GAMA

Sommario

[1.INTRODUCTION 3](#_Toc108421667)

[2.ORGANIZATION OF A MODEL 5](#_Toc108421668)

[2.1.Model header(model species) 5](#_Toc108421669)

[2.1.1.Declaration name of the model 5](#_Toc108421670)

[2.1.2.Imports 5](#_Toc108421671)

[2.1.3.Definition of global species 6](#_Toc108421672)

[2.2.Species declaration 6](#_Toc108421673)

[2.3.Experiment declarations 7](#_Toc108421674)

[2.3.1.GUI Experiment 7](#_Toc108421675)

[2.3.2.BATCH Experiment 7](#_Toc108421676)

[2.3.3.Test experiment 8](#_Toc108421677)

[2.3.4.Memorize experiment 8](#_Toc108421678)

[2.4.Basic skeleton of a model 8](#_Toc108421679)

[3.BASIC PROGRAMMING CONCEPTS IN GAML 9](#_Toc108421680)

[4.MANIPULATE BASIC SPECIES 10](#_Toc108421681)

[4.1.GLOBAL SPECIES 10](#_Toc108421682)

[4.1.1.Declaration 10](#_Toc108421683)

[4.1.2.Environment size 11](#_Toc108421684)

[4.1.3.Built-in attributes 11](#_Toc108421685)

[4.1.4.Built-in actions 11](#_Toc108421686)

[4.1.5.The init statement 11](#_Toc108421687)

[4.2.REGULAR SPECIES 12](#_Toc108421688)

[4.2.1.Declaration 12](#_Toc108421689)

[4.2.2.Built-in attributes 12](#_Toc108421690)

[4.2.3.Built-in action 13](#_Toc108421691)

[4.2.4.The init statement 13](#_Toc108421692)

[4.2.5.The aspect statement 13](#_Toc108421693)

[4.2.6.Instantiate an agent 14](#_Toc108421694)

[4.2.7.Example: people who don’t move 15](#_Toc108421695)

[4.3.DEFINING ACTIONS AND BEHAVIORS 16](#_Toc108421696)

[4.3.1.Declare an action 16](#_Toc108421697)

[4.3.2.Call an action 16](#_Toc108421698)

[4.3.3.Behavior 16](#_Toc108421699)

[4.3.4.Example: ballons 17](#_Toc108421700)

[4.3.5.Example: people moving 18](#_Toc108421701)

[4.3.6.Example: happy people model 19](#_Toc108421702)

[4.4.INTERATION BETWEEN AGENTS 20](#_Toc108421703)

[4.4.1.The ask statement 20](#_Toc108421704)

[4.5.ATTACHING SKILLS 21](#_Toc108421705)

[4.6.INHERITANCE 21](#_Toc108421706)

[5.DEFINING ADVANCED SPECIES 22](#_Toc108421707)

[5.1.GRID SPECIES 22](#_Toc108421708)

[5.2.GRAPH SPECIES 22](#_Toc108421709)

[5.3.MIRROR SPECIES 22](#_Toc108421710)

[5.4.MULTI-LEVEL ARCHITECTURE 22](#_Toc108421711)

[6.DEFINING GUI EXPERIMENT 22](#_Toc108421712)

[7.EXPLORING MODELS 22](#_Toc108421713)

[8.OPTIMIZING MODELS 23](#_Toc108421714)

[9.MULTI-PARADIGM MODELLING 24](#_Toc108421715)

[9.1.CONTROL ARCHITECTURE 24](#_Toc108421716)

[Finite state machine 24](#_Toc108421717)

[Task-Based 25](#_Toc108421718)

[Rules-based architecture 25](#_Toc108421719)

[User control architecture 25](#_Toc108421720)

[9.2.USING EQUATIONS(molto interessante…da integrare quando ho tempo) 25](#_Toc108421721)

[10.RECIPES(pratical “how to”s on various subjects) 25](#_Toc108421722)

[10.1.MANIPULATE OSM DATAS 25](#_Toc108421723)

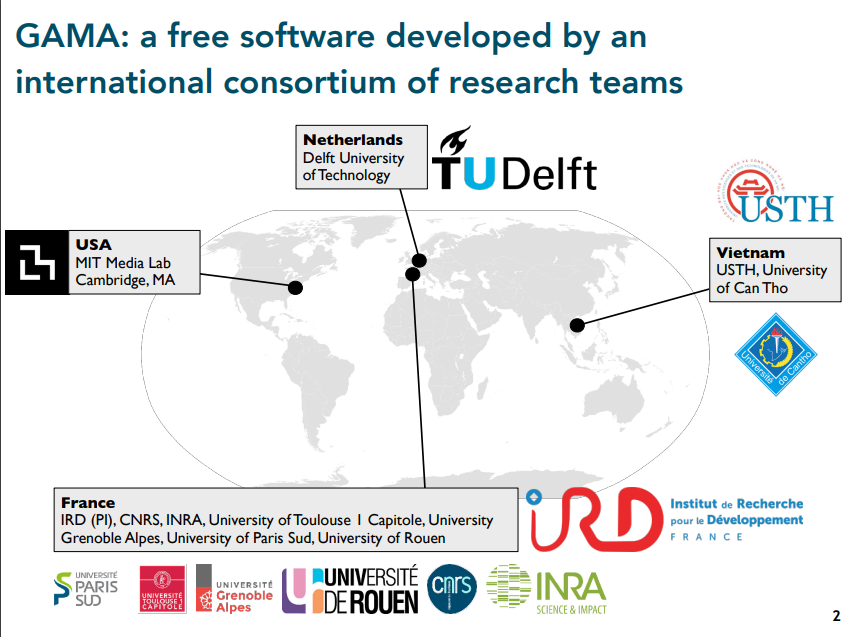
[TERMINOLOGIA 26](#_Toc108421724)

[Agent-oriented programming 26](#_Toc108421725)

[Software agent 26](#_Toc108421726)

[RIFERIMENTI 27](#_Toc108421727)

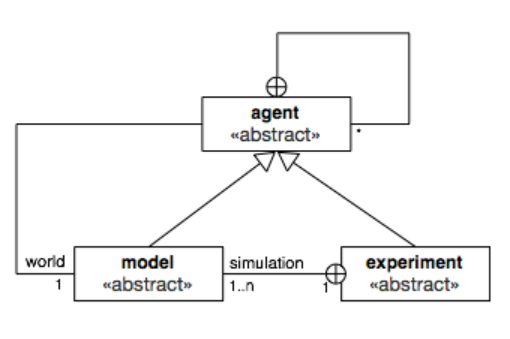
# 1.INTRODUCTION

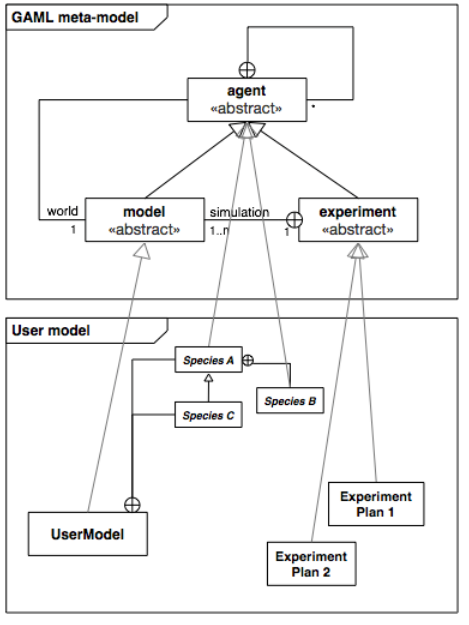


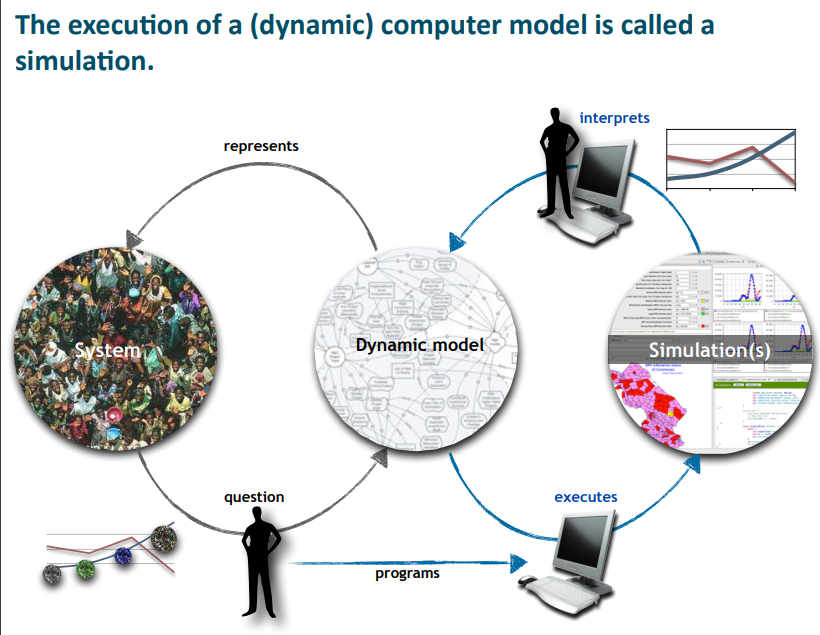
GAML is an **agent-oriented language** dedicated to the definition of agent-based simulations. It takes its roots in object-oriented languages like Java or Smalltalk, but extends the object-oriented programming approach with powerful concepts (like skills, declarative definitions or agent migration)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

The role of GAML is to support modelers to write their **models**, which are specifications of **simulations** that can be executed and controlled during **experiments,** themselves specified by **experiment plans.**

Like in the object-oriented paradigm, where the notion of class is used to supply a specification of objects, agents in GAML are specified by their **species** which provides them with a set of **attributes (what they know), actions (what they can do), behaviors (what they actually do)** and also specifies properties of their **population,** for istance its **topology (how they are connected)** or **schedule (in which order and when they should execute).**

****





# 2.ORGANIZATION OF A MODEL

## 2.1.Model header(model species)

### 2.1.1.Declaration name of the model



Importing a model can take 2 forms. The first one, called **inheritance import,** in which all the declarations of the model imported will be merged with those of the current model.The second one, called **usage import,** reserved for **using models as micro-models of the current model.**

### 2.1.2.Imports

Inheritance import

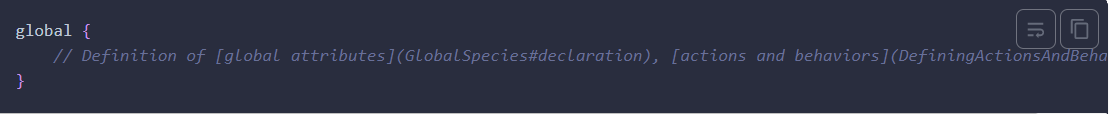


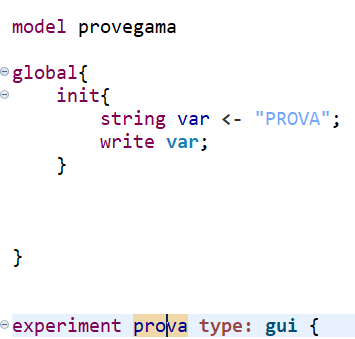
Usage import



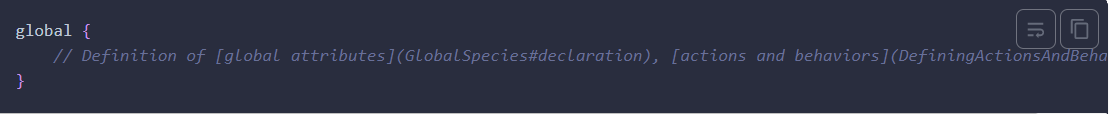
### 2.1.3.Definition of global species

The last part of the **header** is the definition of the global species:



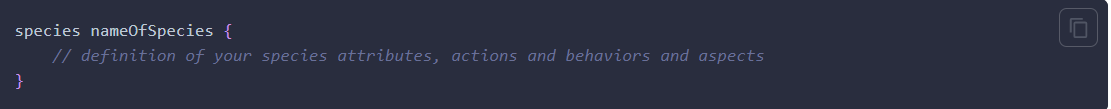


## 2.2.Species declaration

The **special species global** is the world species. You will declare here all the global attributes/actions/behaviors.

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

A species defines its **attributes, actions and behaviors** and **aspects.**

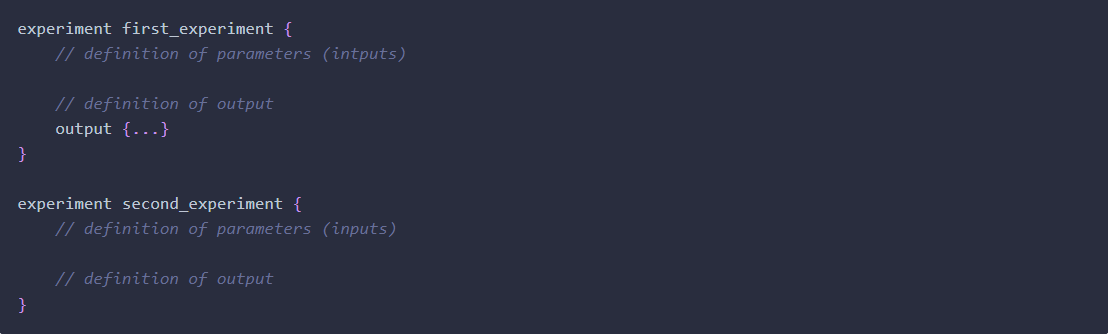
****

Species position:

**Immagine che contiene testo

Descrizione generata automaticamente**

## 2.3.Experiment declarations



### 2.3.1.GUI Experiment

GUI experiment allows you to display a **graphical interface**

****

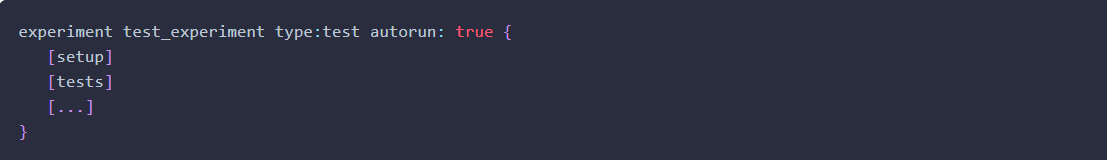
### 2.3.2.BATCH Experiment

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



### 2.3.3.Test experiment

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

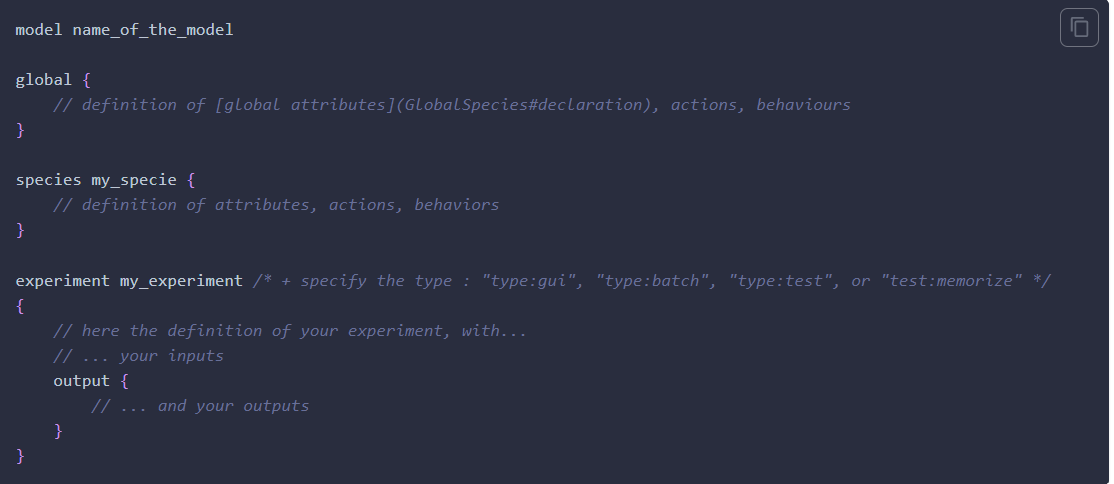


### 2.3.4.Memorize experiment

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



## 2.4.Basic skeleton of a model



# 3.BASIC PROGRAMMING CONCEPTS IN GAML



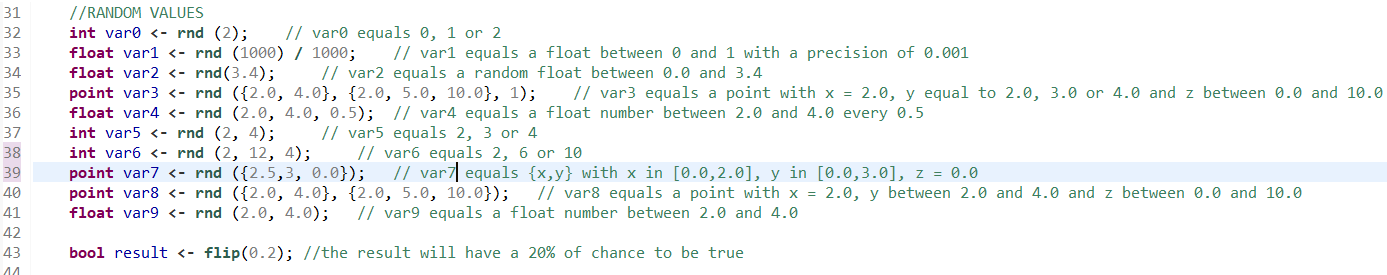


Immagine che contiene testo

Descrizione generata automaticamente



# 4.MANIPULATE BASIC SPECIES

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

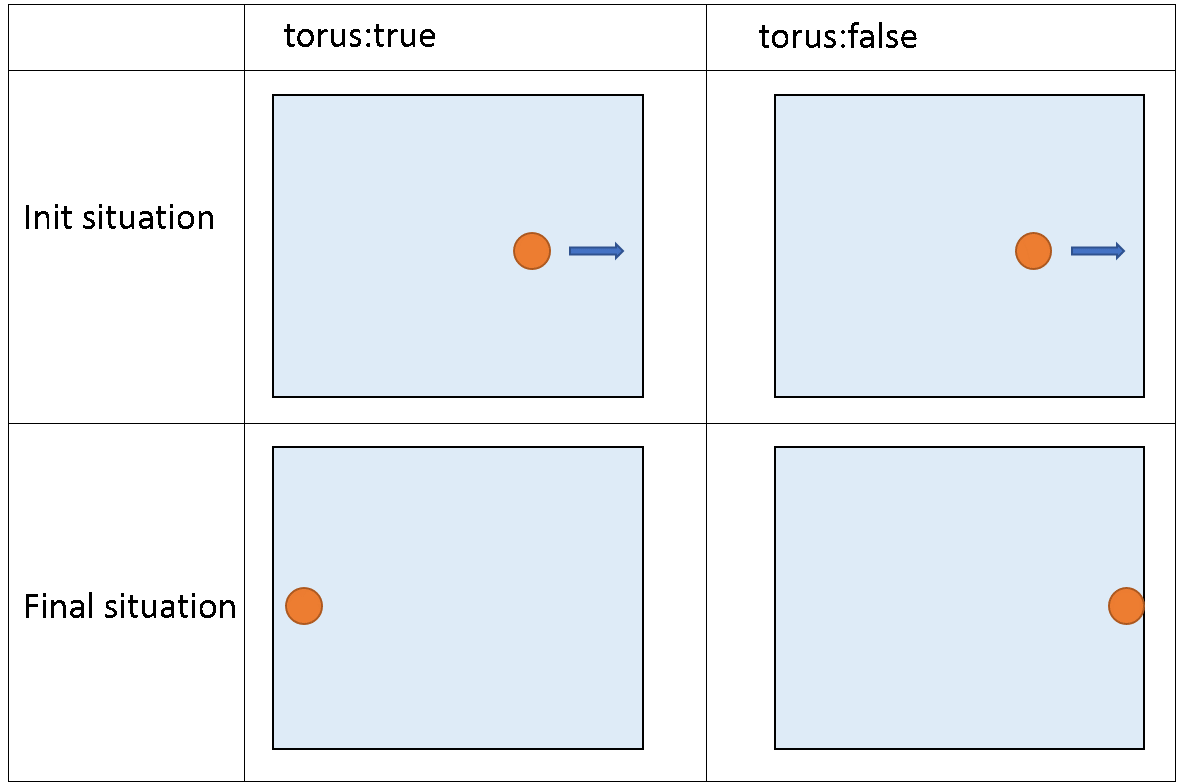
## 4.1.GLOBAL SPECIES

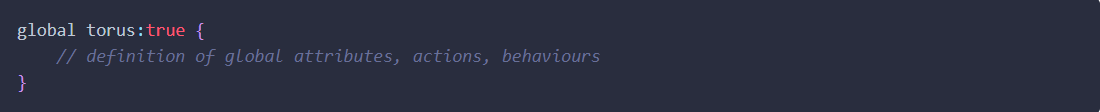
In the **global species** you can define the attributes, actions and behaviors that describe the world agent.

There is one **unique world agent per simulation:** it is this agent that is created when a user runs an experiment and that initializes the simulation through its **init scope**.

### 4.1.1.Declaration

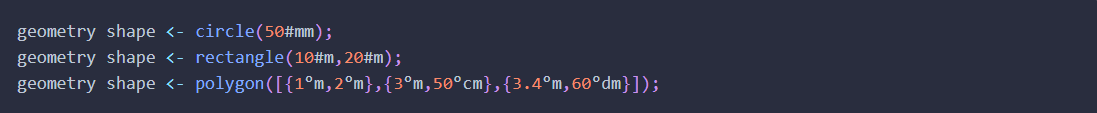
**Global** can use **facets**, such as the **torus** facet, to make the environment s **torus** or not (if it is a torus, all the agents **going out of the environment will appear on the other side**. If it’s not, the agent **won’t be able to go out of the environment**).





### 4.1.2.Environment size

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



### 4.1.3.Built-in attributes

#### World

#### Experiment

#### Cycle

#### Step

#### Time

#### Starting\_date and current\_date

#### Duration

#### Total\_duration

#### Average\_duration

#### Machine\_time

#### Seed

#### agents

### 4.1.4.Built-in actions

#### Pause

#### die

### 4.1.5.The init statement

You can define an initial state, before launching the simulation.

Here, you normally initialize your global variables, and you instantiate your species.

## 4.2.REGULAR SPECIES

**Regular species** are composed of attributes, actions, reflex, aspect, etc…

**Species** represents a **“prototype” of agents**: it defines their common properties.

They describe the behavior of our agents.You **can instantiate as much as you want agents** from a regular species, and you can instantiate as much as you want agents from a regular species, and you can define as much as you want different regular species.You can see a species as a “class” in OOP.

### 4.2.1.Declaration

Species with attribute



### 4.2.2.Built-in attributes

Immagine che contiene testo

Descrizione generata automaticamente

#### Name

#### Location

#### Shape

#### Host

#### Members

### 4.2.3.Built-in action

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

Immagine che contiene testo

Descrizione generata automaticamente

### 4.2.4.The init statement

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



### 4.2.5.The aspect statement

This block allows you to define how you want your species to be represented in the simulation.



In the experiment block, you have to tell the program to display a particular species with a particular aspect:

Immagine che contiene testo

Descrizione generata automaticamente

### 4.2.6.Instantiate an agent

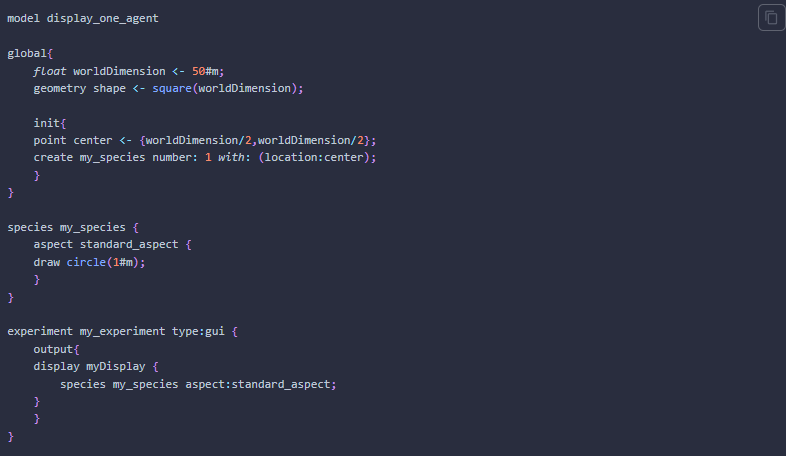
As already said quickly in the last session, the instantiation of the agents is **most often** in the **init** scope of the **global** species (**this is not mandatory** of course.You can instantiate your agents from an action/behavior of any species).

Use the statement **create** to instantiate an agent.

The facet **with** is used to specify some some default values for some attributes of your instance.

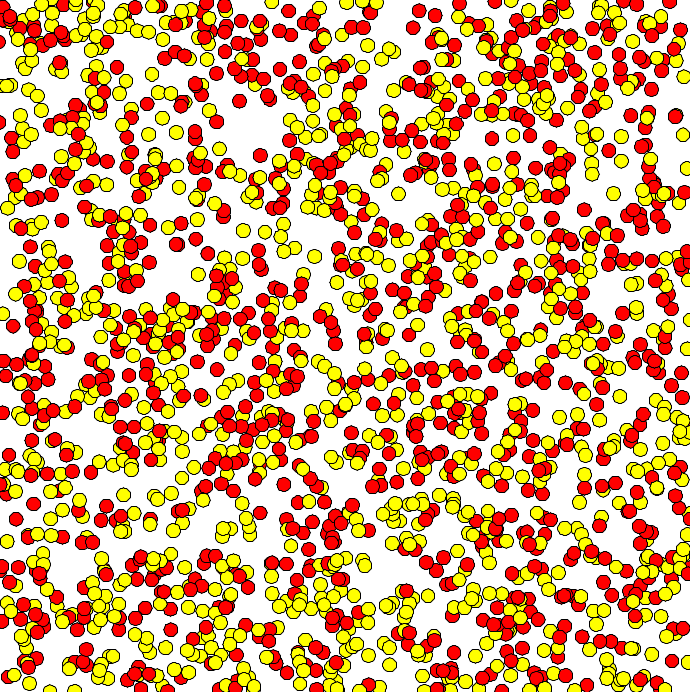
Immagine che contiene testo

Descrizione generata automaticamente



### 4.2.7.Example: people who don’t move

Immagine che contiene testo

Descrizione generata automaticamente

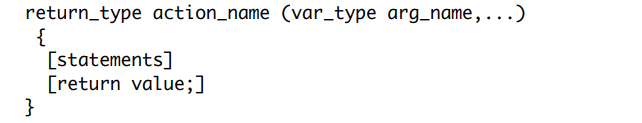
## 4.3.DEFINING ACTIONS AND BEHAVIORS

### 4.3.1.Declare an action

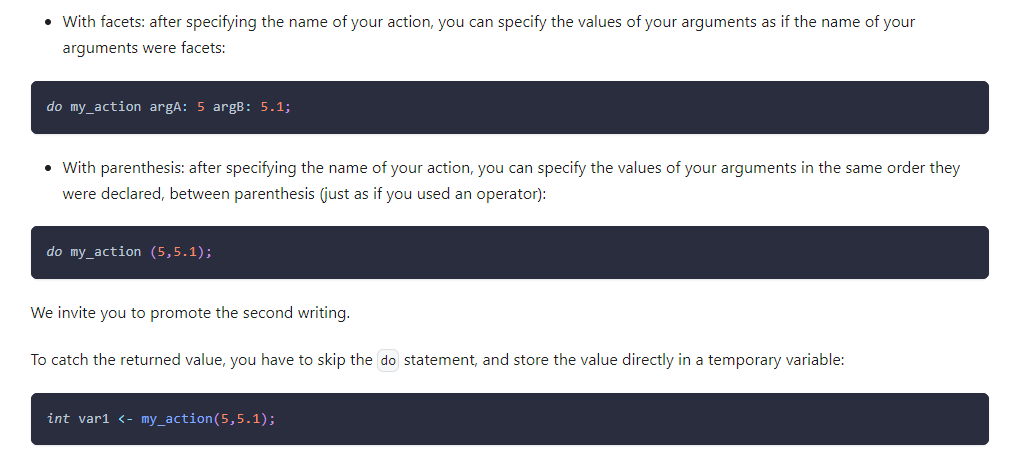
An **action** is a **function or procedure** run by an instance of species. An action can return a value (in that case, the type of return has to be specified just before the name of the action), or not (in that case, you just have to put the keyword action before the name of the action).



Action that returns a value



### 4.3.2.Call an action

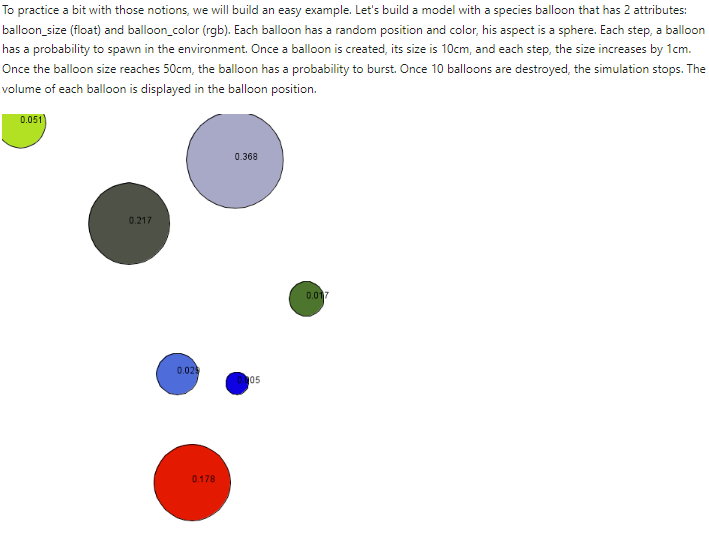


### 4.3.3.Behavior

A **behavior** or **reflex** is a set of statements which is called **automatically** at each time step by an agent.Note that, a behavior is linked to an **architecture;** the reflex based architecture is the default one, others can be used with the controls facet of the species.



### 4.3.4.Example: ballons

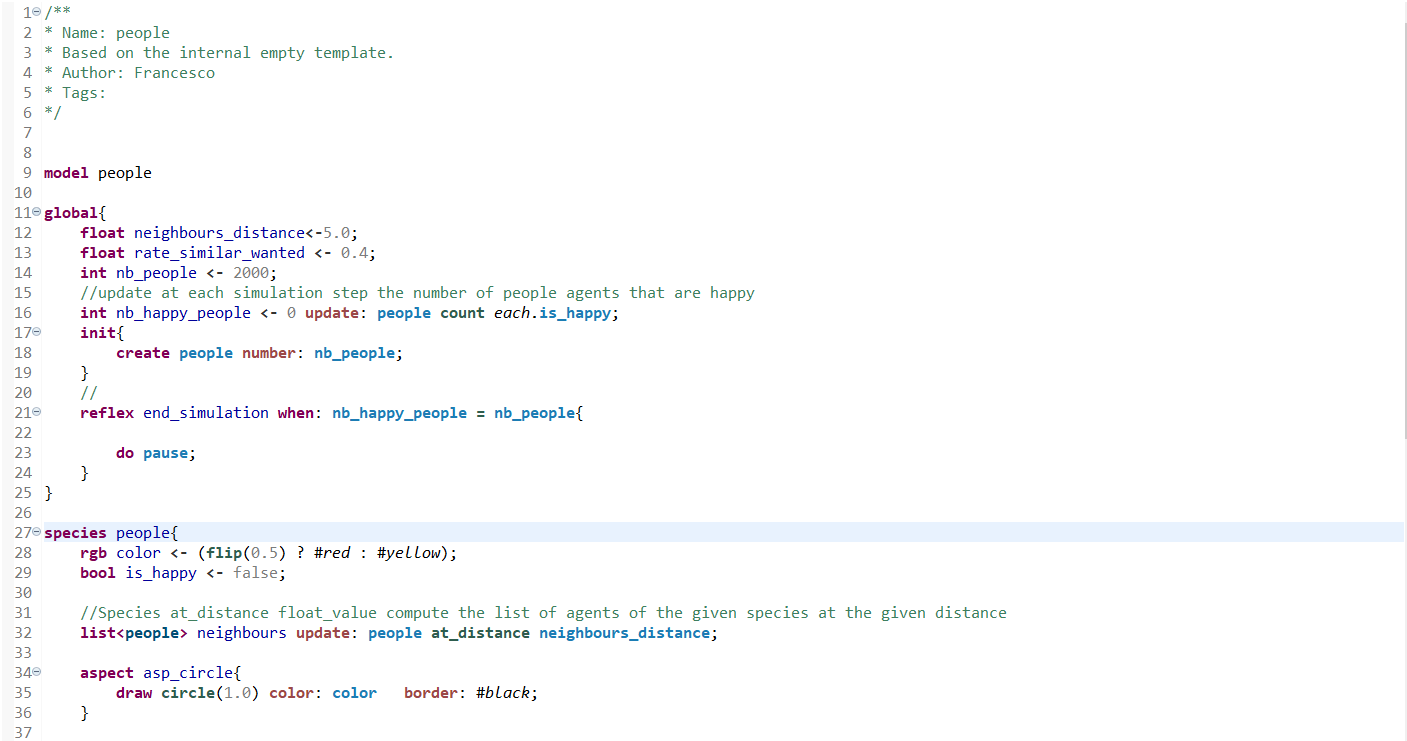
Immagine che contiene testo

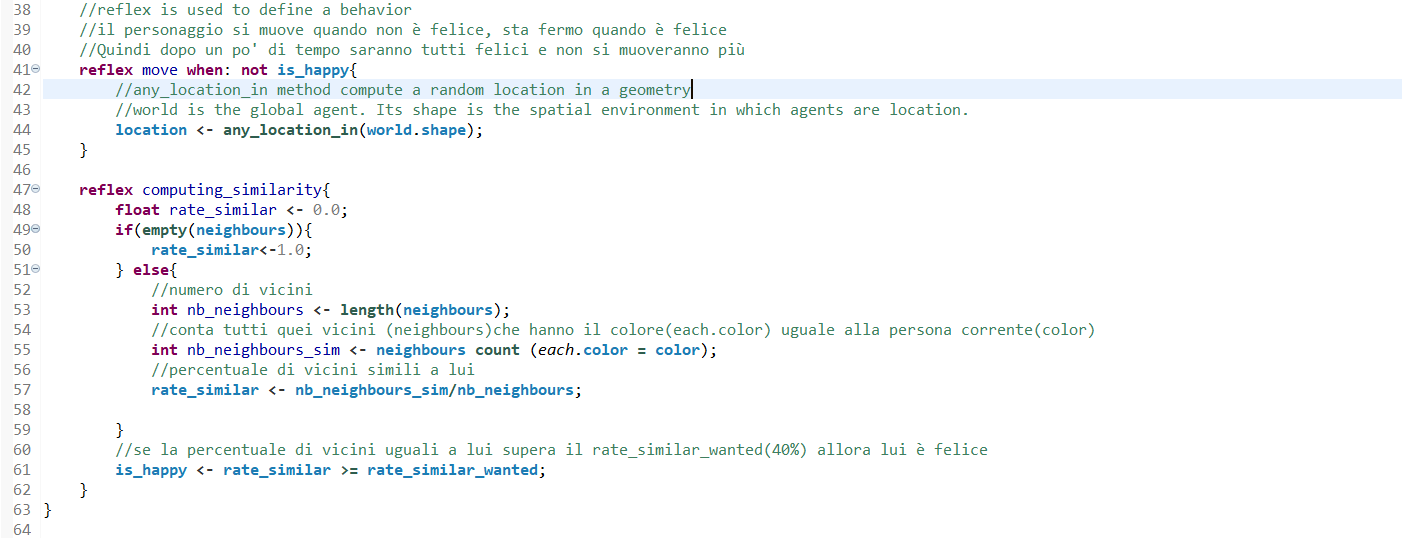
Descrizione generata automaticamente

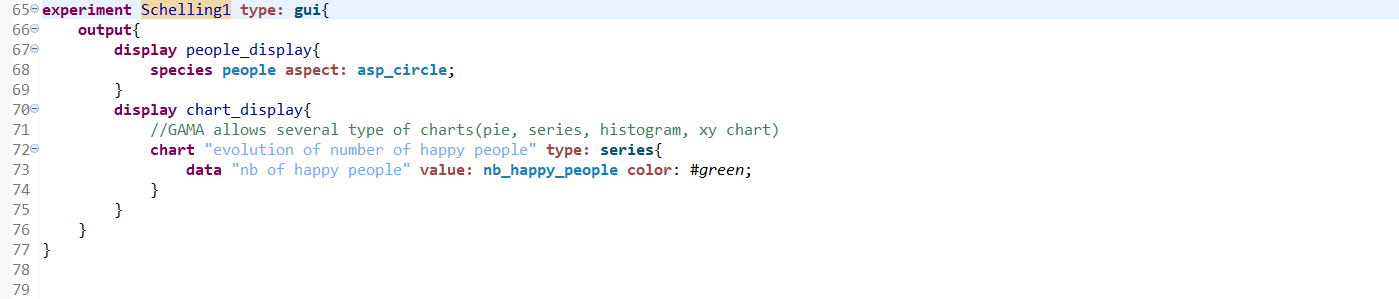
### 4.3.5.Example: people moving

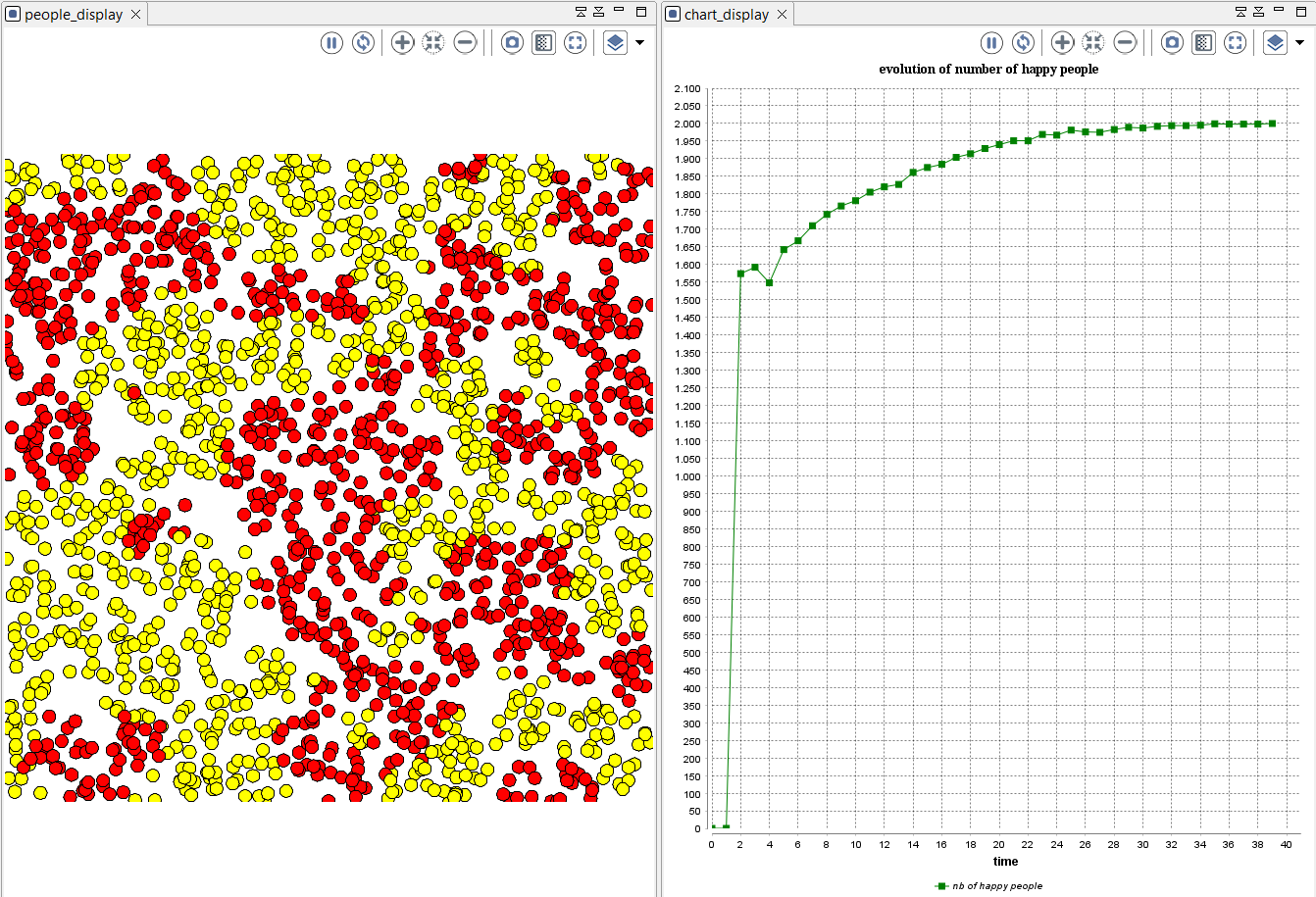


### 4.3.6.Example: happy people model









## 4.4.INTERATION BETWEEN AGENTS

### 4.4.1.The ask statement

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

If you want to interact with one particular agent



If you want interact with a group of agents



If you want interact with agents, as if they were instance of a certain species (can raise an error if it is not the case)



If you want to interact with all the agents of a species(note that the name of the species can be used in the **ask,** and in many other situations, as the population of this species, i.e. the list of agents instance of this species)

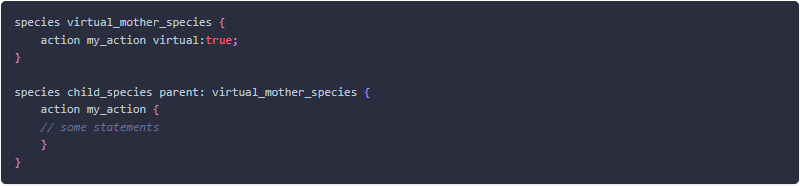


## 4.5.ATTACHING SKILLS

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

## 4.6.INHERITANCE

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



# 5.DEFINING ADVANCED SPECIES

## 5.1.GRID SPECIES

A grid is a particular species of agents. Indeed, a grid is a set of agents that share a grid topology.

**Grid agents are created automatically at the beginning of the simulation.** It is thus not necessary to use the **create statement** to create them. Moreover, in addition to classic built-in variables, **grid** comes with a set of additional built-in variables.

## 5.2.GRAPH SPECIES

## 5.3.MIRROR SPECIES

## 5.4.MULTI-LEVEL ARCHITECTURE

# 6.DEFINING GUI EXPERIMENT

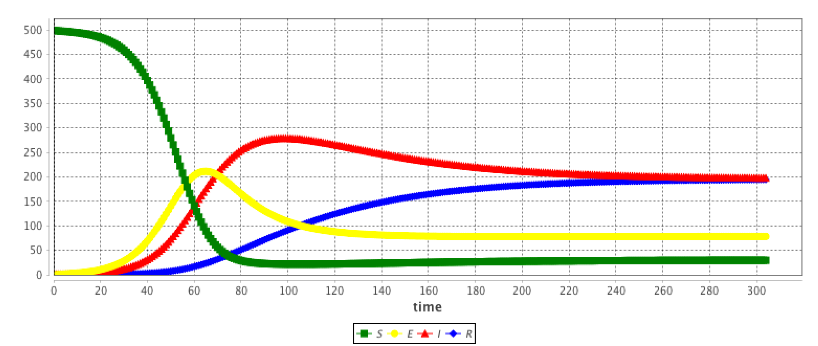
# 7.EXPLORING MODELS

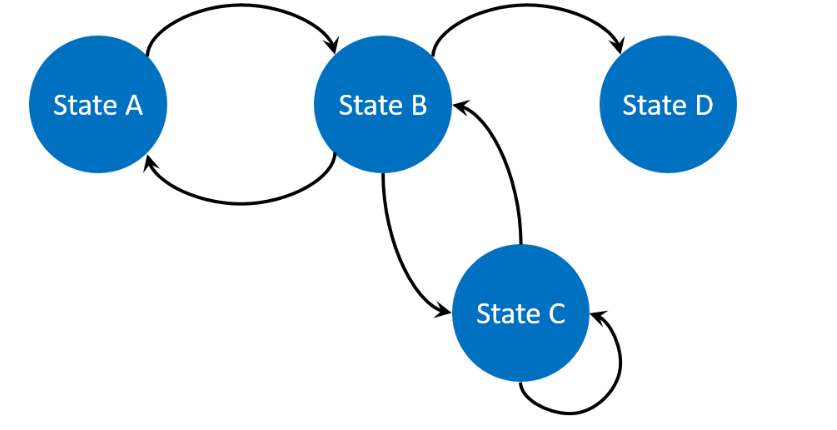
In order to explore the models (by automatically running the experiment using several configurations to analyze the outputs), a first approach is to run **several simulations from the same experiment**, considering each simulation as an agent. A second approach, much more efficient for larger explorations, is to run an other type of experiment: the **BATCH EXPERIMENT.**

# 8.OPTIMIZING MODELS

The execution time of a model depends a lot of how you implemented your model!

# 9.MULTI-PARADIGM MODELLING





Multi-paradigm modelling is a research field focused on how to define a model semantically. Our approach is based on **behavior (or reflex), for each agents**. In this part, we will see that GAMA provides other ways to implement your model, using several control architectures. Sometime, it will be easier to implement your model choosing other paradigms.

## 9.1.CONTROL ARCHITECTURE

GAMA allows the modeller to attach built in control architecture **to agents**.

### Finite state machine

**FSM (Finite State Machine)** is a finite state machine-based behavior model. During its life cycle, the agent can be in several states. At any given time step, it is one single state. Such an agent needs to have one single state (the state in which it will be at its initialization).

### Task-Based

**Task-based**

### Rules-based architecture

### User control architecture

## 9.2.USING EQUATIONS(molto interessante…da integrare quando ho tempo)

# 10.RECIPES(pratical “how to”s on various subjects)

## 10.1.MANIPULATE OSM DATAS

# TERMINOLOGIA

## Agent-oriented programming

**Agent oriented programming** is a programming paradigm where the costruction of the software is centered on the concept of **software agents**.

**Agent oriented programming** can be viewed as a specialization of **object oriented programming.**

The **state of an agent** consists of components such as beliefs, decisions, capabilities, and obligations, for this reason the state of an agent is called its **mental state.**

## Software agent

In computer science a **software agent** is a computer program that acts for a user or other program in a relationship of agency, which derives from the Latin agere(to do). **Agents** are colloquially known as **bots** or **robots.**

They may be embodied, as when execution is paired with a robot body, or as software such as chatbot executing on a phone (Siri) or other computer device. Software agents may be **autonomous** or **work together** with other agent or people.

# RIFERIMENTI

[1] [Documentazione ufficiale di GAMA](https://gama-platform.org/)

[2] [Wikipedia: agent oriented programming](https://en.wikipedia.org/wiki/Agent-oriented_programming)

[3] [Wikipedia: software agent](https://en.wikipedia.org/wiki/Software_agent)

[4] [sciencedirect: agent oriented programming](https://www.sciencedirect.com/science/article/abs/pii/0004370293900349)