Tutorial

Bugs growing in grid space*

implemented in FABLES

Benny Höckner

August 17, 2011

*Idea based on [RaLyJa]

Abstract

This tutorial should give an introduction to the programming language FABLES. This abriviation stands for "Functional Agent-Based language for Simulations" and was developed by AITIA, Inc. and is still under development. It is also a component in the MASS package.
For this paper I suppose that the reader already has successfully installed MASS / FABLES and also has created first examples. A detailed step by step instruction can be found in chapter 4 of the "Game of Life" tutorial, which is available at the AITIA homepage[1].
The reader who is still familiar with the main concepts of Fables, could skip the first chapter and jump directly to the example and its implementation.
As example we will implement bugs which could move freely over an grid space (torodial) and are able to eat food, to eat each other and to reproduce.

Contents

1. Fables **1**
- 1.1. Some facts about Fables 1
- 1.2. Hello world 1
- 1.3. Variables and global parameters 2
- 1.4. Agents 4
- 1.5. Scheduling 6
- 1.6. Functions 8
- 1.7. Conditional expressions 9

2. Bugs growing in grid space **10**
- 2.1. Natural language description 10
- 2.2. Implementation in Fables 10

3. Visualisation **18**
- 3.1. How can I visualize something in Fables? 18
- 3.2. Visualization of the bugs 20

4. Conclusion **25**

A. Complete source code of scenario: bugs growing in grid space **26**

[1] http://mass.aitia.ai/

1. Fables

In this chapter I will give an fast introduction to the programming language Fables. I will only focus on these components, which I need later to implement the bugs example. So if you want to know more about some points, then feel free to have a look at the Fables manual, which is also available on the homepage.

1.1. Some facts about Fables

The Functional Agent-Based Language for Simulations (Fables) is a programming language and its integrated environment developed by AITIA, Inc. specially designed for creating agent-based simulations. It requires minimal programming skills, as its formalism is similar to the mathematical formalism used in publications in the subject. Fables is a dynamically typed, lazy, non-pure functional language. It contains higher level functions, set and sequence expressions, and special language constructs for scheduling events. If you are familiar with programming you might note that Fables is multi-paradigm language. It merges the aspects of object-oriented, functional and procedural languages. [FabMan08]

1.2. Hello world

Now we want to write our basic program[2] in Fables. I assume that the reader knows how to install and start Fables and how to create a new project. If you have created a new Fables project successfully, then you should see a **.fab* file with the name of your project. In our case *example.fab*. This is the place where your code goes into! The file should contain the following lines of code:

```
model Example {
  startUp {
    seed(1);
  }
};
```

The *startUp* section will be executed at the beginning of the simulation and therefore its mandatory. At the beginning of the simulation you should initialize the *pseudo-random generator*. This is something like a function which should produce random values, but it does not. That means, that it could only produce the same values for the same input. In other words, if you do not change the seed value in every simulation, then you will always rerun the same simulation. So we have a deterministic behaviour. This is typical for functional programming language. In our case it is not mandatory to change the seed value, so lets go further.
Now we want to add the part of code that produces our wanted output. The only thing we have to do therefore is to add "printLn("Hello world");" into the startUp function:

[2] Lets call it: *Example*

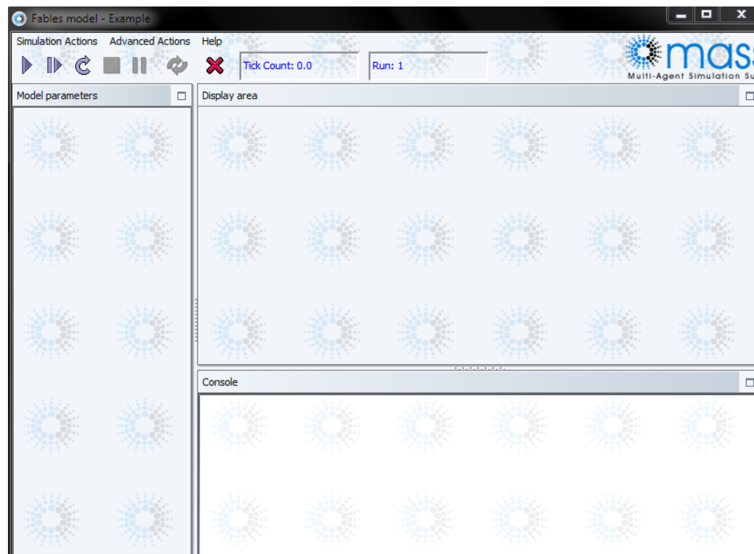

Figure 1.1: Screenshot of the simulation window

```
model Example {
  startUp {
    seed(1);
    printLn("Hello world");
  }
};
```

The next step is to compile our little program. This is a very easy task. The only thing we have to do, is to click on the "Run fables code without GUI"-Button. Alternatively you could go into the menu line → "Fables" → "Run fables code without GUI". The compiling should be successful. After a few seconds (depending on the speed of your computer) a window should appear which looks like the one in figure 1.1.

Now we can run ("play" symbol) our simulation and the output "Hello World" should appear in the *Console* section.

1.3. Variables and global parameters

I expect that everyone knows what are variables, so I only want to have a look to the syntax. In Fables variables are declared by the keyword *var*. There are no keywords like *string*, *boolean*, *char*, *int*, *byte*, *double* and so on available. But in fact Fables supports type checking and the types are not dynamically detected like in Java Script! It seems that the compiler trys to detect the best datatype by himself. So it is no problem to mix *integer* and *double* values in one variable[3], but mixing e.g. *integer* and *string* will not

[3]Please avoid these kind of programming

work. Now lets have a look at an example of how to create a variable:

```
model Example {
  var variable;
  startUp {
    seed(1);
    variable := 1;
    printLn(variable);
  }
};
```

We see that we have to use the operator := for value assignment. And there is another special thing we have to know. In Fables variables are global variables. That means that you have to declare them outside of the startUp function. The example will not work if you put the line "var variable" inside. We will discuss in section 1.6 how we could declare local variables.

Now we learn something about global parameters and why they could be useful. These parameters behave like global constants known from other programming languages like C or Java. So they are *read-only*. That means, that it is not possible to change their values during the runtime. Please note the fact, that for these parameters we could *not* use the var keyword and the symbol for the value assignment is only the =.
Example:

```
model Example {
  global = 1;
  startUp {
    seed(1);
    printLn(global);
  }
};
```

At the moment we do not really have a profit from global constants, but with a little more code we will have.
If we add the constant as a parameter to the startUp function, then we are able to change the value of the variable in the simulation window (figure 1.2). That means that we could change the value of the variable and rerun the simulation without changing and recompiling the code. This is much more comfortable. So lets have a look to the code:

```
model Example {
  global = 1;
  startUp(global) {
    seed(1);
    printLn(global);
  }
};
```

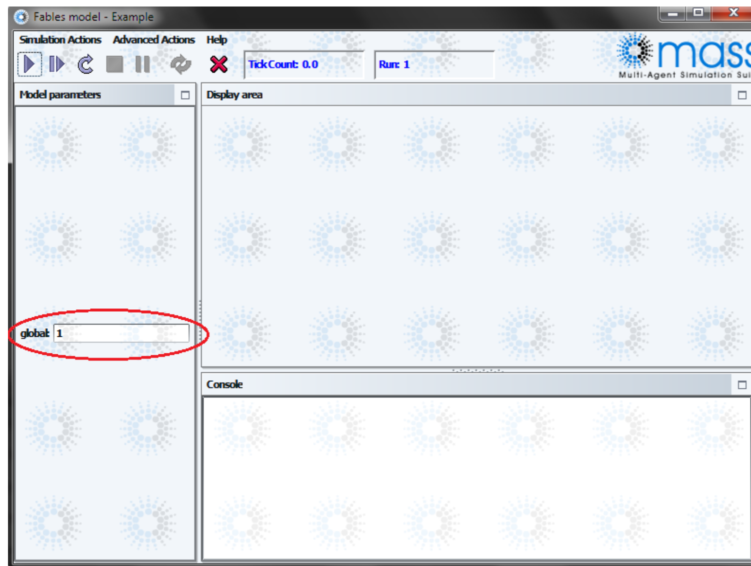

Figure 1.2: Simulation window with a global variable

Alternativly Fables provides a keyword named param. If you write it before a global variable, then you do not have to put this variable as a parameter of the startUp function. It will automatically added for you. So the next listing is equivalent to the last one:

```
model Example {
  param global = 1;
  startUp {
    seed(1);
    printLn(global);
  }
};
```

You can decide which notation you prefer. I will use the first one.

1.4. Agents

In Fables there is not really a distinction between agents and objects. So the declaration and the use are very similar to other object oriented languages. Agents could have attributes and methods. Furthermore they could contain schedule functions. These we will discuss later.
Agents *could* also have a startUp function. These is called then the agent is created. So it is comparable to a constructor function.
Fables also offers the possibility to reference the actual instance of an agent by the keyword self. It is similar to the keyword *this* known from object oriented-languages like Java or C.

The creation and deletion of a agents works also like in object-oriented languages. In Fables therefore are the keywords create[4] and delete with the following syntax:

```
create <ClassName> [ <initial-assignments> ]
delete(<agent>)
```

Now lets have a short look at *sets* and *sequences*. Both data structures could be used to hold many agents (and of course simple variables) of the same type. For the reader which do not know what a sequence is, it could be compared with a single linked list. But the programmer do not have to worry about things like pointers or references. The declaration of sets and sequences remembers on mathematical formalism for creating a set. Lets have a look at the syntax:

```
// set
{ <expression> : <var> is <set | sequence> (when <conditions>) }

// sequence
[ <expression> : <var> is <set | sequence> (when <conditions>) ]
```

Between both structure types are two big differences. Firstly sets could only contain any element once and secondly they are unordered. For our later examples we will use sequences.
At next we need also a possibility to iterate through all of our agents. Therefore a construct named for each exists with the syntax:

```
for each <var> in <Collection> do
  <statement>;
```

Now lets take a look at the next example. There we want to create a fixed number of agents of type *OurAgent* and print all of there attributes.

```
model Example {
  global = 5;
  startUp(global) {
    seed(1);
    [create OurAgent[id := i] : i is [1..global]];

    for each a in OurAgent do {
      printLn(a.id ++ "␣" ++ a.anotherAttribute)
    };
  }
  class OurAgent {
    var id;
    var anotherAttribute;
    startUp() {
```

[4] There is also a keyword new with the same meaning, but the compiler complains often about it. So don't use it.

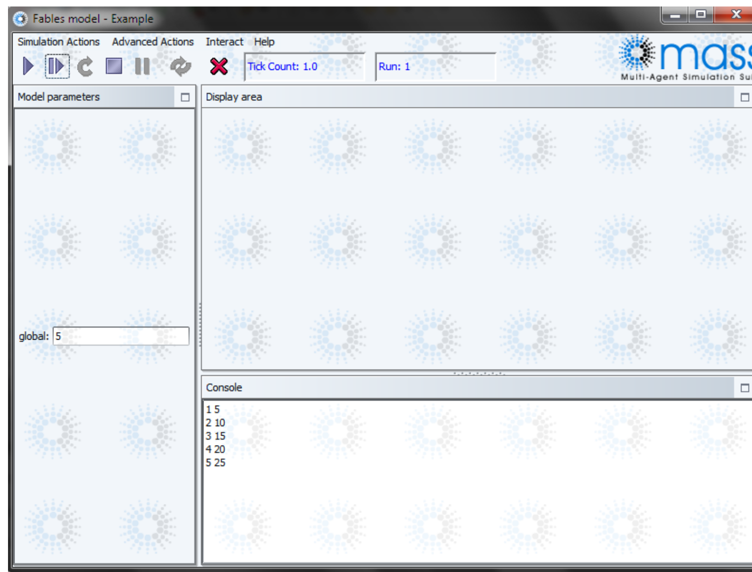

Figure 1.3: Output of simulation with agents

```
      anotherAttribute := id * 5;
    }
  }
};
```

Figure 1.3 shows that our example works as expected. Maybe confusing for the reader is the fact, that our sequence is not assigned to a variable, but this is not necessary. The for each statement allows also to use agent classes as collections. The environment automatically collects all of our existing instances of the class.

1.5. Scheduling

The execution of a model involves scheduling and executing events. Events are contained in schedules and assigned to specified times. Schedules can belong to the model itself or to individual agents (objects). Each event has a positive time value. The order of event activations is determined by their time variables. Events with lower time values are activated first. When two or more events should be executed at the same time, the order is chosen randomly, generated by the standard (deterministic) pseudo-random generator. [FabMan08]
Schedules are defined with the following syntax:

```
schedule <ScheduleName> {
  <events>
}
```

The name of the schedule is optional. It is only necessary if you need it e.g. for event control. But we do not need it (see in [FabMan08] for more details). The mentioned events (in the last listing) have the following syntax:

```
<time> : <expression1>; <expression2>;...; <expressionN>;
```

Furthermore schedules can be cyclic. This is denoted by the cyclic keyword and the period, a positive integer. The syntax is defined as follows:

```
schedule <ScheduleName> cyclic <period> { ... }
```

If a schedule is cyclic, then all of its events occur periodically, according to the specified period. If the schedule's period is P, and an event is defined to be executed at the Nth simulation turn, then it is executed in turns N, $N+P$, $N+2P$, $N+3P$, ..., $N+k*P$, ... [FabMan08]

Now we want to look at a little example. Therefore we extend the previous example with two cyclic schedules one for every agent and one for the simulation. Furthermore we add a schedule to the agents, that will executed after the initialization. In this example it has the same effect if we would write the event into the startUp function. But for demonstration purposes we do not:

```
model Example {
  global = 5;
  startUp(global) {
    seed(1);
    [create OurAgent[id := i] : i is [1..global]];

    for each a in OurAgent do
    {
      printLn(a.id ++ "␣" ++ a.anotherAttribute)
    };
  }
  class OurAgent {
    var id;
    var anotherAttribute;
    startUp() {
      anotherAttribute := id * 5;
    }
    schedule {
      0.1 : {
        printLn("I␣am␣alive!␣My␣id␣is:␣" ++ id)
      };
    }
    schedule nonsenseName cyclic 1 {
      1.1 : {
        printLn("Agent␣" ++ id ++
          ":␣whooohoooo,␣I␣survived␣another␣simulation␣step!")
```

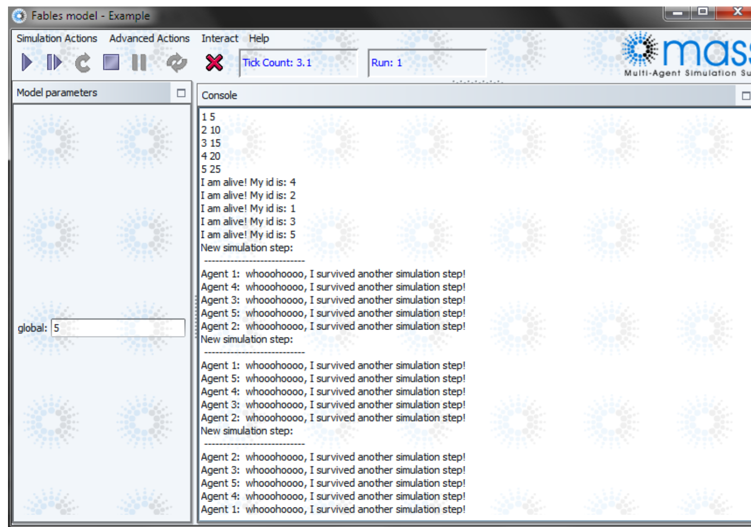

Figure 1.4: Output if simulation with schedules and events

```
      };
    }
  }
  schedule cyclic 1 {
    1 : {
      printLn("New␣simulation␣step:␣\n" ++
          "---------------------------")
    };
  }
};
```

In figure 1.4 we could see the output for 3 simulation steps. The screenshot proves that the event with the time *0.1* is executed directly after the startUp function like we expected. Also we could recognize that this event is never executed again. Then we see that the *New simulation step ...* - output always appear before the outputs of the agents. This is because the time of this event is smaller then the time of the agents. If we had written a *1* instead of a *1.1* then the same would happen, which happens to the agents. They are executed randomly. If you take a closer look at the id's, you will see that their order is different in every step. This is because the event times are equal.

1.6. Functions

Of course Fables provides the possibility to implement own functions. The structure and behave is similar to functions in functional programming languages. It is very important to know that it is not possible to create new variables inside the function body. Therefore an optional where part could be implemented to define local variables.

The syntax of functions is defined as follows:

```
FunctionName ( <parameter list> ) = <expression>
            where ( <local-definition>, <local-definition>,...);
```

Also important to mention is that variables defined in the where part could *not* changed inside the function body. This behaviour is typical for functional languages. An example is shown in section 1.7

1.7. Conditional expressions

Fables is capable of defining partial functions. You can define conditions for each expression and the definition will have the given expression's value if the declared logical condition is *true*. The conditional expressions in Fables have the following form:

```
<condition> => <expression>
| <condition> => <expression>
...
otherwise <expression>
```

The value of the expression is the value of the expression after the first true condition, or the value of the expression after the otherwise keyword, if none of the conditions are *true*. The otherwise branch can be omitted. If there is no branch, and none of the conditions are *true*, the conditional expression results in an exception. For conditionals containing statements only, the otherwise branch can be omitted. There is also an easy way to tell the compiler "do nothing" on a branch by using the empty statement: void. [FabMan08]
Lets let look at a little example which implements a kind of silly *logical and*.

```
model Example {
  var x := 1;
  var y := 0;
  startUp {
    myAnd(x,y);
  }
  myAnd(x,y) =  z == 1  => printLn("Both␣are␣TRUE!")
              | x == 1  => printLn("Only␣x␣is␣TRUE!")
              | y == 1  => printLn("Only␣y␣is␣TRUE!")
              otherwise    printLn("Both␣are␣FALSE!")
          where (z = x * y);
};
```

Again, this example is only for demonstration purposes. Of course Fables supports the three standard logical operators and, or and not.

2. Bugs growing in grid space

In this chapter we will use our knowledge earned in chapter 1 to implement a little and easy to understand simulation example. At first the example will be introduced and then we will implement it step by step.

2.1. Natural language description

Bugs move around randomly in a grid space[5]. They feed and grow. Food is represented as a grid cell property. When a bug arrives on a new cell, it eats the available food. Bugs also reproduce and cannibalize each other, implemented in the following way:

- When two bugs of the same gender meet in one cell the older bug eats the younger one and increases his weight about the weight value of the younger bug.
- When a bug male and female meet each other in a cell they reproduce if both of them are older than the maturity level (= 20 Days), otherwise the older bug eats the younger one.
- The pregnant female bug will give birth to a new bug after some time interval: the pregnancy interval (= 3 Days).
- If a bug found a cell with food he eats 20% of the avaiable food and increases his weight about the eaten foodvalue.
- A bug looses one weight point per simulation step and if his weight reached zero then he will die because of starvation.

2.2. Implementation in Fables

Now we want to implement our simulation. Therefore we need two different types of agents. One for the food and another for the bugs. The food is very easy, so we will implement it first. The *food*-agent only needs two things to know about himself:

- position, for saving the location of the food
- amount of available food

The position is encoded as a sequence of length two. For the amount we define for simplification that it is always a natural number. The code could look like this:

```
class Food
{
  // position of the food
  var pos;
```

[5] 2 dimensional and toroidal

```
  // amount of available food
  var amount;
}
```

Now we need to implement our agent for the bugs. This one is a little more complex but also not very complicated. We need the following attributes:

- position,
- gender,
- id,
- age,
- weight,
- pregnancy.

The position is encoded like the position of the food. Something unusual for the gender is that we distinguish between: child, male and female. This is maybe not understandable at this time, but simply do not think about. The reason is because of visualization purposes. The id attribute is necessary, because we later have to distinguish our bugs. The age and weight attributes are self explaining and for simplification we assume that the values are integer numbers. The pregnant attribute is not a boolean flag. It is an integer to save the time in *how much* simulation steps a child will be born. -1 encode the state that a bug is not pregnant.
Furthermore we need a schedule with the following events, which are all executed in every simulation step:

- movement
- aging
- weight losing and starvation checking
- maturity level checking
- pregnancy checking

The movement is simulated by increasing, decreasing or not changing the x and y value of the position. So it is also possible that the bug stays on his actual position for ever, but this is of course not very probable[6]. To implement the toroidal plane, we use a little function named *norm* which returns the *modulo* value of the given value and the *worldsize*. The code is very short:

[6] 8 neighbour cells + actual cell => 9 possibilities; probability to stay on same position for n iteration is $(\frac{1}{9})^n$

```
// function to create a doughnut world
norm (x) = x mod worldSize;
```

The aging and weight losing are simply implemented by in-and decreasing the corresponding old value. If the weight is lower or equal than zero, the bug must die.
For the maturity level checking we need a condition which checks if the bug is old enough and if this is true then his gender will be chosen randomly. So we automatically know, if a bug is a male or female bug, that he is able to reproduce.
Also the pregnancy value is checked in a condition. If the bug is pregnant then the pregnancy value will be decreased by 1 in every step. If it reached zero then a new bug will be born.
The bugs could be implemented in the following way:

```
class Bug
{
  // attributes
    // position of the bug
    var pos;

    // 2 .. female
    // 1 .. male
    // 0 .. child => not able to reproduce
    var gender;

    // id of the bug
    var id;

    // age of the bug
    var age;

    // weight of the bug
    var weight;

    // pregnant time
    // if 0 then child must be born
    var pregnant;

  // events
    schedule cyclic 1
    {
      // When should the event be executed
      // => 1.1 iterations steps after creating the bug
      1.1 :
      {
        // Change position
        pos := [norm(pos(0) + discreteUniform(-1, 0, 1)),
                norm(pos(1) + discreteUniform(-1, 0, 1))],
```

```
        // Increase age by 1
        age := age + 1,

        // Check if bug must die because of starvation
        // if not decrease weight by 1
        weight <= 0 => delete(self)
        otherwise weight := weight - 1,

        // if bug is old enough, then give him a gender
        age == maturity => gender := discreteUniform(MALE,FEMALE),

        // new child must be born?
        pregnant == 0 =>
        {
          create Bug[
              pos := pos,
              gender := CHILD,
              id := time + bugNum,
              age := 0,
              weight := initWeight,
              pregnant := -1
          ]
        },
        // Decrease pregnant time if bug is pregnant
        pregnant > -1 => pregnant := pregnant - 1
      };
    }
}
```

The next step is to create all of the bugs and food which should be simulated. This could be done in the startUp function. All we have to do is to create two sequences. One for the bugs and one for the food. For that we should create some constants and put them as parameters to the startUp function. So we can easily change the values later without having to change the code and recompile it. The code could look like this:

```
// Constants for bugs
  // number of bugs
  bugNum = 100;
  // constants for the genders
  CHILD  = 1;
  MALE   = 2;
  FEMALE = 3;
  // initial weight
  initWeight = 30;
  // maturity level
  maturity = 20;
  // pregnant interval
  pregnantTime = 3;
```

```
// Constants for food
  // number of available food
  foodNum = 1500;
  // Maximum of avaiable food per cell
  foodMax = 100;
  // percentage of food, which should be eaten by entering a cell
  // value between 1..100
  foodPercentage = 20;

// other constants
  worldSize = 50;
  seedValue = 0;

startUp(bugNum, initWeight, maturity, pregnantTime, foodNum,
        foodMax, foodPercentage, worldSize, seedValue)
{
  seed(seedValue);

  [create Bug[
      pos := [discreteUniform([0..worldSize-1]),
              discreteUniform([0..worldSize-1])],
      gender := CHILD,
      id := i,
      age := 0,
      weight := initWeight,
      pregnant := -1
    ] : i is [1..bugNum]
  ];

  [create Food
    [
      pos := [discreteUniform([0..worldSize-1]),
              discreteUniform([0..worldSize-1])],
      amount := discreteUniform([1..foodMax])
    ] : i is [1..foodNum]
  ];
}
```

The function discreteUniform is predefined and returns a value which is randomly chosen from the given set / sequence. All values have the same probability. Little unhappy fact at this point is that it is possible, that two or more foods are in one cell. So if a bug found some food, he found indeed more foods, but he will eat however all of them.

At last we need to implement the behaviour if two or more agents are in the same grid cell. This could be the case if a bug found some food or two or more bugs meet each other. Because of the fact that this have to be checked in every simulation step, we use a global schedule for this events.

If a bug found something to eat can be checked easily. We only need to compare the positions of every bug-food pair. If both are equal then we now that they have the same position and the bug could eat a little bit.
To check if two or more bugs meet each other we need to check every bug-bug pair. Now we see why the *id* of the bug is useful for us, because we are able to check every pair only once. Otherwise we would check some pairs twice and every bug with himself.
The code looks as follows:

```
// Logic which has to check in every step
schedule cyclic 1
{
  1 :
  {
    // Check if two (or) more bugs are on the same position
    for each a in Bug do
    {
      for each b in Bug do
      {
        a.id < b.id and a.pos == b.pos => bugLogic (a, b)
      };
    },

    // Check if bug has found something to eat
    for each a in Bug do
    {
      for each f in Food do
      {
        // bug found some food?
        a.pos == f.pos =>
        {
          // Yes, he did.
          //-------------
          eatFood(a,f)
        }
      }
    }
  };
}
```

The functions *bugLogic* and *eatFood* implement the description mentioned in section 2.1. Both implementations are also very easy. That is why I think the commented code should be enough documentation:

```
// Bug Logic
bugLogic(a, b) =
{
  // Same gender or one/both is/are child(s)?
  a.gender==b.gender or a.gender==CHILD or b.gender==CHILD =>
```

```
  {
    // Yes.
    //-------------------
    eatEachOther(a,b)
  }
  otherwise
  {
    // => Female bug should get pregnant
    //-----------------------
    a.gender == FEMALE   => a.pregnant := pregnantTime
    otherwise            => b.pregnant := pregnantTime
  }
};
```

```
eatFood (a,f) =
{
  meal == 0 =>
  {
    // Maybe there is still a little bit of food because of
    // integer division. If there is no more food available,
    // the agent could be deleted (performance).
    a.weight := a.weight + f.amount,
    delete(f)
  }
  otherwise
  {
    a.weight := a.weight + meal,
    f.amount := f.amount - meal
  }
}
where meal = (f.amount * foodPercentage) div 100;
```

The function *eatEachOther* used in the *bugLogic* looks as follows:

```
eatEachOther (a,b) =
{
  // Both older than zero? (prevents new childs to be eaten
  // directly by the mother)
  a.age > 0 and b.age > 0 =>
  {
    a.age < b.age =>
    {
      // b is the older bug, so a should be eaten
      // and b should get the mass of a
      b.weight := b.weight + a.weight,
      delete(a)
    }
    otherwise =>
```

```
    {
      // a is the older bug, so b should be eaten
      // and a should get the mass of b
      a.weight := a.weight + b.weight,
      delete(b)
    }
  }
};
```

3. Visualisation

In this chapter we want to learn something about the *Visualization Package*. We will see some basic functionalities and will use them to visualize the bugs simulation introduced in chapter 2.

3.1. How can I visualize something in Fables?

The Visualization Package is a very big pro of Fables. They created a very powerful but also extremely easy to use program. With its help, it is possible to visualize data within seconds without having to implement anything. The *Charting Wizard* helps you a lot and you do not have to do very much. That is why we will directly start with the basic steps of how to use it.
For demonstration we use this easy example:

```
model Example {
  worldSize = 10;
  startUp
  {
    seed(0);
    [create OurAgent[
        pos := [discreteUniform([0..worldSize-1]),
                discreteUniform([0..worldSize-1])],
        gender := discreteUniform(1,2)
      ] : i is [1..25]
    ];
  }
  class OurAgent {
    var pos; // NxN - 2 dimensional
    var gender; // 1 or 2
  }
};
```

To open the Charting Wizard for a specific project just click in the menu bar: *Fables* → *Edit/Create Chart*. Now you should see something like in figure 3.1.

We see a list of available chart types. Of course we could not discuss all of them here. For a deeper introduction see [FabMan08] (Chapter 5). We only will discuss the *2D Grid* because it fits best to our example (and this is of course chosen because of the bugs). But all other charts work similar to this one. So don't be afraid to try something by yourself.
Now lets discuss shortly what we want to see. We have a *position* and a *gender* attribute. For the gender we assume that it could only be 1 or 2. So this is our *color indicator*. Means that the colors we will see depends on these values. The position should correspond as expected to something like coordinates for our grid. So that we have a bijective function

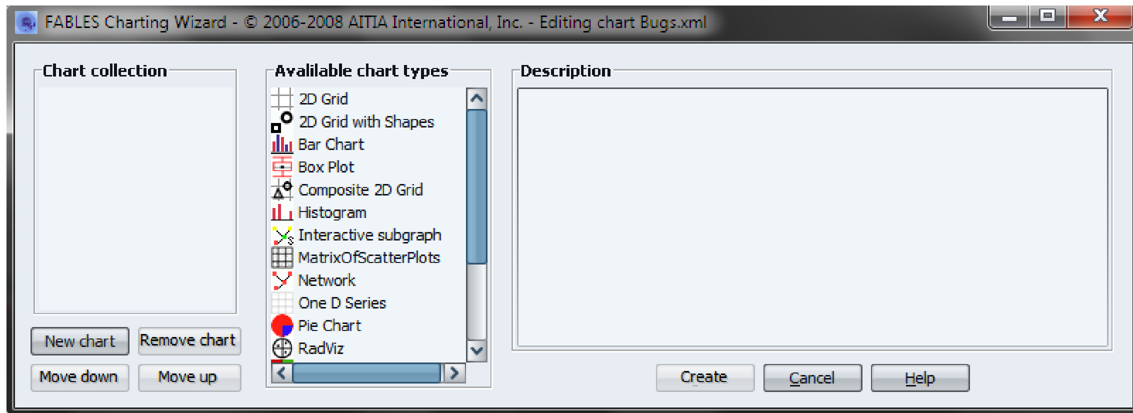

Figure 3.1: Screenshot of the Charting Wizard

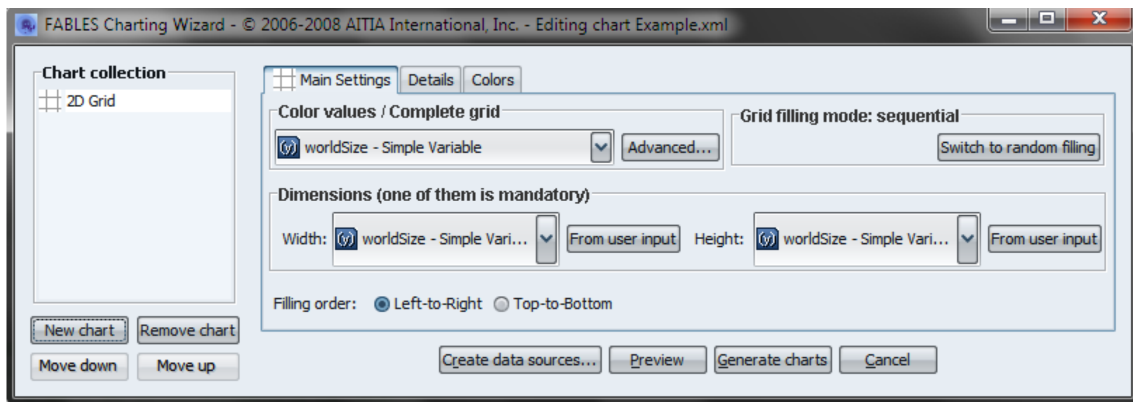

Figure 3.2: Screenshot of the 2D Grid form

between position and grid cell[7]. The task of the constant *worldSize* should be self-evident. So we are ready to transform our thoughts into real actions. Again, as chart type we choose *2D Grid*. So we should see something like in figure 3.2.

The *Main Settings* tab should be visible. If not then click on it. At first we set the size of our grid. So if you take a closer look at the window by yourself, you should agree that *Width* and *Height* in the *Dimensions* section should be our targets. For both we select (if not already done) the variable: *worldSize*. Afterwards we want to set up the *Color values*. The variable we should select in the drop box is: *pos - Collection Variable in class OurAgent*. The tool will automatically transform the values into coordinates for the grid. So don't worry about that you have to do more than that. Furthermore we need

[7]This simply fact is expressed in a very hard way, but so we have an explicit definition. In easier words we would say, that every coordinates pair should point at exactly one grid cell.

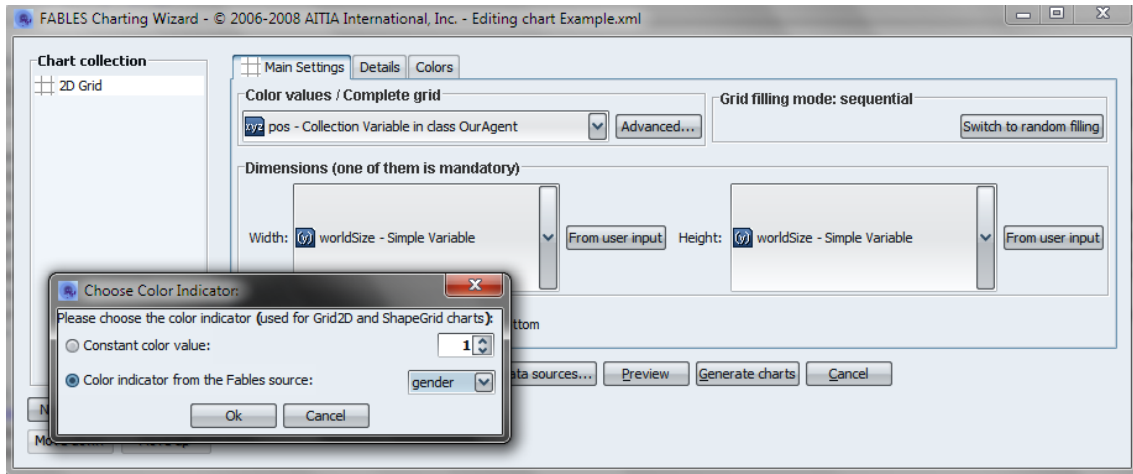

Figure 3.3: Setting up the charting wizard

to choose our color indicator. So we hit the *Advanced...* button and select the second alternative: *Color indicator from the Fables source:* and choose the *gender* variable. If you have done correctly then your selections should be identical to which one seen in figure 3.3.

At last we have to set up the colors for the gender variable. For that choose the *Colors* tab and select for the drop box *Colormap* the value: *Table colormap*. Hit the *Assign color to value* button and you should see something like in figure 3.4 *without* the two rows *1.0* and *2.0*. At these rows to yours and choose a color of your choice for every value.

Ready! To see the result hit *Close* and *Generate charts*. Start the simulation and you should see something similar to figure 3.5

3.2. Visualization of the bugs

Now we should be able to create a visualization for our bugs simulation. In fact it is very similar to the example in section 3.1. So we will not go into detail that much. Lets start directly.
The biggest difference is, that we now have to choose a *Composite 2D Grid* as chart type, because it is only possible to visualize *one* variable per *2D Grid*, but we want to see the *bugs* and the *food*. So we have to create two *2D Grids* and let them overlap. And this is done by the *Composite 2D Grid*. If we have selected this then we see a screen where we could manage some *layers*. To go further we have to create a new layer. The window that now pops up offers the same possibilities known from setting up a *2D Grid*, but it looks a bit different. So lets set it up for the food. If you have all done correctly then it should like the one in figure 3.7. This chart should lie behind the other so we need no transparency. For the color we could use a *Simple Colormap*. So we have a fluent

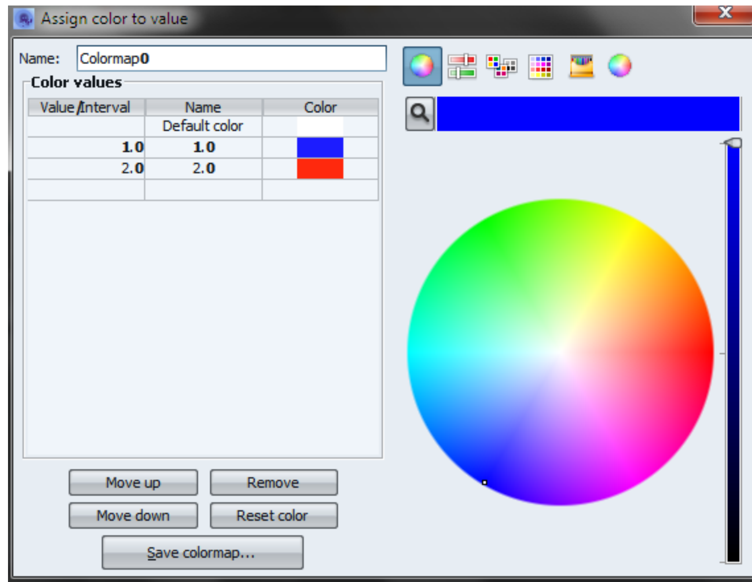

Figure 3.4: Screenshot of selecting the colors

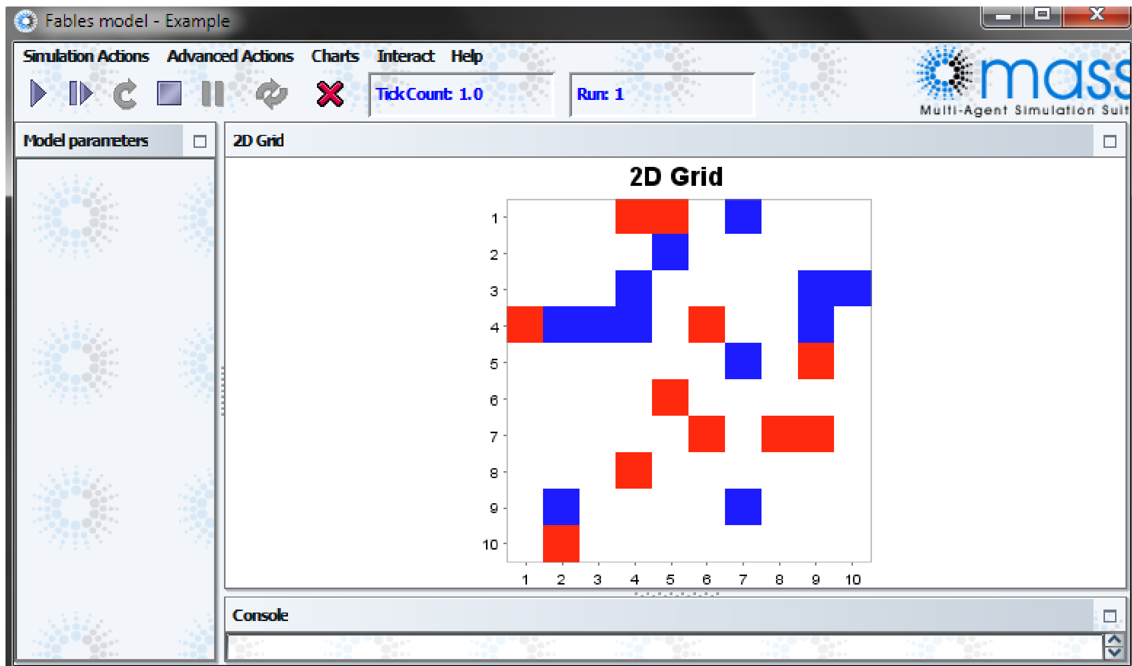

Figure 3.5: Screenshot of the simulation.

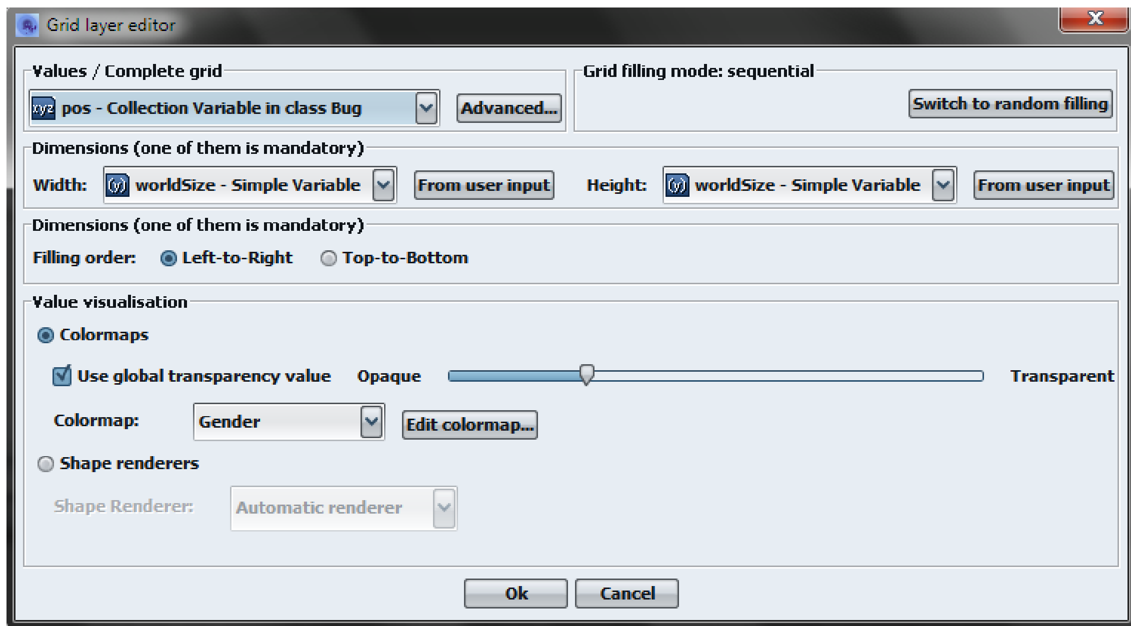

Figure 3.6: Screenshot of the set up for the bugs

transition from *0 food* to *100 food*.
This procedure has to be repeated for the second layer. Figure 3.6 shows a completed form. The transparency value is not that important. Test yourself which value fits best for you. Please mention that sometimes the visualization with the *Compositie 2D Grid* do not work correctly. In my case I got some exceptions during the execution or the colors were wrong. If you have also some problems, then you should create two standard *2D Grid*. One for the bugs and one for the food. These should behave like expected.

The above mentioned chart is more for visualization of the bug movement. But we could also use some charts for statistical investigations. The next example shows how we could use a *Bar Chart* to visualize the population of children, male and female bugs over time. Therefore we have to modify our code a little bit. We have to integrate some counters for each bug group (children, male, female). Every counter should increase if a bug "enters" the corresponding group and decrease if he "leaves". This modifications are not very difficult, but to add them there are changes on many locations necessary. So it would be a nice practice if you do it by yourself.
Now we assume that our three counters *allKids*, *allMen* and *allWom* are integrated correctly. So that we could create our *Bar Chart*. Open the *Charting Wizard* and select *Bar Chart* as *Chart Type*. Now select for the *Data Row* drop box one of our three variables, e.g. bugChildren and hit *Add data row*. Repeat it for the other two missing

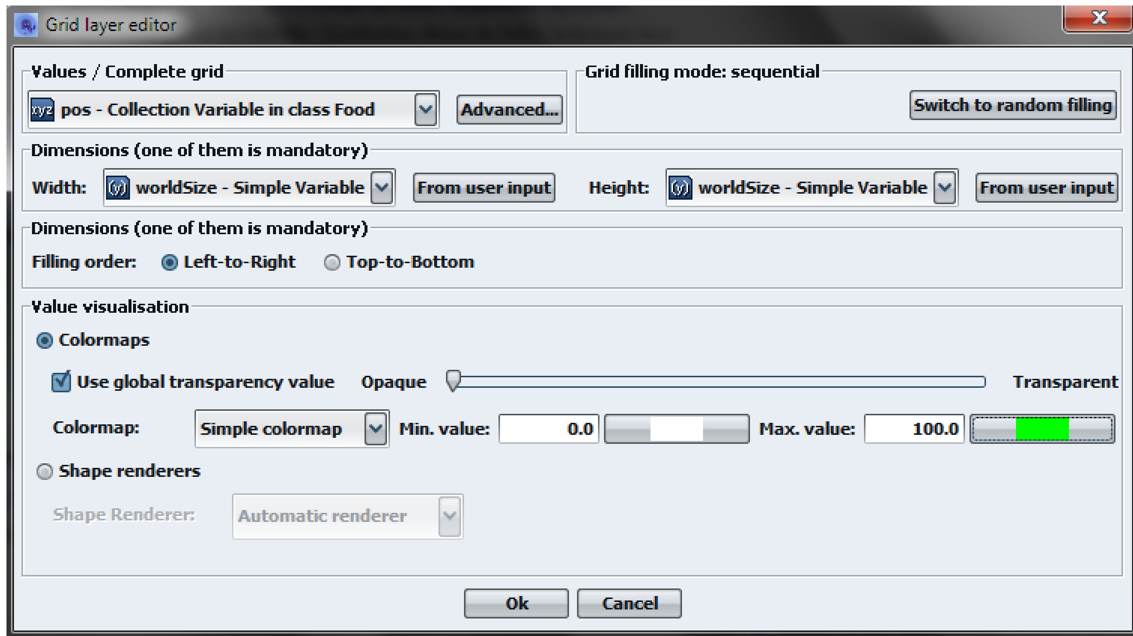

Figure 3.7: Screenshot of the set up for the food

variables. If finished hit *Generate charts*. If you run the simulation for a longer time (at least 300) then your chart should look like the on in figure 3.8.

Figure 3.8: Population of children, male, female bugs in a bar chart

4. Conclusion

If you followed all the instructions and also had understand them, then you should be able to create your own simulations with Fables.
If you want to practice a little bit then it could be good exercise to extend the bugs scenario. For example you could add one (or more) of the following improvements:

- only one food per cell
- food can grow again
- bugs move more natural (bugs move fluently and do not jump)
- bugs get some kind of "battle skill" which is influenced by the age and some kind of "battle experience"
- pregnant bugs lose some weight for giving birth
- own ideas?

A. Complete source code of scenario: bugs growing in grid space

```
//Bugs wandering randomly in 2D world.
model Bugs
{
  // Constants for bugs
    // number of bugs
    bugNum = 100;
    // constants for the genders
    CHILD  = 1;
    MALE   = 2;
    FEMALE = 3;
    // initial weight
    initWeight = 30;
    // maturity level
    maturity = 20;
    // pregnant interval
    pregnantTime = 3;

  // Constants for food
    // number of available food
    foodNum = 1500;
    // Maximum of avaiable food per cell
    foodMax = 100;
    // percentage of food, which should be eaten by entering a
    // cell; value between 1..100
    foodPercentage = 20;

  // other constants
    worldSize = 50;
    seedValue = 0;

  // function to create a doughnut world
  norm (x) = x mod worldSize;


  startUp(bugNum, initWeight, maturity, pregnantTime, foodNum,
          foodMax, foodPercentage, worldSize, seedValue)
  {
    seed(seedValue);

    [create Bug[
        pos := [discreteUniform([0..worldSize-1]),
                discreteUniform([0..worldSize-1])],
        gender := CHILD,
        id := i,
        age := 0,
```

```
        weight := initWeight,
        pregnant := -1
      ] : i is [1..bugNum]
    ];

    [create Food
      [
        pos := [discreteUniform([0..worldSize-1]),
                discreteUniform([0..worldSize-1])],
        amount := discreteUniform([1..foodMax])
      ] : i is [1..foodNum]
    ];
}

class Bug
{
    // position of the bug
    var pos;

    // 2 .. female
    // 1 .. male
    // 0 .. child => not able to reproduce
    var gender;

    // id of the bug
    var id;

    // age of the bug
    var age;

    // weight of the bug
    var weight;

    // pregnant time
    // if 0 then child must be born
    var pregnant;

    schedule cyclic 1
    {
      // When should the event be executed
      // => 1.1 iterations steps after creating the bug
      1.1 :
      {
        // Change position
        pos := [norm(pos(0) + discreteUniform(-1, 0, 1)),
                norm(pos(1) + discreteUniform(-1, 0, 1))],

        // Increase age by 1
        age := age + 1,
```

```
      // Check if bug must die because of starvation
      // if not decrease weight by 1
      weight <= 0 => delete(self)
      otherwise weight := weight - 1,

      // if bug is old enough, then give him a gender
      age == maturity => gender := discreteUniform(MALE,FEMALE),

      // new child must be born?
      pregnant == 0 =>
      {
        create Bug[
            pos := pos,
            gender := CHILD,
            id := time + bugNum,
            age := 0,
            weight := initWeight,
            pregnant := -1
        ]
      },

      // Decrease pregnant time if bug is pregnant
      pregnant > -1 => pregnant := pregnant - 1
    };
  }
}

class Food
{
  // position of the food
  var pos;

  // amount of available food
  var amount;
}

// Logic which has to check in every step
schedule cyclic 1
{
  1 :
  {
    // Check if two (or) more bugs are on the same position
    for each a in Bug do
    {
      for each b in Bug do
      {
        a.id < b.id and a.pos == b.pos => bugLogic (a, b)
      };
```

```
    },

    // Check if bug has found something to eat
    for each a in Bug do
    {
      for each f in Food do
      {
        // bug found some food?
        a.pos == f.pos =>
        {
          // Yes, he did.
          //-------------
          eatFood(a,f)
        }
      }
    }
  };
}

// Bug Logic
bugLogic(a, b) =
{
  // Same gender or one/both is/are child(s)?
  a.gender==b.gender or a.gender==CHILD or b.gender==CHILD =>
  {
    // Yes.
    //------------------
    eatEachOther(a,b)
  }
  otherwise
  {
    // => Female bug should get pregnant
    //----------------------
    a.gender == FEMALE  => a.pregnant := pregnantTime
    otherwise           => b.pregnant := pregnantTime
  }
};

eatEachOther (a,b) =
{
  // Both older than zero? (prevents new childs to be eaten
  // directly by the mother)
  a.age > 0 and b.age > 0 =>
  {
    a.age < b.age =>
    {
      // b is the older bug, so a should be eaten
      // and b should get the mass of a
      b.weight := b.weight + a.weight,
```

```
        delete(a)
      }
      otherwise =>
      {
        // a is the older bug, so b should be eaten
        // and a should get the mass of b
        a.weight := a.weight + b.weight,
        delete(b)
      }
    }
  };

  eatFood (a,f) =
  {
    meal == 0 =>
    {
      // maybe there is still a little bit of food
      // because of integer division.
      // If there is no more food available, the agent
      // could be deleted (performance)
      a.weight := a.weight + f.amount,
      delete(f)
    }
    otherwise
    {
      a.weight := a.weight + meal,
      f.amount := f.amount - meal
    }
  }
  where meal = (f.amount * foodPercentage) div 100;
};
```

References

[AITIA] Marton Ivanyi, Laszlo Gulyas, Richard Legendi, Sandor Bartha, Vilmos Kozma, Robert Meszaros and Vilmos Kozma, *Multi-Agent Simulation Suite*, Budapest

[FabMan08] AITIA International Inc, *Functional Agent-Based Language for Simulation (Manual)*, Budapest, 2008

[RaLyJa] Steven F. Railsback, Steven L. Lytinen, Stephen K. Jackson, *Agent-based Simulation Platforms: Review and Development Recommendations*