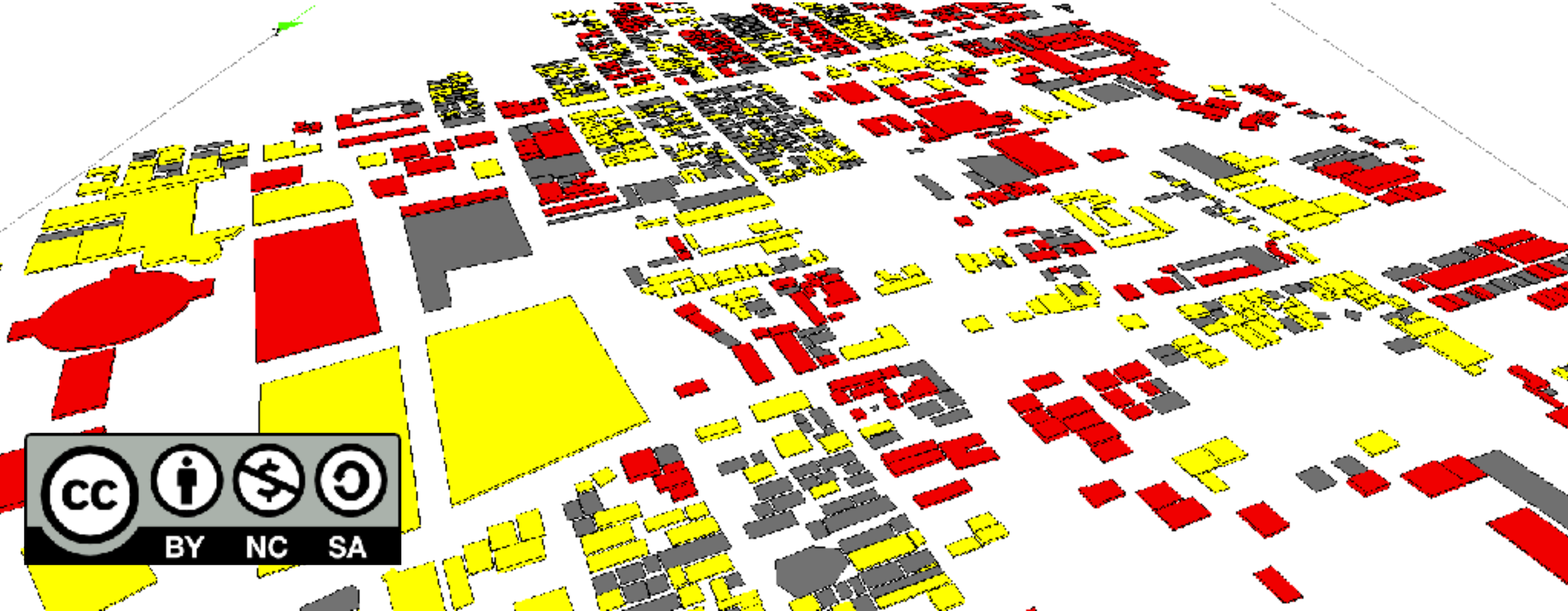
A Practical introduction to GAMA Through a Segregation model

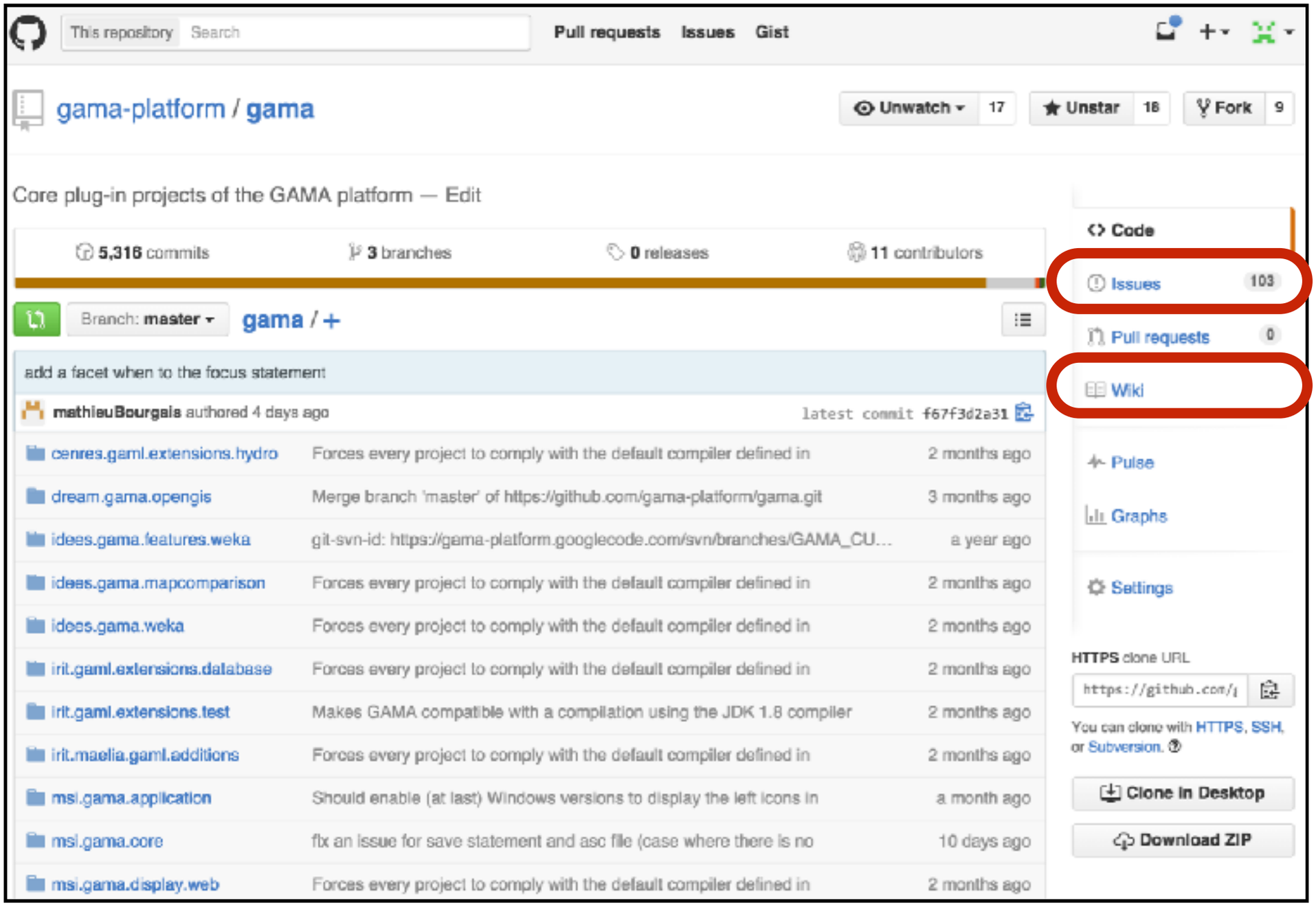
Benoit Gaudou, IRIT (Univ. Toulouse 1), benoit.gaudou@ut-capitole.fr
Patrick Taillandier (INRA, MIA, Toulouse), patrick.taillandier@gmail.com

with the participation of Alexis Drogoul, Jean-Daniel Zucker, Philippe Caillou and Srirāma Bhamidipati



GAMA: How to get documentation … and ask for help if needed

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



GAMA: How to get documentation

Contents

3

Structure of a model/program

Program myFirstModel
 	
global
Defines all global variables, model initialization and global behaviors.
 	
species mySpecies1
Defines variables, behaviors and aspects of agents of the species.
 	
experiment expName
Defines the way the model will be executed (includes the type of the execution, which global parameters can be modified, and what will be displayed during simulation)
	model myFirstModel
	
	global {
	// global variables declaration
	// initialization of the model
	// global behaviors
	}
	
	species mySpecies1 {
	// attributes, initialization, behaviors and aspects of a species
	}
	
	experiment expName {
	// Defines the way the model is executed, the parameters and the outputs.
	}

Comments

Block comments	/* A block comment starts with the an opening symbol /*. The comment runs until the closing symbol below. */
Inline comments	// This is an inline comment. // The // symbol have to be repeated before every line.

Primitive types

Integer number *# value between -2147483648 and 2147483647* **Real number** *# absolute value between $4.9*10^{-324}$ and $1.8*10^{308}$* **String** *# explicit value: "double quotes" or 'simples quotes'* **Boolean value** *# 2 values: true, false*	int float string bool

Other types

pair #with the two elements of undefined types **pair** #with two elements of types type1 and type2 #explicit value using :: symbol: e.g. 1::"one" **color** #explicit value: rgb(255,0,0) for red. (3 components: Red, Green, Blue) **point** #explicit value: {1.0, 3} or {1.0, 3, 6}. #internal representation with 4 coordinates	pair pair<type1, type2> rgb point

Variable or constant declaration, affectation

Declaration of a global variable or an attribute
#Global variables and species attributes can be declared with or without initial value.
 	
#Local variables need to be initialized when they are declared.
# explicit declaration of the type 	
# (if the type of the affected value is different, this value is automatically casted to the declared type) 	
 	
Declaration of a global variable or an attribute with a dynamic value
# value computed at each simulation step 	
 	
# value computed each time the variable is used 	
	// Global variables or species attributes
	int an_int;
	string a_string <- "my string";
	
	// Local variables
	float a_float <- 10.0;
	
	// Global variables or species attributes with dynamic value
	// inc int is incremented by 1 at each simulation step
	int inc_int <- 0 update: inc_int + 1;
	
	// random_int has a new random value each time it is used:
	int random_int -> { rnd(100) };

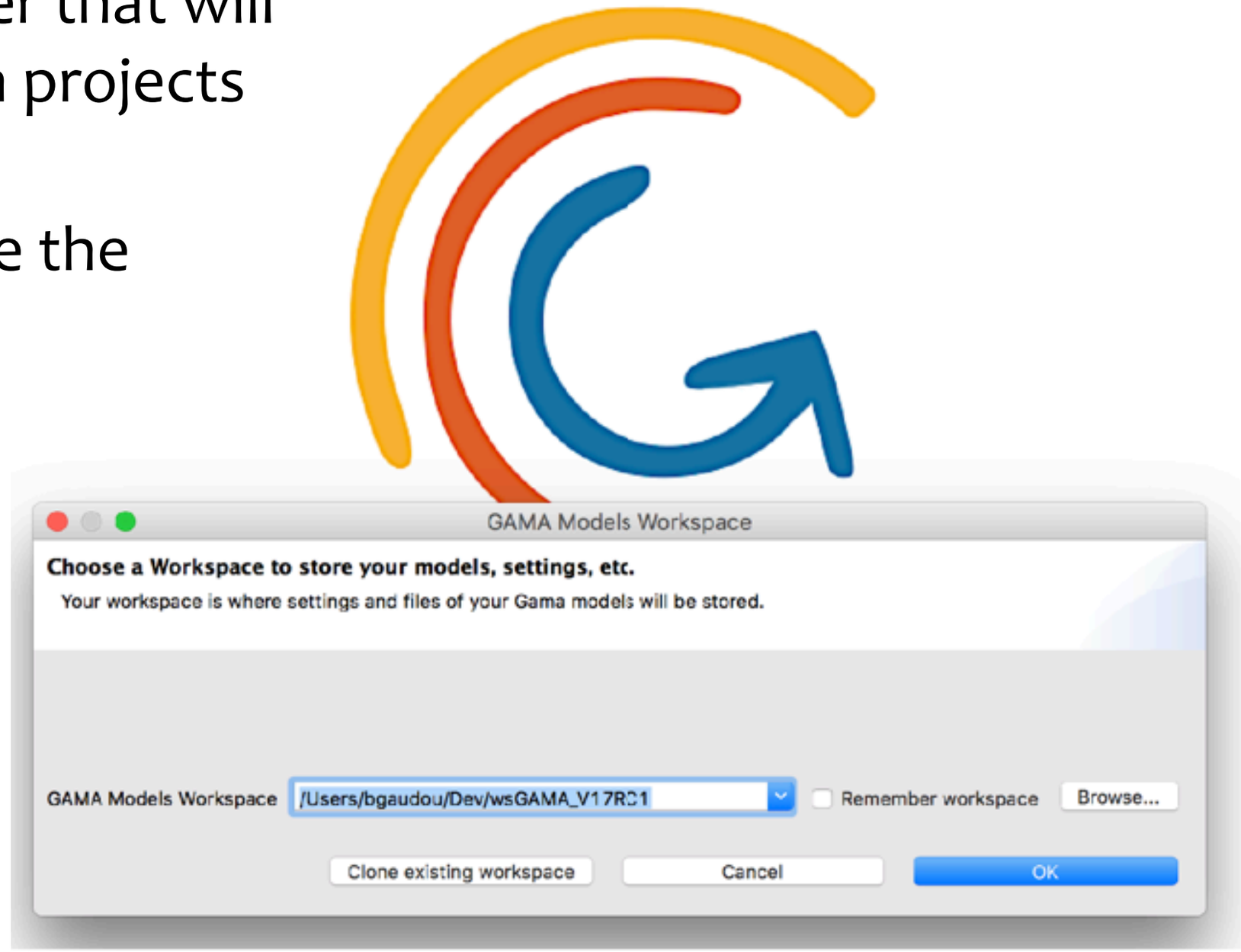
Definition of a constant	float pi <- 3.14 const: true;

Affectation of a value to a variable
Variable ← value or computed expression
	// Affectation of a value to an existing variable
	an_int <- 0;

Full Documentation of GAMA

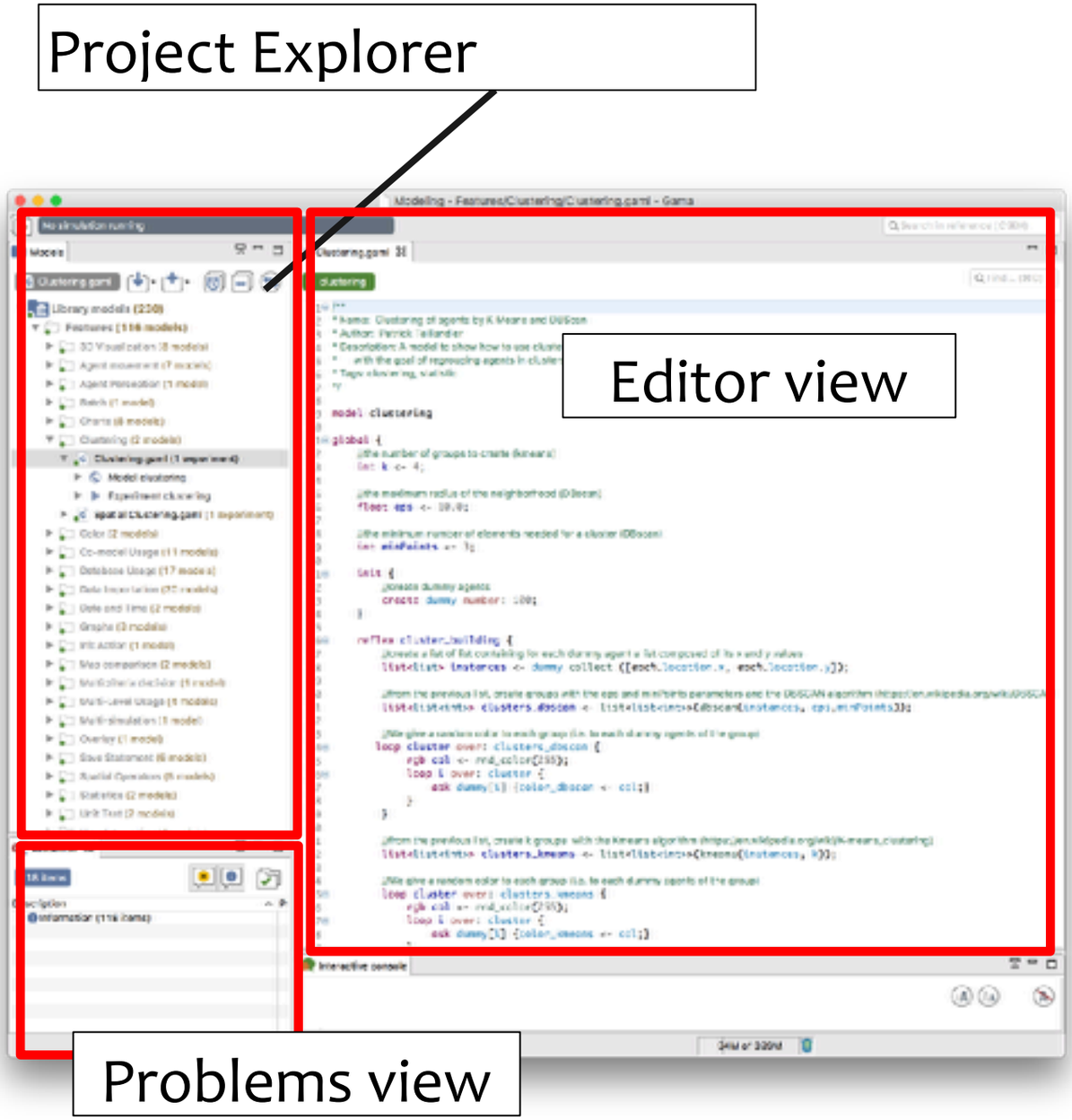
It is now time to run GAMA !

- First GAMA asks you to choose a workspace.
- A workspace is a folder that will contain all your own projects and models.
- You are free to choose the folder you want!



GAMA model files are stored in projects

- Each project may contain several models, as well as additional resources (GIS data, pictures,…).
- Belonging to the same project allows models to use each other and have access to the same resources.
- Projects can be organised in any way, although a default layout (“models”, “doc”, …) is proposed.



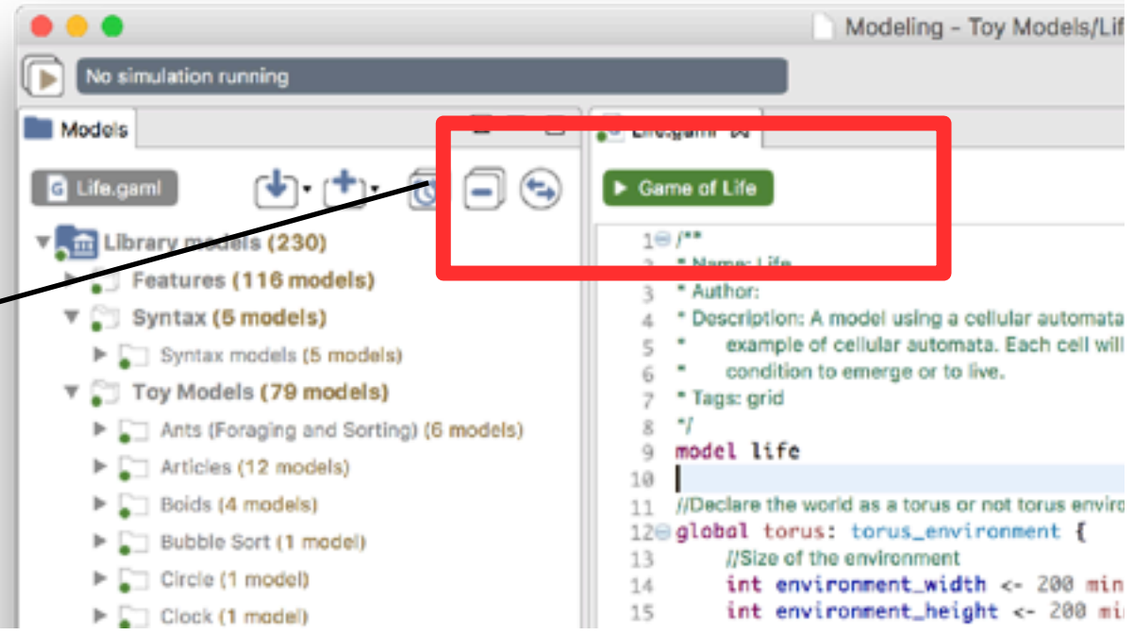
GAMA provides a library of models composed of 4 types of models:

- **Features**: simple models that show how to use some specific features (graph, GIS data, 3D….)
- **Syntax**: models that present the syntax of the GAML language
- **Toy models**: classic agents-based models (ant models, boids, Schelling…)
- **Tutorials**: models linked to online tutorials

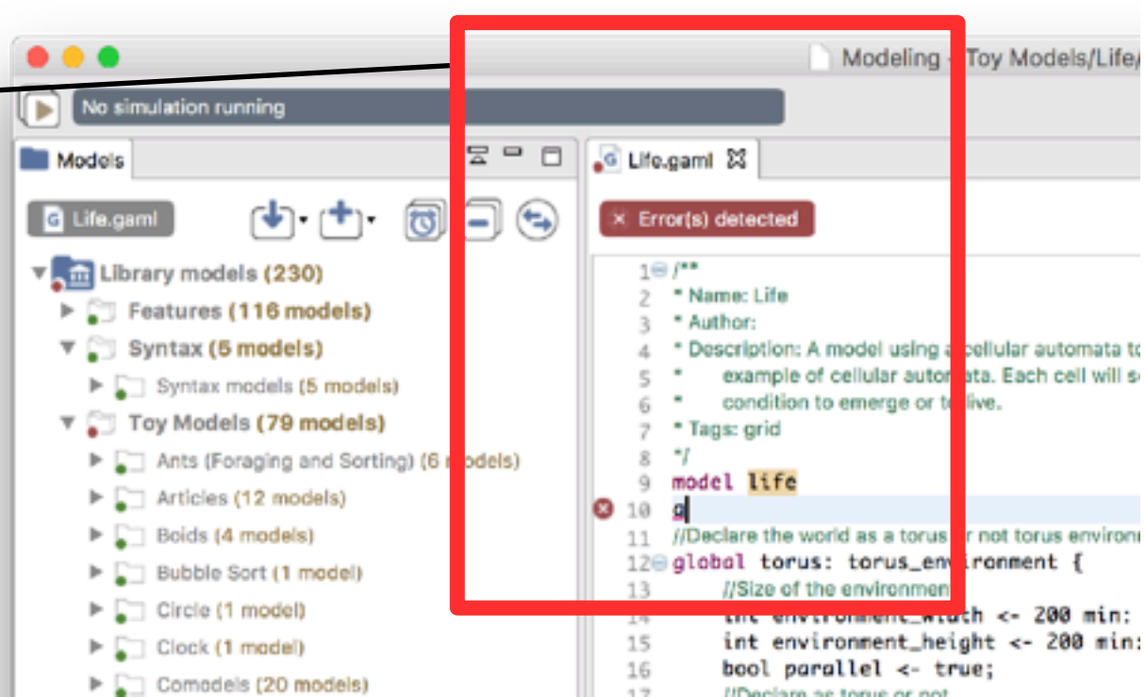
Take a look at “Game of life” model in library

- Models library \ Toy models \
  Life \ Life.gaml

The model can be experimented

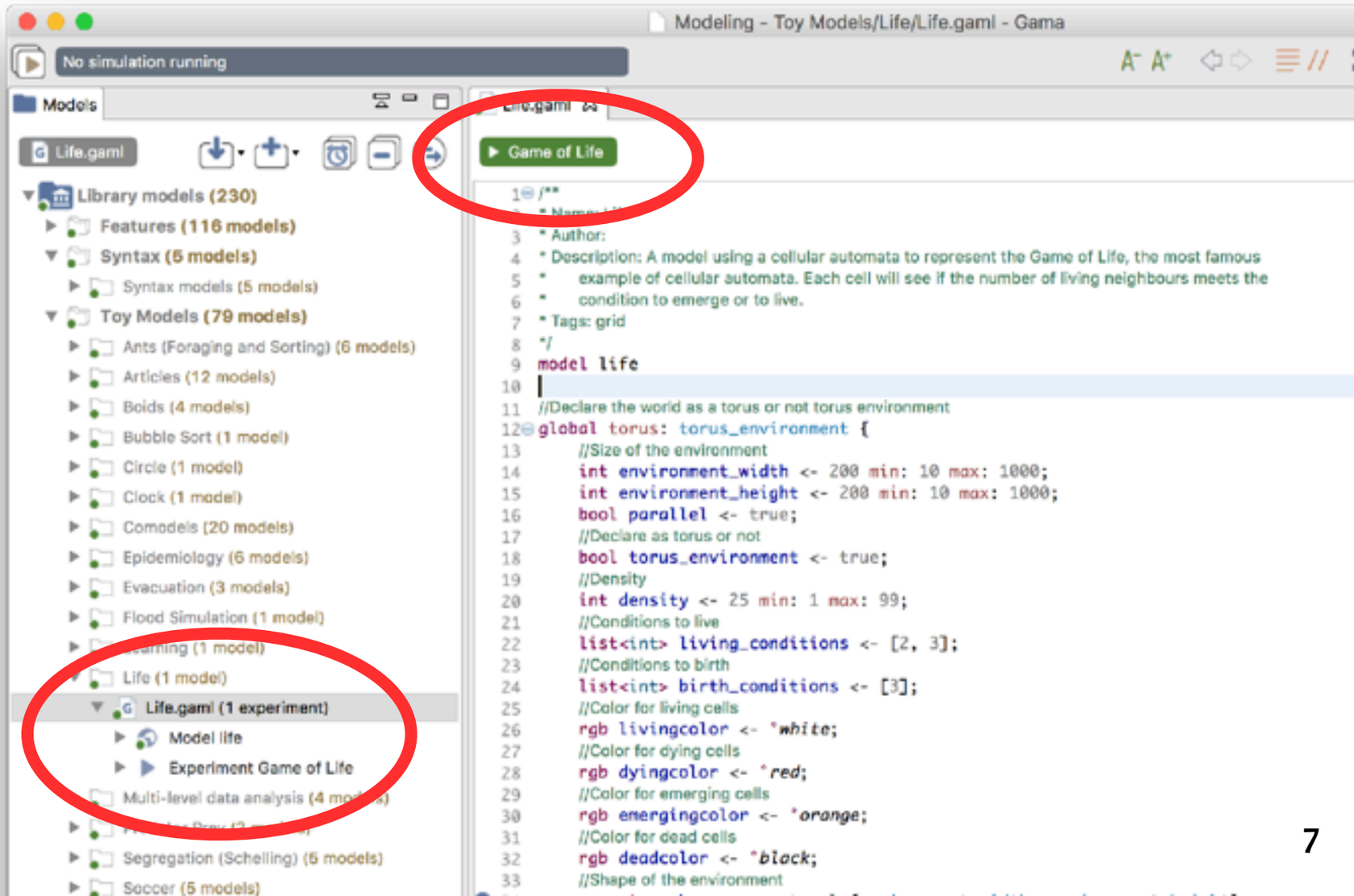


The model has errors



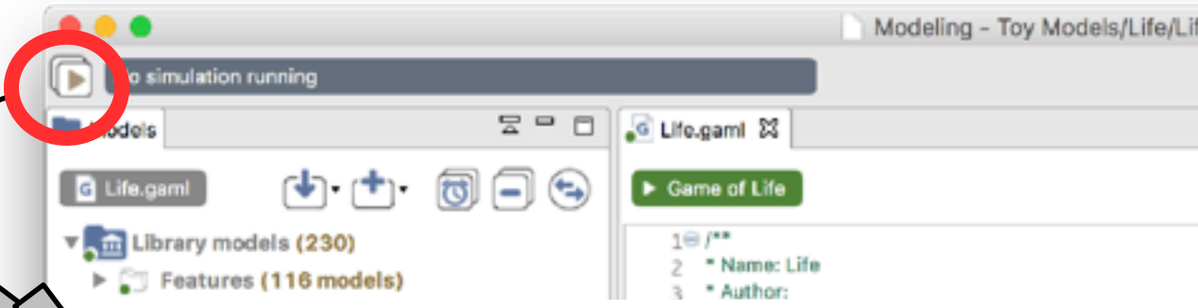
Launch experiment Game of life

- An experiment is a way to “run” a model. It can be reached either by:
  - Clicking on an Experiment button
  - in the Project Explorer, under the name of the model

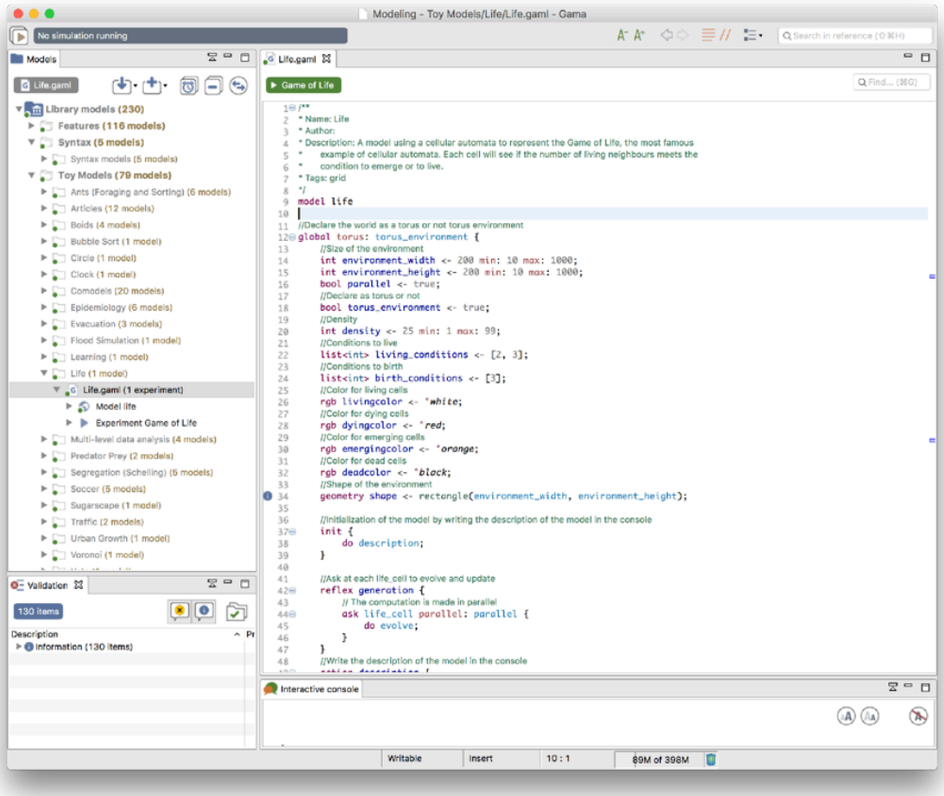


Launching an experiment will switch from *Modelling* to *Simulation* Perspective

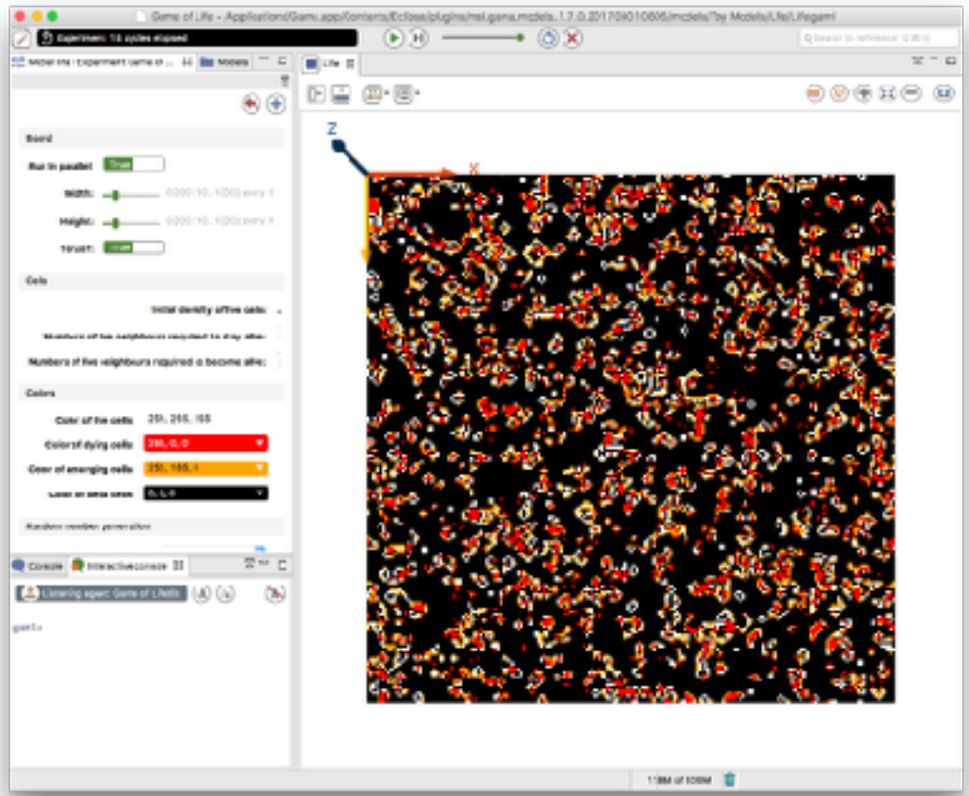
- Allows to change perspective



Modelling perspective

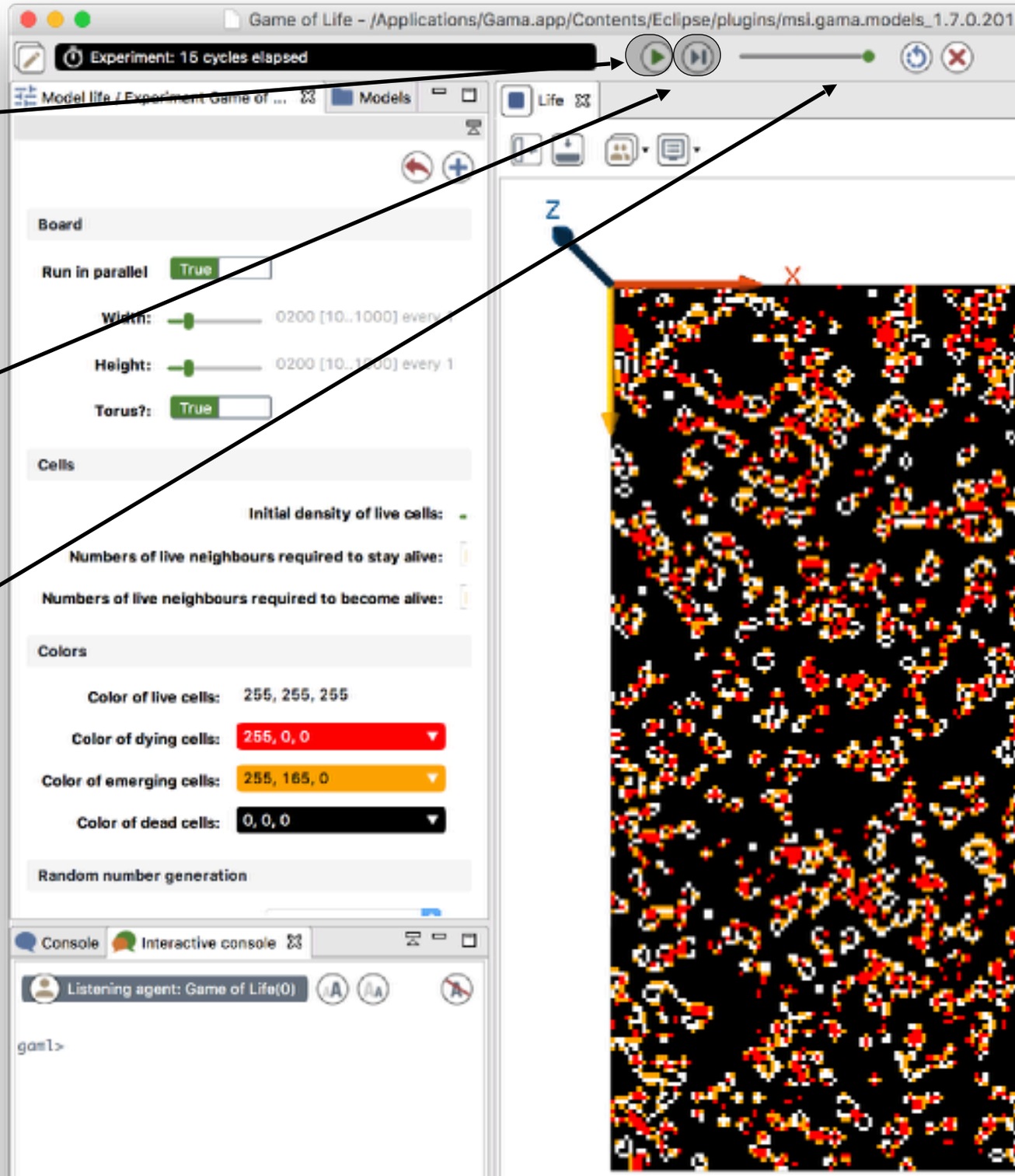


Simulation perspective



Exploring the Simulation perspective

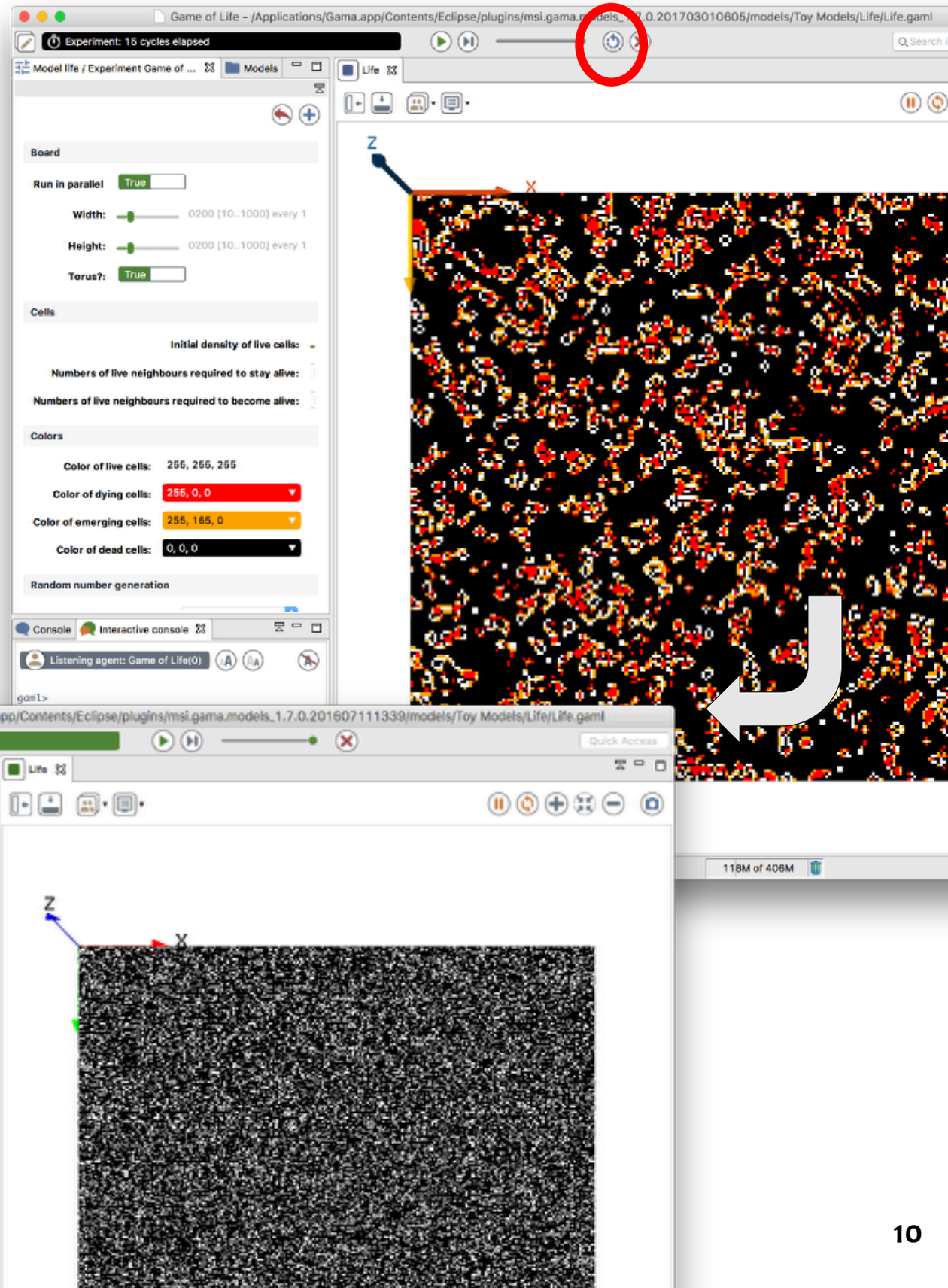
- Start/pause simulation (it will run until pause is clicked again)
- Step the simulation (it will run one cycle of the simulation)
- Adjust the speed of the simulation



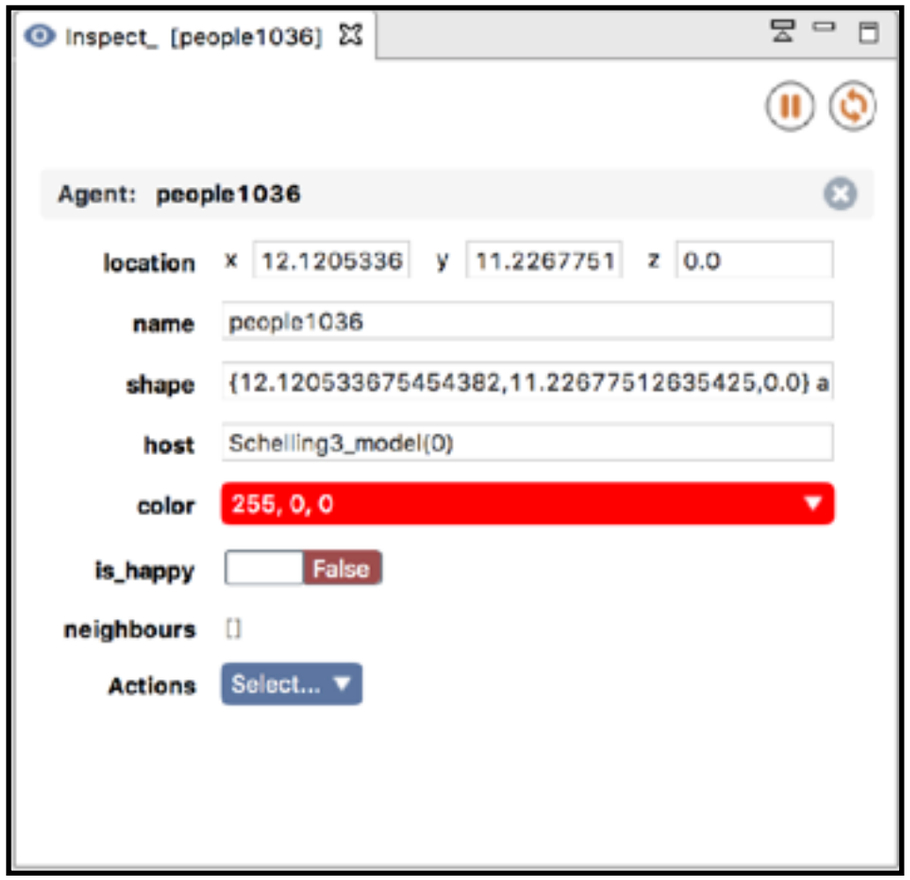
Explore the simulation with parameters modified from Parameters view

- The modifications made to the parameters are either:
  - Used for the current simulation when it makes sense (for instance, if the user changes a color)
  - Used when the user reloads the experiment otherwise (for instance, if the user changes the size of the grid)

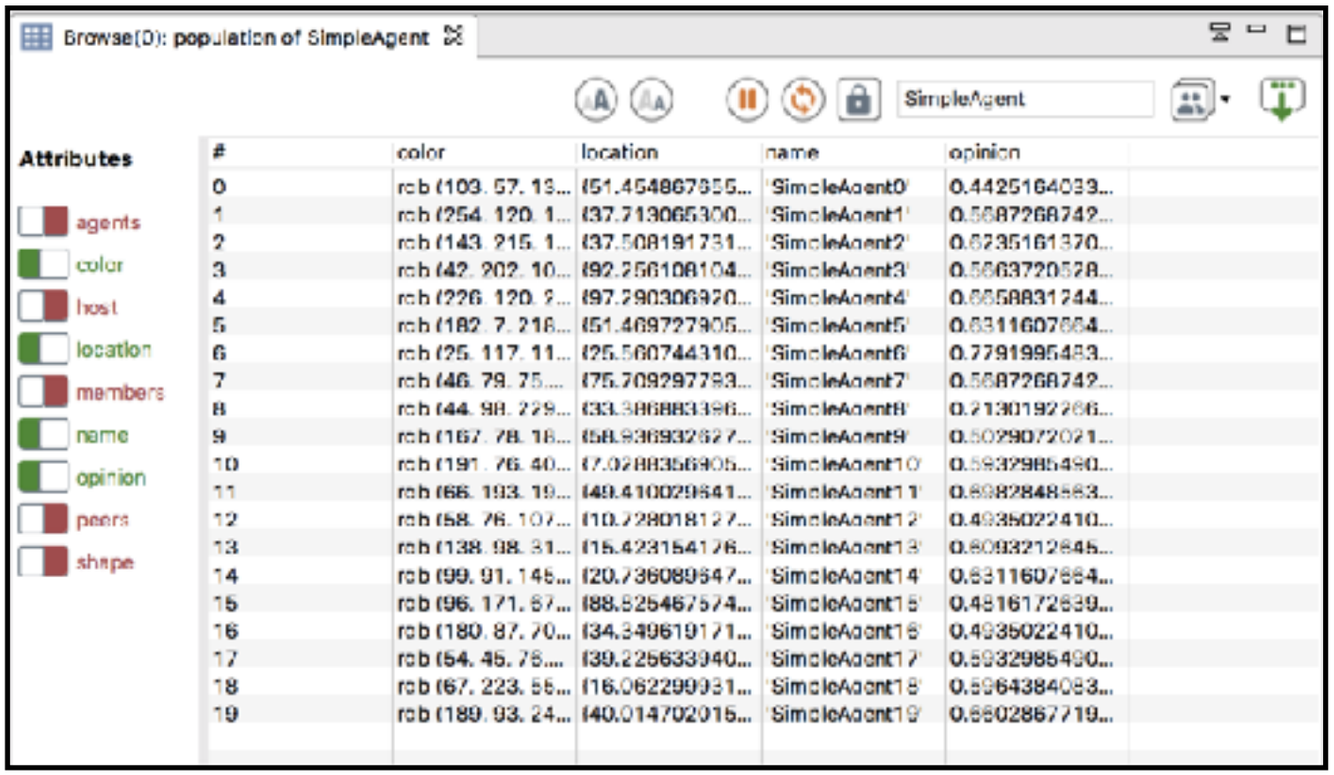
- Launching experiment again (from the model editor) will erase the modifications.



GAMA offers 2 views that display information about one or several agents



agent inspector



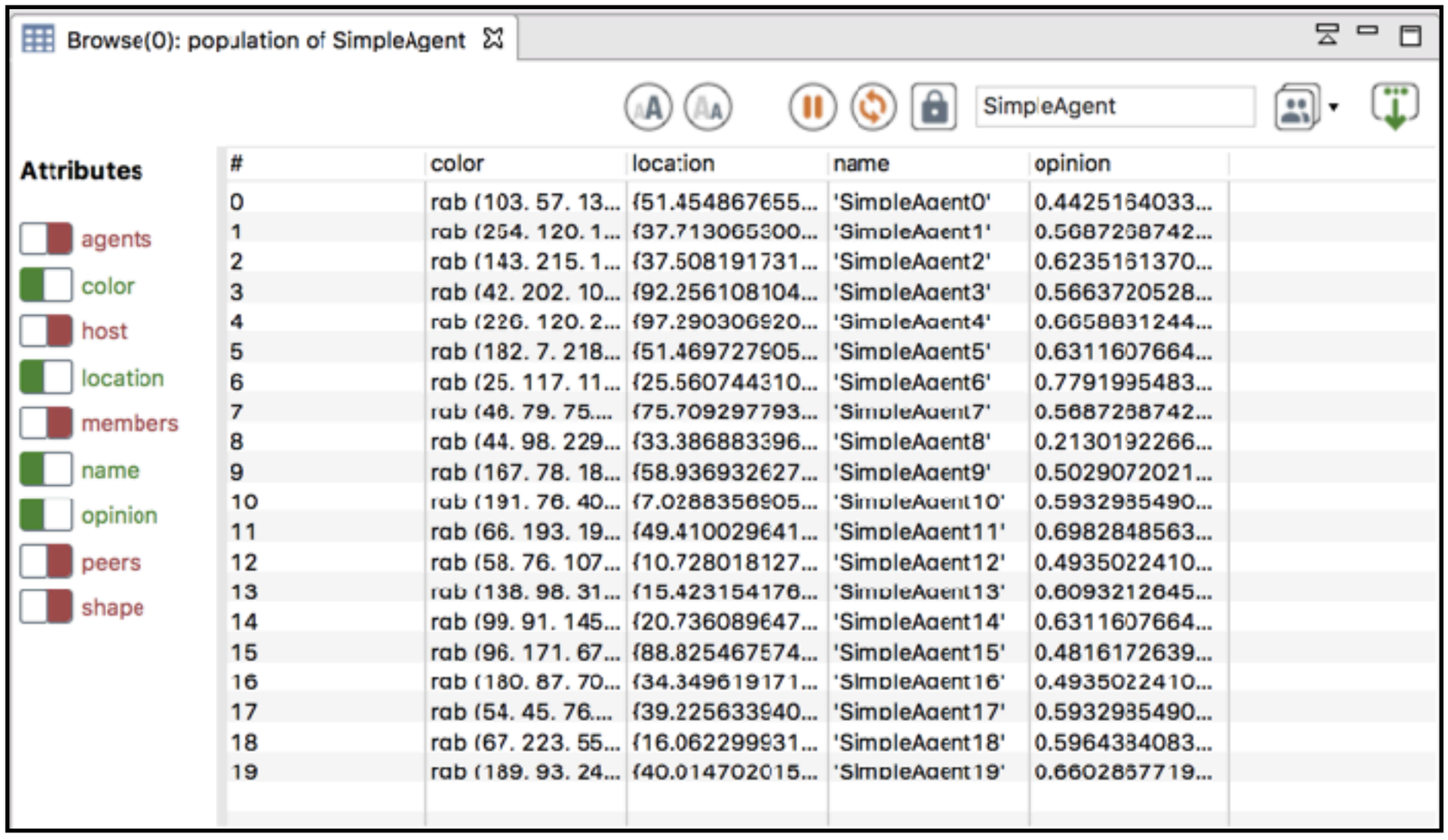
agent browser

Inspect by right clicking on a agent in a display

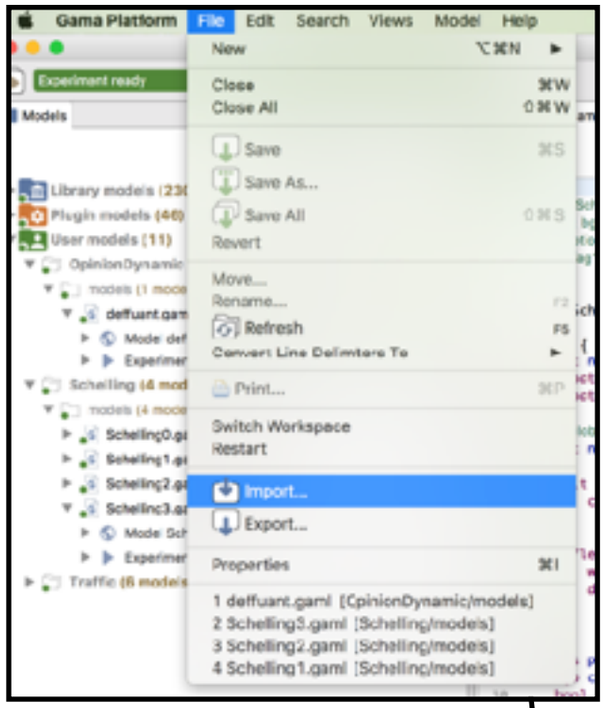
- provides information about one specific agent. It also allows to change the values of its variables during the simulation.
- It is possible to «highlight» the selected agent.

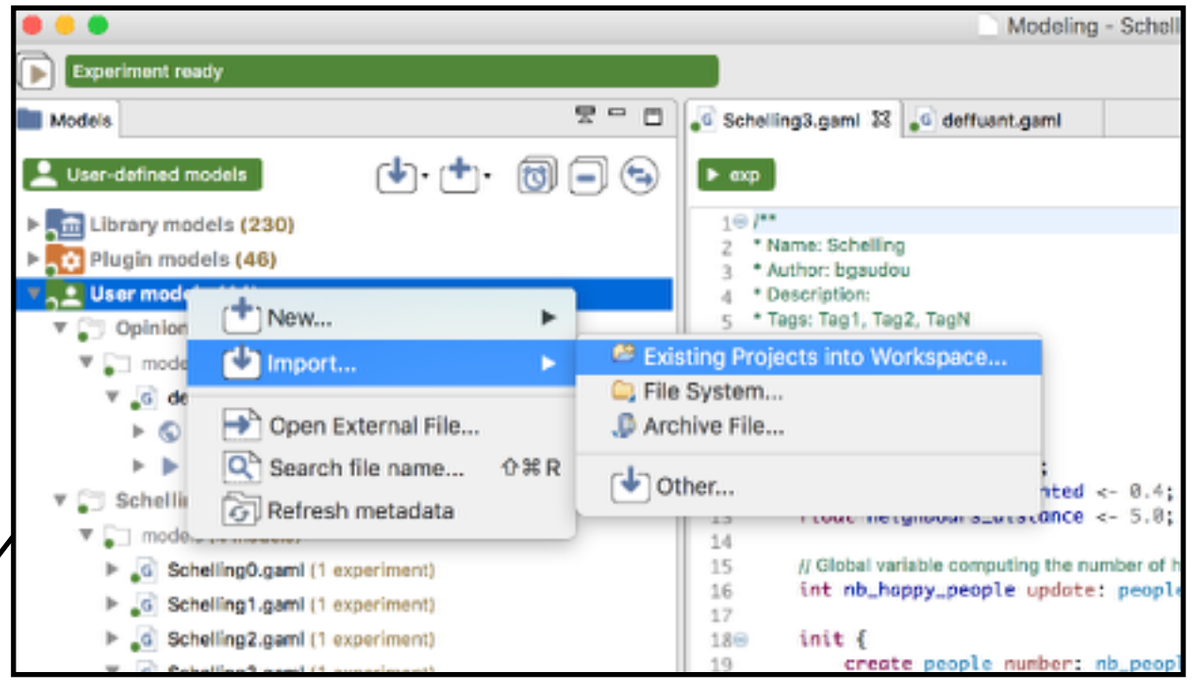
Inspect informations by agent browser

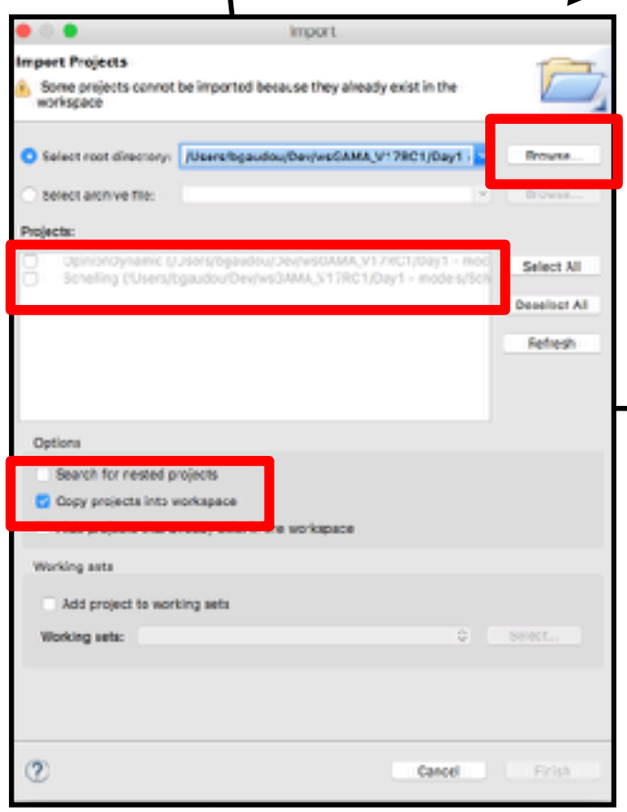
- The species browser provides informations about all or a selection of agents of a species.
- The agent browser is available through the **Agents** menu or by right clicking on a by right_clicking on a display



Import existing projects into the workspace

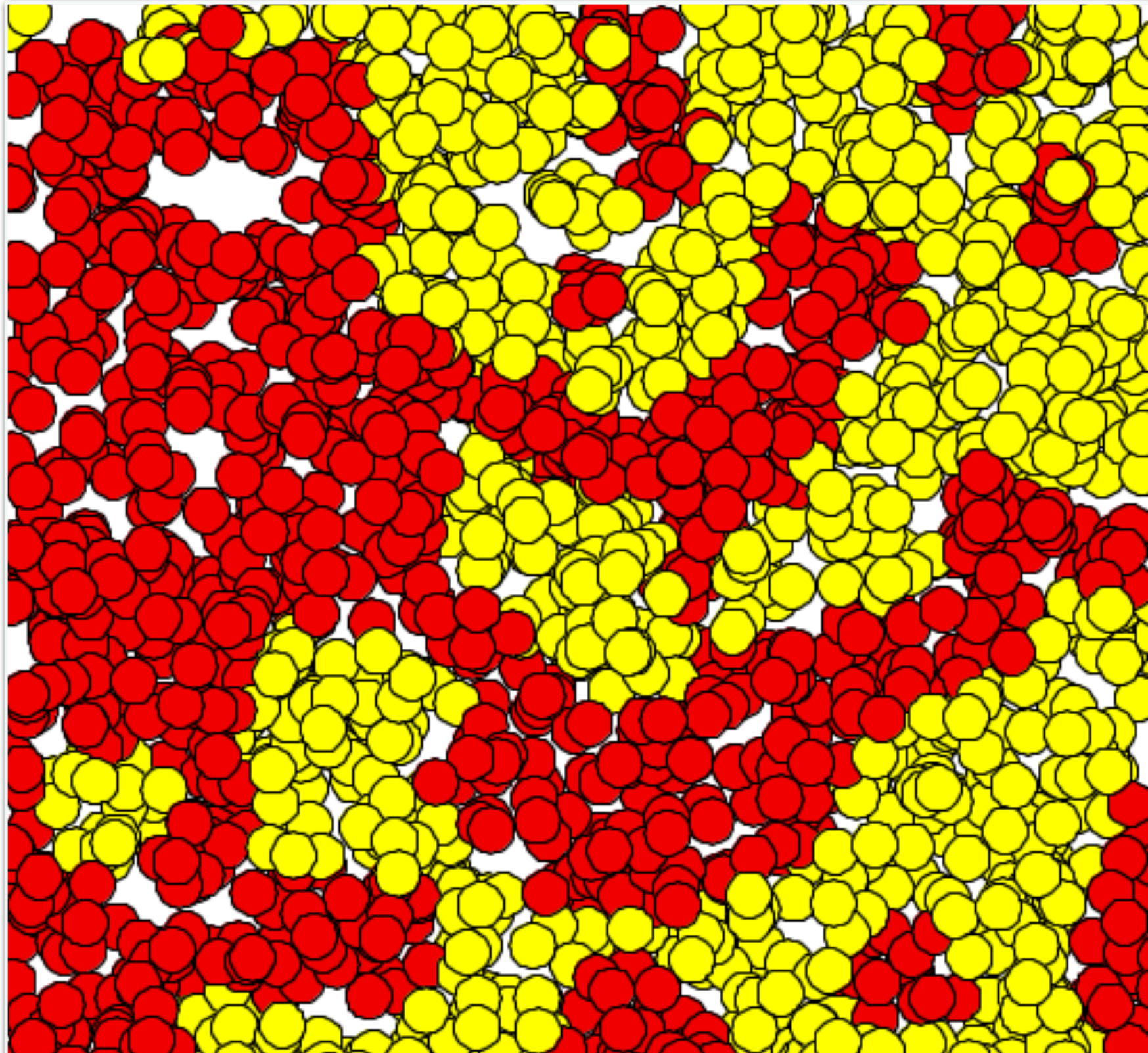






Imported projects are now in the User models

Introduction toGAMA through an example: Segregation Model



Urban Segregation Model proposed by Schelling

- In 1969, Schelling introduced a model of segregation in which individuals of two different colours, positioned on a grid abstract representation of a district), choose where to live based on a preferred percentage of neighbours of the same colour.

- Using coins on a board, he showed that a small preference for one's neighbours to be of the same colour could lead to total segregation.

- It is a good example of a generative model, where the emergence of a phenomenon here, segregation) is not directly predictable from the knowledge of individual

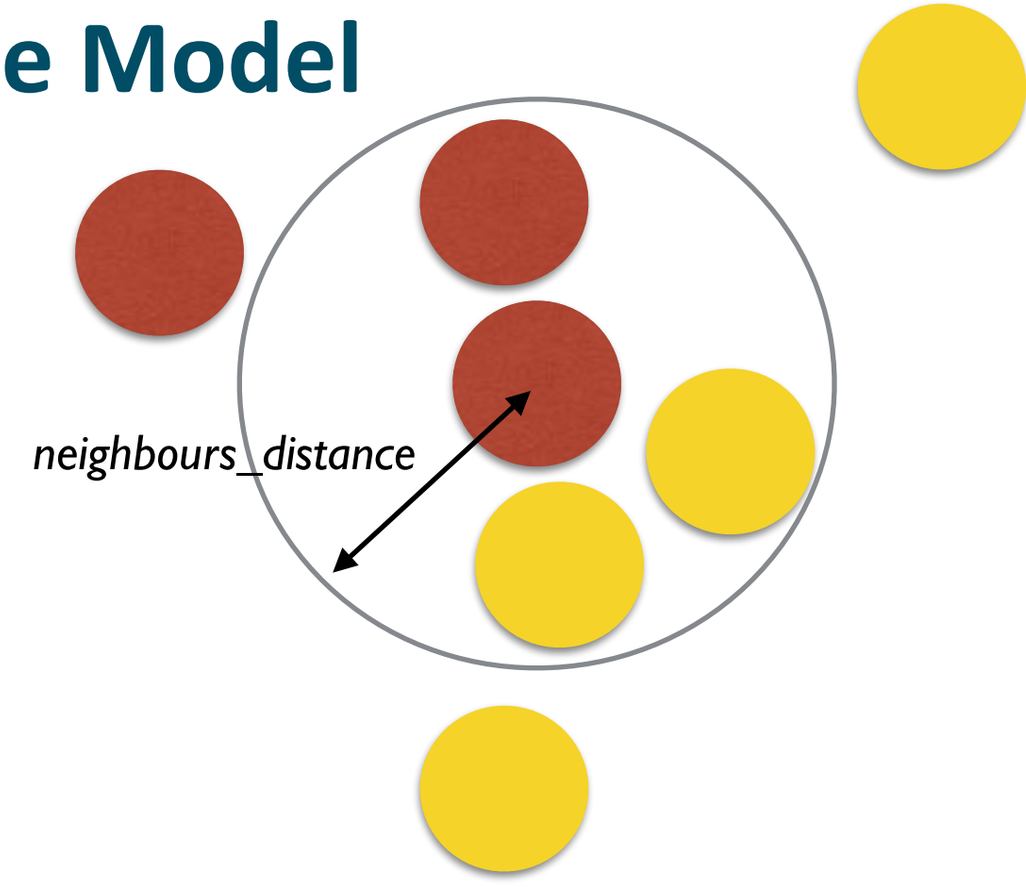
Proposed implementation of the Model

- ***People* agents** of 2 different colors (red and yellow) live in a continuous environment

- At each simulation step, each people agent:

  ➡ computes if it is happy: it is happy if the rate of people agents at a distance *neighbours_distance* of the same color is higher or equals to the threshold *similar_rate_wanted*

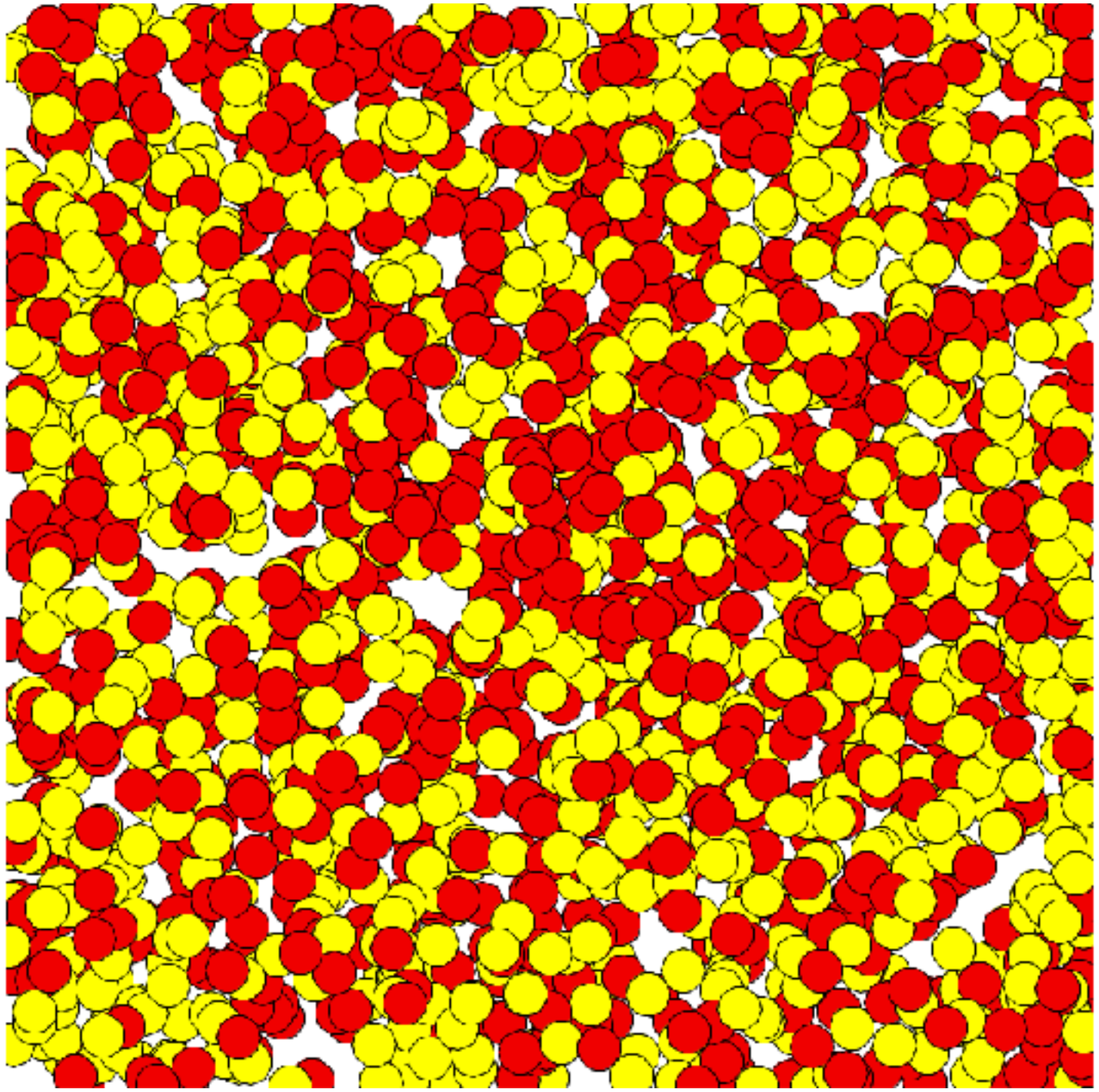
  ➡ if it is not happy, it moves to a random location



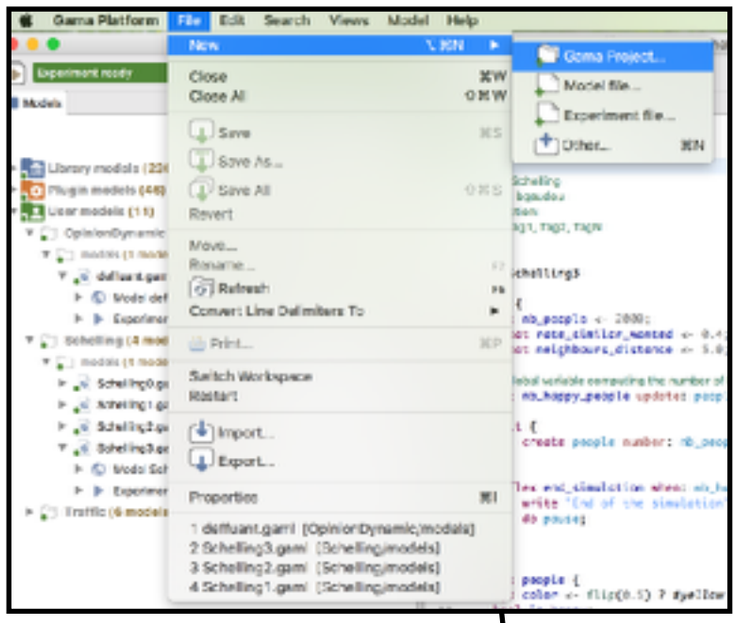
Similar_rate = 1/3 = 0.333
happy if similar_rate >= *similar_rate_wanted*

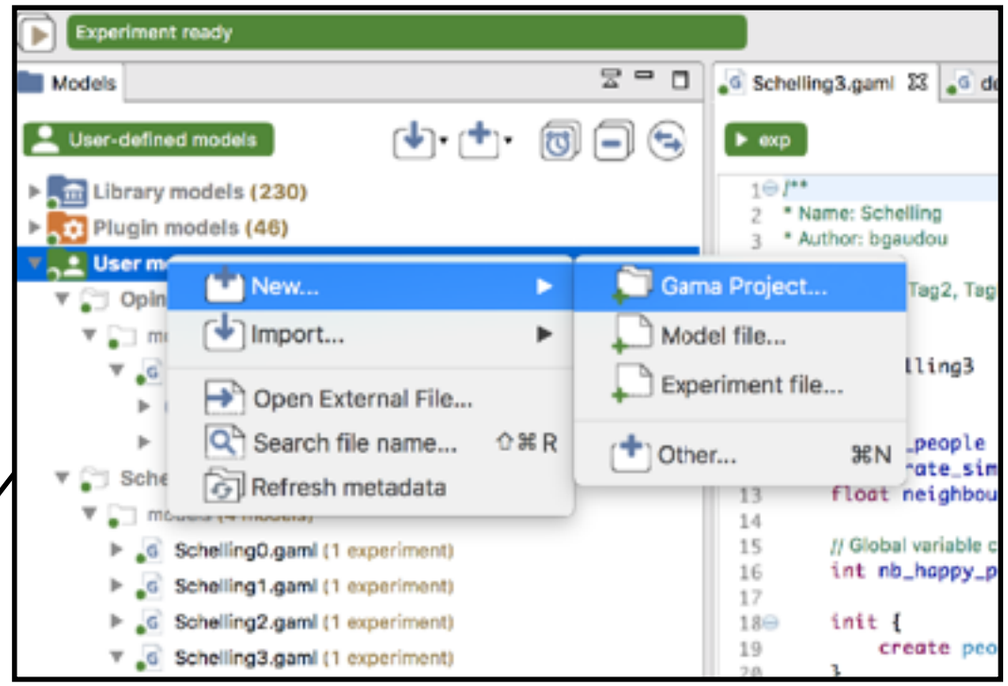
Step 1: definition and display of the people species

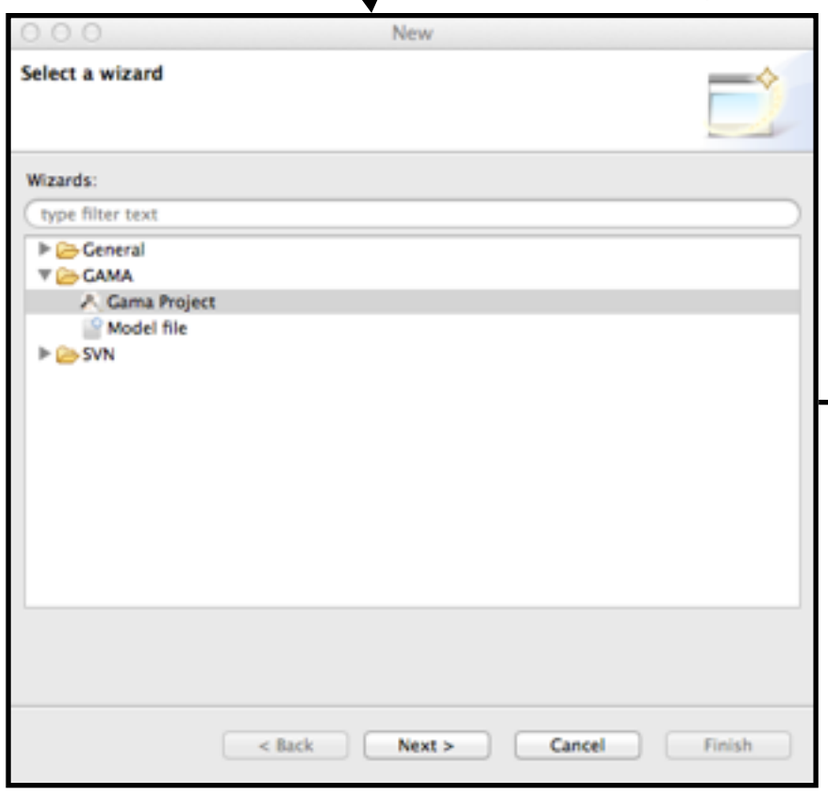
- Objectives:
  - Definition of the *people* species
  - Creation of *2000* people agents randomly located in the environment
  - Display of the agents

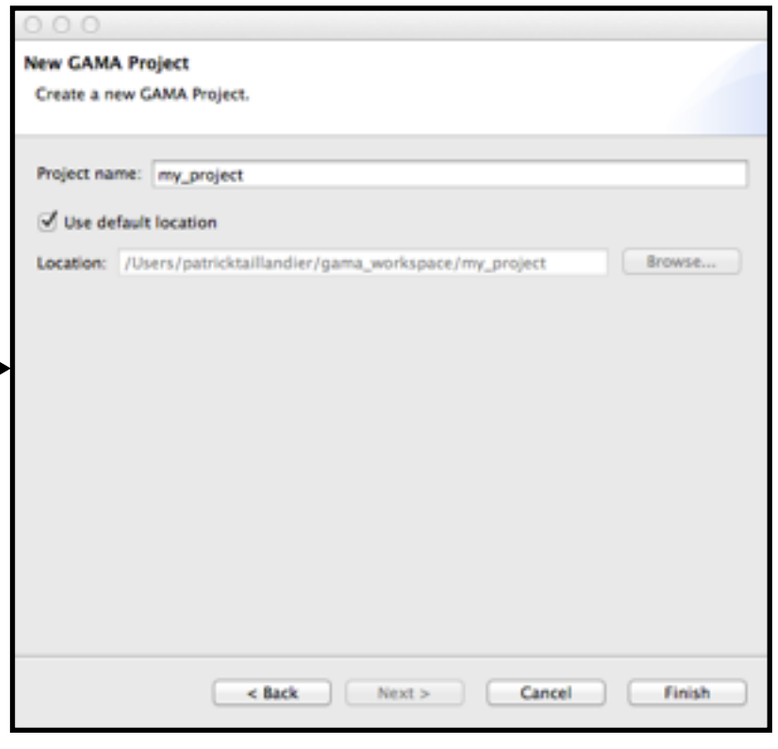


Creation of a new project

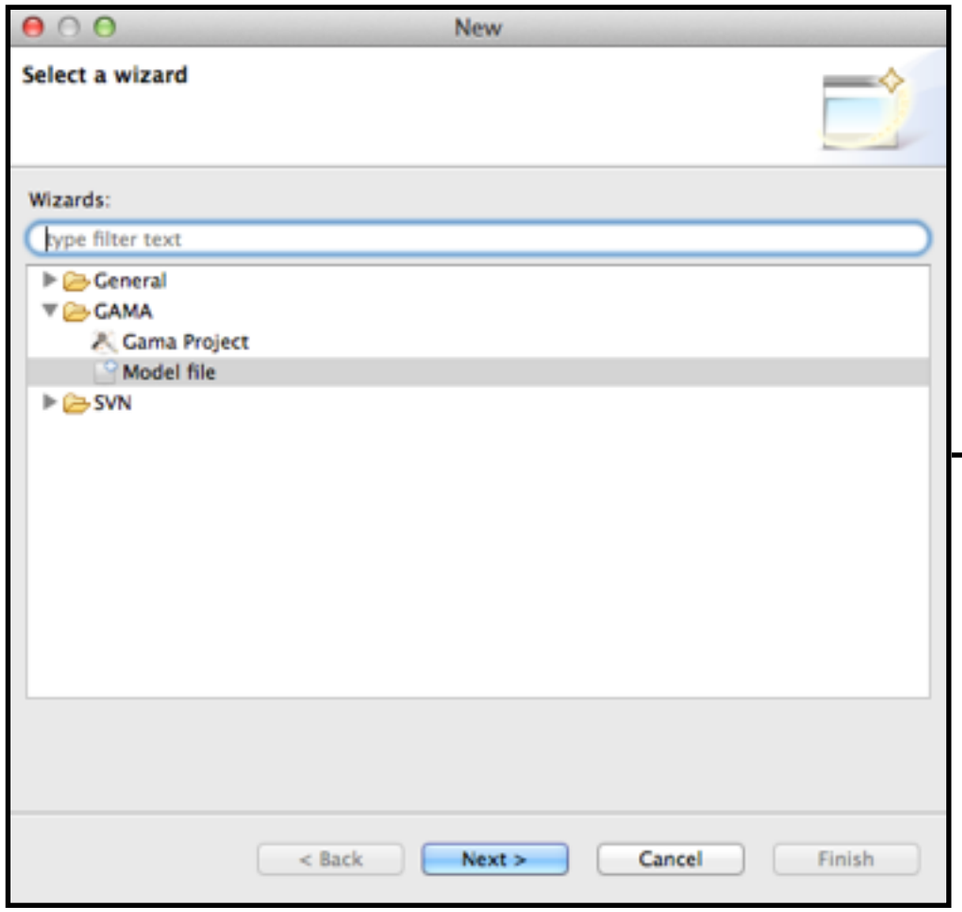


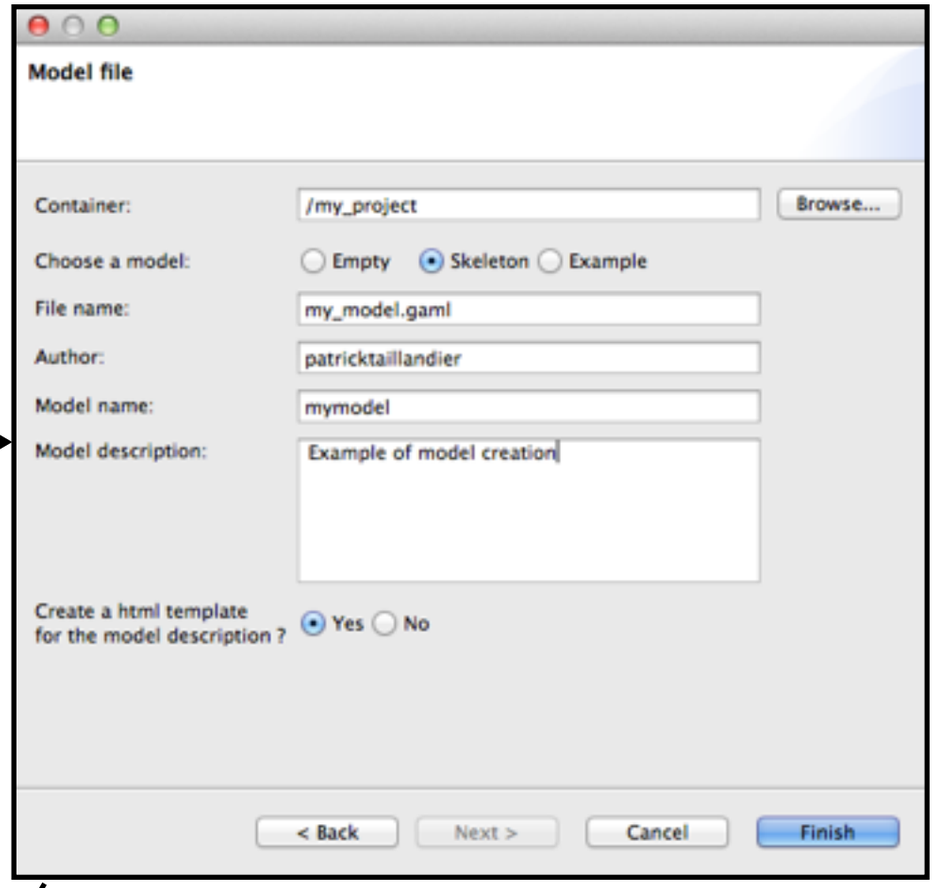






Creation of a new model file






```
model mymodel

global {
	/** Insert the global definitions, variables and actions here */
}

experiment mymodel type: gui {
	/** Insert here the definition of the input and output of the model */
	output {
	}
}
```

Introduction to the GAMA Modelling Language - GAML

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

- Agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they know*), **actions** (*what they can do*), **behaviours** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology**

- **Everything is an agent** in GAML: the model itself (called the *world*), the agents defined in it, the experiments…

Therefore, the structure of a model in GAML is simply a set of *species declaration statements*

- 3 types of species declaration statements are supported:
  - **Global (unique)**: global attributes, actions, dynamics and initialisation.
  - **Species** and **Grid**: agent species. Several species statements can be defined in the same model.
  - **Experiment** : simulation execution context, in particular inputs and outputs. Several experiment statements can be defined.

```
model my_model

global {
    /** Insert the global definitions,
     * variables and actions here
     */
}

species my_species{
    /** Insert here the definition of the
     * species of agents
     */
}

experiment my_model type: gui {
    /** Insert here the definition of the
     * input and output of the model
     */
}
```

A statement represents either a declaration or an imperative command

- It consists in a **keyword**, followed by a list of **facets** (some of them mandatory), ended by ";" or a **block** of statements.

- A **facet** is a keyword, followed by ":", and an **expression**. Note that the keyword of the first facet can usually be omitted. If the statement is a declaration, the first facet contains an **identifier.**

- A **block** is a set of statements enclosed into curly brackets ("{" and "}")

2 ways to write commentaries (texts that are not just part of the model but here for information purpose):

- //… : for one line. Example : //this is a commentary
- /* … */ : can be used for several lines. Example : /* this is as well a commentary */

A species declaration statement begins with a special keyword and is composed of several declarative statements

- Syntax:

keyword identifier {
    [Attributes declaration]
    [Action declaration]
    [Behaviours declaration]
    [Aspect declaration]
}

where **keyword** is either *global*, *species*, *grid* or *experiment*. The identifier is not used for *global*.

```
species my_species{
   string a_variable;
   action do_something{...}
   reflex a_behaviour{...}
   aspect my_representation{...}
}
```

A typical model structure

```
Model new
global{
// attribute declaration
// action declaration
// behaviour declaration
// aspect declaration
}

species my_species{
// attribute declaration
// action declaration
// behaviour declaration
// aspect declaration
}

experiment my_experiment type:gui{
// attribute declaration
// action declaration
// behaviour declaration
// aspect declaration
// output declaration
}
```

Only in experiments, output declarations allow to define how the outcomes of simulations should be saved or displayed

Segregation model 1: People Species definition

- **To do :** define the species *people*:

- **Solution :**

```
species people {
}
```

Species can declare attributes (*what their agents know)* by using *declarative statements* beginning with a data type name.

- Syntax:

  data_type **identifier;**

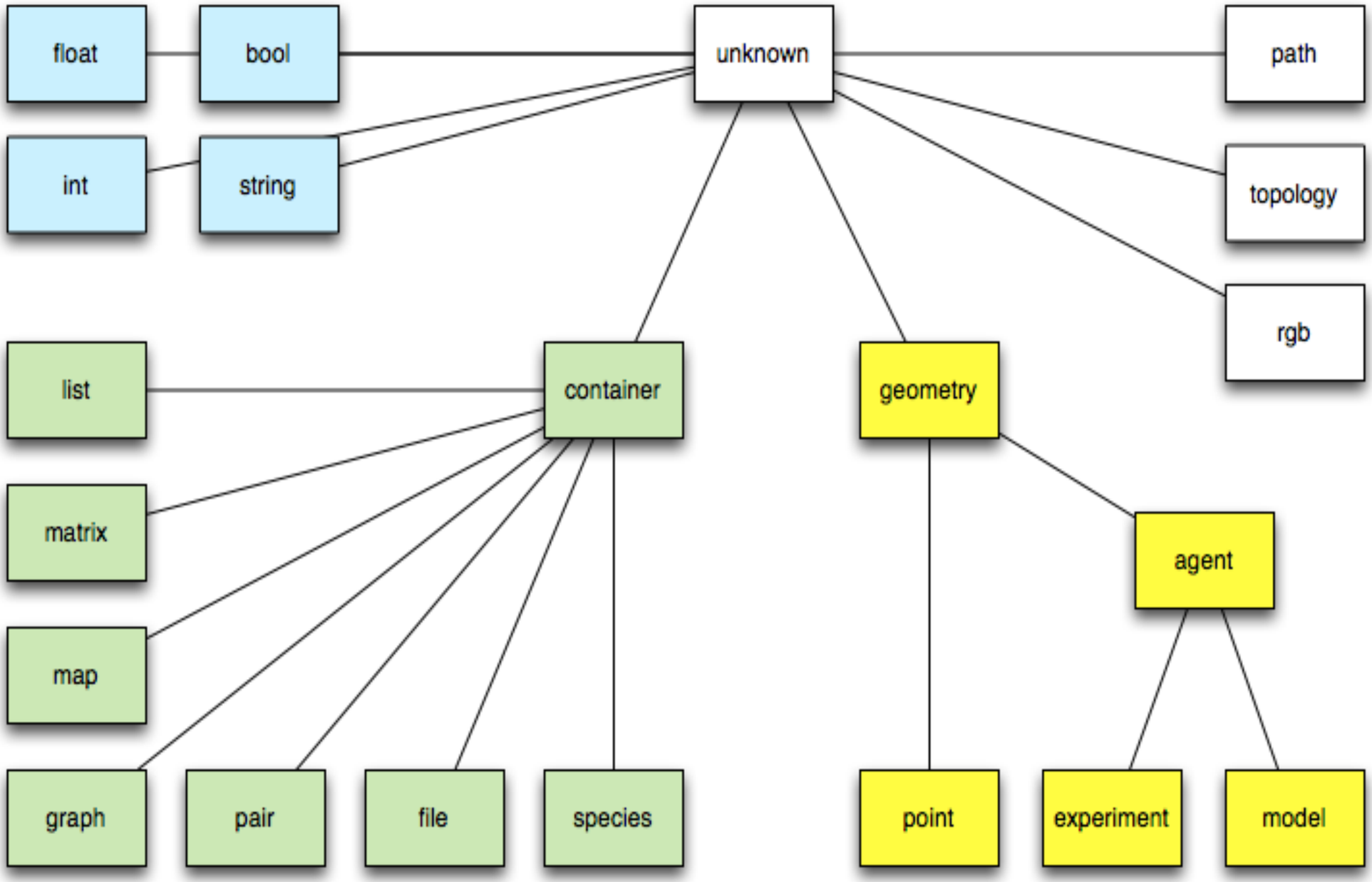
- *data_type* refers to the name of a built-in type or a species declared in the model.
- The **identifier** can be any combination of letters and digits (plus the underscore, "_") that does not begin with a digit.

- Optional facets:
  - <- : initial value
  - update : value computed at each simulation step
  - function or -> + {..} : value computed each time the variable is called
  - min : min value
  - max : max value

All GAMA agents are provided with some built-in attributes :

- ***name*** (*string*)
- ***shape*** (*geometry*) 
- ***location*** (*point*) : centroid of its shape

The hierarchy of data types in GAML is the following:



Examples of data types in GAML

Primitives types:

- Integer:

```
int density <- 25 min: 1 max: 99;
```

- Float

```
float speed of agents min: 0.1  <- 2.0 ;
```

- Bool

```
bool state <- (rnd(100)) < density;
```

- String

```
string message <- "This is a message";
```

Complex types:

- List: a composite datatype holding an ordered collection of values

```
list a_list_of_numbers<-[0,1,2,3];
```

- Map: a composite datatype holding an ordered collection of pairs

```
map pair_of_key_and_value<-["key"::5];
```

- Matrix: represents either a two-dimension array (matrix) or a one-dimension array (vector), holding any type of data (including other matrices).

```
matrix mat_2_3 <- ([
    ["c11","c12","c13"],
    ["c21","c22","c23"]
]);
```

Segregation model 1: People Species attribute definition

- **To do :** define two attributes for the species *people*:
  - ➡ **color** : type: *rgb* (colour), init value (<-): *red* with a probability of *0.5*, *yellow* otherwise
  - ➡ **is_happy**: type: *bool*, init value: *false*

- **Solution :**

```
species people {
    rgb color <- flip(0.5) ? #red : #yellow;
    bool is_happy <- false;
}
```

Define a colour attribute of type *rgb* (color) of which the initial value has a probability of 0.5 to be red or yellow

Note : the **flip***(proba)* operator to test a probability. i.e. returns true with a probability of *proba*

The symbol # allows also to define a colour

Species definition - *aspect*

- An *aspect* represent A possible display for a species of agents : ***aspect*** *aspect_name* ***{...}***

- In an *aspect* block, it is possible to display (as layers) using the ***draw*** statement:
  - a geometry/shape: for example, the agent shape
  - a image: for example, an icon
  - a text

Segregation model 1: People Species circle aspect

- **To do :** define a new aspect called *circle*:
  - ➡ draw a circle of radius *2m* with the colour *color*

- **Solution :**

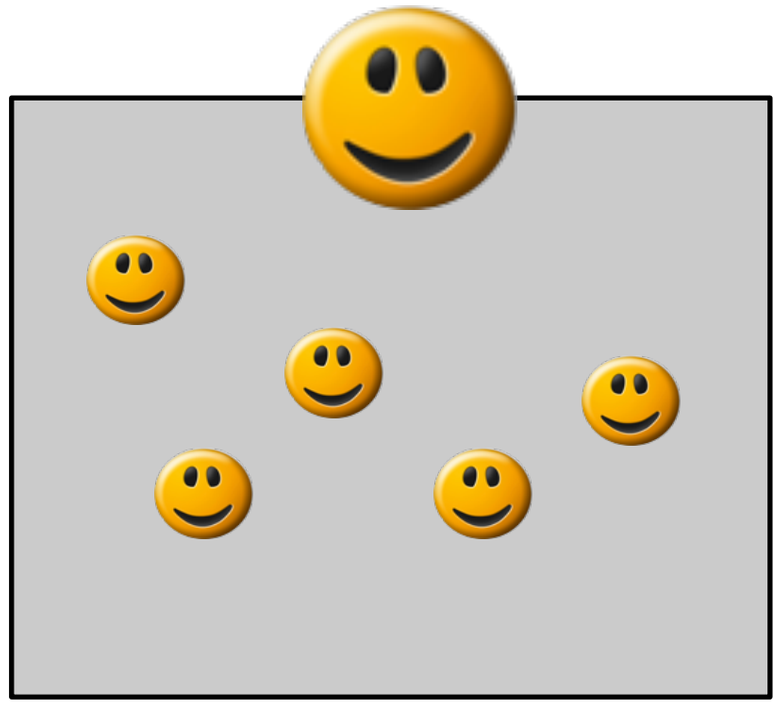
```
species people {
   //attribute definition

   aspect circle {
      draw circle(2) border: #black color: color;
   }
}
```

This aspect allow to display each people agent as circle of radius 2m using the *color* attribute to define its colour

Now, we have finished to define the people species that has several attributes and that can be displayed as a circle.
Next step: creation of people agents !

global block

- Global block
  - Define a specific agent (called *world*)
  - Represent all that is global to the model:
    attributes, actions, reflexes....
  - Initialise the simulation (*init* block): when the experiment button is pushed, the world agent is created and then activates its *init* block
- The geometry of the world agent (*shape*) defines the size of the environment in which all the agents are localised. By default the shape of the world agent is a square of 100m size
- Define the nature of the environment:
  torus or not (by default, not torus)



Segregation model 1: Global attribute definition

- **To do :** define three attributes in the global section :
  - ➡ **nb_people** : type: *int* (integer), init value (<-): *2000*
  - ➡ **rate_similar_wanted**: type: *float*, init value: *0.4*
  - ➡ **neighbours_distance**: type: *float*, init value: *5.0*

- **Solution :**

```
global {
    int nb_people <- 2000;
    float rate_similar_wanted <- 0.4;
    float neighbours_distance <- 5.0;
}
```

Init block

- For each species, an init block can be defined
- Allows to execute a sequence of statements at the creation of the agents
- Activated only once when the agent is created, after the initialisation of its variables, and before it executes any reflex
- **Syntax:**

```
init{
    <<statements>>
}
```

- Only one instance of init per species

```
global {
    ............
    //Only executed when world agent is created

    init {
            write "Executing initialisation";
        }
}
```

Agent creation

- Creation of agents : use of the statement: **create** *species_name +*
  - *number* : number of agent to create (*int*, by default, 1)
  - *from* : GIS or Raster file (*string* or *file*)
  - *with* : allows to give initial values to the agent variables
  - *returns*: list of created agents

Note: By default, agents are **randomly** placed in the environment (except when the facet ***from:*** + *SIG* is used)

Note: The *create* statement can be used in all init/actions/ reflex of the model, not only in the global section

Segregation model 1: Global init

- **To do :** define a global init block that :
  - ➡ **create** *nb_people* people agents

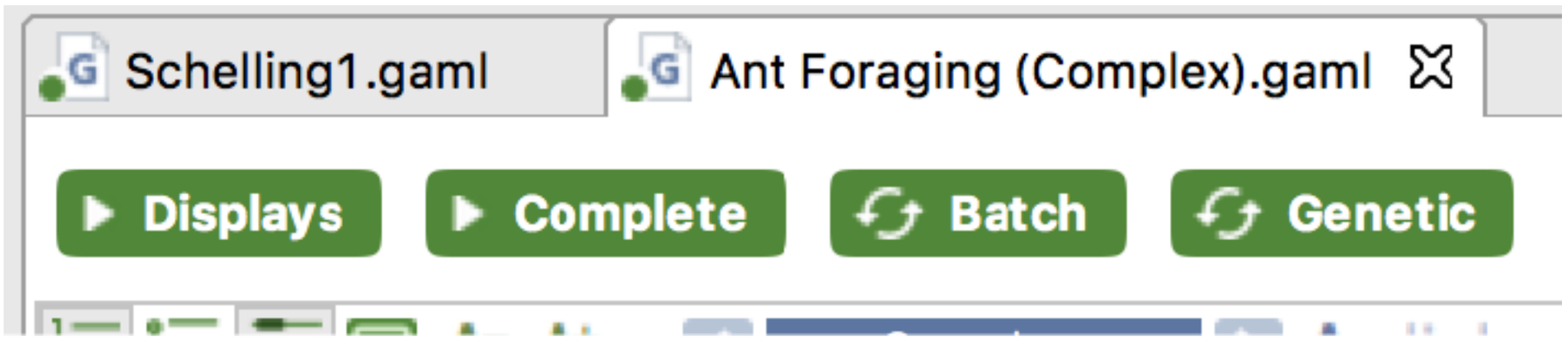
- **Solution :**

```
global {
    //Attribute definition
    init {
        create people number: nb_people;
    }
}
```

Now, we have finished to define global attributes and to create people agents.
Next step: add a display to view the people agents !

experiment block

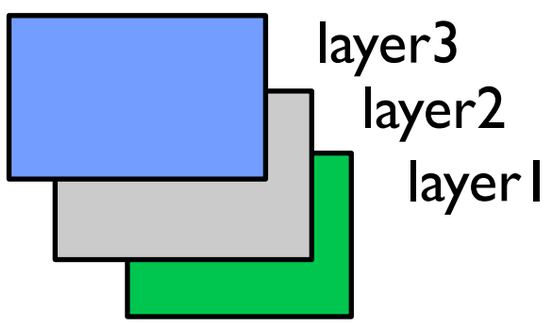
- An *experiment* is a specific agent that allows define the execution context of simulations, in particular the parameters and outputs of the simulation
- Several experiment can be defined (one experiment bouton is created per experiment)
- Define by : **experiment** *xp_name* **type:** *gui*/*batch* **{…}**
  - *gui* : one simulation with graphical interface.
  - *batch* : experiment plan: set of simulations without graphical interface



experiment block: output definition

- The *output* block has to be defined in an *experiment* block

- It allows to define displays:
  - A refreshing rate can be defined: facet **refresh:** *nb (int)*
  - Each *display* can contain different displays:
    - list of agents :
      agents *layer_name* **value:** *agents* **aspect:** *my_aspect;*
    - Agent species (all the agents of the species) :
      species *my_species* **aspect:** *my_aspect*
    - Grids: optimised display of grids:
      grid *grid_name* **lines**: *my_color;*
    - Images:
      image *layer_name* **file:** *image_file*;
    - Charts: see later



Segregation model 1: Output definition

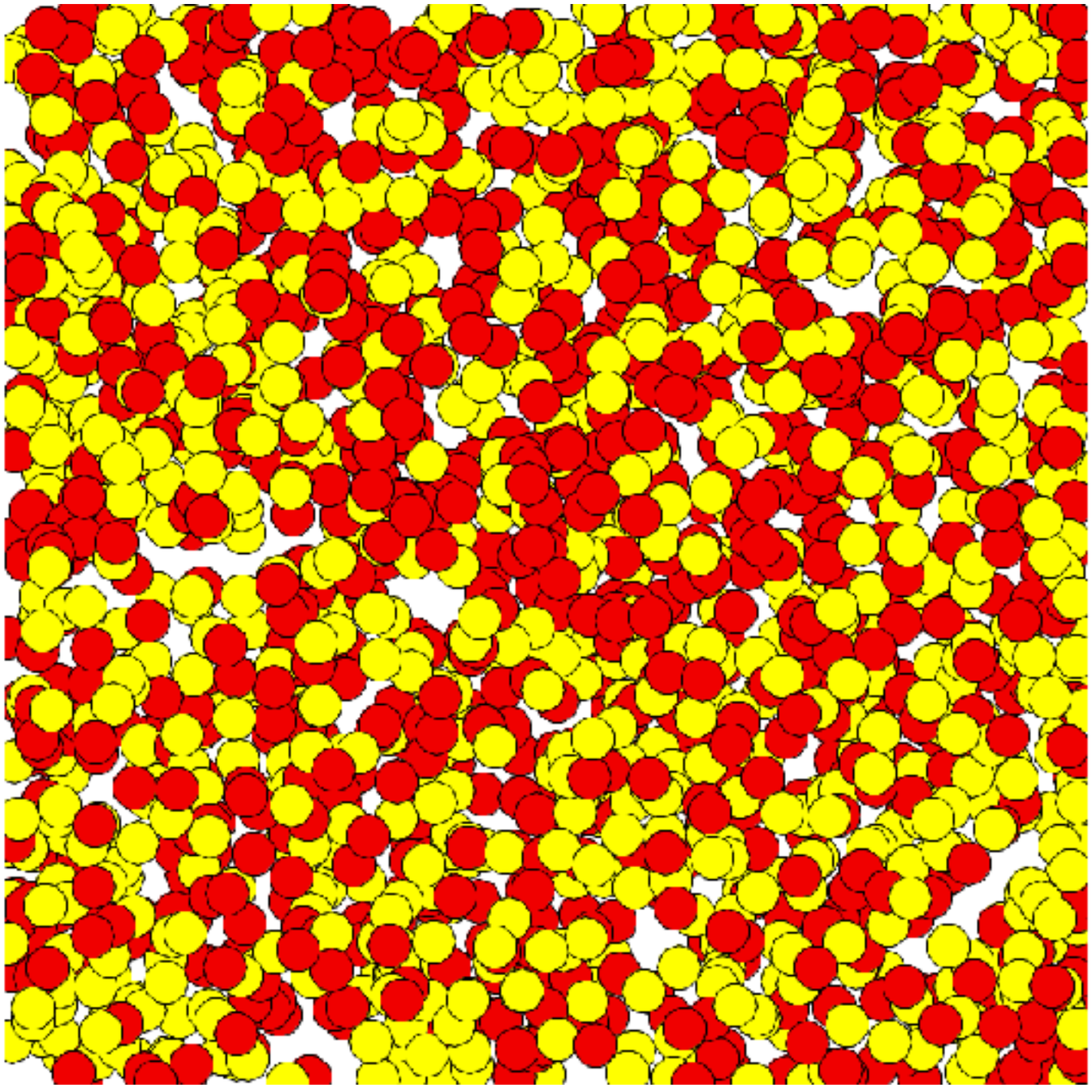
- **To do :** define a display called *map* that displays :
  - ➡ The *people* agents with their aspect *circle*

- **Solution :**

```
experiment main_xp type: gui {
    output {
        display map {
            species people aspect: circle;
        }
    }
}
```

Now, we have finished to define a display to view the people agents.
Next step: Test the model !

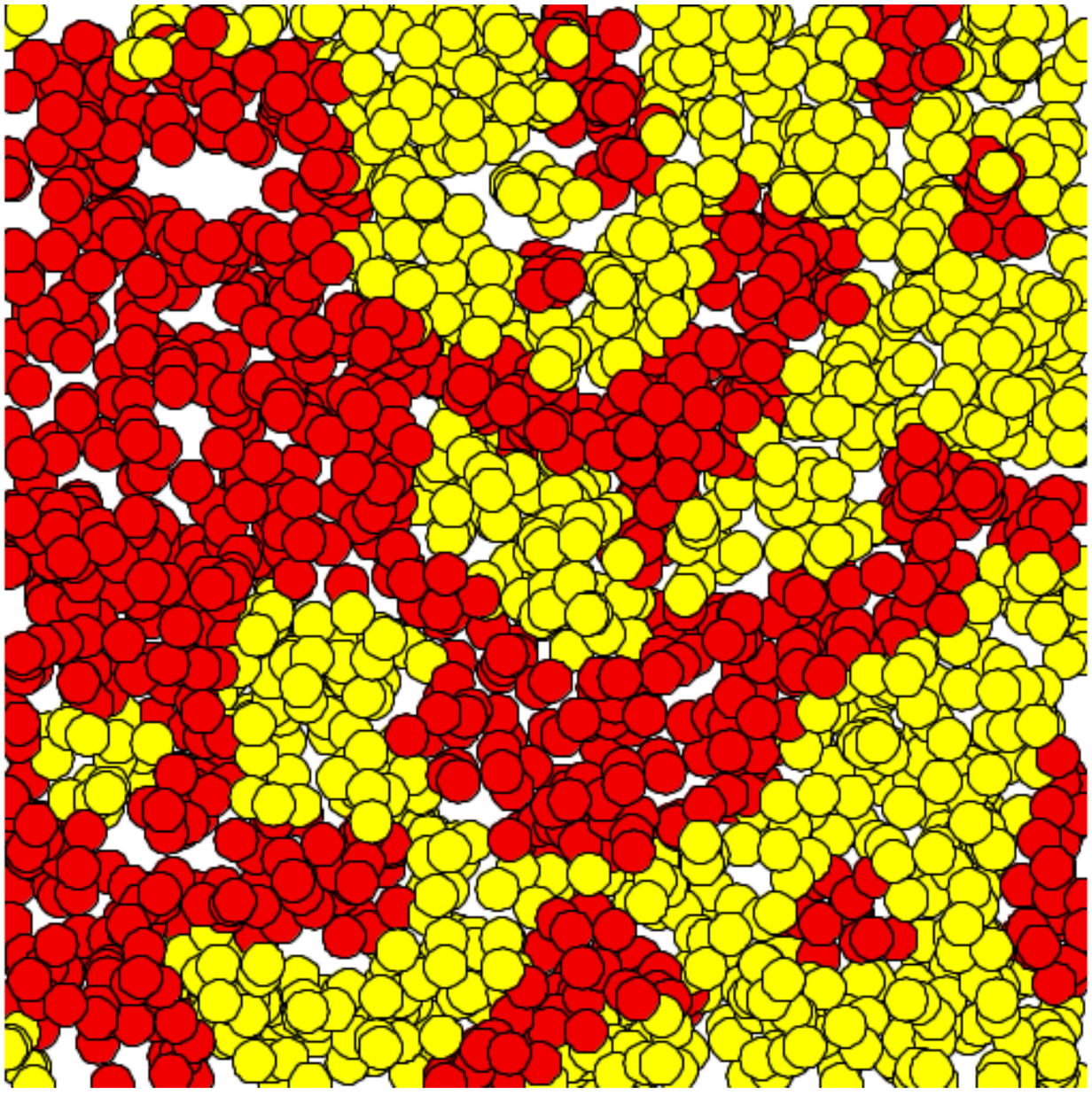
End of step 1



Now it's time to define the dynamic of the models !

Step 2: definition of the people agent dynamic

- Objectives:
  - ➡ Definition of the neighbours attribute for the people agents
  - ➡ Definition of a computing neighbours similarity behaviour for the people agents
  - ➡ Definition of a moving behaviour for the people agents



Species are provided with a simple behavioural structure, based on reflexes (*what they actually do*)

- A reflex is a sequence of statements that can be executed, at each time step, by the agent.

reflex name **when**: condition{

[statements]

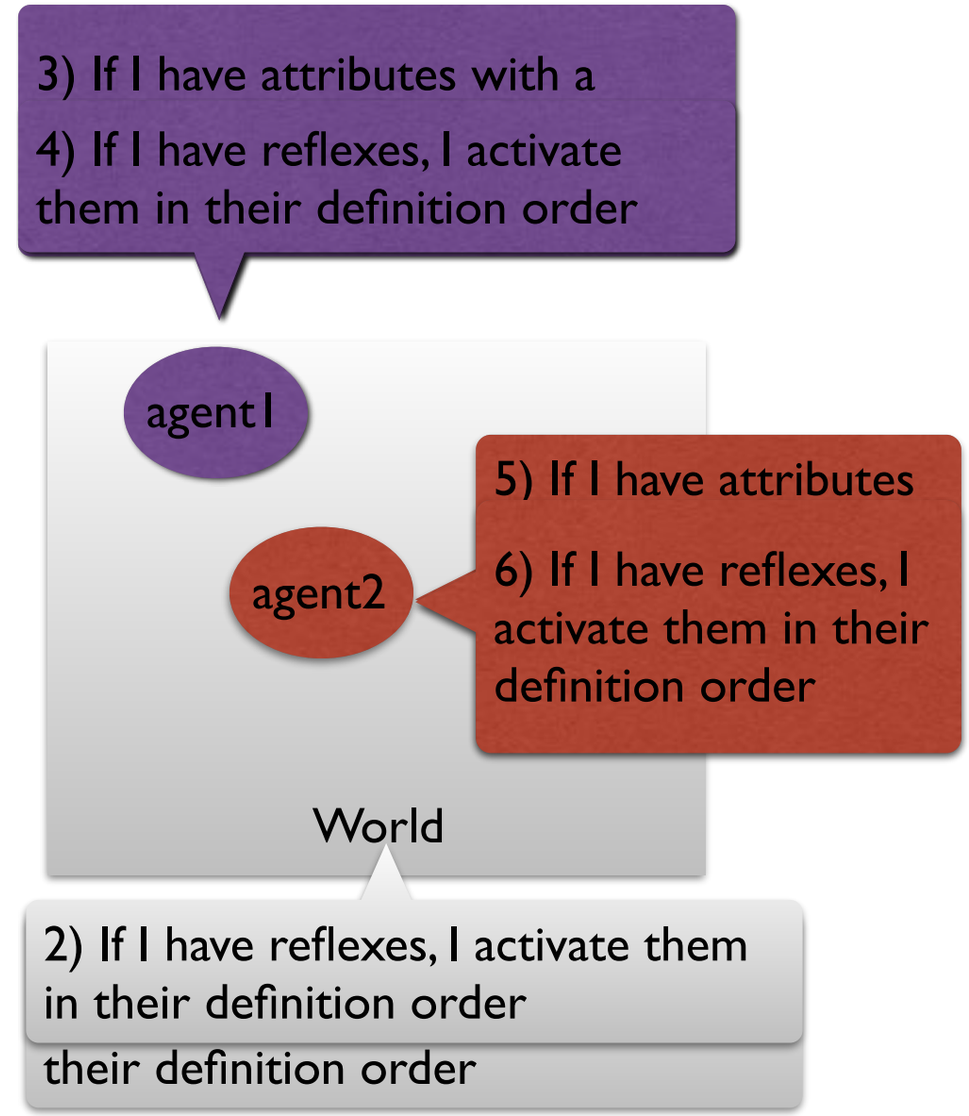
}

- If no facet **when** are defined, it will be executed every time step.
- If there is one, it is executed only if the boolean expression evaluates to true.
- It is a convenient way to specify the behaviour of the agents.

Note: The init block is a specific reflex that is activated only once at the creation of the agent

The scheduler of GAMA

- The basic scheduler of GAMA works as follows:
  - GAMA activates the world agent (global) then all the other agents according to their order of creation
  - When an agent is executed, first its update its attributes (facet **update** of the attributes), then it activates its reflexes in their definition order

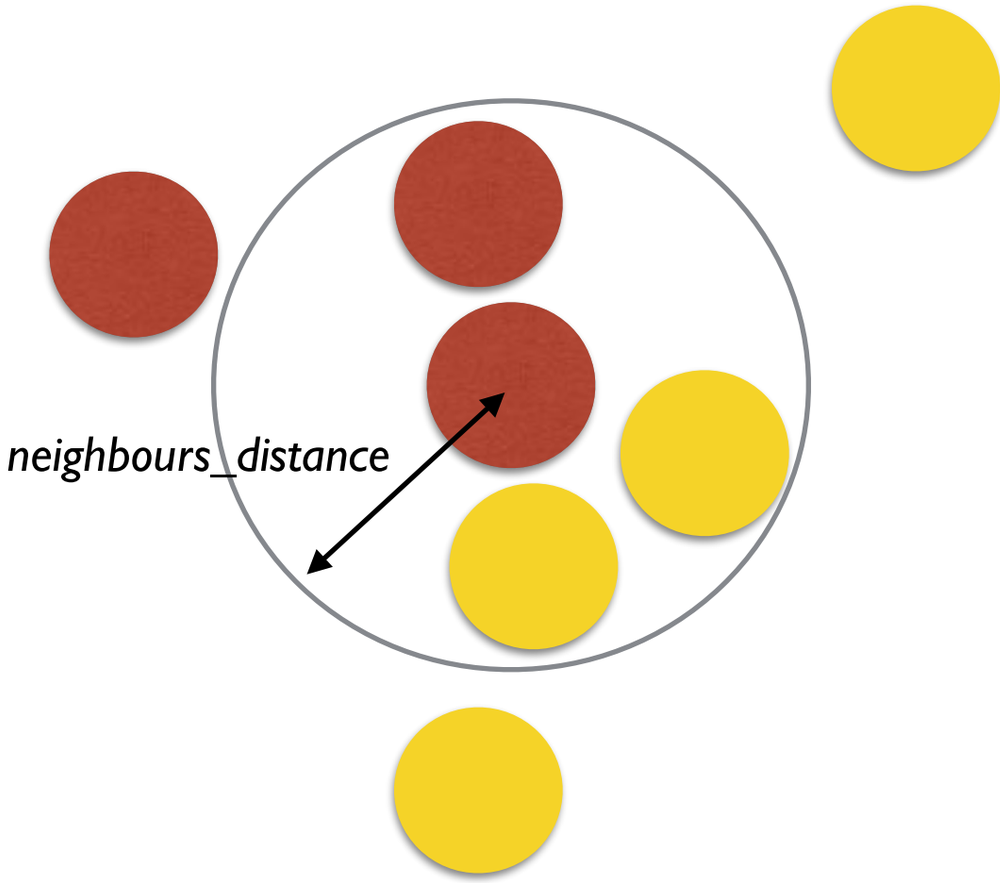


- Of course the Scheduler can be easily tuned through the GAML language:
  - modification of the order of activation of the agents (than can be dynamic)
  - Fine activation of the agents using actions (e.g.: agent1 executes a first action, then agent2 executes an action, then agent1 executes again another action….)

Note: GAMA offers some specific control architectures (finite state machine, tack-oriented architectures…) that can be added to species

Segregation model 2: Neighbours definition for people species

- **To do :** define an attribute for the people species called *neighbours*:
  - ➡ Type: list of *people* agents; update at each simulation step with the *people* agents that are at a distance lower or equal to *neighbours_distance*



- **Solution :**

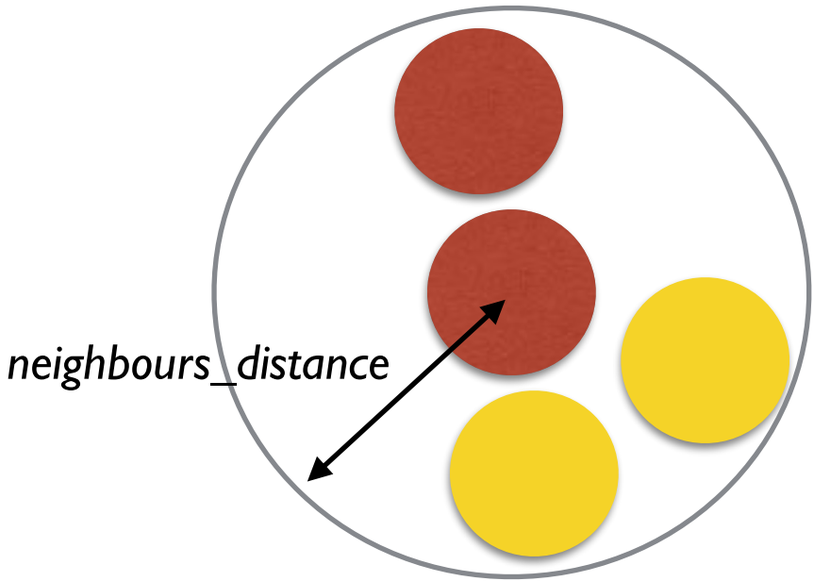
```
species people {
    //other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    //aspect definition
}
```

Note 1: the « *a_container* ***at_distance*** *distance* » operator returns all the agents from *a_container* (that can be a species or a list of agents) that are at distance lower or equal to *distance* to the current agent (the one using the operator).

Note 2: the « <> » symbol allows to specify the type of elements contained in a list

Segregation model 2: Computing Similarity reflex for the people species

- **To do :** define a reflex called *computing_similarity* for the people species:
  - ➡ if the neighbours is empty, set the rate_similar to 1.0
  - ➡ Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the rate_similar to the number of similar neighbours divided by the number of neighbours



Similar_rate = 1/3 = 0.333
happy if similar_rate >= *similar_rate_wanted*

What we need here:

- test statement (if-else)
- modifying the value of an attribute
- local variables (to make the code more readable)
- list operators (is empty, nb of elements…)

Introduction to some classic GAML Statements

- **Test statement**

  if (my_condition1) {…}

  else if (my_condition2) {…}

  …

  else {…}

```
if (my_money <= 50000) {
    write “I am poor";
} else if (my_money > 50000 and my_money < 200000)
{
    write “I am ok";
} else {
    write “I am rich";
}
```

- Attribute value assignment statement

my_attribute **<-** *new_value;*

```
my_money <- 100000;
```

- Definition of local variables: variables that just exist inside a block. These variables are delete from the computer memory at the end of the block

type my_local_variable **<-** *init_value;*

```
int nb_month <- 12;
```

Segregation model 2: Compute Similarity reflex for the people species

- **To do :** define a reflex called *compute_similarity* for the people species:
  - ➡ if the neighbours is empty, set the rate_similar to 1.0
  - ➡ Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the *rate_similar* to the number of similar neighbours divided by the number of neighbours
- **Solution :**

```
species people {
    //attributes
    reflex computing_similarity {
        float rate_similar <- 0.0;
        if (empty(neighbours)) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }

    //aspect definition
}
```

returns true if the list *neighbours* is empty

nb of elements in the list *neighbours*

count the number of elements (from the list *neighbours* for which the colour is equal to the current agent colour

is_happy is set to true if rate_similar (computed just before) is higher or equal than the global variable *rate_similar_wanted*

Note: for list operators, the keyword *each* represents each element of the list

Segregation model 2: Moving reflex for the people species

- **To do :** define a reflex called *moving* for the people species:
  - ➡ activated only if the agent is not happy (*not is_happy*)
  - ➡ The agent chances its location by an random location in the world
- **Solution :**

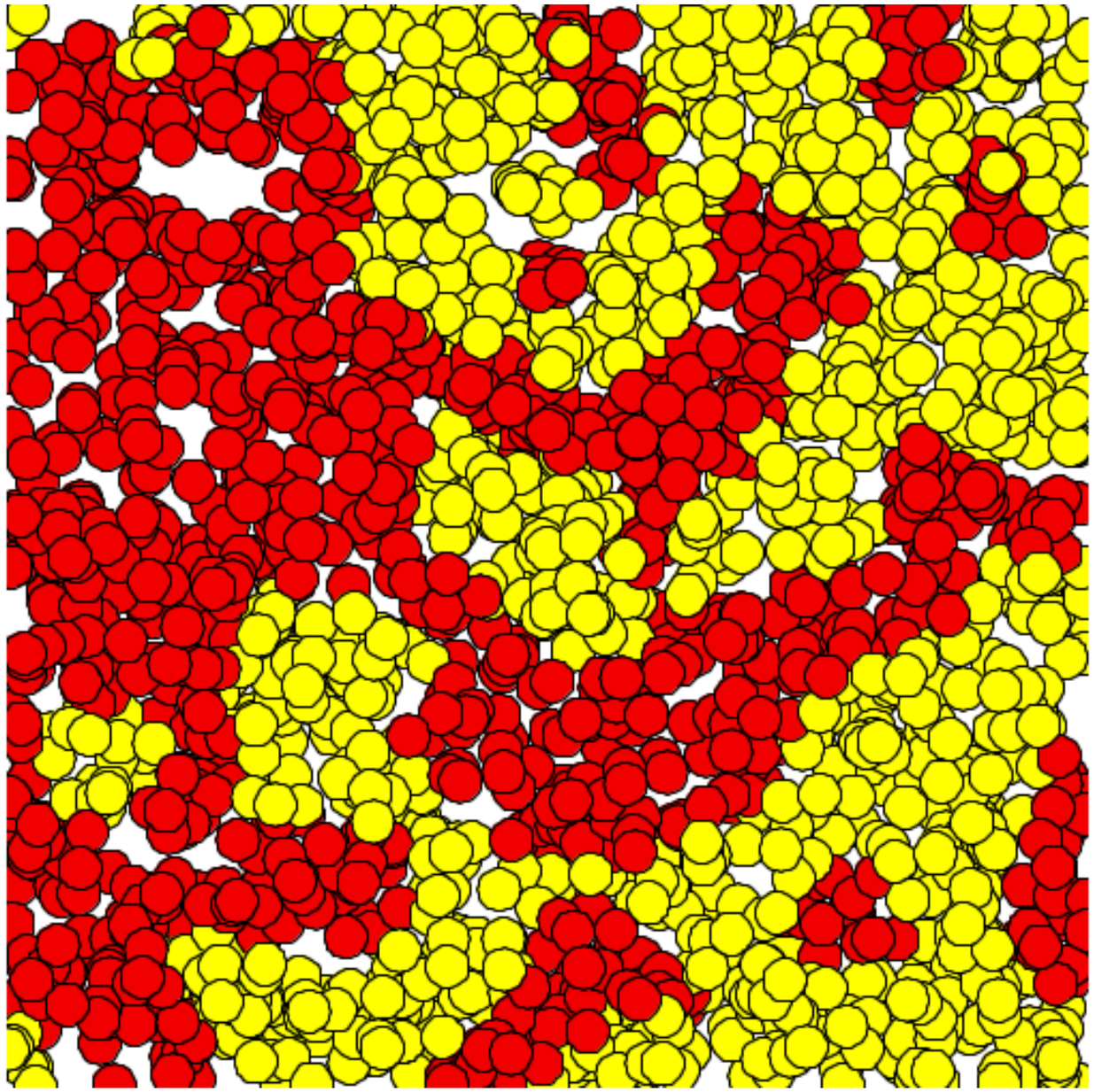
```
species people {
   //attributes
   reflex computing_similarity {
        …
   }
   reflex moving when: not is_happy {
      location <- any_location_in(world);
   }
   //aspect definition
}
```

the *any_location_in* operator returns a random location (point) inside a geometry or the geometry of an agent.

Note 1: all GAMA agents are provided with a **location** attribute that defined the coordinate of the centroid of its shape. When the location is modified, the shape of the agent is translated to the new location and when the shape is modified, the location is re-computed.

Note2: the world (global) agent can be accessed by the « **world** » keyword

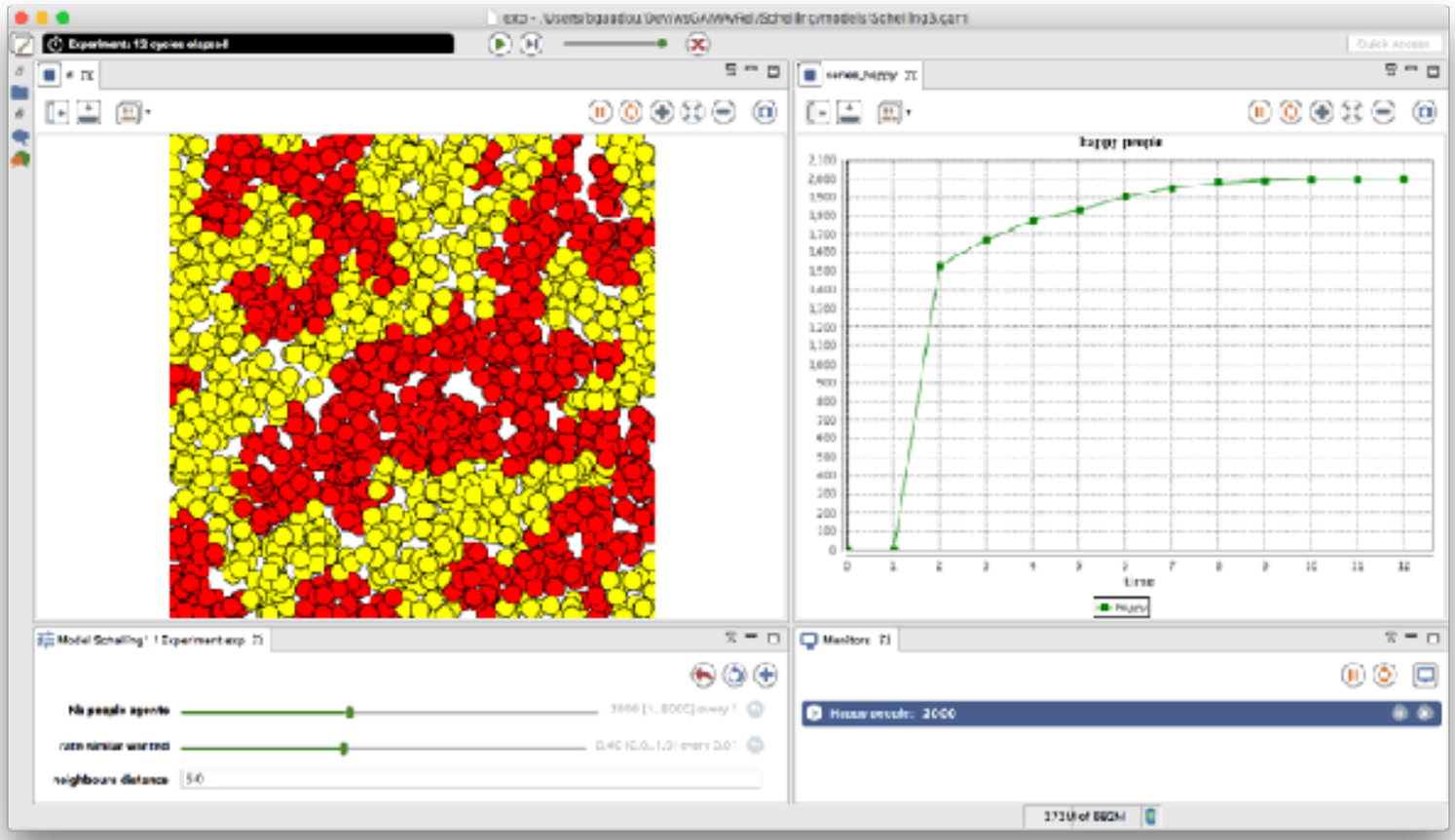
End of step 2



Now it's time to define some new parameters and new outputs for the model !

Step 3: definition of new parameters and outputs

- Objectives:
  - ➡ Definition of a new global variable to compute the number of happy people at each simulation step
  - ➡ Definition of an ending condition (when all people are happy)
  - ➡ Definition of parameters
  - ➡ Definition of a new monitor to follow the number of happy people
  - ➡ Definition of a chart to follow the evolution of the number of happy people



Segregation model 3: computation of the number of happy people

- **To do :** define a global attribute called *nb_happy_people*:
  - ➡ Type: *int*; update at each simulation step with the number of *people* agents that are happy

- **Solution :**

```
global {
    // other attributes
    int nb_happy_people <- 0 update: people count each.is_happy ;
    ...
}
```

We are now able to know at every step the number of happy people
Next step: pause the simulation when all the people agents are happy!

For that we need to use the pause action of the world agent!

Segregation model 3: Simulation Ending condition

- **To do :** define a global reflex called *end_simulation*:
  - ➡ is activated only when the number of happy people is equal to the number of people
  - ➡ call the « pause » action of the world agents that pauses the simulation

- **Solution :**

```
global {
    //attributes and init
    -
    reflex end_simulation when: nb_happy_people = nb_people {
        do pause;
    }
}
```

An action in GAML is a capability available to the agents of a species (*what they can do*)

- It is a block of **statements** that can be used and reused whenever needed. An action can accept arguments.
- An action can return a result (statement return).

```
return_type action_name (var_type arg_name,...) {
 [statements]
 [return value;]
}
```

```
action action_simple {
    write "action simple";
}
```

«write» allow to display a message in console view (best for debugging models)

Action that returns a value

- Some actions are directly available (built-in action, i.e. primitive) for all agents (e.g. die action) or to specific agents (pause action of the world agents)

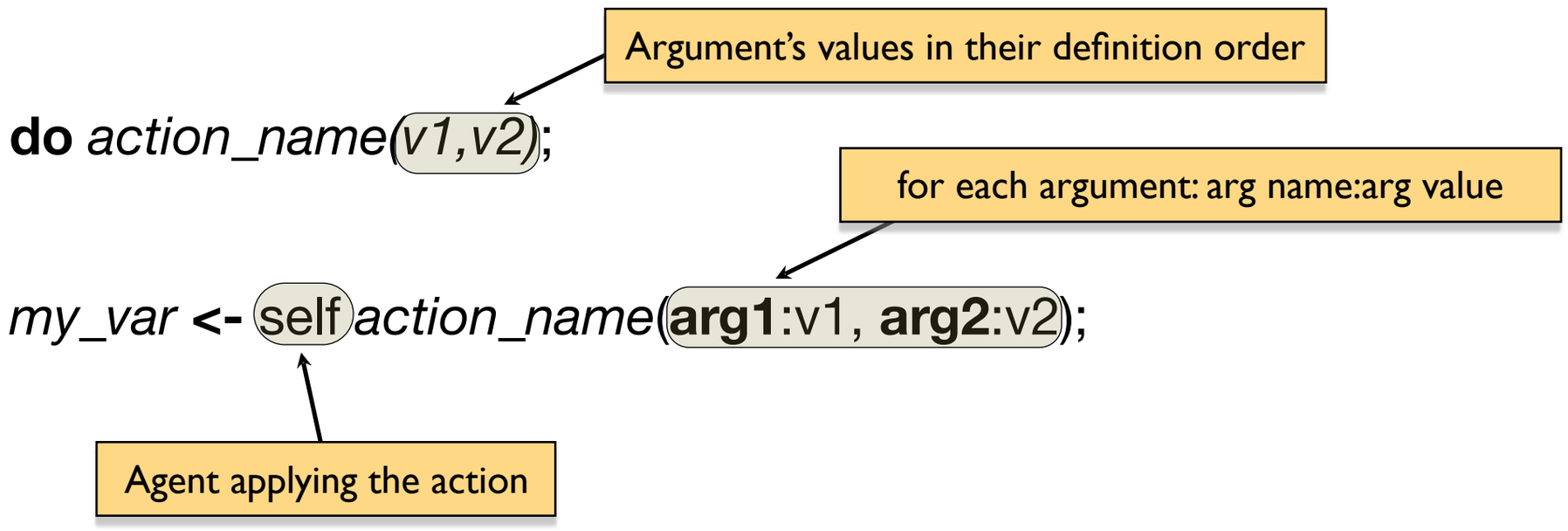
Type of the return value

Definition 2 arguments : the first one named «a», type integer and default value is «100»; the second named «b», type integer

```
int sum (int a <- 100, int b) {
    return a + b;
}
```

return a value, and finish the action

Different ways to call an action in GAML



Examples:

```
do action_simple;
int d <- self add(10,100);
int d <- self add(b:100);
```

Segregation model 3: Simulation Ending condition

- **To do :** define a global reflex called *end_simulation*:
  - ➡ is activated only when the number of happy people is equal to the number of people
  - ➡ call the « pause » action of the world agents that pauses the simulation

- **Solution :**

```
global {
    //attributes and init
    -
    reflex end_simulation when: nb_happy_people = nb_people {
        do pause;
    }
}
```

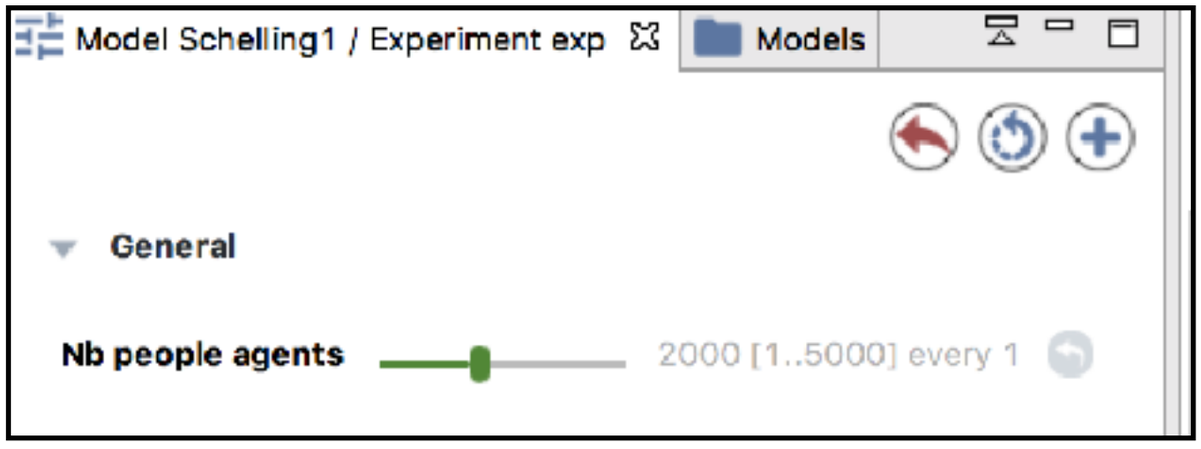
The simulation is now pause when all the agent are happy
Next step: definition of parameters!

experiment block: parameter definition

- Parameter :

  parameter *legend* **var:** *var_name* ***category:*** *my_cat;*

  - Allow to give to the user the possibility to define the value of a global attribute
  - *legend: string* to display
  - *var_name*: reference to a global attribute
  - *category*: *string* (use to better organise the parameters) - optional



Example

```
experiment main_experiment type:gui{
    parameter "Nb people agents" var: nb_people min: 1 max: 5000 step: 1;

}
```

It is possible to define here the min and max values of a parameter as well as its discretisation step

Segregation model 3: Parameter definition

- **To do :** define 3 parameters:
  - ➡ attribute: *nb_people*, legend: « nb of people »
  - ➡ attribute: *rate_similar_wanted*, legend: « rate similar wanted », min: 0.0, max: 1.0
  - ➡ attribute: *neighbours_distance*, legend: « neighbours distance », step: 1.0

- **Solution :**

```
experiment main_xp type: gui {
    parameter "nb of people" var: nb_people;
    parameter "rate similar wanted" var: rate_similar_wanted min: 0.0 max: 1.0;
    parameter "neighbours distance" var: neighbours_distance step: 1.0;

    output {
        ...
    }
}
```

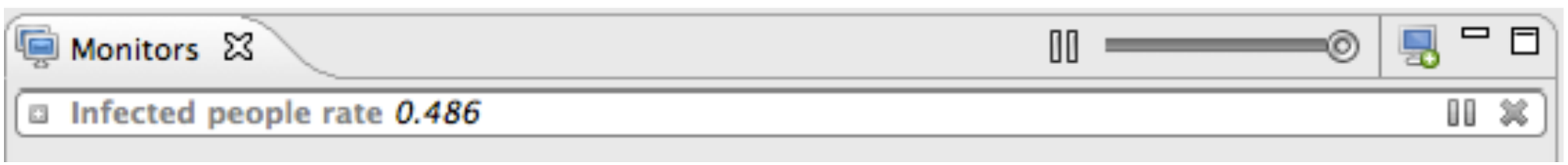
The user can now modify the value of the global attributes through parameters
Next step: definition of a monitor and a chart!

experiment block: monitor definition

- A monitor is an output allowing to display the current value of an expression
- The data to display have to be defined inside the **output** block:
  monitor *legend* **value:** *value*

Example

```
experiment main_experiment type:gui{
    //...parameters
    output {
        monitor "Infected people rate" value: infected_rate;

    //...display
    }
}
```



Segregation model 3: Monitor concerning the number of happy people

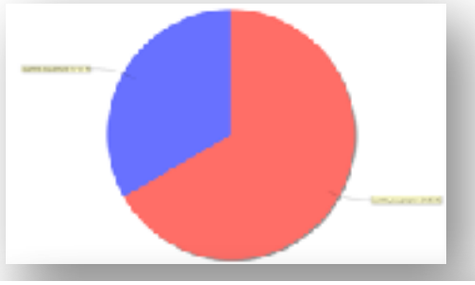
- **To do :** define a monitor to follow the evolution of the number of happy people

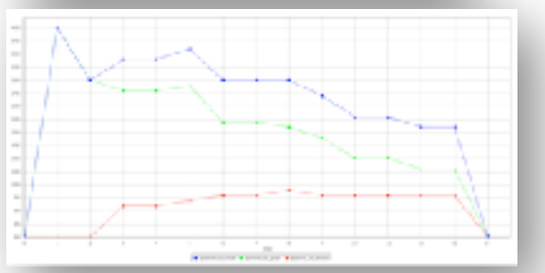
- **Solution :**

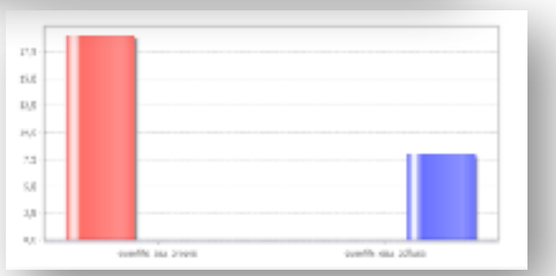
```
experiment main_xp type: gui {
    //parameter definition

    output {
        monitor "nb of happy people" value: nb_happy_people;
        //display definition
    }
}
```

experiment block: chart definition

- GAMA allows to display several type of charts :
  - Pie
  - Series
  - H*istogram*
  - *XY chart*







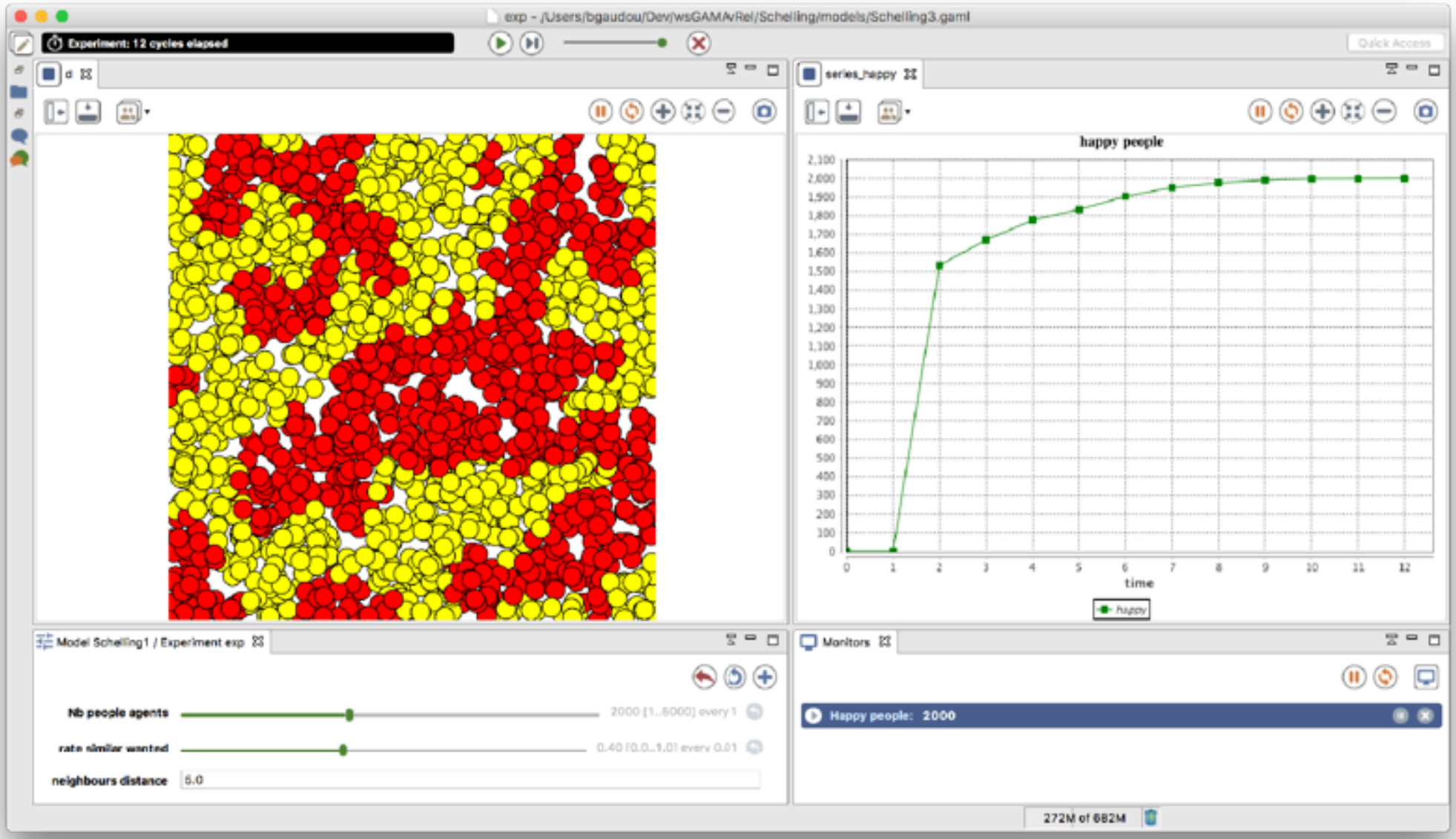
- A chart is a layer in a display: **chart** *legend* **type:** *chart_type*
- The data to display have to be defined inside the **chart** block:
  data *legend* **value:** *value* **color:** *colour*

Segregation model 3: Chart concerning the number of happy people

- **To do :** define a chart in a new display called *chart* to follow the evolution of the number of happy people.
  - ➡ chart name: « evolution of the number of happy people », type: *series*
  - ➡ data: "nb of happy people », value: nb_happy_people, color: green
- **Solution :**

```
experiment main_xp type: gui {
    // parameter definition

    output {
        // display monitor
        // map display definition
        display chart {
            chart "evolution of the number of happy people" type: series{
                data "nb of happy people" value: nb_happy_people color: #green;
            }
        }
    }
}
```

End of step 3

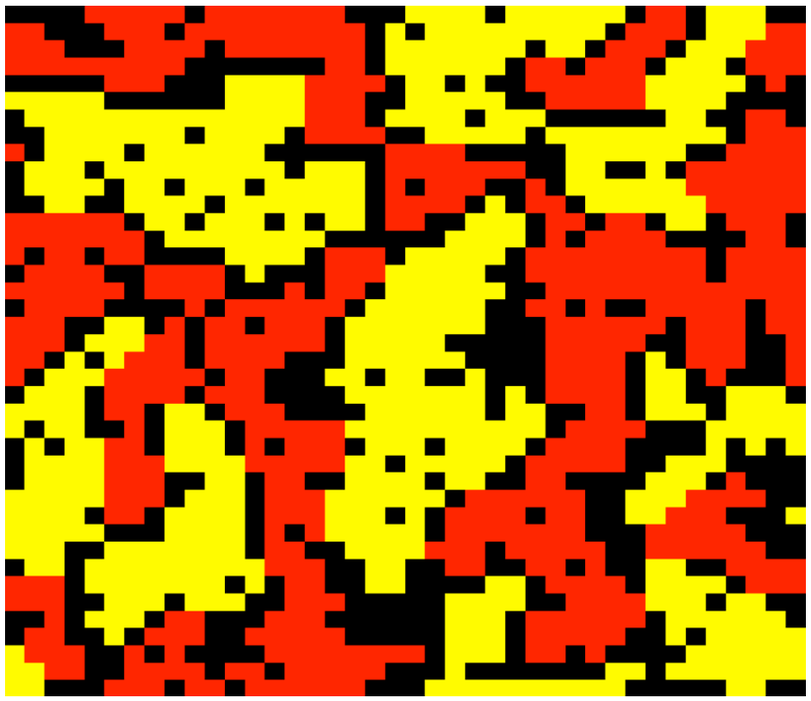


It's time to play with model !

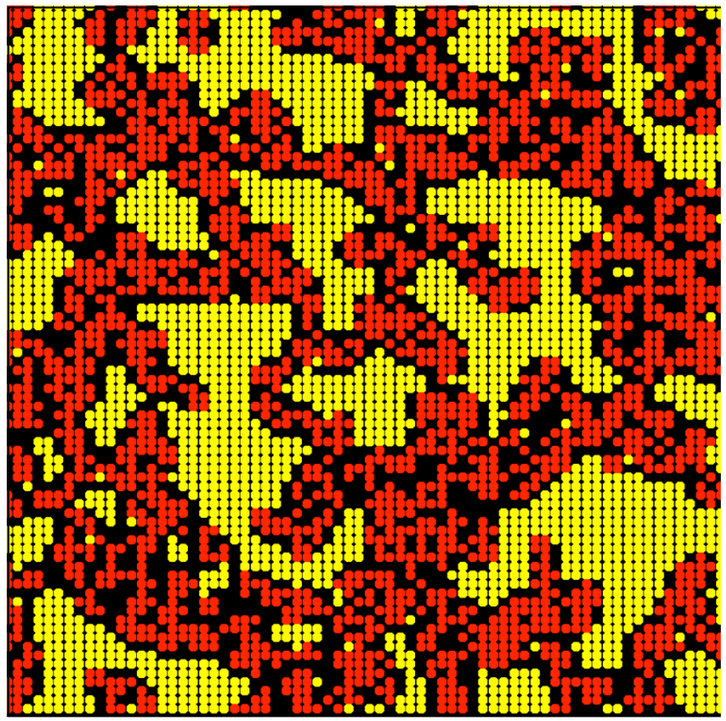
Questions: what is the impact on the number of clusters of the *rate_similarity_wanter* parameter?

Same question for the *neighbours_distance* parameter

Other implementations of the model are possible!



Cellular automata



Agents and grid

Agents and GIS