images/lightRecipes/
  point_light_with_attenuation.png)
```

```
#### Directional lights
```

Directional lights have no real "position": they only have a direction. A directional light will strike all the objects of your world in the same direction. An example of directional light you have in the real world would be the light of the sun: the sun is so far away from us that you can consider that the rays have the same direction and the same intensity wherever they strike.

Since there is no position for directional lights, there is no attenuation either.

Here is an example of our GAML scene using only diffusion light, with a directional light (color red) :

```
![Scene with a red directional light.](resources/images/lightRecipes/direction_light.png)
```

```
### Specular light
```

This is a more advanced concept, giving an aspect a little bit "shiny" to the objects stricken by the specular light. It is used to simulate the interaction between the light and a special material (ex: wood, steel, rubber...).

This specular light is not implemented yet in GAMA, only the two others are.

```
## Default light
```

In your OpenGL display, without specifying any light, you will have only one light, with those following properties :

Those values have been chosen in order to have the same visual effect in both OpenGL and java2D displays, when you display 2D objects, and also to have a nice "3D effect" when using the OpenGL displays. We chose the following setting by default:

```
* The ambient light value: rgb(127,127,127,255)
* diffuse light value: rgb(127,127,127,255)
* type of light: direction
* direction of the light: (0.5,0.5,-1);
```

Here is an example of our GAML scene using the default light:

```
![Scene with the default light.](resources/images/lightRecipes/default_light.png)
```

```
## Custom lights
```

In your OpenGL `display`, you can create several lights, giving them the properties you want.

```
[//]: # (keyword|statement_light)
```

In order to add lights, or modifying the existing lights, you have to use the statement ``light`` inside your ``display`` scope:

```
experiment my_experiment type:gui { output { display "my_display" type:opengl { light "my_light"; } } }
```

A name has to be declared for the light. Through this facet, you can specify which light you want.

Once you are manipulating a light through the ``light`` statement, the light is turned on. To switch off the light, you have to add the facet ``active``, and turn it to ``false``.

The light you are declaring through the ``light`` statement is, in fact, a "diffuse" light. You can specify the color of the diffuse light through the facet ``intensity`` (by default, the color will be turned to white).

Another very important facet is the ``type`` facet. This facet accepts a value among ``#direction``, ``#point`` and ``#spot``.

```
### Ambient light
```

The ambient light can be set when declaring a light, using the `#ambient` constant, through the facet ``intensity``:

```
experiment my_experiment type:gui { output { display "my_display" type:opengl { light #ambient intensity: 100; } } }
```

`_Note for developers_`: Note that this ambient light is set to the `GL_LIGHT0`. This `GL_LIGHT0` only contains an ambient light, and no either diffuse nor specular light.

Declaring direction light

A direction light, as explained in the first part, is a light without any position. Instead of the facet ``position``, you will use the facet ``direction``, giving a 3D vector.

Example of implementation:

```
light "my_direction_light" type: #direction direction: {1,1,1} intensity: #red;
```

Declaring point light

A point light will need a facet ``position``, in order to give the position of the light source.

Example of implementation of a basic point light:

```
light "my_point_light" type: #point location: {10,20,10} intensity: #red;
```

You can add, if you want, a custom attenuation of the light, through the facets ``linear_attenuation`` or ``quadratic_attenuation``.

Example of implementation of a point light with attenuation :

```
light "my_point_light" type: #point location: {10,20,10} intensity: #red linear_attenuation: 0.1;
```

Declaring spot light

A spot light will need the facet ``position`` (a spot light is a positional light) and the facet ``direction``. A spot light will also need a special facet ``spot_angle`` to determine the angle of the spot (by default, this value is set to 45 degree).

Example of implementation of a basic spot light:

```
light "my_spot_light" type: #spot location: {0,0,100} direction: {0.5,0.5,-1} intensity:
#red angle: 20;
```

Same as for `point light`, you can specify an attenuation for a `spot light`.

Example of implementation of a `spot light` with attenuation:

```
light "my_spot_light" type: #spot location: {0,0,100} direction: {0.5,0.5,-1} inten-
sity: #red angle: 30 linear_attenuation: 0.1;
```

Note that when you are working with lights, you can display your lights through the facet ``show`` (of ``light``) to help you to implement your model. The three types of lights are displayed differently:

- * The `**point** light` is represented by a sphere with the color of the `diffuse light` you specified, in the position of your `light source`.
- * The `**spot** light` is represented by a cone with the color of the `diffuse light` you specified, in the position of your `light source`, the orientation of your `light source`. The size of the base of the cone will depend on the angle you specified.
- * The `**direction** light`, as it has no real position, is represented with arrows a bit above the world, with the direction of your `direction light`, and the color of the `diffuse light` you specified.

```
![Scene with direction, spot and point lights.](resources/
images/lightRecipes/draw_light.png)
```

`_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`.

```
[//]: # (endConcept|light)
```

```
# Using Comodel
```

```
## Introduction
In the trend of developing a complex system of multi-
disciplinary, composing and coupling models are days by
days becoming the most attractive research objectives.
GAMA is supporting the co-modeling and co-simulation which are
supposed to be a common coupling infrastructure.

## Example of a Comodel

A Comodel is a model, especially an agent-based model,
composed of several sub-models, called micro-models. A
comodel itself could be also a micro-model of another
comodel. From the point of view of a micro-model, the
comodel is called a macro-model.

A micro-model must be imported, instantiated, and life-
controlled by a macro-model.

![GAMA co-modeling architecture.](resources/images/comodel/
concepts.png)

## Why and when can we use Comodel?

Co-models can definitely be very useful when the whole model
can be decomposed in several sub-models, each of them
representing, in general, a dynamics of the whole model,
and that interact through some entities of the model. In
particular, it allows several modelers to develop the part
of the model dedicated to their expertise field, to test it
extensively, before integrating it inside the whole model
(where integration tests should not be omitted!).

## Use of Comodel in a GAML model

The GAML language has evolved by extending the import section.
The old importation told the compiler to merge all
imported elements into as one model, but the new one allows
```

modelers to keep the elements coming from imported models separately from the caller model.

Definition of a micro-model

Defining a micro-model of comodel is to import an existing model with an alias name. The syntax is:

import as

The identifier is then become the new name of the micro-model.

As an example taken from the model library, we can write:

import "Prey Predator Adapter.gaml" as Organism

Instantiation of a micro-model

After the importation and giving an identifier, micro-model must be explicitly instantiated. It could be done by the `create`` statement.

create . [optional parameter];

The `<experiment name>` is an experiment inside micro-model. This syntax will generate some experiment agents and attach an implicit simulation.

Note: The creation of several instances is not multi-simulation, but multi-experiment. Modelers could create an experiment with multi-simulation by explicitly do the `init` inside the experiment scope.

As an example taken from the model library, we can write:

```
global { init { //instantiate three instant of micro-model PreyPredator create Organism.Simple number: 3 with: [shape::square(100), preyinit::10, predatorinit::1] ; } }
```

Control micro-model life-cycle

A `micro-model` can be controlled as any normal `agent` by asking the corresponding identifier, and also be destroyed by the ``do die;`` statement. And it can be recreated any time we need.

```
ask (. at ). simulation { ... }
```

More generally, to schedule all the created simulations, we can do:

```
reflex simulate_micro_models { // ask all simulation do their job ask (Organism.Simple collect each.simulation) { do step; } }
```

```
## Visualization of the micro-model
```

The `micro-model species` could display in `comodel` with the support of `agent` layer

```
agents "name of layer" value: (. at ).;
```

As an example:

```
display "Comodel display" { agents "agentprey" value: (Organism.Simple accumulate each.get_pre()); agents "agentpredator" value: (Organism.Simple accumulate each.get_predator()); }
```

```
## More details
```

```
## Example of the comodel
```

The following illustrations are taken from the `model` library provided with the GAMA platform.

```
### Urbanization model with a Traffic model
```

```
![Co-modeling example: urbanization model with a Traffic model .](resources/images/comodel/comodel_urban_traffic.png)
```

```
### Flood model with Evacuation model
```

The aim of this `model` is to couple the two existing models:
Flood Simulation and Evacuation.

```
Toy Models/Evacuation/models/continuous_move.gaml
```

```
![Co-modeling example: the evacuation model.](resources/images/  
comodel/continuous_move_model_display.png)
```

```
Toy Models/Flood Simulation/models/Hydrological Model.gaml
```

```
![Co-modeling example: the flood model.](resources/images/  
comodel/hydro_model_display.png)
```

The `comodel` explores the effect of a flood on an evacuation
plan:

```
![Co-modeling example: coupling of the flood and evacuation  
models.](resources/images/comodel/  
comodel_disp_Flood_Evacuation.png)
```

Simulation results:

```
![Co-modeling example: some simulation results.](resources/  
images/comodel/comodel_Flood_Evacuation.png)
```

```
[//]: # (startConcept|use_saveSimulation)
```

```
[//]: # (keyword|concept_save)
```

```
[//]: # (keyword|concept_simulation)
```

```
# 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

[//]: # (keyword|concept_network)
[//]: # (startConcept|network)

# Using network

## Introduction

GAMA provides features to allow agents to communicate with other agents (and other applications) through network and to exchange messages of various types (from simple number to agents). To this purpose, the `network` skill should be used on agents intending to use these capabilities.

Notice that in this communication, roles are asymmetric: the simulations should contain a server and some clients to communicate. Message exchanges are made between agents through this server. 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](https://github.com/gama-
platform/gama/wiki/BuiltInSkills#network).

```

```
## 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)  
* **`port`**: traditionally the port `1883` is used (when  
  ActiveMQ is used as the server application)  
* **`admin`** and **`password`**: traditionally the default  
  login and password are "admin" (when ActiveMQ is used as  
  the server application)
```

Note: if no connection information is provided with the MQTT protocol (no `port`), then GAMA connects to an MQTT server provided by the GAMA community (for test purpose only!).

```
## Sending messages
```

To send any message, the agent has to use the `send` action:

```
do send to: "server" contents: name + " " + cycle + " sent to server";
```

The network skill in GAMA allows the modeler to send simple string messages between agents but also to send more complex objects (and in particular agents). In this case, the use of the MQTT protocol is highly recommended.

```
do send to: "receiver" contents: (9 among NetworkingAgent);
```

```
## Receiving messages
```

The messages sent by other `agents` are received in the ``mailbox`` attribute of each `agent`. So to get its new `message`, the `agent` has simply to check whether it has a new `message` (with action ``has_more_message()``) and fetch it (that gets it and remove it from the mailing box) with the action ``fetch_message()``.

```
reflex fetch when: has_more_message() { message mess <- fetch_message(); write
name + " fecth this message: " + mess.contents;
}
```

Note that `when` an `agent` is received, the fetch of the `message` will recreate the `agent` in the current simulation.

Alternatively, the ``mailbox`` attribute can be directly accessed (notice that the ``mailbox`` is a `list` of messages):

```
reflex receive {
if (length(mailbox) > 0) { write mailbox; } }
```

```
## Broadcasting a message to all the agents' members of a
given group
```

Each time an `agent` creates a connection to another `agent` as a client, a way to communicate with it is stored in the ``network_groups`` attribute.

So an `agent` can use this attribute to broadcast messages to all the `agents` with 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 reference page](BuiltInSkills#network).
* example models can be found in the GAMA model library, in: `
Plugin models > Network`.
```

```
# 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](PredatorPrey_step1).
Headless mode is documented [here](Headless), 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 command, in order to check the logs **if** an **error** appears: ``#rm -rf $passWork``

Setting the **experiment file**

Headless **mode** uses a XML **file** to describe the execution **plan** of a **model**. An example is given in the [headless mode documentation page](Headless).

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](PredatorPrey_step4), located in `~/GAMA/plugins/msi.gama.models_1.7.0.XXXXXXXXXXXXXX/models/Tutorials/Predator Prey/models`` (with `XXXXXXXXXXXXX` 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 be 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`. These images is what you would have seen `if` you had `run` the `model` in the traditional `GUI mode`.

```
## Execution and results
```

Our new version of the `experiment` file is ready :

```

<?xml version="1.0" encoding="UTF-8"?>
<Experiment_plan>
  <Simulation id="2" sourcePath="/absolute/path/to/your/
model/file/Model 04.gaml" finalStep="1000" experiment="
prey_predator">
  <Parameters>
    <Parameter name="Initial number of preys: " type="
INT" value="621" />
    <Parameter name="Prey max energy: " type="FLOAT"
value="1.0" />
  </Parameters>
  <Outputs>
    <Output id="1" name="main_display" framerate="10"

```

```
    />
      <Output id="2" name="Number of preys" framerate="1
" />
    </Outputs>
  </Simulation>
</Experiment_plan>
```

Execution

We have to launch the `gama-headless.sh` script and provide two arguments : the **experiment file** we just completed and the **path** of a directory where the results will be written.

**** Warning **** In this example ,we are lazy and define the **source path** as the absolute **path** to the **model** we want to execute. **If** you want to use a relative **path**, note that it has to be define relatively to the **location** of your **** ExperimentFile.xml location **** (and the **location** where you launched the script)

In a terminal, **position** yourself in the headless directory :
`~/GAMA/headless/'.

Then **type** the following command :

```
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/tutoGamaHeadless/./plugins/msi.gama.models_1
This may be a relative path mistake; try with absolute path.
```

```
java.lang.NumberFormatException: For input string: "1.0"
This may be a problem of type declaration in the parameter
  section.

## Going further

### Experiments of several simulation

You can launch several simulation by replicating the
  simulation declaration in your ExperimentFile.xml and
  varying the values of the parameters.
Since you will have to edit the experiment file by hand, you
  should do that only for a reasonable number of simulations
  (e.g. <10 )

### Design of experiments plans

For more systematic parameter values samples, you should turn
  towards a more adapted tool such as GAMAR, to generate a `
  ExperimentFile.xml` with a huge number of simulations.

# 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. A tutorial
  is available [here](G_GraphicalEditorTutorial).

![images/graphical_editor/gm_predator_preym.png](resources/
  images/graphicalEditor/gm_predator_preym.png)

## Table of contents
```

```

* [The Graphical Editor](#the-graphical-editor)
  * [Installing the graphical editor](#installing-the-graphical-editor)
  * [Creating a first model](#creating-a-first-model)
  * [Status of models in editors](#status-of-models-in-editors)
  * [Diagram definition framework](#diagram-definition-framework)
  * [Features](#features)
    * [agents](#agents)
      * [species](#species)
      * [grid](#grid)
      * [Inheriting link](#inheriting-link)
      * [world](#world)
    * [agent features](#agent-features)
      * [action](#action)
      * [reflex](#reflex)
      * [aspect](#aspect)
    * [experiment](#experiment)
      * [GUI experiment](#gui-experiment)
      * [display](#display)
      * [batch experiment](#batch-experiment)
    * [BDI Architecture](#BDI-Architecture)
      * [plan](#plan)
      * [rule](#rule)
      * [perception](#perception)
    * [Finite State Machine](#Finite-State-Machine-Architecture)
      * [state](#state)
    * [Tasked-based Architecture](#Task-based-Architecture)
      * [task](#task)
  * [Pictogram color modification](#pictogram-color-modification)
  * [GAML Model generation](#gaml-model-generation)

## Installing the graphical editor
Using the graphical editor requires to install the graphical modeling plug-in. See [here](InstallingPlugins) for

```

information about plug-ins and their installation.

The graphical editor plug-in is called ****Graphical_modeling**** and is directly available from the GAMA update site ****http://updates.gama-platform.org/graphical_modeling/1.8.2****

```
![install](resources/images/graphicalEditor/
installing_graphical_editor.JPG)
```

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>"

```
![images/graphical_editor/newDiagram.png](resources/images/
graphicalEditor/newDiagram.png)
```

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

```
![images/graphical_editor/modeldiagramNew.png](resources/
images/graphicalEditor/modeldiagramNew.png)
```

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](ValidationOfModels) 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:

```
![images/graphical_editor/framework.png](resources/images/graphicalEditor/framework.png)
```

```
## Features

### agents
#### species

![images/graphical_editor/species.png](resources/images/graphicalEditor/species.png)

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: -

![images/graphical_editor/Frame_Speciesdef1.png](resources/images/graphicalEditor/Frame_Speciesdef1.png)

#### grid

![images/graphical_editor/grid.png](resources/images/graphicalEditor/grid.png)

The grid feature allows the modeler to define a [species](ManipulateBasicSpecies) with a [grid topology](GridSpecies). A grid is always a micro-species of another species.

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

![images/graphical_editor/Frame_grid.png](resources/images/graphicalEditor/Frame_grid.png)

#### 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)

![[images/graphical_editor/inheriting_link.png]](resources/
  images/graphicalEditor/inheriting_link.png)

#### world

![[images/graphical_editor/world.png]](resources/images/
  graphicalEditor/world.png)

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.

![[images/graphical_editor/Frame_world.png]](resources/images/
  graphicalEditor/Frame_world.png)

### agent features

#### action

![[images/graphical_editor/action.png]](resources/images/
  graphicalEditor/action.png)

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

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

![[images/graphical_editor/Frame_action.png]](resources/images/
  graphicalEditor/Frame_action.png)

#### reflex

![[images/graphical_editor/reflex.png]](resources/images/

```



```
graphicalEditor/reflex.png)
```

The `reflex` feature allows the modeler to define a `reflex` for a `species`.

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

```
![images/graphical_editor/Frame_reflex.png](resources/images/
graphicalEditor/Frame_reflex.png)
```

```
#### aspect
```

```
![images/graphical_editor/aspect.png](resources/images/
graphicalEditor/aspect.png)
```

The `aspect` feature allows the modeler to define an `aspect` for a `species`.

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

```
![images/graphical_editor/Frame_aspect.png](resources/images/
graphicalEditor/Frame_aspect.png)
```

```
![images/graphical_editor/Frame_Aspect_layer.png](resources/
images/graphicalEditor/Frame_Aspect_layer.png)
```

```
#### equation
```

```
![images/graphical_editor/equation.png](resources/images/
graphicalEditor/equation.png)
```

The `equation` feature allows the modeler to define an `equation` for a `species`.

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

```
### experiment
```

```
#### GUI experiment
```

```
![images/graphical_editor/guiXP.png](resources/images/
graphicalEditor/guiXP.png)
```

The GUI **Experiment** feature allows the modeler to define a GUI **experiment**.

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

```
![images/graphical_editor/Frame_Experiment.png](resources/
images/graphicalEditor/Frame_Experiment.png)
```

```
#### display
```

```
![images/graphical_editor/display.png](resources/images/
graphicalEditor/display.png)
```

The **display** feature allows the modeler to define a **display**.

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

```
![images/graphical_editor/Frame_display.png](resources/images/
graphicalEditor/Frame_display.png)
```

```
![images/graphical_editor/Frame_layer_display.png](resources/
images/graphicalEditor/Frame_layer_display.png)
```

```
#### batch experiment
```

```
![images/graphical_editor/batchxp.png](resources/images/
graphicalEditor/batchxp.png)
```

The Batch **Experiment** feature allows the modeler to define a **Batch experiment**.

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

```
### BDI Architecture
#### Plan

![images/graphical_editor/plan.png](resources/images/
graphicalEditor/plan.png)

The Plan feature allows the modeler to define a plan for a BDI
species, i.e. a sequence of statements that will be
executed in order to fulfill a particular intention.

* **source**:: a species with a BDI architecture
* **target**:: -
S
#### Rule

![images/graphical_editor/rule.png](resources/images/
graphicalEditor/rule.png)

The Rule feature allows the modeler to define a rule for a BDI
species, i.e. a function executed at each iteration to
infer new desires or beliefs from the agent's current
beliefs and desires.

* **source**:: a species with a BDI architecture
* **target**:: -

#### Perception

![images/graphical_editor/perception.png](resources/images/
graphicalEditor/perception.png)

The Perception feature allows the modeler to define a
perception for a BDI species, i.e. a function executed at
each iteration that updates the agent's Belief base
according to the agent perception.

* **source**:: a species with a BDI architecture
* **target**:: -
```

```
### Finite State Machine Architecture
#### State

![images/graphical_editor/state.png](resources/images/graphicalEditor/state.png)

The State feature allows the modeler to define a state for a
FSM species, i.e. sequence of statements that will be
executed if the agent is in this state (an agent has a
unique state at a time).

* **source**: a species with a finite state machine
  architecture
* **target**: -

### Task-based Architecture
#### Task

![images/graphical_editor/task.png](resources/images/graphicalEditor/task.png)

The Task feature allows the modeler to define a task for a
Tasked-based species, i.e. sequence of statements that can
be executed, at each time step, by the agent. If an agent
owns several tasks, the scheduler chooses a task to execute
based on its current priority weight value.

* **source**: a species with a task-based architecture
* **target**: -

## Pictogram color modification
It is possible to change the color of a pictogram.

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

```
## GAML Model generation
It is possible to automatically generate a Gaml model from a
diagram.

* Right-click on the graphical framework (where the diagram
is defined), then select the "Generate Gaml model".
A new GAML model with the same name as the diagram is created
(and open).

# Using Git from GAMA to version and share models

## Install the Git client [Tested on the GAMA 1.8.2]

The Git client for GAMA needs to be installed as an external
plugin.
1. Help > Install new plugins...
2. Add the following address in the text field "Work with": `
https://download.eclipse.org/egit/updates`. (press Enter
key)
3. In the available plugins to install, choose `Git
integration for Eclipse` > `Git integration for Eclipse`
4. Click on the Next button and follow the instructions (GAMA
will be relaunched).

## Open the Git view

To use Git in GAMA select Views -> Other... -> Show View ->
Other...

In the Show view window that appears select Git -> Git
Repositories and click on *Open*.

![Show View Window](resources/images/recipes/gitWithGama/
ShowViewWindow.png)

## Create a Local Repository
```

With Git you can easily create local repositories to version your work locally. First, you have to create a GAMA project (e.g `***GitNewProject***`) that you want to share via your local repository.

After you have created your GAMA project, go to the Git Repository view and click on `*Create` a new local Git repository*.

```
![Create New Local Git Repository](resources/images/recipes/gitWithGama/CreateLocalGitRepository.png)
```

In the following window specify the directory for the new repository (select the folder of the created GAMA project - `***GitNewProject***` -), through the button `Browse...`

```
![Select folder new local Repository](resources/images/recipes/gitWithGama/SelectRepositoryFolder.png)
```

then hit the `Create` button.

```
![Create Button](resources/images/recipes/gitWithGama/CreateRepositoryButton.png)
```

Now your local repository is created, you can add models and files into your GAMA project. As you selected the folder of the new created GAMA Project, the repository will not be empty. So, it will be initialized with all the folders and files of the GAMA project. Note the changed icons: the project node will have a repository icon, the child nodes will have an icon with a question mark.

```
![Changed icons](resources/images/recipes/gitWithGama/ChangedIcons.png)
```

Before you can commit the files to your repository, you need to add them. Simply right click the shared project's node and navigate to `Team -> Add to Index`.

```
![Add Ignore Commit from Menu](resources/images/recipes/
```

```
gitWithGama/AddIgnoreCommit.png)
```

After this operation, the question mark should change to a plus symbol.

```
![Icons Changed after add](resources/images/recipes/
gitWithGama/ChangeIconsAfterAddGit.png)
```

To **set** certain folders or files to be ignored by Git, **right** click them and **select** Team -> Ignore. The ignored items will be stored in a **file** called `.gitignore`, which you should **add** to the repository.

```
## Commit
```

Now you can modify files in your project, **save** changes made in your workspace to your repository and commit them. You can **do** commit the project by **right** clicking the project node and selecting Team -> Commit... **from** the context menu. In the Commit wizard, **all** files should be selected automatically. **Enter** a commit **message** and hit the Commit button.

```
![Icons Changed after add](resources/images/recipes/
gitWithGama/FirstCommitLocalRepo.png)
```

If the commit was successful, the plus symbols will have turned into repository icons.

```
![Icons Changed after commit](resources/images/recipes/
gitWithGama/ChangedIconsAfterCommit.png)
```

After changing files in your project, a ">" sign will appear **right** after the icon, telling you the **status** of these files is dirty. Any **parent** folder of this **file** will be marked as dirty as well.

```
![Changes to commit](resources/images/recipes/gitWithGama/
gitChangesToCommit.png)
```

If you want to commit the changes to your repository, right click the project (or the files you want to commit) and select Team -> Commit... . Enter a commit message and click Commit to commit the selected files to your repository.

Add Files

To add a new file to the repository, you need to create it in your shared GAMA project first. Then, the new file will appear with a question mark.

```
![New file added to project](resources/images/recipes/gitWithGama/AddNewFileGit.png)
```

Right click it and navigate to Team -> Add to Index. The question mark will turn into a plus symbol and the file will be tracked by Git, but it is not yet committed. In the next commit, the file will be added to the repository and the plus symbol will turn into a repository icon.

```
![Commit new added file](resources/images/recipes/gitWithGama/AddedFileCommitGit.png)
```

Revert Changes

If you want to revert any changes, there are two options. You can compare each file you want to revert with the HEAD revision (or the index, or the previous version) and undo some or all changes done. Second, you can hard reset your project, causing any changes to be reverted.

Revert via Compare

Right click the file you want to revert and select Compare With -> HEAD Revision. This will open a comparison with the HEAD Revision, highlighting any changes done. You can revert several lines. select the line you want to revert and hit the Copy Current Change from Right to Left button (in the toolbar).

```
![Revert by Compare](resources/images/recipes/gitWithGama/
```



```
RevertFilByCompareWith.png)
```

Revert via Reset

To **reset all** changes made to your project, **right** click the project node and navigate to Team -> **Reset...** . Select the branch you want to **reset** to (if you haven't created any other branches, there will be just one). Click the **reset** button. **All** changes will be **reset** to this branch's last commit. Be careful **with** this option as **all** last changes in your Gama Project will be lost.

```
![Revert by Reset](resources/images/recipes/gitWithGama/ResetGit.png)
```

Clone Repositories

To checkout a remote project, you will have to clone its repository first. Open the GAMA **Import** wizard: **right** click the User models node -> **Import...** -> **Other...**

```
![Import git project](resources/images/recipes/gitWithGama/ImportFromGit.png)
```

Select Git -> Projects **from** Git and click Next.

```
![Import git project - Next](resources/images/recipes/gitWithGama/nextImportGitProject.png)
```

Select "**Clone URI**" and click Next.

```
![Repository URI](resources/images/recipes/gitWithGama/cloneURIGitProject.png)
```

Now you will have to **enter** the repository's **location**. Entering the URI will automatically **fill** some fields. Complete **any** other required fields and hit Next (e.g, Authentication fields). **If** you use GitHub, you can **copy** the URI **from** the web page.

```
![Repository location](resources/images/recipes/gitWithGama/
```

SourceGitRepositoryImport.png)

Select **all** branches you wish to clone and hit Next again.

![Branch Selection](resources/images/recipes/gitWithGama/ImportGitProjetBranchSelection.png)

Hit next, then choose a local storage **location** to **save** the repository **in**.

![Set local location](resources/images/recipes/gitWithGama/ImportProjectLocationNext.png)

To **import** the projects, **select** the cloned repository and hit Next.

Select **Import** Existing Projects and hit Next.

![Select a wizard to use](resources/images/recipes/gitWithGama/ImportProjectSelectWizardToUse.png)

In the following window, **select all** projects you want to **import** and click Finish.

![Select projects to import](resources/images/recipes/gitWithGama/ImportGitSelectProjects.png)

The projects should now appear **in** the Models Explorer. (Note the repository symbol **in** the icons indicating that the projects are already shared.)

![Imported projects](resources/images/recipes/gitWithGama/ImportedProjectsGit.png)

Create Branches

To **create** a new branch in your repository, **right** click your project and navigate to Team -> **Switch** to -> New Branch... **from** the context menu. **Select** the branch you want to **create** a new branch **from**, hit New branch and **enter** a **name** for the

new branch.

```
![Create new branch](resources/images/recipes/gitWithGama/
  CreateNewBranch.png)
```

The new branch (NewBranch) should appear in the branch selection window.

```
![Created new branch](resources/images/recipes/gitWithGama/
  GamaProjectNewBranch.png)
```

You can see all the branches in the Git Repositories view.

```
![New branches view](resources/images/recipes/gitWithGama/
  BranchesView.png)
```

If you would like to checkout the a branch, select it and click Checkout.

```
![Check out a branch](resources/images/recipes/gitWithGama/
  CheckOutBranch.png)
```

Merge

To merge one branch into another, right click the project node and navigate to Team -> Merge...

```
![Merge a branch](resources/images/recipes/gitWithGama/
  GitMerge.png)
```

The merge will execute and a window will pop-up with the results. The possible results are Already-up-to-date, Fast-forward, Merged, Conflicting, Failed.

```
![Merge a branch](resources/images/recipes/gitWithGama/
  MergePopUp.png)
```

Note that a conflicting result will leave the merge process incomplete. You will have to resolve the conflicts and try again. When there are conflicting changes in the working

project, the merge will fail.

Fetch and Pull

To **update** the remote branches **when** cloning remote repositories (Git creates copies of the branches as local branches and as remote branches) you will have to use Fetch. To perform a Fetch, **select** Team -> Fetch **From...** **from** the project's context menu.

To **update** your local branches, you will have to perform a Merge operation after fetching.

Pull

Pull combines Fetch and Merge. **Select** Team -> Pull.

Push

Local changes made to your local branches can be pushed to remote repositories causing a merge **from** your branches into the branches of the remote repository (X pulls **from** Y is the same as Y pushes to X). The Push wizard is pretty much the same as the Fetch wizard.

![Git Push](resources/images/recipes/gitWithGama/GitPush.png)

History View

To show the repository history, **right** click it and **select** Team -> Show **in** History. This will open the History View, giving an overview of the commits and allowing you to perform several actions (creating branches/tags, revert, **reset...**).

![Git Push](resources/images/recipes/gitWithGama/HisrtooryView.png)

Writing Unit Tests in GAML

[Unit testing](https://en.wikipedia.org/wiki/Unit_testing) is an essential instrument to ensure the quality of any software and it has been implemented in GAMA: this allows in particular that parts of the model are behaving as expected and that evolutions in the model do not introduce unexpected changes. To these purposes, the modeler can define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue.

Writing tests in GAML involves the use of 4 keywords:

```
* [`assert` statement](Statements#assert),
* [`test` statement](Statements#test),
* [`setup` statement](Statements#setup),
* [`type: test` facet of `experiment`](ModelOrganization#experiment-declarations).
```

In this unit testing tutorial, we intend to show how to write unit tests in GAML using the statement ``test``.

What is ``test`` in GAML?

In GAML, the statement ``test`` allows the modeler to write a part of code lines to verify if portions of our GAML model are doing exactly what they are expected to do: this is done through the use of several assertions (using ``assert`` statements). This is done independently from other parts of the model.

To write a typical GAML unit test, we can follow three steps:

1. Define a set of attributes to use within the test,
2. Write initialization instructions,
3. Write assertions.

The aim of using unit testing is to observe the resulting behavior of some parts of our model. If the observed behavior is consistent with the expectations, the unit test passes, otherwise, it fails, indicating that there is a problem concerning the tested part of the model.

```
## Introduction to assertions
```

The basis of Unit tests is to check that given pieces of codes provide expected results. To this purpose, the modeler can write some basic tests that should be true: s/he thus asserts that such expression can be evaluated to true using the ``assert`` statement. Here are some examples of ``assert`` uses:

```
assert 1 + 1 = 2; assert isGreater(5, 6) = false; assert rnd(1.0) <= 1.0;
```

With the above statements, the modeler states the ``1+1`` is equal to ``2``, ``isGreater(5,6)`` is false (given the fact that ``isGreater`` is an action defined in a species) and ``rnd(1.0)`` always returns a value below 1.0.

``assert`` can be used in any behavior statement (as an example in a ``reflex``, a ``state`` or in a ``test``. Note that, if they are written outside of a ``test`` and that the test is not fulfilled, then an exception is thrown during their execution.

As an example, the following model throws the exception: ``Assert failed 3>4`` (as obviously 3 is not greater than 4 and that the GAML ``>`` operator is properly implemented on this case).

```
model NewModel
global { init { assert 3 > 4; } }
experiment NewModel type: gui {}
```

To be able to have a dashboard of the state of your model w.r. t. the unit tests, they need to be written in a ``test`` and the model launched with an experiment of type ``test``.

```
## How to write a GAML `test`?
```

A ``test`` statement can be used in any species (regular species, global or experiment species) everywhere a ``reflex`` can be used. Its aim is to gather several asserts in one block. If the tests are executed with any kind of experiment but ``test``, they will be executed, but nothing is reported. With a ``test`` experiment, a kind of dashboard will be displayed.

So we will consider that we start by adding an ``experiment`` with ``type`` set to ``test``. The following code shows an example.

```
experiment MyTest type: test autorun: true { ... }
```

Let's consider the following GAML code:

```
model TestModel
```

```
global { init { create test_agent number: 1; } }
```

```
species test_agent { bool isGreater (int p1, int p2) { if (p1 >= p2) { return true; } else { return false; } } }
```

```
test testsOK {
  assert isGreater(5, 6) = false;
  assert isGreater(6, 5) = true;
}
```

```
test failingTests {
  assert ! isGreater(6, 6);
}
```

```
}
```

```
experiment MyTest type: test autorun: true { }
```

In this example, the defined action, ``isGreater``, returns ``true`` if a parameter ``p1`` is greater than a parameter ``p2`` and ``false`` if not. So to test it, we declare a unit test

using ``test`` and add inside several ``assert`` statements. For instance, ``assert isGreater(5, 6) = false;`` will return ``true`` if the result of ``isGreater(5, 6)`` is really false and ``false`` if not. So, if the action ``isGreater`` is well-defined, it should return ``false``. Considering that "`greater`" and "`greater and equal`" should be two different functions, we add a test to check that ``isGreater`` does not return true in case of equality of its 2 operands. In this case, as the action is not-well implemented the test fails.

The following picture illustrates the GUI dashboard for unit tests, showing for each test and even each assert whether it passes or fails. Clicking on the button will display in the GAML editor the code line.

```
![[Interface for unit tests execution.](resources/images/
  recipes/unit_tests_isgreater.png)
```

```
## Use of the `setup` statement
```

In a species where we want to execute several tests, it is common to want to have the same initial states, in order to prevent the previous tests to have modified the tested object and thus altering the unit test results. To this purpose, we can add the ``setup`` statement in the species and use it to set the expected initial state of the object to be tested. It will be called before every ``test``.

As an example, in the following model, we want to test the operator ``translated_by`` and ``translated_to`` on a point. As each of them will modify the point object to be tested, we add a ``setup`` to reinitialize it.

```
...
model TestModel

global {
  geometry loc <- {0,0};

  setup {
```



```

loc <- {0,0};
}

test translate_to {
loc <- loc translated_to {10,10};
loc <- loc translated_to {10,10};
assert loc.location = {10,10};
}

test translated_by {
loc <- loc translated_by {10,10};
loc <- loc translated_by {10,10};
assert loc.location = {20,20};
}
}

experiment MyTest type: test autorun: true { }
...

```

The test experiment

It is also possible to write tests in the `experiment``. The main idea is here to totally separate the model and its tests.

As an example let's consider the following GAML code, which aims to test several GAML operators, related to the graph datatype:

model TestGraphs

global { graph the_graph;

```

init {
int i <- 10;
create node_agent number: 7 {
location <- {i, i + ((i / 10) mod 2) * 10};
i <- i + 10;
}

the_graph <- as_distance_graph(node_agent, 30.0);
}

```

```
}  
species edge_agent { aspect default { draw shape color: #black; } }  
species node_agent { aspect default { draw circle(1) color: #red; loop neigh over:  
the_graph neighbors_of self { draw line([self.location, agent(neigh).location]) color:  
#black; } } }  
experiment loadgraph type: gui { output { display map type: opengl { species  
edge_agent; species node_agent; } } }  
experiment MyTest type: test autorun: true { test "MyFirstTest" { write the_graph;  
write (node_agent[2]); write ("Degrees"); write (the_graph in_degree_of (node_  
agent[2])); write (the_graph out_degree_of (node_agent[2])); write (the_graph  
degree_of (node_agent[2])); assert the_graph in_degree_of (node_agent[2]) = 4;  
write (the_graph out_degree_of (node_agent[2])); assert the_graph out_degree_of  
(node_agent[2]) = 4; assert the_graph degree_of (node_agent[2]) = 8; } } ""
```

Chapter 27

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 $2^n \times 2^n$ cells in the plan. Here are some solutions for this problem:

- Choose a grid with $2^n \times 2^n$ 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)

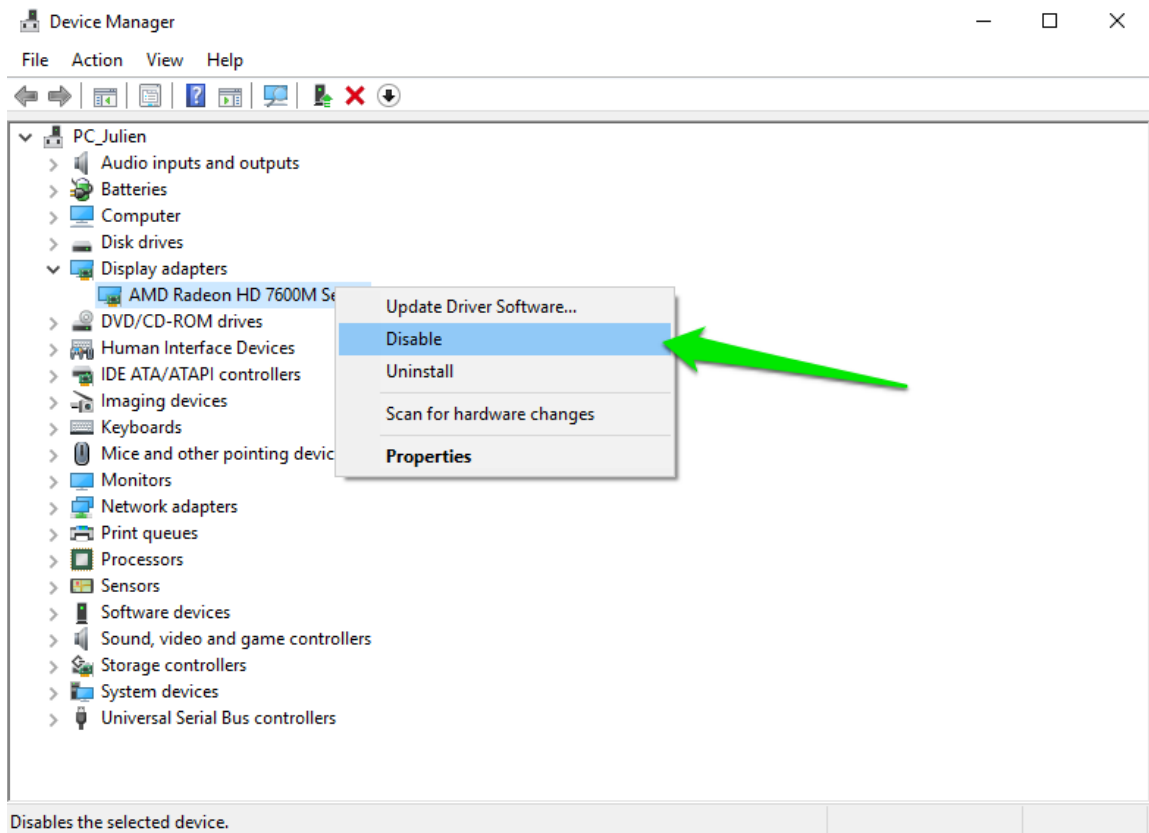


Figure 27.1: resources/images/recipes/disable_amd_radeon.png

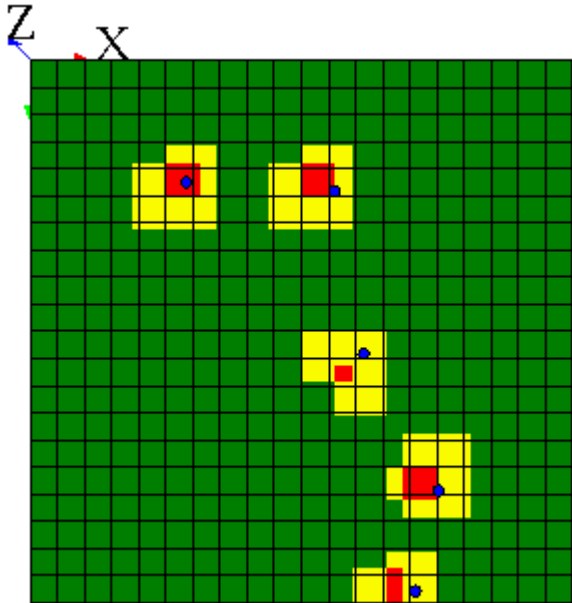


Figure 27.2: resources/images/recipes/grid_display_problem.png

Part V

GAML References

Chapter 28

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 29

Built-in Species

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

It is possible to use in the models a set of built-in agents. These agents allow to directly use some advance features like clustering, multi-criteria analysis, etc. The creation of these agents are similar as for other kinds of agents:

```
create species: my_built_in_agent returns: the_agent;
```

So, for instance, to be able to use clustering techniques in the model:

```
create cluster_builder returns: clusterer;
```

Table of Contents

[agent](#), [AgentDB](#), [base_edge](#), [experiment](#), [graph_edge](#), [graph_node](#), [physical_world](#),

agent

Variables

- **host** (-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_`

Returned type: [unknown](#)

`_step_`

Returned type: [unknown](#)

AgentDB

AgentDB is an abstract species that can be extended to provide agents with capabilities to access databases

Variables

- **agents** (`list`): Returns the list of agents for the population(s) of which the receiver agent is a direct or undirect host
- **members** (`container`): Returns the list of agents for the population(s) of which the receiver agent is a direct host

Actions

`close`

Close the established database connection.

Returned type: `unknown` : Returns null if the connection was successfully closed, otherwise, it returns an error.

`connect`

Establish a database connection.

Returned type: `unknown` : Returns null if connection to the server was successfully established, otherwise, it returns an error.

Additional facets:

- **params** (`map`): Connection parameters

`executeUpdate`

- Make a connection to DBMS - Executes the SQL statement in this PreparedStatement object, which must be an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that returns nothing, such as a DDL statement.

Returned type: `int` : Returns the number of updated rows.

Additional facets:

- **updateComm** (string): SQL commands such as Create, Update, Delete, Drop with question mark
- **values** (list): List of values that are used to replace question mark

getParameter

Returns the list used parameters to make a connection to DBMS (dbtype, url, port, database, user and passwd).

Returned type: `unknown` : Returns the list of used parameters to make a connection to DBMS.

insert

- Make a connection to DBMS - Executes the insert statement.

Returned type: `int` : Returns the number of updated rows.

Additional facets:

- **into** (string): Table name
- **columns** (list): List of column name of table
- **values** (list): List of values that are used to insert into table. Columns and values must have same size

isConnected

To check if connection to the server was successfully established or not.

Returned type: `bool` : Returns true if connection to the server was successfully established, otherwise, it returns false.

select

Make a connection to DBMS and execute the select statement.

Returned type: `list` : Returns the obtained result from executing the select statement.

Additional facets:

- `select` (string): select string
- `values` (list): List of values that are used to replace question marks

setParameter

Sets the parameters to use in order to make a connection to the DBMS (dbtype, url, port, database, user and passwd).

Returned type: `unknown` : null.

Additional facets:

- `params` (map): Connection parameters

testConnection

To test a database connection .

Returned type: `bool` : Returns true if connection to the server was successfully established, otherwise, it returns false.

Additional facets:

- `params` (map): Connection parameters

timeStamp

Get the current time of the system.

Returned type: `float` : Current time of the system in milliseconds

base_edge

A built-in species for agents representing the edges of a graph, from which one can inherit

Variables

- `source` (`agent`): The source agent of this edge
- `target` (`agent`): The target agent of this edge

Actions

experiment

An experiment is a declaration of the way to conduct simulations on a model. Any experiment attached to a model is a species (introduced by the keyword ‘experiment’ which directly or indirectly inherits from an abstract species called ‘experiment’ itself. This abstract species (sub-species of ‘agent’) defines several attributes and actions that can then be used in any experiment. ‘experiment’ defines several attributes, which, in addition to the attributes inherited from agent, form the minimal set of knowledge any experiment will have access to.

Variables

- **minimum_cycle_duration** (**float**): The minimum duration (in seconds) a simulation cycle should last. Default is 0. Units can be used to pass values smaller than a second (for instance '10 °msec')
- **model_path** (**string**): Contains the absolute path to the folder in which the current model is located
- **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. Each time it is set, the random number generator is reinitialized. WARNING: Setting it to zero actually means that you let GAMA choose a random seed
- **simulation** (-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

Returned type: `unknown`

`update_outputs`

Forces all outputs to refresh, optionally recomputing their values

Returned type: `unknown`

Additional facets:

- `recompute` (boolean): Whether or not to force the outputs to make a computation step
-

`graph_edge`

A species that represents an edge of a graph made of agents. The source and the target of the edge should be agents

Variables

- `source` (`agent`): The source agent of this edge
- `target` (`agent`): The target agent of this edge

Actions

graph_node

A base species to use as a parent for species representing agents that are nodes of a graph

Variables

- `my_graph` (`graph`): A reference to the graph containing the agent

Actions

`related_to`

This operator should never be called

Returned type: `bool`

Additional facets:

- `other` (agent): The other agent

physical_world

The base species for models that act as a 3D physical world. Can register and manage agents provided with either the ‘static_body’ or ‘dynamic_body’ skill. Inherits from ‘static_body’, so it can also act as a physical body itself (with a ‘mass’, ‘friction’, ‘gravity’), of course without motion – in this case, it needs to register itself as a physical agent using the ‘register’ action

Variables

- **accurate_collision_detection** (boolean): Enables or not a better (but slower) collision detection
- **automated_registration** (boolean): If set to true (the default), makes the world automatically register and unregister agents provided with either the 'static_body' or 'dynamic_body' skill. Otherwise, they must be registered using the 'register' action, which can be useful when only some agents need to be considered as 'physical agents'. Note that, in any case, the world needs to manually register itself if it is supposed to act as a physical body.
- **gravity** (point): Defines the value of gravity in this world. The default value is set to -9.80665 on the z-axis, that is 9.80665 m/s² towards the 'bottom' of the world. Can be set to any direction and intensity and applies to all the bodies present in the physical world
- **library** (string): This attribute allows to manually switch between two physics library, named 'bullet' and 'box2D'. The Bullet library, which comes in two flavors (see 'use_native') and the Box2D library in its Java version (<https://github.com/jbox2d/jbox2d>). Bullet is the default library but models in 2D should better use Box2D
- **max_substeps** (int): If equal to 0 (the default), makes the simulation engine be stepped alongside the simulation (no substeps allowed). Otherwise, sets the maximum number of physical simulation substeps that may occur within one GAMA simulation step
- **terrain** (31): This attribute is a matrix of float that can be used to represent a 3D terrain. The shape of the world, in that case, should be a box, where the dimension on the z-axis is used to scale the z-values of the DEM. The world needs to be register itself as a physical object
- **use_native** (boolean): This attribute allows to manually switch between the Java version of the Bullet library (JBullet, a modified version of <https://github.com/stephengold/jbullet>, which corresponds to version 2.72 of the original library) and the native Bullet library (Libbulletjme, <https://github.com/stephengold/Libbulletjme>, which is kept up-to-date with

the 3.x branch of the original library).The native version is the default one unless the libraries cannot be loaded, making JBullet the default

Actions

`register`

An action that allows to register agents in this physical world. Unregistered agents will not be governed by the physical laws of this world. If the world is to play a role in the physical world,then it needs to register itself (i.e. `do register([self]);`

Returned type: `unknown`

Additional facets:

- **bodies** (container): the list or container of agents to register in this physical world

Chapter 30

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: [skill11, skill12] {  
    ...  
}
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. An example of skill is the `moving` skill.

So, for instance, if a species is declared as:

```
species foo skills: [moving]{  
    ...  
}
```

Its agents will automatically be provided with the following variables : `speed`, `heading`, `destination` and the following actions: `move`, `goto`, `wander`, `follow` in addition to those built-in in species and declared by the modeller. Most of these variables, except the ones marked read-only, can be customized and modified like normal variables by the modeller. For instance, one could want to set a maximum for the speed; this would be done by redeclaring it like this:

```
float speed max:100 min:0;
```

Or, to obtain a speed increasing at each simulation step:

```
float speed max:100 min:0 <- 1 update: speed * 1.01;
```

Or, to change the speed in a behavior:

```
if speed = 5 {
  speed <- 10;
}
```

Table of Contents

[advanced_driving](#), [driving](#), [dynamic_body](#), [fipa](#), [MDXSKILL](#), [messaging](#), [moving](#), [moving3D](#), [network](#), [public_transport](#), [public_transport_scheduler](#), [skill_road](#), [skill_road_node](#), [SQLSKILL](#), [static_body](#),

advanced_driving

Variables

- `acc_bias` (`float`): the bias term used for asymmetric lane changing, parameter 'a_bias' in MOBIL

- **acc_gain_threshold** (**float**): the minimum acceleration gain for the vehicle to switch to another lane, introduced to prevent frantic lane changing. Known as the parameter 'a_th' in the MOBIL lane changing model
- **acceleration** (**float**): the current acceleration of the vehicle (in m/s^2)
- **allowed_lanes** (**list**): a list containing possible lane index values for the attribute `lowest_lane`
- **current_index** (**int**): the index of the current edge (road) in the path
- **current_lane** (**int**): the current lane on which the agent is
- **current_path** (**path**): the path which the agent is currently following
- **current_road** (**agent**): the road which the vehicle is currently on
- **current_target** (**agent**): the current target of the agent
- **delta_idm** (**float**): the exponent used in the computation of free-road acceleration in the Intelligent Driver Model
- **distance_to_current_target** (**float**): euclidean distance to the current target node
- **distance_to_goal** (**float**): euclidean distance to the endpoint of the current segment
- **final_target** (**agent**): the final target of the agent
- **follower** (**agent**): the vehicle following this vehicle
- **ignore_oneway** (**boolean**): if set to `true`, the vehicle will be able to violate one-way traffic rule
- **lane_change_cooldown** (**float**): the duration that a vehicle must wait before changing lanes again

- **lane_change_limit** (**int**): the maximum number of lanes that the vehicle can change during a simulation step
- **leading_distance** (**float**): the distance to the leading vehicle
- **leading_speed** (**float**): the speed of the leading vehicle
- **leading_vehicle** (**agent**): the vehicle which is right ahead of the current vehicle. If this is set to nil, the leading vehicle does not exist or might be very far away.
- **linked_lane_limit** (**int**): the maximum number of linked lanes that the vehicle can use; the default value is -1, i.e. the vehicle can use all available linked lanes
- **lowest_lane** (**int**): the lane with the smallest index that the vehicle is in
- **max_acceleration** (**float**): the maximum acceleration of the vehicle. Known as the parameter 'a' in the Intelligent Driver Model
- **max_deceleration** (**float**): the maximum deceleration of the vehicle. Known as the parameter 'b' in the Intelligent Driver Model
- **max_safe_deceleration** (**float**): the maximum deceleration that the vehicle is willing to induce on its back vehicle when changing lanes. Known as the parameter 'b_save' in the MOBIL lane changing model
- **max_speed** (**float**): the maximum speed that the vehicle can achieve. Known as the parameter 'v0' in the Intelligent Driver Model
- **min_safety_distance** (**float**): the minimum distance of the vehicle's front bumper to the leading vehicle's rear bumper, known as the parameter s0 in the Intelligent Driver Model
- **min_security_distance** (**float**): the minimal distance to another vehicle
- **next_road** (**agent**): the road which the vehicle will enter next
- **num_lanes_occupied** (**int**): the number of lanes that the vehicle occupies

- **on_linked_road** (boolean): is the agent on the linked road?
- **politeness_factor** (float): determines the politeness level of the vehicle when changing lanes. Known as the parameter 'p' in the MOBIL lane changing model
- **proba_block_node** (float): probability to block a node (do not let other vehicle cross the crossroad), within one second
- **proba_lane_change_down** (float): probability to change to a lower lane (right lane if right side driving) to gain acceleration, within one second
- **proba_lane_change_up** (float): probability to change to a upper lane (left lane if right side driving) to gain acceleration, within one second
- **proba_respect_priorities** (float): probability to respect priority (right or left) laws, within one second
- **proba_respect_stops** (list): probability to respect stop laws - one value for each type of stop, within one second
- **proba_use_linked_road** (float): probability to change to a linked lane to gain acceleration, within one second
- **real_speed** (float): the actual speed of the agent (in meter/second)
- **right_side_driving** (boolean): are vehicles driving on the right size of the road?
- **safety_distance_coeff** (float): the coefficient for the computation of the the min distance between two vehicles (according to the vehicle speed - $\text{security_distance} = \max(\text{min_security_distance}, \text{security_distance_coeff} * \text{min}(\text{self.real_speed}, \text{other.real_speed}))$)
- **security_distance_coeff** (float): the coefficient for the computation of the the min distance between two vehicles (according to the vehicle speed - $\text{safety_distance} = \max(\text{min_safety_distance}, \text{safety_distance_coeff} * \text{min}(\text{self.real_speed}, \text{other.real_speed}))$)

- **segment_index_on_road** (`int`): current segment index of the agent on the current road
- **speed** (`float`): the speed of the agent (in meter/second)
- **speed_coeff** (`float`): speed coefficient for the speed that the vehicle want to reach (according to the max speed of the road)
- **targets** (`list`): the current list of points that the agent has to reach (path)
- **time_headway** (`float`): the time gap that to the leading vehicle that the driver must maintain. Known as the parameter ‘T’ in the Intelligent Driver Model
- **time_since_lane_change** (`float`): the elapsed time since the last lane change
- **using_linked_road** (`boolean`): indicates if the vehicle is occupying at least one lane on the linked road
- **vehicle_length** (`float`): the length of the vehicle (in meters)
- **violating_oneway** (`boolean`): indicates if the vehicle is moving in the wrong direction on an one-way (unlinked) road

Actions

`advanced_follow_driving`

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

Returned type: `float` : the remaining time

Additional facets:

- **path** (`path`): a path to be followed.

- **target** (point): the target to reach
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **time** (float): time to travel

Examples:

```
do osm_follow path: the_path on: road_network;
```

choose_lane

Override this if you want to manually choose a lane when entering new road. By default, the vehicle tries to stay in the current lane. If the new road has fewer lanes than the current one and the current lane index is too big, it tries to enter the most uppermost lane.

Returned type: `int` : an integer representing the lane index

Additional facets:

- **new_road** (agent): the new road that's the vehicle is going to enter

compute_path

Action to compute the shortest path to the target node, or shortest path based on the provided list of nodes

Returned type: `path` : the computed path, or nil if no valid path is found

Additional facets:

- **graph** (graph): the graph representing the road network
- **target** (agent): the target node to reach

- **source** (agent): the source node (optional, if not defined, closest node to the agent location)
- **nodes** (list): the nodes forming the resulting path

Examples:

```
do compute_path graph: road_network target: target_node;  
do compute_path graph: road_network nodes: [node1, node5,  
node10];
```

drive

action to drive toward the target

Returned type: `bool`

Examples:

```
do drive;
```

drive_random

action to drive by chosen randomly the next road

Returned type: `bool`

Additional facets:

- **graph** (graph): a graph representing the road network
- **proba_roads** (map): a map containing for each road (key), the probability to be selected as next road (value)

Examples:

```
do drive_random init_node: some_node;
```

external_factor_impact

action that allows to define how the remaining time is impacted by external factor

Returned type: `float` : the remaining time

Additional facets:

- `new_road` (agent): the road on which to the vehicle wants to go
- `remaining_time` (float): the remaining time

Examples:

```
do external_factor_impact new_road: a_road remaining_time:
  0.5;
```

force_move

action to drive by chosen randomly the next road

Returned type: `float`

Additional facets:

- `lane` (int): the lane on which to make the agent move
- `acceleration` (float): acceleration of the vehicle
- `time` (float): time of move

Examples:

```
do drive_random init_node: some_node;
```

lane_choice

action to choose a lane

Returned type: `int` : the chosen lane, return -1 if no lane can be taken

Additional facets:

- `new_road` (agent): the road on which to choose the lane

Examples:

```
do lane_choice new_road: a_road;
```

on_entering_new_road

override this if you want to do something when the vehicle enters a new road (e.g. adjust parameters)

Returned type: `void`

path_from_nodes

action to compute a path from a list of nodes according to a given graph

Returned type: `path` : the computed path, return nil if no path can be taken

Additional facets:

- `graph` (graph): the graph representing the road network
- `nodes` (list): the list of nodes composing the path

Examples:

```
do compute_path_from_nodes graph: road_network nodes: [node1, node5, node10];
```


ready_to_cross

action to test if the vehicle cross a road node to move to a new road

Returned type: `bool` : true if the vehicle can cross the road node, false otherwise

Additional facets:

- `node` (agent): the road node to test
- `new_road` (agent): the road to test

Examples:

```
do is_ready_next_road new_road: a_road lane: 0;
```

speed_choice

action to choose a speed

Returned type: `float` : the chosen speed

Additional facets:

- `new_road` (agent): the road on which to choose the speed

Examples:

```
do speed_choice new_road: the_road;
```

test_next_road

action to test if the vehicle can take the given road

Returned type: `bool` : true (the vehicle can take the road) or false (the vehicle cannot take the road)

Additional facets:

- `new_road` (agent): the road to test

Examples:

```
do test_next_road new_road: a_road;
```

unregister

remove the vehicle from its current roads

Returned type: `bool`

Examples:

```
do unregister
```

driving

Variables

- `lanes_attribute` (`string`): the name of the attribute of the road agent that determine the number of road lanes
- `living_space` (`float`): the min distance between the agent and an obstacle (in meter)
- `obstacle_species` (`list`): the list of species that are considered as obstacles
- `speed` (`float`): the speed of the agent (in meter/second)
- `tolerance` (`float`): the tolerance distance used for the computation (in meter)

Actions

`follow_driving`

moves the agent along a given path passed in the arguments while considering the other agents in the network.

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **path** (path): a path to be followed.
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weights used for the moving.
- **living_space** (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- **tolerance** (float): tolerance distance used for the computation (replaces the current value of tolerance)
- **lanes_attribute** (string): the name of the attribut of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

Examples:

```
do follow speed: speed * 2 path: road_path;
```

`goto_driving`

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

Returned type: `path` : optional: the path followed by the agent.

Additional facets:

- **target** (geometry): the location or entity towards which to move.
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **on** (any type): list, agent, graph, geometry that restrains this move (the agent moves inside this geometry)
- **return_path** (boolean): if true, return the path followed (by default: false)
- **move_weights** (map): Weights used for the moving.
- **living_space** (float): min distance between the agent and an obstacle (replaces the current value of living_space)
- **tolerance** (float): tolerance distance used for the computation (replaces the current value of tolerance)
- **lanes_attribute** (string): the name of the attribute of the road agent that determine the number of road lanes (replaces the current value of lanes_attribute)

Examples:

```
do gotoTraffic target: one_of (list (species (self))) speed:
  speed * 2 on: road_network living_space: 2.0;
```

dynamic_body**Variables**

- **angular_damping** (float): Between 0 and 1. an angular deceleration coefficient that occurs even without contact

- **angular_velocity** ([point](#)): The angular velocity of the agent in the three directions, expressed as a point.
- **contact_damping** ([float](#)): Between 0 and 1. a deceleration coefficient that occurs in case of contact. Only available in the native Bullet library (no effect on the Java implementation)
- **damping** ([float](#)): Between 0 and 1. a linear deceleration coefficient that occurs even without contact
- **velocity** ([point](#)): The linear velocity of the agent in the three directions, expressed as a point.

Actions

apply

An action that allows to apply different effects to the object, like forces, impulses, etc.

Returned type: [unknown](#)

Additional facets:

- **clearance** (boolean): If true clears all forces applied to the agent and clears its velocity as well
- **impulse** ([point](#)): An idealised change of momentum. Adds to the velocity of the object. This is the kind of push that you would use on a pool billiard ball.
- **force** ([point](#)): Move (push) the object once with a certain moment, expressed as a point (vector). Adds to the existing forces.
- **torque** ([point](#)): Rotate (twist) the object once around its axes, expressed as a point (vector)

fipa

The fipa skill offers some primitives and built-in variables which enable agent to communicate with each other using the FIPA interaction protocol.

Variables

- **accept_proposals** (**list**): A list of ‘accept_proposal’ performative messages in the agent’s mailbox
- **agrees** (**list**): A list of ‘agree’ performative messages.
- **cancel**s (**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.

Returned type: `unknown`

Additional facets:

- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`agree`

Replies a message with an ‘`agree`’ performative message.

Returned type: `unknown`

Additional facets:

- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

`cancel`

Replies a message with a ‘`cancel`’ performative message.

Returned type: `unknown`

Additional facets:

- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

cfp

Replies a message with a ‘cfp’ performative message.

Returned type: [unknown](#)

Additional facets:

- [message](#) (24): The message to be replied
- [contents](#) (list): The content of the replying message

end_conversation

Reply a message with an ‘end_conversation’ performative message. This message marks the end of a conversation. In a ‘no-protocol’ conversation, it is the responsible of the modeler to explicitly send this message to mark the end of a conversation/interaction protocol. Please note that if the contents of the messages of the conversation are not read, then this command has no effect (i.e. it must be read by at least one of the agents in the conversation)

Returned type: [unknown](#)

Additional facets:

- [message](#) (24): The message to be replied
- [contents](#) (list): The content of the replying message

failure

Replies a message with a ‘failure’ performative message.

Returned type: [unknown](#)

Additional facets:

- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

inform

Replies a message with an ‘inform’ performative message.

Returned type: [unknown](#)

Additional facets:

- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

propose

Replies a message with a ‘propose’ performative message.

Returned type: [unknown](#)

Additional facets:

- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

query

Replies a message with a ‘query’ performative message.

Returned type: [unknown](#)

Additional facets:

- **message** (24): The message to be replied
- **contents** (list): The content of the replying message

refuse

Replies a message with a ‘refuse’ performative message.

Returned type: **unknown**

Additional facets:

- **message** (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.

Returned type: **unknown**

Additional facets:

- **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.

Returned type: `unknown`

Additional facets:

- `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.

Returned type: `unknown`

Additional facets:

- `message` (24): The message to be replied
- `contents` (list): The content of the replying message

send

Starts a conversation/interaction protocol.

Returned type: `msi.gaml.extensions.fipa.FIPAMessage`

Additional facets:

- `to` (list): A list of receiver agents
- `contents` (list): The content of the message. A list of any GAML type

- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

start_conversation

Starts a conversation/interaction protocol.

Returned type: `msi.gaml.extensions.fipa.FIPAMessage`

Additional facets:

- **to** (list): A list of receiver agents
- **contents** (list): The content of the message. A list of any GAML type
- **performative** (string): A string, representing the message performative
- **protocol** (string): A string representing the name of interaction protocol

subscribe

Replies a message with a ‘subscribe’ performative message.

Returned type: `unknown`

Additional facets:

- **message** (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`

Returned type: `list<unknown>`

Additional facets:

- `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`

Returned type: `bool`

Additional facets:

- **params** (map): Connection parameters

timeStamp

Returned type: **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

Returned type: **message**

Additional facets:

- **to** (any type): The agent, or server, to which this message will be sent to
 - **contents** (any type): The contents of the message, an arbitrary object
-

moving

The moving skill is intended to define the minimal set of behaviours required for agents that are able to move on different topologies

Variables

- **current_edge** ([geometry](#)): Represents the agent/geometry on which the agent is located (only used with a graph)
- **current_path** ([path](#)): Represents the path on which the agent is moving on (goto action on a graph)
- **destination** ([point](#)): Represents the next location of the agent if it keeps its current speed and heading (read-only). ** Only correct in continuous topologies and may return nil values if the destination is outside the environment **
- **heading** ([float](#)): Represents the absolute heading of the agent in degrees.
- **location** ([point](#)): Represents the current position of the agent
- **real_speed** ([float](#)): Represents the actual speed of the agent (in meter/second)
- **speed** ([float](#)): Represents the speed of the agent (in meter/second)

Actions

follow

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

Returned type: [path](#) : optional: the path followed by the agent.

Additional facets:

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

- **path** (path): a path to be followed.
- **move_weights** (map): Weights used for the moving.
- **return_path** (boolean): if true, return the path followed (by default: false)

Examples:

```
do follow speed: speed * 2 path: road_path;
```

goto

moves the agent towards the target passed in the arguments.

Returned type: **path** : optional: the path followed by the agent.

Additional facets:

- **target** (geometry): the location or entity towards which to move.
- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **on** (any type): graph, topology, list of geometries or map of geometries that restrain this move
- **recompute_path** (boolean): if false, the path is not 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.

Examples:

```
do goto target: (one_of road).location speed: speed * 2 on:  
road_network;
```


move

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

Returned type: `path`

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (float): the angle (in degree) of the target direction.
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

Examples:

```
do move speed: speed - 10 heading: heading + rnd (30) bounds: agentA;
```

wander

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

Returned type: `bool`

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **amplitude** (float): a restriction placed on the random heading choice. The new heading is chosen in the range (heading - amplitude/2, heading+amplitude/2)

- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)
- **on** (graph): the graph that restrains this move (the agent moves on the graph)
- **proba_edges** (map): When the agent moves on a graph, the probability to choose another edge. If not defined, each edge has the same probability to be chosen

Examples:

```
do wander speed: speed - 10 amplitude: 120 bounds: agentA;
```

moving3D

The moving skill 3D is intended to define the minimal set of behaviours required for agents that are able to move on different topologies

Variables

- **destination** (**point**): continuously updated destination of the agent with respect to its speed and heading (read-only)
- **heading** (**float**): the absolute heading of the agent in degrees (in the range 0-359)
- **pitch** (**float**): the absolute pitch of the agent in degrees (in the range 0-359)
- **roll** (**float**): the absolute roll of the agent in degrees (in the range 0-359)
- **speed** (**float**): the speed of the agent (in meter/second)

Actions

move

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

Returned type: `path`

Additional facets:

- **speed** (float): the speed to use for this move (replaces the current value of speed)
- **heading** (int): int, optional, the direction to take for this move (replaces the current value of heading)
- **pitch** (int): int, optional, the direction to take for this move (replaces the current value of pitch)
- **roll** (int): int, optional, the direction to take for this move (replaces the current value of roll)
- **bounds** (geometry): the geometry (the localized entity geometry) that restrains this move (the agent moves inside this geometry)

Examples:

```
do move speed: speed - 10 heading: heading + rnd (30) bounds: agentA;
```

network

The network skill provides new features to let agents exchange message through network.

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.

Returned type: `bool`

Additional facets:

- **protocol** (`string`): protocol type (MQTT (by default), TCP, UDP): the possible value are 'udp_server', 'udp_emitter', 'tcp_server', 'tcp_client', otherwise the MQTT protocol is used.
- **port** (`int`): Port number
- **raw** (`boolean`): message type raw or rich
- **with_name** (`string`): ID of the agent (its name) for the simulation
- **login** (`string`): login for the connection to the server
- **password** (`string`): password associated to the login
- **force_network_use** (`boolean`): force the use of the network even interaction between local agents
- **to** (`string`): server URL (localhost or a server URL)

- `size_packet` (int): For UDP connection, it sets the maximum size of received packets (default = 1024bits).

Examples:

```
do connect with_name:"any_name";
do connect to:"localhost" port:9876 with_name:"any_name";
do connect to:"localhost" protocol:"MQTT" port:9876
  with_name:"any_name";
do connect to:"localhost" protocol:"udp_server" port:9876
  with_name:"Server";
do connect to:"localhost" protocol:"udp_client" port:9876
  with_name:"Client";
  do connect to:"localhost" protocol:"udp_server" port:9877
  size_packet: 4096;
```

execute

Returned type: `string`

Additional facets:

- `command` (string): command to execute

Examples:

fetch_message

Returned type: `message`

has_more_message

Returned type: `bool`

join_group

allow an agent to join a group of agents in order to broadcast messages to other members or to receive messages sent by other members. Note that all members of the group called : "ALL".

Returned type: `bool`

Additional facets:

- `with_name` (string): name of the group

Examples:

```
do join_group with_name:"group name";  
do join_group with_name:"group name";do send to:"group name"  
  contents:"I am new in this group";
```

leave_group

leave a group of agents. The leaving agent will not receive any message from the group. Otherwise, it can send messages to the left group

Returned type: `bool`

Additional facets:

- `with_name` (string): name of the group the agent wants to leave

Examples:

```
do leave_group with_name:"my_group";
```

simulate_step

Simulate a step to test the skill. It must be used for Gama-platform test only

Returned type: `bool` : nothing

Examples:

```
do simulate_step;
```

public_transport**Variables**

- `is_stopped` (`boolean`): Is the transport waiting for passengers
- `next_stop` (`agent`): the next stop for the transport
- `stops` (`list`): The list of stops the bus have and will going through
- `transport_line` (`string`): The name of the bus line
- `transport_state` (`string`): ?

Actions

`define_next_target`

set up next target

Returned type: `bool`

Examples:

```
do define_next_target;
```

define_noria

action to define a bus noria

Returned type: `bool`

Additional facets:

- **pickup_point** (agent): The pickup point where passengers are taken
- **evacuation_point** (agent): The evacuation exit
- **return_point** (agent): The bus re-entry on the graph
- **waiting_time** (int): waiting time at pickup point in second (can be ignored if transport is full)
- **return_time** (int): time before the re-entry on the graph in second

Examples:

```
do define_noria pickup_point: bus_pickup evacuation_point:  
  exit_point return_point: exit_point waiting_time: 300  
  return_time: 600;
```

define_route

action to define the route of a bus

Returned type: `bool`

Additional facets:

- **stops** (list): The stops' list to go by
- **schedule** (list): The times' list for each stop

Examples:

```
do define_route stops: bus_stops schedule: bus_schedule;
```

init_departure

initialise the vehicle

Returned type: `bool`

Examples:

```
do init_departure;
```

is_time_to_go

test the departure time

Returned type: `bool` : returns true if it's time to go, false otherwise

Examples:

```
if(is_time_to_go())...
```

public_transport_scheduler

Variables

- `next_departure` (`int`): ?
- `next_departure_cycle` (`int`): ?
- `schedule` (`matrix`): ?
- `start_time_hour` (`int`): The name of the bus line
- `start_time_minute` (`int`): The name of the bus line
- `start_time_second` (`int`): The name of the bus line
- `stops` (`list`): ?
- `transport_line` (`string`): The name of the bus line

Actions

`check_departure`

action to check if a transport must depart

Returned type: `list<int>`

Examples:

```
do check_departure;
```

`check_next_departure`

action to check next departure time

Returned type: `bool`

Examples:

```
do check_next_departure;
```

define_schedule

action to define the schedule of a bus_line

Returned type: `bool`

Additional facets:

- `schedule` (`matrix`): The stop(x)/time(y) matrix[x,y]

Examples:

```
do define_schedule schedule: busline_schedule;
```

skill_road**Variables**

- `agents_on` (`list`): for each lane of the road, the list of agents for each segment
- `all_agents` (`list`): the list of agents on the road
- `linked_road` (`-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
- `num_lanes` (`int`): the number of lanes

- **num_segments** (`int`): the number of road segments
- **segment_lengths** (`list`): stores the length of each road segment. The index of each element corresponds to the segment index.
- **source_node** (`agent`): the source node of the road
- **target_node** (`agent`): the target node of the road
- **vehicle_ordering** (`list`): provides information about the ordering of vehicle on any given lane

Actions

`register`

register the agent on the road at the given lane

Returned type: `bool`

Additional facets:

- **agent** (`agent`): the agent to register on the road.
- **lane** (`int`): the lane index on which to register; if lane index \geq number of lanes, then register on the linked road

Examples:

```
do register agent: the_driver lane: 0
```

`unregister`

unregister the agent on the road

Returned type: `bool`

Additional facets:

- **agent** (agent): the agent to unregister on the road.

Examples:

```
do unregister agent: the_driver
```

skill_road_node**Variables**

- **block** (map): define the list of agents blocking the node, and for each agent, the list of concerned roads
- **priority_roads** (list): the list of priority roads
- **roads_in** (list): the list of input roads
- **roads_out** (list): the list of output roads
- **stop** (list): define for each type of stop, the list of concerned roads

Actions

SQLSKILL

This skill allows agents to be provided with actions and attributes in order to connect to SQL databases

Variables

Actions

`executeUpdate`

Returned type: `int`

Additional facets:

- `params` (map): Connection parameters
- `updateComm` (string): SQL commands such as Create, Update, Delete, Drop with question mark
- `values` (list): List of values that are used to replace question mark

`getCurrentDateTime`

Returned type: `string`

Additional facets:

- `dateFormat` (string): date format examples: 'yyyy-MM-dd' , 'yyyy-MM-dd HH:mm:ss'

`getDateOffset`

Returned type: `string`

Additional facets:

- **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

Returned type: `int`**Additional facets:**

- **params** (map): Connection parameters
- **into** (string): Table name
- **columns** (list): List of column name of table
- **values** (list): List of values that are used to insert into table. Columns and values must have same size

list2Matrix

Returned type: `matrix`

Additional facets:

- **param** (list): Param: a list of records and metadata
- **getName** (boolean): getType: a boolean value, optional parameter
- **getType** (boolean): getType: a boolean value, optional parameter

select

Returned type: `list`**Additional facets:**

- **params** (map): Connection parameters
- **select** (string): select string with question marks
- **values** (list): List of values that are used to replace question marks

testConnection

Returned type: `bool`**Additional facets:**

- **params** (map): Connection parameters

`timeStamp`

Returned type: `float`

`static_body`

Variables

- `aabb` (`geometry`): The axis-aligned bounding box. A box used to evaluate the probability of contacts between objects. Can be displayed as any other GAMA shapes/geometries in order to verify that the physical representation of the agent corresponds to its geometry in the model
- `friction` (`float`): Between 0 and 1. The coefficient of friction of the agent (how much it decelerates the agents in contact with him). Default is 0.5
- `mass` (`float`): The mass of the agent. Should be equal to 0.0 for static, motionless agents
- `restitution` (`float`): Between 0 and 1. The coefficient of restitution of the agent (defines the ‘bounciness’ of the agent). Default is 0
- `rotation` (`pair`): The rotation of the physical body, expressed as a pair which key is the angle in degrees and value the axis around which it is measured

Actions

`contact_added_with`

This action can be redefined in order for the agent to implement a specific behavior when it comes into contact (collision) with another agent. It is

automatically called by the physics simulation engine on both colliding agents. The default built-in behavior does nothing.

Returned type: [unknown](#)

Additional facets:

- **other** (agent): represents the other agent with which a collision has been detected

contact_removed_with

This action can be redefined in order for the agent to implement a specific behavior when a previous contact with another agent is removed. It is automatically called by the physics simulation engine on both colliding agents. The default built-in behavior does nothing.

Returned type: [unknown](#)

Additional facets:

- **other** (agent): represents the other agent with which a collision has been detected

update_body

This action must be called when the geometry of the agent changes in the simulation world and this change must be propagated to the physical world. The change of location (in either worlds) or the rotation due to physical forces do not count as changes, as they are already taken into account. However, a rotation in the simulation world need to be handled by calling this action. As it involves long operations (removing the agent from the physical world, then reinserting it with its new shape), this action should not be called too often.

Returned type: [unknown](#)

Chapter 31

Built-in Architectures

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

INTRODUCTION

Table of Contents

```
[fsm](#fsm), [parallel_bdi](#parallel_bdi), [
  probabilistic_tasks](#probabilistic_tasks), [reflex](#
  reflex), [rules](#rules), [simple_bdi](#simple_bdi), [
  sorted_tasks](#sorted_tasks), [user_first](#user_first), [
  user_last](#user_last), [user_only](#user_only), [
  weighted_tasks](#weighted_tasks),
```

fsm

Variables

- **state** (string): Returns the name of the current state of the agent
- **states** (list): Returns the list of all the states defined in the species

Actions

parallel_bdi

compute the bdi architecture in parallel

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): the current plan of the agent
- **desire_base** (list): the desire base of the agent
- **emotion_base** (list): the emotion base of the agent
- **extroversion** (float): an extroversion value for the personality
- **ideal_base** (list): the ideal base of the agent
- **intention_base** (list): the intention base of the agent
- **intention_persistence** (float): intention persistence
- **law_base** (list): the law base of the agent
- **neurotism** (float): a neurotism value for the personality
- **norm_base** (list): the norm base of the agent
- **obedience** (float): an obedience value. By default, it is computed with personality
- **obligation_base** (list): the obligation base of the agent
- **openness** (float): an openness value for the personality
- **plan_base** (list): the plan base of the agent
- **plan_persistence** (float): plan persistence
- **probabilistic_choice** (boolean): indicates if the choice is deterministic or probabilistic
- **receptivity** (float): a receptivity value. By default, it is computed with personality
- **sanction_base** (list): the sanction base of the agent

- **social_link_base** (list): the social link base of the agent
- **thinking** (list): the list of the last thoughts of the agent
- **uncertainty_base** (list): the uncertainty base of the agent
- **use_emotions_architecture** (boolean): indicates if emotions are automatically computed
- **use_norms** (boolean): indicates if the normative engine is used
- **use_persistence** (boolean): indicates if the persistence coefficient is computed with personality (false) or with the value given by the modeler
- **use_personality** (boolean): indicates if the personality is used
- **use_social_architecture** (boolean): indicates if social relations are automatically computed

Actions

add_belief

add the predicate in the belief base.

- returns: bool
- **predicate** (predicate): predicate to add as a belief
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_belief_emotion

add the belief about an emotion in the belief base.

- returns: bool
- **emotion** (emotion): emotion to add as a belief
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_belief_mental_state

add the predicate in the belief base.

- returns: bool
- **mental_state** (mental_state): predicate to add as a belief
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_desire

adds the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to add as a desire
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief
- **todo** (predicate): add the desire as a subintention of this parameter

add_desire_emotion

adds the emotion in the desire base.

- returns: bool
- **emotion** (emotion): emotion to add as a desire
- **strength** (float): the strength of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (predicate): add the desire as a subintention of this parameter

add_desire_mental_state

adds the mental state is in the desire base.

- returns: bool
- **mental_state** (mental_state): mental_state to add as a desire
- **strength** (float): the strength of the desire
- **lifetime** (int): the lifetime of the desire
- **todo** (predicate): add the desire as a subintention of this parameter

add_directly_belief

add the belief in the belief base.

- returns: bool
- **belief** (mental_state): belief to add in the belief base

add_directly_desire

add the desire in the desire base.

- returns: bool
- **desire** (mental_state): desire to add in th belief base

add_directly_ideal

add the ideal in the ideal base.

- returns: bool
- **ideal** (mental_state): ideal to add in the ideal base

add_directly_uncertainty

add the uncertainty in the uncertainty base.

- returns: bool
- **uncertainty** (mental_state): uncertainty to add in the uncertainty base

add_emotion

add the emotion to the emotion base.

- returns: bool
- **emotion** (emotion): emotion to add to the base

add_ideal

add a predicate in the ideal base.

- returns: bool
- **predicate** (predicate): predicate to add as an ideal

- **praiseworthiness** (float): the praiseworthiness value of the ideal
- **lifetime** (int): the lifetime of the ideal

add_ideal_emotion

add a predicate in the ideal base.

- returns: bool
- **emotion** (emotion): emotion to add as an ideal
- **praiseworthiness** (float): the praiseworthiness value of the ideal
- **lifetime** (int): the lifetime of the ideal

add_ideal_mental_state

add a predicate in the ideal base.

- returns: bool
- **mental_state** (mental_state): mental state to add as an ideal
- **praiseworthiness** (float): the praiseworthiness value of the ideal
- **lifetime** (int): the lifetime of the ideal

add_intention

check if the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to check

- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_intention_emotion

check if the predicates is in the desire base.

- returns: bool
- **emotion** (emotion): emotion to add as an intention
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_intention_mental_state

check if the predicates is in the desire base.

- returns: bool
- **mental_state** (mental_state): predicate to add as an intention
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_obligation

add a predicate in the ideal base.

- returns: bool
- **predicate** (predicate): predicate to add as an obligation

- **strength** (float): the strength value of the obligation
- **lifetime** (int): the lifetime of the obligation

add_social_link

add the social link to the social link base.

- returns: bool
- **social_link** (social_link): social link to add to the base

add_subintention

adds the predicates is in the desire base.

- returns: bool
- **predicate** (mental_state): the intention that receives the sub_intention
- **subintentions** (predicate): the predicate to add as a subintention to the intention
- **add_as_desire** (boolean): add the subintention as a desire as well (by default, false)

add_uncertainty

add a predicate in the uncertainty base.

- returns: bool
- **predicate** (predicate): predicate to add
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_uncertainty_emotion

add a predicate in the uncertainty base.

- returns: bool
- **emotion** (emotion): emotion to add as an uncertainty
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

add_uncertainty_mental_state

add a predicate in the uncertainty base.

- returns: bool
- **mental_state** (mental_state): mental state to add as an uncertainty
- **strength** (float): the strength of the belief
- **lifetime** (int): the lifetime of the belief

change_dominance

changes the dominance value of the social relation with the agent specified.

- returns: bool
- **agent** (agent): an agent with 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** (predicate): predicate to get

get_belief_emotion

return the belief about the emotion in the belief base (if several, returns the first one).

- returns: `mental_state`
- **emotion** (emotion): emotion about which the belief to get is

get_belief_mental_state

return the belief about the mental state in the belief base (if several, returns the first one).

- returns: `mental_state`
- **mental_state** (`mental_state`): mental state to get

get_belief_with_name

get the predicates is in the belief base (if several, returns the first one).

- returns: `mental_state`
- **name** (string): name of the predicate to check

get_beliefs

get the list of predicates in the belief base

- returns: list
- **predicate** (predicate): predicate to check

get_beliefs_metal_state

get the list of beliefs in the belief base containing the mental state

- returns: list
- **mental_state** (mental_state): mental state to check

get_beliefs_with_name

get the list of predicates is in the belief base with the given name.

- returns: list
- **name** (string): name of the predicates to check

get_current_intention

returns the current intention (last entry of intention base).

- returns: mental_state

get_current_plan

get the current plan.

- returns: BDIPlan

get_desire

get the predicates is in the desire base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to check

get_desire_mental_state

get the mental state is in the desire base (if several, returns the first one).

- returns: `mental_state`
- **mental_state** (`mental_state`): mental state to check

get_desire_with_name

get the predicates is in the belief base (if several, returns the first one).

- returns: `mental_state`
- **name** (string): name of the predicate to check

get_desires

get the list of predicates is in the desire base

- returns: list
- **predicate** (predicate): name of the predicates to check

get_desires_mental_state

get the list of mental states is in the desire base

- returns: list
- **mental_state** (`mental_state`): 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** (emotion): emotion to get

get_emotion_with_name

get the emotion is in the emotion base (if several, returns the first one).

- returns: emotion
- **name** (string): name of the emotion to check

get_ideal

get the ideal about the predicate in the ideal base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to check ad an ideal

get_ideal_mental_state

get the mental state in the ideal base (if several, returns the first one).

- returns: `mental_state`
- **mental_state** (`mental_state`): mental state to return

get_intention

get the predicates in the intention base (if several, returns the first one).

- returns: `mental_state`
- **predicate** (`predicate`): predicate to check

get_intention_mental_state

get the mental state is in the intention base (if several, returns the first one).

- returns: `mental_state`
- **mental_state** (`mental_state`): mental state to check

get_intention_with_name

get the predicates is in the belief base (if several, returns the first one).

- returns: `mental_state`
- **name** (`string`): name of the predicate to check

get_intentions

get the list of predicates is in the intention base

- returns: list
- **predicate** (predicate): name of the predicates to check

get_intentions_mental_state

get the list of mental state is in the intention base

- returns: list
- **mental_state** (mental_state): mental state 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** (predicate): predicate to return

get_plan

get the first plan with the given name

- returns: BDIPlan
- **name** (string): the name of the plan to 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** (social_link): social link to check

get_social_link_with_agent

get the social link with the agent concerned (if several, returns the first one).

- returns: social_link
- **agent** (agent): an agent with who I get a social link

get_uncertainty

get the predicates is in the uncertainty base (if several, returns the first one).

- returns: mental_state
- **predicate** (predicate): predicate to return

get_uncertainty_mental_state

get the mental state is in the uncertainty base (if several, returns the first one).

- returns: `mental_state`
- **mental_state** (`mental_state`): mental state to return

has_belief

check if the predicates is in the belief base.

- returns: `bool`
- **predicate** (`predicate`): predicate to check

has_belief_mental_state

check if the mental state is in the belief base.

- returns: `bool`
- **mental_state** (`mental_state`): mental state to check

has_belief_with_name

check if the predicate is in the belief base.

- returns: `bool`
- **name** (`string`): name of the predicate to check

has_desire

check if the predicates is in the desire base.

- returns: bool
- **predicate** (predicate): predicate to check

has_desire_mental_state

check if the mental state is in the desire base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_desire_with_name

check if the prediate is in the desire base.

- returns: bool
- **name** (string): name of the predicate to check

has_emotion

check if the emotion is in the belief base.

- returns: bool
- **emotion** (emotion): emotion to check

has_emotion_with_name

check if the emotion is in the emotion base.

- returns: bool
- **name** (string): name of the emotion to check

has_ideal

check if the predicates is in the ideal base.

- returns: bool
- **predicate** (predicate): predicate to check

has_ideal_mental_state

check if the mental state is in the ideal base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_ideal_with_name

check if the predicate is in the ideal base.

- returns: bool
- **name** (string): name of the predicate to check

has_obligation

check if the predicates is in the obligation base.

- returns: bool
- **predicate** (predicate): predicate to check

has_social_link

check if the social link base.

- returns: bool
- **social_link** (social_link): social link to check

has_social_link_with_agent

check if the social link base.

- returns: bool
- **agent** (agent): an agent with who I want to check if I have a social link

has_uncertainty

check if the predicates is in the uncertainty base.

- returns: bool
- **predicate** (predicate): predicate to check

has_uncertainty_mental_state

check if the mental state is in the uncertainty base.

- returns: bool
- **mental_state** (mental_state): mental state to check

has_uncertainty_with_name

check if the predicate is in the uncertainty base.

- returns: bool
- **name** (string): name of the uncertainty to check

is_current_intention

check if the predicates is the current intention (last entry of intention base).

- returns: bool
- **predicate** (predicate): predicate to check

is_current_intention_mental_state

check if the mental state is the current intention (last entry of intention base).

- returns: bool
- **mental_state** (mental_state): mental state to check

is_current_plan

tell if the current plan has the same name as tested

- returns: bool
- **name** (string): the name of the plan to test

remove_all_beliefs

removes the predicates from the belief base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_belief

removes the predicate from the belief base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_belief_mental_state

removes the mental state from the belief base.

- returns: bool
- **mental_state** (mental_state): mental state to remove

remove_desire

removes the predicates from the desire base.

- returns: bool
- **predicate** (predicate): predicate to remove from desire base

remove_desire_mental_state

removes the mental state from the desire base.

- returns: bool
- **mental_state** (mental_state): mental state to remove from desire base

remove_emotion

removes the emotion from the emotion base.

- returns: bool
- **emotion** (emotion): emotion to remove

remove_ideal

removes the predicates from the ideal base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_ideal_mental_state

removes the mental state from the ideal base.

- returns: bool
- **mental_state** (mental_state): metal state to remove

remove_intention

removes the predicates from the intention base.

- returns: bool
- **predicate** (predicate): intention's predicate to remove
- **desire_also** (boolean): removes also desire

remove_intention_mental_state

removes the mental state from the intention base.

- returns: bool
- **mental_state** (mental_state): intention's mental state to remove
- **desire_also** (boolean): removes also desire

remove_obligation

removes the predicates from the obligation base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_social_link

removes the social link from the social relation base.

- returns: bool
- **social_link** (social_link): social link to remove

remove_social_link_with_agent

removes the social link from the social relation base.

- returns: bool
- **agent** (agent): an agent with who I get the social link to remove

remove_uncertainty

removes the predicates from the uncertainty base.

- returns: bool
- **predicate** (predicate): predicate to remove

remove_uncertainty_mental_state

removes the mental state from the uncertainty base.

- returns: bool
- **mental_state** (mental_state): mental state to remove

replace_belief

replace the old predicate by the new one.

- returns: bool
 - **old_predicate** (predicate): predicate to remove
 - **predicate** (predicate): predicate to add
-

sorted_tasks**Variables****Actions**

user_first**Variables****Actions**

user_last

Variables

Actions

user_only

Variables

Actions

weighted_tasks

Variables

Actions

Chapter 32

Statements

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

Table of Contents

=, 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, explicit, focus, focus_on, generate, genetic, graphics, highlight, hill_climbing, if, image, inspect, law, layout, let, light, loop, match, mesh, migrate, monitor, norm, output, output_file, overlay, parameter, perceive, permanent, plan, pso, put, reactive_tabu, reflex, release, remove, return, rotation, rule, rule, run, sanction, save, set, setup, simulate, sobol, 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,

Statements by kinds

- **Batch method**
 - `annealing`, `exhaustive`, `explicit`, `genetic`, `hill_climbing`, `pso`, `reactive_tabu`, `sobol`, `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`, `mesh`, `overlay`, `rotation`,
- **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`, `generate`, `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`, `generate`, `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`, `generate`, `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`, `generate`, `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`, `generate`, `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, generate, 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, explicit, genetic, hill_climbing, output, parameter, permanent, pso, reactive_tabu, reflex, rule, setup, simulate, sobol, state, tabu, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,

- **Layer**

- add, ask, benchmark, draw, error, focus_on, highlight, if, let, loop, put, remove, set, status, switch, trace, try, using, warn, write,

- **Model**

- action, aspect, coping, equation, experiment, law, norm, output, perceive, plan, reflex, rule, rule, run, sanction, setup, species, start_simulation, state, task, test, user_command, user_init, user_panel, Variable_container, Variable_number, Variable_regular,

- **Output**

- ask, if,

- **Sequence of statements or action**

- add, ask, assert, assert, benchmark, break, capture, conscious_contagion, create, data, datalist, diffuse, do, draw, emotional_contagion, enforcement, error, focus, focus_on, generate, highlight, if, inspect, let, loop, migrate, put, release, remove, return, save, set, 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`, `mesh`, `overlay`, `rotation`,
- **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...;
}
```

=

Facets

- **right** (float), (omissible) : the right part of the equation (it is mandatory that it can be evaluated as a float)
- **left** (any type): the left part of the equation (it should be a variable or a call to the diff() or diff2() operators)

Definition

Allows to implement an equation in the form $\text{function}(n, t) = \text{expression}$. The left function is only here as a placeholder for enabling a simpler syntax and grabbing the variable as its left member.

Usages

- The syntax of the = statement is a bit different from the other statements. It has to be used as follows (in an equation):

```
float t;
float S;
float I;
equation SI {
    diff(S,t) = (- 0.3 * S * I / 100);
    diff(I,t) = (0.3 * S * I / 100);
}
```

- See also: [equation](#), [solve](#),

Embedments

- The = statement is of type: **Single statement**
- The = statement can be embedded into: [equation](#),
- The = statement embeds statements:

action

Facets

- **name** (an identifier), (omissible) : identifier of the action
- **index** (a datatype identifier): if the action returns a map, the type of its keys
- **of** (a datatype identifier): if the action returns a container, the type of its elements
- **type** (a datatype identifier): the action returned type
- **virtual** (boolean): whether the action is virtual (defined without a set of instructions) (false by default)

Definition

Allows to define in a species, model or experiment a new action that can be called elsewhere.

Usages

- The simplest syntax to define an action that does not take any parameter and does not return anything is:

```
action simple_action {  
    // [set of statements]  
}
```

- If the action needs some parameters, they can be specified between brackets after the identifier of the action:

```
action action_parameters(int i, string s){  
    // [set of statements using i and s]  
}
```

- If the action returns any value, the returned type should be used instead of the “action” keyword. A return statement inside the body of the action statement is mandatory.

```
int action_return_val(int i, string s){
  // [set of statements using i and s]
  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

Facets

- `to` (any type in [container, species, agent, geometry]): an expression that evaluates to a container

- `item` (any type), (omissible) : any expression to add in the container
- `all` (any type): Allows to either pass a container so as to add all its element, or 'true', if the item to add is already a container.
- `at` (any type): position in the container of added element

Definition

Allows to add, i.e. to insert, a new element in a container (a list, matrix, map, ...). Incorrect use: The addition of a new element at a position out of the bounds of the container will produce a warning and let the container unmodified. If `all`: is specified, it has no effect if its argument is not a container, or if its argument is 'true' and the item to add is not a container. In that latter case

Usages

- The new element can be added either at the end of the container or at a particular position.

```
add expr to: expr_container;    // Add at the end
add expr at: expr to: expr_container;  // Add at position
expr
```

- Case of a list, the expression in the facet `at`: should be an integer.

```
list<int> workingList <- [];add 0 at: 0 to: workingList ;//
workingList equals [0]add 10 at: 0 to: workingList ;//
workingList equals [10,0]add 20 at: 2 to: workingList ;//
workingList equals [10,0,20]add 50 to: workingList; //
workingList equals [10,0,20,50]add [60,70] all: true to:
workingList; //workingList equals [10,0,20,50,60,70]
```

- Case of a map: As a map is basically a list of pairs `key::value`, we can also use the `add` statement on it. It is important to note that the behavior of the statement is slightly different, in particular in the use of the `at` facet, which denotes the key of the pair.


```
map<string,string> workingMap <- [];add "val1" at: "x" to:
  workingMap;//workingMap equals ["x":"val1"]
```

- If the at facet is omitted, a pair `expr_item::expr_item` will be added to the map. An important exception is the case where the `expr_item` is a pair: in this case the pair is added.

```
add "val2" to: workingMap;//workingMap equals ["x":"val1", "val2":"val2"]
add "5":"val4" to: workingMap; //workingMap equals ["x":"val1", "val2":"val2", "5":"val4"]
```

- Notice that, as the key should be unique, the addition of an item at an existing position (i.e. existing key) will only modify the value associated with the given key.

```
add "val3" at: "x" to: workingMap;//workingMap equals ["x":"val3", "val2":"val2", "5":"val4"]
```

- On a map, the all facet will add all value of a container in the map (so as `pair_val_cont::val_cont`)

```
add [{"val4","val5"} all: true at: "x" to: workingMap;//workingMap equals ["x":"val3", "val2":"val2", "5":"val4","val4":"val4","val5":"val5"]
```

- In case of a graph, we can use the facets `node`, `edge` and `weight` to add a node, an edge or weights to the graph. However, these facets are now considered as deprecated, and it is advised to use the various `edge()`, `node()`, `edges()`, `nodes()` operators, which can build the correct objects to add to the graph

```
graph g <- as_edge_graph([{1,5}::{12,45}]);
add edge: {1,5}::{2,3} to: g;
list var <- g.vertices; // var equals [{1,5},{12,45},{2,3}]
list var <- g.edges; // var equals [polyline
  ({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]
add node: {5,5} to: g;
list var <- g.vertices; // var equals
  [{1.0,5.0},{12.0,45.0},{2.0,3.0},{5.0,5.0}]
list var <- g.edges; // var equals [polyline
  ({1.0,5.0}::{12.0,45.0}),polyline({1.0,5.0}::{2.0,3.0})]
```

- Case of a matrix: this statement can not be used on matrix. Please refer to the statement `put`.
- See also: `put`, `remove`,

Embedments

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

agents

Facets

- `value` (container): the set of agents to display
- `name` (a label), (omissible) : Human readable title of the layer
- `aspect` (an identifier): the name of the aspect that should be used to display the species
- `fading` (boolean): Used in conjunction with ‘trace:’, allows to apply a fading effect to the previous traces. Default is false
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point $\{0.5, 0.5, 0.5\}$, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- `refresh` (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, useful in case of agents that do not move)
- `rotate` (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- `selectable` (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true

- **size** (point): extent of the layer in the screen from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in ‘position’, an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **trace** (any type in $[\text{boolean}, \text{int}]$): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency level of the layer (between 0 – opaque – and 1 – fully transparent)
- **visible** (boolean): Defines whether this layer is visible or not

Definition

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

Usages

- The general syntax is:

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

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

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

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

Embedments

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

annealing

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `nb_iter_cst_temp` (int): number of iterations per level of temperature
- `temp_decrease` (float): temperature decrease coefficient
- `temp_end` (float): final temperature
- `temp_init` (float): initial temperature

Definition

This algorithm is an implementation of the Simulated Annealing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the annealing statement uses `method annealing` instead of the expected `annealing name: id :`

```
method annealing [facet: value];
```

- For example:

```
method annealing temp_init: 100 temp_end: 1 temp_decrease:
  0.5 nb_iter_cst_temp: 5 maximize: food_gathered;
```

Embedments

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

ask

Facets

- `target` (any type in [container, agent]), (omissible) : an expression that evaluates to an agent or a list of agents
- `as` (species): an expression that evaluates to a species
- `parallel` (any type in [boolean, int]): (experimental) setting this facet to ‘true’ will allow ‘ask’ to use concurrency when traversing the targets; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is false by default.

Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. If the value of the target facet is nil or empty, the statement is ignored.

Usages

- Ask a set of receiver agents, stored in a container, to perform a block of statements. The block is evaluated in the context of the agents' species

```
ask ${receiver_agents} {
    ${cursor}
}
```

- Ask one agent to perform a block of statements. The block is evaluated in the context of the agent's species

```
ask ${one_agent} {
    ${cursor}
}
```

- If the species of the receiver agent(s) cannot be determined, it is possible to force it using the `as` facet. An error is thrown if an agent is not a direct or undirect instance of this species

```
ask${receiver_agent(s)} as: ${a_species_expression} {
    ${cursor}
}
```

- To ask a set of agents to do something only if they belong to a given species, the `of_species` operator can be used. If none of the agents belong to the species, nothing happens

```
ask ${receiver_agents} of_species ${species_name} {
    ${cursor}
}
```

- Any statement can be declared in the block statements. All the statements will be evaluated in the context of the receiver agent(s), as if they were defined in their species, which means that an expression like `self` will represent the receiver agent and not the sender. If the sender needs to refer to itself, some of its own attributes (or temporary variables) within the block statements, it has to use the keyword `myself`.

```
species animal {
  float energy <- rnd (1000) min: 0.0;
  reflex when: energy > 500 { // executed when the energy is
    above the given threshold
      list<animal> others <- (animal at_distance 5); //
find all the neighboring animals in a radius of 5 meters
      float shared_energy <- (energy - 500) / length (
others); // compute the amount of energy to share with each
of them
      ask others { // no need to cast, since others has
already been filtered to only include animals
        if (energy < 500) { // refers to the energy of
each animal in others
          energy <- energy + myself.shared_energy; //
increases the energy of each animal
          myself.energy <- myself.energy - myself.
shared_energy; // decreases the energy of the sender
        }
      }
}
}
```

- If the species of the receiver agent cannot be determined, it is possible to force it by casting the agent. Nothing happens if the agent cannot be casted to this species

Embedments

- The `ask` statement is of type: **Sequence of statements or action**
- The `ask` statement can be embedded into: chart, Behavior, Sequence of statements or action, Layer, Output,

- The `ask` statement embeds statements:
-

aspect

Facets

- `name` (an identifier), (omissible) : identifier of the aspect (it can be used in a display to identify which aspect should be used for the given species). Two special names can also be used: ‘default’ will allow this aspect to be used as a replacement for the default aspect defined in preferences; ‘highlighted’ will allow the aspect to be used when the agent is highlighted as a replacement for the default (application of a color)

Definition

Aspect statement is used to define a way to draw the current agent. Several aspects can be defined in one species. It can use attributes to customize each agent’s aspect. The aspect is evaluate for each agent each time it has to be displayed.

Usages

- An example of use of the aspect statement:

```
species one_species {
  int a <- rnd(10);
  aspect aspect1 {
    if(a mod 2 = 0) { draw circle(a);}
    else {draw square(a);}
    draw text: "a= " + a color: #black size: 5;
  }
}
```


Embedments

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

assert

Facets

- `value` (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

Definition

Allows to check if the evaluation of a given expression returns true. If not, an error (or a warning) is raised. If the statement is used inside a test, the error is not propagated but invalidates the test (in case of a warning, it partially invalidates it). Otherwise, it is normally propagated

Usages

- Any boolean expression can be used

```
assert (2+2) = 4;
assert self != nil;
int t <- 0; assert is_error(3/t);
(1 / 2) is float
```

- if the ‘warn:’ facet is set to true, the statement emits a warning (instead of an error) in case the expression is false

```
assert 'abc' is string warning: true
```

- See also: `test`, `setup`, `is_error`, `is_warning`,

Embedments

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

benchmark

Facets

- `message` (any type), (omissible) : A message to display alongside the results. Should concisely describe the contents of the benchmark
- `repeat` (int): An int expression describing how many executions of the block must be handled. The output in this case will return the min, max and average durations

Definition

Displays in the console the duration in ms of the execution of the statements included in the block. It is possible to indicate, with the 'repeat' facet, how many times the sequence should be run

Usages

Embedments

- The `benchmark` statement is of type: **Sequence of statements or action**

- The **benchmark** statement can be embedded into: Behavior, Sequence of statements or action, Layer,
 - The **benchmark** statement embeds statements:
-

break

Facets

Definition

break allows to interrupt the current sequence of statements.

Usages

Embedments

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

camera

Facets

- **name** (string), (omissible) : The name of the camera. Will be used to populate a menu with the other camera presets. Can provide a value to the ‘camera:’ facet of the display, which specifies which camera to use. Using the special constant `#default` will make it the default of the surrounding display
- **distance** (float): If the ‘location:’ facet is not defined, defines the distance (in world units) that separates the camera from its target. If ‘location:’ is defined, especially if it is using a symbolic position, allows to specify the distance to keep from the target. If neither ‘location:’ or ‘distance:’ is defined, the default distance is the maximum between the width and the height of the world

- **dynamic** (boolean): If true, the location, distance and target are automatically recomputed every step. Default is false. When true, will also set ‘locked’ to true, to avoid interferences from users
- **lens** (any type in [float, int]): Allows to define the lens – field of view in degrees – of the camera. Between 0 and 360. Defaults to 45°
- **location** (any type in [point, string]): Allows to define the location of the camera in the world, i.e. from where it looks at its target. If ‘distance:’ is specified, the final location is translated on the target-camera axis to respect the distance. Can be a (possibly dynamically computed) point or a symbolic position (`#from_above`, `#from_left`, `#from_right`, `#from_up_right`, `#from_up_left`, `#from_front`, `#from_up_front`) that will be dynamically recomputed if the target moves. If ‘location:’ is not defined, it will be that of the default camera (`#from_top`, `#from_left...`) defined in the preferences.
- **locked** (boolean): If true, the user cannot modify the camera location and target by interacting with the display. It is automatically set when the camera is dynamic, so that the display can ‘follow’ the coordinates; but it can also be used with fixed coordinates to ‘focus’ the display on a specific scene
- **target** (any type in [point, agent, geometry]): Allows to define the target of the camera (what does it look at). It can be a point (in world coordinates), a geometry or an agent, in which case its (possibly dynamic) location is used as the target. This facet can be complemented by ‘distance:’ and/or ‘location:’ to specify from where the target is looked at. If ‘target:’ is not defined, the default target is the centroid of the world shape.

Definition

camera allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way. Several preset cameras are provided and accessible in the preferences (to choose the default) or in GAML using the keywords `#from_above`, `#from_left`, `#from_right`, `#from_up_right`, `#from_up_left`, `#from_front`, `#from_up_front`, `#isometric`. These cameras are unlocked (so that they can be manipulated by the user), look at the center of the world from a symbolic position, and the distance between this position and the target is equal to the maximum of the width and height of the world’s shape. These preset cameras can be reused when defining new cameras, since their names can become symbolic positions for them. For instance: camera ‘my_camera’ location: `#from_top` distance: 10; will lower (or extend) the distance between the camera and the center of the world to 10. camera ‘my_camera’ locked:

true location: #from_up_front target: people(0); will continuously follow the first agent of the people species from the up-front position.

Usages

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

Embedments

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

capture

Facets

- `target` (any type in [agent, container]), (omissible) : an expression that is evaluated as an agent or a list of the agent to be captured
- `as` (species): the species that the captured agent(s) will become, this is a micro-species of the calling agent's species
- `returns` (a new identifier): a list of the newly captured agent(s)

Definition

Allows an agent to capture other agent(s) as its micro-agent(s).

Usages

- The preliminary for an agent A to capture an agent B as its micro-agent is that the A's species must defined a micro-species which is a sub-species of B's species (cf. [`Species161#Nesting_species Nesting species`]).

```
species A {  
  ...  
}  
species B {  
  ...  
  species C parent: A {  
    ...  
  }  
  ...  
}
```

- To capture all “A” agents as “C” agents, we can ask an “B” agent to execute the following statement:

```
capture list(B) as: C;
```

- Deprecated writing:

```
capture target: list (B) as: C;
```

- See also: [release](#),

Embedments

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

catch

Facets

Definition

This statement cannot be used alone

Usages

- See also: [try](#),

Embedments

- The `catch` statement is of type: **Sequence of statements or action**
 - The `catch` statement can be embedded into: `try`,
 - The `catch` statement embeds statements:
-

chart

Facets

- `name` (string), (omissible) : the identifier of the chart layer
- `axes` (rgb): the axis color
- `background` (rgb): the background color
- `color` (rgb): Text color
- `gap` (float): minimum gap between bars (in proportion)
- `label_background_color` (rgb): Color of the label background (for Pie chart)
- `label_font` (any type in [string, font]): Label font face. Either the name of a font face or a font
- `label_text_color` (rgb): Color of the label text (for Pie chart)
- `legend_font` (any type in [string, font]): Legend font face. Either the name of a font face or a font

- **memorize** (boolean): Whether or not to keep the values in memory (in order to produce a csv file, for instance). The default value, true, can also be changed in the preferences
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-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 horizontal bar charts)
- **series_label_position** (an identifier), takes values in: {default, none, legend, onchart, yaxis, xaxis}: Position of the Series names: default (best guess), none, legend, onchart, xaxis (for category plots) or yaxis (uses the first serie name).
- **size** (point): the layer resize factor: $\{1,1\}$ refers to the original size whereas $\{0.5,0.5\}$ divides by 2 the height and the width of the layer. In case of a 3D layer, a 3D point can be used (note that $\{1,1\}$ is equivalent to $\{1,1,0\}$, so a resize of a layer containing 3D objects with a 2D points will remove the elevation)
- **style** (an identifier), takes values in: {line, whisker, area, bar, dot, step, spline, stack, 3d, ring, exploded, default}: The sub-style style, also default style for the series.
- **tick_font** (any type in [string, font]): Tick font face. Either the name of a font face or a font. When used for a series chart, it will set the font of values on the axes, but When used with a pie, it will modify the font of messages associated to each pie section.
- **tick_line_color** (rgb): the tick lines color
- **title_font** (any type in [string, font]): Title font face. Either the name of a font face or a font
- **title_visible** (boolean): chart title visible
- **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 horizontal lines and values on the left of the axis).
- `x_tick_values_visible` (boolean): X tick values visible
- `y_label` (string): the title for the Y axis
- `y_log_scale` (boolean): use Log Scale for Y axis
- `y_range` (any type in [float, int, point, list]): range of the y-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `y_serie_labels` (any type in [list, float, int, a label]): for heatmaps/3d charts, change the default y serie for an other value (string or numerical in a list or cumulative).
- `y_tick_line_visible` (boolean): Y tick line visible
- `y_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).
- `y_tick_values_visible` (boolean): Y tick values visible
- `y2_label` (string): the title for the second Y axis
- `y2_log_scale` (boolean): use Log Scale for second Y axis
- `y2_range` (any type in [float, int, point, list]): range of the second y-axis. Can be a number (which will set the axis total range) or a point (which will set the min and max of the axis).
- `y2_tick_unit` (float): the tick unit for the x-axis (distance between vertical lines and values bellow the axis).

Definition

`chart` allows modeler to display a chart: this enables to display specific values of the model at each iteration. GAMA can display various chart types: time series (series), pie charts (pie) and histograms (histogram).

Usages

- The general syntax is:

```
display chart_display {
  chart "chart name" type: series [additional options] {
    [Set of data, datalists statements]
  }
}
```

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

Embedments

- The `chart` statement is of type: **Layer**
- The `chart` statement can be embedded into: `display`,
- The `chart` statement embeds statements: `add`, `ask`, `data`, `datalist`, `do`, `put`, `remove`, `set`, `simulate`, `using`,

conscious__contagion

Facets

- `emotion_created` (emotion): the emotion that will be created with the contagion
- `emotion_detected` (emotion): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `intensity` (float): The intensity value of the emotion added to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly add an emotion of a perceived specie if the perceived agent ges a patricular emotion.

Usages

- Other examples of use:

```
conscious_contagion emotion_detected:fear emotion_created:
  fearConfirmed;
conscious_contagion emotion_detected:fear emotion_created:
  fearConfirmed charisma: 0.5 receptivity: 0.5;
```

Embedments

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

coping

Facets

- `name` (an identifier), (omissible) : The name of the rule
- `belief` (predicate): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `desire` (predicate): The mandatory desire
- `desires` (list): The mandatory desires
- `emotion` (emotion): The mandatory emotion
- `emotions` (list): The mandatory emotions
- `ideal` (predicate): The mandatory ideal

- `ideals` (list): The mandatory ideals
- `lifetime` (int): the lifetime value of the mental state created
- `new_belief` (predicate): The belief that will be added
- `new_beliefs` (list): The belief that will be added
- `new_desire` (predicate): The desire that will be added
- `new_desires` (list): The desire that will be added
- `new_emotion` (emotion): The emotion that will be added
- `new_emotions` (list): The emotion that will be added
- `new_ideal` (predicate): The ideal that will be added
- `new_ideals` (list): The ideals that will be added
- `new_uncertainties` (list): The uncertainty that will be added
- `new_uncertainty` (predicate): The uncertainty that will be added
- `obligation` (predicate): The mandatory obligation
- `obligations` (list): The mandatory obligations
- `parallel` (any type in [boolean, int]): setting this facet to ‘true’ will allow ‘perceive’ to use concurrency with a `parallel_bdi` architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `remove_belief` (predicate): The belief that will be removed
- `remove_beliefs` (list): The belief that will be removed
- `remove_desire` (predicate): The desire that will be removed
- `remove_desires` (list): The desire that will be removed
- `remove_emotion` (emotion): The emotion that will be removed
- `remove_emotions` (list): The emotion that will be removed
- `remove_ideal` (predicate): The ideal that will be removed
- `remove_ideals` (list): The ideals that will be removed
- `remove_intention` (predicate): The intention that will be removed
- `remove_obligation` (predicate): The obligation that will be removed
- `remove_obligations` (list): The obligation that will be removed
- `remove_uncertainties` (list): The uncertainty that will be removed
- `remove_uncertainty` (predicate): The uncertainty that will be removed
- `strength` (any type in [float, int]): The strength of the mental state created
- `threshold` (float): Threshold linked to the emotion.
- `uncertainties` (list): The mandatory uncertainties
- `uncertainty` (predicate): The mandatory uncertainty
- `when` (boolean):

Definition

enables to add or remove mental states depending on the emotions of the agent, after the emotional engine and before the cognitive or normative engine.

Usages

- Other examples of use:

```
coping emotion: new_emotion("fear") when: flip(0.5) new_desire
: new_predicate("test")
```

Embedments

- The `coping` statement is of type: **Behavior**
 - The `coping` statement can be embedded into: `simple_bdi`, `parallel_bdi`, `Species`, `Model`,
 - The `coping` statement embeds statements:
-

create

Facets

- **species** (any type in [species, agent]), (omissible) : an expression that evaluates to a species, the species of the agents to be created. In the case of simulations, the name 'simulation', which represents the current instance of simulation, can also be used as a proxy to their species
- **as** (species): optionally indicates a species into which to cast the created agents.
- **from** (any type): an expression that evaluates to a localized entity, a list of localized entities, a string (the path of a file), a file (shapefile, a .csv, a .asc or a OSM file) or a container returned by a request to a database
- **number** (int): an expression that evaluates to an int, the number of created agents

- **returns** (a new identifier): a new temporary variable name containing the list of created agents (a list, even if only one agent has been created)
- **with** (map): an expression that evaluates to a map, for each pair the key is a species attribute and the value the assigned value

Definition

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

Usages

- Its simple syntax to create **an_int** agents of species **a_species** is:

```
create a_species number: an_int;
create species_of(self) number: 5 returns: list5Agents;
5
```

- In GAML modelers can create agents of species **a_species** (**with** two **attributetypeandnaturewith** types corresponding to the types of the shapefile **attributes**) **from** a shapefile **the_shapefile** while reading attributes 'TYPE_OCC' and 'NATURE' of the shapefile. One agent will be created by object contained in the shapefile:

```
create a_species from: the_shapefile with: [type::read('
TYPE_OCC '), nature::read('NATURE')];
```

- In order to create agents from a .csv file, facet **header** can be used to specified whether we can use columns header:

```
create toto from: "toto.csv" header: true with:[att1::read("
NAME"), att2::read("TYPE")];
or
create toto from: "toto.csv" with:[att1::read(0), att2::read
(1)]; //with 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)
}
```

- Deprecated uses:

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

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

Embedments

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

data

Facets

- `legend` (string), (omissible) : The legend of the chart
- `value` (any type in [float, point, list]): The value to output on the chart
- `accumulate_values` (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- `color` (any type in [rgb, list]): color of the serie, for heatmap can be a list to specify [minColor,maxColor] or [minColor,medColor,maxColor]
- `fill` (boolean): Marker filled (true) or not (false)
- `line_visible` (boolean): Whether lines are visible or not
- `marker` (boolean): marker visible or not
- `marker_shape` (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker
- `marker_size` (float): Size in pixels of the marker

- **style** (an identifier), takes values in: {line, 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)

Definition

This statement allows to describe the values that will be displayed on the chart.

Usages

Embedments

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

datalist

Facets

- **value** (list): the values to display. Has to be a matrix, a list or a List of List. Each element can be a number (series/histogram) or a list with two values (XY chart)

- **legend** (list), (omissible) : the name of the series: a list of strings (can be a variable with dynamic names)
- **accumulate_values** (boolean): Force to replace values at each step (false) or accumulate with previous steps (true)
- **color** (list): list of colors, for heatmaps can be a list of [minColor,maxColor] or [minColor,medColor,maxColor]
- **fill** (boolean): Marker filled (true) or not (false), same for all series.
- **line_visible** (boolean): Line visible or not (same for all series)
- **marker** (boolean): marker visible or not
- **marker_shape** (an identifier), takes values in: {marker_empty, marker_square, marker_circle, marker_up_triangle, marker_diamond, marker_hor_rectangle, marker_down_triangle, marker_hor_ellipse, marker_right_triangle, marker_vert_rectangle, marker_left_triangle}: Shape of the marker. Same one for all series.
- **marker_size** (list): the marker sizes to display. Can be a list of numbers (same size for each marker of the series) or a list of list (different sizes by point)
- **style** (an identifier), takes values in: {line, 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)

Definition

add a list of series to a chart. The number of series can be dynamic (the size of the list changes each step). See Ant Foraging (Charts) model in ChartTest for examples.

Usages

Embedments

- The `datalist` statement is of type: **Single statement**
 - The `datalist` statement can be embedded into: chart, Sequence of statements or action,
 - The `datalist` statement embeds statements:
-

default

Facets

- `value` (any type), (omissible) : The value or values this statement tries to match

Definition

Used in a switch match structure, the block prefixed by default is executed only if no other block has matched (otherwise it is not).

Usages

- See also: `switch`, `match`,

Embedments

- The `default` statement is of type: **Sequence of statements or action**
 - The `default` statement can be embedded into: switch,
 - The `default` statement embeds statements:
-

diffuse

Facets

- **var** (an identifier), (omissible) : the variable to be diffused. If diffused over a field, then this name will serve to identify the diffusion
- **on** (any type in [species, 31, list]): the list of agents (in general cells of a grid), or a field on which the diffusion will occur
- **avoid_mask** (boolean): if true, the value will not be diffused in the masked cells, but will be restitute to the neighboring cells, multiplied by the proportion value (no signal lost). If false, the value will be diffused in the masked cells, but masked cells won't diffuse the value afterward (lost of signal). (default value : false)
- **cycle_length** (int): the number of diffusion operation applied in one simulation step
- **mask** (matrix): a matrix that masks the diffusion (created from an image for instance). The cells corresponding to the values smaller than “-1” in the mask matrix will not diffuse, and the other will diffuse.
- **matrix** (matrix): the diffusion matrix (“kernel” or “filter” in image processing). Can have any size, as long as dimensions are odd values.
- **method** (an identifier), takes values in: {convolution, dot_product}: the diffusion method. One of ‘convolution’ or ‘dot_product’
- **min** (float): if a value is smaller than this value, it will not be diffused. By default, this value is equal to 0.0. This value cannot be smaller than 0.
- **propagation** (a label), takes values in: {diffusion, gradient}: represents both the way the signal is propagated and the way to treat multiple propagation of the same signal occurring at once from different places. If propagation equals ‘diffusion’, the intensity of a signal is shared between its neighbors with respect to ‘proportion’, ‘variation’ and the number of neighbors of the environment places (4, 6 or 8). I.e., for a given signal S propagated from place P, the value transmitted to its N neighbors is : $S' = (S / N / \text{proportion}) - \text{variation}$. The intensity of S is then diminished by $S * \text{proportion}$ on P. In a diffusion, the different signals of the same name see their intensities added to each other on each place. If propagation equals ‘gradient’, the original intensity is not modified, and each neighbors receives the intensity : $S / \text{proportion} - \text{variation}$. If multiple propagation occur at once, only the maximum intensity is kept on each place. If ‘propagation’ is not defined, it is assumed that it is equal to

‘diffusion’.

- **proportion** (float): a diffusion rate
- **radius** (int): a diffusion radius (in number of cells from the center)
- **variation** (float): an absolute value to decrease at each neighbors

Definition

This statements allows a value to diffuse among a species on agents (generally on a grid) depending on a given diffusion matrix.

Usages

- A basic example of diffusion of the variable phero defined in the species cells, given a diffusion matrix `math_diff` is:

```
matrix<float> math_diff <- matrix
  ([[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]);
diffuse var: phero on: cells matrix: math_diff;
```

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

```
diffuse var: phero on: cells matrix: math_diff mask: mymask;
```

- A convenient way to have an uniform diffusion in a given radius is (which is equivalent to the above diffusion):

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

Embedments

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

display

Facets

- **name** (a label), (omissible) : the identifier of the display
- **antialias** (boolean): Indicates whether to use advanced antialiasing for the display or not. The default value is the one indicated in the preferences of GAMA ('false' is its factory default). Antialiasing produces smoother outputs, but comes with a cost in terms of speed and memory used.
- **autosave** (any type in [boolean, point, string]): Allows to save this display on disk. This facet accepts bool, point or string values. If it is false or nil, nothing happens. 'true' will save it at a resolution of 500x500 with a standard name (containing the name of the model, display, resolution, cycle and time). A non-nil point will change that resolution. A non-nil string will keep 500x500 and change the filename (if it is not dynamically built, the previous file will be erased). Note that setting autosave to true in a display will synchronize all the displays defined in the experiment
- **axes** (boolean): Allows to enable/disable the drawing of the world shape and the ordinate axes. Default can be configured in Preferences
- **background** (rgb): Allows to fill the background of the display with a specific color
- **camera** (string): Allows to define the name of the camera to use. Default value is 'default'. Accepted values are (1) the name of one of the cameras defined using the 'camera' statement or (2) one of the preset cameras, accessible using constants: #from_above, #from_left, #from_right, #from_up_left, #from_up_right, #from_front, #from_up_front, #isometric
- **fullscreen** (any type in [boolean, int]): Indicates, when using a boolean value, whether or not the display should cover the whole screen (default is false). If an integer is passed, specifies also the screen to use: 0 for the primary monitor, 1 for the secondary one, and so on and so forth. If the monitor is not available, the first one is used
- **keystone** (container): Set the position of the 4 corners of your screen ([topLeft,topRight,botLeft,botRight]), in (x,y) coordinate (the (0,0) position is the top left corner, while the (1,1) position is the bottom right corner). The default value is : [{0,0},{1,0},{0,1},{1,1}]
- **light** (boolean): Allows to enable/disable the light at once. Default is true
- **orthographic_projection** (boolean): Allows to enable/disable the orthographic projection. Default can be configured in Preferences

- **parent** (an identifier): Declares that this display inherits its layers and attributes from the parent display named as the argument. Expects the identifier of the parent display or a string if the name of the parent contains spaces
- **refresh** (boolean): Indicates the condition under which this output should be refreshed (default is true)
- **show_fps** (boolean): Allows to enable/disable the drawing of the number of frames per second
- **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.

Definition

A display refers to an independent and mobile part of the interface that can display species, images, texts or charts.

Usages

- The general syntax is:

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

- Each display can include different layers (like in a GIS).

```

display gridWithElevationTriangulated type: opengl
  ambient_light: 100 {
    grid cell elevation: true triangulation: true;
    species people aspect: base;
  }

```

Embedments

- The **display** statement is of type: **Output**
- The **display** statement can be embedded into: output, permanent,
- The **display** statement embeds statements: **agents**, **camera**, **chart**, **display_grid**, **display_population**, **event**, **graphics**, **image**, **light**, **mesh**, **overlay**, **rotation**,

display_grid

Facets

- **species** (species), (omissible) : the species of the agents in the grid
- **border** (rgb): the color to draw lines (borders of cells)
- **elevation** (any type in [matrix, float, int, boolean]): Allows to specify the elevation of each cell, if any. Can be a matrix of float (provided it has the same size than the grid), an int or float variable of the grid species, or simply true (in which case, the variable called 'grid_value' is used to compute the elevation of each cell)
- **grayscale** (boolean): if true, give a grey value to each polygon depending on its elevation (false by default)
- **hexagonal** (boolean):
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point $\{0.5, 0.5, 0.5\}$, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.

- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **rotate** (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- **selectable** (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- **size** (point): extent of the layer in the screen from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in ‘position’, an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **smooth** (boolean): Applies a simple convolution (box filter) to smooth out the terrain produced by this field. Does not change the values of course.
- **text** (boolean): specify whether the attribute used to compute the elevation is displayed on each cells (false by default)
- **texture** (file): Either file containing the texture image to be applied on the grid or, if not specified, the use of the image composed by the colors of the cells
- **transparency** (float): the transparency level of the layer (between 0 – opaque – and 1 – fully transparent)
- **triangulation** (boolean): specifies 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)
- **visible** (boolean): Defines whether this layer is visible or not
- **wireframe** (boolean): if true displays the grid in wireframe using the lines color

Definition

display_grid is used using the **grid** keyword. It allows the modeler to display in an optimized way all cell agents of a grid (i.e. all agents of a species having a grid topology).

Usages

- The general syntax is:

```
display my_display {
```

```

grid ant_grid lines: #black position: { 0.5, 0 } size:
{0.5,0.5};
}

```

- To display a grid as a DEM:

```

display my_display {
  grid cell texture: texture_file text: false triangulation:
true elevation: true;
}

```

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

Embedments

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

display_population

Facets

- **species** (species), (omissible) : the species to be displayed
- **aspect** (an identifier): the name of the aspect that should be used to display the species
- **fading** (boolean): Used in conjunction with ‘trace:’, allows to apply a fading effect to the previous traces. Default is false
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. `{0.5,0.5}` refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined

between 0 and 1. The position can only be a 3D point {0.5, 0.5, 0.5}, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.

- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **rotate** (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- **selectable** (boolean): Indicates whether the agents present on this layer are selectable by the user. Default is true
- **size** (point): extent of the layer in the screen from its position. Coordinates in [0,1[are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in 'position', an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **trace** (any type in [boolean, int]): Allows to aggregate the visualization of agents at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency level of the layer (between 0 – opaque – and 1 – fully transparent)
- **visible** (boolean): Defines whether this layer is visible or not

Definition

The `display_population` statement is used using the `species` keyword. It allows modeler to display all the agent of a given species in the current display. In particular, modeler can choose the aspect used to display them.

Usages

- The general syntax is:

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

- Species can be superposed on the same plan (be careful with the order, the last one will be above all the others):

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

- Each species layer can be placed at a different z value using the opengl display. position:{0,0,0} means the layer will be placed on the ground and position:{0,0,1} means it will be placed at an height equal to the maximum size of the environment.

```
display my_display type: opengl{
  species agent1 aspect: base ;
  species agent2 aspect: base position:{0,0,0.5};
  species agent3 aspect: base position:{0,0,1};
}
```

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

Embedments

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

do

Facets

- `action` (an identifier), (omissible) : the name of an action or a primitive
- `internal_function` (any type):
- `with` (map): a map expression containing the parameters of the action

Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this [BuiltIn161 page]; for the list of primitives defined by the different skills, see this [Skills161 page]. Finally, see this [Species161 page] to know how to declare custom actions.

Usages

- The simple syntax (when the action does not expect any argument and the result is not to be kept) is:

```
do name_of_action_or_primitive;
```

- In case the action expects one or more arguments to be passed, they are defined by using facets (enclosed tags or a map are now deprecated):

```
do name_of_action_or_primitive arg1: expression1 arg2:  
  expression2;
```

- In case the result of the action needs to be made available to the agent, the action can be called with the agent calling the action (`self` when the agent itself calls the action) instead of `do`; the result should be assigned to a temporary variable:

```
type_returned_by_action result <- self  
  name_of_action_or_primitive [];
```

- In case of an action expecting arguments and returning a value, the following syntax is used:

```
type_returned_by_action result <- self  
  name_of_action_or_primitive [arg1::expression1, arg2::  
  expression2];
```

- Deprecated uses: following uses of the `do` statement (still accepted) are now deprecated:

```
// Simple syntax:
do action: name_of_action_or_primitive;

// In case the result of the action needs to be made available
// to the agent, the `returns` keyword can be defined; the
// result will then be referred to by the temporary variable
// declared in this attribute:
do name_of_action_or_primitive returns: result;
do name_of_action_or_primitive arg1: expression1 arg2:
  expression2 returns: result;
type_returned_by_action result <- name_of_action_or_primitive(
  self, [arg1::expression1, arg2::expression2]);

// In case the result of the action needs to be made available
// to the agent
let result <- name_of_action_or_primitive(self, []);

// In case the action expects one or more arguments to be
// passed, they can also be defined by using enclosed `arg`
// statements, or the `with` facet with a map of parameters:
do name_of_action_or_primitive with: [arg1::expression1, arg2
  ::expression2];

or

do name_of_action_or_primitive {
  arg arg1 value: expression1;
  arg arg2 value: expression2;
  ...
}
```

Embedments

- The `do` statement is of type: **Single statement**
- The `do` statement can be embedded into: chart, Behavior, Sequence of statements or action,

- The `do` statement embeds statements:
-

draw

Facets

- `geometry` (any type), (omissible) : any type of data (it can be geometry, image, text)
- `anchor` (point): Only used when perspective: true in OpenGL. The anchor point of the location with respect to the envelope of the text to draw, can take one of the following values: `#center`, `#top_left`, `#left_center`, `#bottom_left`, `#bottom_center`, `#bottom_right`, `#right_center`, `#top_right`, `#top_center`; or any point between `{0,0}` (`#bottom_left`) and `{1,1}` (`#top_right`)
- `at` (point): location where the shape/text/icon is drawn
- `begin_arrow` (any type in [int, float]): the size of the arrow, located at the beginning of the drawn geometry
- `border` (any type in [rgb, boolean]): if used with a color, represents the color of the geometry border. If set to false, expresses that no border should be drawn. If not set, the borders will be drawn using the color of the geometry.
- `color` (any type in [rgb, container]): the color to use to display the object. In case of images, will try to colorize it. You can also pass a list of colors : in that case, each color will be matched to its corresponding vertex.
- `depth` (float): (only if the display type is opengl) Add an artificial depth to the geometry previously defined (a line becomes a plan, a circle becomes a cylinder, a square becomes a cube, a polygon becomes a polyhedron with height equal to the depth value). Note: This only works if the geometry is not a point
- `end_arrow` (any type in [int, float]): the size of the arrow, located at the end of the drawn geometry
- `font` (any type in [font, string]): the font used to draw the text, if any. Applying this facet to geometries or images has no effect. You can construct here your font with the operator “font”. ex : `font:font(“Helvetica”, 20 , #plain)`
- `lighted` (boolean): Whether the object should be lighted or not (only applicable in the context of opengl displays)
- `material` (material): 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.
- **precision** (float): (only if the display type is opengl and only for text drawing) controls the accuracy with which curves are rendered in glyphs. Between 0 and 1, the default is 0.1. Smaller values will output much more faithful curves but can be considerably slower, so it is better if they concern text that does not change and can be drawn inside layers marked as ‘refresh: false’
- **rotate** (any type in [float, int, pair]): orientation of the shape/text/icon; can be either an int/float (angle) or a pair float::point (angle::rotation axis). The rotation axis, when expressed as an angle, is by default {0,0,1}
- **size** (any type in [float, point]): Size of the shape/icon/image to draw, expressed as a bounding box (width, height, depth; if expressed as a float, represents the box as a cube). Does not apply to texts: use a font with the required size instead
- **texture** (any type 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
- **wireframe** (boolean): a condition specifying whether to draw the geometry in wireframe or not

Definition

draw is used in an aspect block to express how agents of the species will be drawn. It is evaluated each time the agent has to be drawn. It can also be used in the graphics block.

Usages

- Any kind of geometry as any location can be drawn when displaying an agent (independently of his shape)

```
aspect geometryAspect {
  draw circle(1.0) empty: !hasFood color: #orange ;
}
```

- Image or text can also be drawn


```

aspect arrowAspect {
  draw "Current state= "+state at: location + {-3,1.5} color
  : #white font: font('Default', 12, #bold) ;
  draw file(ant_shape_full) rotate: heading at: location
  size: 5
}

```

- Arrows can be drawn with any kind of geometry, using `begin_arrow` and `end_arrow` facets, combined with the empty: facet to specify whether it is plain or empty

```

aspect arrowAspect {
  draw line([20, 20], [40, 40]) color: #black begin_arrow
  :5;
  draw line([10, 10],[20, 50], [40, 70]) color: #green
  end_arrow: 2 begin_arrow: 2 empty: true;
  draw square(10) at: {80,20} color: #purple begin_arrow: 2
  empty: true;
}

```

Embedments

- The `draw` statement is of type: **Single statement**
- The `draw` statement can be embedded into: aspect, Sequence of statements or action, Layer,
- The `draw` statement embeds statements:

else

Facets

Definition

This statement cannot be used alone

Usages

- See also: `if`,

Embedments

- The `else` statement is of type: **Sequence of statements or action**
 - The `else` statement can be embedded into: `if`,
 - The `else` statement embeds statements:
-

`emotional_contagion`

Facets

- `emotion_detected` (emotion): the emotion that will start the contagion
- `name` (an identifier), (omissible) : the identifier of the emotional contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `emotion_created` (emotion): the emotion that will be created with the contagion
- `intensity` (float): The intensity value of the emotion created to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to make conscious or unconscious emotional contagion

Usages

- Other examples of use:

```
emotional_contagion emotion_detected:fearConfirmed;  
emotional_contagion emotion_detected:fear emotion_created:  
    fearConfirmed;  
emotional_contagion emotion_detected:fear emotion_created:  
    fearConfirmed charisma: 0.5 receptivity: 0.5;
```

Embedments

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

enforcement

Facets

- `name` (an identifier), (omissible) : the identifier of the enforcement
- `law` (string): The law to enforce
- `norm` (string): The norm to enforce
- `obligation` (predicate): The obligation to enforce
- `reward` (string): The positive sanction to apply if the norm has been followed
- `sanction` (string): The sanction to apply if the norm is violated
- `when` (boolean): A boolean value to enforce only with a certain condition

Definition

apply a sanction if the norm specified is violated, or a reward if the norm is applied by the perceived agent

Usages

- Other examples of use:

```
focus var:speed /*where speed is a variable from a species
  that is being perceived*/
```

Embedments

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

enter

Facets

Definition

In an FSM architecture, `enter` introduces a sequence of statements to execute upon entering a state.

Usages

- In the following example, at the step it enters into the state `s_init`, the message 'Enter in `s_init`' is displayed followed by the display of the state name:

```
state s_init {
  enter { write "Enter in" + state; }
        write "Enter in" + state;
}
write state;
}
```

- See also: [state](#), [exit](#), [transition](#),

Embedments

- The `enter` statement is of type: **Sequence of statements or action**
 - The `enter` statement can be embedded into: state,
 - The `enter` statement embeds statements:
-

equation

Facets

- `name` (an identifier), (omissible) : the equation identifier
- `params` (list): the list of parameters used in predefined equation systems
- `simultaneously` (list): a list of species containing a system of equations (all systems will be solved simultaneously)
- `type` (an identifier), takes values in: {SI, SIS, SIR, SIRS, SEIR, LV}: the choice of one among classical models (SI, SIS, SIR, SIRS, SEIR, LV)
- `vars` (list): the list of variables used in predefined equation systems

Definition

The equation statement is used to create an equation system from several single equations.

Usages

- The basic syntax to define an equation system is:

```
float t;  
float S;  
float I;  
equation SI {  
    diff(S,t) = (- 0.3 * S * I / 100);  
}
```

```

diff(I,t) = (0.3 * S * I / 100);
}

```

- If the `type: facet` is used, a predefined equation system is defined using variables `vars:` and parameters `params:` in the right order. All possible predefined equation systems are the following ones (see [EquationPresentation161 EquationPresentation161] for precise definition of each classical equation system):

```

equation eqSI type: SI vars: [S,I,t] params: [N,beta];
equation eqSIS type: SIS vars: [S,I,t] params: [N,beta,gamma];
equation eqSIR type: SIR vars: [S,I,R,t] params: [N,beta,gamma];
equation eqSIRS type: SIRS vars: [S,I,R,t] params: [N,beta,
  gamma,omega,mu];
equation eqSEIR type: SEIR vars: [S,E,I,R,t] params: [N,beta,
  gamma,sigma,mu];
equation eqLV type: LV vars: [x,y,t] params: [alpha,beta,delta,
  gamma] ;

```

- If the `simultaneously: facet` is used, system of all the agents will be solved simultaneously.
- See also: `=`, `solve`,

Embedments

- The `equation` statement is of type: **Sequence of statements or action**
- The `equation` statement can be embedded into: Species, Model,
- The `equation` statement embeds statements: `=`,

error

Facets

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

Definition

The statement makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

Usages

- Throwing an error

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

Embedments

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

event

Facets

- `name` (an identifier), (omissible): the type of event captured: can be “mouse_up”, “mouse_down”, “mouse_move”, “mouse_exit”, “mouse_enter”, “mouse_menu” or a character
- `action` (action): The identifier of the action to be executed in the context of the simulation. This action needs to be defined in ‘global’ or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the pseudo-constant `#user_location`
- `type` (string): Type of 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, mouse_menu}: an unused facet that serves only for the purpose of declaring the string values

Definition

event allows to interact with the simulation by capturing mouse or key events and doing an action. The name of this action can be defined with the ‘action:’ facet, in which case the action needs to be defined in ‘global’ or in the current experiment, without any arguments. The location of the mouse in the world can be retrieved in this action with the pseudo-constant `#user_location`. The statements to execute can also be defined in the block at the end of this statement, in which case they will be executed in the context of the experiment

Usages

- The general syntax is:

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

- For instance:

```
global {
  // ...
  action myAction () {
    point loc <- #user_location; // contains the location of
    the mouse in the world
    list<agent> selected_agents <- agents inside (10#m
    around loc); // contains agents clicked by the event

    // code written by modelers
  }
}

experiment Simple type:gui {
  display my_display {
    event mouse_up action: myAction;
  }
}
```

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

Embedments

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

exhaustive

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only

Definition

This is the standard batch method. The exhaustive mode is defined by default when there is no method element present in the batch section. It explores all the combination of parameter values in a sequential way. See [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the exhaustive statement uses `method exhaustive` instead of the expected `exhaustive name: id :`

```
method exhaustive [facet: value];
```

- For example:

```
method exhaustive maximize: food_gathered;
```

Embedments

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

exit

Facets

Definition

In an FSM architecture, `exit` introduces a sequence of statements to execute right before exiting the state.

Usages

- In the following example, at the state it leaves the state `s_init`, he will display the message 'EXIT from `s_init`':

```
state s_init initial: true {
  write state;
  transition to: s1 when: (cycle > 2) {
    write "transition s_init -> s1";
  }
  exit {
    write "EXIT from "+state;
  }
}
```

- See also: `enter`, `state`, `transition`,

Embedments

- The `exit` statement is of type: **Sequence of statements or action**
 - The `exit` statement can be embedded into: state,
 - The `exit` statement embeds statements:
-

experiment

Facets

- `name` (a label), (omissible) : identifier of the experiment
- `title` (a label):
- `type` (a label), takes values in: {batch, memorize, gui, 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 frequency of the experiment (default value: 1). If frequency: 10, the experiment is executed only each 10 steps.
- `keep_seed` (boolean): Allows to keep the same seed between simulations. Mainly useful for batch experiments
- `keep_simulations` (boolean): In the case of a batch experiment, specifies whether or not the simulations should be kept in memory for further analysis or immediately discarded with only their fitness kept in memory
- `parallel` (any type in [boolean, int]): When set to true, use multiple threads to run its simulations. Setting it to n will set the numbers of threads to use
- `parent` (an identifier): the parent experiment (in case of inheritance between experiments)
- `repeat` (int): In the case of a batch experiment, expresses how many times the simulations must be repeated

- **schedules** (container): A container of agents (a species, a dynamic list, or a combination of species and containers) , which represents which agents will be actually scheduled when the population is scheduled for execution. For instance, ‘species a schedules: (10 among a)’ will result in a population that schedules only 10 of its own agents every cycle. ‘species b schedules: []’ will prevent the agents of ‘b’ to be scheduled. Note that the scope of agents covered here can be larger than the population, which allows to build complex scheduling controls; for instance, defining ‘global schedules: [] { . . . } species b schedules: []; species c schedules: b + world;’ allows to simulate a model where the agents of b are scheduled first, followed by the world, without even having to create an instance of c.
- **skills** (list):
- **until** (boolean): In the case of a batch experiment, an expression that will be evaluated to know when a simulation should be terminated
- **virtual** (boolean): whether the experiment is virtual (cannot be instantiated, but only used as a parent, false by default)

Definition

Declaration of a particular type of agent that can manage simulations. If the experiment directly imports a model using the ‘model:’ facet, this facet *must* be the first one after the name of the experiment

Usages

Embedments

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

explicit

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **parameter_sets** (list): the list of parameter sets to explore; a parameter set is defined by a map: key: name of the variable, value: expression for the value of the variable

Definition

This algorithm run simulations with the given parameter sets

Usages

- As other batch methods, the basic syntax of the `explicit` statement uses `method explicit` instead of the expected `explicit name: id :`

```
method explicit [facet: value];
```

- For example:

```
method explicit parameter_sets:[["a"::0.5, "b"::10],["a"::0.1, "b"::100]];
```

Embedments

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

focus

Facets

- `agent_cause` (agent): the agentCause value of the created belief (can be nil)
- `belief` (predicate): The predicate to focus on the beliefs of the other agent
- `desire` (predicate): The predicate to focus on the desires of the other agent
- `emotion` (emotion): The emotion to focus on the emotions of the other agent
- `expression` (any type): an expression that will be the value kept in the belief
- `id` (string): the identifier of the focus
- `ideal` (predicate): The predicate to focus on the ideals of the other agent
- `is_uncertain` (boolean): a boolean to indicate if the mental state created is an uncertainty
- `lifetime` (int): the lifetime value of the created belief
- `strength` (any type in [float, int]): The priority of the created predicate
- `truth` (boolean): the truth value of the created belief
- `uncertainty` (predicate): The predicate to focus on the uncertainties of the other agent
- `var` (any type in [any type, list, container]): the variable of the perceived agent you want to add to your beliefs
- `when` (boolean): A boolean value to focus only with a certain condition

Definition

enables to directly add a belief from the variable of a perceived specie.

Usages

- Other examples of use:

```
focus var:speed /*where speed is a variable from a species
that is being perceived*/
```

Embedments

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

`focus_on`

Facets

- `value` (any type), (omissible) : The agent, list of agents, geometry to focus on

Definition

Allows to focus on the passed parameter in all available displays. Passing 'nil' for the parameter will make all screens return to their normal zoom

Usages

- Focuses on an agent, a geometry, a set of agents, etc. . .)

```
focus_on my_species (0);
```

Embedments

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

generate

Facets

- **attributes** (map): To specify the explicit link between agent attributes and file based attributes
- **from** (any type): To specify the input data used to inform the generation process. Various data input can be used:
list of `csv_file`: can be aggregated or micro data
matrix: describe the joint distribution of two attributes
genstar generator: a dedicated gaml type to enclose various genstar options all in one
- **species** (any type in [species, agent]), (omissible) : The species of the agents to be created.
- **generator** (string): To specify the type of generator you want to use: as of now there is only DS (or DirectSampling) available
- **number** (int): To specify the number of created agents interpreted as an int value. If facet is omitted or value is 0 or less, generator will treat data used in the 'from' facet as contingencies (i.e. a count of entities) and infer a number to generate (if distribution is used, then only one entity will be created)

Definition

Allows to create a synthetic population of agent from a set of given rules

Usages

- The syntax to create a minimal synthetic population from aggregated file is:

```
synthesis my_species from: [source_file]; attributes: [age::["
  below 18","19 to 45","more than 46"]
synthesis my_species from: my_matrix number: 5 returns:
  list5Agents;
```


Embedments

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

genetic

Facets

- `name` (an identifier), (omissible) : The name of this method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the aggregation method
- `crossover_prob` (float): crossover probability between two individual solutions
- `improve_sol` (boolean): if true, use a hill climbing algorithm to improve the solutions at each generation
- `max_gen` (int): number of generations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `mutation_prob` (float): mutation probability for an individual solution
- `nb_prelim_gen` (int): number of random populations used to build the initial population
- `pop_dim` (int): size of the population (number of individual solutions)
- `stochastic_sel` (boolean): if true, use a stochastic selection algorithm (roulette) rather a deterministic one (keep the best solutions)

Definition

This is a simple implementation of Genetic Algorithms (GA). See the wikipedia article and [batch161 the batch dedicated page]. The principle of the GA is to search an optimal solution by applying evolution operators on an initial population of solutions. There are three types of evolution operators: crossover, mutation and selection. Different techniques can be applied for this selection. Most of them are based on the solution quality (fitness).

Usages

- As other batch methods, the basic syntax of the `genetic` statement uses `method genetic` instead of the expected `genetic name: id:`

```
method genetic [facet: value];
```

- For example:

```
method genetic maximize: food_gathered pop_dim: 5
  crossover_prob: 0.7 mutation_prob: 0.1 nb_prelim_gen: 1
  max_gen: 20;
```

Embedments

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

graphics

Facets

- `name` (a label), (omissible) : the human readable title of the graphics
- `fading` (boolean): Used in conjunction with 'trace:', allows to apply a fading effect to the previous traces. Default is false
- `position` (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point $\{0.5, 0.5, 0.5\}$, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.

- **refresh** (boolean): (openGL only) specify whether the display of the species is refreshed. (true by default, usefull in case of agents that do not move)
- **rotate** (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- **size** (point): extent of the layer in the screen from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in ‘position’, an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **trace** (any type in [boolean, int]): Allows to aggregate the visualization at each timestep on the display. Default is false. If set to an int value, only the last n-th steps will be visualized. If set to true, no limit of timesteps is applied.
- **transparency** (float): the transparency level of the layer (between 0 – opaque – and 1 – fully transparent)
- **visible** (boolean): Defines whether this layer is visible or not

Definition

graphics allows the modeler to freely draw shapes/geometries/texts without having to define a species. It works exactly like a species [Aspect161 aspect]: the draw statement can be used in the same way.

Usages

- The general syntax is:

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

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

Embedments

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

highlight

Facets

- `value` (agent), (omissible) : The agent to highlight
- `color` (rgb): An optional color to highlight the agent. Note that this color will become the default color for further highlight operations

Definition

Allows to highlight the agent passed in parameter in all available displays, optionally setting a color. Passing 'nil' for the agent will remove the current highlight

Usages

- Highlighting an agent

```
highlight my_species(0) color: #blue;
```

Embedments

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

hill_climbing

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **aggregation** (a label), takes values in: {min, max}: the agregation method
- **init_solution** (map): init solution: key: name of the variable, value: value of the variable
- **iter_max** (int): number of iterations
- **maximize** (float): the value the algorithm tries to maximize
- **minimize** (float): the value the algorithm tries to minimize

Definition

This algorithm is an implementation of the Hill Climbing algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the `hill_climbing` statement uses `method hill_climbing` instead of the expected `hill_climbing name: id`:

```
method hill_climbing [facet: value];
```

- For example:

```
method hill_climbing iter_max: 50 maximize : food_gathered;
```

Embedments

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

if

Facets

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

Definition

Allows the agent to execute a sequence of statements if and only if the condition evaluates to true.

Usages

- The generic syntax is:

```
if bool_expr {  
    [statements]  
}
```

- Optionally, the statements to execute when the condition evaluates to false can be defined in a following statement else. The syntax then becomes:

```
if bool_expr {  
    [statements]  
}  
else {  
    [statements]  
}  
string valTrue <- "";  
if true {  
    valTrue <- "true";  
}  
else {  
    valTrue <- "false";  
}  
//valTrue equals "true"  
string valFalse <- "";
```

```
if false {
  valFalse <- "true";
}
else {
  valFalse <- "false";
} //valFalse equals "false"
```

- ifs and elses can be imbricated as needed. For instance:

```
if bool_expr {
  [statements]
}
else if bool_expr2 {
  [statements]
}
else {
  [statements]
}
```

Embedments

- The `if` statement is of type: **Sequence of statements or action**
 - The `if` statement can be embedded into: Behavior, Sequence of statements or action, Layer, Output,
 - The `if` statement embeds statements: `else`,
-

image

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. In case of negative value OpenGL will position the layer out of the environment.
- **refresh** (boolean): (openGL only) specify whether the image display is refreshed or not. (false by default, true should be used in cases of images that are modified over the simulation)
- **rotate** (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- **size** (point): extent of the layer in the screen from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in ‘position’, an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions
- **transparency** (float): the transparency level of the layer (between 0 – opaque – and 1 – fully transparent)
- **visible** (boolean): Defines whether this layer is visible or not

Definition

image allows modeler to display an image (e.g. as background of a simulation). Note that this image will not be dynamically changed or moved in OpenGL, unless the refresh: facet is set to true.

Usages

- The general syntax is:

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



```
}

```

- For instance, in the case of a bitmap image

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

- Or in the case of a shapefile:

```
display my_display {
  image testGIS gis: "../includes/building.shp" color: rgb('
  blue');
}
```

- It is also possible to superpose images on different layers in the same way as for species using `opengl display`:

```
display my_display {
  image image1 file:"../images/image1.jpg";
  image image2 file:"../images/image2.jpg";
  image image3 file:"../images/image3.jpg" position:
  {0,0,0.5};
}
```

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

Embedments

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

inspect

Facets

- **name** (any type), (omissible) : the identifier of the inspector
- **attributes** (list): the list of attributes to inspect. A list that can contain strings or pair<string,type>, or a mix of them. These can be variables of the species, but also attributes present in the attributes table of the agent. The type is necessary in that case
- **refresh** (boolean): Indicates the condition under which this output should be refreshed (default is true)
- **type** (an identifier), takes values in: {agent, table}: the way to inspect agents: in a table, or a set of inspectors
- **value** (any type): the set of agents to inspect, could be a species, a list of agents or an agent

Definition

inspect (and **browse**) statements allows modeler to inspect a set of agents, in a table with agents and all their attributes or an agent inspector per agent, depending on the type: chosen. Modeler can choose which attributes to display. When **browse** is used, type: default value is table, whereas when **inspect** is used, type: default value is agent.

Usages

- An example of syntax is:

```
inspect "my_inspector" value: ant attributes: ["name", "location"];
```

Embedments

- The **inspect** statement is of type: **Output**
- The **inspect** statement can be embedded into: output, permanent, Behavior, Sequence of statements or action,

- The `inspect` statement embeds statements:
-

law

Facets

- `name` (an identifier), (omissible) : The name of the law
- `all` (boolean): add an obligation for each belief
- `belief` (predicate): The mandatory belief
- `beliefs` (list): The mandatory beliefs
- `lifetime` (int): the lifetime value of the mental state created
- `new_obligation` (predicate): The predicate that will be added as an obligation
- `new_obligations` (list): The list of predicates that will be added as obligations
- `parallel` (any type in [boolean, int]): setting this facet to ‘true’ will allow ‘perceive’ to use concurrency with a `parallel_bdi` architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `strength` (any type in [float, int]): The strength of the mental state created
- `threshold` (float): Threshold linked to the obedience value.
- `when` (boolean):

Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

Usages

- Other examples of use:

```
rule belief: new_predicate("test") when: flip(0.5) new_desire:
  new_predicate("test")
```

Embedments

- The `law` statement is of type: **Single statement**
- The `law` statement can be embedded into: Species, Model,
- The `law` statement embeds statements:

layout

Facets

- `value` (any type), (omissible) : Either `#none`, to indicate that no layout will be imposed, or one of the four possible predefined layouts: `#stack`, `#split`, `#horizontal` or `#vertical`. This layout will be applied to both experiment and simulation display views. In addition, it is possible to define a custom layout using the `horizontal()` and `vertical()` operators
- `consoles` (boolean): Whether the consoles are visible or not (true by default)
- `controls` (boolean): Whether the experiment should show its control toolbar on top or not
- `editors` (boolean): Whether the editors should initially be visible or not
- `navigator` (boolean): Whether the navigator view is visible or not (true by default)
- `parameters` (boolean): Whether the parameters view is visible or not (true by default)
- `tabs` (boolean): Whether the displays should show their tab or not
- `toolbars` (boolean): Whether the displays should show their toolbar or not
- `tray` (boolean): Whether the bottom tray is visible or not (true by default)

Definition

Represents the layout of the display views of simulations and experiments

Usages

- For instance, this layout statement will allow to split the screen occupied by displays in four equal parts, with no tabs. Pairs of `display::weight` represent the

number of the display in their order of definition and their respective weight within a horizontal and vertical section

```
layout horizontal([vertical([0::5000,1::5000])::5000,vertical
([2::5000,3::5000])::5000]) tabs: false;
```

Embedments

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

let

Facets

- **name** (a new identifier), (omissible) : The name of the variable declared
- **index** (a datatype identifier): The type of the index if this declaration concerns a container
- **of** (a datatype identifier): The type of the contents if this declaration concerns a container
- **type** (a datatype identifier): The type of the variable
- **value** (any type): The value assigned to this variable

Definition

Allows to declare a temporary variable of the specified type and to initialize it with a value

Usages

Embedments

- The `let` statement is of type: **Single statement**

- The **let** statement can be embedded into: Behavior, Sequence of statements or action, Layer,
 - The **let** statement embeds statements:
-

light

Facets

- **name** (string), (omissible) : The name of the light source, must be unique (otherwise the last definition prevails). Will be used to populate a menu where light sources can be easily turned on and off. Special names can be used: Using the special constant `#ambient` will allow to redefine or control the ambient light intensity and presence Using the special constant `#default` will replace the default directional light of the surrounding display
- **active** (boolean): a boolean expression telling if the light is on or off. (default value if not specified : true)
- **angle** (float): the angle of the spot light in degree (only for spot light). (default value : 45)
- **direction** (point): the direction of the light (only for direction and spot light). (default value : {0.5,0.5,-1})
- **dynamic** (boolean): specify if the parameters of the light need to be updated every cycle or treated as constants. (default value : true).
- **intensity** (any type in [int, rgb]): an int / rgb / rgba value to specify either the color+intensity of the light or simply its intensity. (default value if not specified can be set in the Preferences. If not, it is equal to: (160,160,160,255)).
- **linear_attenuation** (float): the linear attenuation of the positionnal light. (default value : 0)
- **location** (point): the location of the light (only for point and spot light) in model coordinates. Default is {0,0,20}
- **quadratic_attenuation** (float): the quadratic attenuation of the positionnal light. (default value : 0)
- **show** (boolean): If true, draws the light source. (default value if not specified : false).
- **type** (string): the type of light to create. A value among {`#point`, `#direction`, `#spot`}

Definition

`light` allows to define diffusion lights in your 3D display. They must be given a name, which will help track them in the UI. Two names have however special meanings: `#ambient`, which designates the ambient luminosity and color of the scene (with a default intensity of (160,160,160,255) or the value set in the Preferences) and `#default`, which designates the default directional light applied to a scene (with a default medium intensity of (160,160,160,255) or the value set in the Preferences in the direction given by (0.5,0.5,1)). Redefining a light named `#ambient` or `#regular` will then modify these default lights (for example changing their color or deactivating them). To be more precise, and given all the default values of the facets, the existence of these two lights is effectively equivalent to redefining: `light #ambient intensity: gama.pref_display_light_intensity; light #default type: #direction intensity: gama.pref_display_light_intensity direction: {0.5,0.5,-1};`

Usages

- The general syntax is:

```
light 1 type:point location:{20,20,20} color:255,
  linear_attenuation:0.01 quadratic_attenuation:0.0001
  draw_light:true update:false
light 'spot1' type: #spot location:{20,20,20} direction
  :{0,0,-1} color:255 angle:25 linear_attenuation:0.01
  quadratic_attenuation:0.0001 draw:true dynamic: false
light 'point2' type: #point direction:{1,1,-1} color:255 draw:
  true dynamic: false
```

- See also: [display](#),

Embedments

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

loop

Facets

- **name** (a new identifier), (omissible) : a temporary variable name
- **from** (int): an int expression
- **over** (any type in [container, point]): a list, point, matrix or map expression
- **step** (int): an int expression
- **times** (int): an int expression
- **to** (int): an int expression
- **while** (boolean): a boolean expression

Definition

Allows the agent to perform the same set of statements either a fixed number of times, or while a condition is true, or by progressing in a collection of elements or along an interval of integers. Be aware that there are no prevention of infinite loops. As a consequence, open loops should be used with caution, as one agent may block the execution of the whole model.

Usages

- The basic syntax for repeating a fixed number of times a set of statements is:

```
loop times: an_int_expression {  
    // [statements]  
}
```

- The basic syntax for repeating a set of statements while a condition holds is:

```
loop while: a_bool_expression {  
    // [statements]  
}
```

- The basic syntax for repeating a set of statements by progressing over a container of a point is:


```
loop a_temp_var over: a_collection_expression {  
  // [statements]  
}
```

- The basic syntax for repeating a set of statements while an index iterates over a range of values with a fixed step of 1 is:

```
loop a_temp_var from: int_expression_1 to: int_expression_2 {  
  // [statements]  
}
```

- The incrementation step of the index can also be chosen:

```
loop a_temp_var from: int_expression_1 to: int_expression_2  
  step: int_expression3 {  
  // [statements]  
}
```

- In these latter three cases, the name facet designates the name of a temporary variable, whose scope is the loop, and that takes, in turn, the value of each of the element of the list (or each value in the interval). For example, in the first instance of the “loop over” syntax :

```
int a <- 0;  
loop i over: [10, 20, 30] {  
  a <- a + i;  
} // a now equals 60
```

- The second (quite common) case of the loop syntax allows one to use an interval of integers. The from and to facets take an integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the inclusive interval (i.e. [to, from]). If the step is not defined, it is assumed to be equal to 1 or -1, depending on the direction of the range. If it is defined, its sign will be respected, so that a positive step will never allow the loop to enter a loop from i to j where i is greater than j

```
list the_list <-list (species_of (self));
loop i from: 0 to: length (the_list) - 1 {
    ask the_list at i {
        // ...
    }
} // every agent of the list is asked to do something
```

Embedments

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

match

Facets

- `value` (any type), (omissible) : The value or values this statement tries to match

Definition

In a `switch...match` structure, the value of each match block is compared to the value in the switch. If they match, the embedded statement set is executed. Four kinds of match can be used, equality, containment, betweenness and regex matching

Usages

- match block is executed if the switch value is equals to the value of the match:

```
switch 3 {
    match 1 {write "Match 1"; }
    match 3 {write "Match 2"; }
}
```

- `match_between` block is executed if the switch value is in the interval given in value of the `match_between`:

```
switch 3 {
  match_between [1,2] {write "Match OK between [1,2]"; }
  match_between [2,5] {write "Match OK between [2,5]"; }
}
```

- `match_one` block is executed if the switch value is equals to one of the values of the `match_one`:

```
switch 3 {
  match_one [0,1,2] {write "Match OK with one of [0,1,2]"; }
  match_between [2,3,4,5] {write "Match OK with one of
[2,3,4,5]"; }
}
```

- See also: `switch`, `default`,

Embedments

- The `match` statement is of type: **Sequence of statements or action**
- The `match` statement can be embedded into: `switch`,
- The `match` statement embeds statements:

mesh

Facets

- `source` (any type in [file, matrix, species]), (omissible) : Allows to specify the elevation of each cell by passing a grid, a raster, image or csv file or directly a matrix of int/float. The dimensions of the field are those of the file or matrix.
- `border` (rgb): the color to draw lines (borders of cells)

- **color** (any type in [rgb, list, map]): if true, and if neither ‘grayscale’ or ‘texture’ are specified, displays the field using the given color or colors. List of colors, palettes (with interpolation), gradients and scales are supported
- **grayscale** (boolean): if true, gives a grey color to each polygon depending on its elevation (false by default). Supersedes ‘color’ if it is defined.
- **no_data** (float): Can be used to specify a ‘no_data’ value, forcing the renderer to not render the cells with this value. If not specified, that value will be searched in the field to display
- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-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, but should be deactivated if the field is static)
- **rotate** (float): Defines the angle of rotation of this layer, in degrees, around the z-axis.
- **scale** (float): Represents the z-scaling factor, which allows to scale all values of the field.
- **size** (any type in [point, float]): Represents the extent of the layer in the screen from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Like in ‘position’, an elevation can be provided with the z coordinate, allowing to scale the layer in the 3 directions. This latter possibility allows to limit the height of the field. If only a flat value is provided, it is considered implicitly as the z maximal amplitude (or z scaling factor if < 1)
- **smooth** (any type in [boolean, int]): Applies a simple convolution (box filter) to smooth out the terrain produced by this field. If true, one pass is done with a simple 3x3 kernel. Otherwise, the user can specify the number of successive passes (up to 4). Specifying 0 is equivalent to passing false
- **text** (boolean): specify whether the value that represents the elevation is displayed on each cell (false by default)
- **texture** (file): A file containing the texture image to be applied to the field. If not specified, the field will be displayed either in color or grayscale, depending on the other facets
- **transparency** (float): the transparency level of the layer (between 0 – opaque –

and 1 – fully transparent)

- **triangulation** (boolean): specifies whether the cells of the field will be triangulated: if it is false, they will be displayed as horizontal squares at a given elevation, whereas if it is true, cells will be triangulated and linked to neighbors in order to have a continuous surface (false by default)
- **visible** (boolean): Defines whether this layer is visible or not
- **wireframe** (boolean): if true displays the field in wireframe using the lines color

Definition

Allows the modeler to display in an optimized way a field of values, optionally using elevation. Useful for displaying DEMs, for instance, without having to load them into a grid. Can be fed with a matrix of int/float, a grid, a csv/raster/image file and supports many visualisation options

Usages

- The general syntax is:

```
display my_display {
  field a_filename lines: #black position: { 0.5, 0 } size:
  {0.5,0.5} triangulated: true texture: anothe_file;
}
```

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

Embedments

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

migrate

Facets

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

Definition

This command permits agents to migrate from one population/species to another population/species and stay in the same host after the migration. Species of source agents and target species respect the following constraints: (i) they are “peer” species (sharing the same direct macro-species), (ii) they have sub-species vs. parent-species relationship.

Usages

- It can be used in a 3-levels model, in case where individual agents can be captured into group meso agents and groups into clouds macro agents. migrate is used to allows agents captured by groups to migrate into clouds. See the model ‘Balls, Groups and Clouds.gaml’ in the library.

```
migrate ball_in_group target: ball_in_cloud;
```

- See also: [capture](#), [release](#),

Embedments

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

monitor

Facets

- **name** (a label), (omissible) : identifier of the monitor
- **value** (any type): expression that will be evaluated to be displayed in the monitor
- **color** (rgb): Indicates the (possibly dynamic) color of this output (default is a light gray)
- **refresh** (boolean): Indicates the condition under which this output should be refreshed (default is true)

Definition

A monitor allows to follow the value of an arbitrary expression in GAML.

Usages

- An example of use is:

```
monitor "nb preys" value: length(preys 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

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 instaneous
- **intention** (predicate): the intention triggering the norm
- **lifetime** (int): the lifetime of the norm
- **obligation** (predicate): the obligation triggering of the norm
- **priority** (float): the priority value of the norm
- **threshold** (float): the threshold to trigger the norm
- **when** (boolean): the boolean condition when the norm is active

Definition

a norm indicates what action the agent has to do in a certain context and with and obedience value higher than the threshold

Usages

Embedments

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

output

Facets

- **autosave** (any type in [boolean, string]): Allows to save the whole screen on disk. A value of true/false will save it with the resolution of the physical screen. Passing it a string allows to define the filename Note that setting autosave to true (or to any other value than false) in a display will synchronize all the displays defined in the experiment

Definition

output blocks define how to visualize a simulation (with one or more display blocks that define separate windows). It will include a set of displays, monitors and files statements. It will be taken into account only if the experiment type is `gui`.

Usages

- Its basic syntax is:

```
experiment exp_name type: gui {  
  // [inputs]  
  output {  
    // [display, file, inspect, layout or monitor statements  
  ]  
  }  
}
```

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

Embedments

- The **output** statement is of type: **Output**
 - The **output** statement can be embedded into: Model, Experiment,
 - The **output** statement embeds statements: [display](#), [inspect](#), [layout](#), [monitor](#), [output_file](#),
-

output_file

Facets

- **name** (an identifier), (omissible) : The name of the file where you want to export the data

- **data** (string): The data you want to export
- **footer** (string): Define a footer for your export file
- **header** (string): Define a header for your export file
- **refresh** (boolean): Indicates the condition under which this file should be saved (default is true)
- **rewrite** (boolean): Rewrite or not the existing file
- **type** (an identifier), takes values in: {csv, text, xml}: The type of your output data

Definition

Represents an output that writes the result of expressions into a file

Usages

Embedments

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

overlay

Facets

- **background** (rgb): the background color of the overlay displayed inside the view (the bottom overlay remains black)
- **border** (rgb): Color to apply to the border of the rectangular shape of the overlay. Nil by default
- **center** (any type): an expression that will be evaluated and displayed in the center section of the bottom overlay
- **color** (any type in [list, rgb]): the color(s) used to display the expressions given in the 'left', 'center' and 'right' facets
- **left** (any type): an expression that will be evaluated and displayed in the left section of the bottom overlay

- **position** (point): position of the upper-left corner of the layer. Note that if coordinates are in $[0,1[$, the position is relative to the size of the environment (e.g. $\{0.5,0.5\}$ refers to the middle of the display) whereas it is absolute when coordinates are greater than 1 for x and y. The z-ordinate can only be defined between 0 and 1. The position can only be a 3D point $\{0.5, 0.5, 0.5\}$, the last coordinate specifying the elevation of the layer. In case of negative value OpenGL will position the layer out of the environment.
- **right** (any type): an expression that will be evaluated and displayed in the right section of the bottom overlay
- **rounded** (boolean): Whether or not the rectangular shape of the overlay should be rounded. True by default
- **size** (point): extent of the layer in the view from its position. Coordinates in $[0,1[$ are treated as percentages of the total surface of the view, while coordinates > 1 are treated as absolute sizes in model units (i.e. considering the model occupies the entire view). Unlike ‘position’, no elevation can be provided with the z coordinate
- **transparency** (float): the transparency rate of the overlay (between 0 – opaque and 1 – fully transparent) when it is displayed inside the view. The bottom overlay will remain at 0.75
- **visible** (boolean): Defines whether this layer is visible or not

Definition

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

Usages

- To display information in the bottom overlay, the syntax is:

```
overlay "Cycle: " + (cycle) center: "Duration: " +
  total_duration + "ms" right: "Model time: " + as_date(time,
  "") color: [#yellow, #orange, #yellow];
```

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

Embedments

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

parameter

Facets

- `var` (an identifier): the name of the variable (that should be declared in global)
- `name` (a label), (omissible) : The message displayed in the interface
- `among` (list): the list of possible values that this parameter can take
- `category` (a label): a category label, used to group parameters in the interface
- `colors` (list): The colors of the control in the UI. An empty list has no effects. Only used for sliders and switches so far. For sliders, 3 colors will allow to specify the color of the left section, the thumb and the right section (in this order); 2 colors will define the left and right sections only (thumb will be dark green); 1 color will define the left section and the thumb. For switches, 2 colors will define the background for respectively the left 'true' and right 'false' sections. 1 color will define both backgrounds
- `disables` (list): a list of global variables whose parameter editors will be disabled when this parameter value is set to true or to a value that casts to true (they are otherwise enabled)
- `enables` (list): a list of global variables whose parameter editors will be enabled when this parameter value is set to true or to a value that casts to true (they are otherwise disabled)
- `extensions` (list): Makes only sense for file parameters. A list of file extensions (like 'gaml', 'shp', etc.) that restricts the choice offered to the users to certain file types (folders not concerned). Default is empty, effectively accepting all files
- `in_workspace` (boolean): Makes only sense for file parameters. Whether the file selector will be restricted to the workspace or not
- `init` (any type): the init value
- `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
- `updates` (list): a list of global variables whose parameter editors will be updated when this parameter value is changed (their min, max, step and among values will be updated accordingly if they depend on this parameter. Note that it might lead to some inconsistencies, for instance a parameter value which becomes out of range, or which does not belong anymore to a list of possible values. In these cases, the value of the affected parameter will not change)

Definition

The parameter statement specifies which global attributes (i) will change through the successive simulations (in batch experiments), (ii) can be modified by user via the interface (in gui experiments). In GUI experiments, parameters are displayed depending on their type.

Usages

- In gui experiment, the general syntax is the following:

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

- In batch experiment, the two following syntaxes can be used to describe the possible values of a parameter:

```
parameter 'Value of toto:' var: toto among: [1, 3, 7, 15, 100];  
parameter 'Value of titi:' var: titi min: 1 max: 100 step: 2;
```

Embedments

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

perceive

Facets

- `target` (any type in [container, agent]): the list of the agent you want to perceive
- `name` (an identifier), (omissible) : the name of the perception
- `as` (species): an expression that evaluates to a species
- `emotion` (emotion): The emotion needed to do the perception
- `in` (any type in [float, geometry]): a float or a geometry. If it is a float, it's a radius of a detection area. If it is a geometry, it is the area of detection of others species.
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a `parallel_bdi` architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `threshold` (float): Threshold linked to the emotion.
- `when` (boolean): a boolean to tell when does the perceive is active

Definition

Allow the agent, with a bdi architecture, to perceive others agents

Usages

- the basic syntax to perceive agents inside a circle of perception

```

perceive name_of-perception target:
  the_agents_you_want_to_perceive in: a_distance when:
  a_certain_condition {
Here you are in the context of the perceived agents. To refer
to the agent who does the perception, use myself.
If you want to make an action (such as adding a belief for
example), use ask myself{ do the_action}
}

```

Embedments

- The `perceive` statement is of type: **Sequence of statements or action**
- The `perceive` statement can be embedded into: Species, Model,
- The `perceive` statement embeds statements:

permanent

Facets

- `tabs` (boolean): Whether the displays should show their tab or not
- `toolbars` (boolean): Whether the displays should show their toolbar or not

Definition

Represents the outputs of the experiment itself. In a batch experiment, the permanent section allows to define an output block that will NOT be re-initialized at the beginning of each simulation but will be filled at the end of each simulation.

Usages

- For instance, this permanent section will allow to display for each simulation the end value of the `food_gathered` variable:

```

permanent {
  display Ants background: rgb('white') refresh_every: 1 {
    chart "Food Gathered" type: series {
      data "Food" value: food_gathered;
    }
  }
}

```

Embedments

- The `permanent` statement is of type: **Output**
- The `permanent` statement can be embedded into: Experiment,
- The `permanent` statement embeds statements: `display`, `inspect`, `monitor`, `output_file`,

plan

Facets

- `name` (an identifier), (omissible) :
- `emotion` (emotion):
- `finished_when` (boolean):
- `instantaneous` (boolean):
- `intention` (predicate):
- `priority` (float):
- `threshold` (float):
- `when` (boolean):

Definition

define an action plan performed by an agent using the BDI engine

Usages

Embedments

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

ps0

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `iter_max` (int): number of iterations
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `num_particles` (int): number of particles
- `weight_cognitive` (float): weight for the cognitive component
- `weight_inertia` (float): weight for the inertia component
- `weight_social` (float): weight for the social component

Definition

This algorithm is an implementation of the Particle Swarm Optimization algorithm. Only usable for numerical paramaters and based on a continuous parameter space search. See the wikipedia article for more details.

Usages

- As other batch methods, the basic syntax of the `ps0` statement uses `method ps0` instead of the expected `ps0 name: id :`

```
method pso [facet: value];
```

- For example:

```
method pso iter_max: 50 num_particles: 10 weight_inertia:0.7
  weight_cognitive: 1.5 weight_social: 1.5 maximize:
  food_gathered ;
```

Embedments

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

put

Facets

- `in` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression
- `all` (any type): any expression
- `at` (any type): any expression
- `key` (any type): any expression

Definition

Allows the agent to replace a value in a container at a given position (in a list or a map) or for a given key (in a map). Note that the behavior and the type of the attributes depends on the specific kind of container.

Usages

- The allowed parameters configurations are the following ones:

```
put expr at: expr in: expr_container;
put all: expr in: expr_container;
```

- In the case of a list, the position should 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

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
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `nb_tests_wthout_col_max` (int): number of movements without collision before shortening the tabu list
- `tabu_list_size_init` (int): initial size of the tabu list
- `tabu_list_size_max` (int): maximal size of the tabu list
- `tabu_list_size_min` (int): minimal size of the tabu list

Definition

This algorithm is a simple implementation of the Reactive Tabu Search algorithm ((Battiti et al., 1993)). This Reactive Tabu Search is an enhance version of the Tabu search. It adds two new elements to the classic Tabu Search. The first one concerns the size of the tabu list: in the Reactive Tabu Search, this one is not constant anymore but it dynamically evolves according to the context. Thus, when the exploration process visits too often the same solutions, the tabu list is extended in order to favor the diversification of the search process. On the other hand, when the process has

not visited an already known solution for a high number of iterations, the tabu list is shortened in order to favor the intensification of the search process. The second new element concerns the adding of cycle detection capacities. Thus, when a cycle is detected, the process applies random movements in order to break the cycle. See [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the `reactive_tabu` statement uses `method reactive_tabu` instead of the expected `reactive_tabu name: id`:

```
method reactive_tabu [facet: value];
```

- For example:

```
method reactive_tabu iter_max: 50 tabu_list_size_init: 5
  tabu_list_size_min: 2 tabu_list_size_max: 10
  nb_tests_wthout_col_max: 20 cycle_size_min: 2
  cycle_size_max: 20 maximize: food_gathered;
```

Embedments

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

reflex

Facets

- `name` (an identifier), (omissible) : the identifier of the reflex
- `when` (boolean): an expression that evaluates a boolean, the condition to fulfill in order to execute the statements embedded in the reflex.

Definition

Reflexes are sequences of statements that can be executed by the agent. Reflexes prefixed by the 'reflex' keyword are executed continuously. Reflexes prefixed by 'init' are executed only immediately after the agent has been created. Reflexes prefixed by 'abort' just before the agent is killed. If a facet when: is defined, a reflex is executed only if the boolean expression evaluates to true.

Usages

- Example:

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

Embedments

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

release

Facets

- `target` (any type in [agent, list, attributes]), (omissible) : an expression that is evaluated as an agent/a list of the agents to be released or an agent saved as a map
- `as` (species): an expression that is evaluated as a species in which the micro-agent will be released
- `in` (agent): an expression that is evaluated as an agent that will be the macro-agent in which micro-agent will be released, i.e. their new host

- **returns** (a new identifier): a new variable containing a list of the newly released agent(s)

Definition

Allows an agent to release its micro-agent(s). The preliminary for an agent to release its micro-agents is that species of these micro-agents are sub-species of other species (cf. [Species161#Nesting_species Nesting species]). The released agents won't be micro-agents of the calling agent anymore. Being released from a macro-agent, the micro-agents will change their species and host (macro-agent).

Usages

- We consider the following species. Agents of “C” species can be released from a “B” agent to become agents of “A” species. Agents of “D” species cannot be released from the “A” agent because species “D” has no parent species.

```
species A {
...
}
species B {
...
  species C parent: A {
    ...
  }
  species D {
    ...
  }
...
}
```

- To release all “C” agents from a “B” agent, agent “C” has to execute the following statement. The “C” agent will change to “A” agent. They won't consider “B” agent as their macro-agent (host) anymore. Their host (macro-agent) will be the host (macro-agent) of the “B” agent.

```
release list(C);
```

- The modeler can specify the new host and the new species of the released agents:

```
release list (C) as: new_species in: new host;
```

- See also: [capture](#),

Embedments

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

remove

Facets

- `from` (any type in [container, species, agent, geometry]): an expression that evaluates to a container
- `item` (any type), (omissible) : any expression to remove from the container
- `all` (any type): an expression that evaluates to a container. If it is true and the value a list, it removes the first instance of each element of the list. If it is true and the value is not a container, it will remove all instances of this value.
- `index` (any type): any expression, the key at which to remove the element from the container
- `key` (any type): any expression, the key at which to remove the element from the container

Definition

Allows the agent to remove an element from a container (a list, matrix, map...).

Usages

- This statement should be used in the following ways, depending on the kind of container used and the expected action on it:

```
remove expr from: expr_container;
remove index: expr from: expr_container;
remove key: expr from: expr_container;
remove all: expr from: expr_container;
```

- In the case of list, the facet `item:` is used to remove the first occurrence of a given expression, whereas `all` is used to remove all the occurrences of the given expression.

```
list<int> removeList <- [3,2,1,2,3];remove 2 from: removeList;
//removeList equals [3,1,2,3]remove 3 all: true from:
removeList;//removeList equals [1,2]remove index: 1 from:
removeList;//removeList equals [1]
```

- In the case of map, the facet `key:` is used to remove the pair identified by the given key.

```
map<string,int> removeMap <- ["x":5, "y":7, "z":7];remove
key: "x" from: removeMap;//removeMap equals ["y":7, "z
":7]remove 7 all: true from: removeMap;//removeMap equals
map([])
```

- In addition, a map can be managed as a list with pair key as index. Given that, facets `item:`, `all:` and `index:` can be used in the same way:

```
map<string,int> removeMapList <- ["x":5, "y":7, "z":7, "t"
::5];remove 7 from: removeMapList;//removeMapList equals ["
x":5, "z":7, "t":5]remove [5,7] all: true from:
removeMapList;//removeMapList equals ["t":5]remove index:
"t" from: removeMapList;//removeMapList equals map([])
```

- In the case of a graph, both edges and nodes can be removed using `node:` and `edge facets`. If a node is removed, all edges to and from this node are also removed.

```
graph removeGraph <- as_edge_graph
  ([{1,2}::{3,4},{3,4}::{5,6}]);
remove node: {1,2} from: removeGraph;
remove node(1,2) from: removeGraph;
list var <- removeGraph.vertices; // var equals [{3,4},{5,6}]
list var <- removeGraph.edges; // var equals [polyline
  ({3,4}::{5,6})]
remove edge: {3,4}::{5,6} from: removeGraph;
remove edge({3,4},{5,6}) from: removeGraph;
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

Facets

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

Definition

Allows to immediately stop and tell which value to return from the evaluation of the surrounding action or top-level statement (`reflex`, `init`, etc.). Usually used within the declaration of an action. For more details about actions, see the following [Section161 section].

Usages

- Example:

```
string foo {
    return "foo";
}

reflex {
    string foo_result <- foo();    // foos_result is now
    equals to "foo"
}
```

- In the specific case one wants an agent to ask another agent to execute a statement with a return, it can be done similarly to:

```
// In Species A:
string foo_different {
    return "foo_not_same";
}
///  

// In Species B:
reflex writing {
    string temp <- some_agent_A.foo_different []; // temp is
    now equals to "foo_not_same"
}
```

Embedments

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

rotation

Facets

- **angle** (any type in [float, int]), (omissible) : Defines the angle of rotation around the axis. No default defined.
- **axis** (point): The axis of rotation, defined by a vector. Default is {0,0,1} (rotation around the z axis) This facet can be complemented by ‘distance:’ and/or ‘location:’ to specify from where the target is looked at. If ‘target:’ is not defined, the default target is the centroid of the world shape.
- **dynamic** (boolean): If true, the rotation is applied every step. Default is false.
- **location** (point): Allows to define the center of the rotation. Default value is not specified is the center of mass of the world (i.e. {width/2, height/2, max(width, height) / 2})

Definition

camera allows the modeler to define a camera. The display will then be able to choose among the camera defined (either within this statement or globally in GAMA) in a dynamic way. Several preset cameras are provided and accessible in the preferences (to choose the default) or in GAML using the keywords `#from_above`, `#from_left`, `#from_right`, `#from_up_right`, `#from_up_left`, `#from_front`, `#from_up_front`. These cameras are unlocked (so that they can be manipulated by the user), look at the center of the world from a symbolic position, and the distance between this position and the target is equal to the maximum of the width and height of the world's shape. These preset cameras can be reused when defining new cameras, since their names can become symbolic positions for them. For instance: camera 'my_camera' location: `#from_top` distance: 10; will lower (or extend) the distance between the camera and the center of the world to 10. camera 'my_camera' locked: true location: `#from_up_front` target: `people(0)`; will continuously follow the first agent of the people species from the up-front position.

Usages

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

Embedments

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

rule

Facets

- **name** (an identifier), (omissible) : the identifier of the rule

- **when** (boolean): The condition to fulfill in order to execute the statements embedded in the rule. when: true makes the rule always activable
- **priority** (float): An optional priority for the rule, which is used to sort activable rules and run them in that order

Definition

A simple definition of a rule (set of statements which execution depend on a condition and a priority).

Usages

Embedments

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

rule

Facets

- **name** (an identifier), (omissible) : The name of the rule
- **all** (boolean): add a desire for each belief
- **belief** (predicate): The mandatory belief
- **beliefs** (list): The mandatory beliefs
- **desire** (predicate): The mandatory desire
- **desires** (list): The mandatory desires
- **emotion** (emotion): The mandatory emotion
- **emotions** (list): The mandatory emotions
- **ideal** (predicate): The mandatory ideal
- **ideals** (list): The mandatory ideals
- **lifetime** (any type in [int, list]): the lifetime value of the mental state created
- **new_belief** (predicate): The belief that will be added

- `new_beliefs` (list): The belief that will be added
- `new_desire` (predicate): The desire that will be added
- `new_desires` (list): The desire that will be added
- `new_emotion` (emotion): The emotion that will be added
- `new_emotions` (list): The emotion that will be added
- `new_ideal` (predicate): The ideal that will be added
- `new_ideals` (list): The ideals that will be added
- `new_uncertainties` (list): The uncertainty that will be added
- `new_uncertainty` (predicate): The uncertainty that will be added
- `obligation` (predicate): The mandatory obligation
- `obligations` (list): The mandatory obligations
- `parallel` (any type in [boolean, int]): setting this facet to 'true' will allow 'perceive' to use concurrency with a parallel_bdi architecture; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet is true by default.
- `remove_belief` (predicate): The belief that will be removed
- `remove_beliefs` (list): The belief that will be removed
- `remove_desire` (predicate): The desire that will be removed
- `remove_desires` (list): The desire that will be removed
- `remove_emotion` (emotion): The emotion that will be removed
- `remove_emotions` (list): The emotion that will be removed
- `remove_ideal` (predicate): The ideal that will be removed
- `remove_ideals` (list): The ideals that will be removed
- `remove_intention` (predicate): The intention that will be removed
- `remove_obligation` (predicate): The obligation that will be removed
- `remove_obligations` (list): The obligation that will be removed
- `remove_uncertainties` (list): The uncertainty that will be removed
- `remove_uncertainty` (predicate): The uncertainty that will be removed
- `strength` (any type in [float, int, list]): The strength of the mental state created
- `threshold` (float): Threshold linked to the emotion.
- `uncertainties` (list): The mandatory uncertainties
- `uncertainty` (predicate): The mandatory uncertainty
- `when` (boolean):

Definition

enables to add a desire or a belief or to remove a belief, a desire or an intention if the agent gets the belief or/and desire or/and condition mentioned.

Usages

- 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) : Indicates the name of the experiment to run
- `of` (string): Indicates the model containing the experiment to run
- `core` (int): Indicates the number of cores to use to run the experiments
- `end_cycle` (int): Indicates the cycle at which the experiment should stop
- `seed` (int): Provides a predetermined seed instead of letting GAMA choose one
- `with_output` (map):
- `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

Facets

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

Definition

declare the actions an agent execute when enforcing norms of others during a perception

Usages

Embedments

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

save

Facets

- `data` (any type), (omissible) : the data that will be saved to the file

- **attributes** (any type in [map, list]): Allows to specify the attributes of a shape file or GeoJson file where agents are saved. Can be expressed as a list of string or as a literal map. When expressed as a list, each value should represent the name of an attribute of the shape or agent. The keys of the map are the names of the attributes that will be present in the file, the values are whatever expressions needed 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): a boolean expression specifying whether to erase the file if it exists or append data at the end of it. Only applicable to "text" or "csv" files. Default is true
- **to** (string): an expression that evaluates to an string, the path to the file, or directly to a file
- **type** (an identifier), takes values in: {shp, text, csv, asc, geotiff, image, kml, kmz, json, dimacs, dot, gexf, graphml, gml, graph6}: an expression that evaluates to an string, the type of the output file (it can be only "shp", "asc", "geotiff", "image", "text" or "csv")

Definition

Allows to save data in a file. The type of file can be "shp", "asc", "geotiff", "text" or "csv".

Usages

- Its simple syntax is:

```
save data to: output_file type: a_type_file;
```

- To save data in a text file:

```
save (string(cycle) + "->" + name + ":" + location) to: "
  save_data.txt" type: "text";
```

- To save the values of some attributes of the current agent in csv file:

```
save [name, location, host] to: "save_data.csv" type: "csv";
```

- To save the values of all attributes of all the agents of a species into a csv (with optional attributes):

```
save species_of(self) to: "save_csvfile.csv" type: "csv"  
header: false;
```

- To save the geometries of all the agents of a species into a shapefile (with optional attributes):

```
save species_of(self) to: "save_shapefile.shp" type: "shp"  
attributes: ['nameAgent '::name, 'locationAgent '::location]  
crs: "EPSG:4326";
```

- To save the grid_value attributes of all the cells of a grid into an ESRI ASCII Raster file:

```
save grid to: "save_grid.asc" type: "asc";
```

- To save the grid_value attributes of all the cells of a grid into geotiff:

```
save grid to: "save_grid.tif" type: "geotiff";
```

- To save the grid_value attributes of all the cells of a grid into png (with a worldfile):

```
save grid to: "save_grid.png" type: "image";
```

- The save statement can be 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

Facets

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

Definition

Allows to assign a value to the variable or attribute specified

Usages

Embedments

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

setup

Facets

Definition

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

Usages

- As every test should be independent from the others, the setup will mainly contain initialization of variables that will be used in each test.

```
species Tester {
  int val_to_test;

  setup {
    val_to_test <- 0;
  }

  test t1 {
    // [set of instructions, including asserts]
  }
}
```

- See also: [test](#), [assert](#),

Embedments

- The `setup` statement is of type: **Sequence of statements or action**
- The `setup` statement can be embedded into: Species, Experiment, Model,
- The `setup` statement embeds statements:

simulate

Facets

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

Definition

Allows an agent, the sender agent (that can be the [Sections161#global world agent]), to ask another (or other) agent(s) to perform a set of statements. It obeys the following syntax, where the target attribute denotes the receiver agent(s):

Usages

- Other examples of use:

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

Embedments

- The `simulate` statement is of type: **Single statement**
- The `simulate` statement can be embedded into: chart, Experiment, Species, Behavior, Sequence of statements or action,
- The `simulate` statement embeds statements:

sobol

Facets

- **name** (an identifier), (omissible) : The name of the method. For internal use only
- **outputs** (list): The list of output variables to analyse through sobol indexes
- **sample** (an identifier): The size of the sample for the sobol sequence
- **report** (string): The path to the file where the Sobol report will be written
- **results** (string): The path to the file where the automatic batch report will be written

Definition

This algorithm runs a Sobol exploration - it has been built upon the moea framework at <https://github.com/MOEAFramework/MOEAFramework> - disabled the repeat facet of the experiment

Usages

- For example:

```
method sobol sample_size:100 outputs:['my_var'] report:'../  
path/to/report/file.txt';
```

Embedments

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

socialize

Facets

- **name** (an identifier), (omissible) : the identifier of the socialize statement
- **agent** (agent): the agent value of the created social link
- **dominance** (float): the dominance value of the created social link
- **familiarity** (float): the familiarity value of the created social link
- **liking** (float): the appreciation value of the created social link
- **solidarity** (float): the solidarity value of the created social link
- **trust** (float): the trust value of the created social link
- **when** (boolean): A boolean value to socialize only with a certain condition

Definition

enables to directly add a social link from a perceived agent.

Usages

- Other examples of use:

```
socialize;
```

Embedments

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

solve

Facets

- **equation** (an identifier), (omissible) : the equation system identifier to be numerically solved
- **max_step** (float): maximal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- **method** (string): integration method (can be one of “Euler”, “Three-Eighthes”, “Midpoint”, “Gill”, “Luther”, “rk4” or “dp853”, “AdamsBashforth”, “AdamsMoulton”, “DormandPrince54”, “GraggBulirschStoer”, “HighamHall54”) (default value: “rk4”) or the corresponding constant
- **min_step** (float): minimal step, (used with dp853 method only), (sign is irrelevant, regardless of integration direction, forward or backward), the last step can be smaller than this value
- **nSteps** (float): Adams-Bashforth and Adams-Moulton methods only. The number of past steps used for computation excluding the one being computed (default value: 2)
- **scalAbsoluteTolerance** (float): allowed absolute error (used with dp853 method only)
- **scalRelativeTolerance** (float): allowed relative error (used with dp853 method only)
- **step** (float): (deprecated) integration step, use with fixed step integrator methods (default value: 0.005*step)
- **step_size** (float): integration step, use with fixed step integrator methods (default value: 0.005*step)
- **t0** (float): the first bound of the integration interval (default value: cycle*step, the time at the begining of the current cycle.)
- **tf** (float): the second bound of the integration interval. Can be smaller than t0 for a backward integration (default value: cycle*step, the time at the beginning of the current cycle.)

Definition

Solves all equations which matched the given name, with all systems of agents that should solved simultaneously.

Usages

- Other examples of use:

```
solve SIR method: #rk4 step:0.001;
```

Embedments

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

species

Facets

- `name` (an identifier), (omissible) : the identifier of the species
- `cell_height` (float): (grid only), the height of the cells of the grid
- `cell_width` (float): (grid only), the width of the cells of the grid
- `compile` (boolean):
- `control` (skill): defines the architecture of the species (e.g. fsm...)
- `edge_species` (species): In the case of a species defining a graph topology for its instances (nodes of the graph), specifies the species to use for representing the edges
- `file` (file): (grid only), a bitmap file that will be loaded at runtime so that the value of each pixel can be assigned to the attribute 'grid_value'
- `files` (list): (grid only), a list of bitmap file that will be loaded at runtime so that the value of each pixel of each file can be assigned to the attribute 'bands'
- `frequency` (int): The execution frequency of the species (default value: 1). For instance, if frequency is set to 10, the population of agents will be executed only every 10 cycles.
- `height` (int): (grid only), the height of the grid (in terms of agent number)

- **horizontal_orientation** (boolean): (hexagonal grid only),(true by default). Allows use a hexagonal grid with a horizontal or vertical orientation.
- **mirrors** (any type in [list, species]): The species this species is mirroring. The population of this current species will be dependent of that of the species mirrored (i.e. agents creation and death are entirely taken in charge by GAMA with respect to the demographics of the species mirrored). In addition, this species is provided with an attribute called ‘target’, which allows each agent to know which agent of the mirrored species it is representing.
- **neighbors** (int): (grid only), the chosen neighborhood (4, 6 or 8)
- **optimizer** (string): (grid only),(“A ” by default). Allows to specify the algorithm for the shortest path computation (“BF”, “Dijkstra”, “A” or “JPS”)
- **parallel** (any type in [boolean, int]): (experimental) setting this facet to ‘true’ will allow this species to use concurrency when scheduling its agents; setting it to an integer will set the threshold under which they will be run sequentially (the default is initially 20, but can be fixed in the preferences). This facet has a default set in the preferences (Under Performances > Concurrency)
- **parent** (species): the parent class (inheritance)
- **schedules** (container): A container of agents (a species, a dynamic list, or a combination of species and containers) , which represents which agents will be actually scheduled when the population is scheduled for execution. Note that the world (or the simulation) is *always* scheduled first, so there is no need to explicitly mention it. Doing so would result in a runtime error. For instance, ‘species a schedules: (10 among a)’ will result in a population that schedules only 10 of its own agents every cycle. ‘species b schedules: []’ will prevent the agents of ‘b’ to be scheduled. Note that the scope of agents covered here can be larger than the population, which allows to build complex scheduling controls; for instance, defining ‘global schedules: [] {...} species b schedules: []; species c schedules: b;’ allows to simulate a model where only the world and the agents of b are scheduled, without even having to create an instance of c.
- **skills** (list): The list of skills that will be made available to the instances of this species. Each new skill provides attributes and actions that will be added to the ones defined in this species
- **topology** (topology): The topology of the population of agents defined by this species. In case of nested species, it can for example be the shape of the macro-agent. In case of grid or graph species, the topology is automatically computed and cannot be redefined
- **torus** (boolean): is the topology toric (default: false). Needs to be defined on the global species.

- `use_individual_shapes` (boolean): (grid only),(true by default). Allows to specify whether or not the agents of the grid will have distinct geometries. If set to false, they will all have simpler proxy geometries
- `use_neighbors_cache` (boolean): (grid only),(true by default). Allows to turn on or off the use of the neighbors cache used for grids. Note that if a diffusion of variable occurs, GAMA will emit a warning and automatically switch to a caching version
- `use_regular_agents` (boolean): (grid only),(true by default). Allows to specify if the agents of the grid are regular agents (like those of any other species) or minimal ones (which can't have sub-populations, can't inherit from a regular species, etc.)
- `virtual` (boolean): whether the species is virtual (cannot be instantiated, but only used as a parent) (false by default)
- `width` (int): (grid only), the width of the grid (in terms of agent number)

Definition

The species statement allows modelers to define new species in the model. `global` and `grid` are special cases of species: `global` being the definition of the global agent (which has automatically one instance, world) and `grid` being a species with a grid topology.

Usages

- Here is an example of a species definition with a FSM architecture and the additional skill moving:

```
species ant skills: [moving] control: fsm {
```

- In the case of a species aiming at mirroring another one:

```
species node_agent mirrors: list(bug) parent: graph_node
  edge_species: edge_agent {
```

- The definition of the single grid of a model will automatically create gridwidth x gridheight agents:

```
grid ant_grid width: gridwidth height: gridheight file:
  grid_file neighbors: 8 use_regular_agents: false {
```

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

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

Embedments

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

start__simulation

Facets

- `name` (string), (omissible) : The name of the experiment to run
- `of` (string): The path to the model containing the experiment
- `seed` (int):
- `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

Facets

- **name** (an identifier), (omissible) : the identifier of the state
- **final** (boolean): specifies whether the state is a final one (i.e. there is no transition from this state to another state) (default value= false)
- **initial** (boolean): specifies whether the state is the initial one (default value = false)

Definition

A state, like a reflex, can contains several statements that can be executed at each time step by the agent.

Usages

- Here is an exemple integrating 2 states and the statements in the FSM architecture:

```
state s_init initial: true {
  enter { write "Enter in" + state; }
        write "Enter in" + state;
}

write state;

transition to: s1 when: (cycle > 2) {
  write "transition s_init -> s1";
}

exit {
  write "EXIT from "+state;
}
}
state s1 {

  enter {write 'Enter in '+state;}
```

```

    write state;

    exit {write 'EXIT from '+state;}
}

```

- See also: [enter](#), [exit](#), [transition](#),

Embedments

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

status

Facets

- `message` (any type), (omissible) : Allows to display a necessarily short message in the status box in the upper left corner. No formatting characters (carriage returns, tabs, or Unicode characters) should be used, but a background color can be specified. The message will remain in place until it is replaced by another one or by nil, in which case the standard status (number of cycles) will be displayed again
- `color` (rgb): The color used for displaying the background of the status message

Definition

The statement makes the agent output an arbitrary message in the status box.

Usages

- Outputting a message

```
status ('This is my status ' + self) color: #yellow;
```

Embedments

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

switch

Facets

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

Definition

The “switch... match” statement is a powerful replacement for imbricated “if ... else ...” constructs. All the blocks that match are executed in the order they are defined, unless one invokes ‘break’, in which case the switch statement is exited. The block prefixed by default is executed only if none have matched (otherwise it is not).

Usages

- The prototypical syntax is as follows:

```
switch an_expression {
    match value1 {...}
    match_one [value1, value2, value3] {...}
    match_between [value1, value2] {...}
    default {...}
}
```

- Example:


```
switch 3 {
  match 1 {write "Match 1"; }
  match 2 {write "Match 2"; }
  match 3 {write "Match 3"; }
  match_one [4,4,6,3,7] {write "Match one_of"; }
  match_between [2, 4] {write "Match between"; }
  default {write "Match Default"; }
}
```

- See also: [match](#), [default](#), [if](#),

Embedments

- The `switch` statement is of type: **Sequence of statements or action**
- The `switch` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
- The `switch` statement embeds statements: [default](#), [match](#),

tabu

Facets

- `name` (an identifier), (omissible) : The name of the method. For internal use only
- `aggregation` (a label), takes values in: {min, max}: the agregation method
- `init_solution` (map): init solution: key: name of the variable, value: value of the variable
- `iter_max` (int): number of iterations
- `maximize` (float): the value the algorithm tries to maximize
- `minimize` (float): the value the algorithm tries to minimize
- `tabu_list_size` (int): size of the tabu list

Definition

This algorithm is an implementation of the Tabu Search algorithm. See the wikipedia article and [batch161 the batch dedicated page].

Usages

- As other batch methods, the basic syntax of the tabu statement uses `method tabu` instead of the expected `tabu name: id`:

```
method tabu [facet: value];
```

- For example:

```
method tabu iter_max: 50 tabu_list_size: 5 maximize:  
  food_gathered;
```

Embedments

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

task

Facets

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

Definition

As reflex, a task is a sequence of statements that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler chooses a task to execute based on its current priority weight value.

Usages

Embedments

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

test

Facets

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

Definition

The test statement allows modeler to define a set of assertions that will be tested. Before the execution of the embedded set of instructions, if a setup is defined in the species, model or experiment, it is executed. In a test, if one assertion fails, the evaluation of other assertions continue.

Usages

- An example of use:

```
species Tester {
  // set of attributes that will be used in test

  setup {
    // [set of instructions... in particular
    initializations]
  }

  test t1 {
    // [set of instructions, including asserts]
  }
}
```

- See also: `setup`, `assert`,

Embedments

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

trace

Facets

Definition

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

Usages

Embedments

- The `trace` statement is of type: **Sequence of statements or action**

- The `trace` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
 - The `trace` statement embeds statements:
-

transition

Facets

- `to` (an identifier): the identifier of the next state
- `when` (boolean), (omissible) : a condition to be fulfilled to have a transition to another given state

Definition

In an FSM architecture, `transition` specifies the next state of the life cycle. The transition occurs when the condition is fulfilled. The embedded statements are executed when the transition is triggered.

Usages

- In the following example, the transition is executed when after 2 steps:

```
state s_init initial: true {
  write state;
  transition to: s1 when: (cycle > 2) {
    write "transition s_init -> s1";
  }
}
```

- See also: `enter`, `state`, `exit`,

Embedments

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

try

Facets

Definition

Allows the agent to execute a sequence of statements and to catch any runtime error that might happen in a subsequent **catch** block, either to ignore it (not a good idea, usually) or to safely stop the model

Usages

- The generic syntax is:

```
try {  
    [statements]  
}
```

- Optionally, the statements to execute when a runtime error happens in the block can be defined in a following statement 'catch'. The syntax then becomes:

```
try {  
    [statements]  
}  
catch {  
    [statements]  
}
```

Embedments

- The `try` statement is of type: **Sequence of statements or action**
 - The `try` statement can be embedded into: Behavior, Sequence of statements or action, Layer,
 - The `try` statement embeds statements: `catch`,
-

unconscious_contagion

Facets

- `emotion` (emotion): the emotion that will be copied with the contagion
- `name` (an identifier), (omissible) : the identifier of the unconscious contagion
- `charisma` (float): The charisma value of the perceived agent (between 0 and 1)
- `decay` (float): The decay value of the emotion added to the agent
- `receptivity` (float): The receptivity value of the current agent (between 0 and 1)
- `threshold` (float): The threshold value to make the contagion
- `when` (boolean): A boolean value to get the emotion only with a certain condition

Definition

enables to directly copy an emotion presents in the perceived specie.

Usages

- Other examples of use:

```
unconscious_contagion emotion:fearConfirmed;  
unconscious_contagion emotion:fearConfirmed charisma: 0.5  
    receptivity: 0.5;
```

Embedments

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

user__command

Facets

- `name` (a label), (omissible) : the identifier of the user__command
- `action` (action): the identifier of the action to be executed. This action should be accessible in the context in which the user__command is defined (an experiment, the global section or a species). A special case is allowed to maintain the compatibility with older versions of GAMA, when the user__command is declared in an experiment and the action is declared in 'global'. In that case, all the simulations managed by the experiment will run the action in response to the user executing the command
- `category` (a label): a category label, used to group parameters in the interface
- `color` (rgb): The color of the button to display
- `continue` (boolean): Whether or not the button, when clicked, should dismiss the user panel it is defined in. Has no effect in other contexts (menu, parameters, inspectors)
- `when` (boolean): the condition that should be fulfilled (in addition to the user clicking it) in order to execute this action
- `with` (map): the map of the parameters::values required by the action

Definition

Anywhere in the global block, in a species or in an (GUI) experiment, user__command statements allows to either call directly an existing action (with or without arguments) or to be followed by a block that describes what to do when this command is run.

Usages

- The general syntax is for example:

```
user_command kill_myself action: some_action with: [arg1::val1
, arg2::val2, ...];
```

- See also: [user_init](#), [user_panel](#), [user_input](#),

Embedments

- The `user_command` statement is of type: **Sequence of statements or action**
 - The `user_command` statement can be embedded into: `user_panel`, `Species`, `Experiment`, `Model`,
 - The `user_command` statement embeds statements: [user_input](#),
-

`user_init`

Facets

- `name` (an identifier), (omissible) : The name of the panel
- `initial` (boolean): Whether or not this panel will be the initial one

Definition

Used in the user control architecture, `user_init` is executed only once when the agent is created. It opens a special panel (if it contains `user_commands` statements). It is the equivalent to the `init` block in the basic agent architecture.

Usages

- See also: [user_command](#), [user_init](#), [user_input](#),

Embedments

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

`user_input`

Facets

- `init` (any type): the init value
- `returns` (a new identifier): a new local variable containing the value given by the user
- `name` (a label), (omissible) : the displayed name
- `among` (list): the set of acceptable values, only for string inputs
- `max` (float): the maximum value
- `min` (float): the minimum value
- `slider` (boolean): Whether to display a slider or not when applicable
- `type` (a datatype identifier): the variable type

Definition

It allows to let the user define the value of a variable.

Usages

- Other examples of use:

```
user_panel "Advanced Control" {
  user_input "Location" returns: loc type: point <- {0,0};
  create cells number: 10 with: [location::loc];
}
```

- See also: `user_command`, `user_init`, `user_panel`,

Embedments

- The `user_input` statement is of type: **Single statement**
- The `user_input` statement can be embedded into: `user_command`,
- The `user_input` statement embeds statements:

user_panel

Facets

- `name` (an identifier), (omissible) : The name of the panel
- `initial` (boolean): Whether or not this panel will be the initial one

Definition

It is the basic behavior of the user control architecture (it is similar to state for the FSM architecture). This `user_panel` translates, in the interface, in a semi-modal view that awaits the user to choose action buttons, change attributes of the controlled agent, etc. Each `user_panel`, like a state in FSM, can have a enter and exit sections, but it is only defined in terms of a set of `user_commands` which describe the different action buttons present in the panel.

Usages

- The general syntax is for example:

```

user_panel default initial: true {
  user_input 'Number' returns: number type: int <- 10;
  ask (number among list(cells)){ do die; }
  transition to: "Advanced Control" when: every (10);
}

user_panel "Advanced Control" {
  user_input "Location" returns: loc type: point <- {0,0};
  create cells number: 10 with: [location::loc];
}

```

- See also: `user_command`, `user_init`, `user_input`,

Embedments

- The `user_panel` statement is of type: **Behavior**
- The `user_panel` statement can be embedded into: `fsm`, `user_first`, `user_last`, `user_init`, `user_only`, `Species`, `Experiment`, `Model`,
- The `user_panel` statement embeds statements: `user_command`,

using

Facets

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

Definition

`using` is a statement that allows to set the topology to use by its sub-statements. They can gather it by asking the scope to provide it.

Usages

- All the spatial operations are topology-dependent (e.g. neighbors are not the same in a continuous and in a grid topology). So `using` statement allows modelers to specify the topology in which the spatial operation will be computed.

```
float dist <- 0.0;
using topology(grid_ant) {
  d (self.location distance_to target.location);
}
```

Embedments

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

Variable__container

Facets

- **name** (a new identifier), (omissible) : The name of the attribute
- **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

Definition

Allows to declare an attribute of a species or an experiment

Usages

Embedments

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

Variable__number

Facets

- `name` (a new identifier), (omissible) : The name of the attribute
- `among` (list): A list of constant values among which the attribute can take its value
- `category` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `const` (boolean): Indicates whether this attribute can be subsequently modified or not
- `function` (any type in [int, float, point, date]): 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, point, date]): The initial value of the attribute
- `max` (any type in [int, float, point, date]): The maximum value this attribute can take. The value will be automatically clamped if it is higher.
- `min` (any type in [int, float, point, date]): The minimum value this attribute can take. The value will be automatically clamped if it is lower.
- `on_change` (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- `parameter` (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- `step` (any type in [int, float, point, date]): A discrete step (used in conjunction with min and max) that constrains the values this variable can take
- `type` (a datatype identifier): The type of the attribute, either ‘int’, ‘float’, ‘point’ or ‘date’

- **update** (any type in [int, float, point, date]): An expression that will be evaluated each cycle to compute a new value for the attribute

Definition

Allows to declare an attribute of a species or experiment; this type of attributes accepts min:, max: and step: facets, automatically clamping the value if it is lower than min or higher than max.

Usages

Embedments

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

Variable_regular

Facets

- **name** (a new identifier), (omissible) : The name of the attribute
- **among** (list): A list of constant values among which the attribute can take its value
- **category** (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- **const** (boolean): Indicates whether this attribute can be subsequently modified or not
- **function** (any type): Used to specify an expression that will be evaluated each time the attribute is accessed. This facet is incompatible with both 'init:', 'update:' and 'on_change:' (or the equivalent final block)
- **index** (a datatype identifier): The type of the index used to retrieve elements if the type of the attribute is a container type

- **init** (any type): The initial value of the attribute
- **of** (a datatype identifier): The type of the elements contained in the type of this attribute if it is a container type
- **on_change** (any type): Provides a block of statements that will be executed whenever the value of the attribute changes
- **parameter** (a label): Soon to be deprecated. Declare the parameter in an experiment instead
- **type** (a datatype identifier): The type of this attribute. Can be combined with facets 'of' and 'index' to describe container types
- **update** (any type): An expression that will be evaluated each cycle to compute a new value for the attribute

Definition

Allows to declare an attribute of a species or an experiment

Usages

Embedments

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

warn

Facets

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

Definition

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

Usages

- Emitting a warning

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

Embedments

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

write

Facets

- `message` (any type), (omissible) : the message to display. Modelers can add some formatting characters to the message (carriage returns, tabs, or Unicode characters), which will be used accordingly in the console.
- `color` (rgb): The color with wich the message will be displayed. Note that different simulations will have different (default) colors to use for this purpose if this facet is not specified

Definition

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

Usages

- Outputting a message

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

Embedments

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

Chapter 33

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

- **Types (Under Construction)**
 - **Primitive built-in types**
 - * **bool**
 - * **float**

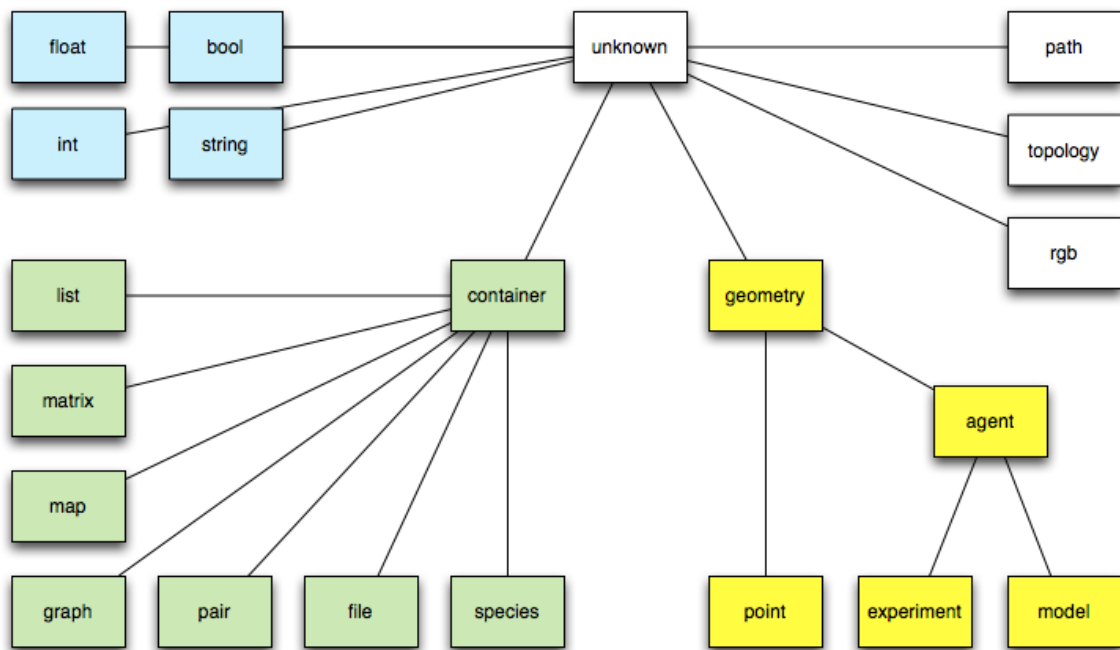


Figure 33.1: images/types_hierarchy.png

- * `int`
- * `string`
- Complex built-in types
 - * `agent`
 - * `container`
 - * `conversation`
 - * `field`
 - * `file`
 - * `geometry`
 - * `graph`
 - * `list`
 - * `map`
 - * `matrix`
 - * `message`

 - * `pair`
 - * `path`
 - * `point`
 - * `rgb`
 - * `species`
 - * Species names as types
 - * `topology`
- Defining custom types

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.
- **Built-in attributes:** these attributes are common to any agent of the simulation
 - `location` (type = `point`): the location of the agent
 - `shape` (type = `geometry`): the shape of the agent
 - `name` (type = `string`): name of the agent (not necessarily unique in its population)
 - `peers` (type = list of agents of the same species): the population of agents of the same species, in the same host, minus the receiver agent
 - `host` (type = `agent`): the agent that hosts the population of the agent

- **Comments:** This datatype is barely used since species name can be directly used as datatypes themselves.
- **Declaration:** the agent casting operator can be applied to any unknown object to cast it as an agent.

[Top of the page](#)

container

- **Definition:** a generic datatype that represents a collection of data.
- **Comments:** a container variable can be a list, a matrix, a map... Conversely, each list, matrix, and map is a kind of container. In consequence, every container can be used in container-related operators.
- **See also:** [Container operators](#)
- **Declaration:**

```
container c <- [1,2,3];
container c <- matrix [[1,2,3],[4,5,6]];
container c <- map ["x"::5, "y"::12];
container c <- list species1;
```

[Top of the page](#)

conversation

- **Definition:** a datatype that represents a conversation between agents in a FIPA-ACL interaction. It contains in particular all the exchanged messages.
- **Built-in attributes:**
 - **messages** (type = list of messages): the list of messages that compose this conversation
 - **protocol** (type = string): the name of the protocol followed by the conversation
 - **initiator** (type = agent): the agent that has initiated this conversation
 - **participants** (type = list of agents): the list of agents that participate to this conversation
 - **ended** (type = bool): whether this conversation has ended or not

[Top of the page](#)

field

- **Definition:** Fields are two-dimensional matrices holding float values. They can be easily created from arbitrary sources (grid, raster or DEM files, matrices grids) and of course by hand. The values they hold are accessible by agents like grids are, using their current location. They can be the target of the ‘diffuse’ statement and can be displayed using the ‘mesh’ layer definition. As such, they represent a lightweight alternative to grids, as they hold spatialized discrete values without having to build agents, which can be particularly interesting for models with large raster data. Several fields can of course be defined, and it makes sense to define them in the global section as, for the moment, they cover by default the whole environment, exactly like grids, and are created alongside them.
- **Built-in attributes:** a field is a kind of matrix, it thus inherits from the matrix’s attributes.
 - dimension (type = point): the dimension (columns x rows) of the receiver matrix
 - columns (type = int): the number of columns of the receiver matrix
 - rows (type = int): the number of rows of the receiver matrix
 - cell_size (type = point): the dimension of an individual cell as a point (width, height). Setting it will only change the interpretation made by the field of the values it contains, but not the values themselves.
 - bands (type = list of field): The list of bands that are optionally present in the field. The first band is the primary field itself, and each of these bands is a field w/o bands
 - no_data (type = float): the value that indicates the absence of data. Setting it will only change the interpretation made by the field of the values it contains, but not the values themselves.
- **See also:** [Field operators](#)
- **Declaration:** a field can be created from a raster datafile (such as .asc or .tif files), a matrix or by specifying its dimensions.
 - a field can be created from a raster datafile “`// Initialize a field from a asc simple raster file field field_from_asc <- field(grid_file(“includes/grid.asc”));`

```
// initialize using a tiff raster file field field_from_tiff <- field(grid_file(“includes/Lesponne.tif”));
```

```
* a field can be created manually:
```

```
// Init from a user defined matrix field field_from_matrix <-
field(matrix([[1,2,3],[4,5,6],[7,8,9]]));
// init an empty field of a given size field empty_field_from_size <- field(10,10);
// init a field for of a given value field full_field_from_size<- field(10,10,1.0);
// init a field of given size, with a given value and no data field full_field_from_
size_with_nodata <- field (1,1,1.0,0.0);
```

```
* a field can be created from a grid of cells, the value
stored will be the grid's grid_value attribute
```

```
global { field field_from_grid <- field(matrix(cell)); } grid cell width: 100 height:
100 { float grid_value <- rnd(1.0,self distance_to world.location); }
```

```
### 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](Operators#files-related-
  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](#table-of-contents)

### 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](Operators#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](#table-of-contents)

### 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](Operators#graph-related-
  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](#table-of-contents)
```

```
### list
```

```
* **Definition:** a composite datatype holding an ordered
  collection of values.
* **Comments:** lists are more or less equivalent to instances
```

```

    of ArrayList in Java (although they are backed up by a
    specific class). They grow and shrink as needed, can be
    accessed via an index (see @ or index\_of), support set
    operations (like union and difference), and provide the
    modeller with a number of utilities that make it easy to
    deal with collections of agents (see, for instance, shuffle
    , reverse, where, sort\_by, ...).
* Remark: lists can contain values of any datatypes,
  including other lists. Note, however, that due to
  limitations in the current parser, lists of lists cannot be
  declared literally; they have to be built using
  assignments. Lists are also a particular kind of container
  and can thus be manipulated using the container operators
  and commands.
* Literal 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 myList <- [1,2,3,4]; myList[2] => 3
```

```

[Top of the page](#table-of-contents)

### 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](#table-of-contents)

### matrix
* **Definition:** a composite datatype that represents either
a two-dimension array (matrix) or a one-dimension array (
vector), holding any type of data (including other matrices
).
* **Built-in attributes:**
* dimension (type = point): the dimension (columns x rows)
of the receiver matrix
* columns (type = int): the number of columns of the
receiver matrix
* rows (type = int): the number of rows of the receiver
matrix
* **Comments:** Matrices are fixed-size structures that can be
accessed by index (point for two-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 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](#table-of-contents)

### message
* **Definition:** a datatype containing a message (sent during a communication, such as the one sent/received in a FIPA interaction).
* **Built-in attributes:**
  * contents (type = unknown): the contents of this message, as a list of arbitrary objects
  * sender (type = unknown): the sender that has sent this message
  * unread (type = bool): whether this message is unread or not
  * emission_timestamp (type = int): the emission time stamp of this message (I.e. at what cycle it has been emitted)
  * reception_timestamp (type = int): the reception time stamp of this message (I.e. at what cycle it has been received)

### pair
* **Definition:** a datatype holding a key and its associated value.
* **Built-in attributes:**
  * key (type = 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.
* **Literal 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](#table-of-contents)

[//]: # (keyword|type_path)
### 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 <- poly-
gon({{10,20},{11,21},{10,21},{11,22}}); path pa <- rect path_to poly; //
built a path between rect and poly, in the topology
// of the current agent (i.e. a line in a& continuous topology, // a path in a graph in
a graph topology )

```

a_topology path_between a_container_of_geometries // idem with an explicit topology and the possibility // to have more than 2 geometries // (the path is then built incrementally)

path_between (a_graph, a_source, a_target) // idem with a the given graph as topology

```

[Top of the page](#table-of-contents)

[//]: # (keyword|type_point)
### 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](#table-of-contents)
```

```

[//]: # (keyword|type_rgb)
### 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.
* **Literal 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 <- 0xRRGGBB. 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](#table-of-contents)
```

```
[//]: # (keyword|type_species)
```

```
### 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](#table-of-contents)
```

```
### 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](#table-of-contents)

[//]: # (keyword|type_topology)
### 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; } }
```

```
[//]: # (startConcept|load_complex_datas)
# 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:

```
sepalwidth sepalwidth petalwidth 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

* [File Types](#file-types)
  * [Text File](#text-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [CSV File](#csv-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [Shapefile](#shapefile)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [OSM File](#osm-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [Grid File](#grid-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [Image File](#image-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
  * [SVG File](#svg-file)
```



```
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
* [Property File](#property-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
* [R File](#r-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
* [3DS File](#3ds-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)
* [OBJ File](#obj-file)
    * [Extensions](#extensions)
    * [Content](#content)
    * [Operators](#operators)

## Text File
### Extensions
Here the list of possible extensions for text file:

* "txt"
* "data"
* "csv"
* "text"
* "tsv"
* "xml"

Note that when trying to define the type of a file with the
default file loading operator (**file**), GAMA will first
try to test the other type of file. For example, for files
with ".csv" extension, GAMA will cast them as csv file and
not as text file.

### Content
```

The content of a `text file` is a list of `string` corresponding to each line of the `text file`.

For example:

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

will give:

```
sepalwidth,sepalwidth,petalwidth,petalwidth,type      5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa 4.7,3.2,1.3,0.2,Iris-setosa
```

Operators

List of operators related to `text` files:

- * `**text_file(string path)**`: load a `file` (with an authorized extension) as a `text file`.
- * `**text_file(string path, list content)**`: load a `file` (with an authorized extension) as a `text file` and fill it with the given content.
- * `**is_text(op)**`: tests whether the operand is a `text file`

CSV File

Extensions

Here the list of possible extensions for `csv file`:

- * `"csv"`
- * `"tsv"`

Content

The content of a `csv file` is a `matrix` of objects: each row of the `matrix` is a line of the `file`; each column of the `matrix` is values delimited by the separator. By `default`, the delimiter is the `"`,`"` and the datatype depends on the dataset.

For example:

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

will give:

```
sepalength sepalwidth petallength petalwidth type 5.1 3.5 1.4 0.2 Iris-setosa 4.9 3.0
1.4 0.2 Iris-setosa ...
```

To manipulate easily the `data`, we can consider the `contents`` of the `data file`, that is a `matrix`.

As an example, we can access the `number of lines` and columns of a `data file` named ``my_file`` with ``my_file.contents.dimension``.

Operators

There are many operators available to load a `csv_file`.

- * `**csv_file(string path)**`: load a `file` (with an authorized extension) as a `csv file` with `default` separator ("`,`"), and no assumption on the `type` of `data`.
- * `**csv_file(string path,bool header)"**`: load a `file` as a `CSV file` with the `default separator` (coma), with specifying if the `model` has a `header` or not (boolean), and no assumption on the `type` of `data`.
- * `**csv_file(string path, string separator)**`: load a `file` (with an authorized extension) as a `csv file` with the given `separator`, without making any assumption on the `type` of `data`. `Headers` should be detected automatically if they exist.

```
file my_file <- csv_file("../includes/data.csv", ",");
```

- * `**csv_file(string path, string separator, bool header)**`: load a `file` (with an authorized extension) as a `csv file`, specifying (1) the `separator` used; (2) if the `model` has a `header` or not, without making any assumption on the `type` of `data`.
- * `**csv_file(string path, string separator, string text_qualifier, bool header)**`: load a `file` as a `csv file` specifying (1) the `separator` used; (2) the `text` `qualifier` used; (3) if the `model` has a `header` or not, without making any assumption on the `type` of `data`",

```
* **csv\_file(string path, string separator, type datatype)
**: load a file as a csv file specifying a given separator,
no header, and the type of data. No text qualifier will be
used.
```

```
file my_file <- csv_file("../includes/data.csv", ";", int);
```

```
* **csv\_file(string path, string separator, string
text_qualifier, type datatype)**: load a file as a csv file
specifying the separator, text qualifier to use, and the
type of data to read. Headers should be detected
automatically if they exist.
* **csv\_file(string path, string separator, type datatype,
bool header)**: load a file as a csv file specifying the
given separator, the type of data, with specifying if the
model has a header or not (boolean). No text qualifier will
be used".
* **csv\_file(string path, string separator, type datatype,
bool header, point dimensions)**: load a file as a csv file
specifying a given separator, the type of data, with
specifying the number of cols and rows taken into account.
No text qualifier will be used.
* **csv\_file(string path, matrix content)**: This file
constructor allows to store a matrix in a CSV file (it does
not save it - just store it in memory)
```

Finally, it is possible to check whether a file is a csv file:

```
* **is\_csv(op)**: tests whether the operand is a csv file
```

```
## Shapefile
```

```
Shapefiles are classical GIS data files. A shapefile is not
simple file, but a set of several files (source: wikipedia)
:
```

```
* Mandatory files :
* .shp - shape format; the feature geometry itself
* .shx - shape index format; a positional index of the
feature geometry to allow seeking forwards and backwards
quickly
```

```

    * .dbf - attribute format; columnar attributes for each
    shape, in dBase IV format
  * Optional files :
    * .prj - projection format; the coordinate system and
    projection information, a plain text file describing the
    projection using well-known text format
    * .sbn and .sbx - a spatial index of the features
    * .fbn and .fbx - a spatial index of the features for
    shapefiles that are read-only
    * .ain and .aih - an attribute index of the active fields
    in a table
    * .ixs - a geocoding index for read-write shapefiles
    * .mxs - a geocoding index for read-write shapefiles (ODB
    format)
    * .atx - an attribute index for the .dbf file in the form
    of shapefile.columnname.atx (ArcGIS 8 and later)
    * .shp.xml - geospatial metadata in XML format, such as
    ISO 19115 or other XML schema
    * .cpg - used to specify the code page (only for .dbf) for
    identifying the character encoding to be used

More details about shapefiles can be found [here](http://en.
wikipedia.org/wiki/Shapefile).

### Extensions
Here the list of possible extension for shapefile:

  * "shp"

### Content
The content of a shapefile is a list of geometries
  corresponding to the objects of the shapefile.
For example:

```

```

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

```

```

will give:

```

```

Polygon Polygon Polygon Polygon Polygon Polygon Polygon ...

```

Note that the `attributes` of each object of the shapefile are stored in their corresponding GAMA `geometry`. The operator `"get"` (or `"read"`) allows to get the `value` of corresponding `attributes`.

For example:

```
file my_file <- shape_file("../includes/data.shp"); write "my_file:" + my_-
file.contents; loop el over: my_file { write (el get "TYPE"); }
```

Operators

List of operators related to shapefiles:

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

```
file my_file <- shape_file("../includes/data.shp", "EPSG:32601");
```

- * `**shape_file(string path, list content)**`: load a `file` (with an authorized extension) as a shapefile and `fill` it with the given content.
- * `**is_shape(op)**`: tests whether the operand is a shapefile

OSM File

OSM (Open Street Map) is a collaborative project to `create` a free editable `map` of the world. The `data` produced in this

project (OSM File) represent physical features on the ground (e.g., roads or buildings) using tags attached to its basic data structures (its nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way or relation (source: openstreetmap.org).

More details about OSM data can be found [here](http://wiki.openstreetmap.org/wiki/Map_Features).

Extensions

Here the list of possible extension for shapefile:

```
* "osm"
* "pbf"
* "bz2"
* "gz"
```

Content

The content of an OSM data is a list of geometries corresponding to the objects of the OSM file.

For example:

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

will give:

```
Point Point Point Point Point LineString LineString Polygon Polygon Polygon ...
```

Note that like for shapefiles, the attributes of each object of the osm file is stored in their corresponding GAMA geometry. The operator "get" (or "read") allows to get the value of corresponding attributes.

Operators

List of operators related to osm file:

```
* **osm\_file(string path)**: load a file (with an authorized extension) as an osm file with default
```

```

projection (if a prj file is defined, use it, otherwise use
the default projection defined in the preference). In this
case, all the nodes and ways of the OSM file will become a
geometry.
* **osm\_file(string path, string code)**: load a file (with
an authorized extension) as an osm file with the given
projection (GAMA will automatically decode the code. For a
list of the possible projections see: http://
spatialreference.org/ref/). In this case, all the nodes and
ways of the OSM file will become a geometry.
* **osm\_file(string path, int EPSG\_ID)**: load a file (
with an authorized extension) as an osm file with the given
projection (GAMA will automatically decode the epsg code.
For a list of the possible projections see: http://
spatialreference.org/ref/). In this case, all the nodes and
ways of the OSM file will become a geometry.

```

```
file my_file <- osm_file("../includes/data.gz", "EPSG:32601");
```

```

* **osm\_file(string path, map filter)**: load a file (with
an authorized extension) as an osm file with default
projection (if a prj file is defined, use it, otherwise use
the default projection defined in the preference). In this
case, only the elements with the defined values are loaded
from the file.

```

```
//map used to filter the object to build from the OSM file according to attributes.
map filtering <- map(["highway"::["primary", "secondary", "tertiary", "motorway",
"living_street", "residential", "unclassified"], "building"::["yes"]]);
```

```
//OSM file to load file 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](http://en.wikipedia.org/wiki/Esri_grid).

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 ...

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](http://en.
  wikipedia.org/wiki/Scalable_Vector_Graphics).
```

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:

```
sepalength = 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](http://www.r-project.
org/).

Note that GAMA also integrates some operators to manage R
scripts:
* [R\_compute](Operators#R_compute)
* [R\_compute\_param](Operators#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](http://en.
  wikipedia.org/wiki/.3ds).

### 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](http://en.
  wikipedia.org/wiki/Wavefront_.obj_file).

### 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

[//]: # (endConcept|load_complex_datas)
```

Expressions

Expressions in GAML are the **value** part of the [statements](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](Literals), a [unit](UnitsAndConstants), a [constant](UnitsAndConstants), a [variable](PseudoVariables), an [attribute](VariablesAndAttributes) or the application of one or several [operators](Operators) to compose a complex **expression**.

```
\part{Tutorials}
```

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](GamlReference)
around when undertaking one of these tutorials.

* Predator Prey
* Road Traffic
* 3D Tutorial
* Luneray's flu
* Incremental Model
* BDI architecture

## [Predator Prey tutorial](PredatorPrey)

![resources/images/tutorials/predator_pre.png](resources/
images/tutorials/predator_pre.png)

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](http://
www.scholarpedia.org/article/Agent_based_modeling)). It is
particularly adapted to beginners that want to quickly
learn how to build a simple model in GAMA.

## [Road Traffic](RoadTrafficModel)

![resources/images/tutorials/road_traffic.png](resources/
images/tutorials/road_traffic.png)

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](ThreeD)
```

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](LuneraysFlu)
```

```
![resources/images/tutorials/Luneray.jpg](resources/images/tutorials/Luneray.jpg)
```

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.

```
## [Incremental Model](IncrementalModel)
```

```
![resources/images/tutorials/incremental_model.jpg](resources/images/tutorials/incremental_model.jpg)
```

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](BDIAgents)
```

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 their agents, taking into account their emotions and social relationships.

```
# Some pedagogical materials

## Initiation to algorithms with Scratch
A set of exercises for your first step to algorithms using the
  graphical tool Scratch: [PDF](resources/other/
  PedagogicalMaterial/InitiationtoAlgorithmicswithScratch.pdf
  ).

## Memo GAML
A summary of the organization of a GAML model, its main parts,
  and the main keywords, statements: [PDF](resources/other/
  PedagogicalMaterial/1.8.2/MementoAlgoGAMLv1.8.2.pdf).

## Class materials
This is a set of slides used to introduce GAMA in a practical
  way to master students in Computer Science. (Only PDF files
  are provided as the sources can be very heavy. Do not
  hesitate to ask them if needed).

1. [Introduction to Agent-Based Modeling and simulation](
  resources/other/PedagogicalMaterial/Courses/1-Intro_ABMS.
  pdf)
2. [Introduction to the GAMA Platform](resources/other/
  PedagogicalMaterial/Courses/2-Intro_GAMA.pdf)
3. [A modeling process cycle](resources/other/
  PedagogicalMaterial/Courses/3-Modeling_process.pdf)
4. [First steps in GAMA with the Schelling model](resources/
  other/PedagogicalMaterial/Courses/4-Schelling_model.pdf)
  * **A correction:** [gama project](resources/other/
  PedagogicalMaterial/Courses/Models/Course-Exercice-
  Schelling.zip)
5. [Introduction of grids in GAMA models with the ChouChevLoup
  model](resources/other/PedagogicalMaterial/Courses/5-
  ChouchevLoup.pdf)
  * **Data file:** [Environment asc file](resources/other/
  PedagogicalMaterial/Courses/Data/hab10.asc)
6. [Introduction of GIS data in GAMA models, using a traffic
  model](resources/other/PedagogicalMaterial/Courses/6-
  Traffic_model.pdf)
```

```

    * **Data files:** [Environment files](resources/other/
PedagogicalMaterial/Courses/Data/DataTraffic.zip)
    * **A correction:** [gama project](resources/other/
PedagogicalMaterial/Courses/Models/Course-Exercice-Traffic.
zip)
7. [Application exercise: Evacuation of the Phuc Xa district
of Hanoi](resources/other/PedagogicalMaterial/Courses/7-
Exercice_model-Evacuation_of_Phuc_Xa.pdf)
    * **Data files:** [Environment files](resources/other/
PedagogicalMaterial/Courses/Data/DataPhucXa.zip)

## First Exercices (*)
A set of exercises going from the building of simple models (
opinion diffusion) to training exercises about specific
modeling aspects (iterators on containers, scheduling...)
* **Keywords:** grid, displays, plot, containers, iterators,
scheduling.
* **Subject:** [PDF](resources/other/PedagogicalMaterial
/1.8.2/PedagogicalMaterials-Exercices-GAMAv1.8.2.pdf)
* **A correction:** [gaml files](c)

## Exercice (*): Firefly synchronization
From UML diagram, implement a GAMA model simulating the
synchronization of fireflies.

* **Keywords:** grid, displays, plot, synchronization.
* **Subject:** [PDF](resources/other/PedagogicalMaterial
/1.8.2/Exercice-FireFly/Fireflies-Subject.pdf)
* **A correction:** [gaml file](resources/other/
PedagogicalMaterial/1.8.2/Exercice-FireFly/luciole_on_grid.
gaml)

## 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](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Firefighter/Firefighter-
  Model-description.pdf)
* Guide:** [PDF](resources/other/PedagogicalMaterial/1.8.2/
  Exercice-Firefighter/Firefighter-Guide.pdf)
* A correction:** [gaml file](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Firefighter/Exercice-
  Firefighters.zip)

## Exercice (**): Wolves, Goats, Cabbages model
Implement an extended version of the Prey-Predator model. It
allows you to manipulate grids and inheritance.

* Keywords:** grid, inheritance, displays, plot, prey-
  predator model.
* Detailed model description:** [PDF](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-ChouChevLoup/
  ChouchevLoup-DetailedDescription.pdf)
* Data file:** [Environment asc file](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-ChouChevLoup/hab10.asc)
* A correction:** [gaml file](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-ChouChevLoup/Exercice-
  ChouChevLoup.zip)

## Exercice (**): Schelling model
Implement the segregation Schelling model on an environment (
either a grid or a shafile).

* Keywords:** grid, GIS data, displays, plot, Graphical
  modeling, Schelling model.
* Subject:** [PDF](resources/other/PedagogicalMaterial
  /1.8.2/Exercice-Schelling/MISSABMS_2014-Schelling.pdf)
* Data file:** [Environment files](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Schelling/buildings.zip)
* A correction:** [gaml file](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Schelling/Exercice-
  Schelling.zip)

## Exercice (**): Traffic model

```

```

* **Keywords:** GIS data, graph, skills, moving skill,
  displays, plot, mobility model.
* **Subject:** [PDF](resources/other/PedagogicalMaterial
  /1.8.2/Exercice-Traffic/MISSABMS2014-Traffic.pdf)
* **Data file:** [Environment files](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Traffic/Datafiles.zip)
* **A correction:** [gaml file](resources/other/
  PedagogicalMaterial/1.8.2/Exercice-Traffic/Exercice-Traffic
  .zip)

## Exercice (**): Shortest path on a grid by distance
diffusion

An algorithm-oriented exercise to compute shortest paths on a
grid.

* **Keywords:** grid, move, displays, diffusion model,
  algorithm.
* **Subject:** [PDF](resources/other/PedagogicalMaterial
  /1.8.2/Exercice-ShortestPathDiffusion/
  Shortest_Path_on_Grid_by_diffusion.pdf)
* **A model:** [gaml file](resources/other/PedagogicalMaterial
  /1.8.2/Exercice-ShortestPathDiffusion/Exercice-
  ShortestPathDiffusion.zip)

# 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
introduces 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.

![Tutorial models Predator-Prey.](resources/images/tutorials/
predator_prey.png)

## 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 GAML
code of the model.

1. [Basic model (prey agents)](PredatorPrey_step1)
1. [Dynamic of the vegetation (grid)](PredatorPrey_step2)
1. [Behavior of the prey agent](PredatorPrey_step3)
1. [Use of Inspectors/monitors](PredatorPrey_step4)
1. [Predator agents (parent species)](PredatorPrey_step5)
1. [Breeding of prey and predator agents](PredatorPrey_step6)
1. [Agent display (aspect)](PredatorPrey_step7)
1. [Complex behaviors for the preys and predators](
PredatorPrey_step8)
1. [Adding of a stopping condition](PredatorPrey_step9)
1. [Definition of charts](PredatorPrey_step10)
1. [Writing files](PredatorPrey_step11)
1. [Image loading (raster data)](PredatorPrey_step12)
1. [Exploration of the model](PredatorPrey_step13)

# 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 Library models (`Tutorials`, `Road Traffic`).

If you are not familiar with agent-based models or GAMA, we advise you to have a look at the [prey-predator](PredatorPrey) model first.

Model Overview

The model built in this tutorial concerns the study of 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.

![Road traffic tutorial: a screenshot of the final state of the model.](resources/images/tutorials/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)](RoadTrafficModel_step1)


```

1. [Definition of people agents](RoadTrafficModel_step2)
1. [Movement of the people agents](RoadTrafficModel_step3)
1. [Definition of weight for the road network](
  RoadTrafficModel_step4)
1. [Dynamic update of the road network](RoadTrafficModel_step5
)
1. [Definition of a chart display](RoadTrafficModel_step6)
1. [Automatic repair of roads](RoadTrafficModel_step7)

# 3D Tutorial

This tutorial introduces the 3D features offered by GAMA.

## Model Overview

![Result of the 3D tutorial.](resources/images/tutorials/3
D_model_3.png)](http://www.youtube.com/watch?feature=
player\_embedded&v=6Z1BU6xTcfw)

## 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](ThreeD_step1)
1. [Moving cells](ThreeD_step2)
1. [Moving cells with neighbors](ThreeD_step3)

# 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 aim at simulating 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 a 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.

![Final display of the tutorial: Incremental model.](resources/images/tutorials/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](IncrementalModel_step1)
1. [Charts](IncrementalModel_step2)
1. [Integration of GIS Data](IncrementalModel_step3)

```
1. [Movement on Graph](IncrementalModel_step4)
1. [Visualizing in 3D](IncrementalModel_step5)
1. [Multi-Level](IncrementalModel_step6)
1. [Differential Equations](IncrementalModel_step7)

# 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](resources/other/models/Luneray_flu.zip).

The importation of models is described [here](ImportingModels)
.

## Model Overview
The model built in this tutorial concerns the flu spreading in
the city of Luneray (Normandie, France).

![Introduction to the Luneray's flu tutorial models.](
resources/images/tutorials/Luneray.jpg)

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 their 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
* 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 a 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](LuneraysFlu_step1)
1. [Definition of monitors and chart outputs](LuneraysFlu_step2)
1. [Importation of GIS data](LuneraysFlu_step3)
1. [Use of a graph to constraint the movements of people](LuneraysFlu_step4)
1. [Definition of 3D displays](LuneraysFlu_step5)
1. [Exploration of the model](LuneraysFlu_step6)

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](Using-BEN-simple-bdi).

If you are not familiar with agent-based models or GAMA we advise you to have a look at the [prey-predator](PredatorPrey) 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](BDIAgents_step1)
1. [Definition of the BDI miners](BDIAgents_step2)
1. [Definition of social relations between miners](BDIAgents_step3)
1. [Use of emotions and personality for the miners](BDIAgents_step4)
1. [Adding norms, obligations and enforcement](BDIAgents_step5)

\part{Developing GAMA}

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](GamaArchitecture) gives a general overview of the organization of Java packages and Eclipse plugins, and should be read first. In the following subsections we give a more practical introduction.

- * 1. [Introduction to GAMA Java API](Introduction-To-Gama-Java-API)
 - * 1. [Installing the GIT version](InstallingGitVersion)
- * 2. [Developing Extensions](DevelopingExtensions)
- * 2. [Create a release of Gama](CreatingAReleaseOfGama)
- * 3. [Generation of the documentation](Documentation)

Introduction to GAMA Java API

This introduction to the Java API is dedicated to programmers that want to participate in the java code of GAMA. The main purpose is to describe the main packages and classes of the API to makes it simple to find such crucial information such as: how GAMA create containers, agent and geometries, how exceptions and log are managed, how java code maintain Type safety, etc.

Table of content

[Concepts](#Concepts)

1. [Factories](#Factories)
2. [Spatial](#Spatial)
3. [Type](#Type)

```

4. [IScope](#IScope)
5. [Exception](#Exception)
6. [Debug](#Debug)
7. [Test](#Test)

[Packages](#Packages)

* 1.[Core](#Core)

***
# 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](https://
github.com/gama-platform/gama/blob/master/msi.gama.core/src
/msi/gaml/types/Types.java).

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](https://github.com/gama-platform/gama/
blob/master/msi.gama.core/src/msi/gama/util/GamaListFactory
.java))

As an example:

```

```
IList distribution = GamaListFactory.create(Types.FLOAT);
```

```
To create `List` object without specifying the type, use `
```

```
Types.NO_TYPE` :
```

```
IList result = GamaListFactory.create(Types.NO_TYPE);
```

```
or only:
```

```
IList result = GamaListFactory.create();
```

```
2. GamaMapFactory : factory to create map of different type (
  see [Java class](https://github.com/gama-platform/gama/blob
  /master/msi.gama.core/src/msi/gama/util/GamaMapFactory.java
  ))
```

As an example:

```
final IMap<String, IList<?>» ncdData = 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](https://bitbucket.org/
trove4j/trove/src/master/core/src/main/java/gnu/trove/map/)
```

```
4. TLinkedHashSet : see [trove api](https://bitbucket.org/
trove4j/trove/src/master/core/src/main/java/gnu/trove/set/
hash/)
```


5. Stream : you can use java build-in streams but there is a special version in Gama taken from [StreamEx](<https://github.com/amaembo/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](OperatorsSZ#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](https://github.com/gama-
  -platform/gama/blob/master/msi.gama.core/src/msi/gaml/types
  /Types.java), 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](https://github.com/gama-platform/gama/tree/GAMA_1.8.2/msi.gama.core/src/msi/gama/runtime/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](https://github.com/gama-platform/gama/tree/GAMA_1.8.2/ummisco.gama.annotations/src/ummisco/gama/dev/utils)
```

```
- 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
```

```
***
# 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](https://github.com/gama-platform/gama/tree/GAMA_1.8.2/msi.gama.core/src/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

# 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](DevelopingPlugins) (in the OSGI sense, see [
  here](http://www.eclipse.org/equinox/)) 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.

- * 1. [Developing Plugins](DevelopingPlugins)
- * 2. [Developing Skills](DevelopingSkills)
- * 3. [Developing Statements](DevelopingStatements)
- * 4. [Developing Operators](DevelopingOperators)
- * 5. [Developing Types](DevelopingTypes)
- * 6. [Developing Species](DevelopingSpecies)
- * 7. [Developing Control Architectures](DevelopingControlArchitectures)
- * 8. [IScope](DevelopingIScope)
- * 9. [Index of annotations](DevelopingIndexAnnotations)

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"`

```
# Generation of the documentation
```

```
## Table of contents
```

```
* [Requirements](#requirements)
  * [Configuration](#configuration)
  * [Generated files location](#generated-files-location)
```

```

* [Workflow to generate wiki files](#workflow-to-generate-wiki-files)
* [Workflow to generate PDF files](#workflow-to-generate-pdf-files)
* [Workflow to generate unit tests](#workflow-to-generate-unit-tests)
* [Main internal steps](#main-internal-steps)
  * [Generate wiki files](#generate-wiki-files)
  * [Generate pdf files](#generate-pdf-files)
  * [Generate unit test files](#generate-unit-test-files)
* [How to document](#how-to-document)
  * [The @doc annotation](#the-doc-annotation)
  * [the @example annotation](#the-example-annotation)
  * [How to document operators](#how-to-document-operators)
  * [How to document statements](#how-to-document-statements)
  * [How to document skills](#how-to-document-skills)
* [How to change the processor](#how-to-change-the-processor)
* [General workflow of file generation](#general-workflow-of-file-generation)

```

Documentation

The GAMA documentation comes in 2 formats: a set of wiki files available from the wiki section of the GitHub website and a PDF file. The PDF file is produced from the wiki files.

In the wiki files, some are hand-written by the GAMA community and some others are generated automatically from the Java code and the associated java annotations.

The section summarizes:

- * how to generate this wiki files,
- * how to generate the PDF documentation,
- * how to generate the unit tests from the java annotations,
- * how to add documentation in the java code.

Requirements

To generate automatically the documentation, the GAMA Git version is required. See [Install Git version](InstallingGitVersion) 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](<http://pandoc.org/>). Follow the [Pandoc installation instructions to install it](<http://pandoc.org/installing.html>). 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](<http://miktex.org/download>) (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 generated from the Java code thanks to
 the Java additional processor, using mainly information
 from Java classes or methods and from the Java annotations.
 (see [the list of all annotations](
 DevelopingIndexAnnotations) for more details about
 annotations).

The `@doc` annotation

Most of the annotations can contain a [`@doc`](
 DevelopingIndexAnnotations#doc) annotation, that can
 contain the main part of the documentation.

For example, the `inter` ([inter](Operators#inter)) operator
 is commented using:
```java
@doc(
  value = "the intersection of the two operands",

```

```

comment = "both containers are transformed into sets (so
without duplicated element, cf. remove_duplicates operator)
before the set intersection is computed.",
usages = {
  @usage(value = "if an operand is a graph, it will be
transformed into the set of its nodes"),
  @usage(value = "if an operand is a map, it will be
transformed into the set of its values", examples = {
    @example(value = "[1::2, 3::4, 5::6] inter [2,4]",
equals = "[2,4]"),
    @example(value = "[1::2, 3::4, 5::6] inter [1,3]",
equals = "[]") }),
  @usage(value = "if an operand is a matrix, it will be
transformed into the set of the lines", examples =
    @example(value = "matrix([[1,2,3],[4,5,4]]) inter [3,4]"
, equals = "[3,4]")) },
examples = {
  @example(value = "[1,2,3,4,5,6] inter [2,4]", equals = "[
2,4]"),
  @example(value = "[1,2,3,4,5,6] inter [0,8]", equals = "[]
") },
see = { "remove_duplicates" })

```

This @docannotation contains 5 parts:

- value: describes the documented element,
- comment: a general comment about the documented element,
- usages: a set of ways to use the documented element, each of them being in a @usage annotation. The usage contains mainly a description and set of examples,
- examples: a set of examples that are not related to a particular usage,
- see: other related keywords.

the @example annotation

This annotation contains a particular use example of the documented element. It is also used to generate unit test and patterns.

The simplest way to use it:

```
@example(value = "[1::2, 3::4, 5::6] inter [2,4]", equals = "[2,4]")
```

In this example:

- `value` contains an example of use of the operator,
- `equals` contains the expected results of expression in value.

This will become in the documentation:

```
list var3 <- [1::2, 3::4, 5::6] inter [2,4]; // var3 equals [2,4]
```

When no variable is given in the annotation, an automatic name is generated. The type of the variable is determined thanks to the return type of the operator with these parameters.

This example can also generate a unit test model. In this case, the value in the variable will be compared to the `equals` part.

By default, the `@example` annotation has the following default values:

- `isTestOnly = false`, meaning that the example will be added to the documentation too,
- `isExecutable = true`, meaning that content of `value` can be added in a model and can be compiled (it can be useful to switch it to false, in a documentation example containing name of species that have not been defined),
- `test = true`, meaning that the content of value will be tested to the content of equals,
- `isPattern = false`.

How to document operators

A GAML operator is defined by a Java method annotated by the `@operator` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the operators are defined in the plugin `msi.gama.core` and in the package `msi.gaml.operators`.

The documentation generator will use information from:

- the `@operator` annotation:
 - **value**: it provides the name(s) of the operator (if an operator has several names, the other names will be considered as alternative names)
 - **category**: it is used to 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 annotated by the `@symbol` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the statements are defined in the plugin `msi.gama.core` and in the package `msi.gaml.statements`.

The documentation generator will use information from:

- `@symbol` annotation,
- `@facets` annotation (each facet can contain a documentation in a `@doc` annotation),
- `@inside` annotation (where the statement can be used),
- `@doc` annotation

How to document skills

A GAML skill is defined by a Java class annotated by the `@skill` annotation (see [the list of all annotations](#) for more details about annotations). In the core of GAMA, most of the skills are defined in the plugin `msi.gama.core` and in the package `msi.gaml.skills`.

The documentation generator will use information from:

- `@skill` annotation,
- `@vars` annotation (each var can contain a documentation in a `@doc` annotation),
- `@doc` annotation

How to change the processor

If you make some modifications in the plugin processor, you have to rebuild the .jar file associated to the processor to take into account the changes. Here are the several steps you have to do:

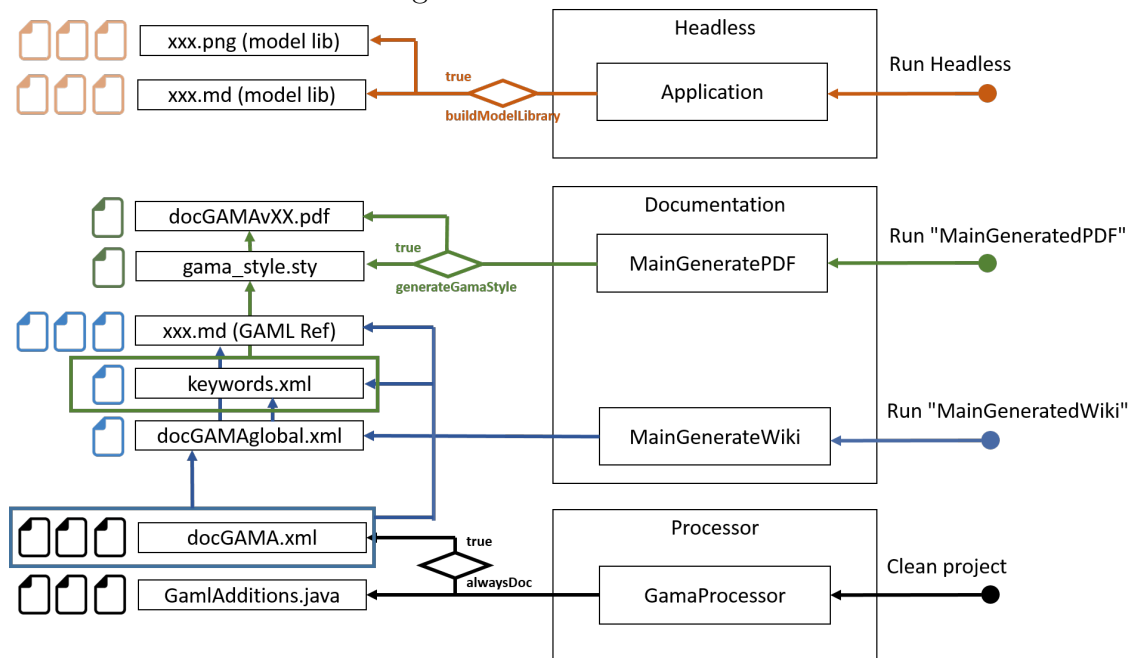
- In the `msi.gama.processor` plugin, click on `Generate Processor.jardesc` (in `processor`)
- Click on `Finish` (you can check that `msi.gama.processor` and `ummisco.gama.annotations` are checked). Accept the warning popup.
- It should have changed the `processor / plugins / msi.gama.processor_1.4.0.jar` file.
- Right-click on the folder `processor` to refresh.

In case some projects have errors after the update of the processor: * Clean and build the projects * Close Eclipse and reopen it and clean and build the projects * Check that Eclipse has been launched with the same JVM as GAMA. To this purpose, have a look at `Eclipse / About Eclipse, Installation details` and check the java version (i.e. after the `-vm` option). If it does not fit with the one used for eclipse plugin, change it (in the `eclipse.ini` file).

Chapter 34

General workflow of file generation

This following diagram explains roughly the work-flow for the generation of the different files:



Part VI

Projects using GAMA

Chapter 35

Projects

Publications

This page is an attempt to list the projects using the GAMA platform as a modeling and simulation platform. Interesting readers can also have a look at [the page listing the Ph.D. theses and articles related to and/or using GAMA platform.](#)

Projects

SWITCH: Simulating the transition of transport Infrastructures Toward smart and sustainable Cities (ANR 2019-)

Description:

Transport infrastructures play a large part in defining the city of the future, which should be smart, sustainable and resilient. Their management will need to deal with the emergence of novel technologies (i.e. autonomous cars, Internet of Things) and the increase of novel modalities and practices (increase of multi-modality, electric bicycles, shared cars). These aspects could favour and accelerate the transition to the city of the future with positive social, environmental and economic impacts, in order to address foreseen trends (climate change and new requirements in terms of pollution, security, and global costs). The SwITCh project aims at supporting decision-making for urban planning by simulating the gradual introduction of disruptive innovations

on technology, usage and behaviour of infrastructure. It requires providing a model that is able to assess the impact of these innovations on several key indicators on mobility, user satisfaction and security, economic costs and air pollution. SwITCh integrates a large variety of urban transport modalities (private car, walk, tramway, etc.) and associated infrastructures (pavement, bicycle path, etc.). Achieving such an objective requires building a model that includes current and future infrastructures and modalities, and considering the transition process between current and future situations. SwITCh uses agent- based modelling (ABM) and participative simulation as a unifying framework that allows coupling different models and taking into account both temporal and spatial scales in order to build a holistic model. It will include a city model based on real geographic data (GIS) and a complex realistic model of population behaviour. The model will be designed as a support tool for helping stakeholders (i.e. decision-makers, managers, technicians and citizens) to enrich their reflection and build a shared project to improve transport infrastructures to meet the challenges of future cities. The SwITCh project will be centred on the design and on the implementation of an ABM that will result in an interactive simulator and a serious game. The interactive simulator will be used by the city planners to explore the potential impact of innovations in various evolutionary contexts. It will thus support the urban planning team in making relevant decisions regarding the evolution of their transport infrastructures, by letting them test and assess different alternatives and situations. The interactive simulator will also allow the researchers to highlight potential futures or unexpected side effects to the urban planners and other stakeholders, based on a participatory simulation approach. The serious game will be used by students and the larger public in order to enrich their understanding of the issues involved in the city of the future and the transport infrastructures. It will be based on the interactive simulator but will be enhanced by specific work on the game design in order to be a real support for learning and raising awareness. The interactive simulator and the serious game will be developed with the GAMA open-source platform and will be used in a real context for two case studies: Bordeaux Metropole and the Urban Community of Dijon. The SwITCh project will deliver several main results. Firstly, it will generate and formalize knowledge on future transport infrastructures. Secondly, the project will result in a simulation tool that could have significant socio-economic impacts: by helping infrastructure managers and urban planners, as a reflection support, to adapt infrastructures to future needs, by accelerating the transition to a more sustainable city which should have positive environmental (e.g. air pollution, global warming), economical (e.g. maintenance cost, commercial appeal) and social (e.g. traffic, living environment) impacts. The model will be flexible, easily adaptable to any city, and able to integrate a wide variety of

prospective and disruptive scenarios.

Website: <https://www6.inrae.fr/switch>

Contact: Franck Taillandier

COMOKIT (2020-)

Description:

Since its emergence in China, the COVID-19 pandemic has spread rapidly around the world. Faced with this unknown disease, public health authorities were forced to experiment, in a short period of time, with various combinations of interventions at different scales. However, as the pandemic progresses, there is an urgent need for tools and methodologies to quickly analyze the effectiveness of responses against COVID-19 in different communities and contexts. In this perspective, computer modelling appears to be an invaluable lever as it allows for the *in silico* exploration of a range of intervention strategies prior to the potential field implementation phase. More specifically, we argue that, in order to take into account important dimensions of policy actions, such as the heterogeneity of the individual response or the spatial aspect of containment strategies, the branch of computer modeling known as agent-based modelling is of immense interest. We present in this paper an agent-based modelling framework called COVID-19 Modelling Kit (COMOKIT), designed to be generic, scalable, and thus portable in a variety of social and geographical contexts. COMOKIT combines models of person-to-person and environmental transmission, a model of individual epidemiological status evolution, an agenda-based one-hour time step model of human mobility, and an intervention model. It is designed to be modular and flexible enough to allow modellers and users to represent different strategies and study their impacts in multiple social, epidemiological or economic scenarios.

COMOKIT aims at supporting deciders in answering the most pressing of these questions using an integrated model that combines:

- * A sub-model of individual clinical dynamics and epidemiological status
- * A sub-model of direct transmission of the infection from agent to agent
- * A sub-model of environmental transmission through the built environment
- * A sub-model of policy and interventions design and implementation
- * An agenda-based model of people activities at a one-hour time step

Website: <https://comokit.org/>

Contact: Alexis Drogoul

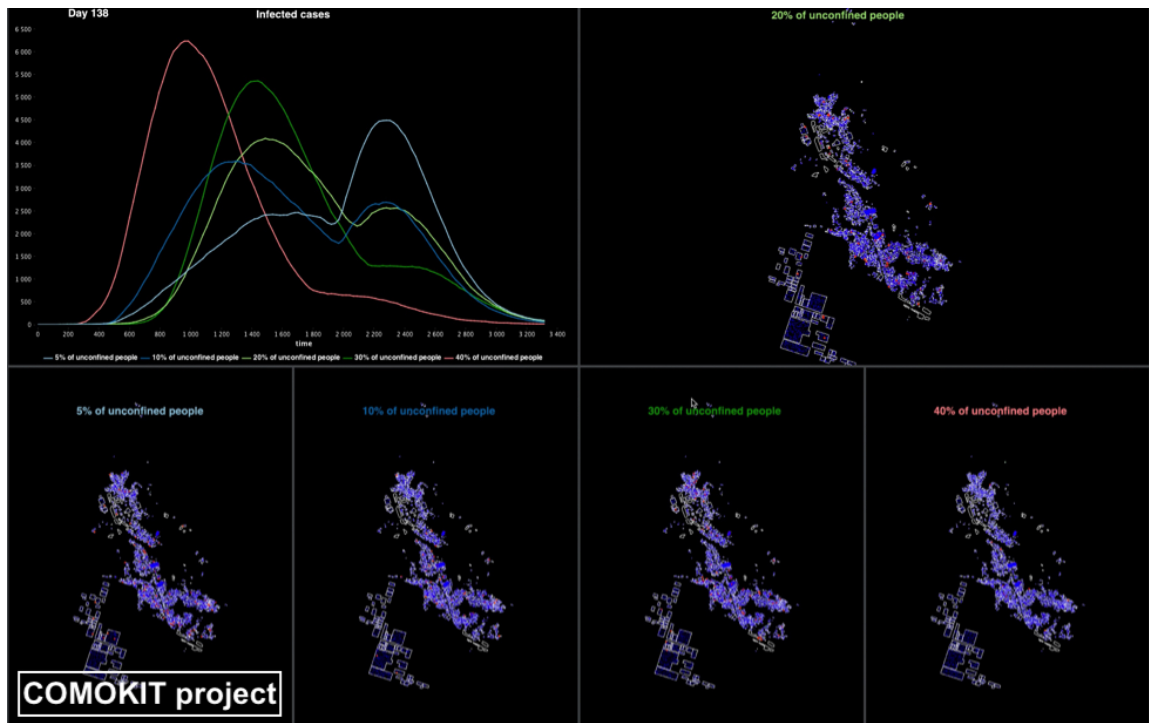


Figure 35.1: COMOKIT visualisation

TSH-system project (2021-)

Description: The main objective of the project is to develop decision-making tools to support urban Transport-Spaces-Humans system planning and design.

The project aims to assist in: 1. analysing the interdependence between the urban transportation system, public space system, and their users both qualitatively and quantitatively at multiple spatial dimensions; 2. quantifying the impact of architecture layouts and transport-land use plans; 3. predicting the usage of different land uses, activity supports, automobile travel demands, active travel demands, and transport mode choice; 4. supporting the selection of plan scenarios.

Publication:

Website: <https://nmyangliu.wixsite.com/tsh-system>

Contact: Liu Yang (Research Fellow in the School of Architecture, Southeast University (Nanjing, China))

LittoSIM / LittoGEN / LittoKong

Description:

LittoSIM is a participatory simulation platform for local actors. The serious game is presented in the form of a simulation integrating both a model of marine submersion, the modeling of actors acting on the territory (defense association, State services, etc.), and game actions performed in situ by elected officials and technicians (municipalities and inter-municipal authorities). The simulation focuses on the southwestern tip of the island of Oléron and offers a reflection on the effects of types of land use planning on the management of the risk of submersion (frontal defenses, modes of urbanization, soft defenses, withdrawal strategic). The game aims to explore different scenarios for managing the risks of submersion, the course of which is induced both by the players' layout choices and by the simulation as such, thus constraining the trajectories of the game.

This companion modeling is an opportunity to provide testimony on the learning effects that the game allows for elected officials and managers, and therefore on the apprehension of space and territory in situ by the actors of the development.

The LittoSIM project has been extended on several other case studies in France (through the LittoGEN project) and in Vietnam (through the LittoKONG project).

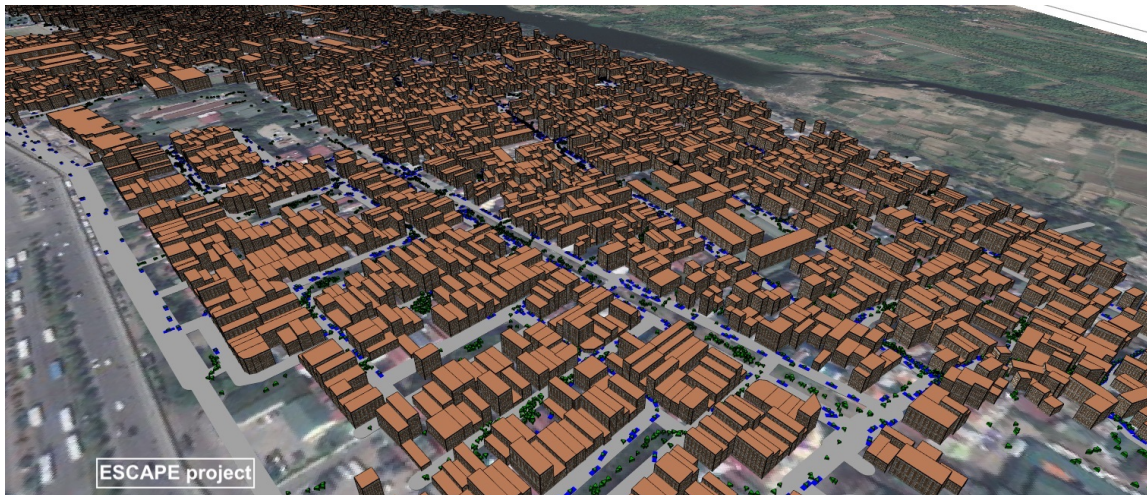


Figure 35.2: ESCAPE Hanoi visualisation

Publication: * 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>

Website: <https://littosim.hypotheses.org/>

Contact: Nicolas Bécu and Marion Amalric

ESCAPE: Exploring by Simulation Cities Awareness on Population Evacuation (ANR 2016-2020)

Description:

A summary is available on the [ANR website](#).

Publication: * 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.

Contact: Eric Daudé

HoanKiemAir (French Embassy 2019-2020)

Description:

The development of permanent or temporary pedestrian areas, whether for leisure or to decrease air pollution, has become an integral part of urban planning in numerous cities around the world. Hanoi, the capital of Vietnam, began to implement its first area, around the iconic Hoan Kiem lake, a few years ago. In most cases, however, a road closure is likely to deport traffic to nearby neighborhoods with the consequences of intensifying congestion and, possibly, increasing air pollution in these areas. Because this outcome might appear counter-intuitive to most stakeholders, it is becoming more and more necessary to analyze, assess, and share the impacts of these developments in terms of traffic and pollution shifts before implementing them. In the HoanKiemAir project, we used the GAMA platform to build an agent-based model that simulates the traffic, its emissions of air pollutants, and the diffusion of these pollutants in the district of Hoan Kiem. This simulation has been designed so as to serve either as a decision support tool for local authorities or as an awareness-raising tool for the general public: thanks to its display on a physical 3D model of the district, people can effectively and very naturally interact with it at public venues. Although still in progress, the simulation is already able to reflect traffic and air pollution peaks during rush hours, allowing residents and developers to understand the impact of pedestrianization on air quality in different scenarios.

Publication: * [Duc, P.M., Chapuis, K., Drogoul, A., Gaudou, B., Grignard, A., Marilleau, N. and Nguyen-Huu, T., 2020. HoanKiemAir: simulating impacts of urban management practices on traffic and air pollution using a tangible agent-based model. In 2020 RIVF International Conference on Computing and Communication Technologies \(RIVF\) \(pp. 1-7\). IEEE.](#)

Contact: Benoit Gaudou

ACTEUR: Cognitive Territorial Agents for the Study of Urban Dynamics and Risks (ANR 2014-2018)

Description:

Every year, the number of urban residents is growing. Diverse questions related to sustainability are rise from this growth. For example, for large and attractive territories, which urban planning policies to implement? How to manage and prevent technological or environmental hazards? Decision-makers have to take all of these

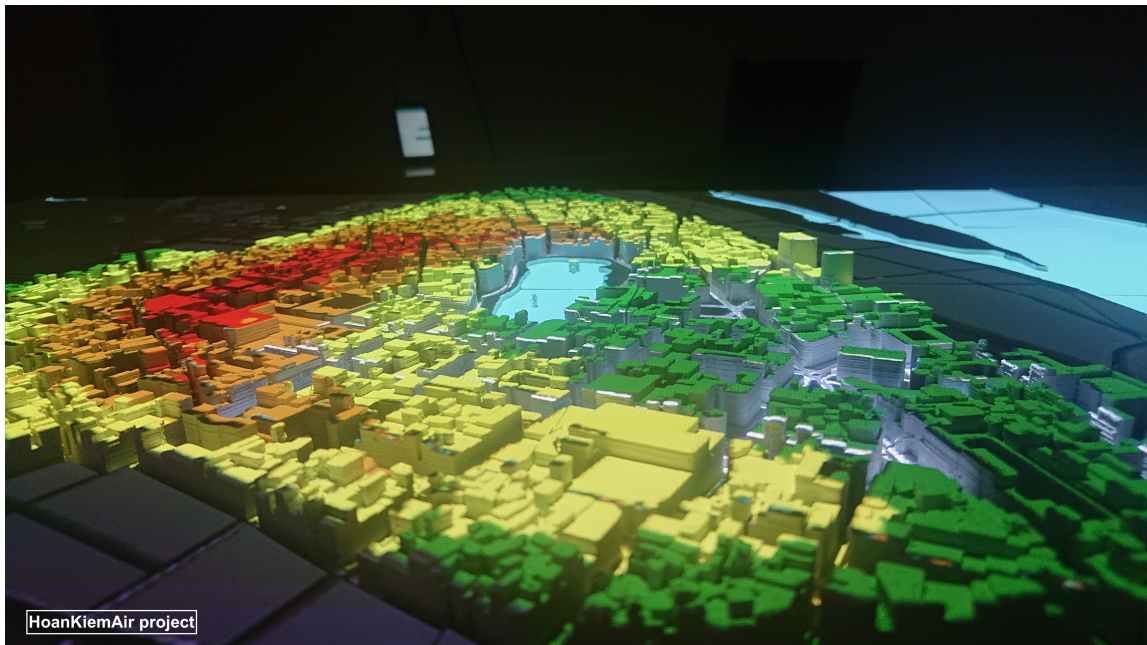


Figure 35.3: HKA visualisation

issues into account when defining their urban planning policies. Unfortunately, the assessment of the impacts of possible policies is difficult due to the complex and stochastic interplay between society and infrastructure. One of the most promising approaches to face this difficulty is agent-based modeling. This approach consists in modeling the studied system as a collection of interacting decision-making entities called agents. An agent-based model can provide relevant information about the dynamics of the real-world urban system it represents. Moreover, it can allow them to be used as a virtual laboratory to test new urban planning policies. The use of agent-based models to study urban systems is booming for the last ten years. Another tendency is the development of more and more realist models. However, if models have to make a lot of progress concerning the integration of geographical and statistical data, the agents used to represent the different actors influencing the dynamic of the system (inhabitants, decision-makers. . .) are often simplistic (reactive agents). Yet, for some urban models, being able to integrate these cognitive agents, i.e. agents able to make complex reasoning such as planning to achieve their goals, is mandatory to improve the realism of models and test new scenarios. Unfortunately, developing large-scale models that integrate cognitive agents requires high-level programming skills. Indeed, if there are nowadays several software platforms that propose to help

modelers to define their agent-based models through a dedicated modeling language (Netlogo, GAMA...) or through a graphical interface (Starlogo TNG, Modelling4All, Repast Symphony, MAGéo...), none of them are adapted to the development of such models by modelers with low-level programming skills: either they are too complex to use (Repast, GAMA) or too limited (Netlogo, Starlogo TNG, Modelling4All, Repast Symphony, MAGéo). As a result, geographers and urban planners that have no programming skills have to rely on computer scientists to develop models, which slows the development and the use of complex and realist spatial agent-based models. The objective of the ACTEUR project is to develop to help modelers, in particular geographers and urban planners, to design and calibrate through a graphical language cognitive agents able to act in a complex spatial environment. The platform has also for ambition to be used as a support of model discussion -participatory modeling- between the different actors concerned by a model (geographers, sociologists, urban planners, decision-makers, representatives...). These tools will be integrated into the GAMA platform that enables us to build large-scale models with thousands of hundreds of agents and that was already used to develop models with cognitive agents. In order to illustrate the utility and the importance of the developed tools, we will use them in two case studies. The first concerns the urban evolution of La Réunion island. The second case study will focus on the adaption to industrial hazards in Rouen. These two case studies are part of funded projects carried out by partners of the ACTEUR project.

Website:**Contact:** Patrick Taillandier

Genstar (ANR 2014-2016)

Description:

The Gen* project has the ambition to propose tools and methods to generate realistic synthetic populations for agent-based social simulation: it aims at combining applied mathematics and computer science approaches in order to incorporate arbitrary data and to generate statistically valid populations of artificial agents.

Publication: * 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

Website: <http://www.irit.fr/genstar/>

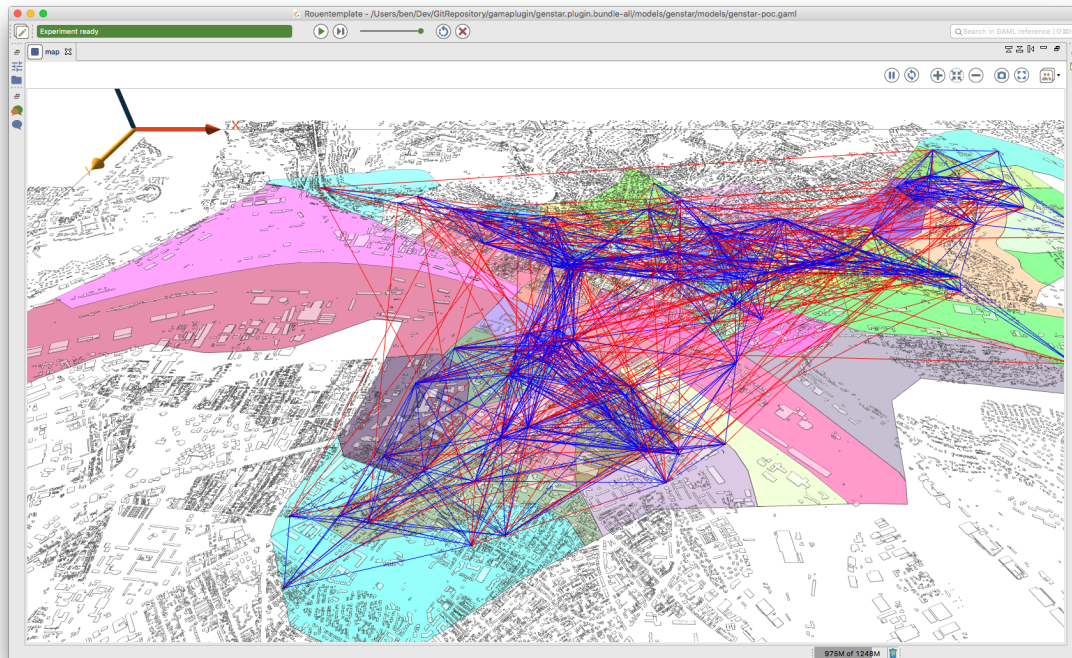


Figure 35.4: GENSTAR visualisation

Contact: Alexis Drogoul & Kevin Chapuis

MAELIA

Description:

Maelia is a multi-agent platform for integrated assessment and modeling of agricultural territories (landscape) and territorial bioeconomy systems. It enables to assess the environmental, economic and social impacts of the combined changes in agricultural activities, transformation and recycling of biomass, natural resource management strategies (e.g. water) and global (demography, dynamics of land cover and climate changes).

Currently, this platform allows to handle at fine spatio-temporal scales the interactions between agricultural activities (rotation and crop management strategies within each production system), the hydrology of the different water resources (based on the SWAT® model's formalisms) and the water resources management (water withdrawals,

restrictions, choices between resources). It is currently used to assess the impacts of scenarios (i) of distribution of agro-ecological cropping systems on green and blue water, nitrogen and carbon flows in watersheds, and (ii) of production exchanges between arable and livestock farmers on individual and collective environmental and socio-economic performances.

Publication: * 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.

Website: <http://maelia-platform.inra.fr/>

Contact: Olivier Thérond

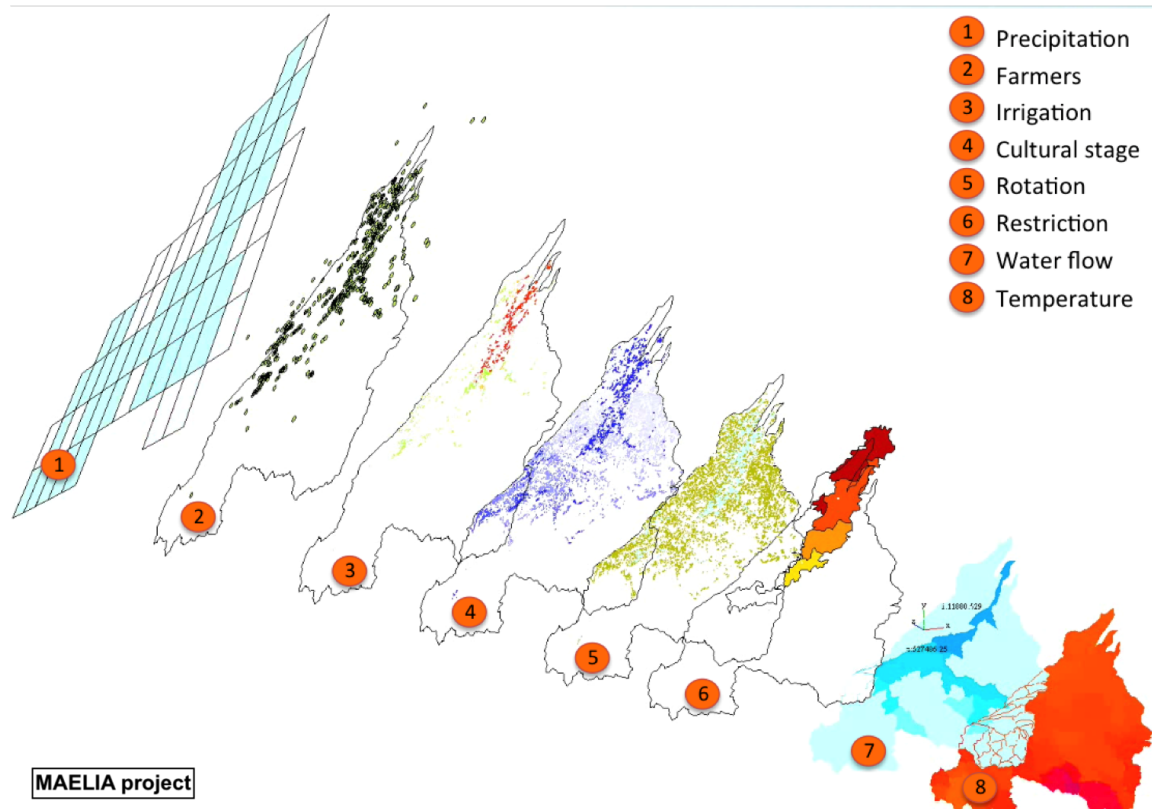


Figure 35.5: MAELIA visualisation

Chapter 36

Scientific 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>.](https://doi.org/10.1007/s10707-018-00339-6)

Table of Contents

- [Papers about GAMA](#)
- [HDR theses](#)
- [PhD theses](#)
- [PhD theses that use GAMA as modeling/simulation support](#)
- [Master theses that use GAMA as modeling/simulation support](#)
- [Research papers that use GAMA as modeling/simulation support](#)

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.” *Cybergeog: 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 Bourgeois](#), “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-making in agent-based models : an application to land use planning 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.
 - **Edouard Amouroux**, “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.

PhD theses that use GAMA as modeling/simulation support

- **Robin Cura**, “Modéliser des systèmes de peuplement en interdisciplinarité. Co-construction et exploration visuelle d’un modèle de simulation”, Université Paris 1 Panthéon-Sorbonne, defended March 6th, 2020.
- **Alice Micolier**, “Development of a methodology for a consistent and integrated evaluation of the health, energy and environmental performance of residential building design solutions”, Université de Bordeaux, defended December 13th, 2019.
- **Dimitrios Panagiotis Chapizanis**, “Exposomic analysis: emerging methodologies for environmental exposure measurements”, Aristotle University of Thessaloniki, 2019.
- **Julius Bañgate**, “Multi-Agent Modelling of seismic crisis”, Université Grenoble Alpes, defended December 18th, 2019.
- **Johan Arcile**, “Conception, modélisation et vérification formelle d’un système temps-réel d’agents coopératifs Application aux véhicules autonomes

- communicants”, Université Paris-Saclay, defended December 13th, 2019.
- **Mélodie DUBOIS**, “Effets combinés de la pêche et des perturbations naturelles sur la dynamique des écosystèmes coralliens”, Université de recherche Paris Sciences et Lettres, defended Mai 24th, 2019.
 - **Allan Lao**, “Agent-Based Mesoscopic Pedestrian Modeling and Simulation”, University of the Cordilleras, defended in February, 2019.
 - **Jérémy Sobieraj**, *Méthodes et outils pour la conception de Systèmes de Transport Intelligents Coopératifs*, Université Paris-Saclay; Université d’Evry-Val-d’Essonne, defended November 7th, 2018.
 - **Myriam Grillot**, *Modélisation multi-agents et pluri-niveaux de la réorganisation du cycle de l’azote dans des systèmes agro-sylvo-pastoraux en transition : Le cas du bassin arachidier au Sénégal*, IRD, defended March 16th, 2018.
 - **Mahefa Rakotoarisoa**, *Les risques hydrologiques dans les bassins versants sous contrôle anthropique : modélisation de l’aléa, de la vulnérabilité et des conséquences sur les sociétés. : Cas de la région Sud-ouest de Madagascar*, Univeristé d’Angers et Université de Tuléa, defended in December, 2017.
 - **Justin Emery**, *La ville sous électrodes : de la mesure à l’évaluation de la pollution atmosphérique automobile. : vers une simulation multi-agents du trafic routier en milieu urbain*, Université de Bourgogne-Franche-Comté, defended December 16th, 2016.
 - **Hugo Thierry**, “Élaboration d’un modèle spatialisé pour favoriser le contrôle biologique de ravageurs de cultures par gestion du paysage agricole”, INPT, defended November 11th, 2015.
 - **Inès Hassoumi**, *Approche multi-agents de couplage de modèles pour la modélisation des systèmes complexes spatiaux. Application à l’aménagement urbain de la ville de Métouia*, Université Pierre et Marie Curie – Paris VI et université de Tunis, defended December, 2014.

Master theses that use GAMA as modeling/simulation support

2020

- Claerhoudt, X. “How opinion leaders can manipulate opinion dynamics in hierarchical social networks”, Master’s thesis, Ghent University, 2020.
- Doelman, V. J. “An agent-based approach to the assessment of carrying capacity

- in Amsterdam”, Master’s thesis, GIMA program, Utrecht University, 2020.
- de Maat, N.B., 2020. “Improving traffic system performance by combining tolling and intention-based prediction: an agent-based model”, Master’s thesis, GIMA program, Utrecht University, 2020.

Research papers that use GAMA as modeling/simulation support

2021

- Bhowmick, D., Winter, S., Stevenson, M. and Vortisch, P., 2021. Exploring the viability of walk-sharing in outdoor urban spaces. *Computers, Environment and Urban Systems*, 88, p.101635.
- Polanco, L.D. and Siller, M., 2021. Crowd management COVID-19. *Annual Reviews in Control*.
- Taillandier, P., Salliou, N. and Thomopoulos, R., 2021. Introducing the Argumentation Framework Within Agent-Based Models to Better Simulate Agents’ Cognition in Opinion Dynamics: Application to Vegetarian Diet Diffusion. *Journal of Artificial Societies and Social Simulation*, 24(2).
- Kaziyeva, D., Loidl, M. and Wallentin, G., 2021. Simulating Spatio-Temporal Patterns of Bicycle Flows with an Agent-Based Model. *ISPRS International Journal of Geo-Information*, 10(2), p.88.
- Ramos Corchado, F.F., López Fraga, A.C., Salazar Salazar, R., Ramos Corchado, M.A. and Begovich Mendoza, O., 2021. Cognitive Pervasive Service Composition Applied to Predatory Crime Deterrence. *Applied Sciences*, 11(4), p.1803.
- Salze, P., Sajous, P. and Bertelle, C., 2021. EM3: A Model to Explore the Effects of Ecomobility Policies on an Urban Area. In *Complex Systems, Smart Territories and Mobility* (pp. 233-256). Springer, Cham.
- Zaatour, W., Marilleau, N., Giraudoux, P., Martiny, N., Amara, A.B.H. and Miled, S.B., 2021. An agent-based model of a cutaneous leishmaniasis reservoir host, *Meriones shawi*. *Ecological Modelling*, 443, p.109455.
- Catarino, R., Therond, O., Berthomier, J., Miara, M., Mérot, E., Misslin, R., Vanhove, P., Villerd, J. and Angevin, F., 2021. Fostering local crop-livestock integration via legume exchanges using an innovative integrated assessment and modelling approach based on the MAELIA platform. *Agricultural Systems*, 189, p.103066.

- Chapizanis, D., Karakitsios, S., Gotti, A. and Sarigiannis, D.A., 2021. Assessing personal exposure using Agent Based Modelling informed by sensors technology. *Environmental Research*, Volume 192, p.110141.
- Hutzler G., Klaudel H., Sali A. (2021) Filtering Distributed Information to Build a Plausible Scene for Autonomous and Connected Vehicles. In: Dong Y., Herrera-Viedma E., Matsui K., Omatsu S., González Briones A., Rodríguez González S. (eds) *Distributed Computing and Artificial Intelligence*, 17th International Conference. DCAI 2020. *Advances in Intelligent Systems and Computing*, vol 1237. Springer, Cham.

2020

- Callejas, E., Inostrosa-Psijas, A., Moreno, F., Oyarzún, M. and Carvajal-Schiaffino, R., 2020, November. COVID-19 Transmission During a Tsunami Evacuation in a Lockdown City. In *2020 39th International Conference of the Chilean Computer Science Society (SCCC)* (pp. 1-8). IEEE.
- Barthelemy, J., Amirghasemi, M., Arshad, B., Fay, C., Forehead, H., Hutchison, N., Iqbal, U., Li, Y., Qian, Y. and Perez, P., 2020. Problem-Driven and Technology-Enabled Solutions for Safer Communities: The case of stormwater management in the Illawarra-Shoalhaven region (NSW, Australia). *Handbook of Smart Cities*, pp.1-28.
- Zhong, J. and Hattori, H., 2020. Generation of Traffic Flows in Multi-Agent Traffic Simulation with Agent Behavior Model based on Deep Reinforcement Learning. *arXiv preprint arXiv:2101.03230*.
- Dang-Huu, T., Gaudou, B., Nguyen-Ngoc, D. and Lê, N.C., 2020, November. An agent-based model for mixed traffic in Vietnam based on virtual local lanes. In *2020 12th International Conference on Knowledge and Systems Engineering (KSE)* (pp. 147-152). IEEE.
- Grignard, A., Nguyen-Huu, T., Gaudou, B., Nguyen-Ngoc, D., Brugière, A., Dang-Huu, T., Nghi, H.Q., Khanh, N.T. and Larson, K., 2020, November. CityScope Hanoi: interactive simulation for water management in the Bac Hung Hai irrigation system. In *2020 12th International Conference on Knowledge and Systems Engineering (KSE)* (pp. 153-158). IEEE.
- da Silva Rodrigues, L., Oliveira, S.G.M., Lopez, L.F. and Sichman, J.S., 2019, May. Agent Based Simulation of the Dengue Virus Propagation. In *International Workshop on Multi-Agent Systems and Agent-Based Simulation* (pp. 100-111). Springer, Cham.

- Prudhomme, C., Cruz, C. and Cherifi, H., 2020, An Agent based model for the transmission and control of the COVID-19 in Dijon. In Proc. of The 11th Conference on Network Modeling and Analysis MARAMI, October 14 - 15, 2020
- Taj, F., Klein, M. and van Halteren, A., 2020, October. An Agent-Based Framework for Persuasive Health Behavior Change Intervention. In International Conference on Health Information Science (pp. 157-168). Springer, Cham.
- Gaudou, B., Huynh, N.Q., Philippon, D., Brugière, A., Chapuis, K., Taillandier, P., Larmande, P. and Drogoul, A., 2020. COMOKIT: a modeling kit to understand, analyze and compare the impacts of mitigation policies against the COVID-19 epidemic at the scale of a city. *Frontiers in Public Health*, 8, p.587.
- Drogoul, A., Taillandier, P., Gaudou, B., Choisy, M., Chapuis, K., Huynh, Q.N., Nguyen, N.D., Philippon, D., Brugière, A. and Larmande, P., Designing social simulation to (seriously) support decision-making: COMOKIT, an agent-based modelling toolkit to analyse and compare the impacts of public health interventions against COVID-19. *Review of Artificial Societies and Social Simulation*, 27th April 2020.
- Bucchiarone, A., De Sanctis, M. and Bencomo, N., 2020. Agent-Based Framework for Self-Organization of Collective and Autonomous Shuttle Fleets. *IEEE Transactions on Intelligent Transportation Systems*.
- Iskandar, R., Allaw, K., Dugdale, J., Beck, E., Adjizian-Gérard, J., Cornou, C., Harb, J., Lacroix, P., Badaro-Saliba, N., Cartier, S. and Zaarour, R., 2020, Agent-Based simulation of pedestrians' earthquake evacuation; application to Beirut, Lebanon. In 17th World Conference on Earthquake Engineering, 17WCEE, Sendai, Japan
- Lee, L.W.F. and Mohd, M.H., 2020, October. Stochastic modelling of the biodiversity effect on sin nombre virus (SNV) prevalence. In AIP Conference Proceedings (Vol. 2266, No. 1, p. 050017). AIP Publishing LLC.
- Ngom, B., Diallo, M. and Marilleau, N., 2020, September. MEDART-MAS: MEta-model of Data Assimilation on Real-Time Multi-Agent Simulation. In 2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT) (pp. 1-7). IEEE.
- Alonso Vicario, S., Mazzoleni, M., Bhamidipati, S., Gharesifard, M., Ridolfi, E., Pandolfo, C., Alfonso, L., 2020. Unravelling the influence of human behaviour on reducing casualties during flood evacuation. *Hydrological Sciences Journal*.
- Tannier C., Cura R., Leturcq S. and Zadora-Rio E., 2020. An agent-based model for exploring the combined effects of social and demographic changes on the concentration and hierarchy of rural settlement patterns in North-Western

- Europe during the Middle Ages (800–1200 CE). *Journal of Anthropological Archaeology*, vol. 59.
- Wallentin, G., Kaziyeva, D., Reibersdorfer-Adelsberger, E., 2020. COVID-19 Intervention Scenarios for a Long-term Disease Management, *International Journal of Health Policy and Management*, (), pp. -.
 - Duc, P.M., Chapuis, K., Drogoul, A., Gaudou, B., Grignard, A., Marilleau, N. and Nguyen-Huu, T., 2020. HoanKiemAir: simulating impacts of urban management practices on traffic and air pollution using a tangible agent-based model. In *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)* (pp. 1-7). IEEE.
 - Laatabi, A., Becu, N., Marilleau, N., Pignon-Mussaud, C., Amalric, M., Bertin, X., Anselme, B. and Beck, E., 2020. Mapping and Describing Geospatial Data to Generalize Complex Models: The Case of LittoSIM-GEN. *International Journal of Geospatial and Environmental Research*, 7(1), p.6.
 - Farias, G., Leitzke, B., Born, M., Aguiar, M. and Adamatti, D., 2020. Water Resources Analysis: An Approach based on Agent-Based Modeling. *Revista de Informática Teórica e Aplicada*, 27(2), pp.81-95.
 - Baeza, J.L., Noennig, J.R., Weber, V., Grignard, A., Noyman, A., Larson, K., Saxe, S. and Baldauf, U., 2020. Mobility Solutions for Cruise Passenger Transfer: An Exploration of Scenarios Using Agent-Based Simulation Models. In *Towards User-Centric Transport in Europe 2* (pp. 89-101). Springer, Cham.
 - Jindal, A. and Rao, S., 2020. Lockdowns to Contain COVID-19 Increase Risk and Severity of Mosquito-Borne Disease Outbreaks. *medRxiv*.
 - Haddad, H., Bouyahia, Z. and Jabeur, N., 2020. Socially-Structured Vanpooling: A Case Study in Salalah, Oman. *IEEE Intelligent Transportation Systems Magazine*.
 - Mariano, D.J.K. and Alves, C.D.M.A., 2020. The application of role-playing games and agent-based modelling to the collaborative water management in peri-urban communities. *RBRH*, 25.
 - Emery, J., Marilleau, N., Martiny, N., & Thévenin, T., 2020. Le modèle SCAUP: Simulation multi-agents à partir de données de CAPteurs Urbains pour la Pollution atmosphérique automobile. *Cybergeog: European Journal of Geography*.
 - Taj, F., Klein11, M. C., van Halteren, A., 2020. Towards a generic framework for a health behaviour change support agent. In *ICAART* (1) (pp. 311-318)
 - Thierry, H. and Rogers, H., 2020. Where to rewild? A conceptual framework to spatially optimize ecological function. *Proceedings of the Royal Society B*, 287(1922), p.20193017.

- Baeza, J.L., Noennig, J.R., Weber, V., Grignard, A., Noyman, A., Larson, K., Saxe, S. and Baldauf, U., 2020. Mobility Solutions for Cruise Passenger Transfer: An Exploration of Scenarios Using Agent-Based Simulation Models. In *Towards User-Centric Transport in Europe 2* (pp. 89-101). Springer, Cham.
- Galimberti, A., Alyokhin, A., Qu, H. and Jason, R.O.S.E., 2020. Simulation modelling of potato virus Y spread in relation to initial inoculum and vector activity. *Journal of Integrative Agriculture*, 19(2), pp.376-388.
- Daudé, É. and Tranouez, P., 2020. ESCAPE-SG: un simulateur d'évacuation massive de population pour la formation des acteurs à la gestion de crise. Netcom. Réseaux, communication et territoires.

2019

- Démare, T., Bertelle, C., Dutot, A. and Fournier, D., 2019. Adaptive behavior modeling in logistic systems with agents and dynamic graphs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(3), pp.1-25.
- Olszewski, R., Pałka, P., Turek, A., Kietlińska, B., Płatkowski, T. and Borkowski, M., 2019. Spatiotemporal Modeling of the Smart City Residents' Activity with Multi-Agent Systems. *Applied Sciences*, 9(10), p.2059.
- Larsen, J.B., 2019. Going beyond BDI for agent-based simulation. *Journal of Information and Telecommunication*, 3(4), pp.446-464.
- Larsen J.B., 2019. Adding Organizational Reasoning to Agent-Based Simulations in GAMA. In: Weyns D., Mascardi V., Ricci A. (eds) *Engineering Multi-Agent Systems. EMAS 2018. LNCS*, vol 11375. Springer, Cham.
- Cura R., 2019. « Model Visualization ». In Pumain D. (dir), *Geographical Modeling: Cities and Territories*, John Wiley & Sons, Ltd, pp. 151-91.
- Humann, J. and Pollard, K.A., 2019. Human Factors in the Scalability of Multirobot Operation: A Review and Simulation. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)* (pp. 700-707). IEEE.
- Lammoglia, A., Leturcq, S., Delay, E., 2019. The VitiTerroir model to simulate the spatial dynamics of vineyards on the long term (1836-2014). Example of application in the department of Indre-et-Loire. *Cybergeog: European Journal of Geography*. 2019 Dec 8.
- Taillandier, P., Salliou, N., Thomopoulos, R., 2019. Coupling agent-based models and argumentation framework to simulate opinion dynamics: application to vegetarian diet diffusion. *Social Simulation Conference 2019*, Sep 2019, Mainz, Germany. fhal-02265765f

- 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

- Alonso, L., Zhang, Y.R., Grignard, A., Noyman, A., Sakai, Y., ElKatsha, M., Doorley, R. and Larson, K., 2018, July. Cityscope: a data-driven interactive simulation tool for urban design. Use case Volpe. In International conference on complex systems (pp. 253-261). Springer, Cham.
- Sobieraj, J., Nouveliere, L., Hutzler, G. and Klaudel, H., 2018, October. Modélisation du changement de voie de véhicules autonomes à différents niveaux d'abstraction. Journées Francophones sur les Systèmes Multi-Agents 2018 (JFSMA'2018), Oct 2018, Métabief, France. pp.21–30.
- Tsagkis, P. and Photis, Y.N., 2018. Using Gama platform and Urban Atlas Data to predict urban growth. The case of Athens. 13th International Conference of the Hellenic Geographical Society.
- 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>
- Adam, C., Taillandier, F., 2018. Games ready to use: A serious game for teaching natural risk management. *Simulation and Gaming*, SAGE Publications, 2018.
- 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 Multimodal 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
 - Bourgeois, M., Taillandier, P., Vercouter, L., Adam, C., 2018. Emotion Modeling in Social Simulation: A Survey. *J. Artif. Soc. Soc. Simul.* 21, 5.

- Grillot M., Vayssières J., Masse D., 2018. Agent-based modelling as a time machine to assess nutrient cycling reorganization during past agrarian transitions in West Africa. *Agricultural Systems* 164, 133-151. <https://doi.org/10.1016/j.agsy.2018.04.008>
- 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

- Bañgate, J., Dugdale, J., Beck, E., & Adam, C. (2017, December). SOLACE a multi-agent model of human behaviour driven by social attachment during seismic crisis. In *2017 4th International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)* (pp. 1-9). IEEE.

- Arcile, J., Sobieraj, J., Klaudel, H., & Hutzler, G. (2017). Combination of simulation and model-checking for the analysis of autonomous vehicles' behaviors: A case study. In *Multi-Agent Systems and Agreement Technologies* (pp. 292-304). Springer, Cham.
- 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 TRAFIC 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., Vercoüter, 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](https://doi.org/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](https://doi.org/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](https://doi.org/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) 3D GIS-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., Théron, 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 informa-

tion 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.

Chapter 37

Training Session

SEARCA Phillippines 2021 (Online)

22-26 March, 2021

Trainers: Alexis Drogoul, Arthur Brugière, Patrick Taillandier, Nguyen Ngoc Doanh

AWP 2021(Online)

1-5 March, 2021

Trainers: Arthur Brugière, Alexis Drogoul, Huynh Quang Nghi, Nguyen Ngoc Doanh, Patrick Taillandier, Truong Chi Quang

USTH Training session 2020

May 2020 - Hanoi, Vietnam

Trainers: Benoit Gaudou

SMAC Toulouse 2020

April 2020 - Toulouse, France

Trainers: Benoit Gaudou, Arthur Brugiere, Damien, Nicolas Verestaevel

Application to disaster management and evacuation

17-21 February - Cebu, Phillipines

Trainers: Alexis Drogoul, Benoit Gaudou, Arthur Brugiere

Training session TLU 2019

October 2019 - Hanoi, Vietnam

Trainers: Benoit Gaudou, Nguyen Ngoc Doanh, Arthur Brugiere, Doryan Kaced

AWP Phnom Penh 2019

8-12 July, Phnom Penh, Cambodia

This is a training session that focuses on water urban risks: designing evacuation strategies in case of flooding with Agent-Based Modeling and GAMA

Formation Toulouse 2019

May 2019 - Toulouse, France

Trainers: Patrick Taillandier, Doryan Kaced, Renauld Misslin, Frédéric Amblard, Benoit Gaudou

Training session Brasilia 2019

Feb 201 - Brasilia, Brazil

Trainers: Benoit Gaudou, Ch. Le Page

Application to disaster management and evacuation

28 Jan - 1 Feb, Philippines

Trainers: Alexis Drogoul, Benoit Gaudou, Kevin Chapuis

AWP Can Tho 2018

July - Can Tho, Vietnam

Trainers: Alexis Drogoul, Patrick Taillandier, Benoit Gaudou, Truong Chi Quang, Damien Philippon, Kevin Chapuis, Huynh Quang Nghi

SCEMSITE 2018

May 2018

Trainers: Patrick Taillandier, Oliver Therond

Formation Toulouse 2018

Trainers: Patrick Taillandier, Benoit Gaudou

GAMA 1.7RC1 training session - Pays-Bas

December 2017

Analysis of land use dynamics (JTD 2017)

9-14 July 2017

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 11th JTD, the topic was about the contributions of modeling for the analysis of land use dynamics. Case study: Thanh Phu district, Ben Tre province (Mekong delta) 2000-2010.

Trainers: Alexis Drogoul, Patrick Taillandier, Quang Nghi Huynh, Quang Chi Truong, Damien Philippon

Master TRIAD 2017

Jan 2017

EDSS USTH Master 2016

November 2016 - Hanoi, Vietnam

Trainers: Patrick Taillandier, Nicolas Marilleau, Benoit Gaudou

Design urban energy transition policies (JTD 2016)

10-15 July 2016

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 10th JTD, the topic was about the use of computer models to help design urban energy transition policies.

Trainers: Javier Gil-Quijano, Alexis Drogoul, Benoît Gaudou, Patrick Taillandier, Julien Mazars, Hypatia Nassopoulos, Damien Philippon

Modeling for supporting decisions in urban management issues

7-11 December 2015 - Siem Reap (Cambodia)



Figure 37.1: resources/other/trainingSession/SiemReap2015/photos/group.JPG

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)



Trainers: Drogoul Alexis, Gaudou Benoit, Trung Quang, Philippon Damien, Mazars Julien.

Epidemiological risks and the integration of regional health policies (JTD 2015)

19-24 July 2015

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 9th JTD, the topic was about epidemiological risks and the integration of regional health policies with the application of modeling by GAMA platform to decision support.

Trainers: Alexis Drogoul, Vo Duc An, Benoit Gaudou, Damien Philippon, Chi-Quang Truong

MAPS 8 2015

8-25 June 2015

Nex Days 2015 (GAMA 1.6.1)

May 2015

MISS ABMS 2014

September 2014

MAPS epidemic city tutorial 2014

July 2014

GAMA training session Phillippines

12-16 Jan 2015 - Quezon, Phillippines

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>.

AUF 2013

November 2013

MISS ABM 2013

October 2013

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>.

Can Tho training session 2012

November 2012 - Can Tho, Vietnam

ESSA Tutorial 2012

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>.

Introduction of GAMA 1.4

5 December 2011 - Bondy, France

This training session was held at PDI Doctoral School, Bondy. The main topic of this session is the introduction of GAMA 1.4 with its advances:

- Deep refactoring work of the source code
- New programming language GAML (not based on XML)
- Integration of a true IDE based on Eclipse

- Deep refactoring of the meta-model
- Better integration of multi-levels
- New important notion: topology
- New variable types: geometry, graph, path, topology
- Many more novelties/improvements/enrichments. . .

Trainers: Alexis Drogoul, Patrick Taillandier, Benoit Gaudou, Vo Duc An, Jean-Daniel Zucker, Edouard Amouroux

Formation à IRD Bondy

October 2010 - Bondy (France)

This training session was held at IRD Bondy. The topic of this session was about the future of GAMA and a glance at agent-based models were built based on the GAMA platform.

Trainers: Alexis Drogould, Patrick Taillandier, Edouard Amouroux

Introduction to the GAMA and PAMS platforms (IFI 2009)

26-28 October 2009 - Hanoi (Vietnam)

This training session was held at IFI Hanoi. The topic of this session was about the introduction of GAMA and PAMS platforms, and how to use tools such as GAMA to build models with an agent-based approach.

Trainers: Alexis Drogoul, Vo Duc An, Patrick Taillandier, Benoit Gaudou, Chu Thanh Quang, Jean-Daniel Zucker, François Sempé, Guillaume Chérel, Nicolas Marilleau

Chapter 38

Events

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](#)