How to Use the *jES Open Foundation* Program[ϒ]

Pietro Terna

(August 2004)

Dipartimento di Scienze economiche e finanziarie G.Prato, Università di Torino, Italia
pietro.terna@unito.it

?. RULES AND USE

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

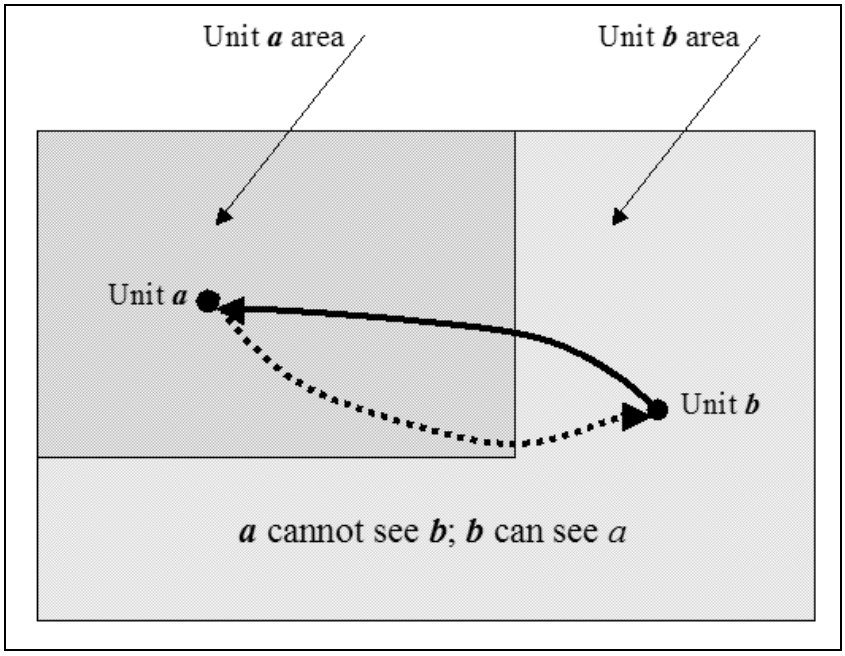

Figure 1. xxx

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 layer 1, 2, …), if existing, have to copy all information about general matrixes from the first model (that in layer 0). The first row of the file must contain exactly the following text sequence "number(from_0_ordered)_rows_cols".

[ϒ] related to jesopenfoundation-0.0.21

?. 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);
- firstEvolApplication: the jESevol app used for SwarmFest 2004, running also in jESOF; use the x.y launching files.

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[2] the ComputationalAssembler.java file (which inherits its default methods from the class ComputationalAssemblerBasic). Computational capabilities are aimed to forecasting, evaluations, auctions to choose procurements, …

[2] 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);
```

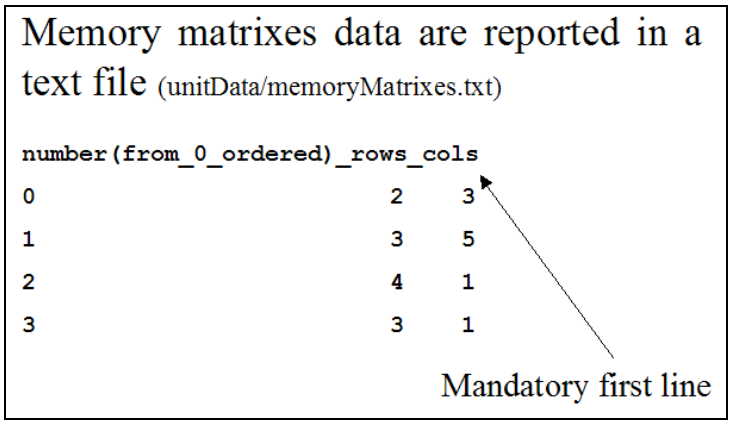

Figure 18?. An example of memory matrixes declarations: the ID numbers are ordered and start from zero.

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 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[3].

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 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_1 … m_n' where **c** is mandatory, *code* is the code of the computation, *n* is the number of matrixes to be used and 'm_1 … m_n' are the ID numbers of those matrixes, as reported in the file unitData/memoryMatrixes.txt (Figure 18).

[3] -0, as a number, is equal to 0, so the first matrix cannot be declared unsensitive to layers.

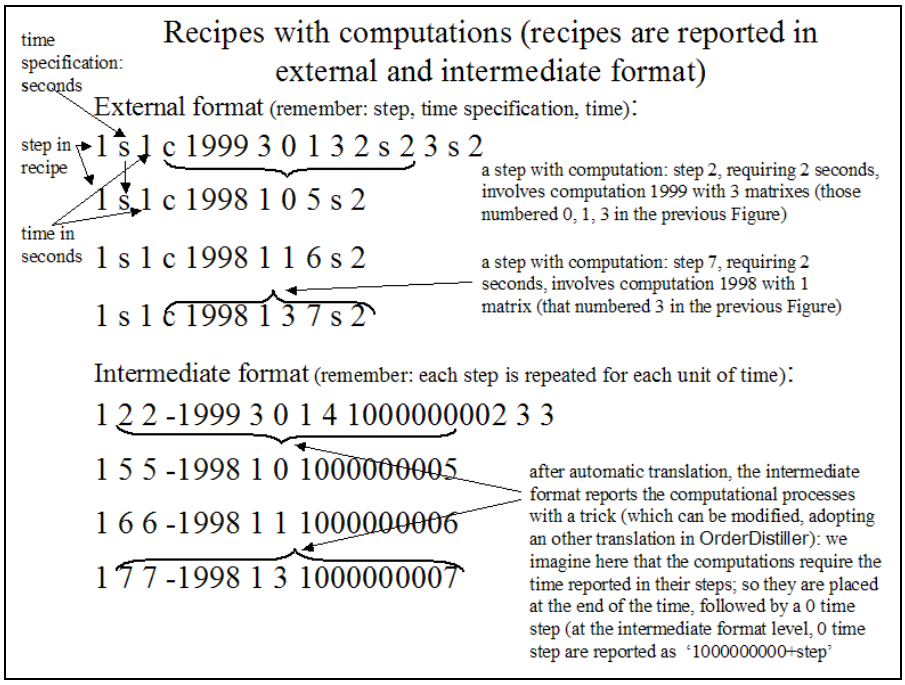

Figure 19?. The format of the computational processes.

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 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[4].

[4] 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*.

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[5]. 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 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*).

[5] See previous note.

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 recipeData/recipesFromRecipes.txt, written in free format. Two restrictions: first, a maximun of 10 recipe codes to be launched is allowed (but can be modified in **ComputationalAssemblerBasic.java**); second, a maximun of 1.000 launches is allowed per tick (but can be modified in **OrderDistiller.java**).