

# GAMA guidebook

20 oct 09 version

# Contents

## Articles

GAMA	1
First steps	5
Interface guide	8
Stupid Tutorial	10
StupidModel1	11
StupidModel10	14
StupidModel11	17
StupidModel12	20
StupidModel13	24
StupidModel14	28
StupidModel15	32
Modeling guide	33
Behaviors	34
Commands	36
Operators	43
Keywords	63
Skills	65
Built-in	72
Species	76
Sections	79

## References

Article Sources and Contributors	85
Image Sources, Licenses and Contributors	86

# GAMA

---

## Description

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 developed by the research team MSI (located in the IFI, Hanoi, and part of the IRD/UPMC International Research Unit UMMISCO) since 2007.

The most important requirements of spatially explicit multi-agent simulations that GAMA fulfils are:

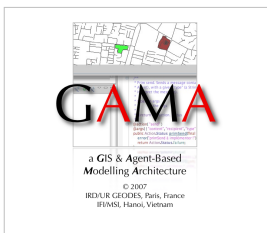
1. The ability to use complex GIS data as an environment for the agents;
2. The ability to handle a vast number of (possibly heterogeneous) agents;
3. The ability to offer a platform for automated controlled experiments (by automatically varying parameters, recording statistics, etc.);
4. The possibility to let non-computer scientists design models and interact with the agents during simulations.

Beyond these features, GAMA also offers:

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

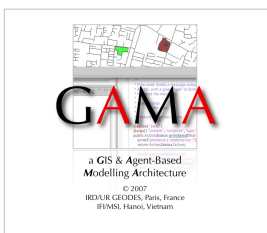
## News

### *October 17, 2009*



A new version of GAMA 1.1 has been released, and is likely the one we will use for the → training session (minus some last minute bugfixes and the incorporation of the new models). It incorporates several changes, including an XML editor coupled with the simulator, the possibility to take snapshots of every graphical window, save parameters and monitors for future reuse, save charts as CSV files, and definitely fixes the memory leaks observed in previous versions. Please see the Downloads section.

### *September 23, 2009*



A new version of GAMA 1.1 has been released. It improves the use and display of parameters, the display of agents, and corrects five important bugs. Please see the Downloads section.

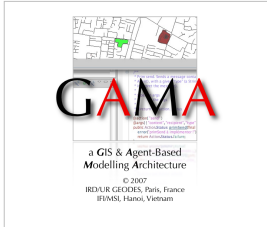
### *September 8, 2009*

A training session on agent-based modeling and a tutorial on GAMA (version 1) will be organized at the IFI from the 26th to the 28th of October, 2009. More information will be available on → this page.

### *June 26, 2009*

We are still working on the source code for Version 2. Right now, we are updating both the metamodel and the parser (with a new, and hopefully easier to input, syntax for the language, as well as a dedicated editor). While it is accessible from the CVS, please note that it is still not functional. There have been some delays in the development of this version, but we hope we'll be able to release it sometime during the summer.

### *June 25, 2009*



Two bugs have been squashed from version 1.1 alpha (the numbering still remains identical). One related to using the movement primitives in a grid environment (random moves were not exactly random), the other being a possible "Index out of bounds" error in the operator among. Please download the software again if you are using these features (and likely having troubles). The debugged source code is available, as well, from the CVS server.

### *March 3, 2009*

The source code for version 2 has been updated, but please note that **the current available version is not functional**. The latest round of changes includes a deep refactoring of the metamodel, in which the notion of "Place" has disappeared, in order to pave the way to the transition to Repast Symphony.

### *February 21, 2009*

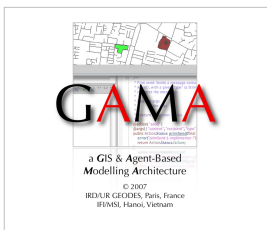
The highly experimental source code for version 2 is available for browsing on the CVS server. It includes some important internal changes that will be documented later. The best way to test it is to check out the branch tagged "Version2" in a new Eclipse project. Be aware that some models (especially those using GIS environments, diffusions and/or file inputs) are not compatible with this version (those included in the distribution will be fixed later, and the documentation will be changed as well when version 2.0 will be released)

### *February 18, 2009*



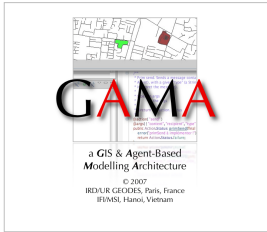
Establishment of an European mirror at the IRD (see Downloads).

### *February 4, 2009*



Some bug fixes have been incorporated in version 1.1 alpha (the numbering remains identical), especially for models composed of multiple species of agents where some species inherit from others. Please download again the software if you have troubles with GAMA. The debugged source code is available, as well, from the CVS server.

*January 16, 2009*



The first non-internal version (1.1 alpha) of GAMA has been released for public use. This version incorporates mainly bug fixes, thanks to the feedback of the Master students of the IFI (Hanoi) and the University of Can Tho. This is however still an alpha version, since several of its features have neither been fully tested nor documented.

## Downloads

GAMA runs on almost any current computer, the only constraint being the amount of memory available (some models may need up to 2GB of free memory to run)

- Windows: Windows Vista, XP, 2000, NT, ME, and 98.
- Mac OS X: Mac OS X 10.4 (or newer) is recommended, but 10.3 and 10.2 are also supported. Please run Software Update to ensure that you have the latest Java Runtime Environment (1.5 required).
- Other platforms: GAMA should work on any platform on which a Sun Java Virtual Machine, version 1.5 or later, is installed.

The current usable (and mainly stable) version of GAMA is version 1.1 alpha. Please download the distribution that corresponds to your operating system:

- GAMA for Windows (Vietnam site).
- GAMA for Ubuntu (Vietnam site).
- GAMA for MacOS X (Vietnam site).

**Note:** GAMA is written in Java. If you don't have the Java Runtime Environment (JRE) version 5 or later installed on your machine, please click here <sup>[1]</sup> for the latest version available. MacOS X users shouldn't need to install anything, as this operating system comes preloaded with a custom Apple JRE.

**Note to users wishing to download GAMA from outside Vietnam:** The limited speed and bandwidth of outgoing Vietnamese Internet lines may prevent you from correctly completing the download. We have thus established an European mirror in France, from which you have access to the very same version:

- GAMA for MacOS X (European mirror) <sup>[2]</sup>
- GAMA for Ubuntu (European mirror). <sup>[3]</sup>
- GAMA for Windows (European mirror). <sup>[4]</sup>

## Source code

*January 16, 2009.* As of version 1.1 alpha, the source code of GAMA is now available for downloading. You can access it from our CVS server using the following read-only account:

**Host:** 210.245.52.197

**Repository Path:** /home/cvs

**User name:** readonly

**Password:** prima

Among the modules available in the repository path, the GAMA source code is located in "GAMA\_RCP/GAMA2". Please note that GAMA comes as an Eclipse <sup>[5]</sup> project, and that it heavily relies on Eclipse features for compiling and running. If you do not have Eclipse installed, please download <sup>[6]</sup> it ("Classic" distribution recommended), and then simply create a new project using the wizard called "Projects from CVS", entering the information above when the wizard asks for the repository.

## Copyright information

This is a free software (distributed under the LGPL), so you can have access to the code, edit it and redistribute it under the same terms.

## Documentation

A new version of the documentation is being designed. It should be finished in time for the release 2 (around April, 2009). Until then, the structure and the contents of the pages should undergo several modifications.

- → First steps with GAMA **Status:** *on hold*. Usable, but should undergo some changes based on the lectures given at the University of Can Tho.
- → Interface guide **Status:** *on hold*. Same thing. The lectures should be used instead in the mean time.
- → Tutorial **Status:** *active*. We are replacing the current grid-based tutorial with a tutorial based on StupidModel<sup>[7]</sup>, and designing a simpler GIS-based tutorial.
- → Modeling guide **Status:** *active*. Many sections are now almost finished.
- A pdf-file<sup>[8]</sup> compiling all the documentation is also available for offline reference (*last updated in june 09*).

## Models repository

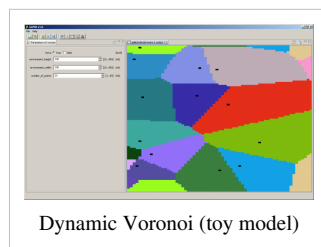
### Under construction

Besides example models included in the release, this section describes some models developed by Phd students working in the GAMA project and Master students during their internships.

- Emergency : Simulation of rescue activities in urban area after an earthquake. Data of the simulated urban area are in GIS format. Several versions of this models have been developed to test the realism and efficiency of rescue activities.
- GAMAVI :
- EDF :
- SORTIE : spatial model of forest growth.
- Synchronisation des cultures et cicadelles brunes (Trang) :
- Virus des élevages de poisson-chat (Le Thi Diem - Can Tho) :
- Virus des élevages de crevette (Truong Thi Thanh Tuyen - Can Tho) :
- propagation des cicadelles brunes à grande échelle (Phan Huy Cuong - Can Tho) :

## Screenshots

### Snapshots



Dynamic Voronoi (toy model)

## Contact us

Feedback from users is very valuable to us in designing and improving GAMA. You can contact us at [9].

- **Reporting Bugs:** If you would like to report a bug that you find in GAMA, send a mail (we will establish a bug reporting tool later) to the address above, and please try to include as much of the following 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.
  - Your system information: GAMA version, OS version, Java version, and so on.
  - Any error messages that were displayed.

## References

- [1] <http://java.sun.com/javase/downloads/index.jsp>
- [2] [http://www.ird.fr/ur079/gama/GAMA-version\\_1.1\\_Alpha-MacOSX.zip](http://www.ird.fr/ur079/gama/GAMA-version_1.1_Alpha-MacOSX.zip)
- [3] [http://www.ird.fr/ur079/gama/GAMA-version\\_1.1\\_Alpha-Ubuntu.zip](http://www.ird.fr/ur079/gama/GAMA-version_1.1_Alpha-Ubuntu.zip)
- [4] [http://www.ird.fr/ur079/gama/GAMA-version\\_1.1\\_Alpha-Windows.zip](http://www.ird.fr/ur079/gama/GAMA-version_1.1_Alpha-Windows.zip)
- [5] <http://www.eclipse.org>
- [6] <http://www.eclipse.org/downloads/>
- [7] <http://condor.depaul.edu/~slytinen/abm/StupidModel/>
- [8] [http://www1.ifi.auf.org/mediawiki/images/f/fb/Gama\\_book\\_pediapress\\_juin\\_09.pdf](http://www1.ifi.auf.org/mediawiki/images/f/fb/Gama_book_pediapress_juin_09.pdf)
- [9] [mailto:gama\\_dev@ifi.edu.vn](mailto:gama_dev@ifi.edu.vn)

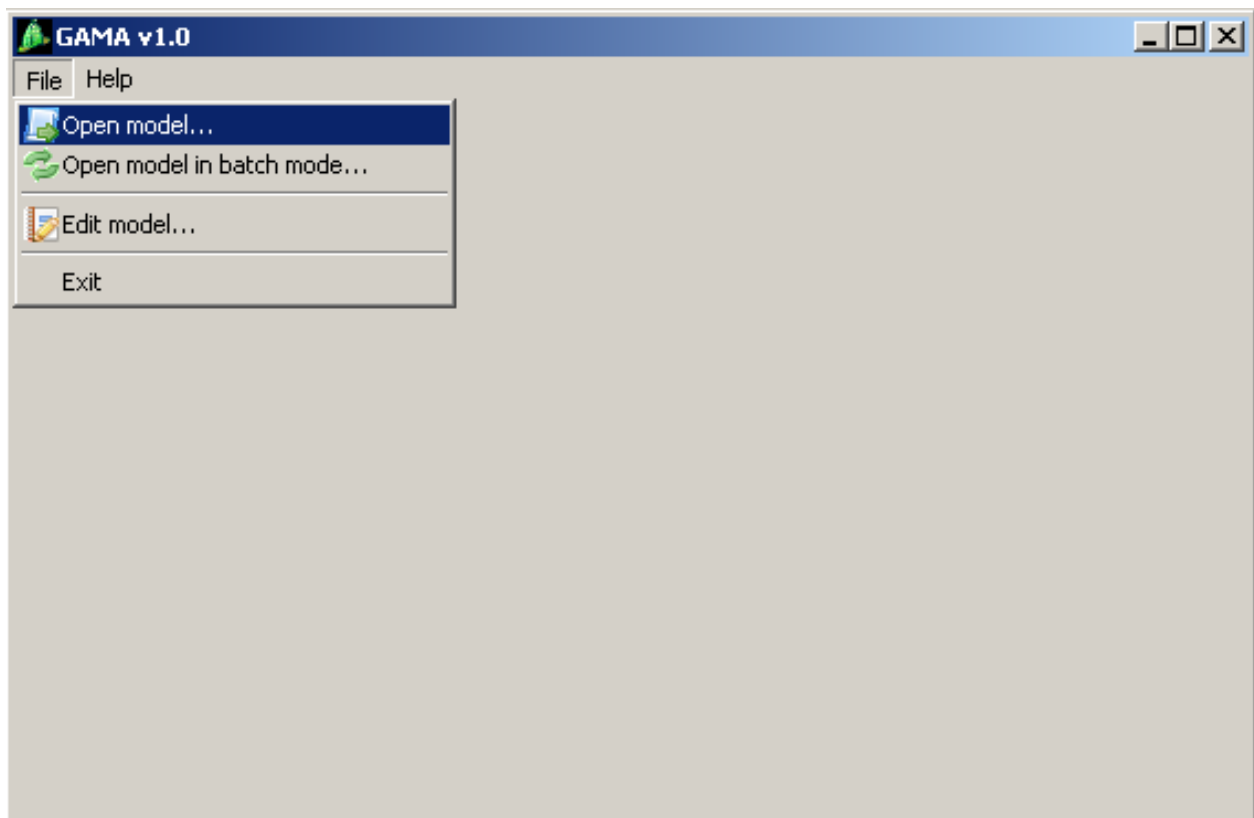
## First steps

---

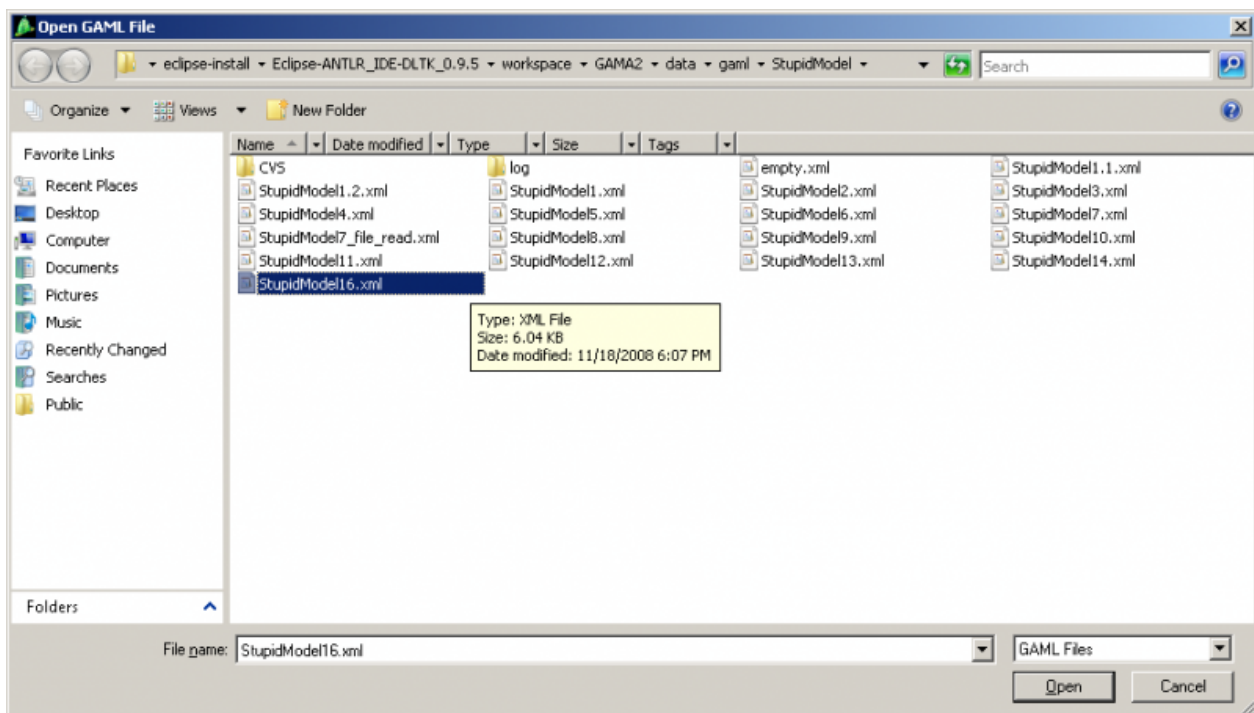
This section will walk you through a simple model named "StupidModel16". The purpose of this section is to show you how to open and "play" with a simple GAMA model.

First of all, please run the GAMA platform.

- This is the graphical user interface of GAMA after starting up:
- Currently, the models of GAMA are written in XML. You can open a model using one of the two following ways:
  - Click the button "Open model..."
  - Or click the menu "File/Open model"

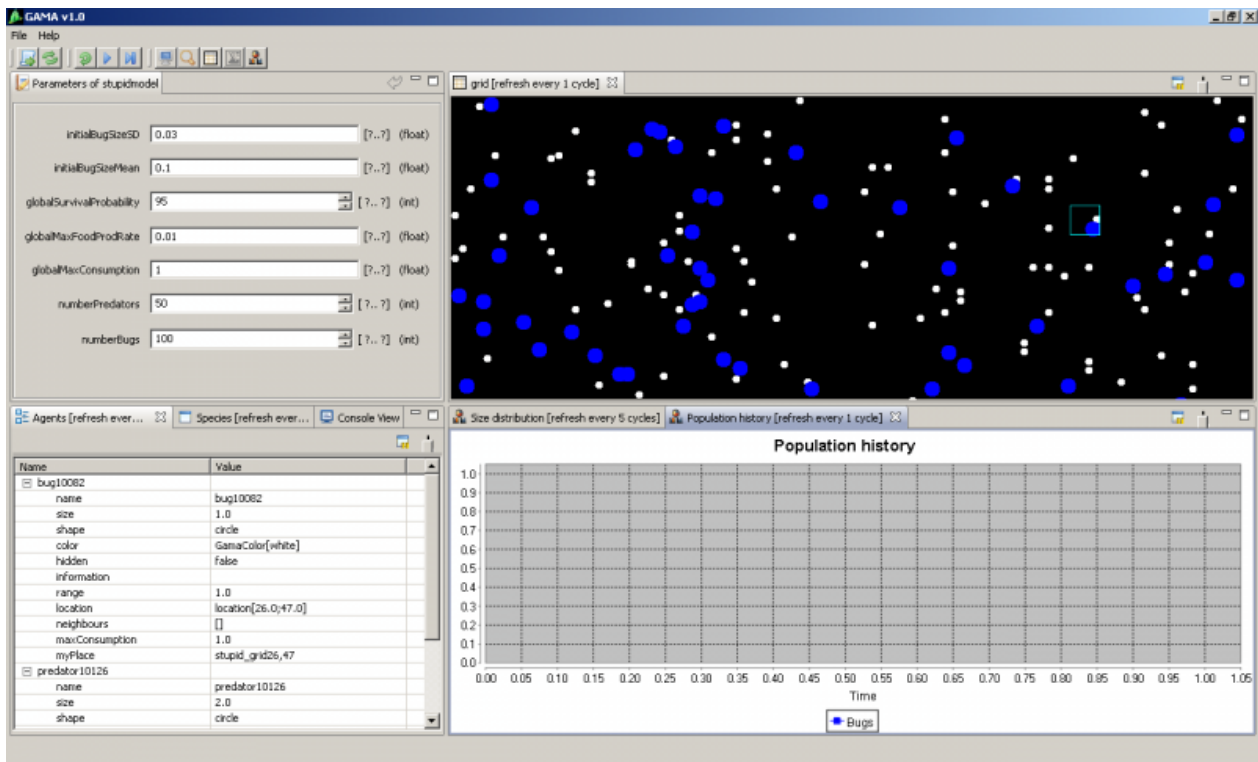


- Select the “StupidModel16.xml” file in "data/gaml/StupidModel:

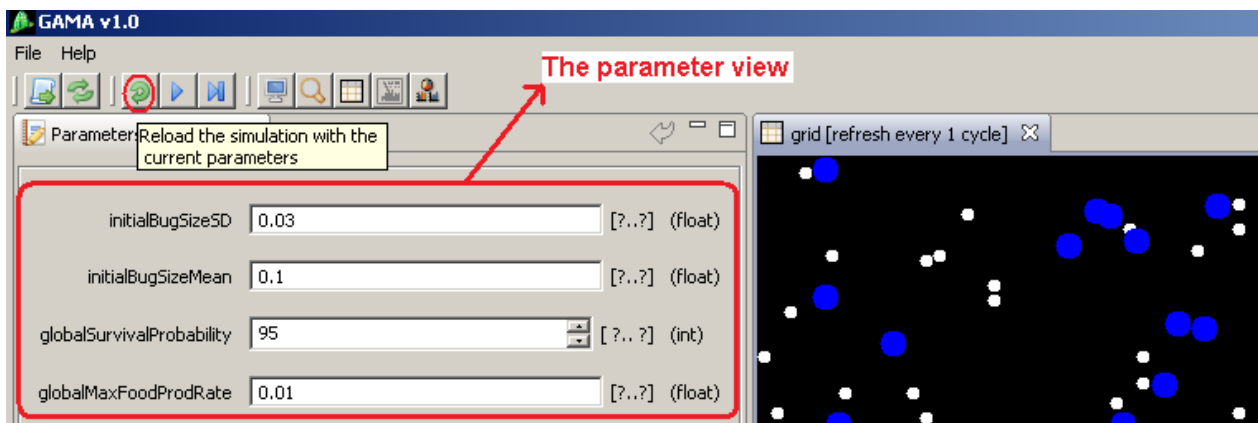


- You receive:

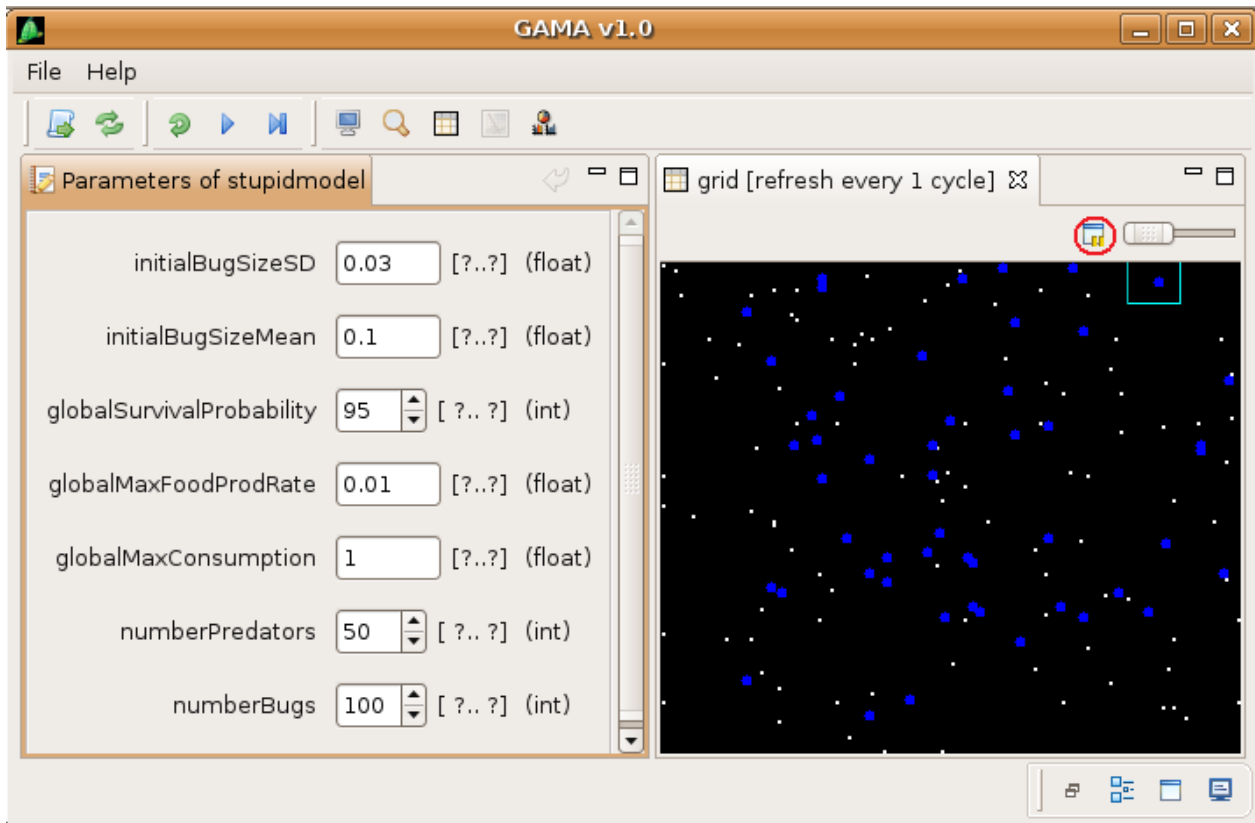




- To run/pause the simulation, you click the button “Run/Pause”:
- To reload the model, you click the button “Reload the simulation with current parameters”:



- If you want to run the simulation step by step, you click the button “Step”:
- You can adjust the "refresh rate" of certain views using their corresponding sliders.
- If you want to stop updating certain views, you can click their corresponding "Pause" button.



## Interface guide

---

### Introduction

As of version 1.1 Alpha, the GAMA 's graphical user interface (GUI) contains two modes : Simulation Mode and Edition Mode.

- **Simulation Mode:** User opens model in this mode if he wishes to run the simulation.
- **Edition Mode:** User opens model in this mode if he wishes to edit the model 's source code.

User can switch back and forth easily between these two modes.

### Create a new model


There are two way to create a new model: 1. Go to File -> Create a new model...

2. Or you can click on the corresponding button on the toolbar to create a new model:

Three template are proposed for a new model:



- Empty: the content of the new model is empty.
  - Skeleton: the new model contains a minimum skeleton.
  - Example: Create a model with a running example.
-

## Open a model

- Open model in Simulation Mode: Go to File -> Run a model or click the  icon on the toolbar.
- Open Model in Edition Mode: Go to File -> Edit a model or click the icon on the toolbar.

## Simulation Mode

### Control the simulation


- Start the simulation : click on the  icon.
- Pause the simulation : click on the icon
- Step the simulation : click on the icon.
- Stop the simulation : click on the  icon.
- Reload the simulation : click on the icon

### Views

GAMA supplies various views serving different needs of modeler.

- Grid view : views the GRID environment.
- GIS view : views a GIS environment.
- Parameter view : views parameters of model. User can edit these parameter then reloads the model with the new values of parameters.
  - Save parameter : saves the current value of parameter to an XML file.
  - Reload model : reloads the model with the current value of parameters.
- Monitor view : views the monitored expressions.
- Chart view : views the charts declared in the model.
- Agents view : views currently selected agent of GRID or GIS view.
- Species view : views all the species of the model.
- Source view : views the source code of the model in a tree based format.

### Control the view

- Capture snapshot on a view: click on the  icon to capture the snapshot of the corresponding view.
- Adjust refresh rate of a view: click on the icon to change the refresh rate of the corresponding view.
- Pause a view: click on the to pause a view.

## Switch to Edition Mode

- Click on the icon to switch to edition mode.

## Edition Mode


This mode helps modeler edit the source code of his model.

---

## Views

- Editors: show content of corresponding files of the models.
- Outline view: shows the tree-based format of the currently selected editor.

## Switch to Simulation Mode

- Click on the  icon to switch to simulation mode.

## Reload the model in the top-most opening editor

- Click on the on the toolbar to reload the model corresponding to the top-most opening editor.

# Stupid Tutorial

---

## Introduction

This tutorial is based on the article: StupidModel and Extensions: A template and teaching tool for agent-based modeling platforms <sup>[1]</sup> by Railsback, Lytinen and Grimm. It is particularly well fitted for this purpose as the complexity increase smoothly and it allows to compare GAMA to other well known platform such as (Mason, Netlogo, Swarm (java & objective-C): see here <sup>[2]</sup> for implementations).

## Preliminary notes

Some explanation about GAMA and the metamodel? @@@TODO@@@

- Any uncertainty that have appeared within the formulation of the previously cited paper have been dealt using the netlogo implementation.

## Stupid model: the basics

@@@TODO@@@ Presentation of the (global / conceptual) model.

## Stupid model: models list

Here the list of the different models that you will build while following the tutorial. This list is the same as in the original paper though some intermediary steps have been added in some cases. For each model we will present its purpose and an explicit formulation (from the original authors) and possibly some disambiguation or specificities due to the GAMA platform.

1. → Basic stupidModel
  2. → Bug growth
  3. → Habitat cells and ressource
  4. → Cell and bug probes
  5. → Parameters and parameters displays
  6. → Histogram output
  7. → Stopping the model
  8. → File output
  9. → Randomized agent actions
  10. → Sorted agent actions
  11. → Optimal movement
  12. → Bug mortality and reproduction
-

13. → Population abundance graph
14. → Random normal initial size
15. → Habitat data from file input
16. → Predators

## References

- [1] <http://condor.depaul.edu/~slytinen/abm/StupidModelFormulation.pdf>  
[2] <http://condor.depaul.edu/~slytinen/abm/>

# StupidModel1

---

## Purpose

This is the basic StupidModel, an extremely simple individual-based model used as a starting point for learning GAMA (or other IBM platforms).

## Formulation

- The space is a two-dimensional grid of dimensions 100 x 100. The space is toroidal, meaning that if bugs move off one edge of the grid they appear on the opposite edge.
- 100 bug agents are created. They have one behavior: moving to a randomly chosen grid location within +/- 4 cells of their current location, in both the X and Y directions. If there already is a bug at the location (including the moving bug itself—bugs are not allowed to stay at their current location unless none of the neighborhood cells are vacant), then another new location is chosen. This action is executed once per time step.
- The bugs are displayed on the space. Bugs are drawn as red circles. The display is updated at the end of each time step.
- Instead of specifying which random number generation algorithm to use, the default generator for each platform is used.

## Models

GAMA is a bit different from the platforms whose the original authors thought of thus we will do several intermediary models Here.

### Model 0 : the minimal set

In order to “run” a minimum model we only have to declare an output. By default the platform will create a “continuous” environment of 100x100 meters and a “grid” of 1x1 cell (named 'grid' and with default white color) but no agent. Here is the minimum text you would have to write:

```
<stupidmodel>
  <modelClass name="msi.gama.kernel.Simulation"/>
  <output>
    <display name="grid" type="grid"/>
  </output>
</stupidmodel>
```

## Model 1.1 : the environment

First we want to define correctly the environment as specified in the previous formulation so we have to add the environment section, which gives us the following result:

```
<stupidmodel>
  <modelClass name="msi.gama.kernel.Simulation"/>
  <environment>
    <grid name="stupid_grid" width="100" height="100"
torus="true"/>
  </environment>
  <output>
    <display name="grid" type="grid" environment="stupid_grid"/>
  </output>
</stupidmodel>
```

The added lign (bold charaters) define a toroidal grid constituted of 100x100 square cells (hexagonal is also available by adding the attribute hexagonal="true" to the grid) with a 4-neighbours structure (by default). We also modify the display lign (italic characters) to take that into account.

## Model 1.2 : Defining the agents

Here we have to define the structure of the bug agents then their behaviour:

- What is a bug agent?
  - It is a situated agent (on the default grid) thanks to the associated skill.
  - It is visible thanks to the associated skill.
  - It has a red color and a circle shape defined using the appropriate → variables.

```
<entities>
  <→ species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb 'red'"/>
    <string name="shape" value="'circle'"/>
  </→ species>
</entities>
```

Here we add an entities section containing the definition of a → species. A → species is the prototype of an agent.

Please note that the visible skill provides default value for the shape and the color (circle and black).

- What is the behaviour of a bug?
  - It select a destination cell or 'place' within a distance of 4 cells where there is no agent.
  - it stays at the same cell only if there is no neighbour place empty.

```
  <reflex name="basic_move">
    <let var="place" value="stupid_grid location"/>
    <let var="destination" value="one_of ((place neighbours_at
4) where (empty each.agents))"/>
    <if condition="destination != nil">
      <set var="location" value="destination"/>
    </if>
  </reflex>
```

We add a reflex within the  $\rightarrow$  species section which declare the behaviour of the bug. This reflex will be executed at each time step by the agent.

First we declare a temporary variable, using the `let  $\rightarrow$`  command, to hold the current cell of the agent calling it 'place', to do so we cast explicitly our location, built-in variable of the situated skill, into a cell by using the name of the environment, "stupidGrid". This cast is quite powerful as it translate a coordinate (location) into a cell, you will discover several powerful cast like this later on.

Then we declare the 'destination' as a cell which is one of the empty (*agents=[]*) neighbour 'place'.

Finally we check that the 'destination' variable is not null and the agent moves, if 'destination' is null (which means that all neighbour are already full) it stays where it is.

### Model 1.3 : Instantiating bugs

- How to instantiate the 100 bugs?
  - As we have no information they will be placed randomly by the system.
  - We introduce here the global section which is responsible to hold global variables and process global action.

```
<global>
  <init>
    <create species="bug" number="100"/>
  </init>
</global>
```

We add a global section which contains an init subsection where we call the create command. The init section will be executed upon the creation of the entity. Here the entity is the system itself, we call it the "**world**". Consequently the bugs will be created before the start of the simulation and will be placed randomly on the default environment (the *stupidGrid*).

#### Nota bene

In GAMA we cannot choose when to draw the agent thus the "The display is updated at the end of each time step." statement is of no interest (though it is the case).

By default, the GAMA random generator is initialized pseudo randomly as it is the basic java one thus the "Instead of specifying which random number generation algorithm to use, the default generator for each platform is used. " is fulfilled.

### Complete model 1

```
<stupidmodel>
  <global>
    <init>
      <create species="bug" number="100"/>
    </init>
  </global>
  <environment>
    <grid name="stupid_grid" width="100" height="100"
torus="true"/>
  </environment>
  <entities>
    <-> species name="bug" skills="situated, visible">
```

```

    <rgb name="color" value="rgb 'red'"/>
    <string name="shape" value="'circle'"/>
    <reflex name="basic_move">
      <let var="place" value="stupid_grid location"/>
      <let var="destination" value="one_of ((place neighbours_at
4) where (emptyeach.agents))"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </reflex>
  </→ species>
</entities>
<output>
  <display name="grid" type="grid" environment="stupid_grid"/>
</output>
</stupidmodel>

```

## StupidModel10

---

### Purpose

Show how to sort a list of agents, and cause an agent action to be executed in size order.

### Formulation

- The list of bugs is sorted by descending size order at the start of each time step.
- The bugs' move action is un-randomized so it is executed in descending size order.

### Models

In the previous → model we already defined the actions and shuffled a list of agents. Now we only need to sort the list according to the size descending order. We would change the world's reflex and define a list like this:

```
<let var="orderedBugs" value="bugs sort_by (bug each).size"/>
```

### Complete model

We obtain the following model:

```

<stupidmodel>
  <global>
  <int name="numberBugs" value="100" parameter="true"/>
  <float name="globalMaxConsumption" value="1" parameter="true"/>
  <float name="globalMaxFoodProdRate" value="0.01"
parameter="true"/>
  <list name="bugs" value="list bug"/>
  <init>
    <create species="bug" number="numberBugs"/>

```



```

</init>
<reflex name="shouldHalt" if="!(empty (bugs where (each.size
> 100)))">
  <do action="halt"/>
</reflex>
<reflex name="managed_execution">
  <let var="orderedBugs" value="bugs sort_by (bug each).size
"/>
  <ask target="orderedBugs">
    <do action="basic_move"/>
    <do action="grow"/>
  </ask>
</reflex>
</global>
<environment>
  <grid name="stupid_grid" width="100" height="100"
torus="true">
    <rgb name="color" init="rgb 'black'"/>
    <float name="maxFoodProdRate"
value="globalMaxFoodProdRate"/>
    <float name="foodProd" value="(rnd(1000) / 1000) *
maxFoodProdRate"/>
    <float name="food" init="0.0" value="food + foodProd"/>
  </grid>
</environment>
<entities>
  <→ species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb [255, 255/size, 255/size]"/>
    <string name="shape" value="'circle'"/>
    <float name="maxConsumption" value="globalMaxConsumption"/>
    <stupid_grid name="myPlace" value="stupid_grid location"/>
    <action name="basic_move">
      <let var="destination" value="one_of ((myPlace neighbours_at
4) where (empty each.agents))"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </action>
    <action name="grow">
      <let var="transfer" value="min [maxConsumption,
myPlace.food]"/>
      <set var="size" value="size + transfer"/>
      <set var="myPlace.food" value="myPlace.food -
transfer"/>
    </action>
  </→ species>
</entities>

```

```
<output>
  <display name="grid" type="grid" environment="stupid_grid"/>
  <inspect name="Agents" type="agent" refresh_every="5"/>
  <inspect name="Species" type="species" refresh_every="5"/>
  <chart type="pie" name="Size distribution" background="rgb
'lightGray'" refresh_every="5" style="exploded">
    <data name="[0;10]" value="bugs [[Operators#count|count]]
(each.size &lt; 10)"/>
    <data name="[10;20]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 10) &amp;
([[Keywords#each|each]].size &lt; 20)"/>
    <data name="[20;30]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 20) &amp;
([[Keywords#each|each]].size &lt; 30)"/>
    <data name="[30;40]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 30) &amp;
([[Keywords#each|each]].size &lt; 40)"/>
    <data name="[40;50]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 40) &amp;
([[Keywords#each|each]].size &lt; 50)"/>
    <data name="[50;60]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 50) &amp;
([[Keywords#each|each]].size &lt; 60)"/>
    <data name="[60;70]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 60) &amp;
([[Keywords#each|each]].size &lt; 70)"/>
    <data name="[70;80]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 70) &amp;
([[Keywords#each|each]].size &lt; 80)"/>
    <data name="[80;90]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 80) &amp;
([[Keywords#each|each]].size &lt; 90)"/>
    <data name="[90;100]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 90) &amp;
([[Keywords#each|each]].size &lt; 100)"/>
  </chart>
  <file name="stupid_results" type="text" data="'cycle '+ (string
time)+ ': minSize '+ (string (bugs min_of each.size)) + ': maxSize '+
(string (bugs max_of each.size))"/>
</output>
</stupidmodel>
```

# StupidModel11

---

## Purpose

Show how agents can identify and rank neighbor cells. Illustrate how to iterate over a list.

## Formulation

- In its move method, a bug identifies a list of all cells that are within a distance of 4 grids but do not have another bug in them. (The bug's current cell is included on this list.)
- The bug iterates over the list and identifies the cell with highest food availability. The bug then moves to that cell.

## Models

We have to filter cells in two ways here. First we will remove, all the accessible with an agent then we select one with most food. To do so we will only change the definition of the temporary variable *destination*.

```
<let var="destination" value="last ((myPlace neighbours_at 4) where
(empty each.agents)) sort_by ((stupid_grid each).food)"/>
```

- As before we filter cells without agent (which we will call *filtered\_List*):

```
(myPlace neighbours_at 4) where (empty each.agents)
```

- Then we sort by ascending of food (which we will *ordered\_list*):

```
filtered_List sort_by ((stupid_grid each).food)
```

- Finally we get the last (because we are in ascending order) cell of the list:

```
last ordered_list
```

## Nota Bene

This is one possibility to get a cell with the maximum food, it is also to work with the `with_max_of`. We leave this to you as an exercise

## Complete model

We obtain the following model:

```
<stupidmodel>
  <global>
  <int name="numberBugs" value="100" parameter="true"/>
  <float name="globalMaxConsumption" value="1" parameter="true"/>
  <float name="globalMaxFoodProdRate" value="0.01"
parameter="true"/>
  <list name="bugs" value="list bug"/>
    <init>
      <create species="bug" number="numberBugs"/>
    </init>
    <reflex name="shouldHalt" if="!(empty (bugs where (each.size
> 100))) ">
```

```

    <do action="halt"/>
</reflex>
<reflex name="managed_execution">
  <let var="orderedBugs" value="bugs sort_by (bug each).size
"/>
  <ask target="orderedBugs">
    <do action="basic_move"/>
    <do action="grow"/>
  </ask>
</reflex>
</global>
<environment>
  <grid name="stupid_grid" width="100" height="100"
torus="true">
    <rgb name="color" init="rgb 'black'"/>
    <float name="maxFoodProdRate"
value="globalMaxFoodProdRate"/>
    <float name="foodProd" value="(rnd(1000) / 1000) *
maxFoodProdRate"/>
    <float name="food" init="0.0" value="food + foodProd"/>
  </grid>
</environment>
<entities>
  <→ species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb [255, 255/size, 255/size]"/>
    <string name="shape" value="'circle'"/>
    <float name="maxConsumption" value="globalMaxConsumption"/>
    <stupid_grid name="myPlace" value="stupid_grid location"/>
    <action name="basic_move">
      <let var="destination" value="last ((myPlace neighbours_at
4) where (empty each.agents)) sort_by ((stupid_grid each).food)"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </action>
    <action name="grow">
      <let var="transfer" value="min [maxConsumption,
myPlace.food]"/>
      <set var="size" value="size + transfer"/>
      <set var="myPlace.food" value="myPlace.food -
transfer"/>
    </action>
  </→ species>
</entities>
<output>
  <display name="grid" type="grid" environment="stupid_grid"/>
  <inspect name="Agents" type="agent" refresh_every="5"/>

```

```
<inspect name="Species" type="species" refresh_every="5"/>
<chart type="pie" name="Size distribution" background="rgb
'lightGray'" refresh_every="5" style="exploded">
  <data name="[0;10]" value="bugs [[Operators#count|count]]
(each.size &lt; 10)"/>
  <data name="[10;20]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 10) &amp;
([[Keywords#each|each]].size &lt; 20)"/>
  <data name="[20;30]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 20) &amp;
([[Keywords#each|each]].size &lt; 30)"/>
  <data name="[30;40]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 30) &amp;
([[Keywords#each|each]].size &lt; 40)"/>
  <data name="[40;50]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 40) &amp;
([[Keywords#each|each]].size &lt; 50)"/>
  <data name="[50;60]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 50) &amp;
([[Keywords#each|each]].size &lt; 60)"/>
  <data name="[60;70]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 60) &amp;
([[Keywords#each|each]].size &lt; 70)"/>
  <data name="[70;80]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 70) &amp;
([[Keywords#each|each]].size &lt; 80)"/>
  <data name="[80;90]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 80) &amp;
([[Keywords#each|each]].size &lt; 90)"/>
  <data name="[90;100]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 90) &amp;
([[Keywords#each|each]].size &lt; 100)"/>
</chart>
<file name="stupid_results" type="text" data="'cycle '+ (string
time)+ ': minSize '+ (string (bugs min_of each.size)) + ': maxSize '+
(string (bugs max_of each.size))"/>
</output>
</stupidmodel>
```

# StupidModel12

---

## Purpose

Show how to “kill” and drop objects from a model, and how to create new objects during a run.

## Formulation

- When a bug’s size reaches 10, it reproduces by splitting into 5 new bugs. Each new bug has an initial size of 0.0, and the old bug disappears.
- New bugs are placed at the first empty location randomly selected within +/- 3 cells of their parent’s last location. If no location is identified within 5 random draws, then the new bug dies.
- A new bug parameter “survivalProbability” is initialized to 0.95. Each time step, each bug draws a uniform random number, and if it is greater than survivalProbability, the bug dies and is dropped.
- This mortality action is scheduled after the bug moves and grows.
- The model stopping rule is changed: the model stops after 1000 time steps have been executed or when the number of bugs reaches zero.

## Models

### Multiplying

To make an agent dies and create new agent is pretty easy, it is done with the create command and the die primitive ([link to add @@@TODO@@@](#)). We will do it within a new action that will be called the by the world (a reflex would have been good if we were not using fixed order of agent).

```
<action name="multiply">
  <if condition="size > 10">
    <let var="possible_nests" value="(myPlace neighbours_at 3) where
(empty each.agents)"/>
    <loop times="5">
      <let var="nest" value="one_of possible_nests"/>
      <if condition="nest != nil">
        <set var="possible_nests" value="possible_nests - nest"/>
        <create species="bug" number="1" result="child"/>
        <ask target="child">
          <set var="location" value="nest.location"/>
        </ask>
      </if>
    </loop>
    <do action="die"/>
  </if>
</action>
```

## Introducing survivability

We have to introduce a new variable 'survivalProbability', we will set it at 95 (percent... because our random generator works on integer only). We will also define action that will manage the death of agents using this parameter. We will leave you this as an exercise, do not forget to define a global parameter in order to let the user change it.

```
<action name="shallDie">
  <if condition="(rnd 100) > 95">
    <do action="die"/>
  </if>
</action>
```

## Changing the stop condition

We have to modify the shouldHalt reflex as follow:

```
<reflex name="shouldHalt" if="(time > 1000) or ((length bugs) =
0) ">
  <do action="halt"/>
</reflex>
```

## Complete model

@@@TODO@@@ all the links in the source code don't work

We obtain the following model:

```
<stupidmodel>
  <global>
  <int name="numberBugs" value="100" parameter="true"/>
  <float name="globalMaxConsumption" value="1" parameter="true"/>
  <float name="globalMaxFoodProdRate" value="0.01"
parameter="true"/>
  <list name="bugs" value="list bug"/>
  <int name="globalSurvivalProbability" value="95"
parameter="true"/>
  <init>
    <create species="bug" number="numberBugs"/>
  </init>
  <reflex name="shouldHalt" if="(time > 1000) or ((length bugs)
= 0) ">
    <do action="halt"/>
  </reflex>
  <reflex name="managed_execution">
    <let var="orderedBugs" value="bugs sort_by (bug
each).size"/>
    <ask target="orderedBugs">
      <do action="basic_move"/>
      <do action="grow"/>
      <do action="multiply"/>
      <do action="shallDie"/>
```

```

    </ask>
  </reflex>
</global>
<environment>
  <grid name="stupid_grid" type="'square'" width="100"
height="100" torus="true">
    <rgb name="color" init="rgb 'black'"/>
    <float name="maxFoodProdRate"
value="globalMaxFoodProdRate"/>
    <float name="foodProd" value="(rnd(1000) / 1000) *
maxFoodProdRate"/>
    <float name="food" init="0.0" value="food + foodProd"/>
  </grid>
</environment>
<entities>
  <species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb [255, 255/size, 255/size]"/>
    <string name="shape" value="'circle'"/>
    <float name="maxConsumption" value="globalMaxConsumption"/>
    <stupid_grid name="myPlace" value="stupid_grid location"/>
    <action name="basic_move">
      <let var="destination" value="last ((myPlace neighbours_at
4) where (empty each.agents)) sort_by ((stupid_grid
each).food)"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </action>
    <action name="grow">
      <let var="transfer" value="min [maxConsumption,
myPlace.food]"/>
      <set var="size" value="size + transfer"/>
      <set var="myPlace.food" value="myPlace.food -
transfer"/>
    </action>
    <action name="multiply">
      <if condition="size > 10">
        <let var="possible_nests" value="(myPlace neighbours_at 3)
where (empty each.agents)"/>
        <loop times="5">
          <let var="nest" value="one_of possible_nests"/>
          <if condition="nest != nil">
            <set var="possible_nests" value="possible_nests -
nest"/>
            <create species="bug" number="1" result="child"/>
            <ask target="child">
              <set var="location" value="nest.location"/>

```



```

        </ask>
    </if>
</loop>
    <do action="die"/>
</if>
</action>
<action name="shallDie">
    <if condition="(rnd 100) > globalSurvivalProbability">
        <do action="die"/>
    </if>
</action>
</species>
</entities>
<[[Output|output]]>
    <[[Output#display|display]] name="grid"
type="[[Output#grid|grid]]" environment="stupid_grid"/>
    <[[Output#inspect|inspect]] name="Agents" type="agent"
refresh_every="5"/>
    <[[Output#inspect|inspect]] name="Species" type="species"
refresh_every="5"/>
    <[[Output#chart|chart]] type="[[Output#pie|pie]]" name="Size
distribution" background="[[Types#rgb|rgb]] 'lightGray'"
refresh_every="5" style="exploded">
        <data name="[0;10]" value="bugs [[Operators#count|count]]
(each.size < 10)"/>
        <data name="[10;20]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 10) &
([[Keywords#each|each]].size < 20)"/>
        <data name="[20;30]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 20) &
([[Keywords#each|each]].size < 30)"/>
        <data name="[30;40]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 30) &
([[Keywords#each|each]].size < 40)"/>
        <data name="[40;50]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 40) &
([[Keywords#each|each]].size < 50)"/>
        <data name="[50;60]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 50) &
([[Keywords#each|each]].size < 60)"/>
        <data name="[60;70]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 60) &
([[Keywords#each|each]].size < 70)"/>
        <data name="[70;80]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 70) &
([[Keywords#each|each]].size < 80)"/>
        <data name="[80;90]" value="bugs [[Operators#count|count]]

```

```

([[Keywords#each|each]].size > 80) &
([[Keywords#each|each]].size < 90)"/>
    <data name="[90;100]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 90) &
([[Keywords#each|each]].size < 100)"/>
    </[[Output#chart|chart]]>
    <[[Output#file|file]] name="stupid_results" type="text"
data="'cycle '+ ([[Operators#string|string]] time)+ ': minSize '+
([[Operators#string|string]] (bugs [[Operators#min_of|min]]_of
[[Keywords#each|each]].size)) + ': maxSize '+
([[Operators#string|string]] (bugs [[Operators#max_of|max_of]]
[[Keywords#each|each]].size))"/>
    <[[Output#chart|chart]] name="Population history"
type="[[Output#series|series]]" background="[[Types#rgb|rgb]]
'lightGray'" refresh-every="10">
    <data name="Bugs" value="length (list bug)"
color="[[Types#rgb|rgb]] 'blue'"/>
    </[[Output#chart|chart]]>
</[[Output|output]]>
</stupidmodel>

```

## StupidModel13

---

### Purpose

Show how to add a simple time series graph to a model. This graph is important for understanding results now that reproduction and mortality change the abundance of bugs.

### Formulation

No change is made to the model formulation. A graph is added to display the number of bugs alive at each time step.

### Models

We → previously defined a pie chart, the time series one follow the same structure, with the help of the output page this exercise will be easy for you.

### Complete model

@@@TODO@@@ **all the links in the source code don't work** You should obtain a model similar to this:

```

<stupidmodel>
  <global>
    <int name="numberBugs" value="100" parameter="true"/>
    <float name="globalMaxConsumption" value="1"
parameter="true"/>
    <float name="globalMaxFoodProdRate" value="0.01"
parameter="true"/>

```

```

<list name="bugs" value="list bug"/>
<int name="globalSurvivalProbability" value="95"
parameter="true"/>
<init>
  <create species="bug" number="numberBugs"/>
</init>
<reflex name="shouldHalt" if="(time > 1000) or ((length bugs)
= 0) ">
  <do action="halt"/>
</reflex>
<reflex name="managed_execution">
  <let var="orderedBugs" value="bugs sort_by (bug
each).size"/>
  <ask target="orderedBugs">
    <do action="basic_move"/>
    <do action="grow"/>
    <do action="multiply"/>
    <do action="shallDie"/>
  </ask>
</reflex>
</global>
<environment>
  <grid name="stupid_grid" type="'square'" width="100"
height="100" torus="true">
    <rgb name="color" init="rgb 'black'"/>
    <float name="maxFoodProdRate"
value="globalMaxFoodProdRate"/>
    <float name="foodProd" value="(rnd(1000) / 1000) *
maxFoodProdRate"/>
    <float name="food" init="0.0" value="food + foodProd"/>
  </grid>
</environment>
<entities>
  <species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb [255, 255/size, 255/size]"/>
    <string name="shape" value="'circle'"/>
    <float name="maxConsumption" value="globalMaxConsumption"/>
    <stupid_grid name="myPlace" value="stupid_grid location"/>
    <action name="basic_move">
      <let var="destination" value="last ((myPlace neighbours_at
4) where (empty each.agents)) sort_by ((stupid_grid
each).food)"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </action>
    <action name="grow">

```

```

        <let var="transfer" value="min [maxConsumption,
myPlace.food]"/>
        <set var="size" value="size + transfer"/>
        <set var="myPlace.food" value="myPlace.food -
transfer"/>
    </action>
    <action name="multiply">
        <if condition="size > 10">
            <let var="possible_nests" value="(myPlace neighbours_at 3)
where (empty each.agents)"/>
            <loop times="5">
                <let var="nest" value="one_of possible_nests"/>
                <if condition="nest != nil">
                    <set var="possible_nests" value="possible_nests -
nest"/>

                    <create species="bug" number="1" result="child"/>
                    <ask target="child">
                        <set var="location" value="nest.location"/>
                    </ask>
                </if>
            </loop>
            <do action="die"/>
        </if>
    </action>
    <action name="shallDie">
        <if condition="(rnd 100) > globalSurvivalProbability">
            <do action="die"/>
        </if>
    </action>
</species>
</entities>
<[[Output|output]]>
    <[[Output#display|display]] name="grid"
type="[[Output#grid|grid]]" environment="stupid_grid"/>
    <[[Output#inspect|inspect]] name="Agents" type="agent"
refresh_every="5"/>
    <[[Output#inspect|inspect]] name="Species" type="species"
refresh_every="5"/>
    <[[Output#chart|chart]] type="[[Output#pie|pie]]" name="Size
distribution" background="[[Types#rgb|rgb]] 'lightGray'"
refresh_every="5" style="exploded">
        <data name="[0;10]" value="bugs [[Operators#count|count]]
(each.size < 10)"/>
        <data name="[10;20]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 10) &
([[Keywords#each|each]].size < 20)"/>
        <data name="[20;30]" value="bugs [[Operators#count|count]]

```

```

([[Keywords#each|each]].size > 20) &
([[Keywords#each|each]].size < 30)"/>
    <data name="[30;40]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 30) &
([[Keywords#each|each]].size < 40)"/>
    <data name="[40;50]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 40) &
([[Keywords#each|each]].size < 50)"/>
    <data name="[50;60]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 50) &
([[Keywords#each|each]].size < 60)"/>
    <data name="[60;70]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 60) &
([[Keywords#each|each]].size < 70)"/>
    <data name="[70;80]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 70) &
([[Keywords#each|each]].size < 80)"/>
    <data name="[80;90]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 80) &
([[Keywords#each|each]].size < 90)"/>
    <data name="[90;100]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size > 90) &
([[Keywords#each|each]].size < 100)"/>
</[[Output#chart|chart]]>
<[[Output#file|file]] name="stupid_results" type="text"
data="'cycle '+ ([[Operators#string|string]] time)+ ': minSize '+
([[Operators#string|string]] (bugs [[Operators#min_of|min]]_of
[[Keywords#each|each]].size)) + ': maxSize '+
([[Operators#string|string]] (bugs [[Operators#max_of|max_of]]
[[Keywords#each|each]].size))"/>
    <[[Output#chart|chart]] name="Population history"
type="[[Output#series|series]]" background="[[Types#rgb|rgb]]
'lightGray'" refresh-every="10">
    <data name="Bugs" value="length (list bug)"
color="[[Types#rgb|rgb]] 'blue'"/>
    </[[Output#chart|chart]]>
</[[Output|output]]>
</stupidmodel>

```

# StupidModel14

---

## Purpose

Illustrate use of random number distributions. A common use of them is to induce variability among initial individuals.

## Formulation

- Two new model parameters are added, and put on the parameter settings window: `initialBugSizeMean` and `initialBugSizeSD`. Values of these parameters are 0.1 and 0.03.
- Instead of initializing bug sizes to 1.0 (Sect. 2.2), sizes are drawn from a normal distribution defined by `initialBugSizeMean` and `initialBugSizeSD`. The initial size of bugs produced via reproduction is 0.0.
- Negative values are very likely to be drawn from normal distributions such as the one used here. To avoid them, a check is introduced to limit initial bug size to a minimum of zero.

## Models

### Adding new parameters

You already know how to add variables and make them parameters.

### Normal randomization of bug size

There is not yet a normal distribution in GAMA though there is a truncated normal distribution thanks to the `TGauss` operator which would be done like this:

```
<float name="size" value="TGauss
[initialBugSizeMean,initialBugSizeSD] "/>
```

### Negative value check

#### Basic style

Because we use a truncated normal distribution we will not get any negative value though it is interesting to know how to prevent that if it were the case. It is possible to do in a very unelegant way:

```
<float name="size" init="Gauss [initialBugSizeMean,initialBugSizeSD]" min="0" "/>
```

nb: `gauss` would be the not truncated normal distribution operator (yet to be implemented).

Unfortunately it will replace all negative values by 0 which will create a peak of probability there.

#### Correct style

If we can generate negative value (or more generally a value outside a desired range) we would have to regenerate new values until we generate a positive one. If we want to do it correctly we need to define a sequence of primitive within the agent's `init`, which would be like this:

```
<init name="generateSize" if="(time > 1000) or ((length bugs) =
0) ">
  <let var="size" value="Gauss
[initialBugSizeMean,initialBugSizeSD] "/>
  <loop while="size < 0">
```

```

    <let var="size" value="Gauss
[initialBugSizeMean,initialBugSizeSD]"/>
  </loop">
</init>

```

## Complete model

We obtain the following model:

```

<stupidmodel>
  <global>
<int name="numberBugs" value="100" parameter="true"/>
<float name="globalMaxConsumption" value="1" parameter="true"/>
<float name="globalMaxFoodProdRate" value="0.01"
parameter="true"/>
<list name="bugs" value="list bug"/>
  <int name="globalSurvivalProbability" value="95"
parameter="true"/>
  <float name="initialBugSizeMean" value="0.1" parameter="true"/>
  <float name="initialBugSizeSD" value="0.03" parameter="true"/>
  <init>
    <create species="bug" number="numberBugs"/>
  </init>
  <reflex name="shouldHalt" if="(time > 1000) or ((length bugs)
= 0) ">
    <do action="halt"/>
  </reflex>
  <reflex name="managed_execution">
    <let var="orderedBugs" value="bugs sort_by (bug each).size"
"/>
    <ask target="orderedBugs">
      <do action="basic_move"/>
      <do action="grow"/>
      <do action="multiply"/>
      <do action="shallDie"/>
    </ask>
  </reflex>
</global>
<environment>
  <grid name="stupid_grid" type="'square'" width="100"
height="100" torus="true">
    <rgb name="color" init="rgb 'black'"/>
    <float name="maxFoodProdRate"
value="globalMaxFoodProdRate"/>
    <float name="foodProd" value="(rnd(1000) / 1000) *
maxFoodProdRate"/>
    <float name="food" init="0.0" value="food + foodProd"/>
  </grid>

```

```

</environment>
<entities>
  <-> species name="bug" skills="situated, visible">
    <rgb name="color" value="rgb [255, 255/size, 255/size]"/>
    <string name="shape" value="'circle'"/>
    <float name="maxConsumption" value="globalMaxConsumption"/>
    <stupid_grid name="myPlace" value="stupid_grid location"/>
    <float name="size" value="TGauss
[initialBugSizeMean,initialBugSizeSD]"/>
    <action name="basic_move">
      <let var="destination" value="last ((myPlace neighbours_at
4) where (empty each.agents)) sort_by ((stupid_grid each).food)"/>
      <if condition="destination != nil">
        <set var="location" value="destination"/>
      </if>
    </action>
    <action name="grow">
      <let var="transfer" value="min [maxConsumption,
myPlace.food]"/>
      <set var="size" value="size + transfer"/>
      <set var="myPlace.food" value="myPlace.food -
transfer"/>
    </action>
    <action name="multiply">
      <if condition="size > 10">
        <let var="possible_nests" value="(myPlace neighbours_at 3)
where (empty each.agents)"/>
        <loop times="5">
          <let var="nest" value="one_of possible_nests"/>
          <if condition="nest != nil">
            <set var="possible_nests" value="possible_nests -
nest"/>
            <create species="bug" number="1" result="child"/>
            <ask target="child">
              <set var="location" value="nest.location"/>
            </ask>
          </if>
        </loop>
        <do action="die"/>
      </if>
    </action>
    <action name="shallDie">
      <if condition="(rnd 100) > globalSurvivalProbability">
        <do action="die"/>
      </if>
    </action>
  </-> species>

```



```

</entities>
<output>
  <display name="grid" type="grid" environment="stupid_grid"/>
  <inspect name="Agents" type="agent" refresh_every="5"/>
  <inspect name="Species" type="species" refresh_every="5"/>
  <chart type="pie" name="Size distribution" background="rgb
'lightGray'" refresh_every="5" style="exploded">
    <data name="[0;10]" value="bugs [[Operators#count|count]]
(each.size &lt; 10)"/>
    <data name="[10;20]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 10) &amp;
([[Keywords#each|each]].size &lt; 20)"/>
    <data name="[20;30]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 20) &amp;
([[Keywords#each|each]].size &lt; 30)"/>
    <data name="[30;40]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 30) &amp;
([[Keywords#each|each]].size &lt; 40)"/>
    <data name="[40;50]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 40) &amp;
([[Keywords#each|each]].size &lt; 50)"/>
    <data name="[50;60]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 50) &amp;
([[Keywords#each|each]].size &lt; 60)"/>
    <data name="[60;70]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 60) &amp;
([[Keywords#each|each]].size &lt; 70)"/>
    <data name="[70;80]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 70) &amp;
([[Keywords#each|each]].size &lt; 80)"/>
    <data name="[80;90]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 80) &amp;
([[Keywords#each|each]].size &lt; 90)"/>
    <data name="[90;100]" value="bugs [[Operators#count|count]]
([[Keywords#each|each]].size &gt; 90) &amp;
([[Keywords#each|each]].size &lt; 100)"/>
  </chart>
  <file name="stupid_results" type="text" data="'cycle '+ (string
time)+ ': minSize '+ (string (bugs min_of each.size)) + ': maxSize '+
(string (bugs max_of each.size))"/>
  <chart name="Population history" type="series" background="rgb
'lightGray'" refresh-every="10">
    <data name="Bugs" value="length (list bug)"
color="[[Types#rgb|rgb]] 'blue'"/>
  </chart>
</output>
</stupidmodel>

```

# StupidModel15

---

## Purpose

Show how to read spatial data in from a file.

## Formulation

- Instead of assuming the space size and assuming cell food production is random (model 3), food production rates are read in from a file. The file also determines the space size.
- The file contains one line per cell, with (a) X coordinate, (b) Y coordinate, and (c) food production rate.
- Food production in a cell is now equal to the production rate read in from the file, and is no longer random.
- Now, because we are representing real habitat with real data, it no longer makes sense for the space to be toroidal. So the space objects and movement-related methods must be modified so bugs cannot move off the edge of their space.
- The input file is Stupid\_Cell.Data. It has X, Y, and food production data for a grid space. X ranges from 0 to 250; Y ranges from 0 to 112. The file starts with three lines of header information that is ignored by the model.
- The cells are now displayed and colored to indicate their current food availability. Cell colors scale from black when cell food availability is zero to green when food availability is 0.5 or higher.
- A change to the bug move method is required to avoid a very strong artifact now that cell food production is no longer random. Near the start of a simulation, many cells will have exactly the same food availability, so a bug simply would move to the first cell on its list of neighbor cells. This is always the top-left cell among the neighbors, so bugs move constantly up and left if all the cells available to them have the same food availability. This artifact is removed by randomly shuffling the list of available cells before the bug loops through it to identify the best.

## Models

@@@TODO@@@ TO BE DONE TO BE DONE TO BE DONE TO BE DONE TO BE DONE TO BE DONE TO BE DONE TO BE DONE TO BE DONE

## Complete model

We obtain the following model: @@@TODO@@@

---

# Modeling guide

---

## Model structure

In GAML, a model is an XML file (or set of files, included in each other) that consist(s) of a number of  $\rightarrow$  sections, among them three are mandatory:

```
<global>
  The definition of the structure and behaviors of the world.
  This is where global data, parameters and dynamics are declared.
</global>

<environment>
  The definition of the environments that compose the world.
  Some of them (like grids) define species of environmental agents.
</environment>

<entities>
  The definition of the species of agents that populate the
environments.
</entities>
```

In addition, GAMA needs to know how to simulate the model and what kind of information are to be extracted from it. Two more sections are then necessary:

```
<output>
  Which specifies the desired outputs of a simulation (graphical
displays, charts, files, etc.).
</output>

<batch> (optional)
  Which can be added to describe the experimental protocols to use
for simulating the model.
</batch>
```

## Species structure

The major entities of a model (the *world*, the agents, some environments) are described in the form of  $\rightarrow$  *species*. Species are composed of:

- *variables* and specific declarations: define *what their agents know*.
- *actions*: define *what their agents can do*.
- $\rightarrow$  *behaviors*: define *what they actually do*.

Species can inherit from other species their variables, actions and behaviors. They can also be defined to make use of:

- $\rightarrow$  *skills*: built-in modules that pack together related variables and actions and make them available to the agents.
  - *controls*: built-in modules that offer alternative ways to describe behaviors (like *finite state machines*, for instance).
-

## GAML Language

- Actions (except *built-in actions*) and behaviors are written as sequences of  $\rightarrow$  *commands*.
- Commands use expressions to define conditions, data changes, computations, etc.
- An expression is composed of variables,  $\rightarrow$  keywords and  $\rightarrow$  operators.
- Every expression (and, therefore, every variable) has a type: either a built-in  $\rightarrow$  *type* or a *species*, as species can be seen as extended *data types*.
- Species come preloaded with *built-in variables* and  $\rightarrow$  *skills variables* (and every species can make use of the *global built-in variables*). All these variables can be redefined in GAML and, of course, *new variables* can be declared.

## Behaviors

---

### Common behaviors

Basic agents (including the world and grid cells) are provided with a simple behavioral structure, based on *reflexes*. Species can define any number of reflexes within their body.

#### reflex

##### Attributes

- **when** or **if**: a boolean expression

##### Definition

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

```
<reflex>
  , Executed every time step
  <arg name="message" value="'Executing the unconditional
reflex'"/>
</reflex>
```

```
<reflex when="flip 0.5"> Only executed when flip returns true
  ,
  <arg name="message" value="'Executing the conditional
reflex'"/>
</reflex>
```

## init

### Attributes

- **when** or **if**: a boolean expression
- 

### Definition

A special form of reflex that is evaluated **only once when the agent is created**, after the initialization of its variables, and before it executes any reflex. Only one instance of *init* is allowed in each species (except in case of inheritance, see this section). Useful for creating all the agents of a model in the definition of the world, for instance.

## EMF-based behaviors

EMF (Etho Modeling Framework) is task based behavior model. Species can define any number of tasks within their body. At any given time, only one task having the highest priority value is executed.

## task

### Sub elements

Besides a sequence of  $\rightarrow$  commands like reflex, a task contains the following sub elements:

- **priority**: Mandatory. The activation level of the task.
  - **durationMin**: Optional. The minimal duration of the task. Until it is reached, the task cannot be interrupted.
  - **durationMax**: Optional. The maximal duration of the task. Beyond this duration, the task is automatically halted.
  - **whileCondition**: Optional. The "while" watchdog : a condition tested every step to ensure that the task can still be executed. If false, the task is stopped (except if the min duration is not reached)
  - **untilCondition**: Optional. The "until" watchdog, a condition tested every step to ensure that the task can still be executed. If true, the task is stopped (except if the min duration is not reached)
  - **endAction**: Optional. The action to execute in case of failure/interruption of the task. No repeat is allowed.
- 

### Definition

Like **reflex**, a task is a sequence of  $\rightarrow$  commands that can be executed, at each time step, by the agent. If an agent owns several tasks, the scheduler choses a task to execute basing on its current priority value. For an example of task, please have a look the definition of "*foodCarrier*" species in this Task exemple.

## FSM-based behaviors

FSM (Finite State Machine) is a finite state machine based behavior model. During its life cycle, agent possesses several states. At any given time step, an agent is in one state.

## state

### Attributes

- **initial**: a boolean expression, indicates the initial state of agent.
  - **final**: a boolean expression, indicates the final state of agent.
- 

### Sub elements

- **enter**: a sequence of  $\rightarrow$  commands to execute upon entering the state.
  - **exit**: a sequence of  $\rightarrow$  commands to execute right before exiting the state.
  - **transition**: specifies the next state of the life cycle.
-

**Definition**

A state like a reflex can contains several  $\rightarrow$  commands that can be executed, at each time step, by the agent. For an exemple of state, please have a look at the definition of "*ant*" species in this FSM exemple.

# Commands

**General syntax**

A command is an XML tag, followed by specific *attributes*, some of them mandatory (in **bold**), some of them optional (in *italic*).

```
<command_tag attribute1="expression1" attribute2="expression2" ...
/>
```

If the command encloses other commands, they are declared between the opening tag and a closing tag, as in:

```
<command_tag1 attribute1="expression1" attribute2="expression2" ...
>
    <command_tag2 attribute1="expression1" attribute2="expression2"
... >
    ...
    </command_tag2>
    <command_tag3 attribute1="expression1" attribute2="expression2"
... />
</command_tag1>
```

**ask****Attributes**

- **target**: an expression that evaluates to an agent or a list of agents
- *as*: an expression that evaluates to a species

**Definition**

Allows an agent, the *sender agent*, to ask another (or other) agent(s) to perform a set of commands. It obeys the following syntax, where the *target* attribute denotes the *receiver agent(s)*:

```
<ask target="receiver_agent(s)">
    [commands]*
</ask>
```

If the value of the *target* attribute is nil or empty, the command is ignored. The species of the receiver agents must be known in advance for this command to compile. If not, it is possible to cast them using the *as* attribute, like :

```
<ask target="receiver_agent(s)" as="a_species_expression">
    [command_set]
</ask>
```

Alternative forms for this casting are :

- if there is only a single receiver agent:

```
<ask target="[[Operators#agent|species_name]] receiver_agent">
  [command_set]
</ask>
```

- if receiver\_agent(s) is a list of agents:

```
<ask target="receiver_agents [[Operators#of_species|of_species]]
species_name">
  [commands]*
</ask>
```

Any command can be declared in *command\_set*, except *ask* itself (although an *ask* can be performed in an action called within the *command\_set*). All the commands 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 *command\_set*, it has to use the keyword *myself*.

```
<species name="animal">
  <float name="energy" init="rnd 1000" min="0.0"/>
  <reflex when="energy > 500"> executed when the energy is
above the given threshold
    <let name="others" value="(self neighbours_at 5) of_species
animal"/> find all the neighbouring animals in a radius of 5 meters
    <let name="shared_energy" value="(energy - 500) / length
others"/> compute the amount of energy to share with each of them
    <ask target="others"> no need to cast, since others has
already been filtered to only include animals
      <if condition="energy < 500"> refers to the
energy of each animal in others
        <set var="energy" value="energy +
myself.shared_energy"/> increases the energy of each animal
        <set var="myself.energy" value="myself.energy -
myself.shared_energy"/> decreases the energy of the sender
      </if>
    </ask>
  </reflex>
</species>
```

## create

### Attributes

- **species**: an expression that evaluates to a species
- **number**: an expression that evaluates to an int
- *return* or *result*: a temporary variable name

---

### Definition

Allows an agent to create *number* agents of species *species*. Its simple syntax is :

```
<create species="a_species" number="an_int"/>
```

---

If *number* equals 0 or *species* is not a species, the command is ignored. The agents created are initialized following the rules of their species. If one wants to refer to them after the command is executed, the *return* (or *result*) keyword has to be defined: the agents created will then be referred to by the temporary variable it declares. For instance, the following command 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="species self" number="rnd 4" result="children"/>
<ask target="children">
...
</ask>
```

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 species="a_species" number="an_int">
  [commands]*
</create>
```

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="species self" number="rnd 4" result="children">
  <set var="location" value="myself.location + {rnd 2, rnd 2}"
/> tells the children to be initially located close to me
  <set var="parent" value="myself" /> tells the children that
their parent is me (provided the variable parent is declared in this
species)
</create>
```

## do

### Attributes

- **action**: the name of an action or a primitive
- *return* or *result*: a temporary variable name

---

### Enclosed tags

- *arg* or a *datatype* → *name* : specify the arguments expected by the action/primitive to execute.

---

### Definition

Allows the agent to execute an action or a primitive. For a list of primitives available in every species, see this page; for the list of primitives defined by the different skills, see this → page. Finally, see this page to know how to declare custom actions. The simple syntax (when the action does not expect any argument and the result is not to be kept) is :

In case the result of the action needs to be made available to the agent, the *return* (or *result*) keyword has to be defined; the result will then be referred to by the temporary variable declared in this attribute:

In case the action expects one or more arguments to be passed, they are defined by using enclosed tags, either *arg* or the name of a → *datatype* if one wants to specify their type. For instance:

```
,
  <arg name="arg1" value="expression1"/>
```

---



```
<*datatype* name="arg2_of_type_datatype"
value="expression2_of_type_datatype"/>
...
```

## let

### Attributes

- **name** or **var**: the name of the temporary variable
- **value**: an expression

### Definition

Allows the agent to declare a temporary variable, local to the scope in which it is defined. The naming rules follow those of the  $\rightarrow$  |variables declarations. In addition, a temporary variable cannot be declared twice in the same scope. The generic syntax is :

```
<let name="temp_var1" value="an_expression"/>
```

The datatype of the variable is inferred from that of the expression (which can be enforced using casting operators if necessary). After it has been declared this way, a temporary variable can be used like regular variables (for instance, the set command should be used to assign it a new value within the same scope).

## set

### Attributes

- **name** or **var**: an expression that either returns a variable or an element of a composite  $\rightarrow$  type
- **value**: an expression

### Definition

Allows the agent to assign a value to a variable or an element of a composite variable (Types#string, list, matrix, point). See this section to know how to access variables. To access an element within a composite variable, the operator at has to be used. Examples:

```
<set name="my_var" value="expression"/>
<set name="temp_var" value="expression" />
<set name="global_var" value="expression" />
<set name="composite_var at index" value="expression" />
```

The variable assigned can be accessed in the *value* attribute. In that case, it represents the value of the variable (or the element) **before** it has been modified. Examples (with temporary variables):

```
<let name="my_list" value="[1,2,3]" />
<set name="my_list at 2" value="(my_list at 2) + 10" /> my_list
now equals [1,2,13]
<let name="my_int" value="1000"/>
<set name="my_int" value="my_int + 1"/> my_int now equals 1001
```

Assignments are the only way to create lists composed of list elements. For instance :

```
<let name="list_of_lists" value="[]"/>
<set name="list_of_lists at 0" value="[1,2,3]"/>
<set name="list_of_lists at 1" value="[4,5,6]"/>
<set name="list_of_lists at 2" value="[7,8,9]"/>
```

creates the list : [ [1,2,3], [4,5,6], [7,8,9] ].

## if

### Attributes

- **condition:** a boolean expression

---

### Enclosed tags

- *else* : encloses alternative commands

---

### Definition

Allows the agent to execute a sequence of commands if and only if the *condition* evaluates to true. The generic syntax is:

```
<if condition="bool_expr">
  [command] *
</if>
```

Optionally, the commands to execute when the *condition* evaluates to false can be defined in an enclosed command *else*. The syntax then becomes:

```
<if condition="bool_expr">
  [command] *
  <else>
    [command] *
  </else>
</if>
```

*ifs* and *elses* can be imbricated as needed. For instance:

```
<if condition="bool_expr">
  [command] *
  <else>
    <if condition="bool_expr2">
      [command] *
      <else>
        [command] *
      </else>
    </if>
  </else>
</if>
```

## loop

### Attributes

- **times**: an int expression, or
- **while**: a boolean expression, or
- **over**: a list expression, together with
- *var*: a temporary variable name

---

### Definition

Allows the agent to perform the same set of commands 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. The basic syntax for each of these usages are:

```
<loop times="an_int_expression">
  [command] *
</loop>
```

Or:

```
<loop while="a_bool_expression">
  [command] *
</loop>
```

Or:

```
<loop over="a_list_expression" var="a_temp_var">
  [command] *
</loop>
```

Or:

```
<loop from="int_expression_1" to="int_expression_2"
var="a_temp_var">
  [command] *
</loop>
```

In these latter two cases, the *var* attribute 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 :

```
<let name="a" value="0"/>
<loop over="[10, 20, 30]" var="i">
  <set var="a" value="a + i"/>
</loop> 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* attributes take a integer expression as arguments, with the first (resp. the last) specifying the beginning (resp. end) of the interval. The step is assumed equal to 1.

```
<let name="the_list" value="list (species_of self)"/>
<loop from="0" to="length the_list" var="i">
  <ask target="the_list at i"/>
  ...
</ask>
```

---

```
</loop> every agent of the list is asked to do something
```

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.

## return

### Attributes

- **value:** an expression

---

### Definition

Allows to specify which value to return from the evaluation of the surrounding command. Usually used within the declaration of an action. Contrary to other languages, using *return* **does not stop the evaluation** of the surrounding command (for instance, a *loop*). It simply indicates what value to return: if it is inside a *loop*, then, only the last evaluation of *return* will be returned. Example:

```
<action name="foo">
  <return value="'foo'"/>
</action>
...
<reflex>

  ,
  <arg name="message" value="foo_result"/>

</reflex>
```

... the agent will print foo on the console at each step

---

# Operators

---

## Definition

An operator performs a function on one, two, or three operands. An operator that only requires one operand is called a unary operator. An operator that requires two operands is a binary operator. And finally, a ternary operator is one that requires three operands. The GAML programming language has only one ternary operator, `?:`, which is a short-hand if-else statement.

Unary operators are written using a prefix notation. Prefix notation means that the operator appears *before* its operand:

```
operator operand
```

All the binary operators use an infix notation, which means that the operator appears *between* its operands:

```
op1 operator op2
```

The ternary operator is also infix; each component of the operator appears between operands:

```
op1 ? op2 : op3
```

In addition to performing operations, operators are functional, i.e. they return a value. The return value and its type depend on the operator and the type of its operands. For example, the arithmetic operators, which perform basic arithmetic operations such as addition and subtraction, return numbers - the result of the arithmetic operation.

Moreover, operators are strictly functional, i.e. they have **no side effects** on their operands. For instance, the shuffle operator, which randomizes the positions of elements in a list, **does not modify** its list operand but returns a new shuffled list.

## Operators by categories

### Arithmetic operators

abs cos sin tan exp sqrt float int ln rnd round + - / \* // div % mean mod ^ with\_precision

### Comparison operators

= != < > < >= <=

### Boolean operators

! & and || or flip

### Casting operators

agent bool float int string point rgb matrix list species

### List-related operators

any accumulate empty first last length list max min mul sum reverse shuffle at + - among collect copy\_between count first\_with last\_with in index\_of last\_index\_of max\_of min\_of with\_max\_of with\_min\_of of\_species sort\_by starts\_with where select

### String-related operators

any empty first last length list max min reverse shuffle at + - among collect copy\_between count first\_with last\_with in index\_of last\_index\_of max\_of min\_of with\_max\_of with\_min\_of sort\_by starts\_with where select > < >= <= tokenize

### Location-related operators

closest\_point\_to next\_point\_towards towards point distance\_to neighbours\_at

### Agent-related operators

agent species species\_of int string neighbours\_at of\_species

## Unary operators

Unary operators take one and only one operand. The syntax is : *operator operand*. The priority between operators is determined from right to left (i.e. in *op1 op2 op3 operand*, *op3* is evaluated before *op2* and *op1*).

!

- *Operand*: a boolean expression.
- *Result*: opposite boolean value.
- *Return type*: bool.
- *Special cases*: if the parameter is not boolean, it is casted to a boolean value.
- *see also*: bool

```
! true → false.
```

### abs

- *Operand*: an arithmetic expression
- *Result*: the absolute value of the operand
- *Return type*: a positive int or float depending on the type of the operand

```
abs (200 * -1 + 0.5) → 200.5
```

### \*species name\*

- *Operand*: an int, agent, point or string.
- *Result*: casting of the operand to an agent (if *\*species name\** is used, casting to an instance of *\*species name\**).
- *Return type*: agent (or *\*species name\**).
- *Special cases*:
  - if the operand is a point, returns the closest agent (resp. closest instance of *\*species name\**) to that point;
  - if the operand is a string, returns the agent (resp. instance of *\*species name\**) with this name;
  - if the operand is an agent, returns the agent (resp. tries to cast this agent to *\*species name\** and returns nil if the agent is instance of another species)
  - if the operand is an int, returns the agent (resp. instance of *\*species name\**) with this unique index;
- *see also*: of\_species, species

**any**

same signification as `one_of` operator.

**one\_of**

- *Operand*: a list, string or matrix.
- *Result*: a random element from the list.
- *Return type*: any.
- *Special cases*: returns **nil** if the list is empty.
- *Comment*: the operand is casted to a list before the expression is evaluated. Therefore, if *foo* is the name of a species, *any foo* will return a random agent from this species (see list).

```
any [1,2,3] → 1, 2, or 3.
one_of [1,2,3] → 1, 2, or 3.
```

**bool**

- *Operand*: any type.
- *Result*: casting of the operand to a boolean value.
- *Return type*: bool.
- *Special cases*: if the operand is an int or a float, returns true if it is greater than 0 (or 0.0). If the operand is a list or a string, bool is formally equivalent to *not empty* (a la Lisp). Otherwise, returns false.

```
bool 0 → false;
bool [1, 2, 3] → true;
```

**closest\_point\_to**

- *Operand*: a localized agent or a point.
- *Result*: the point, closest to the operand, that can be reached by the agent.
- *Return type*: point.
- *Special cases*: returns **nil** if the argument cannot be reached by the agent, or if the agent is neither localized nor able to move. Uses the path finder if the environment is a GIS.
- *see also*: `next_point_towards`, `towards`.

```
closest_point_to {0,0} → returns the point, closest to the origin, that
can be reached by the agent.
```

**cos**

- *Operand*: an int or a float.
- *Result*: the cosinus of the operand (in decimal degrees).
- *Return type*: float.
- *Special cases*: the argument is casted to an int before being evaluated. Integers outside the range "0-359" are normalized.
- *see also*: `sin`, `tan`.

```
cos 0 → 1.
```

## empty

- *Operand*: a list, string or matrix.
- *Result*: true if the argument is *empty*, false otherwise.
- *Return type*: bool.
- *Special cases*: matrices are never considered as empty, and thus always return false when the operand is a matrix.
- *Comments*: the operand is casted to a list before the expression is evaluated. Therefore, if *foo* is the name of a species, *empty foo* indicates whether or not agents of species *foo* are present in the model.

```
empty [] → true;  
empty 'abcd' → false;
```

## exp

- *Operand*: an int or a float.
- *Result*: returns Euler's number *e* raised to the power of the operand.
- *Return type*: float.
- *Special cases*: the operand is casted to a float before being evaluated.
- *see also*: ln.

```
exp 0 → 1.
```

## first

- *Operand*: a list, string, point or matrix.
- *Result*: the first element of the operand
- *Return type*: any.
- *Special cases*: if the operand is a string, returns a string composed of its first character.
- *see also*: last.
- *Comments*: first is used to grab the x-coordinate of a point;

```
first 'abce' → 'a'; first [1, 2, 3] → 1.  
first {10,12} → 10.
```

## flip

- *Operand*: an int or a float (between 0 and 1).
- *Result*: true or false given the probability represented by the operand.
- *Return type*: bool.
- *Special cases*: *flip 0* always returns false, *flip 1* true.
- *see also*: rnd.

```
flip 0.666666 → 2/3 chances to return true.
```



## float

- *Operand*: any type.
- *Result*: casting of the operand to a floating point value.
- *Return type*: float.
- *Special cases*: if the operand is an agent, returns its unique index as a float; if the operand is a string, tries to convert its content to a floating point value; if the argument is a list, a point or a matrix, returns its first element converted to a float; if the operand is a boolean, returns 1.0 for true and 0.0 for false; if the argument is a color, returns its RGB value as a float.
- *see also*: int.

```
float '234.0' → 234.0; float {12,23} → 12.0;
float true → 1.0;
float (rgb 'black') → 0.0;
```

## int

- *Operand*: any type.
- *Result*: casting of the operand to an integer value.
- *Return type*: int.
- *Special cases*: if the operand is a float, returns its value truncated (but not rounded); if the operand is an agent, returns its unique index; if the operand is a string, tries to convert its content to an integer value; if the argument is a list, a point or a matrix, returns its first element converted to an int; if the operand is a boolean, returns 1 for true and 0 for false; if the argument is a color, returns its RGB value as an integer; if the operand is a species, returns the number of its agents.
- *see also*: round, float.

```
int '234.3' → 234;
int {12,23} → 12;
int true → 1;
int (rgb 'black') → 0;
```

## intensity (deprecated)

- *Operand*: a string.
- *Result*: the intensity of the signal denoted by the operand at the agent's location.
- *Return type*: float.
- *Special cases*: returns 0.0 if the agent is not controlled by EMF, if the agent is not localized or if no grids have been defined as an environment.

```
intensity 'pheromone' → returns the intensity of the 'pheromone' signal
on the grid cell where the agent is located.
```

## last

- *Operand*: a list, string, point or matrix.
- *Result*: the last element of the operand
- *Return type*: any.
- *Special cases*: if the operand is a string, returns a string composed of its last character, or an empty string if the operand is empty.
- *see also*: first.
- *Comments*: last is used to grab the y-coordinate of a point; last {10,12} → 12.

```
last 'abce' → 'e'; last [1, 2, 3] → 3.
```

## length

- *Operand*: a list, string or matrix.
- *Result*: the number of elements contained in the list; the number of characters in a string; the size (number of cells) of a matrix.
- *Return type*: int.

```
length 'I am an agent' → 13; length [12,13] → 2.
```

## list

- *Operand*: any type.
- *Result*: casting of the operand to a list.
- *Return type*: list.
- *Special cases*:
  - if the operand is a point, returns a list containing its two coordinates;
  - if the operand is a string, returns a list of strings, each containing one character;
  - if the operand is a matrix, returns a list containing its elements;
  - if the operand is a rgb color, returns a list containing its three integer components;
  - if the operand is a species, return a list of its agents;
  - otherwise returns a list containing the operand.
- *see also*: +, - (for lists).
- *Comment*: list always tries to cast the operand except if it is an int, a bool or a float; to create a list, instead, containing the operand (including another list), use the + operator on an empty list (like [] + 'abc').

```
list 'abc' → ['a', 'b', 'c'];  
list 123 → '123';  
list ['a', 23] → ['a', 23];  
list (rgb 'black') → [0, 0, 0];
```

## ln

- *Operand*: an int or a float (greater than 0).
- *Result*: Returns the natural logarithm (base  $e$ ) of the operand.
- *Return type*: float.
- *Special cases*: an exception is raised if the operand is less than zero.
- *see also*: exp.

## max

- *Operand*: a list, point or matrix.
- *Result*: the maximum element found in the operand.
- *Return type*: float.
- *Special cases*: the elements of the operand are casted to float values before max is evaluated.
- *see also*: min.

```
max [100, 23.2, 34.5] → 100.0.
```

## mean

- *Operand*: a list, point or matrix.
- *Result*: the mean of all the elements of the operand.
- *Return type*: float.
- *Special cases*: the elements of the operand are casted to float values before summing up then the sum value is divided by the number of elements.
- *see also*: sum.

```
mean [4.5, 3.5, 5.5] → 4.5.
```

## min

- *Operand*: a list, point or matrix.
- *Result*: the minimum element found in the operand.
- *Return type*: float.
- *Special cases*: the elements of the operand are casted to float values before min is evaluated.
- *see also*: max.

```
min [100, 23.2, 34.5] → 23.2.
```

## mul

- *Operand*: a list, point or matrix.
- *Result*: the product of all the elements of the operand.
- *Return type*: float.
- *Special cases*: the elements of the operand are casted to float values before mul is evaluated..
- *see also*: sum.

```
mul [12, 10, 3] → 360.0.
```

## next\_point\_towards

- *Operand*: an agent or a point.
- *Result*: the next location of the agent if it heads towards the operand and keeps the same speed.
- *Return type*: point.
- *Special cases*: if the agent is localized but cannot move, returns the location of the agent; if the agent is not localized, returns nil. Uses the path finder if the environment is a GIS.
- *see also*: closest\_point\_to.

## reverse

- *Operand*: a list, point, string or matrix.
- *Result*: the operand elements in the reversed order.
- *Return type*: list, point or string.
- *Special cases*:
  - if the operand is a point {x,y}, returns a new point {y,x};
  - if the argument is a matrix, returns a list of its elements in the reversed order;
- *see also*: shuffle.

```
reverse 'abcd' → 'dcba';  
reverse {10,13} → {13, 10};  
reverse [10,12,14] → [14, 12, 10].
```

## rgb

- *Operand*: any type.
- *Result*: casting of the operand to a rgb color.
- *Return type*: rgb.
- *Special cases*:
  - if the operand is nil, returns white;
  - if the operand is an agent, returns its color if it is visible (otherwise white);
  - if the operand is a string, returns the color denoted by its contents if it is part of the basic colors understood by java.awt.Color; otherwise tries to cast the string to an int and returns this color;
  - if the operand is an int or a float, returns the color denoted by this number (decoding the number using the three R, G, and B integer components);
  - if the operand is a list, and its size is equal or greater than 3, casts the three first elements to integers and interprets them as R, G and B components;
  - if the operand is a boolean, returns black for true and white for false;
  - otherwise casts the operand to an int and returns the associated color;

```
rgb 'white' → white color;  
rgb #FFFFFF → white color;  
rgb [0, 255,0] → green color;  
rgb '255' → blue color;
```

## **rnd**

- *Operand*: an int or a float.
- *Result*: a random integer in the interval [0, operand].
- *Return type*: int.
- *Special cases*: the argument is casted to an int before being evaluated.
- *see also*: flip.
- *Comments* : to obtain a probability between 0 and 1, use the expression  $(rnd\ n) / n$ , where  $n$  is used to indicate the precision;

```
rnd 2 → 0, 1 or 2
(rnd 1000) / 1000 → a float between 0 and 1 with a precision of 0.001
```

## **round**

- *Operand*: an int or a float.
- *Result*: the rounded value of the operand.
- *Return type*: int.
- *Special cases*: the argument is casted to a float before round is evaluated.
- *see also*: int.

```
round 0.51 → 1;
round 100.2 → 100.
```

## **shuffle**

- *Operand*: a list, string or matrix.
- *Result*: the elements of the operand in random order.
- *Return type*: list or string.
- *Special cases*: if the operand is empty, returns an empty list (or string);
- *see also*: reverse.

```
shuffle [12, 13, 14] → [14, 12, 13];
shuffle 'abc' → 'bac'.
```

## **sin**

- *Operand*: an int or a float.
- *Result*: the sinus of the operand (in decimal degrees).
- *Return type*: float.
- *Special cases*: the argument is casted to an int before being evaluated. Integers outside the range "0-359" are normalized.
- *see also*: cos, tan.

```
sin 0 → 0.
```

## species\_of

- *Operand*: an agent expression.
- *Result*: casting of the operand to a species.
- *Return type*: species.
- *Special cases*:
  - if the operand is nil, returns nil;
  - if the operand is an agent, returns its species;
  - if the operand is a string, returns the species with this name (nil if not found);

```
species_of self → the species of the agent;  
species 'world' → the species named 'world';
```

## string

- *Operand*: any type.
- *Result*: casting of the operand to a string.
- *Return type*: string.
- *Special cases*:
  - if the operand is nil, returns 'nil';
  - if the operand is an agent, returns its name;
  - if the operand is a string, returns the operand;
  - if the operand is an int or a float, returns their string representation (as in Java);
  - if the operand is a list, returns its string representation;
  - if the operand is a boolean, returns 'true' or 'false';
  - if the operand is a species, returns its name;
  - if the operand is a color, returns its literal value if it has been created with one (i.e. 'black', 'green', etc.) or the string representation of its hexadecimal value.

```
string (rgb 'black') → 'black';  
string 12.34 → '12.34';  
string world → 'world' if world is a species.
```

## sum

- *Operand*: a list, point or matrix.
- *Result*: the product of all the elements of the operand.
- *Return type*: float.
- *Special cases*: the elements of the operand are casted to float values before sum is evaluated..
- *see also*: mul.

```
sum [12, 10, 3] → 25.0.
```

**tan**

- *Operand*: an int or a float.
- *Result*: the trigonometric tangent of the operand (expressed in decimal degrees).
- *Return type*: float.
- *Special cases*: the argument is casted to an int before being evaluated. Integers outside the range "0-359" are normalized.
- *see also*: sin, cos.

```
tan 180 → 0.
```

**towards**

- *Operand*: an agent or a point.
- *Result*: the direction (in degrees) that the agent must follow to face the operand.
- *Return type*: int.
- *Special cases*: if the agent is not localized, returns 0;
- *see also*: . next\_point\_towards

```
towards {0, 0} → an int between 0 and 359.
```

**Binary operators****=**

- *Left-hand operand*: any expression.
- *Right-hand operand*: any expression.
- *Result*: **true** if both operands are equal, **false** otherwise.
- *Comments*: this operator will return true if the two operands are identical (i.e., the same object) **or** equal. Comparisons between nil values are permitted. Note that floating point and integer values are always considered as different (i.e., 0.0 is not the same as 0).
- *see also*: !=.

```
[1, 2, 3.0] = [1,2,3.0] → true;
int 3.0 = 3 → true;
rgb 'black' = rgb #000000 → true;
float 1 = 1.0 → true;
```

**!=**

same signification as <> operator.

**<>**

- *Left-hand operand*: any expression.
- *Right-hand operand*: any expression.
- *Result*: **true** if both operands are different, **false** otherwise.
- *Comments*: this operator will return false if the two operands are identical (i.e., the same object) **or** equal. Comparisons between nil values are permitted. Note that floating point and integer values are always considered as different (i.e., 0.0 is not the same as 0).
- *see also*: =.

```
[1, 2, 3.0] != [1,2,3.0] → false;
int 3.0 != 3 → false;
rgb 'black' != rgb #000000 → false;
float 1 != 1.0 → false;
```

**>=**

**<=**

- *Left-hand operand*: any expression.
- *Right-hand operand*: any expression.
- *Result*: **true** if the left-hand operand is respectively greater than, less than, greater than or equal to, less than or equal to the right-hand operand. **false** otherwise.
- *Comments*: these operator work with numbers (int and float) as well as strings, for which a lexicographic comparison is performed. **Note on the syntax to use in the XML flavor of GAML**: XML files do not allow the < and > characters to be used within tags. They have then to be replaced by their counterpart in HTML entities: **&lt;**; and **&gt;**.

```
12 > 11.0 → true;
'zzz' > 'abc' → true;
int 12.0 < 12 → false;
```

**@**

same signification as at operator.

**at**

- *Left-hand operand*: a list, string, matrix or point.
- *Right-hand operand*: an integer or a point.
- *Result*: the element of the left-hand operand located at the index specified by the right-hand operand.
- *Comments*:
  - strings, lists, and matrices are zero-based implementations, which means that the first index is 0 and the last one length-1.
  - *at* can be used in the left member of an assignment to assign a new value to one of the positions of a list, a matrix, a string, a point...

```
<let var="my_list" value="[12, 13, 14, 15]" />
<set var="my_list 'at' 2" value="18" /> → my_list is now equal
to [12, 13, 18, 15].
```

- *Special cases*:
  - if the left-hand operand is a string, a list, a point or a one-dimension matrix, and the index is less than 0 or greater than its length, returns nil.
  - if the left-hand operand is a two-dimensions matrix and the index is a point, the same rule applies for both dimensions.
  - if the left-hand operand is a one-dimension matrix, a list, a point or a string, and the index is a point, the x-coordinate of the point is used for the index.
  - if the left-hand operand is a string, returns a one-character string.



```
[1, 10, 3.0] @ 2 → 3.0;
'abcdef' at 0 → 'a';
```

## +

- *Left-hand operand*: a list, string, matrix, point, int or float.
- *Right-hand operand*: a list, string, matrix, point, int or float.
- *Result*: the sum, union or concatenation of the two operands.
- *Special cases*:

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

```
1 + 1 → 2; 1.0 + 1 → 2.0;
```

If the left-hand operand is an int and the right-hand operand is not a number, casts the right-hand operand to an int.

```
1 + '34' → 35;
```

If the left-hand operand is a float and the right-hand operand is not a number, casts the right-hand operand to a float.

```
1.0 + '34' → 35.0;
```

If the left-hand operand is a string, casts the right-hand operand to a string and returns their concatenation.

```
'abc' + 'def' → 'abcdef';
```

If both operands are lists, returns their union.

```
[1, 2, 3] + [4, 5] → [1, 2, 3, 4, 5];
list 'abc' + list 'def' → ['a', 'b', 'c', 'd', 'e', 'f'];
```

If both operands are points, returns their sum.

```
{1, 2} + {4, 5} → {5, 7};
```

If the left-hand operand is a list and the right-hand operand is not a list, returns a new list with the right-hand operand added to the end of the left-hand operand. Matrices, points and strings are not considered as lists and will be added as individual elements.

```
[1, 2, 3] + 4 → [1, 2, 3, 4];
['a', 'b', 'c'] + 'def' → ['a', 'b', 'c', 'def'];
```

- *Comments*: Sum of matrices **is not yet implemented** but should be available sometime in the future with the same syntax.

## -

- *Left-hand operand*: a list, matrix, point, int or float.
- *Right-hand operand*: a list, matrix, point, int or float.
- *Result*: the difference of the two operands.
- *Special cases*:

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

```
1 - 1 → 0; 1.0 - 1 → 0.0;
```

If the left-hand operand is an int and the right-hand operand is not a number, casts the right-hand operand to an int.

```
1 - '34' → -33;
```

If the left-hand operand is a float and the right-hand operand is not a number, casts the right-hand operand to a float.

```
1.0 - '34' → -33.0;
```

If both operands are lists, returns their difference (removes all the elements of the right-hand operand from the left-hand operand).

```
[1, 2, 3] - [3, 4, 5] → [1, 2];  
list 'abc' - list 'abk' → ['c'];
```

If both operands are points, returns their difference.

```
{10, 20} - {4, 5} → {6, 15};
```

If the left-hand operand is a list and the right-hand operand is not a list, returns a new list with the right-hand operand removed from the left-hand operand.

```
[1, 2, 3] - 3 → [1, 2];
```

- *Comments:* Difference of matrices **is not yet implemented** but should be available sometime in the future with the same syntax.

\*

- *Left-hand operand:* an int or float.
- *Right-hand operand:* an int or float.
- *Result:* a float, equal to the product of the two operands.
- *Comments:* Product of matrices **is not yet implemented** but should be available sometime in the future with the same syntax.

/

- *Left-hand operand:* an int or float.
- *Right-hand operand:* an int or float.
- *Result:* a float, equal to the division of the left-hand operand by the right-hand operand.
- *Special cases:* if the right-hand operand is equal to zero, raises no exception and returns the maximum value of float instead.
- *Comments:* Division of matrices **is not yet implemented** but should be available sometime in the future with the same syntax.

//

same signification as div operator.

**div**

- *Left-hand operand:* a int or float.
- *Right-hand operand:* a int or float.
- *Result:* an int, equal to the truncation of the division of the left-hand operand by the right-hand operand.
- *Special cases:* if the right-hand operand is equal to zero, raises no exception and returns the maximum value of int instead.

**%**

same signification as mod operator.

### **mod**

- *Left-hand operand*: a int or float.
- *Right-hand operand*: a int or float.
- *Result*: an int, equal to the remainder of the integer division of the left-hand operand by the righth-hand operand.
- *Special cases*: if the right-hand operand is equal to zero, raises no exception and returns the maximum value of int instead.

**&**

same signification as and operator.

### **and**

- *Left-hand operand*: a boolean expression
- *Right-hand operand*: a boolean expression
- *Result*: a bool value, equal to the logical and between the left-hand operand and the righth-hand operand.
- *Comments*: both operands are always casted to bool before applying the operator. Thus, an expression like *1 and 0* is accepted and returns false.
- *see also*: bool.

**|**

same signification as or operator.

### **or**

- *Left-hand operand*: a boolean expression
- *Right-hand operand*: a boolean expression
- *Result*: a bool value, equal to the logical or between the left-hand operand and the righth-hand operand.
- *Comments*: both operands are always casted to bool before applying the operator. Thus, an expression like *1 or 0* is accepted and returns true.
- *see also*: bool.

**^**

- *Left-hand operand*: a int or float expression
- *Right-hand operand*: a int or float expression
- *Result*: an int or float value, equal the left-hand operand raised to the power of the right-hand operand.
- *Special cases*: if the right-hand operand is equal to 0, returns 1; if it is equal to 1, returns the left-hand operand.

### **accumulate**

- *Left-hand operand*: a list, point or string expression
- *Right-hand operand*: any.
- *Result*: a list containing all elements of left-hand expression and right-hand expression.

```
[1, 2, 3] accumulate [4, 5, 6] → [1, 2, 3, 4, 5, 6]
```

**among**

- *Left-hand operand*: a int expression
- *Right-hand operand*: a list, string, point or matrix.
- *Result*: a list of length the value of the left-hand operand, containing random elements from the right-hand operand.
- *Special cases*: if the right-hand operand is empty, returns an empty list; if the left-hand operand is greater than the length of the right-hand operand, returns the right-hand operand.

```
3 among [1,2,3,4,5,6,7,8] → [3, 6, 7]
```

**collect**

- *Left-hand operand*: a list, matrix, string or point.
- *Right-hand operand*: an arbitrary expression.
- *Result*: a list containing the result of the right-hand expressions applied to each of the elements of the left-hand operand.
- *Comments*: the order of the elements is kept. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4] collect (each + 10) → [11, 12, 13, 14];
'abcd' collect (each > 'a') → [false, true, true, true];
```

**copy\_between**

- *Left-hand operand*: a list or a string
- *Right-hand operand*: a point
- *Result*: a string or a list containing the elements of the left-hand operand between the indexes provided by the x-coordinate (inclusive) and the y-coordinate (exclusive) of the right-hand operand.
- *Comments*: the length of the resulting list or string is (second index - first index).
- *Special cases*: if the first index is less than the second, returns an empty list or string; if the first index is less than 0, it is set to 0; if the second index is greater than the length of the left-hand operand, it is set to this length.

```
[1,2,3,4,5,6,7,8] copy_between {1,3} → [2,3];
'abcdefgh' between {2,5} → 'cde';
```

**count**

- *Left-hand operand*: a list, string, matrix or point
- *Right-hand operand*: a boolean expression
- *Result*: an int, equal to the number of elements of the left-hand operand that make the right-hand operand evaluate to true.
- *Comments*: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] count (each > 3) → 5;
(list species self) count ((first each.location) > 30) → all the
agents of the same species whose x-coordinate is greater than 30.
```

**distance\_to**

- *Left-hand operand*: a point or an agent
- *Right-hand operand*: a point or an agent
- *Result*: a float, equal to the distance (in meters) between the two operands.
- *Comments*: in case a GIS environment is part of the model, the pathfinder is used to compute the distance; otherwise the distance in straight line is returned.
- *Special cases*: if one of the agents is not localized, returns 0.0.

```
((list species self) - self) collect (each distance_to self) → collects
the distances between self and all the agents of the same species.
```

**first\_with**

- *Left-hand operand*: a list, string, matrix or point
- *Right-hand operand*: a boolean expression
- *Result*: any value, equal to the first element of the left-hand operand that make the right-hand operand evaluate to true.
- *Comments*: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] first_with (each > 3) → 4;
(list species self) first_with ((first each.location)> 30) → the
first agent of the same species whose x-coordinate is greater than 30.
```

**in**

- *Left-hand operand*: an expression or a list
- *Right-hand operand*: a list, string, matrix or point
- *Result*: if the left-hand operand is not a list, true if the left-hand operand is equal to one of the elements of the right-hand operand, otherwise false; if it is a list, returns true if all its elements are present in the right-hand operand, otherwise returns false;

```
3 in [1,2,3,4,5,6,7,8] → true;
[3, 10] in [1,2,3,4,5,6,7,8] → false;
'bc' in 'abcded' → true;
['a','b','cd'] in 'abcdef' → true;
```

**index\_of**

- *see*: last\_index\_of.

**last\_index\_of**

- *Left-hand operand*: a list, string, matrix or point
- *Right-hand operand*: an expression
- *Result*: an int, equal to the index of the first (resp. last) occurrence of the right-hand operand in the left-hand operand.
- *Comments*: if the right-hand operand is not present, returns -1.

```
[1,2,3,4,5,6,7,8] index_of 3 → 2;
'abcdefgh' index_of 'bcf' → -1;
```

**max\_of**

- *see:* min\_of.

**min\_of**

- *Left-hand operand:* a list, string, matrix or point
- *Right-hand operand:* an int or float expression
- *Result:* an int or float, equal to the maximum (resp. minimum) value of the right-hand expression evaluated on each of the elements of the left-hand operand
- *Comments:* in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,10,4,7,6,7,8] max_of (each * 100) → 1000;
(list species self) min_of (first each.location) → the smallest
x-coordinate of the agents of the same species as self.
```

**neighbours\_at**

- *Left-hand operand:* an agent
- *Right-hand operand:* an int or float (expressing a distance in meters)
- *Result:* a list, containing all the agents located in the circle of radius equal to the right-hand operand, and center the location of the left-hand operand.
- *Comments:* if the left-hand operand is a grid cell, returns only grid cells of the same environment; if it is a regular agent, returns all the agents, whatever their species, save for grid cells.
- *Special cases:* if the left-hand operand is a not localized agent, returns an empty list.

```
(self neighbours_at (10)) of_species (species self) → all the agents of
the same species situated at a distance greater or equal to 10 of
self.
```

**of\_species**

- *Left-hand operand:* a list (of agents)
- *Right-hand operand:* a species expression
- *Result:* a list, containing the agents of the left-hand operand whose species is that denoted by the right-hand operand.
- *Comments:* in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements. The expression *agents of\_species (species self)* is equivalent to *agents where (species each = species self)*; however, the advantage of using the first syntax is that the resulting list is correctly typed with the right species, whereas, in the second syntax, the parser cannot determine the species of the agents within the list (resulting in the need to cast it explicitly if it is to be used in an ask statement, for instance).

```
(self neighbours_at 10) of_species (species self) → all the
neighbouring agents of the same species.
```

**sort\_by**

- *Left-hand operand*: a list, string, point, or matrix
- *Right-hand operand*: an int, float or string expression.
- *Result*: a list, containing the agents of the left-hand operand sorted in ascending order by the value of the right-hand operand when it is evaluated on them.
- *Comments*: the left-hand operand is casted to a list before applying the operator. Therefore, a species value can be used directly to represent all the agents of this species. In the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
(self neighbours_at 10) sort_by (first each.location) → all the
neighbouring agents, sorted by their x-coordinate.
```

**starts\_with**

- *Left-hand operand*: a string
- *Right-hand operand*: a string
- *Result*: true is the left-hand operand starts with the right-end operand.
- *Comments*: equivalent to  $(\text{left-hand\_operand} [\text{\#index\_of\_index\_of}] \text{right-hand\_operand}) = 0$ , but faster.

**split\_using**

- *see*: tokenize.

**tokenize**

- *Left-hand operand*: a string
- *Right-hand operand*: a string (delimiters)
- *Result*: a list, containing the sub-strings (tokens) of the left-hand operand delimited by each of the characters of the right-hand operand.
- *Comments*: delimiters themselves are excluded from the resulting list.

```
'to be or not to be, that is the question' split_using ' ,' →
['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question'].
'to be or not to be, that is the question' tokenize ' ,' →
['to', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question'].
```

**where**

- *see*: select.

**select**

- *Left-hand operand*: a list, string, matrix or point
- *Right-hand operand*: a boolean expression
- *Result*: a list containing all the elements of the left-hand operand that make the right-hand operand evaluate to true.
- *Comments*: in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
[1,2,3,4,5,6,7,8] select (each > 3) → [4, 5, 6, 7, 8];
(list species self) where ((first each.location)> 30) → all the
agents of the same species whose x-coordinate is greater than 30;
'abcdef' select (each < 'd') → ['a', 'b', 'c'].
```

**with\_max\_of**

- *see:* with\_min\_of.

**with\_min\_of**

- *Left-hand operand:* a list, string, matrix or point
- *Right-hand operand:* an int, float or string expression
- *Result:* a list containing the elements of the left-hand operand that maximize (resp. minimize) the value of the right-hand operand.
- *Comments:* in the right-hand operand, the keyword each can be used to represent, in turn, each of the elements.

```
(list species self) with_max_of (each.var) → all the agents of the
same species whose variable var is equal to the current maximum value
of var (equivalent to (list species self) where (each.var = max (list
species self collect (each.var))) -- only faster and easier to write);
['abc', 'bcde', 'fgh', 'fffde'] with_min_of (each.length) →
['abc', 'fgh'].
```

**with\_precision**

- *Left-hand operand:* an int or float.
- *Right-hand operand:* an int.
- *Result:* round off the value of left-hand operand to the precision given by the value of right-hand operand.

```
12345.78943 with_precision 2 → 12345.79
123 with_precision 2 → 123.00
```

**Ternary operators****? :**

- *Left-hand operand:* a boolean expression
- *Middle operand:* an expression
- *Right-hand operand:* an expression
- *Result:* if the left-hand operand evaluates to true, returns the value of the middle operand, otherwise that of the right-hand operand

```
[10, 19, 43, 12, 7, 22] collect ((each > 20) ? 'above' : 'below') →
['below', 'below', 'above', 'below', 'below', 'above']
```

These functional tests can be combined together (here, the *color* variable takes three values with respect to the value of *food*, above 5, between 2 and 5, and below 2): `<set var="color" value="rgb ((food > 5) ? 'red' : ((food > 2)? 'blue' : 'green'))"/>`



# Keywords

---

## constants

### nil

Represents the null value (or undefined value). It is the default value of variables of type agent, point or species when they are not initialized.

### true, false

Represent the two possible values of boolean variables or expressions. (see bool).

## pseudo-variables

Pseudo-variables are special variables whose value cannot be changed by agents (but can change depending on the context of execution).

### self

*self* is a pseudo-variable (can be read, but not written) that always holds a reference to the executing agent.

- Example (sets the *friend* field of another random agent to self and reversely) :

```
<let var="potential_friend" value="(one_of species ''self'' ) -
''self'')" />
<if condition="potential_friend != nil">
  <set var="potential_friend.friend" value=""self''"/>
  <set var="friend" value="potential_friend" />
</if>
```

### myself

*myself* is the same as *self*, except that it needs to be used instead of *self* in the definition of remotely-executed code (ask and lcreate commands). *myself* represents the sender agent when the code is executed by the target agent.

- Example (asks the first agent of my species to set its color to my color) :

```
<ask target="first (list species self)">
  <set var="color" value=""myself''.color"/>
</ask>
```

- Example (create 10 new agents of my species, share my energy between them, turn them towards me, and make them move 4 times to get closer to me) :

```
<create species="species self" number="10">
  <set var="energy" value=""myself''.energy / 10"/>
  <loop times="4">
    <set var="heading" value="towards ''myself''" />

  </loop>
</create>
```

## context

*context* is a pseudo-variable that represents the "environment" of the agent. Using this keyword prevents from explicitly having to access the individual cell(s) or environments on which it is located. The caveat is that an expression like *context.var* will look for *var* in all the environments defined, which can be costly.

## each

*each* is a pseudo-variable only used in filter expressions following operators such as *where* or *first\_with*. It represents, in turn, each of the elements of the target datatype (a list, string, point or matrix, usually).

## units

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

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

So, an expression like

```
<float name="foo" value="1"/>
```

will be considered as *1 meter* if *foo* is a distance, or *1 second* if it is a duration, or *1 meter/second* if it is a speed. If one wants to specify the unit, it can be done very simply by adding the unit name *after* the numeric value, like:

```
<float name="foo" value="1 centimeter"/>
```

In that case, the numeric value of *foo* will be automatically translated to 0.01 (meter). It is recommended to always use *float* as the type of the variables that might be qualified by units (otherwise, for example in the previous case, they might be truncated to 0). Several units names are allowed as qualifiers of numeric variables. As they can be used in expressions directly, they are considered as reserved keywords (and therefore cannot be used for naming variables and species). Their complete list is:

### length

meter (default), meters, m, centimeters, centimeter, cm, millimeter, millimeters, mm, decimeter, decimeter, dm, kilometer, kilometers, km, mile, miles, yard, yards, inch, inches, foot, feet, ft.

### time

second (default), seconds, sec, s, minute, minutes, mn, hour, hours, h, day, days, d, month, months, year, years, y, millisecond, milliseconds, msec.

### mass

kilogram (default), kilogram, kilo, kg, ton, tons, t, gram, grams, g, ounce, ounces, oz, pound, pounds, lb, lbm.

### surface

square\_meter (default), m2, square\_meters, square\_mile, square\_miles, sqmi, square\_foot, square\_feet, sqft.

### volume

m3 (default), liter, liters, l, centiliter, centiliters, cl, deciliter, deciliters, dl, hectoliter, hectoliters, hl.

These represent the basic metric and US units. Composed and derived units (like velocity, acceleration, special volumes or surfaces) can be obtained by combining these units using the *\** and */* operators. For instance:

```
<float name="one_kmh" init="1 km / h" const ="true"/>
<float name="one_microsecond" init="1 sec / 1000"/>
```

```
<float name="one_cubic_inch" init="1 sqin * 1 inch" />
```

... etc ...

## Skills

---

### Overview

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 name="my_species" skills="skill1, skill2" />
  ...
</species>
```

Skills have been designed to be mutually compatible so that any combination of them will result in a functional species. The list of available skills in GAMA is:

- *visible*: for agents that need to appear graphically on the user interface.
- *situated*: for agents that are situated in (at least) one environment.
- *moving*: for agents that need to move.
- *carrying*: for agents that need to carry other agents.
- *communicating*: for agents that need to communicate using the FIPA <sup>[1]</sup> protocols.
- *exploring*: for agents that need to explore a region of space.

So, for instance, if a species is declared as:

```
<species name="foo" skills="visible, moving">
```

its agents will automatically be provided with the following variables : *shape, color, size, hidden, information, range, location, neighbours (r/o), speed, heading, destination (r/o)*, and the following actions: *display\_range, show, hide, display\_information, display\_pointer, move, goto, wander* 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 name="speed" max="100" min="0"/>
```

Or, to obtain ever growing agents:

```
<float name="size" max="100" value="size + 0.001"/>
```

Or, to change their shape in a behavior:

```
<if condition="shape = 'circle'">
  <set var="shape" value="'square'"/>
</if>
```

## visible

This skill implicitly adds the situated skill to the species.

## variables

### shape

string, defines the shape of the agent. Built-in shapes are: 'square', 'circle', 'dot', 'line', and 'rounded'. Can alternatively contain the path to an icon (JPEG, PNG, GIF).

### color

rgb, defines the color of the agent. Only used if *shape* is a built-in shape.

### hidden

bool, indicates if the agent is hidden or not. A hidden agent is not drawn.

### information

string, a text that can be displayed on the interface.

## actions

### display\_range

tells the agent to display a circle centered on it. Its radius is the value of *range*.

→ **color**: rgb, optional, the color to use to display the circle

### display\_range,

```
<arg name="color" value="rgb 'red'"/>
```

### hide\_range

hides the range circle.

### hide\_range

### display\_information

tells the agent to display the text contained in *information*.

→ **color**: rgb, optional, the color to use to display the information.

### display\_information,

```
<arg name="color" value="rgb 'yellow'"/>
```

### hide\_information

hides the information

### hide\_information

### hide

tell the agent to hide itself.

### hide

### show

tell the agent to show itself.

### show

**display\_pointer**

displays a line between the agent and the target of this action.

→ **target**: agent, mandatory, the agent to point to.

→ **color**: rgb, optional, the color to use to display the information.

**display\_pointer,**

```
<arg name="target" value="one_of list (species self)"/>
```

**hide\_pointer**

tells the agent to hide the pointer line.

**hide\_pointer**

---

**situated****variables****range**

float, the distance at which agents can *see* other agents.

**location**

point, their position in the environment (see the coordinates system of the environment).

**neighbours**

list, read-only, a continuously updated list of their neighbours, with respect to the value of *range*.

**size**

float, their size (i.e. the diameter of the circle in which they are contained).

---

**moving**

This skill implicitly adds the situated skill to the species.

**variables****speed**

float, the speed of the agent, in meter/second.

**heading**

int, the absolute heading of the agent in degrees (in the range 0-359).

**destination**

point, read-only, continuously updated destination of the agent with respect to its *speed* and *heading*.

---

## 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.

→ **speed**: float, optional, the speed to use for this move (replaces the current value of *speed*).

→ **heading**: int, optional, the direction to take for this move (replaces the current value of *heading*).

→ **distance**: float, optional, the distance at which to move (is checked against the *speed* of the agent for compatibility, may change *speed* if necessary).

← **return**: point, the new location of the agent (or nil if an error occurred).

```
,
  <arg name="speed" value="speed - 10"/>
  <arg name="heading" value="heading + rnd 30"/>
  <arg name="distance" value="100" />
```

### 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*.

→ **speed**: float, optional, the speed to use for this move (replaces the current value of *speed*).

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

→ **distance**: float, optional, the distance at which to move (is checked against the *speed* of the agent for compatibility, may change *speed* if necessary).

← **return**: point, the new location of the agent (or nil if an error occurred).

```
,
  <arg name="speed" value="speed - 10"/>
  <arg name="amplitude" value="120"/>
  <arg name="distance" value="100" />
```

### goto

moves the agent towards the *target* passed in the arguments.

→ **target**: point or agent, mandatory, the location or entity towards which to move.

→ **speed**: float, optional, the speed to use for this move (replaces the current value of *speed*).

← **return**: point, the new location of the agent (or nil if an error occurred).

```
,
  <arg name="target" value="one_of (list species self)"/>
  <arg name="speed" value="speed * 2"/>
```

---

## carrying

### variables

#### capacity

int, the maximum number of agents that can be carried

#### contents

list, read-only, the current list of agents carried.

### actions

#### drop

makes the agent leave, at its location, all or some of the agents it was carrying.

→ **agents**: list, optional, the list of agents to drop. If not specified, all the agents carried are dropped.

← **return**: list, the list of agents actually dropped.

```
,
  <arg name="agents" value="contents of_species foo"/>
```

#### load

makes the agent pick & carry all or some of the agents specified by the arguments.

→ **agents**: list, optional, the list of agents to load. If not specified, the action is ignored.

→ **number**: int, optional, the number of agents to load from the list. If not specified, all the agents are loaded until the *capacity* is reached.

← **return**: list, the list of agents actually loaded.

```
,
  <arg name="agents" value="neighbours of_species foo"/>
  <arg name="number" value="10"/>
```

---

## communicating

This skill implements the major FIPA communication and negotiation protocols <sup>[2]</sup> and enable the agents to communicate using them. It brings along two new datatypes : message and conversation. GAMA implements the following interaction protocols of FIPA : FIPA Brokering <sup>[3]</sup>, FIPA Contract Net <sup>[4]</sup>, FIPA Propose <sup>[5]</sup>, FIPA Query <sup>[6]</sup>, FIPA Request <sup>[7]</sup>, FIPA Request When <sup>[8]</sup>, FIPA Subscribe <sup>[9]</sup>. Besides the standard interaction protocols of FIPA <sup>[1]</sup>, GAMA support a special interaction protocol named "no-protocol".

---

## variables

### conversations

list, read-only, the conversation in which the agent is currently engaged.

### messages

list, read-only, the messages received by the agent.

**accept\_proposals, agrees, cancels, cfps, failures, informs, proposes, queries, refuses, reject\_proposals, requests, requestWhens, subscribes**

list, read-only, the messages received by the agent, broken down by performative.

## actions

### send

allows the agent to send an existing message, or to create one and send it immediately.

Either use the argument:

→ **message:** message, optional, the message to send.

Or a combination of the following arguments:

→ **receivers:** list, mandatory, the list of agents to send the message to.

→ **content:** list, mandatory, the content of the message. Can be anything.

→ **performative:** string, mandatory. The performative of the message (see this page)

→ **protocol:** string, mandatory, the protocol followed by this message (see this page)

→ **conversation:** conversation, optional, the conversation in which the message should be inserted. If nil, creates a new conversation.

← **return:** message, the message actually sent or nil if an error has occurred.

```
<create species="message" number="1" return="mes">
  <set name="receivers" value="neighbours" />
  <set name="content" value="['Hello !']" />
  <set name="protocol" value="'no-protocol'" />
  <set name="performative" value="'inform'" />
</create>
'
  <arg name="message" value="mes"/>
```

```
'
  <arg name="receivers" value="neighbours" />
  <arg name="content" value="['Hello !']" />
  <arg name="protocol" value="'no-protocol'" />
  <arg name="performative" value="'inform'" />
```

### end, failure, inform, propose, query, refuse, reject-proposal, request, subscribe

replies a list of messages with the corresponding performatives "end", "failure", "inform", "propose", "query", "refuse", "reject-proposal", "request", "subscribe". The replied messages will be filled automatically with the corresponding performatives.

→ **message:** list, mandatory, messages to reply to.

→ **content:** list, optional, the content of the replied messages.



```
<list name="noProtocolInformMsgs" value="(informs) where ((species
(each.sender) = foodCarrier) and (each.protocol = 'no-protocol'))"/>
,
  <arg name="message" value="noProtocolInformMsgs" />
```

## data types

### message

An instance of message represents a piece of information exchanged between agents. It has the following attributes :

- sender : agent, the sender of the message.
- receivers : a list of agents, contains all the receivers of the message.
- performative : string, the performative of the corresponding message. For more details concerning the performative, please refer to the FIPA Communicative Act Library Specification. <sup>[10]</sup>
- content : list, contains the content of the corresponding message. The modeller can put whatever content in this list.

### conversation

An instance of conversation represents a FIPA interaction protocol. It helps ensure that a conversation between agents respects the specification of a corresponding protocol defined by FIPA. It has the following attributes :

- messages : list, the currently unread messages of the conversation.
- protocol : string, the name of the interaction protocol that this conversation respects.
- initiator : agent, the agent which initiates the conversation/the interaction protocol.
- participants : a list of agents, all the participants of the conversation (except for the initiator).
- ended : bool, helps determine if the corresponding conversation/interaction protocol has already finished or not yet.

### Notes

GAMA supports a special interaction protocol named "no-protocol". If the modeller finds that no supported FIPA Interaction Protocols fits his modelling case, he can use the "no-protocol". Upon declaring a conversation following this protocol, the modeller is free to send messages having whatever performative between agents. But it 's upto the modeller to finish up the corresponding conversation by using the "end" primitive.

## exploring

This skill implements the patrolling capability of agents, enables the agents to go on patrol in a list of given points.

### variables

#### points

list, the list of points to patrol. This list must not be empty before the agents can begin its patrolling activity.

#### patrolling

bool, determines if the corresponding agent is going on patrol or not.

## actions

### patrol

do the patrolling activity if the list of points is not empty.

→ **speed**: float, optional, the speed of the corresponding agent.

```
<task name="patrol_the_terrain" while="patrolTheTerrain">
  <priority if="patrolTheTerrain" value="10" else="0"/>
  <duration max="2 hours"/>
  <repeat action="patrol">
    <arg name="speed" value="maxSpeed"/>
  </repeat>
</task>
```

## References

- [1] <http://www.fipa.org/>
- [2] <http://www.fipa.org/repository/standardspecs.html>
- [3] <http://www.fipa.org/specs/fipa00033/index.html>
- [4] <http://www.fipa.org/specs/fipa00029/index.html>
- [5] <http://www.fipa.org/specs/fipa00036/index.html>
- [6] <http://www.fipa.org/specs/fipa00027/index.html>
- [7] <http://www.fipa.org/specs/fipa00026/index.html>
- [8] <http://www.fipa.org/specs/fipa00028/index.html>
- [9] <http://www.fipa.org/specs/fipa00035/index.html>
- [10] <http://www.fipa.org/specs/fipa00037/SC00037J.html>

# Built-in

---

## Introduction

Species, when declared in a model, inherit some of their attributes (variables, actions) from the Java class that back them behind the scene. For "regular" agents, the number of these attributes is voluntarily limited. In a sense, that class acts exactly like a basic → skill, and gives the agents a (very) limited number of common capabilities. However, some agents in the model are based on more specialized Java classes: this is the case of the *world*, which provides some critical global knowledge and functions to the modeller and the agents. This is also the case of the *environmental agents* defined by grids. In this section, we start by examining the built-in variables and actions common to all the agents. We then describe the global ones defined on the *world*. For information about the ones defined on grids, follow this link.

## Built-in variables

### name

string. Each agent has a default name (concatenation of its species name and unique index), which can be changed at will to something more useful for the modeler (if needed).

### Note

other "built-in" read-only attributes can be accessed through special → operators:

- species or species\_of, which return the species of the agent,
- int, which returns its index;

or → keywords:

- self and myself, which return the agent itself.

## Built-in actions

Three built-in actions are provided to the agents.

### debug

makes the agent output an arbitrary message in the console. The message is automatically prefixed with the cycle of the simulation and followed by a carriage return.

→ **message**: string, mandatory, the message to display.

```
'
  <arg name="message" value="'This is a message from ' +
name"/>
```

### inform

makes the agent output an arbitrary information message in the console. The message is not followed by a carriage return.

→ **message**: string, mandatory, the information 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.

```
'
  <arg name="message" value="'This is an information from ' +
name + "\n"/>
```

### error

makes the agent output an error dialog (if the simulation contains a user interface). Otherwise displays the error in the console.

→ **message**: string, mandatory, the message to display.

```
'
  <arg name="message" value="'This is an error raised by ' +
name"/>
```

### tell

makes the agent output a dialog (if the simulation contains a user interface). The simulation goes on, but its interface is not accessible until the dialog is closed (use with caution, as it may prevent the user from accessing the interface).

→ **message**: string, mandatory, the message to display.

```
'
    <arg name="message" value="'This is a message dialog raised by
' + name"/>
```

## Global built-in variables

Global variables can be accessed by the world and **every other agent** in the model.

### time

int, read-only, represents the current simulated time in seconds (the default unit). Begins at zero. Although *time* is a read-only variable, it is possible to control its maximum value by redeclaring it. When the maximum is reached during a simulation, this simulation is automatically stopped.

```
<global>
...
    <int name="time" max="1000"/>
...
</global>
```

### step

int, represents the time step between two executions of the set of agents, in seconds. Its default value is 1. 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*.

```
<global>
...
    <int name="step" value="10"/>
...
</global>
```

### seed

float, represents the seed used in the computation of random numbers. Keeping the same seed between two runs of the same model ensures that the sequence of events will remain the same, which can be useful when debugging a model. Declaring it as a parameter allows the user or an external process (batch, for instance) to modify it.

```
<global>
...
    <int name="seed" value="354.0" parameter="true"/>
...
</global>
```

### places

list, read-only, returns a list of all the "places" of the model (i.e. all the cells of the different grids that have been defined).

```
<species name="foo" skills="moving, visible">
    <agent name="goal" value="one_of places" />
</species>
```

### 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 target = "agent of_species my_species">
...
</ask>
```

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

```
<ask target="list my_species">
...
</ask>
```

## Global built-in actions

### halt

stops the simulation.

```
<global>
...
<reflex when="length agents <= 0">

</reflex>
</global>
```

### pause

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

```
<global>
...
<reflex when="time = 100">

</reflex>
</global>
</do>
```

### diffuse

diffuses the value of a variable (normally of type float) in a grid environment. Diffusion here means that every cell of the grid distributes to its neighbors a proportion of the amount of the variable's value.

→ **var**: string, mandatory. The name of the variable to diffuse.

→ **proportion**: float, mandatory. The proportion to diffuse, between 0 and 1. This proportion is divided by the number of neighbors of the cell in order to obtain the amount distributed to each cell.

→ **environment**: string, mandatory (optional if only one grid environment is defined). The environment in which to diffuse the variable.

```
<global>
...
<reflex>
```

```
</reflex>
```

```
</global>
```

# Species

---

## Introduction

The agents' species are defined in the entities section. A model can contain any number of species. Species are used to specify the structure and behaviors of agents. Although the definitions below apply to all the species, some of them require specific declarations: the species of the world and the species of the environmental places.

## Species declaration

The simplest way to declare a species is the following:

```
<species name="a_name">
  [variable declarations]
  [action declarations]
  [behaviors]
</species>
```

for example:

```
<species name="foo"/>
```

The agents that will belong to this species will only be provided with some → built-in attributes and actions, a basic → behavioral structure and nothing more. So, for instance, it is possible (somewhere else in the model) to write something like:

```
<create species="foo" number="10" return="foo_agents"/>
<ask target="foo_agents">
  ,
  <arg name="message" value="'my name is:' + name"/>
</ask>
```

Which will result in the 10 agents writing their name, in turn, on the console. If the species declare → variables, the structure of the agents is modified consequently. For instance:

```
<species name="foo">
  <float name="energy" init="rnd 100" min="0" max="100"
value="energy - 0.001" />
</species>
```

Will give each agent an amount of *energy* (between 0 and 100), which will decrease over time until it reaches 0. The species can also declare actions that will supplement the built-in ones and extends the possibilities of the agents. Here, we provide two possible actions for agents of species *foo*, eating and stealing *energy*:

```
<species name="foo">
  <float name="energy" init="rnd 100" min="0" max="100"
value="energy - 0.001" />
```

```

    <action name="eat">
      <set var="energy" value="energy + rnd 2"/>
    </action>
    <action name="steal">
      <let name="another_agent" value="any (list foo - self)"/>
      <if condition="(another_agent != nil) and
((another_agent.energy) > 0)">
        <set var="another_agent.energy"
value="another_agent.energy - 0.01"/>
        <set var="energy" value="energy + 0.01"/>
      </if>
    </action>
  </species>

```

Of course, these actions do nothing unless they are called either by  $\rightarrow$  behaviors or by other agents. One might for example extend the previous example like:

```

<create species="foo" number="1000" return="foo_agents"/>
<ask target="100 among foo">

</ask>
<ask target="100 among foo">

</ask>

```

In this example, we create 1000 *foos*, ask 100 of them to *eat*, and another 100 of them to *steal* energy. If these commands are done repetitively (for example, every turn in the world), they will result in a somewhat complex dynamic distribution of the energy between the *foos*.

Of course, the dynamics of *foos* can also be declared from within their species. If we change slightly the declaration of *foo* like this:

```

<species name="foo">
  <float name="energy" init="rnd 100" min="0" max="100"
value="energy - 0.001" />
  <action name="eat">
    <set var="energy" value="energy + rnd 2"/>
  </action>
  <action name="steal">
    <let name="another_agent" value="any (list foo - self)"/>
    <if condition="(another_agent != nil) and
((another_agent.energy) > 0)">
      <set var="another_agent.energy"
value="another_agent.energy - 0.01"/>
      <set var="energy" value="energy + 0.01"/>
    </if>
  </action>
  <reflex>
    <if condition="flip 0.1">

```

```

        </if>
        <if condition="flip 0.1">

        </if>
    </reflex>
</species>

```

We obtain agents that execute the reflex every turn and decide independently to eat or steal energy. Once they are created using

```
<create species="foo" number="1000"/>
```

they behave by their own.

## skills: behavioral plug-ins

Basic agents like the previous ones cannot, however, do many things. That's what → skills are for. Example:

```

<species name="foo" skills="situated">
    ...
</species>

```

makes *foos* benefit from a set of variables and behaviors declared by the *situated* skill. Skills are like plug-ins written in Java and can provide a lot of new functionality to the agents.

## parent: inheritance of species

A species can be declared as a child of another species, using the *parent* property. For instance :

```

<species name="foo" skills="situated" parent="bar">
    ...
</species>

```

will make *foo* "inherit" from the definition of *bar*. What does "inherit" precisely mean in this context ?

- → skills declared in *bar*, together with their → built-in attributes and actions, are copied to *foo* and added to the possible new skills defined in *foo*.
- → variables declared in *bar*, are identically copied to *foo* unless a variable with the same name is defined in *foo*, in which case this redefinition is kept. This also applies to built-in variables. The type of the variable can be changed in this process as well (but be careful when doing it, since inherited behaviors can rely on the previous type).
- actions declared in *bar* are identically copied to *foo* unless an action with the same name is defined in *foo*, in which case this redefinition is kept.
- reflexes declared in *bar* are identically copied to *foo* unless a reflex with the same name is defined in *foo*, in which case this redefinition is kept. Unnamed reflexes from both species are kept in the definition of *foo*.
- → behaviors declared in *bar* are identically copied to *foo* unless a behavior of the same type with the same name is defined in *foo*, in which case this redefinition is kept.
- inits are treated differently : each of the init reflexes defined in *bar* and *foo* are kept in *foo* and they are executed in the order of inheritance (ie. *bar* 's one first, then *foo* 's one).



## control: behavioral architecture

By default, species are created with a minimal behavioral architecture : they only allow the definition of reflexes as a way to define the agents' behaviors. As reflex-based agents are somewhat limited when it comes to maintaining a state between two steps or enabling the selection of behaviors, GAML provides the modeler with two possible behavioral architectures, EMF (for Etho-Modeling Framework) and FSM (Finite State Machines). Each of them gives the possibility to define new elements in addition to reflexes : respectively tasks and states.

## base: Java foundation

The corresponding class used to initialize agent. An advance feature of the GAMA platform allowing the third party developer to develop their own agent architecture using the Java programming language.

# Sections

---

The XML files that compose a model are structured in several sections.

## include

This section allows to load another model file before loading the current file. This file can contain anything (whole definition of a model, definition of a species, of an environment, of global variables, etc.) as long as it respects the common structure of GAML models.

## global

This "global" section defines the "world" agent, a special agent of a GAMA model. We can define variables and behaviours for the "world" agent. Variables of "world" agent are global variables thus can be referred by agents of other species or other places in the model source code.

*Exemple:*

```
<global>
  <int name="time" init="0" max="10000" />
  <int name="step" init="1s" const="true" />
  <float name="seed" init="675" const="true" />
  <bool name="use_icons" init="false" parameter="true"/>
  <bool name="display_state" init="true" parameter="true"/>
  <float name="evaporation_rate" init="0.1" min="0" max="1"
parameter="true" />
  <float name="diffusion_rate" init="0.5" min="0" max="1"
parameter="true" />
  <bool name="fast_draw" init="false" parameter="true" />
  <int name="gridsize" init="75" const="true"/>
  <int name="ants_number" init="gridsize + 25" parameter="true"
min="1" max="2000"/>
  <point name="center" init="{int (gridsize / 2), int (gridsize /
2)}" const="true" />
  <string name="file" init="'../images/environment' + (string
gridsize) + 'x' + (string gridsize) + '.pgm'" const="true" />
  <ant_grid name="nest_place" />
```

```

<rgb name="black" init="rgb 'black'" const="true" />
<rgb name="blue" init="rgb 'blue'" const="true" />
<rgb name="green" init="rgb 'green'" const="true" />
<rgb name="white" init="rgb 'white'" const="true" />
<rgb name="FF00FF" init="rgb 'gray'" const="true" />
<rgb name="C00CC0" init="rgb '#00CC00'" const="true" />
<rgb name="C00990" init="rgb '#009900'" const="true" />
<rgb name="C00550" init="rgb '#005500'" const="true" />
<rgb name="yellow" init="rgb 'yellow'" const="true" />
<rgb name="red" init="rgb 'red'" const="true" />
<rgb name="orange" init="rgb 'orange'" const="true" />
<string name="ant_shape_empty" init="use_icons?
'../icons/ant.png': 'dot' " const="true" />
  <string name="ant_shape_full" init="use_icons ?
'../icons/full_ant.png' : 'circle' " const="true" />

<reflex>
  <ask target="places" as="ant_grid">
    <if condition="true">
      <set var="color" value="green"/>
    </if>
  /ask
</reflex>

<reflex>
  <do action="disffuse">
    <string name="var" value="'road'"/>
    <float name="proportion" value="disffuse_rate"/>
    <string name="environment" value="'ant_grid'"/>
  </do>
</reflex>
</global>

```

## entities

Definitions of species are placed int this section.

*Example:*

```

<entities>
  <-> species name="ant" skills="moving, visible" control="fsm">
    <rgb name="color" init="'orange'" const="true" />
    <ant_grid name="place" value="ant_grid location" />
    <string name="shape" init="ant_shape_empty" />
    <string name="information" value="state" />
    <float name="speed" init="1m/s" const="true" />
    <float name="size" init="1.0" const="true" />
    <bool name="hasFood" init="false" />

```

```

<action name="pick">
  <set var="shape" value="ant_shape_full" />
  <set var="hasFood" value="true" />
  <set var="place.food" value="place.food - 1" />
</action>

<action name="drop">
  <set var="hasFood" value="false" />
  <set var="shape" value="ant_shape_empty" />
  <set var="heading" value="heading - 180" />
</action>

<action name="choose_best_place">
  <let var="list_places" value="place.neighbours
of_species ant_grid" />
  <if condition="(list_places count (each.food > 0))
> 0 ">
    <return value="point (list_places first_with
(each.food > 0))" />
    <else>
      <let var="min_nest" value=" (list_places
min_of (each.nest))" />
      <set var="list_places" value="list_places
sort ((each.nest = min_nest) ? each.road : 0.0)" />
      <return value="point last list_places" />
    </else>
  </if >
</action>
<state name="init" initial="true">
  <if condition="display_state">
    <do action="display_information">
      <arg name="color" value="yellow"/>
    </do>
  </if>
  <transition to="wandering" when="true"/>
</state>

<state name="wandering">
  <do action="wander">
    <int name="amplitude" value="120" />
  </do>
  <transition to="carryingFood" when="place.food >
0">
    <do action="pick"/>
  </transition>
  <transition to="followingRoad" when="place.road >
0.05"/>

```

```

</state>

<state name="carryingFood">
  <set var="place.road" value="place.road+120" />
  <do action="goto">
    <ant_grid name="target" value="nest_place" />
  </do>
  <transition to="wandering" when="place.isNestLocation"
>
    <do action="drop" />
  </transition>
</state>

<statename="followingRoad">
  <do action="choose_best_place" return="next_place"/>
  <set var="location" value="next_place"/>
  <transition to="carryingFood" when="place.food &gt;
0">
    <do action="pick"/>
  </transition>
  <transition to="wandering" when="(place.road &lt;
0.05)" />
</state>
<→ /species>
</entities>

```

## environment

This section contains definitions of environment. Actually, GAMA supports two kinds of environment: GRID based environment and GIS based environment. The GRAPH based environment is still under testing. A model can have several GRID environments and GIS environments. GAMA platform is responsible for assuring the all the necessary synchronizations between these environments. *Example:*

```

<environment width="gridsize" height="gridsize">
  <grid name="ant_grid" type="'square'" width="gridsize"
height="gridsize" neighbours="8">
    <bool name="multiagent" value="true" />
    <int name="type" from="file" const="true" />
    <bool name="isNestLocation" init="(self distance_to
(translate center)) &lt; 4" const="true" />
    <bool name="isFoodLocation" init="type = 2" const="true"
/>
    <float name="added_road" init="0" min="0" />
    <rgb name="color" value="isNestLocation ? FF00FF:(food
&gt; 0)? blue : ((road &lt; 0.001)? black : ((road &gt; 2)?
white : ((road &gt; 0.5)? (C00CC0) : ((road &gt; 0.2)?
(C009900) : (C005500))))))" />
    <int name="food" init="isFoodLocation ? 5 : 0" />

```

```

    <float name="road" init="0" min="0" max="300" value="(road
    &gt; 0.1) ? road * (1 - evaporation_rate) : 0" />
    <int name="nest" init="300 - (self distance_to (translate
    center))" />
    <init if=" location = translate center">
        <set var="nest_place" value="self" />
        <create species="ant" number="ants_number">
            <set var="location" value="myself.location" />
        </create>
    </init>
</grid>
</environment>

```

For a description of the attributes of environment, please have a look at this section.

## output

This section defines outputs to display in the → simulation mode of GAMA. Several kinds of output are supported.

- **display:** defines views to display environments (defined in the "environment" section). There are two types of display: Grid display and Gis display.
  - *Grid display:* defines a view to display the Grid environment declared in the environment section.
  - *Gis display:* defines a view to display the Gis environment declared in the environment section.
- **chart:** defines a view to display a chart. There are three types of chart: pie, series and histogram.
  - *Pie:* defines a view to display the pie-based chart.
  - *Series:* defines a view to display the series-based chart.
  - *Histogram:* defines a view to display a histogram-based chart.
- **inspect:** inspectors the species and agents defined in the model.
  - *agent inspect:* user clicks on the Grid display or Gis display to select an agent. This view will display all the attributes of the selected agents as well as the change of these attributes over the simulation.
  - *species inspect:* this inspector displays all the species, the corresponding agents and attributes dedined in the model in a tree-based structure.
- **monitor:** a monitor is used to observe the change in value of an expression over the simulation.

*Exemple:*

```

<output>
    <inspect name="Agents" type="agent" refresh_every="1"/>
    <inspect name="Species" type="species" refresh_every="1"/>
    <chart type="pie" name="Carrying Food" background="rgb
    'lightGray'" refresh_every="5" style="exploded">
        <data name="Carrying Food" value=" (list ant) count
    (each.state = 'carryingFood') " />
        <data name="Not Carrying Food" value=" (list ant) count
    (each.state != 'carryingFood') " />
    </chart>
    <chart type="histogram" name="Carrying Food" background="rgb
    'lightGray'" refresh_every="5" style="3d">
        <data name="Carrying Food" value=" (list ant) count
    (each.state = 'carryingFood') " />

```

```
      <data name="Not Carrying Food" value=" (list ant) count
(each.state != 'carryingFood') " />
    </chart>
    <display name="grid" type="grid" environment="ant_grid"
fast_draw="fast_draw" refresh_every="2" />
</output>
```

## **batch**

Please see this section for a detailed description of Batch.

# Article Sources and Contributors

**GAMA** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=GAMA> *Contributors:* Alexis Drogoul, DANG Kim Dung, Edouard, Vo Duc An, 4 anonymous edits

**First steps** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=First\\_steps](http://www1.ifi.auf.org/mediawiki/index.php?title=First_steps) *Contributors:* Alexis Drogoul, DANG Kim Dung, Vo Duc An

**Interface guide** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Interface\\_guide](http://www1.ifi.auf.org/mediawiki/index.php?title=Interface_guide) *Contributors:* Alexis Drogoul, DANG Kim Dung, Edouard, Vo Duc An

**Stupid Tutorial** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Stupid\\_Tutorial](http://www1.ifi.auf.org/mediawiki/index.php?title=Stupid_Tutorial) *Contributors:* Alexis Drogoul, Edouard, Vo Duc An

**StupidModel1** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel1> *Contributors:* Alexis Drogoul, Edouard, Guillaume Chérel

**StupidModel10** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel10> *Contributors:* Alexis Drogoul, Edouard, Guillaume Chérel

**StupidModel11** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel11> *Contributors:* Alexis Drogoul, Edouard

**StupidModel12** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel12> *Contributors:* Edouard, Guillaume Chérel, Vo Duc An

**StupidModel13** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel13> *Contributors:* Edouard, Guillaume Chérel, Vo Duc An

**StupidModel14** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel14> *Contributors:* Alexis Drogoul, Edouard, Vo Duc An

**StupidModel15** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=StupidModel15> *Contributors:* Edouard, Vo Duc An

**Modeling guide** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Modeling\\_guide](http://www1.ifi.auf.org/mediawiki/index.php?title=Modeling_guide) *Contributors:* Alexis Drogoul, Edouard

**Behaviors** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Behaviors> *Contributors:* Alexis Drogoul, Vo Duc An

**Commands** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Commands> *Contributors:* Alexis Drogoul

**Operators** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Operators> *Contributors:* Alexis Drogoul, Edouard, Vo Duc An

**Keywords** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Keywords> *Contributors:* Alexis Drogoul

**Skills** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Skills> *Contributors:* Alexis Drogoul, Edouard, Vo Duc An

**Built-in** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Built-in> *Contributors:* Alexis Drogoul, Guillaume Chérel, Vo Duc An

**Species** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Species> *Contributors:* Alexis Drogoul, Vo Duc An

**Sections** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Sections> *Contributors:* Alexis Drogoul, Vo Duc An

# Image Sources, Licenses and Contributors

**Image:splash1.png** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Splash1.png> *License:* unknown *Contributors:* -

**Image:IRD.jpg** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:IRD.jpg> *License:* unknown *Contributors:* -

**Image:voronoi.png** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Voronoi.png> *License:* unknown *Contributors:* -

**Image:GAMA\_UL\_4.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA\\_UL\\_4.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA_UL_4.png) *License:* unknown *Contributors:* -

**Image:GAMA\_UL\_5.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA\\_UL\\_5.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA_UL_5.png) *License:* unknown *Contributors:* -

**Image:GAMA\_UL\_6.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA\\_UL\\_6.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA_UL_6.png) *License:* unknown *Contributors:* -

**Image:GAMA\_UL\_9.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA\\_UL\\_9.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA_UL_9.png) *License:* unknown *Contributors:* -

**Image:GAMA\_UL\_10.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA\\_UL\\_10.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:GAMA_UL_10.png) *License:* unknown *Contributors:* -

**Image:Page\_go.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Page\\_go.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Page_go.png) *License:* unknown *Contributors:* -

**Image:Start.png** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Start.png> *License:* unknown *Contributors:* -

**Image:Stop.png** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Stop.png> *License:* unknown *Contributors:* -

**Image:Camera.png** *Source:* <http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Camera.png> *License:* unknown *Contributors:* -

**Image:Go\_back\_simulation2.png** *Source:* [http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Go\\_back\\_simulation2.png](http://www1.ifi.auf.org/mediawiki/index.php?title=Image:Go_back_simulation2.png) *License:* unknown *Contributors:* -