v. 0.4.3.3

How to Use the Java Enterprise Simulator (*jES*) Program[ϒ]

Pietro Terna

(August 2004)

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

1. THE DESCRIPTION OF A WORLD WITH "TWO SIDES" AND OF A SIMULATION ENVIRONMENT CONSISTENT WITH THAT KIND OF WORLD

We are developing *jES* (the synthetic name of the Java Enterprise Simulator project), or *jesframe* (the internal name of the project: a frame used to develop enterprise simulation models based on the Java version of Swarm) both (i) to simulate the activities - and the consistent emerging results - of an actual enterprise and (ii) to build virtual[2] or hypothetical enterprises. In the first case we can use the simulator to test the behavior of an emulated enterprise, both as it is and wisely modified, with highly practical goals. In the second case, we are interested in theoretical analysis of enterprise creation, behavior, network interaction.

In any of the two cases, we are building a *model*, of an actual or virtual enterprise, but always a model. Following Gilbert and Terna (2000, p.58), we can say that:

> (. . .) there are three different "symbol systems" available to social scientists: the familiar verbal argumentation and mathematics, but also a third way, computer simulation. Computer simulation, or computational modeling, involves representing a model as a computer program. Computer programs can be used to model either quantitative theories or qualitative ones. They are particularly good at modeling processes and although non-linear relationships can generate some methodological problems, there is no difficulty in representing them within a computer program.

The first approach to how to use *jES* introduces the existence of two independent sides in our description and representation of the enterprise world and, in a consistent way, in our program or, better, in our model.

Our simulated enterprise has both orders to accomplish and production units that perform the different steps of the production process. The orders are described by recipes that contain the "What to Do" (WD) side of the world; the production units represent the "which is Doing What" (DW) side of the same world.

A third formalism is related to the time sequence of the events (the orders to be executed) that occur in the world we are simulating; this is the WDW formalism: When Doing What (see below 5.2.).

Production units can be within the firm or outside. In the second case: (i) constituting other enterprises or (ii) standing alone as small business actors.

[ϒ] This technical appendix is related to jesframe-0.9.9.10.tar.gz; the figures of this *How to* are also reported in the companion file How_to_use_jES_(figures).ppt (you can read it also with OpenOffice, www.openoffice.org).

[2] The term virtual is used here to designate an enterprise that does not exists, useful as a stylised item to elaborate ideas about firm creation, cooperation etc. The term "virtual enterprise" is also used to designate a network of actual firms or of subparts of those firms (see below the reference to the NIIIP Consortium) and it is still compatible with the use and purposes of *jES*, but in would be relevant to (i) instead of (ii).

1.1. A DICTIONARY

It is useful to introduce here a dictionary of our terms:

- a *production unit* is a productive structure within or outside our enterprise; a production unit is able to perform one or more of the steps required to accomplish an order;
- an *order* is the object representing a good to be produced; an order contains technical information (the recipe describing the production steps) and accounting data;
- a *recipe* is a sequence of steps to be executed to produce a good.

The core of the model is the clear separation between the orders and the production units: WD and DW are completely independent, in formalism and in code. So, running the model, we check the consistency of the two sides, as in the actual world, where the output of an enterprise arises from a complex interaction among products and production tools. As we will see below, recipes can also describe internal parallel production paths, computational steps, batch activities and assembly phases, where the typical procurement problems of a supply chain can be reproduced and tested.

1.2. A SIMPLIFIED VIEW

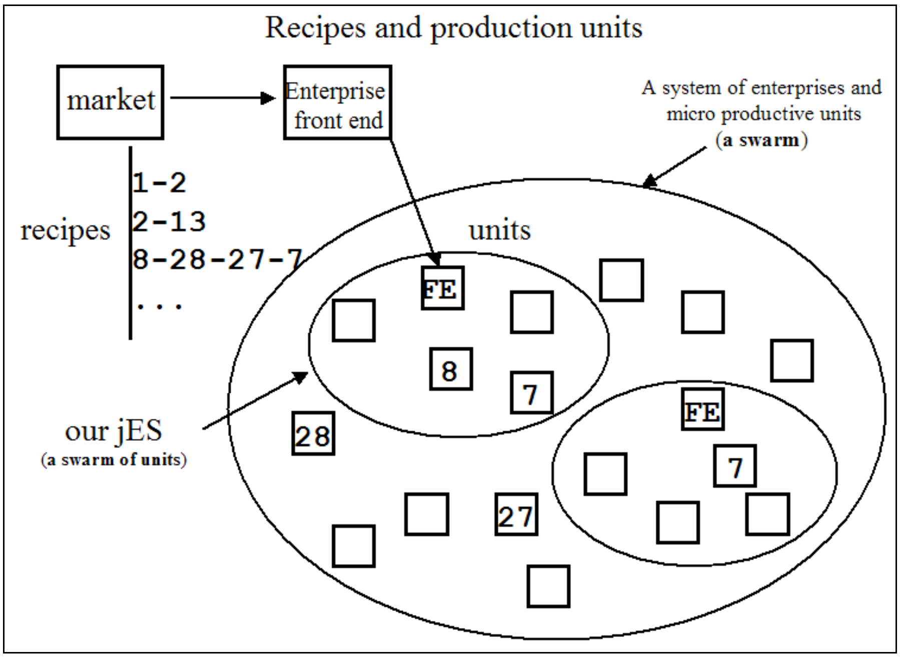


Figure 1. A simplified view of the *jES* components; recipes are here reported in a simplified way, without time specifications.

A first view is that of Figure 1. This is an introductory view of the world, with the recipes written in a simplified way; i.e., as a sequence of steps to be executed without information about the time required by each step. Observing the recipe 8-28-27-7 we can see that the front end (FE) of an enterprise can take in charge the first step, which will be executed by unit 8 (in this simplified version, production unit and step numbers coincide) within the enterprise.

Figure 2 introduces a more dynamic interpretation of the world we are describing. We have here three simple phases (*a*, *b*, *c*) in which the order containing the recipe 8-28-27-7 goes from one production unit to another; in this sequence all the needed information is contained in the order: when the activity of a production unit (as an example, unit 8) is concluded, the production unit asks to the order what is the next step to be performed and then asks to all the production units to reply if they are able to execute that task. In this way, the order makes its journey from unit 8 to unit 28 (which is outside the enterprise and can be considered as a simple business unit) and to unit 27 (similar to 28). In the next step, signed with an *X* in Figure 2, we have a choice problem, having two production units able to perform task 7. Below (3.5.) we will introduce a set of criterions that allow the production units to deal properly with this kind of problem.

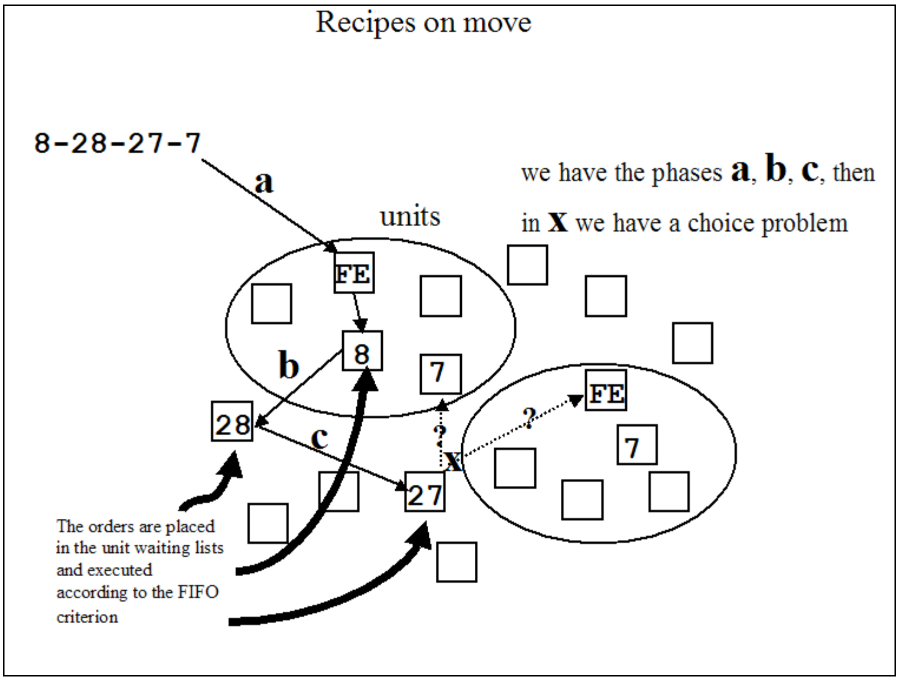

Recipes on move
8-28-27-7
a
units
we have the phases a, b, c, then
in X we have a choice problem
FE
8
7
b
28
c
?
?
X
27
FE
7
The orders are placed in the unit waiting lists and executed according to the FIFO criterion

Figure 2. A dynamic view of the *jES* components; recipes are here reported in a simplified way, without time specifications.

A remark, a little bit more abstract. One of the two units, that able to perform step 7, belongs to another enterprise, so we can imagine we need to open a dialog with the front end of the second enterprise. Anyway we have also to take into consideration the possibility of a direct link with the production unit within the other enterprise. The idea of linking together the subunits of more complex enterprises to create temporary production organizations, brings us directly to the concept of virtual enterprise as an organizational tool: for an important analysis of this concept, look at the NIIIP project[3] (National Industrial Information Infrastructure Protocols), at http://niiip01b.npo.org.

---

[3] In the project web site we can read that: “The NIIIP Consortium consists of a group of leading United States information technology suppliers, industrial manufacturing end users, academic, and standards organizations with a common interest in developing an information infrastructure architecture to enable organizations to operate as "Virtual Enterprises". Virtual Enterprises are teams, consortia or alliances of companies formed to exploit business opportunities that can not be addressed by a single organization.”
“The NIIIP Consortium is national in scope and its members bring a wealth of experience and technology to support Virtual Enterprises. Together with the Federal Government, they share costs and skills to create the

1.3 HOW THE MODEL WORKS

With the *[KF]* sign I will mark below the key features constituting the core of *jES*; these key features are reproduced also in the *jESlet* versions of the program (*jES* light experimental tool), developed using JavaSwarm, JAS, Ascape, RePast and, in a different way, in StarLogo, both for comparative reasons and to help scholars of agent based simulation techniques to introduce themselves to these different instruments, mainly in the social science perspective (see the *jESlet* doc for the explanation of the principal classes, also with a light UML presentation).

From a technical point of view it is important to note that almost all the intelligence of our simulation process *[KF]* is placed into the order (WD) side. We can describe the behavior of the code in the following way (suppose that we are not at the beginning of the simulation, so the process is already running to elaborate orders):

1. production units[4] act operating on the orders existing in their waiting lists, if any, one order per each tick of the simulation clock *[KF]*;
    - if the production unit is idle, having no orders in its waiting list, the unit can produce inventories, in a stand alone way; the use of the “stand alone” expression has the goal of differentiating this kind of production of inventories from the explicit launch of orders related to the supplying of parts or components for the production process, in a supply chain way; the stand alone inventories exist if it is technically possible – due to the nature of the activity of a specific production unit - to produce and to store them; the stand alone inventories anyway are produced only if the simulation parameter[5] useWarehouses is set to *true*;
    - the orders whose step is done are placed in a “made production” list, to be successively diffused to other production units *[KF]*;
    - an order recipe can include a code related to an endUnit (see below 3.3.);
        - in this case, the order regards a component part produced by ourselves or procured externally (the recipe contained in the order can report internal or external steps to be accomplished; in the external case, may be without details);
            - after its production (or its procurement), this kind of order is kept in an actual or virtual warehouse represented by an endUnit;
        - note the difference between this kind of production of components or parts and the stand alone preparation of inventories of each specific production unit;
    - if an operation requires more than one tick of the clock to be concluded, the order is kept into the production unit until all the time is spent, with a direct and immediate reassignment of the order to the unit itself;
    - the production can require a setup process, with related cost and time spent; see below 3.2. for this feature, that is not yet implemented;

---

necessary infrastructure to support Virtual Enterprises across the United States. The NIIIP Consortium has entered into a series of cooperative agreements with the Federal Government and associated agencies to develop, demonstrate, and prototype industrial «Virtual Enterprises».”

[4] ModelActrions2 in ESFrameModelSwarm.java and unitStep1 in Unit.java.

[5] Parameters are set either via the probe of the model or into the jesframe.scm file.

  - the production can be replaced by the use of inventories related to each specific production step,
    - obviously, if that stand alone inventories exist;
    - in this case, more than one order can be treated in a single tick, if we have room in inventories;

2. new orders[6] (each one containing its recipe) are launched in production *[KF]*:
   - each order contains a recipe *[KF]* built as sequence of steps to be done,
   - with several complex tools useful to better describe the step sequences and the related consequences;
   - new orders enter into the simulation:
     - following a script describing, with the WDW formalism, the temporary sequence of the events to be simulated (see below, in 5.1., the use of the orderDistiller object; see also 5.2. for the WDW formalism);
     - while testing the program *[KF]* or for a theoretical use, orders can be randomly generated via the orderGenerator object; this is also the way used in the light version of the program (*jESlet*);
   - the new orders are assigned to the production units *[KF]* in the way described at point 3;
3. each order[7] kept into the made production lists (point 1 above) of the production units, or just launched (point 2 above) makes a search *[KF]* - using the assigning tool code - into the world to discover if one or more production units can perform its first (undone) step;
   - assignments:
     - if (only) one production unit makes a positive reply the order is assigned to the waiting list of that production unit *[KF]*;
       - if at least one replying unit does not exist, the program is stopped in an error condition (we have to correct the description of our world);
     - if more than one production unit is able to perform the required step, we have to choose one of the candidate units;
       - the choice can be made following several criterions (see below 3.5, unitCriterion),
       - in the future this will be a key feature of *jES* (not yet implemented),
         - allowing human interventions: to experiment different situations and solutions,
         - but also to train people
         - and to discover how people decide;

---

[6] modelActrions2distiller in ESFrameModelSwarm.java and distill in OrderDistiller; or, as an alternative, modelActrions2generator in ESFrameModelSwarm.java and createRandomOrderWithNSteps in OrderGenerator.

[7] modelActrions2b in ESFrameModelSwarm.java and unitStep2 in Unit.java.

    - finally, this is a window open toward the introduction of sophisticated optimization tools such as genetic algorithms and classifier systems;
  - orders are kept in the waiting list of the chosen production unit until their specific step is done, according to a FIFO (First In First Out) criterion *[KF]*;
    - the sequence of the order in the waiting list can be managed to improve the firm performance (this feature is not yet implemented);
    - orders can be executed according to a LIFO (Last In First Out) criterion in a specific technical circumstance (sameStepLifoAssignment, see below 3.9.1.), besides the case of operations (steps) requiring more than one tick of the clock to be concluded (introduced above, point 1);
  - an order is dropped, with some accounting, after the last step of its recipe is done *[KF]*;
    - to drop an order has the meaning of eliminating it from the simulation;
    - in other terms, the related item is sold (the recipe steps can include trade actions);

4. if the simulation parameter useNewses is set to *true*, each production unit[8] propagates news about order it will send out to other production units in the next future; this is an attempt to simulate cooperation and information within an organization; the decisions about the production of the stand alone inventories, described at point 1, can be based also on news informing each unit about its future production;
5. the sequence continuously goes back to the phase described at point 1 for the next tick of the clock (other steps are devoted to initializing and accounting operations, but are not reported here, to simplify this presentation).

Time synchronization and parallelism are obtained via a usual trick in simulation: at each tick of the simulation clock, all the production units make the actions described at point 1 above independently (the actual time sequence does not matter); then, always in the same clock tick, the program executes point 2; when all these actions are concluded, orders make the operations described at point 3, again independently and always in the same tick; finally, production units execute point 4, always independently and in the same tick.

2. A CLOSER LOOK TO THE WD SIDE

2.1. ORDERS, RECIPES AND LAYERS

Our simulated enterprise has orders to accomplish; the orders are described by the recipes that contain the WD (What to Do) side or the world. The basic recipe in an order is structured as shown in Figure 3.

Here we have a sequence of steps followed by a time specification and by a time quantity: step $n1$ requires $m1$ units of time, that can be days, hours or seconds, according to the time specification (*ts*) choice. Time quantities are integer numbers.

---

[8] modelActrions2b in ESFrameModelSwarm.java and unitStep3 in Unit.java.

Internally, recipes are represented repeating the same step for each unit of time it lasts: e.g. 10 s 3 is 10 10 10. To go more in deep, if we have heterogeneous units of time in the same simulation (hours and seconds, as an example), internally an hour is represented by a sequence of 3600 steps of one second (See below 5.1 for the discussion about units of time; all this matter is not yet fully implemented).

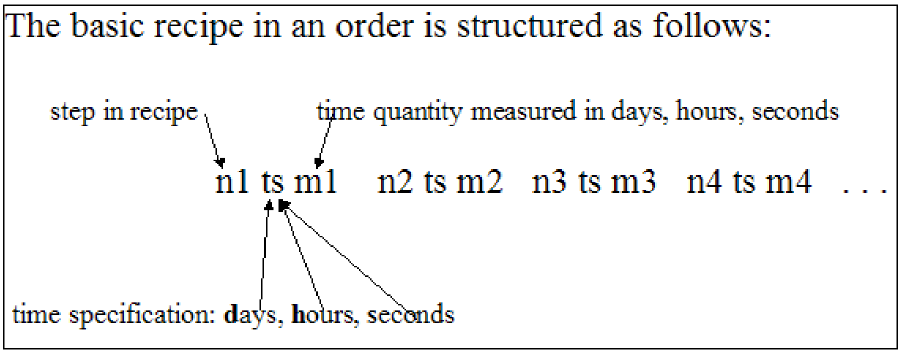

The basic recipe in an order is structured as follows:
step in recipe
time quantity measured in days, hours, seconds
n1 ts m1 n2 ts m2 n3 ts m3 n4 ts m4 . . .
time specification: days, hours, seconds

Figure 3. Basic recipe.

Two orders containing the same recipe can be different from some qualitative point of view. To deal with this product specification, we introduce the concept of a layer: it is a period of time or a set of qualitative conditions that introduce differentiations amid the orders (e.g., two collections in fashion production, with the same technical description into the recipes and different qualitative results).

The number of layers that we can use is explicitly introduced via the totalLayerNumber parameter of the simulation: we use totalLayerNumber - 1 layers; if the limit is set to 1, we use no layers. The attribution of each order to a layer is made by the user while writing the order sequence of the simulation (see below 5.1. about the use of the orderDistiller object; while testing the program, using the orderGenerator object, the attributions to layers are made randomly).

We can also imagine the definition of a special step (with its length) in recipes in which nothing is happening (only the time is elapsing), to be used when a product has to wait a due time (we are simply making it older for some reason) before being sold or used again in production. The unit able to perform this step has unlimited (or limited) capacity of dealing with the waiting list dimension, because doing a step means doing nothing (but keeping together the object we are dealing with can require space).

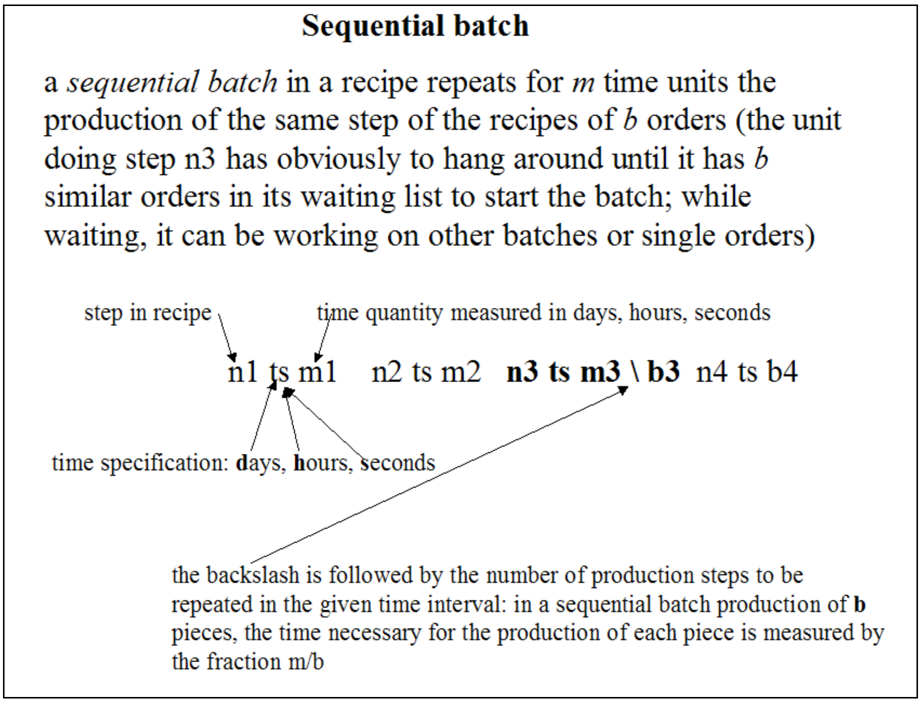
2.2. BATCHES

We have cases in which it is not realistic to think about processes concerning separately a single piece: a realistic view is that of considering the production as made of batches of pieces.

We have two kinds of batches in our simulated world: sequential batches and stand alone batches.

2.2.1. SEQUENTIAL BATCH PROCESS

A sequential batch process – as reported in Figure 4 – deals simultaneously with multiple orders, despite being one of the steps of a recipe. We have to imagine a productive process that is separately managed for each order, but that for certain steps requires an activity referred to a group of orders to be processed together: this is a sequential batch, formally expressed as in Figure 4.


Sequential batch

a *sequential batch* in a recipe repeats for *m* time units the production of the same step of the recipes of *b* orders (the unit doing step n3 has obviously to hang around until it has *b* similar orders in its waiting list to start the batch; while waiting, it can be working on other batches or single orders)

step in recipe
time quantity measured in days, hours, seconds
n1 ts m1   n2 ts m2   **n3 ts m3** \ **b3**   n4 ts b4
time specification: **d**ays, **h**ours, **s**econds

the backslash is followed by the number of production steps to be repeated in the given time interval: in a sequential batch production of **b** pieces, the time necessary for the production of each piece is measured by the fraction m/b

Figure 4. A sequential batch.

Comparing orders to prepare a batch we must decide whether to consider equal orders with equal steps, also if the production units that have made the concluded steps are not the same. If the compareDisregardingUnits parameter is set to *true*, orders are used to compose a batch also in the case that they have not been produced exactly by the same units; the opposite if the parameter is set to *false*.

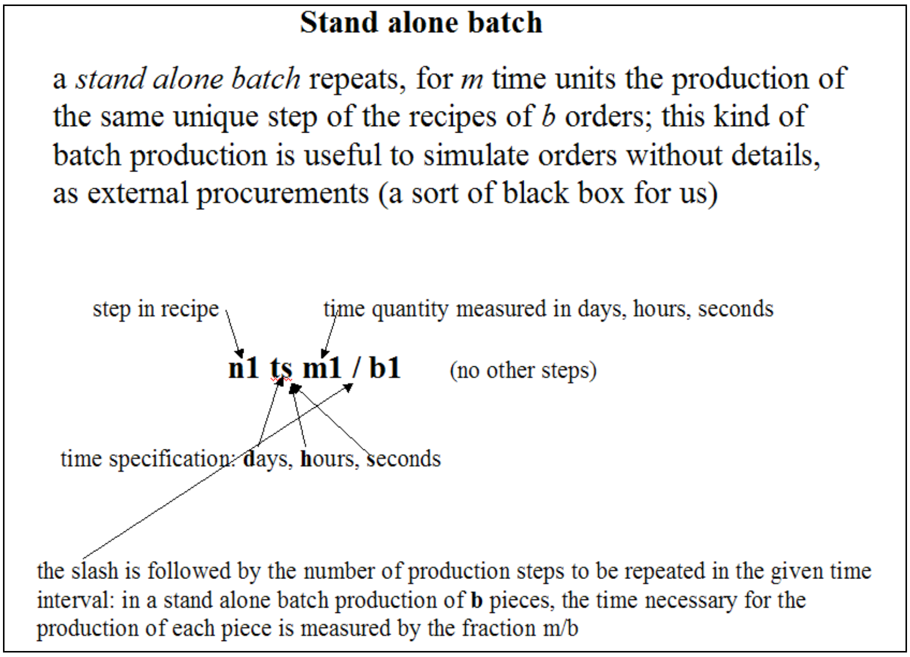
How the sequential batch works: SequentialBatchAssembler identifies equal orders necessary to compose a sequential batch, marks them as properly belonging to a sequential batch and finally places them together at the beginning of the proper unit waiting list. The production of the batch will require the global time of the sequence (*m3* in Figure 4) and all the produced items will be available simultaneously.

2.2.2. STAND ALONE BATCH PROCESS

A stand alone batch process, described in Figure 5, is similar to the sequential one (formally we use here "/" instead of "\"), but it is not included in a recipe with other steps.

It is the only step of a recipe describing a process considered as a whole: imagine an external procurement that our enterprise is ordering in batches of large dimensions, requiring a time to be accomplished. In the just in time perspective, the determination of the time point in which to start a stand alone batch order is very important, considering the delay necessary to produce the whole bunch.

The recipe containing the stand alone batch process must be composed by two parts: the stand alone batch, obviously, and the identifier of an endUnit (see below 3.4. and 3.8. for the double procurement process description and 3.3. for the end units explanation).



Stand alone batch

a *stand alone batch* repeats, for *m* time units the production of the same unique step of the recipes of *b* orders; this kind of batch production is useful to simulate orders without details, as external procurements (a sort of black box for us)

step in recipe
time quantity measured in days, hours, seconds
n1 ts m1 / b1
(no other steps)
time specification: days, hours, seconds
the slash is followed by the number of production steps to be repeated in the given time interval: in a stand alone batch production of b pieces, the time necessary for the production of each piece is measured by the fraction m/b

Figure 5. A stand alone batch.

2.3. PROCUREMENTS IN THE WD SIDE

Procurements are key elements in running an enterprise simulation. In Figure 6 we represent a situation in which step 28, to be executed by the unit signed 28 (in this simplified presentation, recipes are written as a sequence of steps to be executed, without information upon the time required, and production unit ID numbers coincide with step numbers) requires to add some components, internally produced or externally procured, to the output received from production unit 8, as a semifinished product; here we need components identified by codes 121, 34 and 73.

Specific recipes, like those of Figure 7, must prepare these components. Note that an e identifier followed by a numeric code concludes all those recipes. With the sequence "e number" we recall an endUnit (see below 3.3.; an actual or virtual warehouse where the internally produced parts are to be searched when a recipe asks to a unit to procure them).

In Figure 7 we see procurements, coming from external suppliers, treated as "black boxes", either with a single step recipe (that is concluded by the *c1* endUnit code) or by stand alone batches (see above 2.2.2.) recipes (those concluded by *c2* and *c3* endUnit codes); we could decide also to explode our representation of external activities, using a magnifying lens and describing them with the same detail used for the internal one. In the same Figure we have also internally produced parts: the first one (concluded with the *c4* endUnit code) is described in a detailed way and contains a sequential batch process (see above 2.2.1.); those concluded with the endUnit codes *c5* and *c6* are similar to those used to describe without details the externally procured productions.

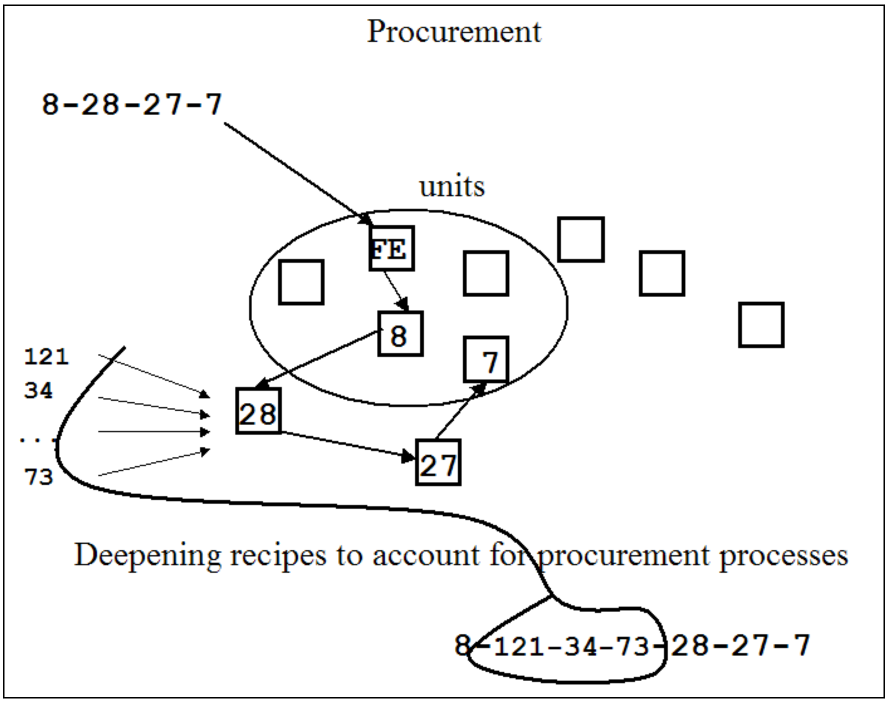
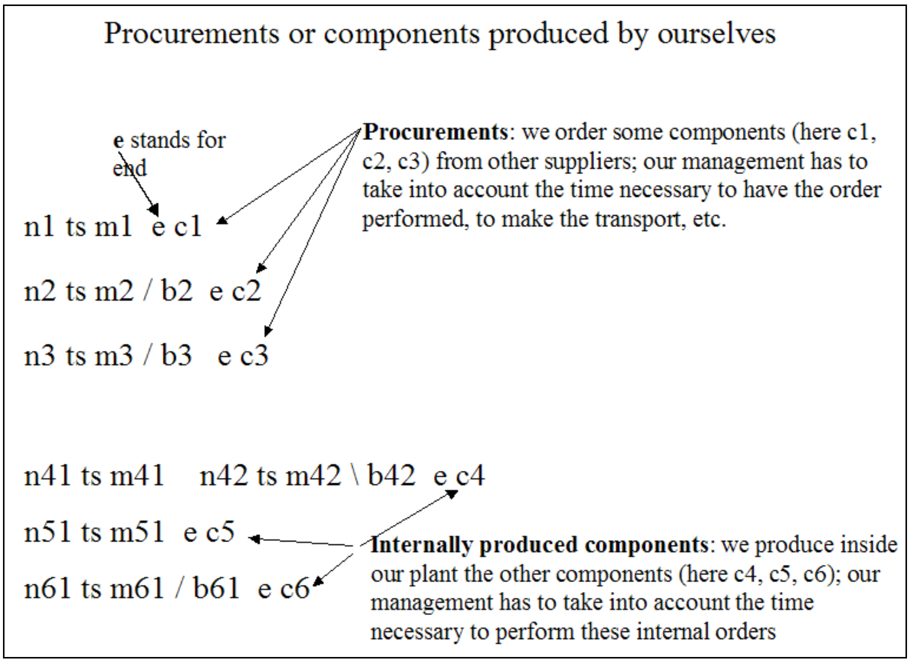


Figure 6. A graphical representation ... ; recipes are here reported in a simplified way, without time specifications.

The format of the recipes using procured components is introduced below in 3.4., when we will examine procurements from the DW perspective.



Procurements: we order some components (here c1, c2, c3) from other suppliers; our management has to take into account the time necessary to have the order performed, to make the transport, etc.

Internally produced components: we produce inside our plant the other components (here c4, c5, c6); our management has to take into account the time necessary to perform these internal orders

Figure 7. Components (to be procured or internally produced) described in recipes.

Instead of the generic codes *c1*, *c2*, … , *c6* we can imagine to have here the codes 121, 34 and 73 of Figure 6.

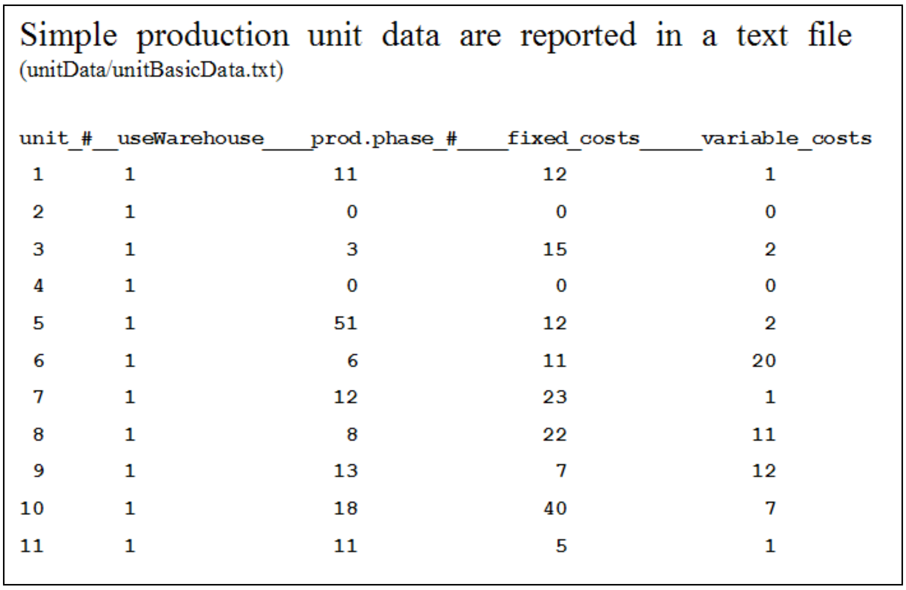
The difference between procurement and internal produced components is anyway merely an *ex post* classification and it is related to our knowledge and decisions about the actual organization of an enterprise: e.g., our firm is able to make activities required by the recipes of the externally produced components and we decide to make them internally.

3. A CLOSER LOOK TO THE DW SIDE

3.1. SIMPLE PRODUCTION UNITS

The DW (which is Doing What) side of the same world is related to the production units and to the “end units”.

A production unit is the elementary production group able to accomplish one or more kind of step of a recipe; steps in recipes are identified by numbers, as we have seen; also the production units report the steps that they are able to accomplish communicating a number.



Simple production unit data are reported in a text file

(unitData/unitBasicData.txt)

```
unit_#__useWarehouse____prod.phase_#____fixed_costs_____variable_costs
 1      1                 11               12               1
 2      1                  0                0               0
 3      1                  3               15               2
 4      1                  0                0               0
 5      1                 51               12               2
 6      1                  6               11              20
 7      1                 12               23               1
 8      1                  8               22              11
 9      1                 13                7              12
10      1                 18               40               7
11      1                 11                5               1
```

Figure 8. Simple production units, with: number; the flag about using or not stand alone warehouses (see below 3.8.); their production phase, fixed and variable costs.

Simple production units, which are able to deal only with one kind of step, are easily described using the unitData/unitBasicData.txt file, which contains the information of Figure 8.

The first line is mandatory, written exactly as is, to force the user to pay attention to the content of the file. Then we have lines reporting: (i) the numbers[9] of the production unit (the

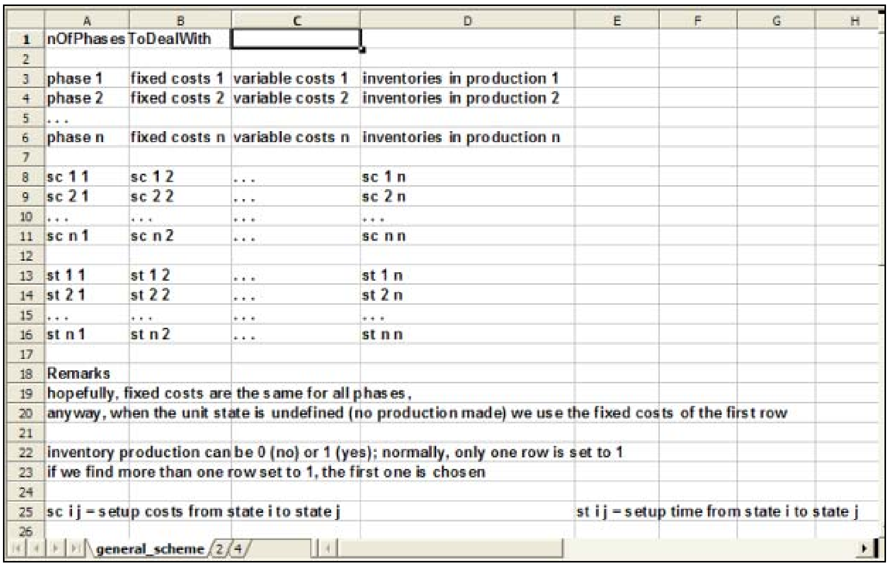
---

[9] Normally, production units are sensitive to layers (two orders with the same recipe are different if belonging to different layers); if the number of the unit is negative in unitData/unitBasicData.txt, that production unit is considered unsensitive to layers. This is useful to avoid the use of layer differentiation to establish if an order belongs to a sequential batch (see below 2.2.1.).

lines can be introduced in any order, i.e., they have not to be sorted by production unit number); (ii) a flag set to 1 if the production unit can produce and use the stand alone inventories (see below 3.8. Warehouses; this flag does not work in rows containing a complex production unit); (iii) the specific step that the production unit is able to do (several production units can be able to perform the same step); (iv) fixed costs for each time unit[10] (seconds, hours, days); (v) variable costs for each time unit (seconds, hours, days).

The unit of time must be the smallest used in the whole recipe set. If we use the internal orderGenerator (see below, 5.1.) – when we are testing the code – all the recipes are internally generated using the same time basic interval (that we can assume to be a second, an hour, a day, …): we have to use consistently that time interval in the table of the file unitData/unitBasicData.txt to measure fixed and variable costs. If we use the orderDistiller (see below 5.1.) - to follow a known order sequence applied to a recipe repertoire – may be we have to deal with recipes using heterogeneous time intervals: orderDistiller has to convert internally all the time measures to the smallest one (not yet implemented): again, that time measure has to be consistently used in the table of the file unitData/unitBasicData.txt to measure fixed and variable costs.

3.2. COMPLEX PRODUCTION UNITS



	A	B	C	D	E	F	G	H
1	nOfPhasesToDealWith							
2								
3	phase 1	fixed costs 1	variable costs 1	inventories in production 1				
4	phase 2	fixed costs 2	variable costs 2	inventories in production 2				
5	. . .							
6	phase n	fixed costs n	variable costs n	inventories in production n				
7								
8	sc 1 1	sc 1 2	. . .	sc 1 n				
9	sc 2 1	sc 2 2	. . .	sc 2 n				
10	. . .	. . .	. . .	. . .				
11	sc n 1	sc n 2	. . .	sc n n				
12								
13	st 1 1	st 1 2	. . .	st 1 n				
14	st 2 1	st 2 2	. . .	st 2 n				
15	. . .	. . .	. . .	. . .				
16	st n 1	st n 2	. . .	st n n				
17								
18	Remarks							
19	hopefully, fixed costs are the same for all phases,							
20	anyway, when the unit state is undefined (no production made) we use the fixed costs of the first row							
21								
22	inventory production can be 0 (no) or 1 (yes); normally, only one row is set to 1							
23	if we find more than one row set to 1, the first one is chosen							
24								
25	sc i j = setup costs from state i to state j				st i j = setup time from state i to state j			
26								

Figure 9. Complex production units: the explanatory table, in the worksheet labeled general_scheme.

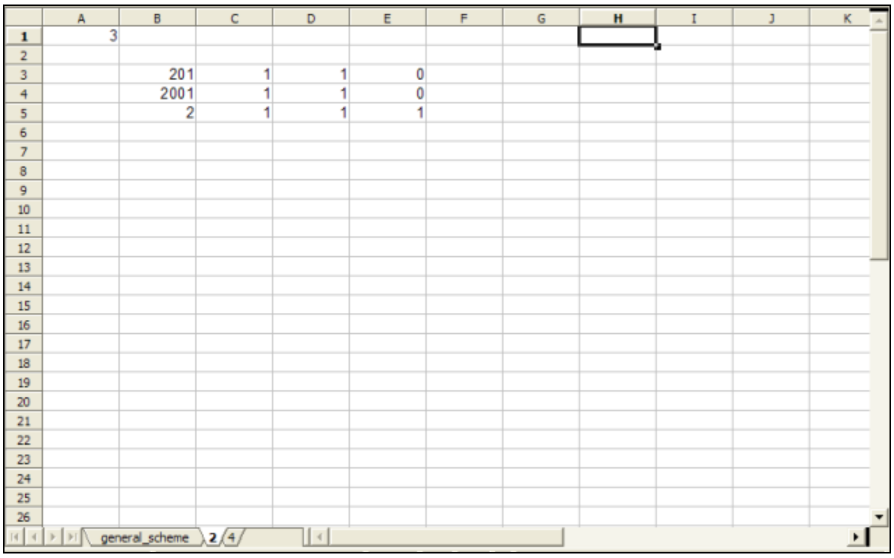
Complex production units are able to deal with a number of production steps; this kind of production unit is identified in the file unitData/unitBasicData.txt (Figure 8) with a 0 in the production phase column. We describe complex production units using a spreadsheet file[11], whose contents are easily identified via the first worksheet (labeled general_scheme) of the

---

[10] In the recipes we will have to use the time in a consistent way to the kind of time unit to which the costs are referred (see below 5.1.).

[11] We can produce the spreadsheet either using a proprietary code or employing an Open Source one, such as OpenOffice (www.openoffice.org).

spreadsheet itself (file unitData/units.xls). The information contained[12] is that of Figure 9: (i) the number of phases (kinds of production steps) the unit is able to perform; (ii) the code of each production phase followed by the data about fixed costs and variable costs per unit of time[13]; we use those costs in accounting when the production unit is executing the specific phase. Hopefully, fixed costs are the same for all phases; anyway, when the unit activity is undefined (i.e., it did not produce anything) we impute the fixed costs from the first row; (iii) finally, we have a flag in each row that, when set to 1, says that in case of absence of activity the complex production unit will operate a stand alone inventory production step of the type referred by the row (according to InventoryRuleMaster.java rules; see below 3.8. for the interpretation of the stand alone inventory production); 0 says that the row is not considered to originate a stand alone inventory production. Usually only one row is set to 1; if we find more than one row, the first one is automatically chosen; if no row contains a flag set to 1, no inventories will be produced. The useWarehouse flag of the file unitData/unitBasicData.txt does not operate in the rows describing complex production units (it can be either 0 or 1).



	A	B	C	D	E
1	3				
2					
3		201	1	1	0
4		2001	1	1	0
5		2	1	1	1

Figure 10. An example of complex production unit, reported in the worksheet labeled 2.

After the rows containing the fixed and variable costs and the warehouse flag, we are planning to introduce two matrixes related to the production unit setup (not yet implemented) reporting setup costs sc_{ij} and setup time st_{ij} needed to change production from state i to state j.

A chain of worksheets labeled with the production unit numbers follows the introductory worksheet. In Figure 10 we have an example of complex production unit (that numbered 2 in Figure 8), with 3 possible production phases: the 201 production step, with its fixed and variable costs expressed for one time unit (in the example, all set to 1) and a flag 0 for the inventory production; the 2001 production step, with similar data; the 2 production step, with the inventory production flag set to 1.

---

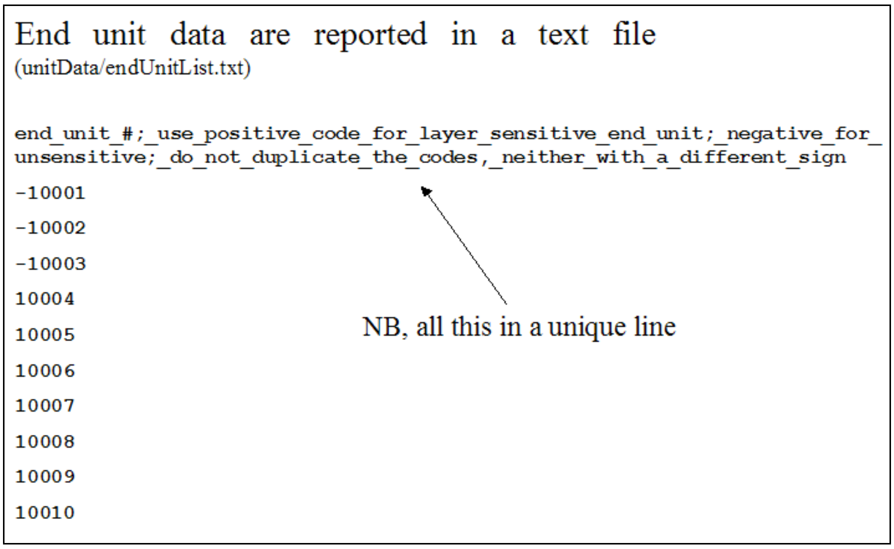
[12] To access spreadsheet data from a Java environment we use the ExcelReader.java class, written by Michele Sonnessa (sonnessa@di.unito.it) and based upon the Andy Khan's excelread library (http://www.andykhan.com/excelread/)

[13] The sequence of the various lines is not relevant, they can be introduced in any order.

3.3. END UNITS

A recipe can be concluded by an ordinary production step (it may be also a trading activity if we want to simulate also the commercial side of the production chain): in this case the order is dropped out from the simulation. Obviously, some accounting is made to record the effect of the production on the enterprise balance sheet.

A recipe can be also concluded by an e code followed by a number identifying a unit that is not a production node in the simulated enterprise, but a node representing an actual or a virtual (when we are producing or procuring something that is not material, such as services) stockpile, where we place, really or metaphorically, the result of the recipe.



End unit data are reported in a text file

(unitData/endUnitList.txt)

```
end_unit_#;_use_positive_code_for_layer_sensitive_end_unit;_negative_for_
unsensitive;_do_not_duplicate_the_codes,_neither_with_a_different_sign
-10001
-10002
-10003
10004
10005
10006
10007
10008
10009
10010
```

NB, all this in a unique line

Figure 11. End units list, with positive code (is sensitive to layers) or negative one (if unsensitive to layers).

End units are described using the file unitData/endUnitList.txt that contains the information of Figure 11. The first line is mandatory, written exactly as is, to force the user to pay attention to the content of the file.

The numerical code of the end units is the same of the components that they contain. Codes have to be different from those assigned to the production units.

End units are of two kinds: layer sensitive (about layers, see above 2.1.), identified by a positive code, and layer unsensitive, identified by a negative code. This difference is relevant in the procurement processes, to determine whether a component (procured or produced internally) has to be differentiated per layer when we are looking for it in an endUnit.

3.4. PROCUREMENTS IN THE DW SIDE (AND ZERO TIME STEPS)

From the point of view of the DW side, a procurement process is a situation in which a recipe, in the form of Figure 12, orders to a production unit (the one able to perform the step that requires the procurements: *n2*, in our case), to look for end units able to provide the required components, both internally produced or procured.

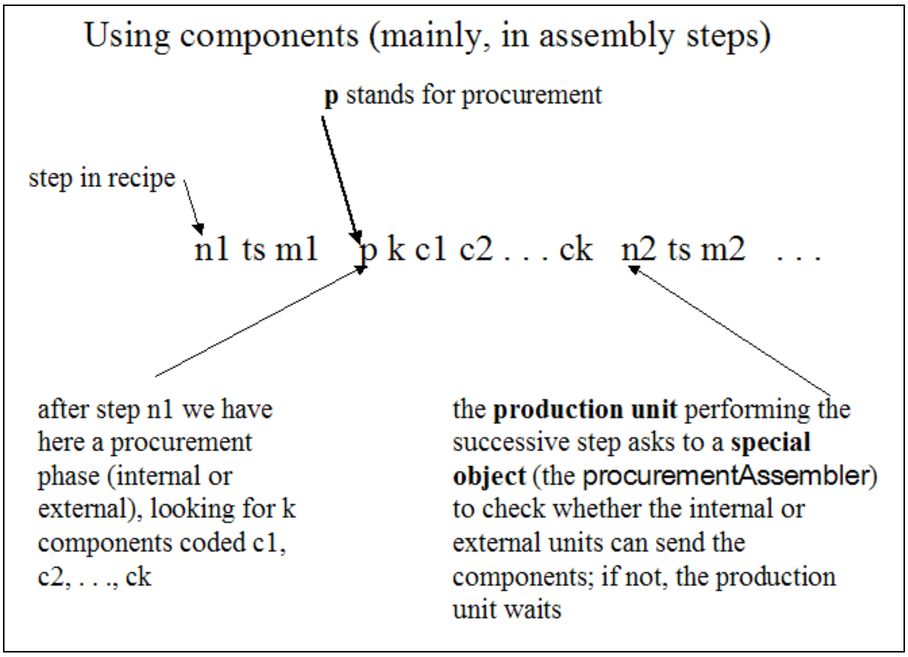

Using components (mainly, in assembly steps)
p stands for procurement
step in recipe
n1 ts m1 p k c1 c2 . . . ck n2 ts m2 . . .
after step n1 we have here a procurement phase (internal or external), looking for k components coded c1, c2, . . ., ck
the production unit performing the successive step asks to a special object (the procurementAssembler) to check whether the internal or external units can send the components; if not, the production unit waits

Figure 12. Production units use components (to be procured or internally produced) described in recipes.

As seen above, end units have the same code of the parts they contain, so we are here looking for end units *c1*, *c2*, …, *ck* or 121, 34 and 73 of Figures 12 and 13 (if one of them is missing in the file unitData/endUnitList.txt, the simulation is stopped, with an error message).

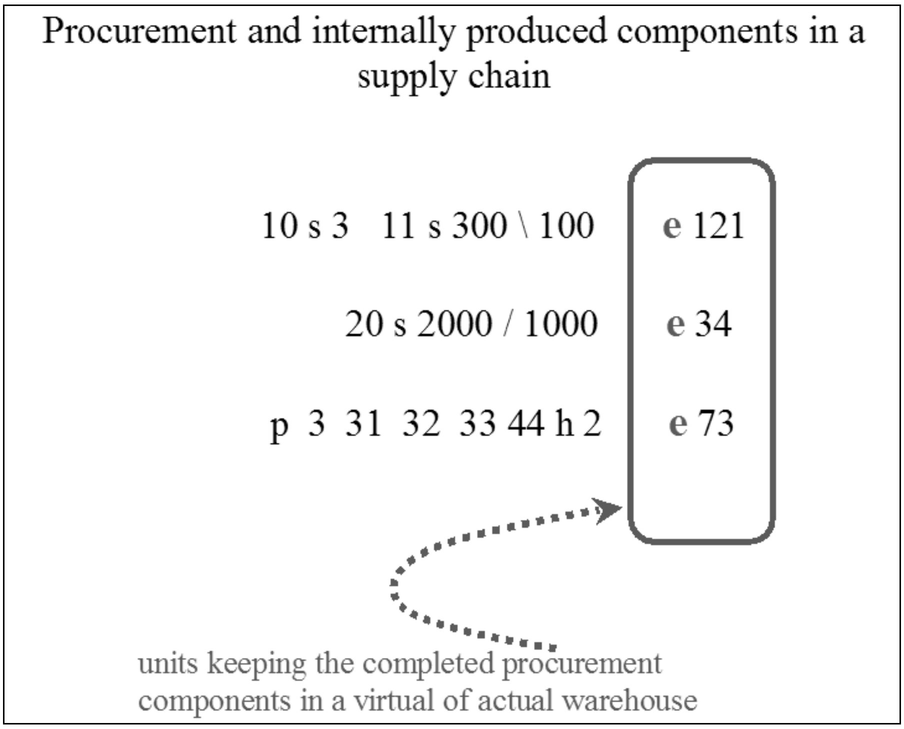

Procurement and internally produced components in a supply chain
10 s 3 11 s 300 \ 100
20 s 2000 / 1000
p 3 31 32 33 44 h 2
e 121
e 34
e 73
units keeping the completed procurement components in a virtual of actual warehouse

Figure 13. Procurements and internally produced components are held in instances of the EndUnit class.

The end units can be empty, if the orders containing the recipes necessary to produce the required components have not been launched in the due time; this is a key problem in a supply chain and it is useful to simulate this kind of emergence.

The contents of an end unit can be subdivided by layers (see above 2.1.): so a specific layer can be empty.

If some component parts are lacking, the production unit that is looking for them waits, pausing its production (in the future, we will develop a queue managing feature to avoid that also the residual part of the list had to wait in similar cases until the head of the list is made empty).

In the example of Figure 13 we have: (a) the component part 121, probably produced internally, since we describe in detail its production[14]; (b) the component part 34, probably produced externally, as we describe it as a black box with a stand alone batch producing 1000 items in 2000 seconds; (c) the component part 73, may be internally produced, in which the unique step describer, the 44 lasting 2 hours, calls for parts 31, 32 and 33 as described by the procurement description ‘p 3 31 32 33’.

A procurement sequence cannot be applied to a sequential batch step[15]: in that case we have to introduce a zero time step 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). So the correct recipe is p 2 21 32 1111 s 0 5 s 1000 \ 500 and not p 2 21 32 1111 s 1000 \ 500.

3.5. CHOOSING WHICH IS DOING WHAT, WHEN MORE THAN ONE UNIT IS ABLE TO DO A STEP

An important feature in DW side of the simulation concern decisions.

As seen above in 1.3., each order (both new or old) makes an inquiry into the world to discover if one or more production units can perform its first undone step; if more than one unit is able to perform the required step, we have to choose one of them; the choice amid the valid units can be made following several criterions.

At present, we have the following possibilities, which are related to the value of the parameter unitCriterion:

0 - the first of the valid unit as listed in the unitData/unitBasicData.txt file;

1 - one of the valid units is chosen in a random way, which is a realistic approximation of many actual situations;

2 - the valid unit with the shortest waiting list (in case of a few candidates with the same value of the shortest waiting list, criterion 0 is used among them).

We stress that in the future this will be a key feature of *jES* (not yet implemented), allowing human interventions: to experiment different situations and solutions, but also to train people and to discover how people decide; finally, this is a window open toward the introduction of sophisticated optimization tools such as genetic algorithms (GA) and classifier systems (CS).

---

[14] A step of 3 seconds of type 10, followed by a sequential batch requiring a set of 100 orders to be executed, globally in 300 seconds.

[15] This is due to internal code reasons.

To implement human interventions, we can easily include real economic subjects into the simulation framework, introducing artificial agents acting as avatars[16] of actual agents. Artificial agents will ask their represented personality (e.g., via a web page) what to do (in this case, what unit we chose to assign the production) while the simulation is running.

To implement soft computing techniques such as GA and CS, we plan to use the simulator (*jES*) to evaluate the fitness of populations of solutions (GA) or of populations of rules (CS).

3.6. RESOURCES REQUIRED BY ACTIVE PRODUCTION UNITS AND STOPPED UNITS

In the future we will implement the possibility for production units to lock, when activated, other production units, to simulate situations in which we have a resource that can alternatively be applied to several units, but simultaneously only to a subset of them.

We will also introduce the possibility of stopping production units[17] (a few of them or all), e.g., to reduce activity in a production shift or to put out of use a production unit (e.g., to emulate a breakdown).

3.7. NEWS PROPAGATION, INFORMATION AND COOPERATION

With the term news we designate information about an incoming order that a production unit sends to the successive units listed in the recipe (Figure 14).

Technically pieces of news are objects, so a piece of news is referred to as “a news” and more of them are called “newses”. Units send information one to another if they are linked in the matrix contained in the unitData/informationFlowMatrix.txt; that is, a matrix (without spaces in rows), where the rows refer to sending units and the columns to receiving ones. To use news propagation, at present the number of the units must be consecutive and sorted in ascending order in unitData/unitBasicData.txt.

A “1” in a cell means that the unit on the row can send messages to the unit on the columns; with a “0” the message cannot be sent. Production units never send news to themselves, so the values on the diagonal are not relevant.

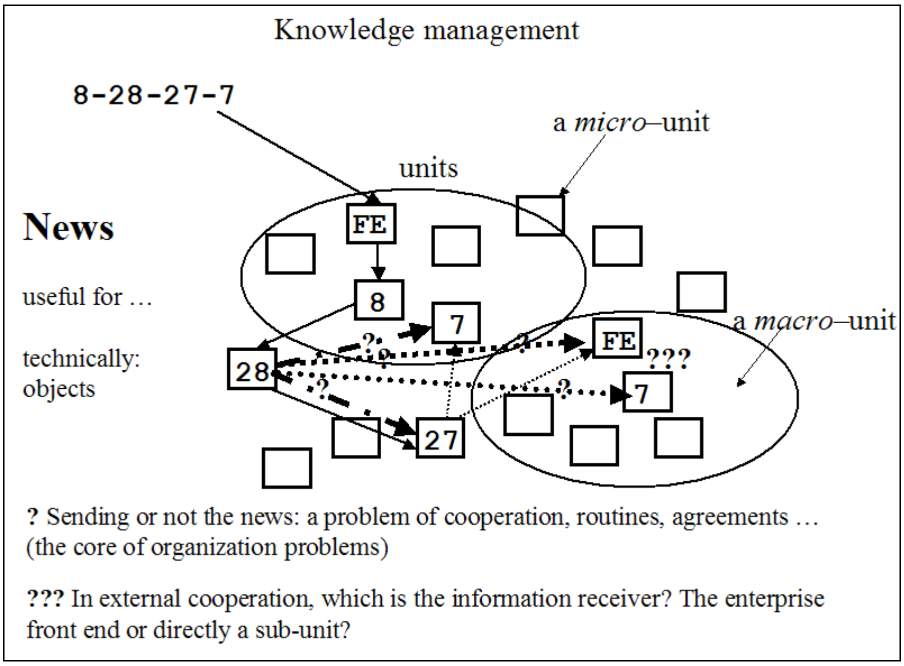
The whole information process is active if the parameter useNewses is set to *true*; the number of steps ahead in the recipe to be taken into account to find the production units to which a unit has to send information is defined by the infDeepness parameter.

The use of the informationFlowMatrix is managed by an instance of the class InformationRuleMaster.

---

[16] From www.babylon.com: s. avatar (Hindu mythology) earthly incarnation of a god, human embodiment of a deity; (Internet) online image that represents a user in chat rooms or in a virtual “space”.

[17] May be the better solution is to place this kind on information into the file recipeData/orderSequence.xls; see below 5.2.



? Sending or not the news: a problem of cooperation, routines, agreements … (the core of organization problems)

??? In external cooperation, which is the information receiver? The enterprise front end or directly a sub-unit?

Figure 14. News and elementary knowledge management.

3.8. WAREHOUSES AND STAND ALONE DECISIONS OF INVENTORY PRODUCTION

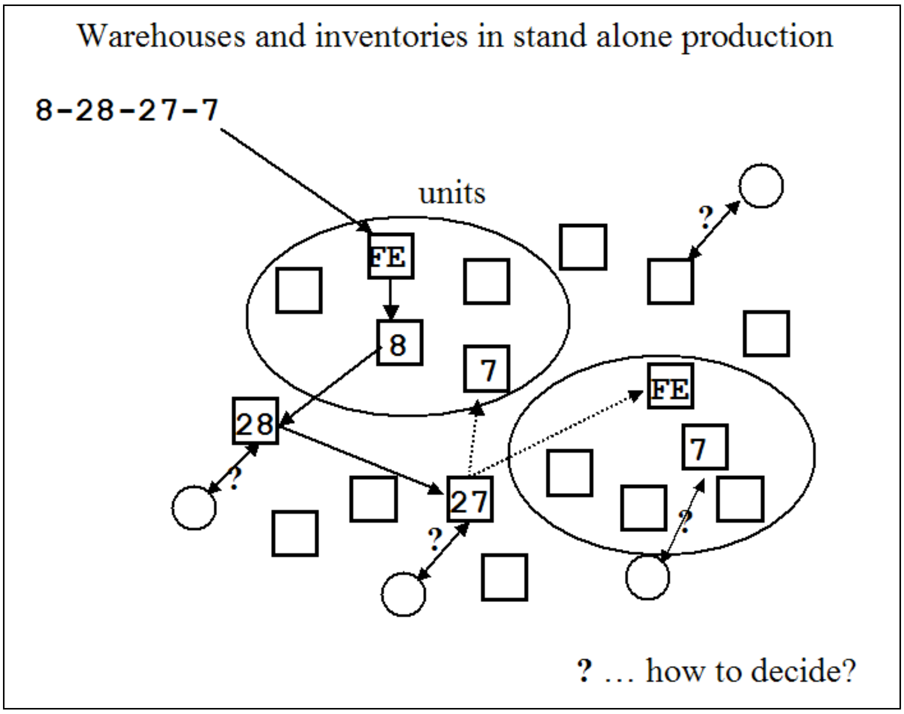


Figure 15. Warehouses and the stand alone production of inventories. The squares are the units and the circles are the warehouses (note that not all the units have a warehouse)

We have here again a decision problem: if the useWarehouse parameter is set to *true* and the flags, related to simple or complex units, allow this kind of activity (see above 3.1. and

3.2.), our production units can accumulate inventories (Figure 15) in their warehouses, limitedly to their specific production ability (for the case of multiple abilities, see 3.2. above).

Each unit in idle status, before producing for its warehouse, asks to its instance of InventoryRuleMaster[18] if it has effectively to produce. The rule master gives its agreement to the inventory production if the warehouse level would result to be less than the maxInWarehouses parameter value.

If the use of newses (see above 3.7.) is active in the simulation, the agreement to the production of inventories requires that (i) the warehouse level is less than maxInWarehouses and there exists a number of newses, signaling incoming orders, greater than the nOfNewsesToProduce parameter; or (ii) lacking this second condition, the warehouse level is less than minInWarehouses parameter value.

Stand alone production of inventories is not based on recipes (such as those of the parts required in the procurement processes) but is simply the application of the unit capability, if it has sense to work alone to accumulate inventories. If inventories exists, as explained in 1.3., the production can be replaced by using inventories related to each specific production step; here, more than one order can be treated in a single tick by a specific production unit, if it has parts in inventories.

3.9.DETAILS IN DW SIDE

3.9.1. TECHNICAL DETAILS ABOUT THE ASSIGNMENT PROCESS

The assignment process, seen in 1.3. above, requires the explanation of some technical details, related to the case of presence of a repeated step in a recipe. Consider the recipe 101 s 5 101 s 4 with the production units 11 and 12 both able to execute the step 101. In a simulation run we could discover a sequence of use of the unit 12 and then of the unit 11, with an unrealistic effect. Setting assignEqualStepsToSameUnit to *true*, the assignment of the second 101 step uses the same unit of the first one. A linked problem is related to the way we reassign the order to the first used unit: maybe we want that its execution is continued without interruption, so we have to put the reassigned order at the first place of the waiting list, accordinf to a LIFO criterion. This effect is obtained setting to *true* the sameStepLifoAssignment parameter.

So the first parameter sets the condition that manages the assignment of an order with the same step to the same unit (a realistic condition); the second one sets the way in which an order is re-assigned to a production unit if we are making the same kind of step (LIFO or FIFO).

A minor problem is related to cases in which we do not want, for some technical reason, that a recipe 101 s 5 101 s 4 is treated exactly as a recipe 101 s 9 when the two parameters are set to *true*. Here a trick is that of writing the recipe as 101 s 5 1010 s 0 101 s 4, where 1010 is a fictitious production unit and the 0 identify a zero time production step (see above, 3.4.).

---

[18] We have a few RuleMaster classes in *jES*, telling the their agents what to do; this scheme is related to the idea of the ERA (Environment, Rules, Agents) framework, introduced at web.econ.unito.it/terna/ct-era/ct-era.html.

3.9.2. EFFECTS OF THE TIME SPENT BY AN ORDER IN A PRODUCTION UNIT

If maxTickInAUnit is set to a positive value, orders waiting in a production unit for more than maxTickInAUnit are dropped and disappear from the simulation.

4. NEWLY BACK TO THE WD SIDE

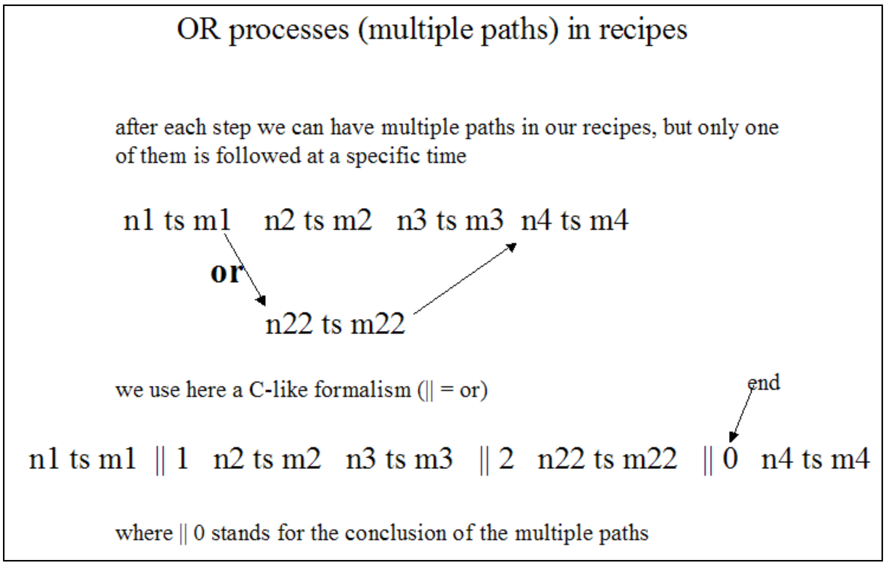
4.1. A TRIPLE FORMAT FOR THE RECIPES

The way the recipes of the orders are written is triple: external, intermediate and internal. The user always writes its recipes in the external format, creating a recipe repertoire into the file recipeData/recipes.xls placed in the folder where *jES* is running. For example, look at appCases\Case_i\Caso-1_base in the distribution of the program.

The external (human readable) and the intermediate formats of the recipes are easily understood on the base of the comments contained in the OrderGenerator.java file. The internal one, apparently poor in details, can be examined looking at the comments and at the instructions contained in the Order.java file. Anyway we write the recipes using the external code; the translation mechanism from the external to the intermediate code is contained in OrderDistiller class; from the intermediate to the internal, in Order class.

A technical detail: in the internal format, the particulars of the recipes are reported in accompanying objects; the order contains a recipe and several lists of those objects.

4.2. OR PROCESSES



OR processes (multiple paths) in recipes

after each step we can have multiple paths in our recipes, but only one of them is followed at a specific time

n1 ts m1
n2 ts m2
n3 ts m3
n4 ts m4
or
n22 ts m22

we use here a C-like formalism (|| = or)

end
n1 ts m1 || 1 n2 ts m2 n3 ts m3 || 2 n22 ts m22 || 0 n4 ts m4

where || 0 stands for the conclusion of the multiple paths

Figure 16. An or process, with its branches (|| 1 and || 2)

We can insert an or choice in a recipe using the format introduced in Figure 16. In the example reported here, after step 1, we can have the sequence with the two steps n2 n3 or that with the unique n22; then the execution of the recipe continues with the step n4. The number of branches into the or sequence has no limits.

What branch chooses the sequence within the or? We have to look at the values assigned to the orCriterion parameter:

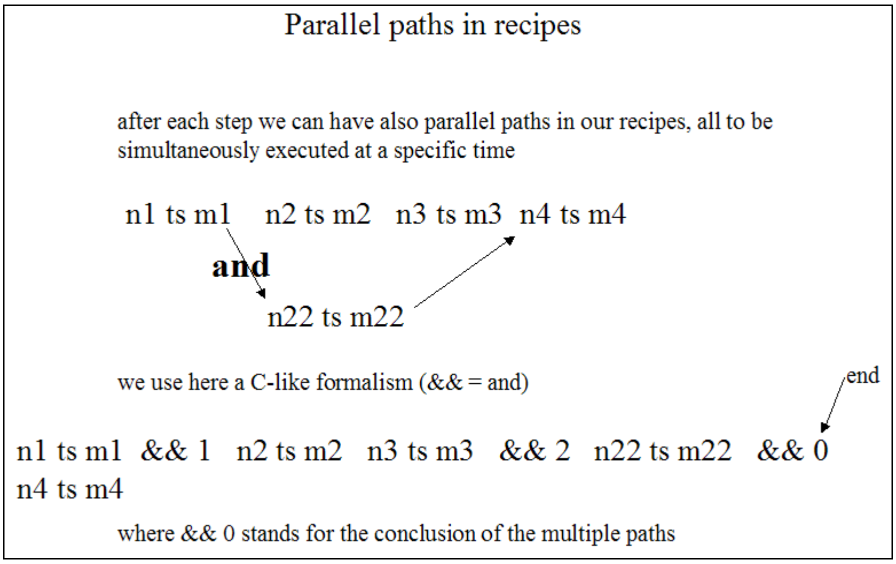
- 0, all branches are executed in sequence (useful only for test purposes);
- 1, the first branch is chosen;
- 2, the second branch is chosen;
- 3, the choice of the branch in made randomly (a good simulated solution if we have to balance the loading of several production processes);
- 4, the branch whose first step has the shortest waiting list is chosen;
- 5, the choice of the branch is based on the result of a computational step (see below: 4.4. about computational capabilities and memory matrixes; 4.5. about computational capabilities and or sequences).

An example of or sequence is the following, containing also a procurement process (see above 2.3. and 3.4.) in one of the or branches:

10 s 3 c 1997 1 2 12 s 0 || 1 11 s 2 p 1 101 10 s 1 9 s 2

|| 2 c 1995 1 0 1 s 0 14 s 3 || 0 6 s 2

Where || 1 and || 2 are two nodes each opening a branch of the or sequence and || 0 ends the sequence; in the first branch we can identify the simple procurement sequence ‘p 1 101 10 s 1’.



Parallel paths in recipes

after each step we can have also parallel paths in our recipes, all to be simultaneously executed at a specific time

we use here a C-like formalism (&& = and)

n1 ts m1 && 1 n2 ts m2 n3 ts m3 && 2 n22 ts m22 && 0
n4 ts m4

where && 0 stands for the conclusion of the multiple paths

Figure 17. An and process, with its branches (&& 1 and && 2)

The or sequences are managed by the code of *jES* in a simple way: all the steps of the discarded branches are immediately signed as executed, then execution proceeds in sequence, avoiding those steps fictitiously executed.

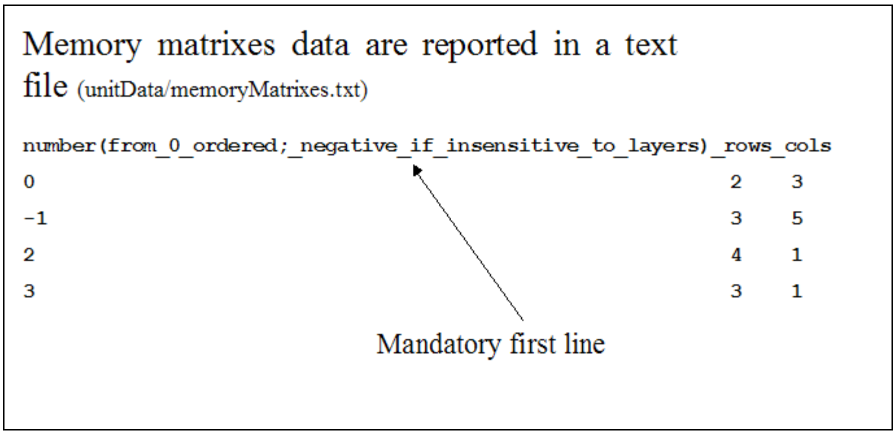
Besides a procurement process, an or sequence can contain also a computational step (see below 4.4).

4.3. AND PROCESSES AND PARALLEL PATHS

and processes are not yet implemented; the and described in Figure 17 is asynchronous, as both the branches of the and sequence have to be executed, but independently as regard to time. We can also imagine a synchronous and process whit all the branches to be executed together.

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[19] 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;_negative_if_insensitive_to_layers)_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 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

---

[19] 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:

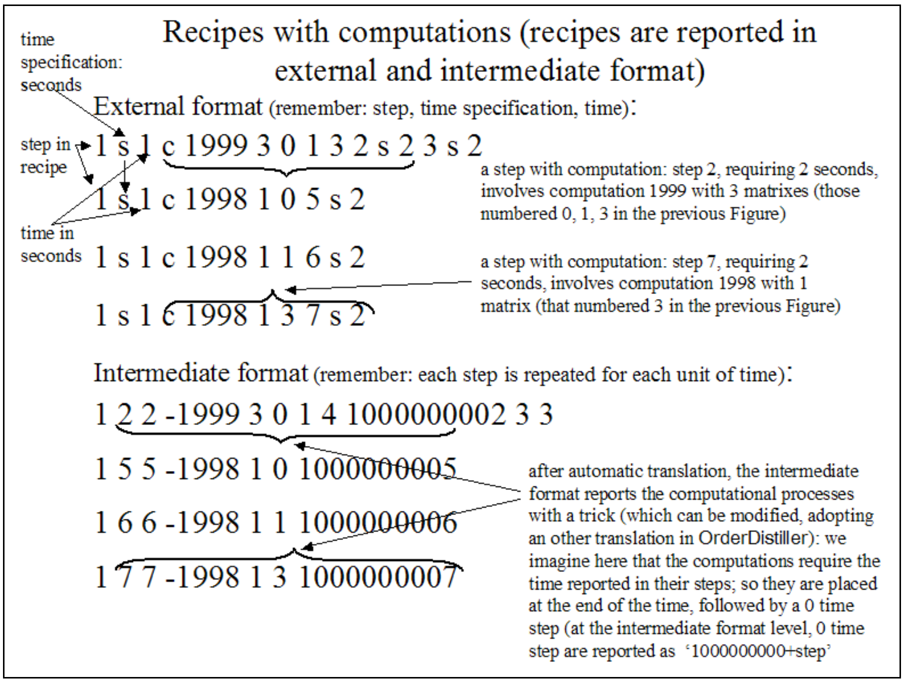
```
Class c = this.getClass();
Method m = c.getMethod("c"+(-1*t),null);
m.invoke(this, null);
```

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[20].

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* m₁ … mₙ’ where c is mandatory, *code* is the code of the computation, *n* is the number of matrixes to be used and ‘m₁ … mₙ’ are the ID numbers of those matrixes, as reported in the file unitData/memoryMatrixes.txt (Figure 18).



Recipes with computations (recipes are reported in external and intermediate format)

External format (remember: step, time specification, time):

```
1 s 1 c 1999 3 0 1 3 2 s 2 3 s 2
1 s 1 c 1998 1 0 5 s 2
1 s 1 c 1998 1 1 6 s 2
1 s 1 c 1998 1 3 7 s 2
```

a step with computation: step 2, requiring 2 seconds, involves computation 1999 with 3 matrixes (those numbered 0, 1, 3 in the previous Figure)

a step with computation: step 7, requiring 2 seconds, involves computation 1998 with 1 matrix (that numbered 3 in the previous Figure)

Intermediate format (remember: each step is repeated for each unit of time):

```
1 2 2 -1999 3 0 1 4 1000000002 3 3
1 5 5 -1998 1 0 1000000005
1 6 6 -1998 1 1 1000000006
1 7 7 -1998 1 3 1000000007
```

after automatic translation, the intermediate format reports the computational processes with a trick (which can be modified, adopting an other translation in OrderDistiller): we imagine here that the computations require the time reported in their steps; so they are placed at the end of the time, followed by a 0 time step (at the intermediate format level, 0 time step are reported as ‘1000000000+step’

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).

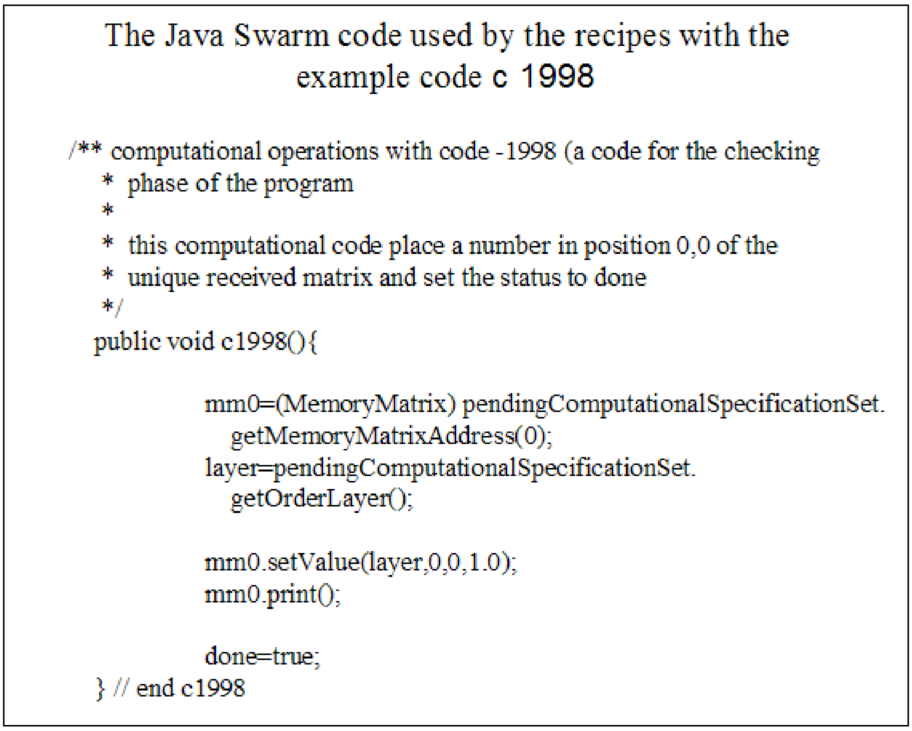
In Figure 19 we can see how computational codes are represented following their external and intermediate formats (anyway, remember that we write recipes in external code). Pay attention: computational codes at the intermediate format representation level are reported as negative, following the internal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

---

[20] -0, as a number, is equal to 0, so the first matrix cannot be declared unsensitive to layers.

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`[21].



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`[22]. 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).

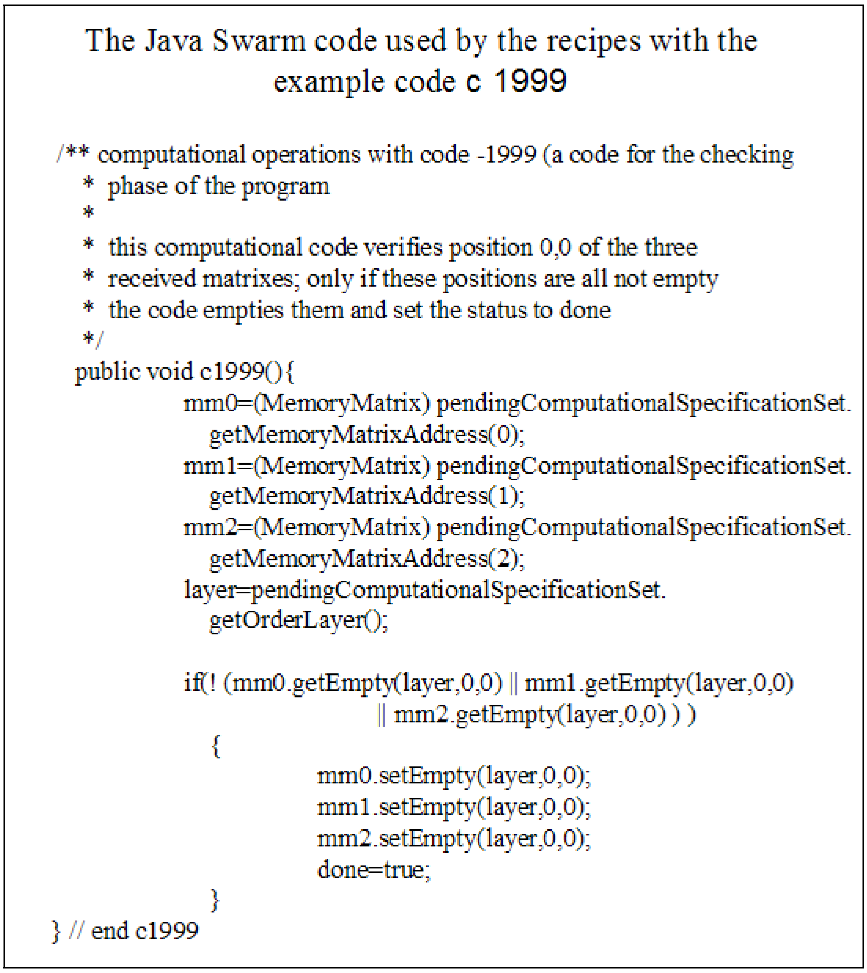
---

[21] 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*.

[22] See previous note.

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*).



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**).

4.5. COMPUTATIONAL CAPABILITIES AND or SEQUENCES

If orCriterion is equal to 5 (see above 4.2.) computational results are also useful to choose what branch to execute in an or process.

We choose the branch whose number is stored in (x,0) position in the memoryMatrix designated by the orMemoryMatrix parameter; the matrix may be sensitive or insensitive to layers. The range of the branch number is from 1 to the number of branches.

x is 0 if the first node in the or sequence is numbered 1; is *kk* if the first node is numbered 10*kk* with *kk* in the interval 00 to 99. If orCriterion is not equal to 5, the codes 10*kk* are considered as is they where 1.

A computational sequence can be included in an or branch adding great flexibility to the computational processes[23].

---

[23] This aspect is strategic for the development of *jES* in simulating both the financial side of the enterprise and the enterprise information system.

5. RUNNING A SIMULATION

5.1. USING THE orderGenerator OR THE orderDistiller

The simulation mechanism is activated by the orders that contain the recipes. The recipes report the steps to be done and the time necessary to accomplish each step.

To run a simulation, we have to define the time parameters: ticksInATimeUnit says how many ticks, of the simulation clock, are necessary to complete a unit of time[24] (a day, a shift, …; note that with this value we set the time granularity, i.e., its details, from 1, which is the minimum granularity, to any value); timeToFinish is the number of ticks (ticks in a time unit multiplied by time units) after which the program stops the simulation (if zero, never). The graphics, in the axis *x*, adopt the same value.

The orders are, alternatively: (i) randomly generated by the orderGenerator instance of the OrderGenerator class, when we are testing the code or reproducing a situation in which we have no information about the sequences of the orders (and so we have to generate them in a random way); (ii) distilled from a repertoire of recipes, following a time schedule, by the orderDistiller instance of the OrderDistiller class; this is the common case of application of *jES* to actual situations.

In the first case, using orderGenerator, all the recipes are internally generated using the same basic time interval (that we can assume to mean seconds, hours, days, …); we have to use consistently that time interval to measure fixed and variable costs both in the table of the file unitData/unitBasicData.txt (simple units, see 3.1. above) and in the spreadsheet reporting the data of the complex units (see 3.2. above) unitData/units.xls.

In the second case, using orderDistiller, we introduce the When Doing What (WDW) formalism (see 5.2. below). Recipes can contain the time expressed in seconds or in minutes (orderDistiller automatically converts minutes in seconds; in the future also hours and days will be introduced). Internally, in a specific simulation we use only one time unit, always the same; in orderDistiller we can set the smallest time-unit in a specific simulation and then establish the ratios with the other measures used into the recipes.

Also in the orderDistiller case, the time measure (the smallest one; seconds, at present) has to be consistently used to measure fixed and variable costs, for simple units (see 3.1. above) in the table of the file unitData/unitBasicData.txt and, for complex units (see 3.2. above) into the various sheets of the file unitData/units.xls.

We have to develop an intelligent version of the distiller, able to deal both with time scale changes and with changes in the time-unit used to measure quantities in recipes. Recipes contain references to time to measure the length of each step and of each batch process, either sequential or stand alone; if the time unit changes, to speed up the simulation (with less granularity in time description) we have to convert hundreds of seconds or thousands of seconds into a single time unit, modifying automatically the interpretation of the recipes, also in batch productions. In the same way, if we change the basic unit used to measure the quantity of orders (e.g. one stays for one hundred or one thousand), we have to remember that the recipes contain references to quantities produced in each sequential of stand alone batch:

---

[24] This definition is relevant for accounting purposes; from the production point of view the results are exactly the same if we describe ten time units each of one tick or a time unit with ten ticks and then place the same order launches in each of the ten elementary ticks.

those quantities have to be adjusted in size. Besides this, if we change the time unit or the quantity unit, also the contents of the schedule of the events have to be reinterpreted, both with respect to the number of orders to be launched and to the time steps to be considered to launch those orders. Finally, all the fixed and variable costs have to be adjusted in size, according to the modified time-unit.

At present all changes have to be made by hand modifying the contents of the files described into the next paragraph. It will be probably impossible to automate fully those operations, but some step in this direction is possible.

5.2. WHEN DOING WHAT (WDW)

The WDW formalism is a set of conventions useful to create a repertoire of recipes and to schedule them over the time.

The repertoire is contained in the file recipeData/recipes.xls; the recipes are reported in the first worksheet of the spreadsheet[25]; the name of the worksheet (the default one or that chosen by the user) will be reported in the error messages, if any.

We have examples of this file both in the appCases/ folder (see the various subfolders) and in the testCases/ folder (see the subfolders containing recipeData/).

The format does not require the use of particular rows and columns; only the order (from left to right and from top to bottom) has a meaning; empty cells are allowed. Anyway: rows containing comments have to start in column 1 with a # sign alone in the first cell (and require no ending sign); each recipe starts with a name - better if space free - and an ID number (used by ourselves and by the program for error messages), then we have the body of the recipe as seen above and below in the various Figures and examples and, finally, we have a mandatory "**;**" sign ending the recipe. All the elements of a recipe, such as name, ID number, numbers of the steps, time codes, time lengths, special codes \, /, c, p, e, ||, &&, ;, etc. are written one per cell. We can use colors and break long lines to improve readability. We can also use notes and comments written with the spreadsheet internal tools: the class reading the spreadsheet content does not see these elements.

The schedule of the order launches is contained in the first worksheet of the file recipeData/orderSequence.xls; we have examples of this file below.

The format is free as above, with an item per cell, and the comment lines have the same limitation expressed for the file containing the recipes. In the same way, we can use colors, break lines and use note and comments written with the internal tools of the spreadsheet.

Each block of the file (ended by a **;** sign) describes the events occurring in a tick of the simulation clock; each block is related to a tick, in order, starting with the first and regardless the number used to identify the block. A block starts with an ID number (may be ordered, starting from zero or one, but this is not mandatory) and contains the recipe of the orders to be launched in that tick; each recipe code is followed by a * sign and by a repetition factor (one if we launch one order; *n* if we launch *n* orders); the first item after the ID or between the launches of the recipes may also contain a layer code (see 2.1.): an "l" letter followed by the layer number. In the example, 4 is the block number, 33 and 34 are recipes, 2 and 3 are layers

---

[25] We can produce the spreadsheet either using a proprietary code or employing an Open Source one, such as OpenOffice (www.openoffice.org).

(extra spaces are introduced to improve readability; each item of the block in inserted in a cell, also leaving empty cells within the sequence):

4 l 2 33 * 1 l 3 33 * 3 l 2 34 * 1 ;

We will obtain, in this tick, one order with recipe 33 belonging to layer 2; three orders with recipe 33 belonging to layer 3; one order with recipe 34 belonging to layer 2. Once a layer is set, the choice is valid also in the successive blocks, until a new choice is made. Layer 0 is the default one.

The layer numbers, as said in 2.1. above, must be less than the value of the totalLayerNumber parameter.

The contents of the recipeData/orderSequence.xls file are executed one per tick[26]; if in a tick nothing has to occur, we have to prepare an empty block with the ID number and the ending ; sign.

Once we are at the end of the recipeData/orderSequence.xls file, the execution restarts with the first block etc.

The file recipeData/orderStartingSequence.xls contains exactly the same information of the recipeData/orderSequence.xls file, but it is used only once when the simulation starts. The purpose is that of establishing a special starting sequence of orders, e.g., to assure immediately the presence of procurements, eventually launching orders containing special recipes used only in the opening phase of the simulation.

Both recipeData/orderStartingSequence.xls and recipeData/orderSequence.xls must exist: if one of them is actually unnecessary, place in it a unique empty block, such as 0 ; (the ID can be any number).

5.3. TECHNICAL DETAILS IN RUNNING THE SIMULATIONS

If you open the terminal (a Cygwin or Linux/Unix terminal) and go to the folder where you have *jES*, typing make run a default random simulation starts[27]. make run is the basic command, sufficient if you use *jES* as is. If you have to modify *jES* look at the README.TXT file[28]. An alternative to the basic command is make runBig used to improve the quantity of memory assigned to the simulation for huge problems.

To use one example of the folders appCases/ and testCases/ delete, from the main folder of *jES*, the folders unitData/ and recipeData/ (if it exists) and the file jesframe.scm; then copy and paste the analogous folders and file, from the folder of the chosen example to the main folder of *jES*[29].

---

[26] We repeat that from the production point of view the results are exactly the same if we describe ten time units each of one tick or a time unit with ten ticks and then place the same order launches in each of the ten elementary ticks.

[27] Look at the file readme_linux_windows.txt.

[28] Besides the README.TXT file, in the folder of *jES* we have also several script files for Windows or Linux (using a Bash shell): look always to the README.TXT file for explanations.

[29] In testCases/development/using_OrderGenerator_basic_run/ we have the starting configuration.

6. ACCOUNTING[30]

jES is capable of automated accounting of the production unit activities and of the order accomplishment. We have double accounting: (i) an order is charged of variable and fixed costs related to the production units that it has used in the production process (up the present time); (ii) idle production units do not account for variable costs and their fixed costs are not charged to any order. A unit producing inventories in the stand alone way (see above 3.8.) accounts fixed and variable costs, included in the inventory evaluation; when the step of an order is accomplished using inventories, the related costs are charged to the order. Inventories cause a financial cost, measured by an interest rate; this cost is based on the inventoryFinancialRate parameter, expressed as annual unitary rate[31]. The same has to be done for the components kept into the end units (not yet implemented).

Finished orders are fully accounted on the side of the revenues, such as partially accomplished orders.

6.1. COSTS

About costs, remember that fixed and variable costs are stated per each unit (and in case of a complex unit, per each activity) as explained in 3.1. and 3.2. above. Costs are related to a time unit, as recalled also in 5.1.

In the Costs/ folder we have the output files about costs that, as reported into the Costs/readmeCosts file, are:

1) totalDailyCosts.txt: the sum of fixed and variable daily costs. It is set to 0 at the beginning of each day, i.e., block of ticks (ticksInATimeUnit, see above 5.1.) that we interpret as a time unit (a day, a shift, …);
2) totalCosts.txt: the cumulated sum of totalDailyCost.txt data. It is set to 0 at the beginning of the simulation;
3) finishedOrderCosts.txt: the final cost of the concluded orders from the beginning of the simulation; it does not include: fixed costs registered in idle units, fixed and variable costs related to the inventory production, semi-manufactured products and financial costs;
4) dailySemimanufacturedOrderCosts.txt: the costs of the orders in production at the end of a specific day or time unit;
5) totalInventoryFinancialCost.txt: the financial cost of inventories from the beginning of the simulation.

The value to which we apply the interest rate follows the inventoryEvaluationCriterion, with the possibilities:

1 = variable costs,

2 = fixed+variable costs,

3 = value *v*, see 6.2. below.

---

[30] Look at 10. below, for a few examples of accountancy.

[31] The annual rate is applied on a daily basis dividing it by 200 (number of working days per year, in an approximate way); a day is a time unit composed of ticksInATimeUnit ticks (see above 5.1.); if we are using another time unit (e.g., a shift) we have to set properly the rate used or to modify the Unit.java code at the line: `eSFrameModelSwarm.getInventoryFinancialRate()/(float) 200;`

The criterion 3, in the financial cost determination is internally substituted by the criterion 2, because it would be a non-sense to apply financial costs to virtual revenues.

6.1.1. A FEW REMARKS ABOUT COSTS

Above we introduced above the cost view from the production unit side; *jES* makes accounting also within each order, considering procured items; these costs are reported in the file log/concludedOrderLog.txt[32].

When a unit is operating, it accounts for fixed and variable costs; if it is idle, it accounts only for fixed costs. A unit is operating when: (i) it is making a step of a recipe contained in an order; (ii) it is producing inventories in a stand alone way (see 3.8. above).

From the production unit side, the inventory costs are accounted if inventories are produced. From the order side, when an order is passing in a production unit, its cost accounting is the same both if the step is presently produced and if it is retrieved from the unit warehouse using previously produced inventories (in the stand alone way), so the costs related to the inventory production is transferred to order production costs.

When a recipe, describing an internally produced or externally procured component of an order, is concluded, the result is placed in an end unit. The related costs are accounted from the unit side and, as previously seen, they will be included in the final evaluation of the order that used the stored procurements.

If noAccountingInFirstTimeUnit parameter is set to *false* we do not make cost accounting only for the first tick of the first time unit; if it is set to *true*, we do not make accounting for the whole first time unit. We have the possibility of not making accounting about fixed costs in the first tick of the first day (time unit: day, shift, ...) or in the whole first time unit (if we have prepared an empty orderStartingSequence.xls file describing a consistent situation) because in this cases we suppose that, unless we use warehouses with the immediate possibility of producing inventories, we have nothing in our world.

6.2. REVENUES

Revenues are accounted for finished orders as they would be sold; anyway, we can include the trade step in our recipes.

In the Revenues/ folder we have the output files about revenues, as reported into the Revenues/readmeRevenues file:

1) dailyRevenues.txt: the revenues from finished orders (evaluated using value *v*). It is set to 0 at the beginning of each day (or time unit);
2) totalRevenues.txt: the total of dailyRevenues from the beginning of the simulation;

---

[32] To read the file into an Excel spreadsheet, choose *text file*, starting the input from line 10, using comma *delimited fields* and, as *text separator*, the *quote character* " ; if you are using Excel with the *comma decimal separator* as the default option, in the *advanced option* choose the point (*thousands separator*, none).
To read the file into an OpenOffice Calc spreadsheet, choose *text csv file*, starting the input from line 10, with comma as *field separator* and quote " as *text separator*; if you are using Calc with the *comma decimal separator* as the default option, convert commas to points with a text editor.

3) dailyStoredComponentValue: the value of the inventories at a specific time, following inventoryEvaluationCriterion with the possibilities:

1 = variable costs,
2 = fixed+variable costs,
3 = value *v*.

4) dailySemimanufacturedProductRevenues.txt: the value of semi-manufactured orders at a specific time (evaluated using value *v*).

v is obtained as:

v = [# of ticks (also of the procured items, if it is a finished order)] *
revenuePerEachRecipeStep +
[costs (also of the procured items, if it is a finished order)] *
revenuePerCostUnit

If revenuePerEachRecipeStep=0 we use the second criterion and vice versa; revenuePerCostUnit, if used, is normally set to (1 + markup per unit).

A trick about externally made components (procured parts): we can charge the production units (and so the products obtained) both for fixed and variable costs or, better, including fixed costs into variable ones: if it is the case of pure procurements, we have no reason to account fixed costs when the external production units are idle.

6.3. BENEFIT

The enterprise benefit is reported in the Benefit/benefit.txt file. As explained in the Benefit/readmeBenefit file, the benefit from the beginning of the simulation is obtained as the result of the following operation:

totalRevenues (obtained from units, from the beginning of the run)
+ dailyStoredComponentValue (obtained from units, as current value of their warehouses)
+ totalSemimanufacturedRevenues (obtained from orders in execution)
- totalCosts (obtained from units)
- totalInventoryFinancialCosts (obtained from units, referring to their warehouses from the beginning of the run)

Note that the costs come from production units and include the fixed costs also for the idle units; revenues come from the orders.

6.4. IMPROVEMENTS OF THE ACCOUNTING SYSTEM

The accounting system, synthetically described in Figure 22, will be enhanced considering both the possibility of environmental accounting and the introduction of the capability of emulating enterprise information systems, may be via computational objects (see above 4.4.).

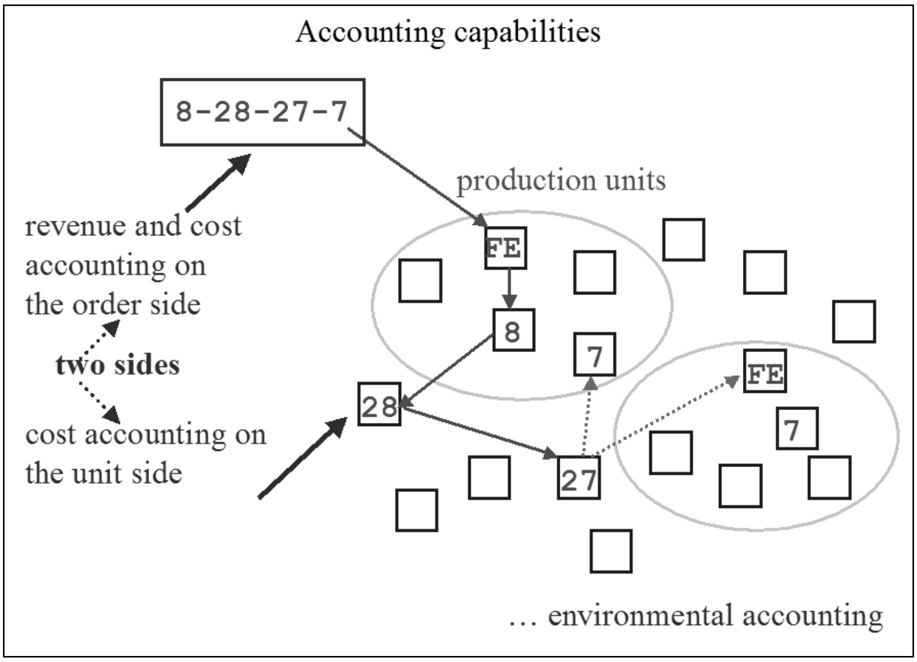


Figure 22. *jES* accounting system.

jES produces also a log/concludedOrderLog.txt file, with the detailed report of concluded orders; this file can be used as an input for other statistical analyses. Its format can be modified in the Unit class.

7. SIMULATION PARAMETERS, HISTOGRAMS AND GRAPHS

When we start *jES* (see above 5.3.) we obtain two parameter panes[33]. In the first pane we have the ESFrameObserverSwarm parameters:

- displayFrequency, which states the frequency of the display updating while the simulation is running: 1 for updating the display in each simulation clock tick; 2 for updating it every two ticks etc.
- verboseChoice, which, if *true*, requires the program to generate a lot on printed lines related to its internal activities;
- printMatrixes, see above 4.4.;
- checkMemorySize, that activates messages about used memory (in case of huge applications);
- unitHistogramXPos and unitHistogramYPos, that are the coordinates of the left upper corner of the window containing the histogram of the production units; the coordinates are related to the screen and expressed in pixels (in 0,0 we have the left upper point of the screen);
- endUnitHistogramXPos and endUnitHistogramYPos, that are the coordinates of the left uppur corner of the window of the end unit histogram, as before;

---

[33] When changing a parameter, remember to press the Enter key to effectively modify the value into the system. Logic values are *true* and *false*; the shortened forms *t* and *f* do not work here. In the jesframe.scm file explained at the end of the paragraph, we have to use #t and #f for *true* and *false*.

- timeToFinish, see above 5.1.

The second pane reports the ESFrameModelSwarm parameters[34]:

- ticksInATimeUnit, see above 5.1.;
- totalUnitNumber is the number of simple or complex production units populating our simulation (see above 3.1. and 3.2.);
- totalEndUnitNumber is the number of endUnits used (see 3.3.); when using the orderGenerator (see above 5.1.) a totalEndUnitNumber value greater than 1 determines the production of complex units with procurements, batches and computational steps;
- totalLayerNumber, see above 2.1.; when using the orderGenerator (see above 5.1.) a totalLayerNumber value greater than 1 determines the production of layered orders;
- totalMemoryMatrixNumber, see above 4.4.;
- queuesInUnitBySteps, see below 7.1.;
- sameStepLifoAssignment, see above 3.9.1.;
- assignEqualStepsToSameUnit, see above 3.9.1.;
- compareDisregardingUnits, see above 2.2.1.;
- maxTickInAUnit, see above 3.9.2.;
- useWarehouses, see above 3.8.;
- useNewses, see above 3.7.;
- maxInWarehouses and minInWarehouses, see above 3.8.;
- infDeepness, see above 3.7.;
- inventoryFinancialRate, see above 6.;
- inventoryEvaluationCriterion, 6.1. and 6.2.;
- revenuePerEachRecipeStep and revenuePerCostUnit, see above 6.2;
- nOfNewsesToProduce, see above 3.8.;
- nOfNewsesToBeCleared is the number of newses to be cleared after the decision of producing to increase inventories (a simple solution is to have it equal to nOfNewsesToProduce);
- nOfOrdersInNewses is the number of orders in a production unit waiting list for which newses are propagated; about newses propagation, see above 3.7.;
- orCriterion, see above 4.2. and 4.5.;
- orMemoryMatrix, see above 4.5.;
- unitCriterion, see above 3.5.;
- noAccountingInFirstTimeUnit, see above 6.1.1.;

---

[34] In old jesframe.scm files we can find the distillerMultiplicity parameter, no longer used. The correspondent variable exists internally to assure backward compatibility with those files.

- useOrderDistiller determines the internal generation of orders via orderGenerator (see above 5.1.) when set to *false*; if it is *true*, the order schedule follows the description of 5.2. above, with the WDW formalism;
- maxStepNumber is the maximum number of steps contained in a recipe describing an internally generated order, when using the orderGenerator;
- maxStepLength is the maximum number of units of time (e.g. seconds) attributable to the execution of a step in an internally generated order, when using the orderGenerator.

All the parameters above can be set either in the compiled code or in the pane described in this paragraph; moreover they can be set modifying the jesframe.scm file, which is written according to the *Scheme* formalism (*Scheme* is a dialect of the *Lisp* language; we can use it simply imitating the existing examples of the jesframe.scm file, both in the main *jES* folder and in appCases/ or testCases/ folders).

7.1. HISTOGRAM LEGENDS AND GRAPHS CONTENTS

The histogram "Procument int. or ext." reports the quantities of procured items ("q. in endU.") in each end unit. The histogram "Orders in Units" reports: the number of orders (or of steps of each order in each unit, if queuesInUnitBySteps is set to *true*) waiting for production in each unit ("Queues in u."); the quantities in the stand alone warehouses of each unit ("Quant. in w."); the number of orders waiting for procured parts (and so kept into the procumentAssembler object, giving the reason for the legend "Q. in proc.ass.").

The other graphs report the "Enterprise benefit" since the beginning of the simulation; the min, max and average "Waiting list" in all the production units; the min, max and average "Quantities in the warehouses" (if any) in all the production units; the "Ratio total time / total lengths" expressing the ratio between the actual production time of the done orders and the expected time as expressed in each recipe.

8. HOW TO OBTAIN *jES*

You can look for the latest version of *jES* at http://web.econ.unito.it/terna/jes/, looking for files such as jesframe-x.y.z.tar.gz, where x.y.z is the version number; the distribution contains also this explanatory file (how_to_use_jES.pdf) and a PowerPoint file (how_to_use_jES_(figures).ppt) reporting all the Figures used in this presentation. The file contains also *jESlet*, the "java Enterprise Simulation light experimental Tool", prepared for comparative and didactic reasons. *jES* is distributed under the Open Source Academic Free License, see license.txt in the distribution and www.opensource.org/licenses/academic.php.

9. FUTURE IMPROVEMENTS

Besides a lot of technical improvement reported in the todo.txt file of the distribution, the main improvement (anticipated in 3.5. above) of *jES* will be the introduction of agents representing decision nodes, where rules and algorithms (like genetic algorithms or classifier systems), or *avatars* of actual people, take action. Avatars' decisions are taken asking actual people what to do: in this way we can simulate the effects of actual choices; we can also use the simulator as a training tool and, simultaneously, as a way to run economic experiments to understand how people behave and decide in organizations.

10. A FEW EXAMPLES OF ACCOUNTANCY

All the files that are necessary to run the examples reported below[35], can be found in the testCases/ directory and used copying them (also whole directories, when necessary) into the main directory of *jES* (see above 5.3.).

10.1 CASE 0[36]

To introduce a few notes about accountancy problems employing end units and internally produced or procured items stored in them, we make, first of all, a test without end units and with some simple recipes in orders, using OrderGenerator (see above 5.1.). In jesframe.scm we have: ticksInATimeUnit 1, useOrderDistiller #f, totalUnitNumber 3, totalMemoryMatrixNumber 0, maxStepNumber 4, maxStepLength 2, useWarehouses #f, useNewses #f.

unitData/unitBasicData.txt contains:

```
unit_#__useWarehouse____prod.phase_#____fixed_costs_____variable_costs
 1         1                 1                10          1
 2         1                 2                10          1
 3         1                 3                10          1
```

revenuePerEachRecipeStep = 21 and revenuePerCostUnit = 0 in jesframe.scm (this value gives a benefit of 10 with fixed and variable costs 10+1).

We press 10 times Next button[37] (at the end the graphs show 9 in the *x* scale).

The concluded order log (file log/ concludedOrderLog.txt) is:

```
Each line contains: final time unit, tick in the final time unit,
                    "recipe name", order layer, order number,
                    starting time unit, tick in the s. time unit,
                    number of steps, "the recipe steps",
                    "the units that have been doing the various steps of the
                    order (-1=step not executed, 'or' sequence)",
                    total final cost of the order,
                    number of direct and indirect steps done,
                    order final evaluation
2, 0, "noName", 0, 2, 1, 0, 1, "1 ", "1 ", 11.0, 1.0, 21.0
3, 0, "noName", 0, 1, 0, 0, 2, "1 1 ", "1 1 ", 22.0, 2.0, 42.0
4, 0, "noName", 0, 3, 2, 0, 2, "2 2 ", "2 2 ", 22.0, 2.0, 42.0
7, 0, "noName", 0, 4, 3, 0, 4, "3 3 1 1 ", "3 3 1 1 ", 44.0, 4.0, 84.0
7, 0, "noName", 0, 7, 6, 0, 1, "3 ", "3 ", 11.0, 1.0, 21.0
8, 0, "noName", 0, 5, 4, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
9, 0, "noName", 0, 9, 8, 0, 1, "2 ", "2 ", 11.0, 1.0, 21.0
```

---

[35] The testCases/development/ subdirectory contains cases used to check the consistency of the code while developing it.

[36] Look at the contents of the folder case0_OrderGenerator/ in testCases/ folder.

[37] Close the program to obtain the output files; otherwise, they are hold in a temporary buffer.

10.1.1. COSTS OF CASE 0

We generate the file Costs/totalInventoryFinancialCosts.txt also when we have no stand alone inventory production to avoid misunderstandings related to any previously generated file produced by a run of the simulation that was using inventories: in our case the file is filled with zeros.

The same choice is made for the file Revenues/dailyStoredComponentValue.txt.

The file Costs/totalDailyCosts.txt reports the sum of fixed and variable daily costs. It is set to 0 at beginning of the day (here, each tick). In this case it contains[38]:

```
+0.0000000000000000e+00
+3.1000000000000000e+01
+3.1000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.3000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
```

The interpretation of the data reported above is the following (rows are numbered from 0):

- row (or time unit) 0: at the beginning of the simulation, the production units are idle, because orders are launched immediately after the execution of the production step of each time unit; if the units are not producing for their warehouses to store inventories, our simulated word starts with the first day[39] orders (see above 6.1.1.); so in row 0 we have to register no costs;
- row 1: an order has been launched at time 0; looking at the log of the finished orders above we know that it is the second order in the list, reporting 0 as starting time unit; its recipe[40] is “1 1”, employing the same production unit (the one able to perform the step 1, i.e., the unit 1, in this case) twice; within this time unit, the production unit 1 accounts fixed and variable costs, for an amount of 11; the other units are idle and account only fixed costs, for a global amount of 20;
- row 2: the second order in the log list is always active, but the order launched at time 1 (reported in the first line of the log list of the finished orders, being this order concluded before the previous one) overpasses it in a FIFO sequence (Fist In First Out) that ignores the fact that the first order was already in production in the same unit and that now its production has been suspended; how all this works: when the first step of the previous order is concluded, the order goes newly to the same unit, but at the end of the waiting list; to avoid this effect we can use the sameStepLifoAssignment option; within this time unit, the production unit 1 accounts fixed and variable costs, for an amount of 11; the other units are idle and account only fixed costs, for a global amount of 20;
- row 3: the third order in the log list, launched at time 2, is now employing production unit 2, able to do a step 2; simultaneously, the first order is doing its second step in production

---

[38] The scientific format is used here for the output because we have no *ex ante* idea about the scale of the results. Anyway, this is the automatic output produced by EZGraph Swarm object that writes the file content.

[39] Or shift or any other denomination of the unit of time we are considering.

[40] This is the intermediate form (see 4.1. above) of the recipe, with each step repeated several times if its execution time exceeds one time unit (second, minute, hour, day, …).

unit 1; within this time unit, the production units 1 and 2 account fixed and variable costs, for a global amount of 22; the other unit is idle and accounts only fixed costs, for an amount of 10;

- etc.

10.1.2. REVENUES OF CASE 0

The file Revenues/dailyRevenues.txt reports the revenues from finished orders (evaluated using revenuePerEachRecipeStep set to 21, with revenuePerCostUnit set to 0). It is set to 0 at the beginning of each day. We can compare the content of this file with that of the log of the concluded orders. In our example the content is:

```
+0.0000000000000000e+00
+0.0000000000000000e+00
+2.1000000000000000e+01
+4.2000000000000000e+01
+4.2000000000000000e+01
+0.0000000000000000e+00
+0.0000000000000000e+00
+1.0500000000000000e+02
+6.3000000000000000e+01
+2.1000000000000000e+01
```

The file Revenues/dailySemimanufacturedOrderRevenues.txt reports the value, evaluated as above, of semi-manufactured orders at a specific time. We can compare the content of the file with the launched and not concluded orders at each tick, as reported in the log file. The content of the file is:

```
+0.0000000000000000e+00
+2.1000000000000000e+01
+2.1000000000000000e+01
+2.1000000000000000e+01
+2.1000000000000000e+01
+6.3000000000000000e+01
+1.2600000000000000e+02
+6.3000000000000000e+01
+4.2000000000000000e+01
+6.3000000000000000e+01
```

10.1.3. BENEFIT OF CASE 0

The file Benefit/benefit.txt reports benefit data from the beginning of the simulation; here it shows the following results:

```
+0.0000000000000000e+00
-1.0000000000000000e+01
-2.0000000000000000e+01
-1.0000000000000000e+01
+0.0000000000000000e+00
+1.0000000000000000e+01
+4.0000000000000000e+01
+5.0000000000000000e+01
+6.0000000000000000e+01
+7.0000000000000000e+01
```

10.2. Case 1[41]

Now we run a second test without end units and with some simple recipes in orders, using OrderDistiller (see above 5.1.). In jesframe.scm we have: useOrderDistiller #t, totalUnitNumber 3, maxStepNumber 4, maxStepLength 2, useWarehouses #f, useNewses #f.

unitData/unitBasicData.txt contains the same data as above in Case 0.

After 10 simulation time units, the concluded order log (file log/ concludedOrderLog.txt) is:

```
Each line contains: final time unit, tick in the final time unit,
                    "recipe name", order layer, order number,
                    starting time unit, tick in the s. time unit,
                    number of steps, "the recipe steps",
                    "the units that have been doing the various steps of the
                    order (-1=step not executed, 'or' sequence)",
                    total final cost of the order,
                    number of direct and indirect steps done,
                    order final evaluation
1, 0, "recipeB", 0, 1, 0, 0, 1, "1 ", "1 ", 11.0, 1.0, 21.0
4, 0, "recipeA", 0, 2, 1, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
5, 0, "recipeA", 0, 3, 1, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
6, 0, "recipeA", 0, 4, 2, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
7, 0, "recipeA", 0, 5, 2, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
8, 0, "recipeA", 0, 6, 3, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
9, 0, "recipeA", 0, 7, 3, 0, 3, "1 2 3 ", "1 2 3 ", 33.0, 3.0, 63.0
```

This log is consistent with the content of the recipeData/recipes.xls file:

```
#  Recipes   ;
   RecipeA  101  1  S  1  2  s  1  3  s  1  ;
   RecipeB  100  1  S  1  ;
```

The simulation follows the schedules:

recipeData/orderStartingSequence.xls file:

```
1   100   *   1   ;
```

recipeData/orderSequence.xls file:

```
1   101   *   2   ;
```

10.2.1. Costs of case 1

The file Costs/totalDailyCosts.txt reports the sum of fixed and variable daily costs. It is set to 0 at beginning of the day. In this case it contains:

---

[41] Look at the contents of the folder Case1_OrderDistiller/ in testCases/ folder.

```
+0.0000000000000000e+00
+3.1000000000000000e+01
+3.1000000000000000e+01
+3.2000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
```

10.2.2. Revenues of case 1

The file Revenues/dailyRevenues.txt reports the revenues from finished orders (evaluated using revenuePerEachRecipeStep set to 21, with revenuePerCostUnit set to 0). It is set to 0 at the beginning of each day. In our example its content is:

```
+0.0000000000000000e+00
+2.1000000000000000e+01
+0.0000000000000000e+00
+0.0000000000000000e+00
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
```

The file Revenues/dailySemimanufacturedOrderRevenues.txt reports the value, evaluated as above, of semi-manufactured orders at a specific time. The content of the file is:

```
+0.0000000000000000e+00
+0.0000000000000000e+00
+2.1000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
+6.3000000000000000e+01
```

10.2.3. Benefit of case 1

The file Benefit/benefit.txt reports benefit data from the beginning of the simulation; here it shows the following results:

```
+0.0000000000000000e+00
-1.0000000000000000e+01
-2.0000000000000000e+01
-1.0000000000000000e+01
+2.0000000000000000e+01
+5.0000000000000000e+01
+8.0000000000000000e+01
+1.1000000000000000e+02
```

```
+1.4000000000000000e+02
+1.7000000000000000e+02
```

10.3. CASE 2[42]

The third test uses end units (only one), with some simple recipes in orders, employing OrderDistiller (see above 5.1.). In jesframe.scm we have: useOrderDistiller #t, totalUnitNumber 3, totalEndUnitNumber 1, maxStepNumber 4, maxStepLength 2, useWarehouses #f, useNewses #f.

unitData/unitBasicData.txt contains the same data as above.

unitData/endUnitList.txt contains (the initial double line has to be read as a single line):

```
end_unit_#;_use_positive_code_for_layer_sensitive_end_unit;_negative_for_un
sensitive;_do_not_duplicate_the_codes,_neither_with_a_different_sign
10
```

After 10 simulation time units, the concluded order log (file log/ concludedOrderLog.txt) is[43]:

```
Each line contains: final time unit, tick in the final time unit,
                    "recipe name", order layer, order number,
                    starting time unit, tick in the s. time unit,
                    number of steps, "the recipe steps",
                    "the units that have been doing the various steps of the
                    order (-1=step not executed, 'or' sequence)",
                    total final cost of the order,
                    number of direct and indirect steps done,
                    order final evaluation
4, 0, "recipeA", 0, 2, 1, 0, 3, "1 2 3 ", "1 2 3 ", 44.0, 4.0, 84.0
6, 0, "recipeA", 0, 4, 2, 0, 3, "1 2 3 ", "1 2 3 ", 44.0, 4.0, 84.0
8, 0, "recipeA", 0, 6, 3, 0, 3, "1 2 3 ", "1 2 3 ", 44.0, 4.0, 84.0
```

This log is consistent with the content of the recipeData/recipes.xls file:

#	Recipes	;													
	RecipeA	101	1	S	1	2	s	1	p	1	10	3	s	1	;
	RecipeB	100	1	S	1	e	10	;							

In this recipe list we have both a p process (procurement, see 2.3. and 3.4. above) and an e key introducing the endUnit number 10.

The simulation follows the schedules:

recipeData/orderStartingSequence.xls file:

1	100	*	1	;

recipeData/orderSequence.xls file:

---

[42] Look at the contents of the folder Case2_OrderDistiller/ in testCases/ folder.

[43] We can tremendously increase the output of our simulated enterprise adding a second unit able to perform the step 1.

```
1     101       *     1    100    *     1       ;
```

10.3.1 COSTS OF CASE 2

The file Costs/totalDailyCosts.txt reports the sum of fixed and variable daily costs. It is set to 0 at beginning of the day. In this case it contains:

```
+0.0000000000000000e+00
+3.1000000000000000e+01
+3.1000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
+3.2000000000000000e+01
```

10.3.2 REVENUES OF CASE 2

The file Revenues/dailyRevenues.txt reports the revenues from finished orders (evaluated using revenuePerEachRecipeStep set to 21, with revenuePerCostUnit set to 0). It is set to 0 at the beginning of each day. In our example its content is:

```
+0.0000000000000000e+00
+0.0000000000000000e+00
+0.0000000000000000e+00
+0.0000000000000000e+00
+8.4000000000000000e+01
+0.0000000000000000e+00
+8.4000000000000000e+01
+0.0000000000000000e+00
+8.4000000000000000e+01
+0.0000000000000000e+00
```

The file Revenues/dailySemimanufacturedOrderRevenues.txt reports the value, evaluated as above, of semi-manufactured orders at a specific date. The content of the file is:

```
+0.0000000000000000e+00
+2.1000000000000000e+01
+4.2000000000000000e+01
+8.4000000000000000e+01
+4.2000000000000000e+01
+8.4000000000000000e+01
+4.2000000000000000e+01
+8.4000000000000000e+01
+4.2000000000000000e+01
+8.4000000000000000e+01
```

10.3.3. BENEFIT OF CASE 2

The file Benefit/benefit.txt reports benefit data from the beginning of the simulation; here it shows the following results:

```
+0.0000000000000000e+00
-1.0000000000000000e+01
```

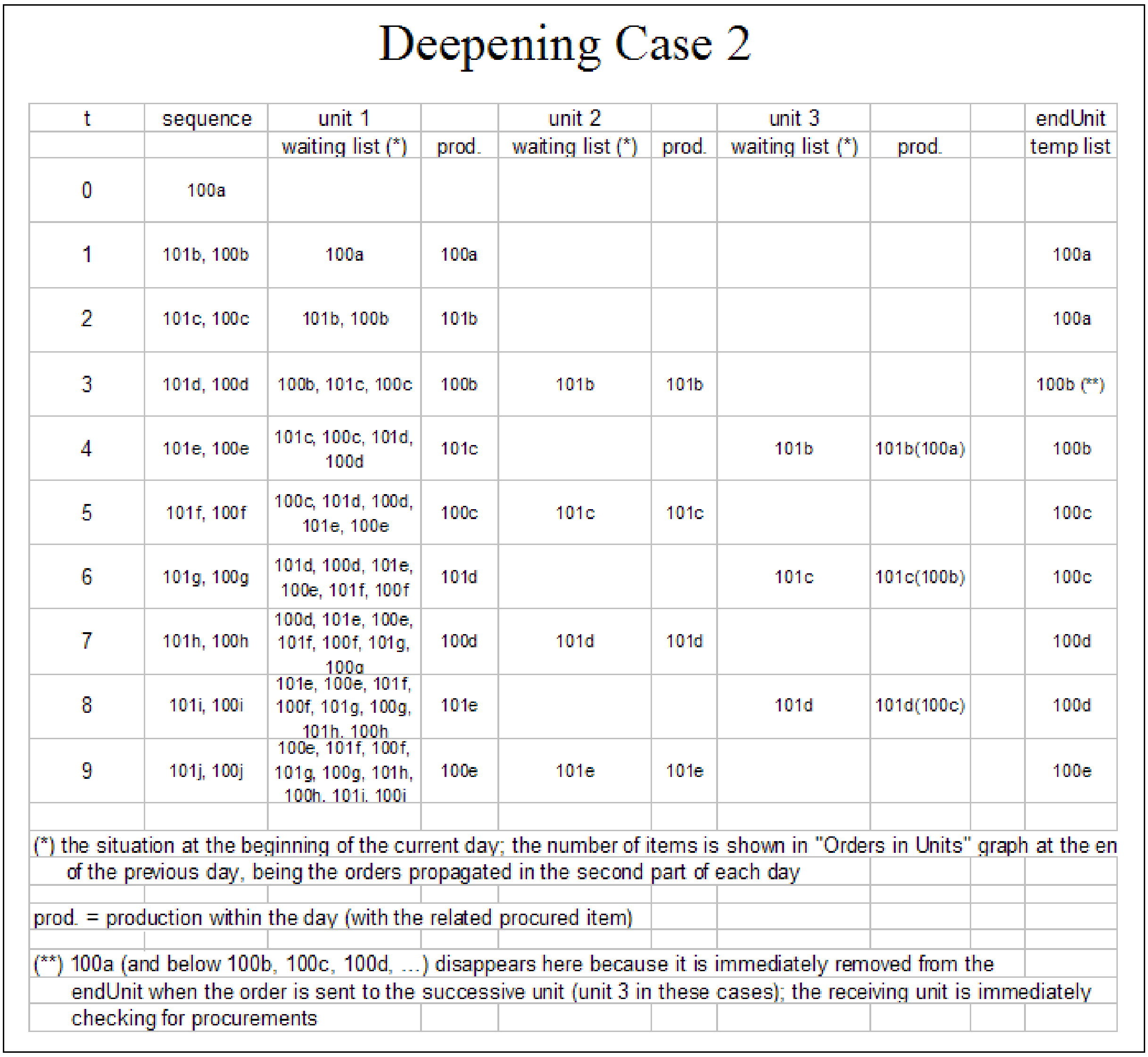
```
-2.0000000000000000e+01
-1.0000000000000000e+01
+0.0000000000000000e+00
+1.0000000000000000e+01
+2.0000000000000000e+01
+3.0000000000000000e+01
+4.0000000000000000e+01
+5.0000000000000000e+01
```

10.3.4. DEEPENING CASE 2

Now we introduce a thorough reconstruction, in Figure 23, of the sequence of the events.

In a parallel way we introduce, in Figure 24, a detailed reconstruction of the accounting operations.

The reader that want to check completely these tables, has all the necessary data above; she can also run the simulation using the file of the `testCases/Case2_OrderDistiller/` folder.



Deepening Case 2

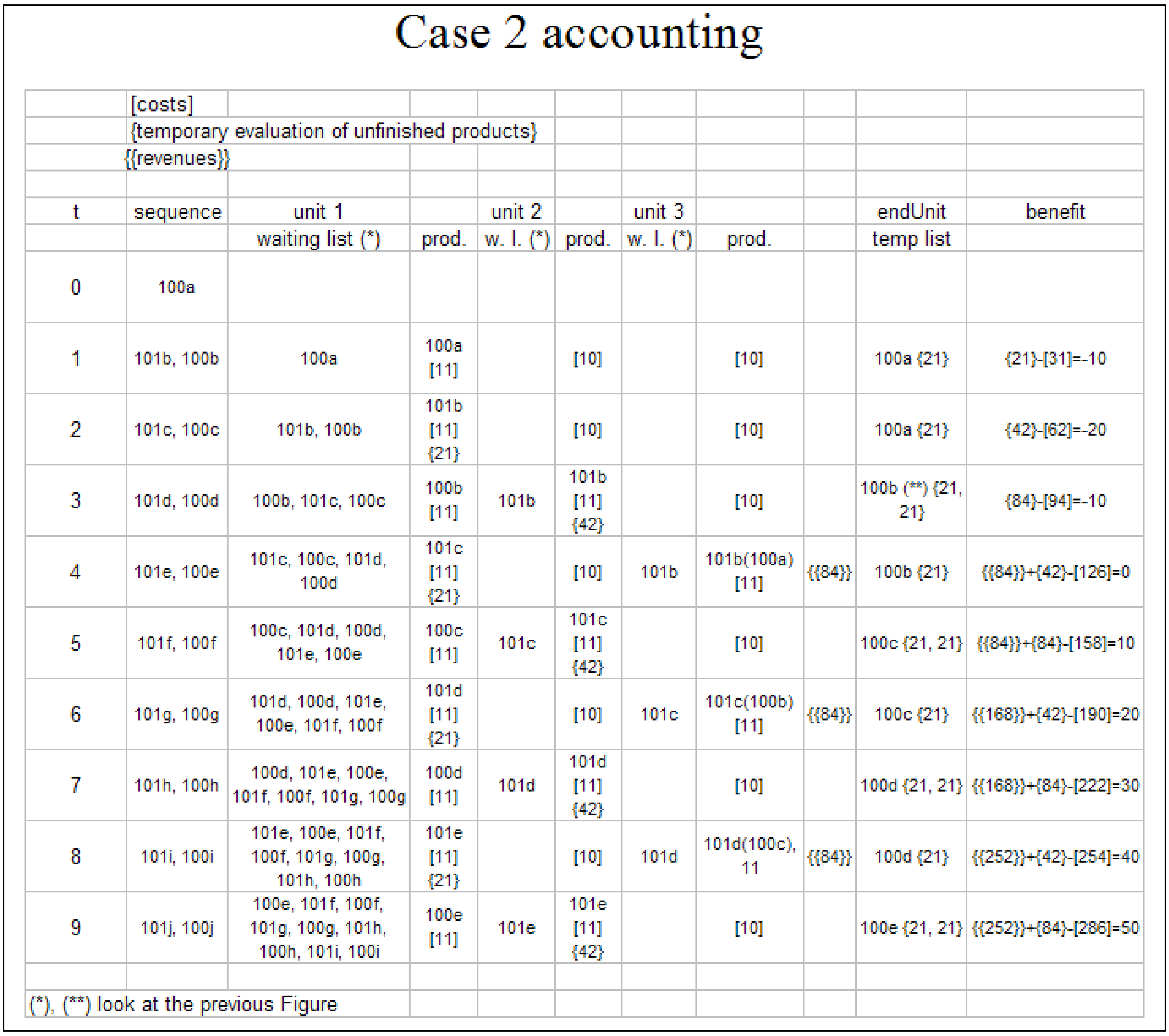
t	sequence	unit 1		unit 2		unit 3			endUnit
		waiting list (*)	prod.	waiting list (*)	prod.	waiting list (*)	prod.		temp list
0	100a								
1	101b, 100b	100a	100a						100a
2	101c, 100c	101b, 100b	101b						100a
3	101d, 100d	100b, 101c, 100c	100b	101b	101b				100b (**)
4	101e, 100e	101c, 100c, 101d, 100d	101c			101b	101b(100a)		100b
5	101f, 100f	100c, 101d, 100d, 101e, 100e	100c	101c	101c				100c
6	101g, 100g	101d, 100d, 101e, 100e, 101f, 100f	101d			101c	101c(100b)		100c
7	101h, 100h	100d, 101e, 100e, 101f, 100f, 101g, 100g	100d	101d	101d				100d
8	101i, 100i	101e, 100e, 101f, 100f, 101g, 100g, 101h, 100h	101e			101d	101d(100c)		100d
9	101j, 100j	100e, 101f, 100f, 101g, 100g, 101h, 100h, 101i, 100i	100e	101e	101e				100e

(*) the situation at the beginning of the current day; the number of items is shown in "Orders in Units" graph at the en of the previous day, being the orders propagated in the second part of each day

prod. = production within the day (with the related procured item)

(**) 100a (and below 100b, 100c, 100d, ...) disappears here because it is immediately removed from the endUnit when the order is sent to the successive unit (unit 3 in these cases); the receiving unit is immediately checking for procurements

Figure 23. The sequence of the events in Case 2.



Case 2 accounting

[costs]
{temporary evaluation of unfinished products}
{{revenues}}

t	sequence	unit 1		unit 2		unit 3			endUnit	benefit
		waiting list (*)	prod.	w. l. (*)	prod.	w. l. (*)	prod.		temp list	
0	100a									
1	101b, 100b	100a	100a [11]		[10]		[10]		100a {21}	{21}-[31]=-10
2	101c, 100c	101b, 100b	101b [11] {21}		[10]		[10]		100a {21}	{42}-[62]=-20
3	101d, 100d	100b, 101c, 100c	100b [11]	101b	101b [11] {42}		[10]		100b (**) {21, 21}	{84}-[94]=-10
4	101e, 100e	101c, 100c, 101d, 100d	101c [11] {21}		[10]	101b	101b(100a) [11]	{{84}}	100b {21}	{{84}}+{42}-[126]=0
5	101f, 100f	100c, 101d, 100d, 101e, 100e	100c [11]	101c	101c [11] {42}		[10]		100c {21, 21}	{{84}}+{84}-[158]=10
6	101g, 100g	101d, 100d, 101e, 100e, 101f, 100f	101d [11] {21}		[10]	101c	101c(100b) [11]	{{84}}	100c {21}	{{168}}+{42}-[190]=20
7	101h, 100h	100d, 101e, 100e, 101f, 100f, 101g, 100g	100d [11]	101d	101d [11] {42}		[10]		100d {21, 21}	{{168}}+{84}-[222]=30
8	101i, 100i	101e, 100e, 101f, 100f, 101g, 100g, 101h, 100h	101e [11] {21}		[10]	101d	101d(100c), 11	{{84}}	100d {21}	{{252}}+{42}-[254]=40
9	101j, 100j	100e, 101f, 100f, 101g, 100g, 101h, 100h, 101i, 100i	100e [11]	101e	101e [11] {42}		[10]		100e {21, 21}	{{252}}+{84}-[286]=50

(*), (**) look at the previous Figure

Figure 24. Case 2 accounting.

10.4. OTHER CASES

In the testCases/ folder we have also Case 2sab[44] and Case 2sb[45], not reported here: 2sab uses stand alone batches (see 2.2.2. above) and 2sb uses sequential batches (see 2.2.1 above).

REFERENCES

GILBERT N., TERNA P. (2000), How to build and use agent-based models in social science. In *Mind & Society*, no. 1, pp. 57-72.

---

[44] Look at the contents of the folder Case2sab_OrderDistiller/ in testCases/ folder.

[45] Look at the contents of the folder Case2sb_OrderDistiller/ in testCases/ folder.