

## How to Use the *jES Open Foundation Program*<sup>Y</sup> to Discover Complexity in Multi-Stratum Models

Pietro Terna

(October 2004)

Dipartimento di Scienze economiche e finanziarie G.Prato, Università di Torino, Italia  
[pietro.terna@unito.it](mailto:pietro.terna@unito.it)

I shall discover all: I'll bring thee word  
Straight, how 'tis like to go.  
Shakespeare, Antony and Cleopatra

?. WORKERS, SKILLS AND FIRMS

a first trace in **apps/**

?. RULES AND USE

[to use *jESOF* for a large simulated time extension (a lot of ticks), we used to run **make runBig**: now it is fixed with my hibernate trick, i.e. the dropped object are used again after a new initialization]

internal unit numbers (also printed on the active terminal if some error arises) are different from those contained in **unitData?/ unitBasicData.txt** file (where ? is the stratum number); the internal number is indeed obtained from that of the file and adding to it two zeros in case of a unit created while the simulation is running or a progressive number from **00** to **99** in case of initial creation (multiple creation requires this addition to the number)

when we assign (via **OrderDistiller** or via **OrderGenerator**) an order, if no unit can receive it, the potential production (obviously lost) is anyway accounted

if more than a unit can perform a task, the first one in the unit list is used; look below, the unit lists are randomly reordered

the order of the sequence of the models and of the unit within a model is randomly changed in each cycle, via the schedule of each model (the order of the models is changed only by one of the models; each model reorders its unit list); this is generally sufficient to avoid any systematic effect in accessing to units

anyway, if we assign a lot of equal orders in a cycle, all the orders are kept by the same (first) unit able to do the first step, regardless to all the other units able to do the same step; to avoid this effect, **assigningTool.java**, when executing the **assign()** method, do not repeat assignment to the same unit if **uniqueAssignmentInEachCycle** is set to *true*; the default is

---

<sup>Y</sup> related to *jesopenfoundation-0.1.60*

*false*, to speed up the execution and for compatibility reasons with the previous jESevol simulations

the choice above is not convenient if we have a quantity of assignments that overcome the that of the unit in each cycle (after having assigned to each unit an order, all the remaining assignment are lost); in this case it is better to shuffle the list at each assignment, admitting more than one assignment to the same unit in the same cycle (also having some unit without assignments); this is done if the `shuffleListsAtEachAssignment` is set to *true*; the default is *false*, to speed up the execution and for compatibility reasons with the previous jESevol simulations

When do work `uniqueAssignmentInEachCycle` and `shuffleListsAtEachAssignment`?

For `uniqueAssignmentInEachCycle` the choice *true* is active for the units belonging to the stratum where the choice is made, independently from the stratum of the sending unit or of the order generator/distiller.

For `shuffleListsAtEachAssignment` the choice *true* is active for the units receiving an order (with a unit or the order generator/distiller as sources) from the stratum where the choice is made, independently from the stratum receiver.

use carefully `uniqueAssignmentInEachCycle` and `shuffleListsAtEachAssignment` together.

NB NB if a zero time step uses the same production phase or the unit the order is in, the order is LIFO reassigned to the same unit and executed, regardless the `uniqueAssignmentInEachCycle` value; this shortcut operates only for assignment from unit to unit, also in the hidden case of the added zero time steps always used to conclude the computational steps

zero time step are in the form `1111 s 0`, which are immediately executed in a clock tick regardless of how many they are (1111 here is a fictitious production unit).

the min level of inter-visibility; below this level two units cannot exchange products; this value is not used in assignment coming from OrderGenerator; this value can be different from one model to another: **that of the sending unit is used**

between to units, if the second (destination of the order) does not belongs to the area of the first (sending units) inter visibility does not work; the inverse situation works

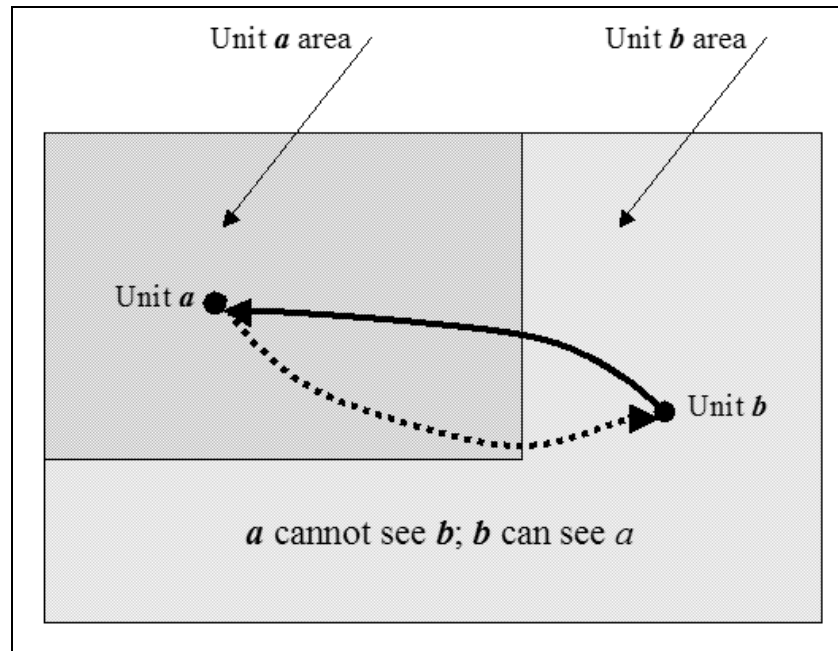


Figure 1. xxx

Using computational steps we can also use general matrixes; if this is the case, we have to create a `memoryMatrixes.txt` file in `unitData0/` folder; the other models (in stratum 1, 2, ...), if existing, have to copy all information about general matrixes from the first model (that in stratum 0). The first row of the file must contain exactly the following text sequence “number(from\_0\_ordered)\_rows\_cols”.

Each unit as a public memory matrix of 10 rows and 10 cols; all the element of each matrix are initially filled with 0. Units have the `printDirectlyMemoryMatrix()` method that prints their memory matrix on the active terminal, regardless the `printMatrixes` value. This method is introduced mainly to be used via the probe of the unit). -1 is the conventional number for unit matrixes.

We have also a probe, from the observer, to each memory matrix; placing the matrix number (remember, starting from 0), in the right cell of the `printMemoryMatrixNumber` probe, pressing the enter button of the keyboard to confirm the inserted value and finally pressing the `printMemoryMatrixNumber` button itself we obtain the content of the matrix in the active terminal. Remember: before accessing the matrixes, the program has to be run to execute its starting tasks.

Placement rules for a new unit: when a new unit is created, if its destination position in unit space is occupied (and when a copy is made, the destination position is always occupied), we search for a free position in one of the eight possible directions around the initial position, moving forward following a straight line until we find a free position; the max movement is conventionally set in `unitSpaceXSize + unitSpaceYSize` steps. After that, the unit is not created.

As in `jesframe`, zero time orders are introduced in units in a LIFO way.

If `maxInactivity == 0` units are never dropped if inactive; the same if `maxUnsentProducts == 0`, the units are never dropped if unable to send products. In both cases the unit histograms (down and right) are not shown.

Unsent orders, when becoming a lot, can slow down the simulation, for useless attempt of assignments (e.g. preys not finding grass to eat and not perishing for some reason); in this case use the `maxTicksAsUnsent` parameter, related to the units of each stratum, stating the max number of ticks the orders can spend in the unsent condition in a unit; when their time expires, the orders are dropped.

To run a simulation press twice the **Start** button; the double use of the button is necessary to allow us to change parameters in the observer pane, obtaining that these changes act on the models created with the second hit to **Start**. We can record the position of the various windows on the screen pressing **Save**.

## ?. APPLICATIONS

Applications are:

- `test0_generator(basic_run)`: up to 4 independent models with `orderGenerator`
- `test1_distiller`: 2 or more models with `orderDistiller`; models 0 and 1 are interdependent (recipes and sequence are in 0); model 2 and 3 if used do not receive orders (but we can switch them to `orderGenerator`)
- `test2_distiller_generator`: 4 models, with 0 and 1 using `orderDistiller` and interdependent; 2 and 3 with `orderGenerator`, but 3 receiving also recipes from model 0 `orderDistiller` (recipes and sequence are in 0)
- `test3_distiller_comp_steps`: 2 models, with `orderDistiller` and computational steps; models 0 and 1 are interdependent (recipes and sequence are in 0, with test computational 1998 and 1999 and with a recipe containing a zero time step);
- `test4_11xx_comp_steps`: 3 models, only the third – used strictly as a recipe launcher – with `orderDistiller`; the recipes contains 1100, 1101, 1102 and 1103 computational steps, specific of the jeSOF environment (matrix values loader, unit creation, unit drop and unit copy, also with zero time steps); we have here two different `.smc` files; that not active contains the shuffle choice (see above), without queue in sending orders (which appear if we chose uniqueness of assignment, due to the impossibility/difficulty of finding a free receiving unit)
- `firstEvolApplication`: the jESevol app used for SwarmFest 2004, running also in jESOF; use the x.y launching files;
- tutorial, with several steps (see above, paragraph ?.?):
  - `step1`: grass creation (the grass is the food of the preys);
  - `step1`: the preys' life (the preys are the food of the predators);
- `workers_skills_firms`, with
  - `workers_skills_firms_v_0`, where ...

#### 4.4. COMPUTATIONAL CAPABILITIES AND MEMORY MATRIXES

*jES* has computational capabilities that can be associated to each step of a recipe. To use this feature of the program it is necessary to understand the Java language, as we have to modify<sup>2</sup> the `ComputationalAssembler.java` file (which inherits its default methods from the class `ComputationalAssemblerBasic`). Computational capabilities are aimed to forecasting, evaluations, auctions to choose procurements, ...

Memory matrixes data are reported in a text file (unitData/memoryMatrixes.txt)		
number(from_0_ordered)	rows	cols
0	2	3
1	3	5
2	4	1
3	3	1

Mandatory first line

**Figure 18?** An example of memory matrixes declarations: the ID numbers are ordered and start from zero.

Computations use data contained in memory matrixes created according both the `totalMemoryMatrixNumber` parameter and the contents of the file `unitData/memoryMatrixes.txt`, shown, as an example, in Figure 18. Memory matrixes use order layers (see above 2.1.) in a completely automated way; we can prevent them from using layers setting their ID number as negative in each specific declaration into the file `unitData/memoryMatrixes.txt`. In the example reported here, the second matrix (numbered 1, being 0 the number of the first one) is insensitive to layers<sup>3</sup>.

Examples of recipes containing computational steps are reported in Figure 19; obviously, to understand the meaning and the behavior of a computation it is necessary to consider together both the sequence of the events emerging from the various orders in execution (with the

<sup>2</sup> We have not to modify the basic file (`ComputationalAssemblerBasic.java`), which is included in the `src/` folder. Instead, we have to copy the file `ComputationalAssembler.java` from `src/` in the main folder. The 'make run' command uses the classes contained in `lib/jesframe.jar` (which are those contained in `src/`), but the classes in `/.` override those in `jesframe.jar`.

`ComputationalAssembler.java` contains no method; we simply add methods, following the examples reported below and using as a guide the full code or the methods reported in `ComputationalAssemblerBasic.java`. New methods are automatically used by the `checkingComputationsAndFreeingOrders()` method of `ComputationalAssembler` class (which inherits it from its parent class): the trick used to convert the numerical code of the computational steps into a recognized method reference is based on the Java reflection mechanism. To understand the trick, looks at the following lines in `ComputationalAssemblerBasic.java` code:

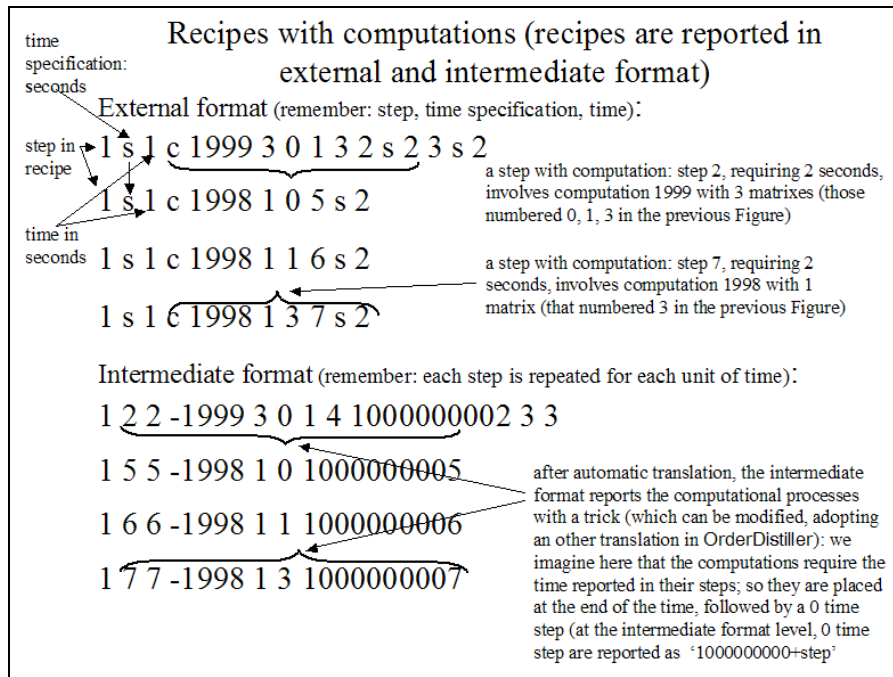
```
Class c = this.getClass();
Method m = c.getMethod("c"+(-1*t), null);
m.invoke(this, null);
```

<sup>3</sup> -0, as a number, is equal to 0, so the first matrix cannot be declared insensitive to layers.

related operations interesting the memory matrixes) and the content of the Java code of the computational operator itself.

It is important here to consider both the external (human readable) format of the recipes and the intermediate one, always human readable, but semi-translated (see above 4.1.). Code numbers of the computational steps are established in the range 1001-1999.

The format of a computation is: '**c code n m<sub>1</sub> ... m<sub>n</sub>**' where **c** is mandatory, **code** is the code of the computation, **n** is the number of matrixes to be used and '**m<sub>1</sub> ... m<sub>n</sub>**' are the ID numbers of those matrixes, as reported in the file `unitData/memoryMatrixes.txt` (Figure 18).



**Figure 19?**. The format of the computational processes.

We introduce some recipes (Figure 19) with computations as a complete example, to explain the dynamics of the events and the Java code related to them. To prepare other computational tools, we have to add lines similar to those introduced below (Figures 20 and 21) into the `ComputationalAssembler` class (`ComputationalAssembler.java`, as explained in note above).

In Figure 19 we can see how computational codes are represented following their external and intermediate formats (anyway, remember that we write recipes in external code). Pay attention: computational codes at the intermediate format representation level are reported as negative, following the internal convention of *jES*, where all the codes related to production steps are positive, while numbers bearing special meanings are negative.

The Java codes, extracted from `ComputationalAssembler.java` and reported in Figures 20 and 21, interact with the recipes of Figure 19.

When an order with recipe '1 s 1 c 1998 1 0 5 s 2' is executed, at the end of the two units of time required by step 5, matrix 0 is interested by a writing operation in position (0,0) in the proper order layer (determined by the level of the order containing the recipe); if the order contains the recipe '1 s 1 c 1998 1 1 6 s 2' the writing operation, at the end of step 6,

concerns matrix 1 at position (0,0) without layer, being that matrix insensitive to layers by construction; if the order contains the recipe ‘1 s 1 c 1998 1 3 7 s 2’, the writing operation, at the end of step 7, concerns matrix 3 at position (0,0) in the proper layer, as above. In the Java code of Figure 20 we can see how these operations, made on mm0 matrix (but we can use any name), are related to the actual matrix via the `getMemoryMatrixAddress` method; the `setValue` method set the 1.0 value at position (0,0). If the matrix is insensitive to layers, the layer value set in this method is disregarded. Finally, the computational step is set to `done`<sup>4</sup>.

**The Java Swarm code used by the recipes with the  
example code c 1998**

```

/** computational operations with code -1998 (a code for the checking
 * phase of the program
 *
 * this computational code place a number in position 0,0 of the
 * unique received matrix and set the status to done
 */
public void c1998(){

    mm0=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    mm0.setValue(layer,0,0,1.0);
    mm0.print();

    done=true;
} // end c1998

```

**Figure 20?** The Java code (simplified eliminating a control statement related to the consistence of the declared number of matrixes with the internal ones).

When an order with the recipe ‘1 s 1 c 1999 3 0 1 3 2 s 2 3 s 2’ is executed, at the end of the two units of time required by step 2, a check (see Figure 21) is performed on matrixes 0, 1 and 3 to verify whether positions (0,0) are empty at the proper layer; if they are not empty, the ‘c 1999’ computation sets those positions (at the proper layers) to empty and finally sets the computational step to `done`<sup>5</sup>. Into the code of this example, the matrixes mm0, mm1 and mm2 are linked to actual matrixes 0, 1 and 3 via the list ‘0 1 3’ used into the recipe (the internal names are completely free).

The effect of those four recipes (the `OrderGenerator`, while testing the program, if `totalEndUnitNumber` is greater than 0, launches those recipes at random) is the following: the recipe containing the code ‘c 1999’ cannot proceed to step 2 if the effects of one of each of the recipes containing codes ‘c 1998’ do not exist (those effects are produced when recipes are executed at least at step 5 or 6 or 7). When the recipe containing the code ‘c 1999’ finally

<sup>4</sup> If the Java code related to a computational method does not set the `done` variable to `true` the order is not freed and does not proceed to its successive recipe step; the computational step will be repeated in any simulation cycle, until the `done` variable becomes `true`.

<sup>5</sup> See previous note.

proceeds to its successive step, the effects of the “used” recipes are eliminated and must be renewed by other similar orders.

Methods accepted by the `MemoryMatrix` instances are `setValue`, `getValue`, `setEmpty`, `getEmpty` (returning *true* or *false*).

**The Java Swarm code used by the recipes with the  
example code c 1999**

```

/** computational operations with code -1999 (a code for the checking
 * phase of the program
 *
 * this computational code verifies position 0,0 of the three
 * received matrixes; only if these positions are all not empty
 * the code empties them and set the status to done
 */
public void c1999(){
    mm0=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(0);
    mm1=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(1);
    mm2=(MemoryMatrix) pendingComputationalSpecificationSet.
        getMemoryMatrixAddress(2);
    layer=pendingComputationalSpecificationSet.
        getOrderLayer();

    if(! (mm0.getEmpty(layer,0,0) || mm1.getEmpty(layer,0,0)
        || mm2.getEmpty(layer,0,0) ) )
    {
        mm0.setEmpty(layer,0,0);
        mm1.setEmpty(layer,0,0);
        mm2.setEmpty(layer,0,0);
        done=true;
    }
} // end c1999

```

**Figure 21?** The Java code (simplified eliminating a control statement related to the consistence of the declared number of matrixes with the internal ones.

The syntax is (leave `layer` as is and set the proper value of the variable as shown in the examples):

- `setValue(layer, (int) row, (int) col, (double) value)` or `setValue(layer, (int) row, (int) col, (float) value)`
- `(float) getValue(layer, (int) row, (int) col)`
- `setEmpty(layer, (int) row, (int) col)`
- `(boolean) getEmpty(layer, (int) row, (int) col)`

Where the `setEmpty` and the `getEmpty` methods are useful to manage conditional situations; to set to “not empty” a position of a matrix, we simply put a value in it; `getEmpty` returns *true* if no value is found, otherwise it returns *false*.

To look directly to the content of a matrix we can use the `print` method, as shown above in Figure 20; if, in the probe of the observer, the field `printMatrixes` is set to *true*, the `print`



method displays the content of the matrix on the current terminal; the empty positions of the matrix are reported as not available (NA).

#### 4.4.1. A SPECIAL CASE: RECIPES LAUNCHING RECIPES

A special case is that of recipes launching other recipes, via the computational step 1002: in '1 s 1 c 1002 1 3 7 s 2' at the end of step 7 (which, in this case, is lasting 2 units - of type s - of time). The computational step launches the recipe whose code is contained in position (0,0) of the matrix with code 3.

We can use *m* memory matrixes (of dimensions 1x1 or bigger) to hold the codes of recipe *s* to be launched from other recipes.

To fill the (0,0) positions of these matrixes with the recipe codes, we use the computational step 1001, with a recipe containing 'c 1001 3 0 1 3 100 s 1' (suppose here *m*=3); here unit 100 can be a fictitious one, used only to allow this kind of computations. The recipe codes to be placed in matrixes 0, 1 and 3 (note that it is not necessary that they are ordered and consecutive) will be retrieved in `recipeData0/recipesFromRecipes.txt`, written in free format. Two restrictions: first, a maximum of 10 recipe codes to be launched is allowed (but can be modified in `ComputationalAssemblerBasic.java`); second, a maximum of 1.000 launches is allowed per tick (but can be modified in `OrderDistiller.java`). NB underline that in `jESOF recipesFromRecipes.txt` must be in `recipeData0/`.

#### 4.4.2. COMPUTATIONAL STEPS DEALING WITH UNITS

From 1101 to 1199 we can have computational steps dealing with units; these are definitive computational steps, kept in `ComputationalAssemblerBasic.java`, in the `src/` folder.

The memory matrixes are referred here by absolute positions; this positions can be changed by the *ad hoc* computational steps using the row and column displacements (see below the preys-predators tutorial).

At present we have:

1100 –this computational step acts as generalized matrix loader that places the values found in `unitData0/memoryMatrixContents.txt` file in the matrixes and rows and cols reported in the same file; the number of the matrixes used here must be less or equal to `totalMemoryMatrixNumber` (the first *m* matrixes of these ones); beside this, *m* has to be less than 10; it is mandatory to enter all the *m* defined matrixes in their original order in the `c 1100 m m1 m2 ...` command; values in the `unitData0/memoryMatrixContents.txt` can be in any order; finally this computational step changes the status to done; may be this kind of computational steps is included in a recipe launched by an `OrderStartingSequence.xls` file;

1101 – this computational code creates a new random unit in the model stratum reported in position 0,1 of the unique received matrix, with the probability set in position 0,0 of the same matrix; it chose randomly a unit in the pro-forma unit list, generated from the stratum `unitBasicData.txt` file, from the top to the 'potentialUnitTypes-1' row; finally it changes the status to done;

1102 – this computational code drops the unit the order is in, with the probability set in position 0,0 of the first received matrix and increases the position 0,1 of the second received matrix by 1, to count the dropped units; this computational step must be in the last step of the

order, because the dropped unit cannot send it to the successive one; the computational step, before setting the unit to be dropped, applies the `orderDoneStep()` method to the order, for accounting purposes; finally it changes the status to done;

**1103** – this computational code creates a copy of the unit the order is in, in the model stratum of the original unit, near to it with the standard placement rule, with the probability set in position 0,0 of the first received matrix; the count of the created units is reported increasing by one the position 0,1 of the second received matrix; finally it changes the status to done;

**1104** – this computational code creates a copy of the unit the order is in, in the model stratum of the original unit, in a random position with the standard placement rule, with the probability set in position 0,0 of the first received matrix; the count of the created units is reported increasing by one the position 0,1 of the second received matrix; finally it changes the status to done

**1110** – this computational code creates a copy of the unit the order is in, in the same x,y place of the copied unit, but in another model stratum, with the probability set in position 0,0 of the unique received matrix; the created unit is set in the stratum indicated in position 0,1 of the unique received matrix; the unit is created, with the above probability, if the content of the unit memory matrix in position 0,0 is positive (the actual position can be modified via the calling computational step, i.e. here **1210**); if the destination position in the new stratum is not empty, the new unit is not created, without any advice; finally, the computational step changes the status to done; **1110** is not normally utilized directly, but via a special computational step (i.e. **12xx** below) setting the position and content of the switch cell ( $\leq 0$  or  $> 0$ ) indicated above; the memory matrix of the copied unit is shared among the original unit itself and its copies, avoiding the creation of new matrixes for the copies<sup>6</sup>; the new unit is not created if the original one is in the destination stratum, so we can easily avoid copies of copies, but also easily obtain them, using a destination stratum as an original one in a different recipe; the new unit has the same visibility of the old one;

**1111** – this computational code acts via the **1110** code, to creates a copy of the unit the order is in, as above; as **1110**, **1111** is not normally utilized directly, but via a special computational step (i.e. **12xx** below); the difference with **1110** is that here we specify a fixed value to be added to the production phase of the unit when we copy it (the new unit has a production phase that of the original one plus the fixed value); in this way we can differentiate the original and copied units from the point of view of the production phase; the addendum value is stored (as an integer value) in position 0,2 of the unique received matrix;

**1120** – this computational code modifies the position 0,0 both of the memory matrix of the unit the order is in and of the memory matrix of the unit the order comes from; the amount of the two modifications is defined by two addenda stored in position 0,0 and 0,1 of the unique received matrix (the actual position can be modified via the calling computational step, of the class **12xx**); also the position of the two memory matrixes can be modified via the **12xx** calling computational step; in 0,0 the addendum 1 for the unit the order is in and addendum 2 for the unit the order comes from; finally it changes the status to done;

**1199** – this highly usable computational code, when received by one unit in a stratum, acts vs. all the units of the same type (i.e., with the same production phase) in all strata; actions: with the probability set in position 0,0 of the first received matrix, **c1999** adds the value

---

<sup>6</sup> Dropping an original or a copied unit, we do not drop the memory matrix, which belongs to the whole set of unit coordinated units. Memory matrixes contain an integer field counting the number of units using them; when this field is 0, the memory matrix becomes droppable.

contained in 0,5 of the second received matrix to each unit memory matrix a row and column reported in pos 0,3 and 0,4 of the second matrix; if the resulting content is  $<$  the level contained in position 0,2 of the second matrix, the considered unit is dropped (if it is the unit the order is in, also the order is dropped), counting the dropped units in position 0,1 of the second matrix (obviously, first and second matrixes can be coincident);

#### 4.4.3. UTILITIES

From 1201 to 1299 we can have computational steps used only in specific projects.

As examples we have the 12xx steps used in the tutorial below.

#### ?.?. TUTORIAL

The tutorial is related to the implementation of a Predator Prey model<sup>7</sup>, with three strata:

- i. the grass stratum, where we have the grass continuously growing; if the preys are too dense in the space, grass growth rate may be insufficient to assure the food for all of them;
- ii. the prey stratum, being the preys the food of the predators (two versions);
- iii. the predator stratum.

#### ?.?. ?. THE TUTORIAL STEP 1: THE GRASS

In level (i) we create a unique type of units (the grass), e.g. 10, with initial creation prob.= 1 and new unit generation prob.= 0 in successive cycles, in a 20×20 square; each unit has eight small squares of fixed visibility around it (the visibility is used by the preys to find and eat the grass). We use **OrderDistiller** as external source of the events, launching a certain number of events in each cycle of the simulation.

We launch, in each cycle, following **recipeData0/orderSequence.xls**, the recipe **grassGeneration** (look at **recipeData0/recipe.xls** spreadsheet file) containing a c1103 computational steps, which tells to its receiving unit of creating a copy of itself, placing it in its neighborhood. The 1103 computational step looks at its unique matrix to read, in (1,0), the probability of acting (0.8 in our case). We have one memory matrix; its content is loaded via the 1100 computational step contained in the utility recipe in **recipeData0/recipe.xls**; this step load the values contained in **dataUnit0/memoryMatrixContents.txt**.

Having **maxInactivity** = 0 and **maxUnsentProduct** = 0 units never die.

The utility recipe is launched only once in the simulation, via **recipeData0/orderStartingSequence.xls**; the **grassGeneration** recipe is launched repetitively in each cycle of the simulation (10 in each cycle), via **recipeData0/orderSequence.xls**; we can change both the quantity in each cycle (try for example with 50, with a logistic effect in the unit number curve), both with **uniqueAssignmentInEachCycle** set to *true* (as is) and alternatively with

---

<sup>7</sup> Remember O. Malcai, O. Biham, P. Richmond and S. Solomon, Theoretical Analysis and Simulations of the Generalized Lotka-Volterra Model, file 0208514.pdf in my PC.

`shuffleListsAtEachAssignment` set to *true* and the other parameter switched off (try also to switch off both).

We can also use `OrderGenerator` as source of external events (with small unused recipes in this case, having set `maxStepNumber` = 1 and `maxStepLength` = 1 in random recipes), setting by hand (interactively) `newUnitGenerationP` to a non zero value (also to 1).

Obviously we can start with immediately more grass, as in the next step, but anyway the process of grass development has to be managed.

### ???. THE TUTORIAL STEP 2A: THE PREYS

Also in level (ii) we create a unique type units, that of the preys. Then we launch, in each cycle, following `recipeData1/orderSequence.xls`:

- a. orders with recipe n.1 '`preysEating`', containing the computational operations with step `c1202` (a code for the prey-predator tutorial); this computational code acts via `c1102` code to drop the unit the order is in, with the probability set in position 0,0 of the first received matrix and increases the position 0,1 of the second received matrix by 1, to count the dropped units; finally, it adds 1 to pos. 0,0 of the sending unit memory matrix, to count its eating acts;
- b. orders with recipe n.2 '`preysPerishing`' (unsufficient food, i.e. energy), containing the computational code `c1204` (a code for the prey-predator tutorial); this computational code acts via `c1102` code to drop the unit the order is in, if pos. 0,0 value of the unit memory matrix is less then the parameter in 1,2 (`minEnergy`) of the second received matrix; `c1102` acts with the probability set in position 1,0 of the first received matrix (internal trick: `c1102` looks for `0+rd,0+cd` using temporary `rd`, the row displacement, set to 1; `c1102` increases the position 1,1 (always via the displacement, being the internal coordinates `0+rd,1+cd`, of the second received matrix by 1, to count the dropped units.
- c. orders with recipe n.3 '`preysConsuming`' (decreasing energy), containing the computational code `c1205` (a code for the prey-predator tutorial); this computational code acts with the probability set in position 2,0 of the unique received matrix to decrease of 1 unit the position 0,0 of the memory matrix of the unit the order is in;
- d. orders with recipe n.4 '`preysReproducing`' (creating a copy of themselves near to their position, if their energy is sufficient), containing the computational code `c1206` (a code for the prey-predator tutorial); this computational code acts via the `c1103` code to create, with the probability set in position 3,0 of the first received matrix (`c1103` see it with `rd=3` as `0+rd,0+cd`), a copy of the unit the order is in, near to it, if the unit the order is in has sufficient energy (comparing position 0,0 of the unit memory matrix and position 3,2 of the second received matrix, i.e. if  $\text{energy} \geq \text{requested level to reproduce}$ ); using `c1103` we have also in 0,1 of the second matrix the count of the created unit (here in position 3,1 of the second received matrix, with `rd=3`).

The content of the memory matrixes is now:

matrix 0

Probability for recipe 1 in stratum 0, grass creation	Count of the created grass
---	----------------------------

matrix 1

acting probability for recipe 1 in stratum 1, preys eating grass	count of grass ate (dropped)	NA
acting probability for recipe 2 in stratum 1, preys perishing	count of perished preys (dropped)	perishing if energy in 0,0 of unit memory matrix is < the level set here
acting probability for recipe 3 in stratum 1, preys consuming	NA	NA
acting probability for recipe 4 in stratum 1, preys reproducing	count of the created preys	Reproducing if energy in 0,0 of unit memory matrix is $\geq$ the level set here

The structure of step 2a is complex and difficult to drive to obtain a planned result, but can be very rich, with a lot of parameter, for a selection-evolution process employing genetic algorithms.

???. THE TUTORIAL STEP 2B: THE PREYS, SECOND ALTERNATE VERSION

In level (ii) again we create a unique type units, that of the preys. Then we launch, in each cycle, following `recipeData1/orderSequence.xls`, a simplified set of recipes, if compared to that of **step2a**, both to speed up the simulation and to obtain a better control of events like the **preysConsuming** and the **preysPerishing**, which must attain once all the units in each cycle (the other two recipes are the same of step 2a):

- orders with recipe n.1 '**preysEating**';
- one order with a general recipe n.23 '**timeStep**', using C1199 via C1299 (a code for the prey-predator tutorial) computational step to decrease position 0,0 of the memory matrix of each unit by the content of position 0,5 of the second matrix (preys consuming energy) and dropping the unit if its energy is < of position 0,2 of the second matrix (preys perishing); the usual trick of `rd=1` is used here;
- orders with recipe n.4 '**preysReproducing**'; dropped unit in b) can anyway reproduce within the cycle, being the drop put into effect at the end of each cycle. If we want avoid this effect, we can check the units in c1206, forbidding the reproduction of unit when the droppable quality set to *true*.

Memory matrixes 1 is now:

acting probability for recipe 1 in stratum 1, preys eating grass	count of grass ate (dropped)	NA	NA	NA	NA
acting probability for recipe 23, time step (preys consuming and, if any, perishing),	count of perished preys (dropped)	Perishing if energy in row,col of unit memory matrix is < the level set here	Row of the memory matrix we refer to	Col of the memory matrix we refer to	value to be added to the row,col position of unit memory matrix
NA	NA	NA	NA	NA	NA
acting probability for recipe 4, preys reproducing	count of the created preys	Reproducing if energy in 0,0 of unit memory matrix is $\geq$ the level set here	NA	NA	NA

The **step2b** chosen parameters creates a world where the weight of preys on the environment destroy the grass; consequently the preys disappear. A set of parameters producing this result is easily found. With the **step2a** structure instead it is not easy to manage the results to obtain, by hand, a desired result. We have to experiment about the following proposition, but it is highly probable that using genetic algorithms to find the parameters useful to obtain a given solution we can more easily work with a widely open form like that of **step2a** than with that of narrow form used with **step2b**.

### ???. THE TUTORIAL STEP 3A: THE PREDATORS

We use here the scheme of **step2b**, introducing the **memoryMatrix n.2**.

In level (iii) again we create a unique type units, that of the predators. Then we launch, in each cycle, following **recipeData2/orderSequence.xls**, the recipes controlling directly events like the **predatorsConsuming** and the **predatorsPerishing**, which must attain once all the units in each cycle (the other two recipes are the same of step 2a):

- orders with recipe n.1 '**predatorsEating**';
- one order with a general recipe n.23 '**timeStep**', using C1199 via C1299 (a code for the prey-predator tutorial) computational step to decrease position 0,0 of the memory matrix of each unit by the content of position 0,5 of the second matrix (preys consuming energy) and dropping the unit if its energy is < of position 0,1 of the second matrix (preys perishing): usual trick of  $rd=1$  here is used;
- orders with recipe n.4 '**predatorsReproducing**'; dropped unit in b) can anyway reproduce within the cycle, being the drop put into effect at the end of each cycle. If we want avoid this effect, we can check the units in c1206, forbidding the reproduction of unit when the droppable quality set to *true*.

Memory matrix 2 is:

acting probability for recipe 1 in stratum 2, predators eating preys	count of preys ate (dropped)	NA	NA	NA	NA
acting probability for recipe 23 in stratum 2, time step (predatorss consuming and, if any, perishing),	count of perished predators (dropped)	Perishing if energy in row,col of unit memory matrix is < the level set here	Row of the memory matrix we refer to	Col of the memory matrix we refer to	value to be added to the row,col position of unit memory matrix
NA	NA	NA	NA	NA	NA
acting probability for recipe 4, predators reproducing	Count of the created predators	Reproducing if energy in 0,0 of unit memory matrix is $\geq$ the level set here	NA	NA	NA

Now enjoy the prey-predator dynamic and try to modify the various parameters. (NB, In step3a the orders are sent to grass unit with `shuffleListsAtEachAssignment` *true* and `uniqueAssignmentInEachCycle` *false*. In step2 and step3a the first parameter is *true* and the second one is *false*.

NB, after the first hit to the **Start** button, we can chose the **stratumNumber** value: if 1, we are running the grass stratum alone; if 2, we are running the grass and prey strata (to much preys destroys the grass); if 3, we are running grass, preys and predators.

???. THE TUTORIAL STEP 3B: THE PREDATORS, SECOND VERSION WITH RANDOM POSITIONS IN THE GENERATION OF THE GRASS

Using **c1201** in grass generation we obtain a wider grass distribution, but mainly strong effect on lowering the frequencies of prey and predator oscillations.

The computational step 1201 (a code for the prey-predator tutorial) acts via **c1101** and creates a new random unit in the model stratum reported in position 0,1 of the first received matrix, with the probability set in position 0,0 of the same matrix and increases the position 0,2 of the second received matrix by 1, to count the created units; finally it changes the status to done;

matrix 0 is now

Probability for recipe 1 in stratum 0, grass creation	The stratum where we find the model in which units (the grass) are created	Count of the created grass
---	--	----------------------------

Despite the random grass distribution, the co-evolution of the grass, preys, predator (indirectly) forces grass to form close district.

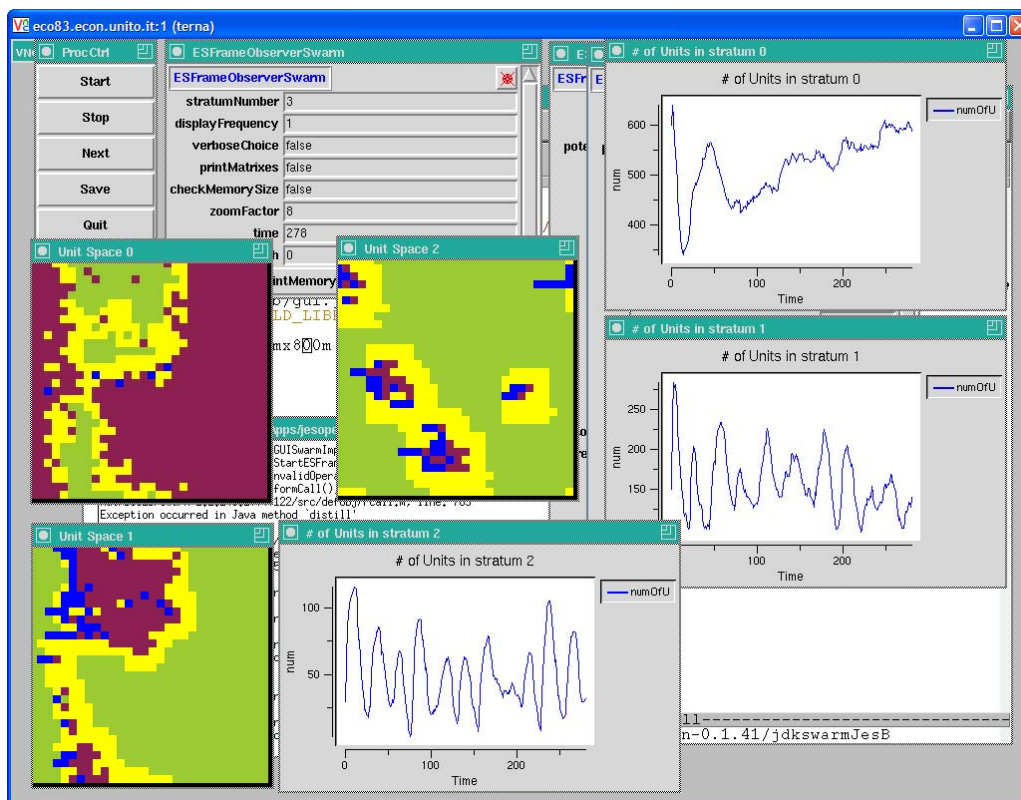


Figure ?. Step 3b, time 278, look at the complex codetermination of grass, preys and predators.

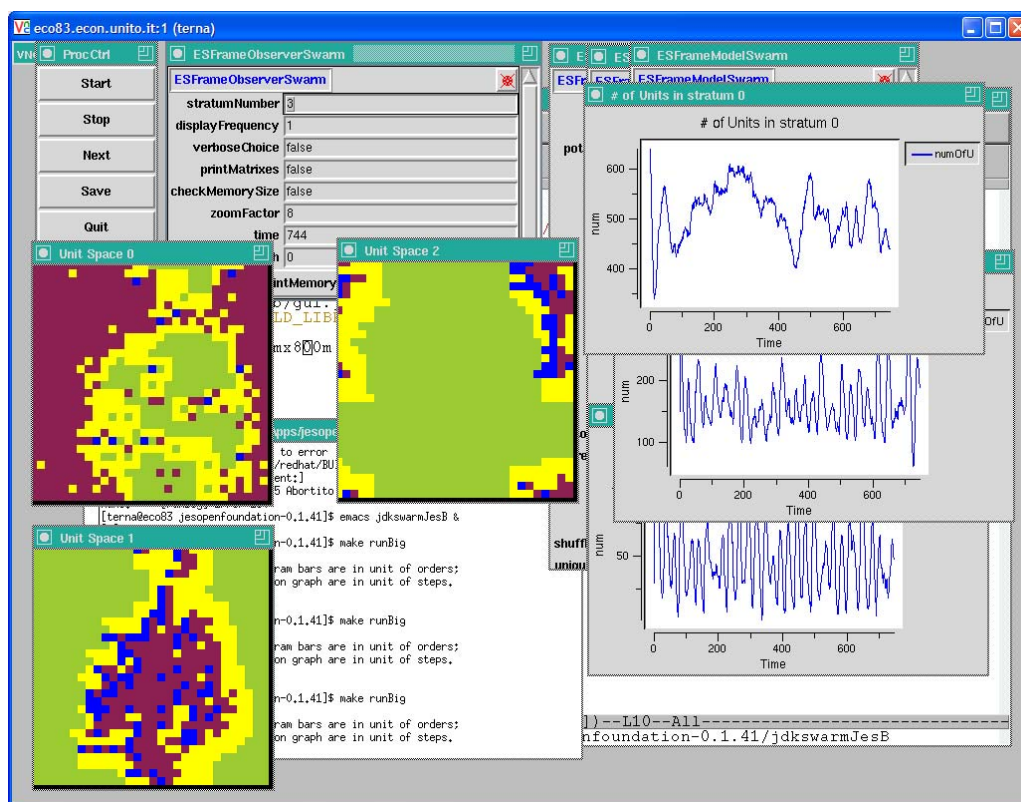


Figure ?. Step 3b, time 744, another codetermined configuration.



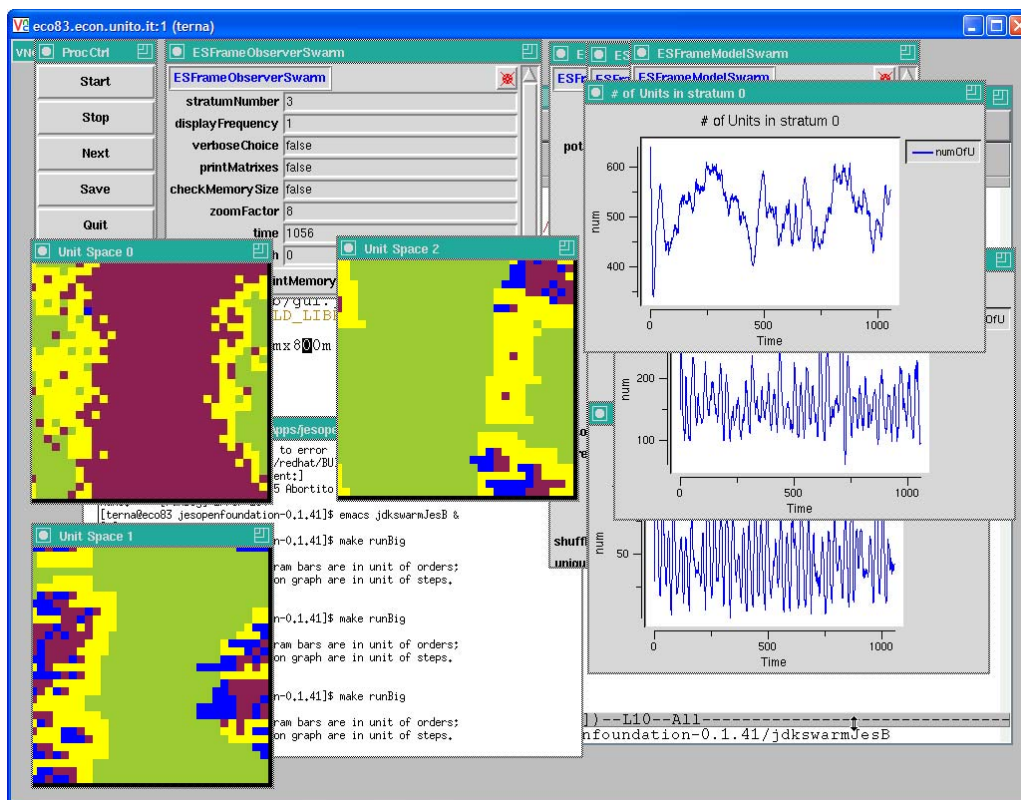


Figure ?. Step 3b, time 1056, another codetermined configuration.

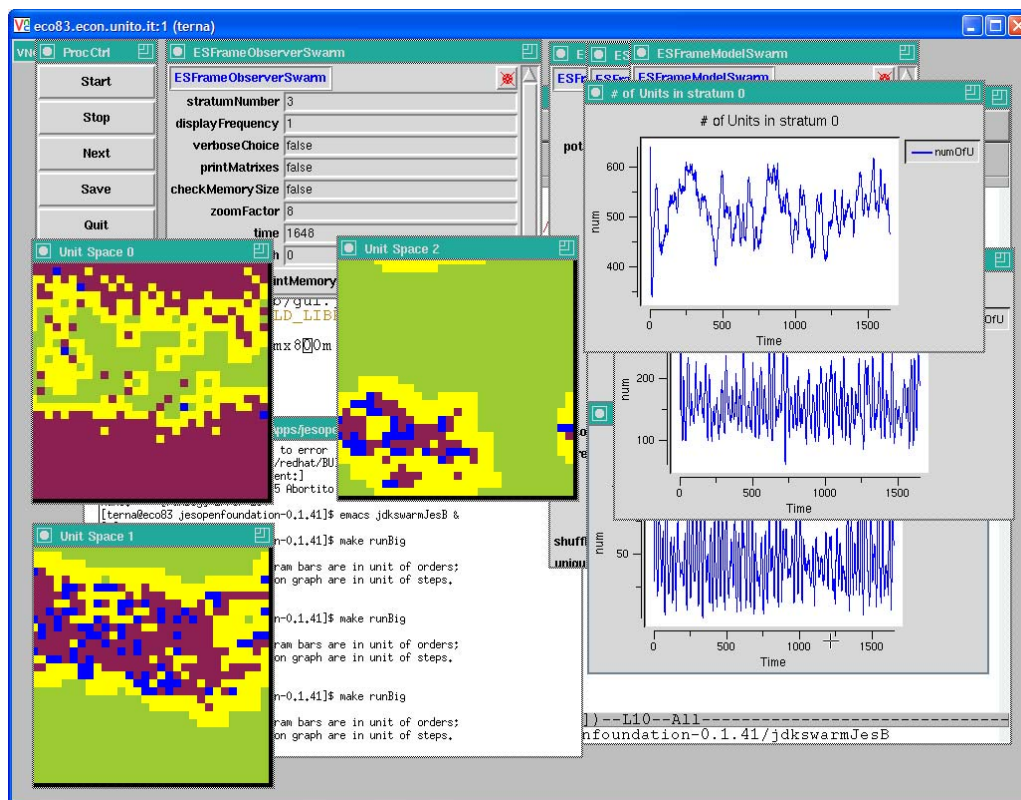


Figure ?. Step 3b, time 1648, another codetermined configuration.

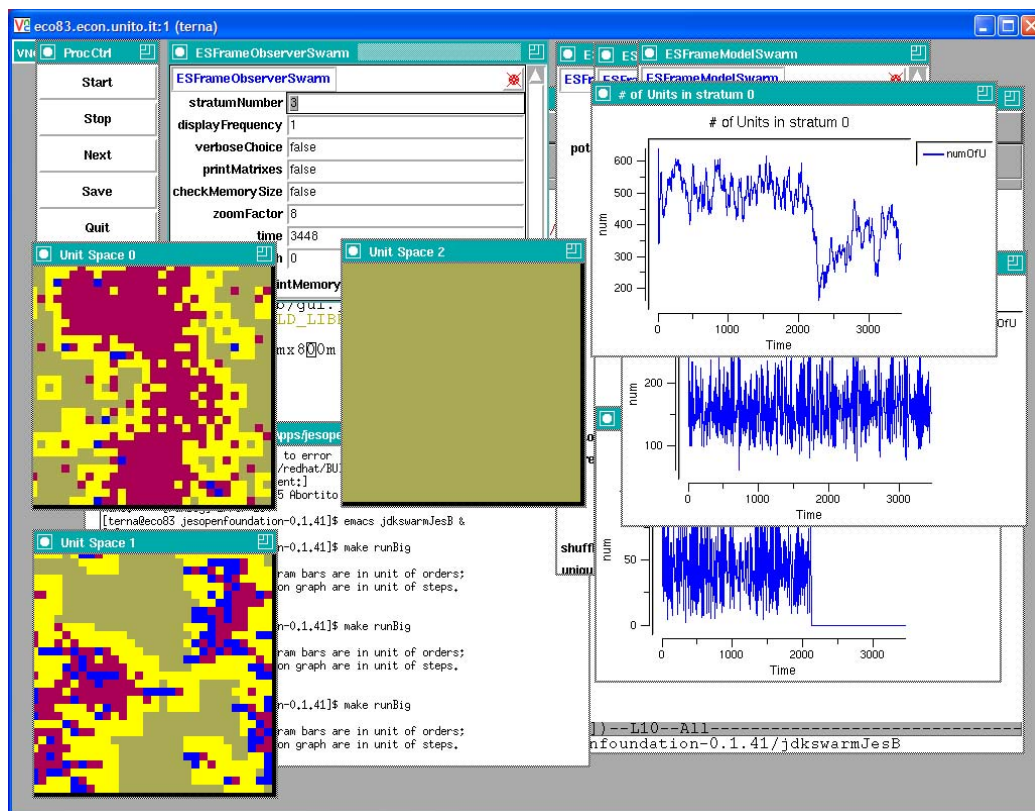


Figure ?. Step 3b, time 3448, predators disappeared at 2132, now we have a simplified two stage model.

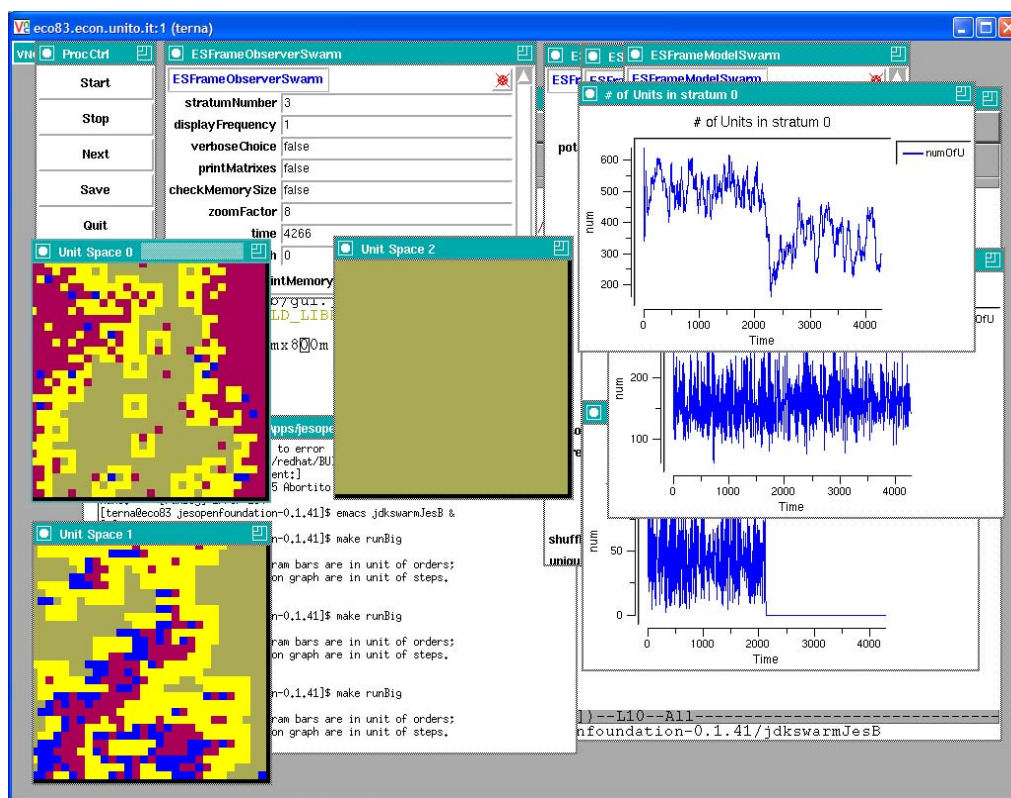


Figure ?. Step 3b, time 4266, do the grass survive?

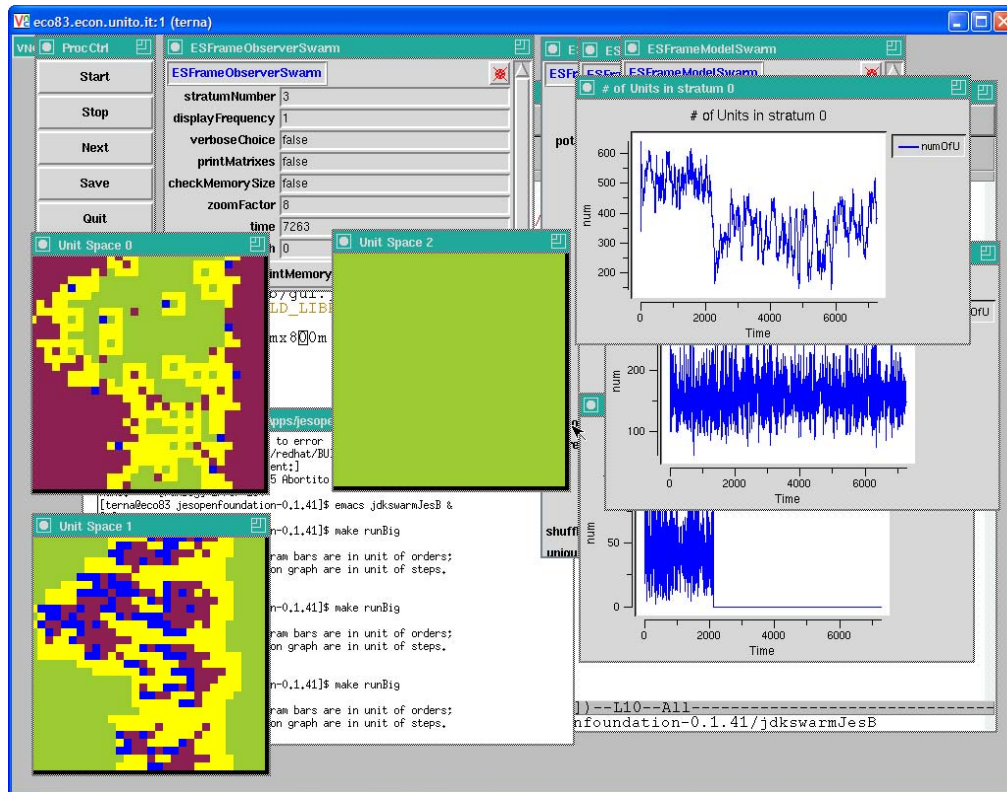


Figure ?. Step 3b, time 7263, do the grass survive?

At 10,660 it is alive!

Figure ?. Step 3b, time 7263, do the grass survive?

???. THE TUTORIAL STEP 3C: THE PREDATORS, THIRD VERSION WITH RANDOM GENERATION OF THE GRASS AND PREDATORS INCREASING IN SIZE

Using c1210 to increase predator size, via the creation of a copy of the unit in another stratum

The computational step 1210 (a code for the prey-predator tutorial) acts via c1110 and creates a new unit in the same x,y position, but in the model stratum reported in position 0,1 of the first received matrix, with the probability set in position 0,0 of the same matrix, if the value (energy) in 0,0 of unit memory matrix is > the level set in position 0,3 of the second received matrix, and increases the position 0,2 of the second received matrix by 1, to count the created units.

c1110, called by c1210, acts if a specific cell in the unit memory matrix is >0; this cell is identified via the urd, ucd trick (the same used for rd and cd, but applied to the unit memory matrix) in c1210, where also its content is calculated; the done assignment is in c1110.

Memory matrix 2 now is:

acting probability for recipe 1 in stratum 2, predators eating preys	count of preys ate (dropped)	NA	NA	NA	NA
acting probability for recipe 23 in stratum 2, time step (predatorss consuming and, if any, perishing),	count of perished predators (dropped)	perishing if energy in row,col of unit memory matrix is < the level set here	row of the memory matrix we refer to	col of the memory matrix we refer to	value to be added to the row,col position of unit memory matrix
NA	NA	NA	NA	NA	NA
acting probability for recipe 4, predators reproducing	Count of the created predators	reproducing if energy in 0,0 of unit memory matrix is $\geq$ the level set here	the stratum in which we allow the units to reproduce	NA	NA
acting probability for recipe 5 in stratum 3, predators growing	destination stratum	count of the grown predators	growing if energy in 0,0 of unit memory matrix is > the level set here	NA	NA

In our case, the cell is 0,1 in unit memory matrix and the new unit is created (i.e., the old one grows in the same position of the stratum # 2, but in stratum # 3).

To avoid the reproduction of grown units in stratum 3, we have to replace the computational step 1206, with the 1207 step, which acts only in the specified stratum (here 2, as we can read in position 0,3 of the second received matrix, with rd=3).

The computational code c1207 (a code for the prey-predator tutorial) uses c1206, but acting only if the unit the order is in, is in a specific stratum (that above, in the previous paragraph).