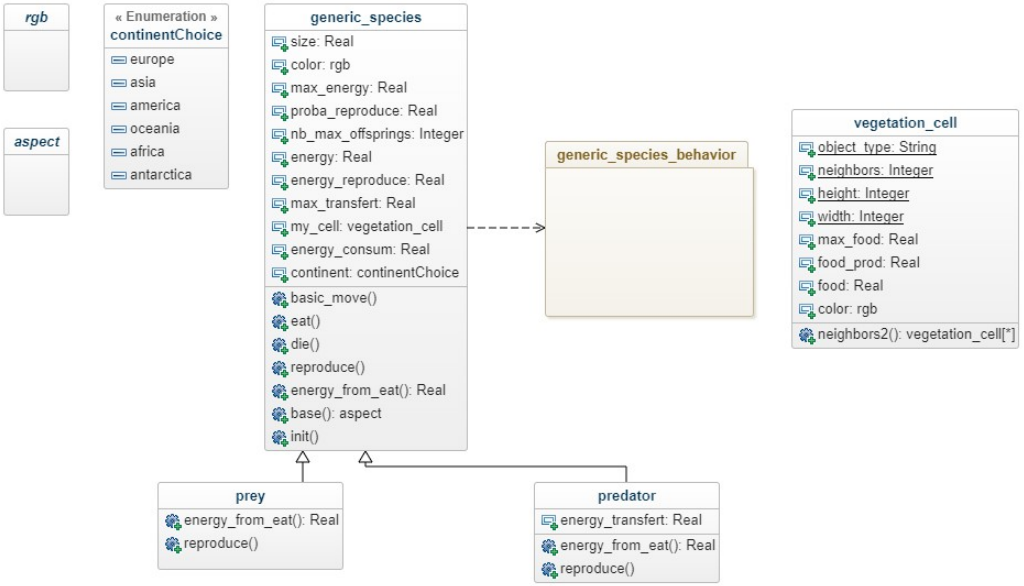
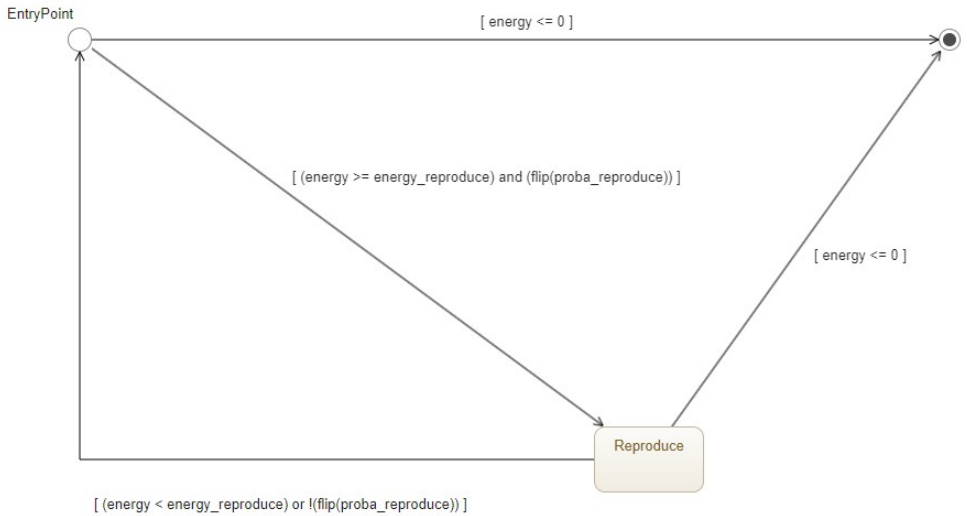
Un fichier gaml comprend 3 principales parties : « global », « experiment » et la partie décrivant les différentes classes (*species*, *grid*, etc.). Ces différentes parties sont représentées par les différents package respectivement : « global », « experiment » et « meta_model ». Le dernier package comporte le diagramme objet utilisé pour initialiser les agents de la scène.

Le package meta_model contient un diagramme de classes décrivant nos classes et un autre package décrivant le comportement d'une de nos classes. Lorsqu'une classe est abstraite comme les classes « rgb » et « aspect » elles ne sont pas interprétées par le transformateur. Elles servent uniquement à indiquer un type de donnée inhérent à gaml. La classe « vegetation_cell » présente 4 attributs *static*. Le transformateur détecte ces attributs pour configurer correctement la classe sous gaml :

```
grid vegetation_cell width: 50 height: 50 neighbors: 4 {}
```

Les valeurs de ces attributs statiques sont indiquées en *defaultValue*.
Le type Enumeration n'existe pas sous gaml, actuellement les énumérations sont traduites par des tableaux au sein de la partie *global*.
Le package « generic_species_behavior » est détecté comme un comportement par le modèle grâce à l'ajout d'une propriété « behavior » au package. Ensuite, une simple relation de dépendance permet de réaliser le lien entre une classe et un comportement.





Le package « generic_species_behavior » représente le comportement de la classe « generic_species ». C'est un diagramme d'états dans lequel nous pouvons définir la condition de changement d'état (attribut *guard*) et les différentes fonctions à exécuter. Ces fonctions doivent être indiquées sous la forme de propriétés attachées aux états et aux transitions. Le diagramme doit contenir un « EntryPoint » et un « FinalPoint » afin d'indiquer aux transformateur l'état initial et final du FSM. Voici la traduction de ce diagramme :

```
    state EntryPoint  initial: true  {
        do basic_move();
        do eat();
        transition to: Reproduce when: (energy >= energy_reproduce)
and (flip(proba_reproduce));
        transition to: FinalPoint when: energy <= 0;
    }


    state Reproduce  {
        do reproduce();
        transition to: EntryPoint when: (energy < energy_reproduce)
or !(flip(proba_reproduce));
        transition to: FinalPoint when: energy <= 0;
    }


    state FinalPoint  final: true  {
        do die();
    }
```

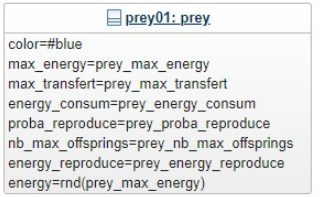
Les fonctions « basic_move », « eat », etc. doivent être présentes sous la forme d'**opération** au sein de la classe liée au comportement.

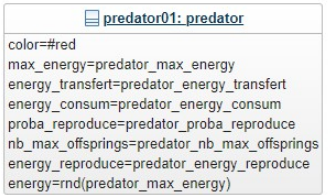
La package global contient un diagramme de classe avec la classe global indiquant l'ensemble des variables globales au script. Les valeurs de ces variables sont indiquées en *defaultValue*.

```
global {
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.4;
    int prey_nb_max_offsprings <- 5;
    ...
```

Enfin le package « instanciation » contient un diagramme d'objets dans lequel nous devons instancier nos objets. Ces instances sont transformées au sein du bloc *init* contenu dans *global*.

```
init {
        create prey {
            color <- #blue;
            max_energy <- prey_max_energy;
            max_transfert <- prey_max_transfert;
            energy_consum <- prey_energy_consum;
            proba_reproduce <- prey_proba_reproduce;
            nb_max_offsprings <- prey_nb_max_offsprings;
            energy_reproduce <- prey_energy_reproduce;
            energy <- rnd(prey_max_energy);
    }
        create predator {
            color <- #red;
            max_energy <- predator_max_energy;
            energy_transfert <- predator_energy_transfert;
            energy_consum <- predator_energy_consum;
            proba_reproduce <- predator_proba_reproduce;
            nb_max_offsprings <- predator_nb_max_offsprings;
            energy_reproduce <- predator_energy_reproduce;
            energy <- rnd(predator_max_energy);
    }
  }
```





```
    "generic_species" : {
        "basic_move": "my_cell <- one_of(my_cell.neighbors2()); location <- my_cell.location;",
        "eat": "energy <- energy + energy_from_eat();",
        "die": "do die;",
        "reproduce": "return;",
        "energy_from_eat": "return 0.0;",
        "base": "draw circle(size) color: color;",
        "init": "location <- my_cell.location;"
    },
    "prey": {
        "reproduce": "int nb_offsprings <- rnd(1, nb_max_offsprings); create species(self) number: nb_offsprings { color <- #blue; max_energy <- prey_max_energy; max_transfert <-
        prey_max_transfert; energy_consum <- prey_energy_consum; proba_reproduce <- prey_proba_reproduce; nb_max_offsprings <- prey_nb_max_offsprings; energy_reproduce <-
        prey_energy_reproduce;  energy <- myself.energy / nb_offsprings; my_cell <- myself.my_cell; location <- my_cell.location; } energy <- energy / nb_offsprings;",
        "energy_from_eat": "float energy_transfert <- 0.0; if(my_cell.food > 0) { energy_transfert <- min([max_transfert, my_cell.food]); my_cell.food <- my_cell.food - energy_transfert; }
        return energy_transfert;",
        "choose_cell": "return (my_cell.neighbors2()) with_max_of (each.food);"
    },
    "predator": {
        "reproduce": "int nb_offsprings <- rnd(1, nb_max_offsprings); create species(self) number: nb_offsprings { color <- #red; max_energy <- predator_max_energy; energy_transfert <-
        predator_energy_transfert; energy_consum <- predator_energy_consum; proba_reproduce <- predator_proba_reproduce; nb_max_offsprings <- predator_nb_max_offsprings; energy_reproduce
        <- predator_energy_reproduce; energy <- myself.energy / nb_offsprings; my_cell <- myself.my_cell; location <- my_cell.location; } energy <- energy / nb_offsprings;",
        "energy_from_eat": "list<prey> reachable_preys <- prey inside (my_cell); if(! empty(reachable_preys)) { ask one_of (reachable_preys) { do die; } return energy_transfert; } return 0.
        0;",
        "choose_cell": "vegetation_cell my_cell_tmp <- shuffle(my_cell.neighbors2()) first_with (!(empty(prey inside (each)))); if my_cell_tmp != nil { return my_cell_tmp; } else { return
        one_of(my_cell.neighbors2()); }"
    },
    "vegetation_cell": {
        "neighbors2": "return (self neighbors_at 2);"
    },
    "prey_predator": {
        "main_display": "grid vegetation_cell lines: #black; species prey aspect: base; species predator aspect: base;"
    }
}
```

Enfin, il faut un fichier .json permettant de stocker les fonctions de comportement sous gama. Pour utiliser ces fonctions, il faut indiquer le nom des **opérations** au sein des classes (représentées ici sous la forme de clés de notre .json).

Les diapositives suivantes présentent le code généré à partir de ces éléments.

```
global {
    float prey_max_energy <- 1.0;
    float prey_max_transfert <- 0.1;
    float prey_energy_consum <- 0.05;
    float predator_max_energy <- 1.0;
    float predator_energy_transfert <- 0.5;
    float predator_energy_consum <- 0.02;
    float prey_proba_reproduce <- 0.4;
    int prey_nb_max_offsprings <- 5;
    float prey_energy_reproduce <- 0.5;
    float predator_proba_reproduce <- 0.6;
    int predator_nb_max_offsprings <- 3;
    float predator_energy_reproduce <- 0.5;

    init {

        create prey {
            color <- #blue;
            max_energy <- prey_max_energy;
            max_transfert <- prey_max_transfert;
            energy_consum <- prey_energy_consum;
            proba_reproduce <- prey_proba_reproduce;
            nb_max_offsprings <- prey_nb_max_offsprings;
            energy_reproduce <- prey_energy_reproduce;
            energy <- rnd(prey_max_energy);
        }

        create predator {
            color <- #red;
            max_energy <- predator_max_energy;
            energy_transfert <- predator_energy_transfert;
            energy_consum <- predator_energy_consum;
            proba_reproduce <- predator_proba_reproduce;
            nb_max_offsprings <- predator_nb_max_offsprings;
            energy_reproduce <- predator_energy_reproduce;
            energy <- rnd(predator_max_energy);
        }
    }
}

experiment prey_predator type: gui  {
    output {
        display main_display {
            grid vegetation_cell lines: #black; species prey aspect: base; species predator aspect: base;
        }
    }
}
```

```
species generic_species control: fsm {
    float size <- 1.0 ;
    rgb color ;
    float max_energy ;
    float proba_reproduce ;
    int nb_max_offsprings ;
    float energy update: energy - energy_consum max: max_energy;
    float energy_reproduce ;
    float max_transfert ;
    vegetation_cell my_cell <- one_of(vegetation_cell) ;
    float energy_consum ;
    int continent <- 0 ;

    action basic_move {
        my_cell <- one_of(my_cell.neighbors2()); location <- my_cell.location;
    }
    action eat {
        energy <- energy + energy_from_eat();
    }
    action die {
        do die;
    }
    action reproduce {
        return;
    }
    float energy_from_eat {
        return 0.0;
    }
    aspect base {
        draw circle(size) color: color;
    }
    init {
        location <- my_cell.location;
    }

    state EntryPoint  initial: true  {
        do basic_move();
        do eat();
        transition to: Reproduce when: (energy >= energy_reproduce) and (flip(proba_reproduce));
        transition to: FinalPoint when: energy <= 0;
    }
    state Reproduce  {
        do reproduce();
        transition to: EntryPoint when: (energy < energy_reproduce) or !(flip(proba_reproduce));
        transition to: FinalPoint when: energy <= 0;
    }
    state FinalPoint  final: true  {
        do die();
    }
}
```

```
species prey parent: generic_species {

    float energy_from_eat {
        float energy_transfert <- 0.0; if(my_cell.food > 0) { energy_transfert <- min([max_transfert, my_cell.food]); my_cell.food <-
my_cell.food - energy_transfert; } return energy_transfert;
    }
    action reproduce {
        int nb_offsprings <- rnd(1, nb_max_offsprings); create species(self) number: nb_offsprings { color <- #blue; max_energy <-
prey_max_energy; max_transfert <- prey_max_transfert; energy_consum <- prey_energy_consum; proba_reproduce <- prey_proba_reproduce;
nb_max_offsprings <- prey_nb_max_offsprings; energy_reproduce <-  prey_energy_reproduce;  energy <- myself.energy / nb_offsprings; my_cell <-
myself.my_cell; location <- my_cell.location; } energy <- energy / nb_offsprings;
    }
}


species predator parent: generic_species {
    float energy_transfert ;

    float energy_from_eat {
        list<prey> reachable_preys <- prey inside (my_cell); if(! empty(reachable_preys)) { ask one_of (reachable_preys) { do die; } return
energy_transfert; } return 0.0;
    }
    action reproduce {
        int nb_offsprings <- rnd(1, nb_max_offsprings); create species(self) number: nb_offsprings { color <- #red; max_energy <-
predator_max_energy; energy_transfert <- predator_energy_transfert; energy_consum <- predator_energy_consum; proba_reproduce <-
predator_proba_reproduce; nb_max_offsprings <- predator_nb_max_offsprings; energy_reproduce <- predator_energy_reproduce; energy <-
myself.energy / nb_offsprings; my_cell <- myself.my_cell; location <- my_cell.location; } energy <- energy / nb_offsprings;
    }
}


grid vegetation_cell width: 50 height: 50 neighbors: 4 {
    float max_food <- 1.0 ;
    float food_prod <- rnd(0.01) ;
    float food <- rnd(1.0) max: max_food update: food + food_prod ;
    rgb color <- rgb(int(255 * (1 - food)), 255, int(255 * (1 - food))) update: rgb(int(255 * (1 - food)), 255, int(255 * (1 - food))) ;

    list<vegetation_cell> neighbors2 {
        return (self neighbors_at 2);
    }
}
```