In corso
accountancy with @@@3 (as an appendix)
@@@1 unit criterion
@@@2 accounting
@#@ jESlet

How to use jES program (July 2003, Pietro Terna)

An introduction to an *enterprise simulator*

(related to jesframe-0.9.7.60.tar.gz, *How to* v. 0.1.6.4.8; the figures of this *How to* are also reported in the companion file How_to_use_jES_(figures).ppt[1]

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

We are developing *jES* (the external name of the project: Java Enterprise Simulator), 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 actions - with 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, as is or modified (may be, also operating in never seen situations), with highly practical goals. In the second case, we are interested in theoretical analysis of enterprise creation, behavior, network interaction, with speculative purposes.

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

> (. . .) there are three main way to build a model: 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 world description and representation and, in a consistent way, in our program or, better, in our model.

Our simulated enterprise has both orders to accomplish – each described by a "recipe" that contains the WD (What to Do) side or the world - and production units that perform the different steps of the production process, which represent the DW (which is Doing What) side of the same world.

---

[1] We 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 of virtual enterprise is also used to designate operating as a network of actual firms or of subparts of those firms (see below the reference to ne NIIIP Consortium) and it is compatible too with the use and purposes of jES, but in the (i) side purpsoses.

A third formalism is related to the time sequence of the events (the orders to be executed) that occur in the word that we want reproduce using our simulation tool; this is the WDW formalism: When Doing What.

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

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 clean separation between the order 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 above, 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 tested (with or without *just in time* requirements).

A SIMPLIFIED VIEW

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 are coincident) within the enterprise.

Figure 2 now introduces a more dynamic interpretation of the world we are describing.

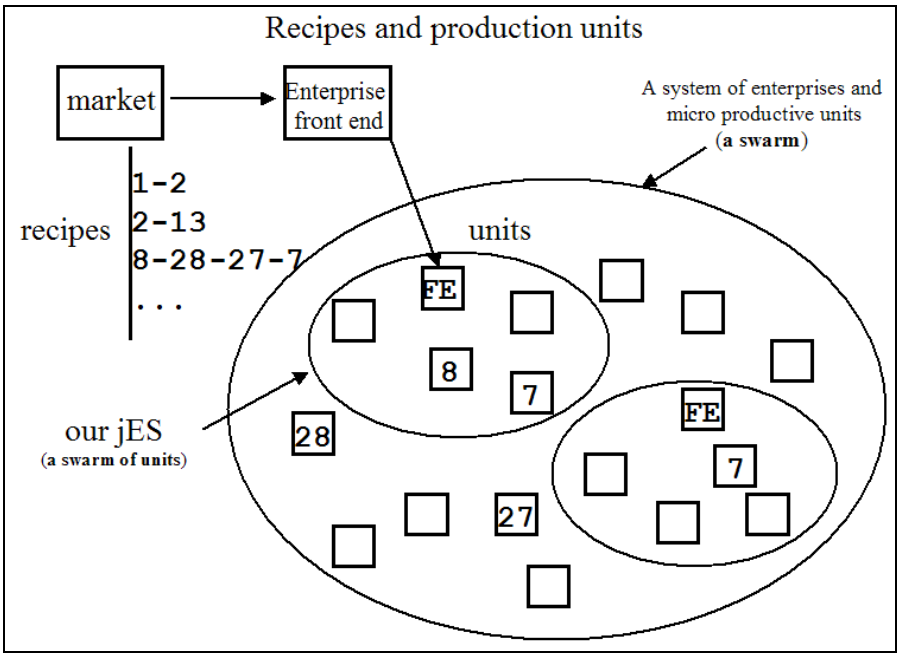


Figure 1. A simplified view of the jES components; recipes are here reported in a simplified way, without time specifications.

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 it asks to all the production units that is 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 unit able to perform task 7. Below we will introduce a set of production unit criterions properly to deal with this kind of problem in our simulation.

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 of having to open a dialog with the front end of the other enterprise. Anyway we have also to take in 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 obtain specific productive results bring directly to the concept of virtual enterprise as an organizational tool: as an example, look at NIIIP project (National Industrial Information Infrastructure Protocols), which as a site at http://niiip01b.npo.org[4].

[3] In the 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 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»."

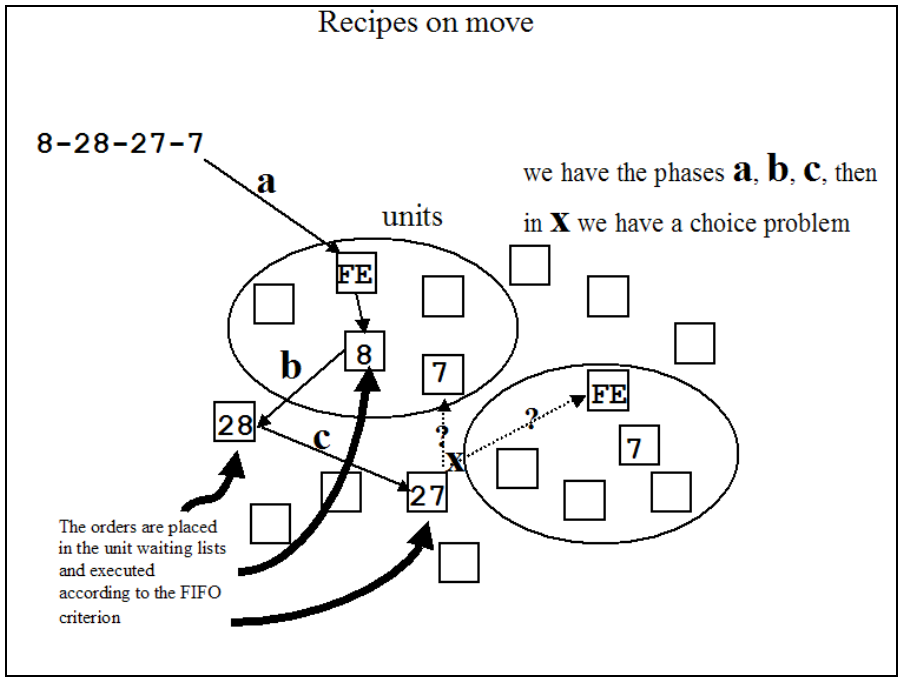


Figure 2. A dynamic view of the jES components; recipes are here reported in a simplified way, without time specifications.

HOW THE MODEL WORKS

With the ***[KF]*** sign we identify below the key features constituting the core of jES; these features are reproduced also in a light versions (jESlet, jES light experimental tool), developed using StarLogo, JAS, Ascape, RePast) both for comparative reasons and to help scholars of agent based simulation techniques in social science to introduce themselves to these different instruments.

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 imagine 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. each existing order (see step 2 about how they appear) makes an inquiry ***[KF]*** into the world to discover if one or more production units can perform its first undone step;
   - each order contains a recipe ***[KF]*** conceived as sequence of steps to be done,
   - with several complex tools useful to better describe the step sequences and the related consequences;

---

[4] In the 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 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»."

2. new orders (each containing its recipe) are launched in production ***[KF]***:
    - following a script describing the temporary sequence of the events to be simulated (see below the use of the orderDistiller object);
    - while testing the program, randomly generating the orders, via the orderGenerator object;
3. if only one production unit makes a positive reply the order is assigned to the waiting list of that production unit ***[KF]***;
    - if does not exist at least a replying unit, the program is stopped in error condition (we have to correct the description of our world); @#@
4. if more than one unit is able to perform the required step, we have to choose one of them;
    - the choice can be made following several unit criterions (see below),
    - but 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 to the introduction of sophisticated optimizations tools such as genetic algorithms and classifier systems;
5. (if the simulation parameter useNewses allows this function, each production unit propagates news about order to be expected in the near future by subsequent production units; this is an attempt to experiment with cooperation and information within an organization; decisions of production of stand alone inventories, described at point 7, can be based also on news informing each unit about future productions);
6. orders stay in the waiting list of the chosen unit up to their specific step is done ***[KF]***;
    - the sequence of the order in the waiting list can be managed to improve the firm performance (this feature is not yet implemented);
7. production units act operating on the orders, one per tick of the clock ***[KF]***; NB each unit acts; this is the first action per each tick of our simulation clock;
    - if an operation requires more than one tick of the clock the order is kept into the unit until all the time is spent;
    - the production can require a setup processes, with related cost and time spent; see below for this feature, that is not yet implemented;
    - the production can be replaced by the use of inventories related to each specific production step,
        - if inventories exist and, must of all, if it is technically possible to store the activity resulting from the production step;
        - in this case, more than one order can be treated in a single tick, if we have room in inventories;
        - inventories are produced and stored when the units are idle (unused), but only if the simulation parameter useWarehouses allows this function;

- (the stand alone production of inventories is discussed below);

8. an order is dropped, after some accounting, when a step of its recipe is done and its recipe does not contain other undone steps ***[KF]***;
   - to drop an order has the meaning of eliminating it from the simulation;
   - in other terms, the related good is sold;
     - the recipe steps can include trade actions;
   - an order can contain a recipe that include at its end a code related to an "end unit" (see below);
     - in this case, the order is related to a component part produced by ourselves or procured externally;
       - after its production (procurement) it is kept in an actual or virtual warehouse represented by and "end unit";
     - note the difference between this kind of production of component parts and the stand alone preparation of inventories of activities related to each single step of a recipe;
9. the sequence continuously goes back to the time phase described in row 1.

Time synchronization is obtained via a usual trick in simulation: at each tick of the simulation clock all the production units make the actions described in row 7 above; only when all this actions are concluded, orders make the operations described in row 8 and – after row 1 is executed – those of row 2.

A CLOSER LOOK TO THE WD SIDE

ORDERS, RECIPES AND LAYERS

Our simulated enterprise has orders to accomplish; the orders are described by the recipes that contains the WD (What to Do) side or the world.

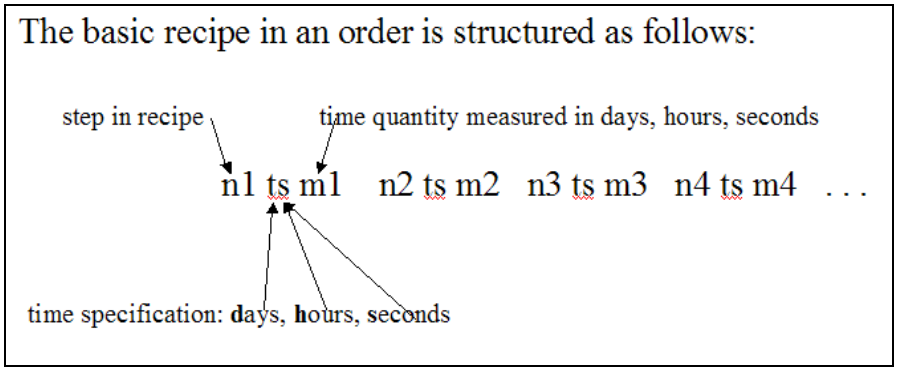


Figure 3. Basic recipe.

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 *n*1 requires *m*1 units of time (days, hours o seconds, following *ts* choice). Time quantities are integer numbers.

Internally, recipes are represented repeating the step for each unit of time it lasts: e.g. 10 s 3 is 10 10 10. Deepening the problem, if we have heterogonous 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 the discussion about units of time; all this subject is not yet fully implemented).

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

The number of layers that we can use explicitly introduced as one of the parameters of the simulation (if it is set to 1, we use no layers). The attribution of each order to a layer is made by the user when she is writing the order sequence of the simulation (see below the use of the orderDistiller object; while testing the program (using the orderGenerator object) layer attributions are made randomly.

We can also imagine to define 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 to be sold or used again in production. The unit able of "doing" this step has unlimited capacity of treating any waiting list dimension, because doing a step means doing nothing.

BATCHES

We have cases in which it is not realistic to think about processes concerning separately single piece (e.g. productions requiring less than one time unit to make a certain step): this kind of occurrences given, a realistic view is that of considering the production as batches of pieces.

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

SEQUENTIAL BATCH PROCESS

A sequential batch process – as reported in Figure 4 – deals simultaneously with a lot of 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.

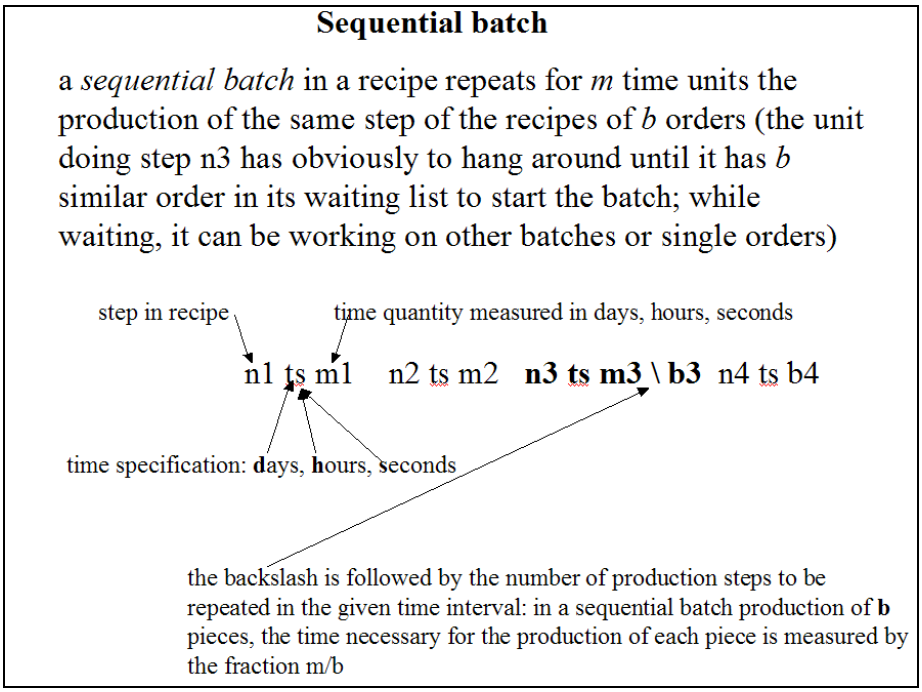


Figure 4. A sequential batch.

in seq. batch ordine uguale se ... (disregard unit number ...)

How it works: SequentialBatchAssembler identifies equal orders composing a sequential batch, signs them as properly belonging to a sequential batch and finally places them together at the beginning of the Unit waitingList.

STAND ALONE BATCH PROCESS

A stand alone batch process, described in Figure 5, is similar to a sequential one (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 in this case external procurements that our enterprise is ordering in batches of large dimensions, requiring a time delay 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, properly due to the time 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 and "end unit" (see below the double procurement process description and the end unit explanation).

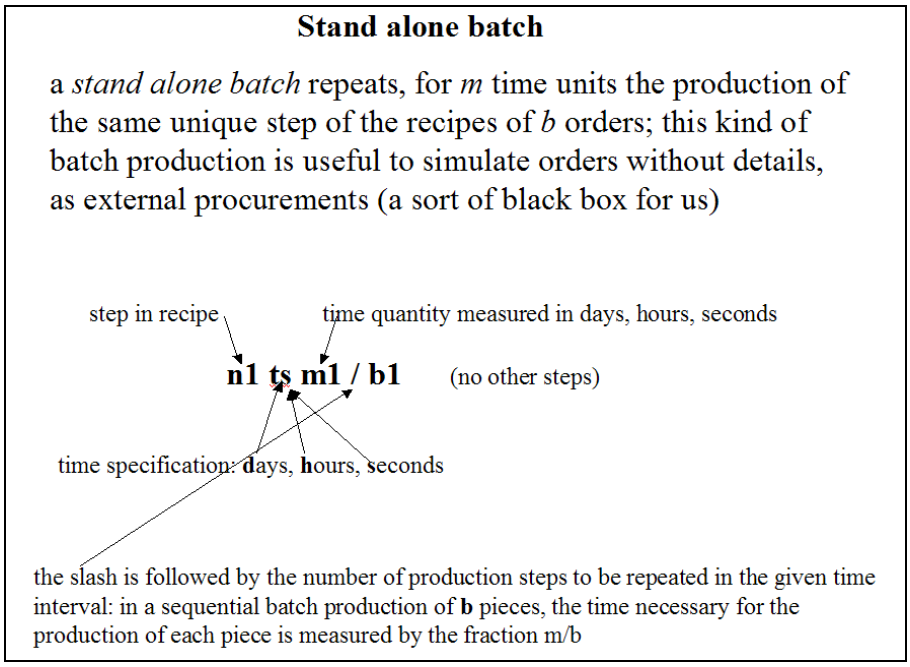


Figure 5. A stand alone batch.

PROCUREMENTS IN THE WD SIDE

Now it is the time of introducing the procurements.

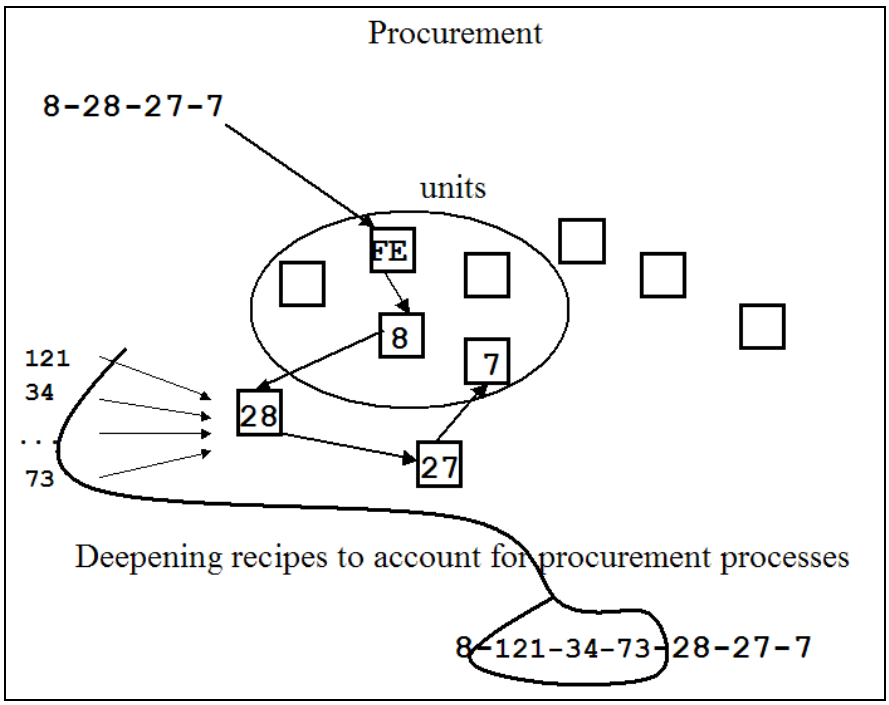


Figure 6. A graphical representation ... ; recipes are here reported in a simplified way, without time specifications.

Procurement are key elements in running the enterprise simulation. In Figure 6 we represent a situation in which step 28, to be executed by the unit signed 28 (remember that in the simplified presentation, recipes are written as a sequence of steps to be executed without information about the time required and that production unit and step numbers are coincident) requires to join 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.

These component must be prepared by specific recipes, like those of Figure 7. Note that those recipes are all concluded by a "e" identifier followed by a numeric code. With "e number" we recall an end unit (see below), i.e. an actual or virtual warehouse where the internally produces parts are to be searched when a recipe tells to a unit to procure them.

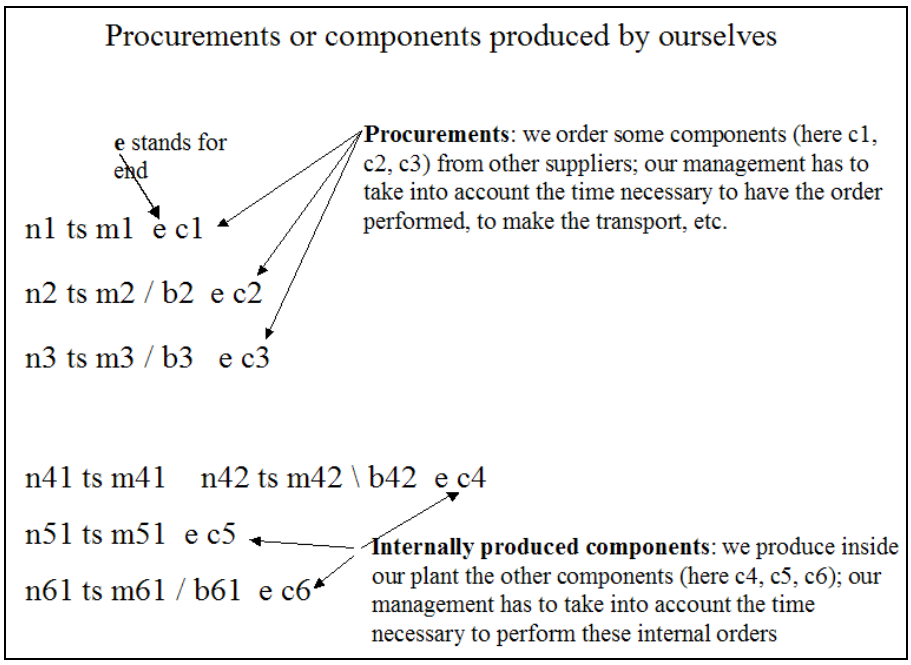


Figure 7. Components (to be procured or internally produced) described in recipes.

In Figure 7 we see procurement order from other supplies, treated as "black boxes", both with a single step recipe (that concluded by *c1* end unit code) and by two stand alone batch (see above) recipes (those concluded by *c2* and *c3* end unit codes); we could decide also the 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 internal produced parts: to first one (concluded with the *c4* end unit code) is described in a detailed way and contains a sequential batch process (see above); those concluded with the end unit codes *c5* and *c6* are similar to those used to describe in a compact way the externally procured productions.

The difference between procurement or internal produced components is anyway merely an *ex post* classification and it is related to our knowledge of the situation: our firm is able to make activities required by the recipes of the internal produced components and we decide of making them internally.

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

A CLOSER LOOK TO THE DW SIDE

SIMPLE PRODUCTION UNITS

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

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

Simple production units, which are able to deal only with one kind of step, are easily described using the file unitData/unitBasicData.txt that 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[5] of the production unit (the 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 use stand alone warehouse (see below, the paragraph about 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 production unit of time (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, *not yet written*) – when we are testing the code – all the recipes are internally generated using the same time basic interval (seconds, hours, days, …): 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, *not yet written*) - to follow a known order sequence applied to a recipe repertoire – may be we have to deal with different time interval used in the recipes: orderDistiller has to convert internally all the time measures to the smallest one: newly, that time measure has to be consistently used in the table of the file unitData/unitBasicData.txt to measure fixed and variable costs.

[5] 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 reported as 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).

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, *not yet written*); their production phase, fixed and variable costs.

COMPLEX PRODUCTION UNITS

Complex production units are able to deal with different production steps; this kind of production unit is identified in the file of Figure 8 with a 0 in the production phase column. We describe them using a spreadsheet file[6], whose contents[7] easily identified using the first worksheet (labeled general_scheme) of the spreadsheet itself (file unitData/unit.xls): the information contained are those of Figure 9.

First of all, the number of phases or production steps the unit is able to perform; then, without any meaning in the order of the various line, the code of each production phase followed by the data about fixed costs and variable costs per unit of time (we account those costs if the production unit is executing the specific phase of activity; hopefully, fixed costs are the same for all phases; anyway, when the unit start is undefined (no production made) we account the fixed costs of the first row. Finally, in each row we have a flag: a 1 value states that in case of absence of activity oir complex production unit will operate a stand alone step of the type referred by the row (according to InventoryRuleMaster.java rules; see below the "Warehouses" paragraph for the interpretation of the stand alone inventory production); a 0 value simply state that the row is not considered to make a stand alone inventory production. Normally only one row is set to 1; if we find more than one row set to 1, the first one is chosen. If no row contains a flag set to 1, no inventories are produced. Remember that the useWarehouse flag of the unitData/unitBasicData.txt do not operate for the rows related to complex production unit (it can be either 0 or 1).

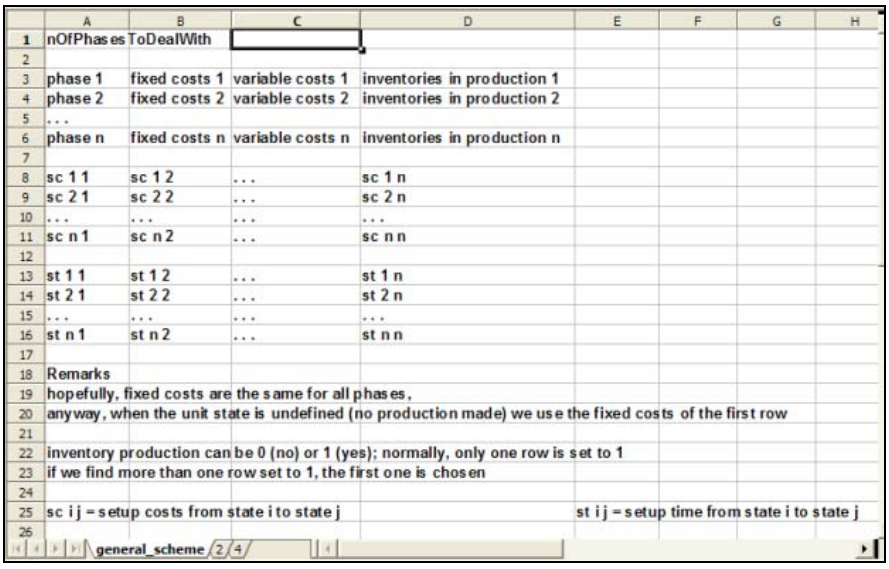
Below the rows containing the fixed and variable costs and the warehouse flag, we are planning to introduce two matrixes related to the production unit setup (not yet implemented)

---

[6] We can produce the spreadsheet both via proprietary code or employing an Open Source one, such as OpenOffice (www.openoffice.org).

[7] 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/)

reporting setup costs sc_{ij} and setup time st_{ij} that we have to sustain to modify to production in the unit from state *i* to state *j*.



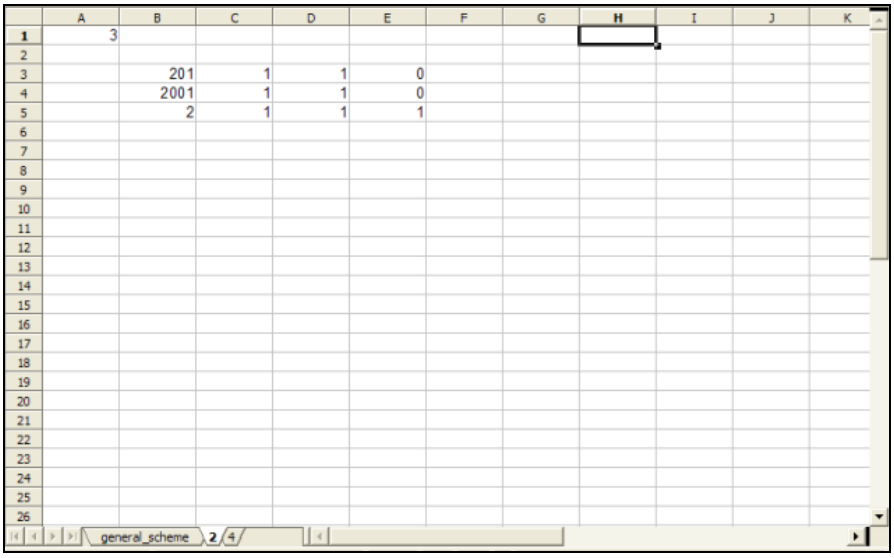
	A	B	C	D	E	F	G	H
1	nOfPhasesToDealWith							
2								
3	phase 1	fixed costs 1	variable costs 1	inventories in production 1				
4	phase 2	fixed costs 2	variable costs 2	inventories in production 2				
5	...							
6	phase n	fixed costs n	variable costs n	inventories in production n				
7								
8	sc 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								

general_scheme 2 4

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

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

That production unit can so operate those three different production steps.



	A	B	C	D	E	F	G	H	I	J	K
1	3										
2											
3		201	1	1	0						
4		2001	1	1	0						
5		2	1	1	1						
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											
19											
20											
21											
22											
23											
24											
25											
26											

general_scheme 2 4

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

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 phases of an activity) and in this case the order is dropped out from the simulation, being concluded and, ma be, the relative merchandise sold. Obviously, some accounting is made to record the effect of the production on the enterprise benefit.

A recipe can be also concluded by an 'e' code followed by a numbed identifying a unit that is not a production node in the simulated enterprise, but a node representing an actual or a virtual (existing only in the mind of a decisional actor, as "We have ordered a bundle of product that will be here just in time to …") store, where we place, really or metaphorical, the result of the recipe.

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.

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).

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

End units are of two kinds of end units: layer sensitive (about layers, see above), identified by a positive code; layer unsensitive end units, identified by a negative code. This difference is relevant in the procurement processes, to determine if a component (procured or produced internally) is differentiated per layer or not when has to be found in and end unit to use it.

PROCUREMENTS IN THE DW SIDE

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, order to a production unit (that able to perform the step which is requiring the procurements: *n2*, in the case of the Figure), to look for end units to find the required components, both internally produced or procured.

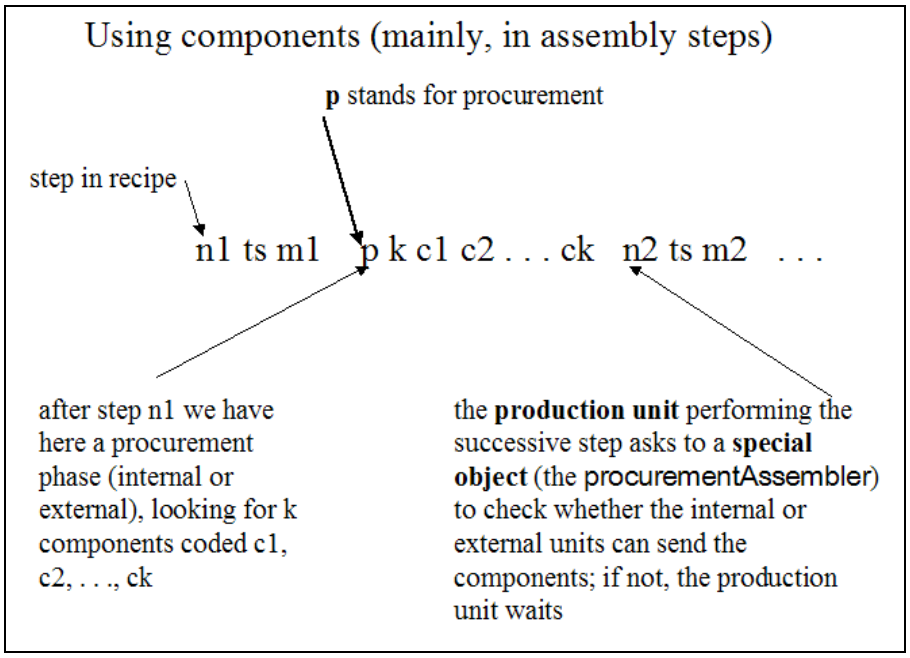


Figure 12. Production units use components (to be procured or internally produced) described in recipes.

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 6 and 13 (if one of them is missing in the file unitData/endUnitList.txt, the simulation is stopped, with an error message).

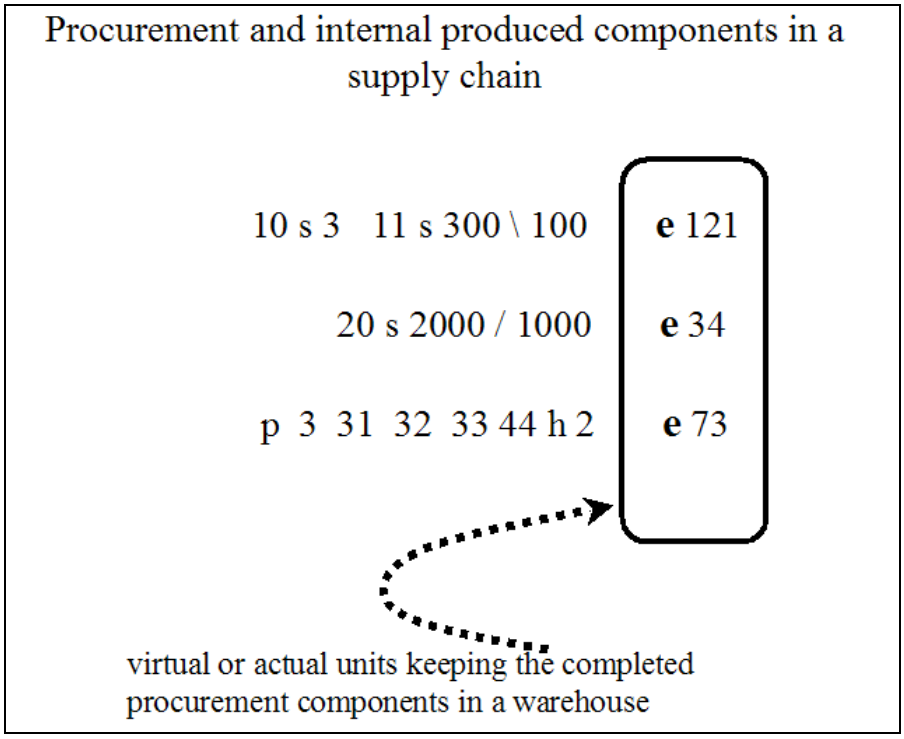


Figure 13. Procurements and internal produced components are held in instances of the EndUnit class.

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. The contents of an end unit can be subdivided by layers (see above): so a specific layers can be empty. If the components part are lacking, the production units which is looking for them waits, pausing its production. (In the future, we will develop a cueing managing feature to overcame the head of the cue in a similar case).

In the example of Figure 13 we have: (a) the component part 121, probably produced internally, since we describe in detail its production: 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; (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'.

CHOOSING WHERE TO GO, WHEN MORE THAN ONE UNIT IS ABLE TO DO A STEP

An important feature in DW side of the simulation is related about decisions.

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 can be made following several unit criterions.

At present, we have the following criterions:

- 1 - the first unit in the unitData/unitBasicData.txt file
- 2 - one of the valid units is chosen in a random way
- 3 - the valid unit with the shortest waiting list is chosen

@@@1

RESOURCES REQUIRED BY ACTIVE PRODUCTION UNITS

To be implemented: production units locking resources for other production units

WAREHOUSES AND STAND ALONE DECISIONS OF INVENTORY PRODUCTION

Warehouses

Decisions newly

Remember the usewarehouse flag in complex unit worksheet …

Remember also to explain the rules contained in InventoryRuleMaster.java … and the meaning and interpretation of a stand alone inventory production …

Stand alone production of inventories is not based upon recipes (such as those of the parts that are necessary in the procurement processes)

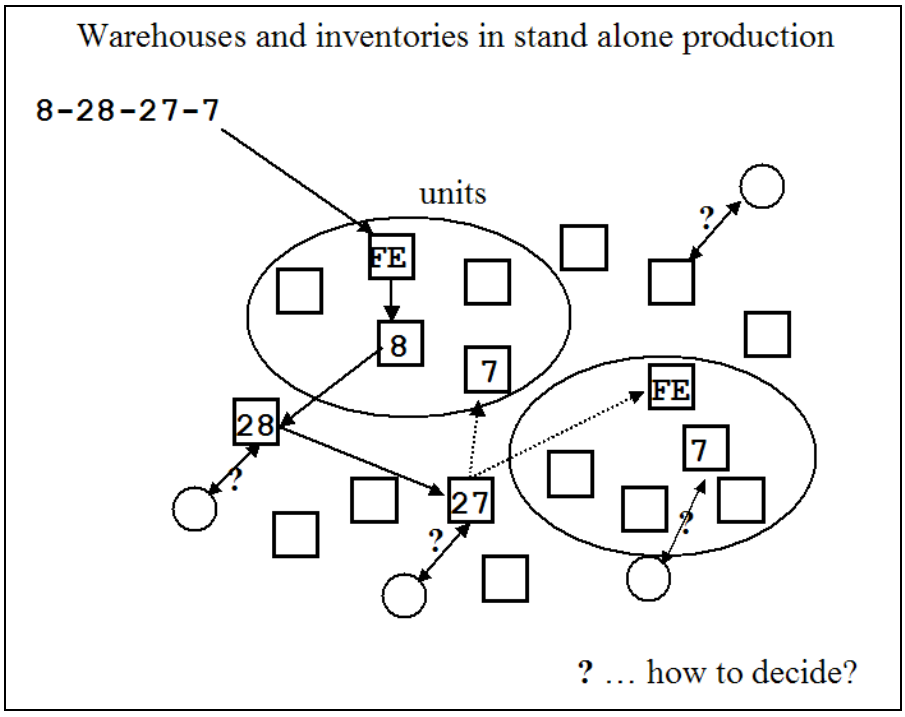


Figure 14. Warehouses and inventories in stand alone production.

NEWS PROPAGATION, INFORMATION AND COOPERATION

News are

Technically a news is an objects, so are dealing with “newses”.

Decisions newly, in the field of elementary knowledge management …

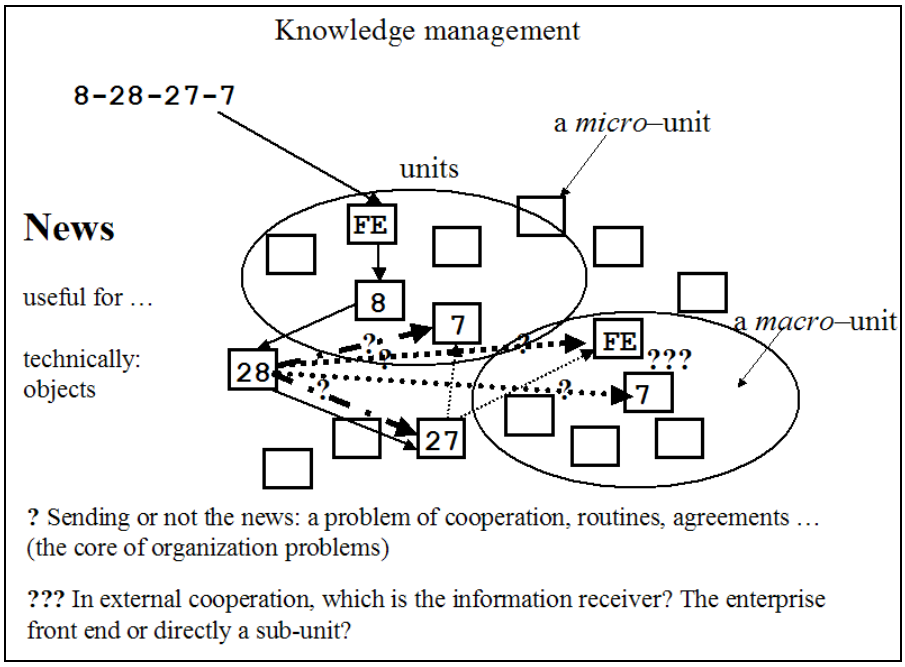


Figure 15. News and elementary knowledge manament.

appunto per 0.32 con sameStepLifoAssignment and assignEqualStepsToSameUnit: per aggirare il fatto che un `101 s 5 101 s 5` sarebbe trattato come `101 s 10` alla stessa unit, se assignEqualStepsToSameUnit è true, una soluzione è `101 s 5 1010 s 0 101 s 5`, in cui 1010 è una unit fittizia

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

If maxTickInAUnit is set to a positive value orders waiting in a unit for more than maxTickInAUnit the order is dropped and disappears from the simulation.

NEWLY BACK TO THE WD SIDE

External (more human readable) and intermediate format of recipes (the internal one, apparently poor in details, can be examined looking at the comments in Order.java file); anyway we write recipes in external code; the translation mechanism from external to intermediate code is contained in OrderDistiller class; from intermediate to internal, in Order class

OR PROCESSES IN WD SIDE

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 to choose into the or? We have to look at the orCriterion variable, which is set either via the probe of the model or into the jesframe.scm file.

If orCriterion is:
== 0, all branches are executed in sequence (useful mainly 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 different production subprocesses;
== 4, we choose the branch whose first step has the shortest waiting list;
== 5, we use the result of a computational step to choose the branch (see below, the paragraphs "Computational capabilities and memory matrixes" and "Computational capabilities and 'OR' sequences").

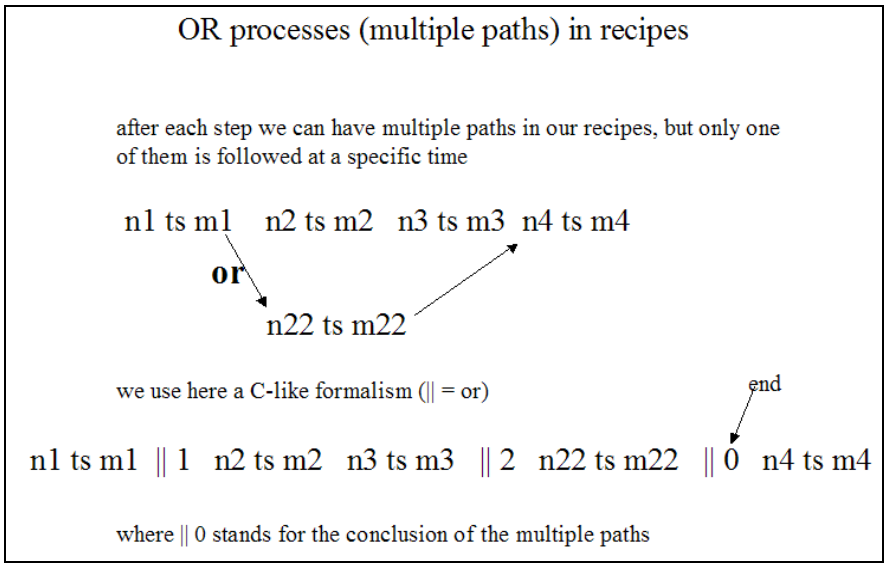


Figure 16. An 'or' process, with its branches (|| 1 and || 2)

An example of or sequence is the following, containing also a procurement process (see above; ... *not yet written*) in I 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 opening two branches of the of the 'or' sequence and || 0 concludes them; in the first branch we can identify the simple procurement sequence 'p 1 101 10 s 1'.

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

AND PROCESSES IN WD SIDE

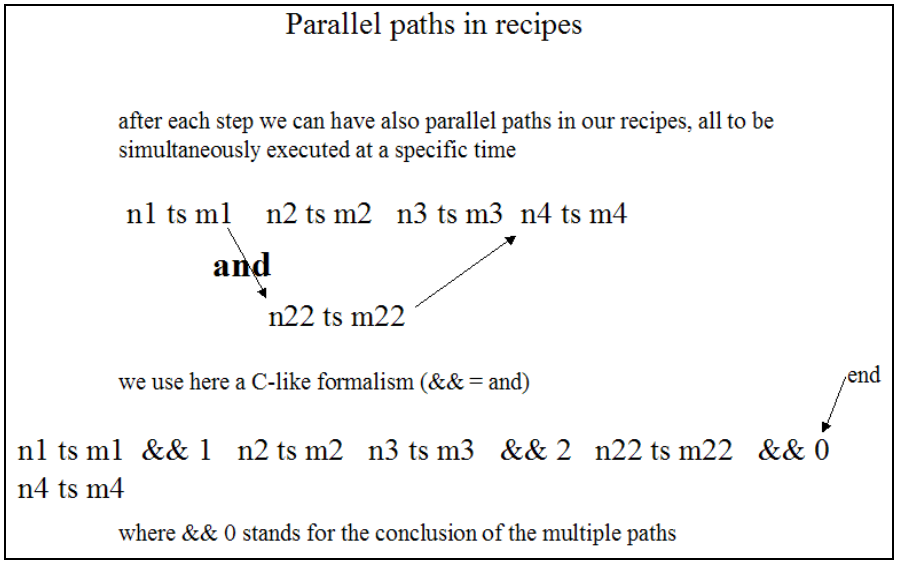


Figure 17. An 'and' process, with its branches (&& 1 and && 2)

AND is also not yet implemented; that described in Figure 17 is an asynchronous AND, as both the braches of the AND sequence have to be executed, but independently as regard to time. We can also imagine a synchronous AND process, where the to branches have to be executed together.

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 Java language, as we have to modify[8] the ComputationalAssembler.java file (which inherits its default methods from the class ComputationalAssemblerBasic). The goal of the computational capabilities is that of dealing with forecasting, evaluations, auctions to chose procurements, …

Computations use data contained in memory matrixes created following both the totalMemoryMatrixNumber of the model probe (this parameter, stating how many matrixes we are creating, can also be set via the jesframe.scm file) and the contents of the file unitData/memoryMatrixes.txt shown in Figure 18. Memory matrixes use layers in a

[8] We have not to modify the basic file (ComputationalAssemblerBasic.java), which is included in the src/ folder. Instead, we have to copy in the main folder of the program, from src/, the file ComputationalAssembler.java.
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 che ckingComputationsAndFreeingOrders() method of ComputationalAssembler class (which inherits it form its parent class): the tick used to convert the numerical code of the computational steps into a recognized method reference is based upon 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);
```

completely automated way; we can prevent them from using layers setting their number as negative in each specific declaration into the file unitData/memoryMatrixes.txt. In the example reported here, the second matrix (numbered 1, being 0 the number of the first one) in insensitive to layers

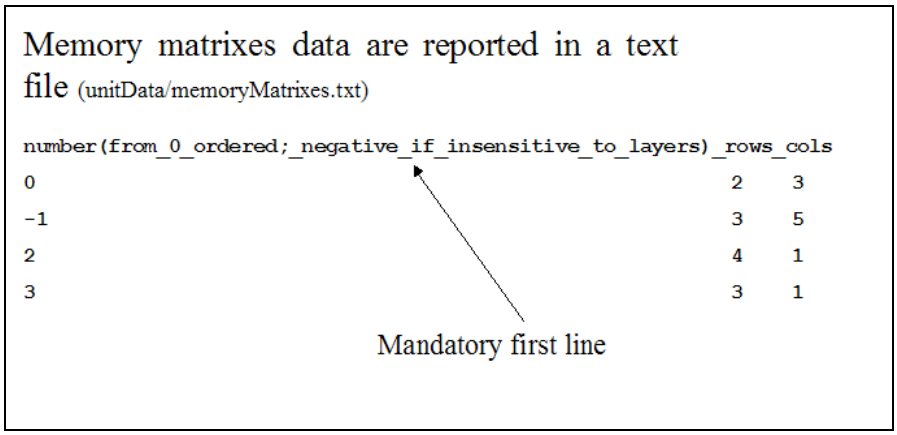


Figure 18. Memory matrixes declarations.

Examples of recipes containing computational steps as reported in Figure 19; obviously, to understand the meaning and the behaviour 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.

As seen above (… *not yet written*), 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. To see the internal code (apparently poor in details) you can have a look to the comment lines in Order.java file. Code number of computational steps are accepted in the range 1001-1999.

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

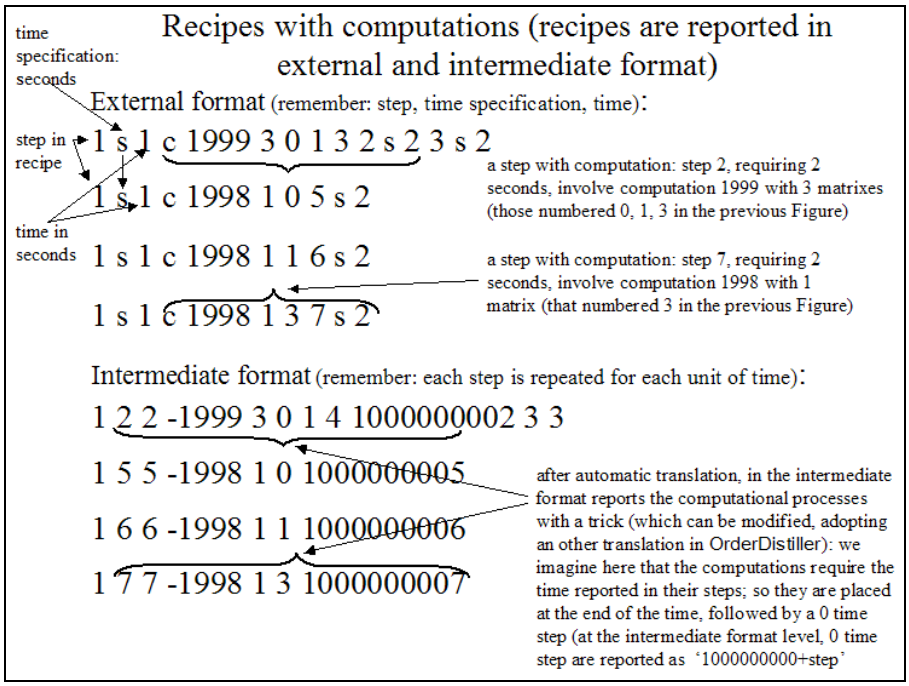


Figure 19. The format of the computational processes.

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

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

The Java Swarm codes, extracted from ComputationalAssembler.java and reported in Figure 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 recipe '1 s 1 c 1998 1 0 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 recipe '1 s 1 c 1998 1 0 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 those operation made upon mm0 matrix (but we can use any name) related to the actual matrix via the getMemoryMatrixAddress method; the setValue method set the 1.0 value at (0,0). If the matrix is insensitive to layers, the layer value set in this method is disregarded. Finally, the computational step is set as 'done'[9].

[9] If the Java code related to a computational method does not set 'done' boolean variable to 'true' the order is not freed and does not proceed to its successive recipe steps; the computational step will be repeated in any simulation cycle, until 'done' variable becomes 'true'.

The Java Swarm code used by the recipes with computation of this example

```
/** 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 Swarm code … (simplified eliminating a control statement related to the consistence of the declared number of matrixes with the internal one).

When an order with 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, matrix 0, 1 and 3 are interested by a check operation to verify if positions (0,0) are empty at the proper layer; if not empty, the 'c 1999' set those positions (at those layers) empty and finally set as 'done'[10] the computational step. Into the code of this example, matrixes mm0, mm1 and mm2 are linked to actual matrixes 0, 1, 3 (the internal name are completely free).

The effect of those four recipes (OrderGenerator, while testing the program, if totalEndUnitNumber > 0, launches those recipes at random) is the following: the recipe containing the code 'c 1999' cannot proceed in step 2 if does not exist the effects of one of each of the recipes containing codes 'c 1998' (produced when those recipes are executed at least at step 5 or 6 or 7 deepness). When the recipe containing the code 'c 1999' finally proceeds to its successive step, the effects or the "used" recipes is eliminated and must be reviewed by other similar orders.

[10] See previous note.

The Java Swarm code used by the recipes with computation of this example

```
/** 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 Swarm code … (simplified eliminating a control statement related to the consistence of the declared number of matrixes with the internal one.

Method accepted by MemoryMatix instances are setValue, getValue, setEmpty, getEmpty (returning true or false).

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 situation; obviously, to set not empty a position of a matrix, we simply put a value in it; getEmpty returns 'true' if no value is found, other wise it return '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 on the current terminal the content of the matrix; the empty position of the matrix are reported as not available (NA).

COMPUTATIONAL CAPABILITIES AND 'OR' SEQUENCES

If orCriterion == 5 (se above "OR processes in WD side") computational results are also useful to chose 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 orMemoryMatrix in the probe of the model or in the file jesframe.scm; the matrix may be sensitive or insensitive to layers. Range of the branch number: from 1 to the number of branches.

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

A computational sequence can be included in an 'or' branch with a great flexibility of computational processes[11].

RUNNING A SIMULATION: USING THE ORDERGENERATOR OR THE ORDERDISTILLER

Explain the toe order mechanisms …

We have to introduce now the two different situations in which we can run a simulation: (i) ; (ii) [riusare qui con maggiore dettaglio quanto sopra detto con … The unit of time must be the smallest used in the whole recipe set. If we use the internal orderGenerator – 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 – all the recipes are internally generated using the same time basic interval (seconds, hours, days, …): 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 - to follow a known order sequence applied to a recipe repertoire – may be we have to deal with different time interval used in the recipes: orderDistiller has to convert internally all the time measures to the smallest one: newly, that time measure has to be consistently used in the table of the file unitData/unitBasicData.txt to measure fixed and variable costs.]

We have to develop an intelligent version of the distiller, able to deal both with time scale changes and with changes in the unit used to measure quantities in recipes. Recipes - as reported in the database file recipeData/recipes.xls - contain references to time about the length of each steps and of each batch process, both sequential or stand alone; if the time unit changes, because we account it by a hundred of seconds or a thousand of seconds or …, those length must be rescaled. 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; those quantities have to be rescaled. Beside this, if we change time unit and quantity unit, also the contents of the file recipeData/orderSequence.xls has to be reinterpreted, about both the number of orders to be launched and the time steps to be considered to launch those orders.

At present all changes have to made by hand modifying the contents of the files recipeData/recipes.xls and recipeData/orderSequence.xls. It will be probably impossible to fully automate those operations, but some step in this direction is certainly possible.

---

[11] This aspect is strategic for the development of jES with the capabilities of simulating both the financial side of the enterprise and the enterprise information system.

We have to introduce the possibility of defining idle time cycles, to simulate pauses in the production process; within idle time cycles all production units or a part of them are stopped. May be the better solution is to place this kind on information into the file recipeData/orderSequence.xls.

ACCOUNTING

jES is capable of automated accounting of production units activities and of order accomplishment. We have so a two sides accounting, with some differences. 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; idle production unit do not account for variable costs, but their fixed costs are not charged to any order, so we can have (correct) differences on the two sides. A unit producing inventories in the stand alone way accounts also variable costs; when the step an order is accomplished using inventories, the related costs are charged to the order. Upon inventories we charge a financial cost, measured by an interest rate, introduced as a parameter of the simulation. The same as to be done for the components in the end units (not yet implemented).

Fixed costs are accounted also by a units doing nothing, regardless their content; variable costs are accounted by units when operating, newly regardless their content.

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

The inputs and outputs about costs are the following (as reported into the costs/readmeCosts file):

INPUT (from CostParameters/)

1) fixedCosts.txt - the fixed cost amount for each time tick in each unit (to be read as a sequence in the same order of the unit numbers); these costs are independent from activity (they have to be considered as sunk costs);
2) variableCosts.txt - the variable cost amount for each time tick in each unit (to be read as a sequence in the same order of the unit numbers); these costs run only in case of activity.

OUTPUT (to Costs/)

1) totalDailyCosts.txt - the sum of fixed and variable daily costs. It is set to 0 at beginning of the day;
2) totalCosts.txt - the sum of totalDailyCost. It is set to 0 only at the beginning of the simulation;
3) finishedOrderCosts.txt - the final cost of the orders made from the beginning of the simulation
4) dailySemimanufacturedOrderCosts.txt - the costs of the orders in production at a specific day;
5) totalInventoryFinancialCost.txt - the financial cost of inventories from the beginning of the simulation.

A few remarks about costs.

I - The costs from the production unit and the order sides

That introduced above is the cost view from the production unit side; jesframe makes accounting also within each order, but no direct access to those data is at present implemented

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 how_to_use_jES.pdf file).

From the production unit side, the inventory costs are accounted as 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 inventories production is transferred to order production costs.

II - Internally produced or externally procured components and end units

When a recipe describing a 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, from the specific sub-order or component side.

@@@2

WARNING: at present the costs are not transferred into the orders collecting the internally produced or externally procured components.

Revenues are accounted for finished orders as they would be sold (anyway, we can include the trade step in our recipes), following:

> value = [# of tick (also of the procured items, if it is a final value)] * revenuePerEachRecipeStep + [costs (also of the procured items, if it is a final value)] * revenuePerCostUnit

If revenuePerEachRecipeStep=0 we use the second criterium and viceversa; revenuePerCostUnit, if used, is normally about (1 + markup per unit)

We have also to take in account inventories (stand alone produces inventories using warehouses), using the inventoryEvaluationCriterion (three cases of warehouses evaluation: 1=variable costs, 2=fixed+variable costs, 3=value, as above)

The evalution of the inventory financial costs is obtained applying the financial rate to the quantity of the inventories, with a value established with the criteria above, but with case 3 used as case 2, because it would be a non sense to apply financial cost to a virtual revenue.

The inputs and the outputs about costs are the following (as reported into the !!! file

FILE …

NB - Error of missing evaluation of the translation of the costs in for components stored in end unit.

For procured parts we can charge the unit (and so the products obtained) both for fixed and variable costs or better only with variable costs for the global amount: if it is the case of a pure procurement, if the related production unit does not works, for our enterprise we account no costs.

we do not make accounting about fixed costs in the first day (time unit: day, shift, ...) [12] unless we use warehouses with the immediate possibility of producing inventories; otherwise, being order launched after the production step of each day, in day 0 simply we have nothing in our world

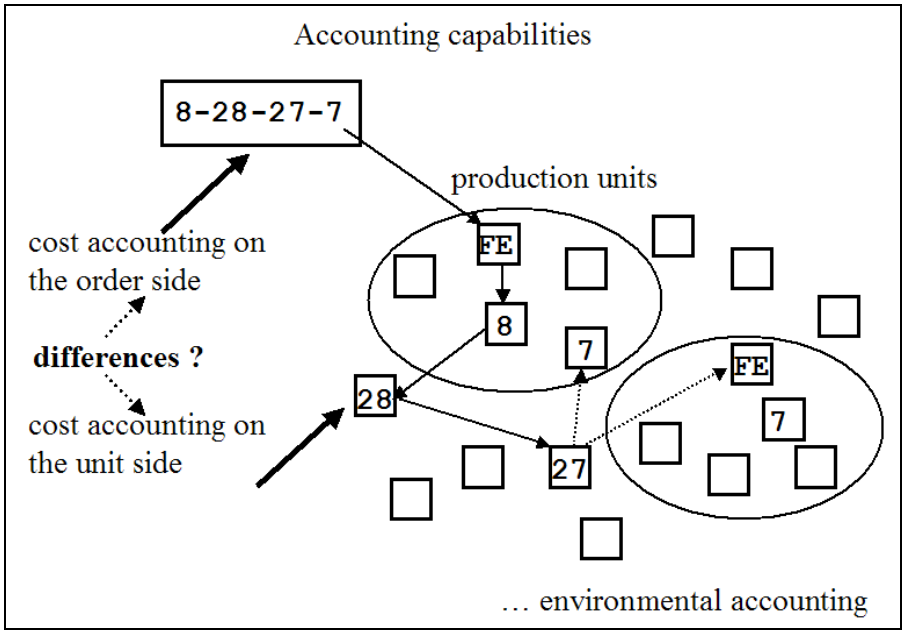


Figure 22. Accounting ….

Accounting and information systems (the last, via computational objects?) …

Environmental accounting …

Recipes without accounting (e.g. a forecasting activity); [if the recipe is externally activated if has 0 costs and so 0 revenue; if it is internal …] pensare a BasicNet

Mettere in Arial tutti i riferimenti a file, metodi, variabili, oggetti che si trovano nel testo

NB NB cambiare l'ordine dei parametri nella probe del model mettendo in fondo quelli che intessano solo le fasi di test NB spiegare l'uso del distiller di B&B&B rispondendo a "ma come si passano le ricette"

[12] More exactly, it starts with the first tick of the first day or shift or any other denomination of the unit of time we are considering.

Sviluppare il Distiller; istruzioni per l'uso QUI e in un file a sé, how to use distiller o qualcosa del genere, con un tar.gz dentro al tar.gz generale

Con il distiller preparare un semplice esempio che poi si complica usando le varie possibilità

Presentare le probe dell'observer e del model e collegare a jesframe.scm

NB How to get jES

Time=s/d/h or multiples of s/h/d: (i) in orderGenerator 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 ratio among the three definitions (s/h/d).

Commentare il README, tutti gli scripts (Windows/Linux) e i contenuti delle directory, ricordare runBig

Esplorare changelog.txt per scoprire cose da commentare nell'How to

Indicare le cose da fare riportate nel TO DO, tra cui la simulazione del sistema informativo appoggiata alla simulazione dell'impresa.

Nel mio sito, link a jes, dove devo mettere i file tar.gz, l'How to anche a parte, il ppt con le figure (e indicarlo sulla prima pagina del ppt)

Non dimenticare:

layer usato *ma non definito* (è così??)

criteri or

rami or con procurement

il meccanismo degli oggetti "appesi" alle ricette

numerare e paragrafi e sostituire tutti i see above e below con indicazione secche di paragrafi.

SIMULATION PARAMETERS

Mettere qui tutti i parametri delle sonde e del file jesframe.scm

How to obtain jES

You can look to 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.

JES is distributed under the Open Source Academic Free License (?) see http://www.opensource.org/licenses/academic.php from http://www.opensource.org/licenses/.

Future improvements

Improve the explicit presence of decisional agents, using existing RuleMaster as their reference and introducing RuleMaster, such as UnitCriterionRuleMaster, NewRecipeRuleMaster, NewSubRecipeRuleMaster, NewenterpriseRuleMaster (vedere punto su explicit introduction of agents in jES.doc)

In Book Antiqua via via segnare le cose che farò come aggiunte???

An appendix about accountancy, with a few examples

- annotare quanto fatto sull'accounting in readmeCosts e quindi riportare in How to; anche in Revenues/readmeRevenues si deve trattare il problema ??

- In to do: Unit, line 1367, error in evaluating financial costs (the interest rate is applied to a physical quantity)

All the data files which are necessary to run the test introduced below can be found in the testCases directory and used copying them (also a whole directory, when necessary) into the main directory of jES. As an example …

(The development subdirectory contains cases used to check the consistence of the code while developing it).

A few notes about accountancy problem having end units and internal produced or procured stored in them.

Some test without end units and some simple recipes, with jesframe.scm reporting: 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 in jesframe.scm (this value[13] gives a benefit of 10 with fixed and variable costs 10+1).

[13] INTRODURRE UN FATTORE MOLTIPLICATIVO DEI COSTI, PER TENERE CONTO DI COSTI NON UGUALI PER STEP??

CASE 0[14]

This test is run without the presence of the application OrderDistiller.java class in the main directory, so the effect of useOrderDistiller #t is only seemingly true and the order are randomly generated by the internal OrderGenerator.java class.

We press 10 times Next button (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 cost of the order]; multiplicity
2 0 noName 0 2 1 0 (1) 1 { 1 } [11.0] 1
3 0 noName 0 1 0 0 (2) 1 1 { 1 1 } [22.0] 1
4 0 noName 0 3 2 0 (2) 2 2 { 2 2 } [22.0] 1
7 0 noName 0 4 3 0 (4) 3 3 1 1 { 3 3 1 1 } [44.0] 1
7 0 noName 0 7 6 0 (1) 3 { 3 } [11.0] 1
8 0 noName 0 5 4 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
9 0 noName 0 9 8 0 (1) 2 { 2 } [11.0] 1
```

COSTS

We generate the file Costs/totalInventoryFinancialCosts.txt also when we have no stand alone inventory production, in that case filled with zeros, to avoid misunderstandings related to a previously generated file produced by a run of the simulation that was using inventories. 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. In this case it contains[15]:

```
+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 cost data reported above is the following (rows are numbered from 0 to 9):

- row (or time unit[16]) 0: at the beginning of the simulation, the production units are idle, because orders are launched immediately after the execution of the production

[14] Look at the file jesframe.scm.Case0_OrderGenerator and to the content of the directory unitData.case0_OrderGenerator in testCases directory.

[15] The scientific format is used here for the output as we have no *ex ante* idea about the scale of the results. Anyway, this is the automatic output produced by EZGraph Swarm object we writing on a file.

step of each time unit; if the units are not producing for their warehouses, storing inventories, "our simulated word starts with the first day orders[17]"; 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[18] is "1 1", employing the same unit (that able to perform the step 1, i.e. that #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) overpass 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, which is reported in the jesframe.scm file or is accessible from the model probe (see above, *not yet written*); in that case
- @@@3

REVENUES

The file Revenues/dailyRevenues.txt reports the revenues from finished orders. It is set to 0 at the beginning of each day. In our example its 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 of semimanufactured orders at a specific date (evaluated using revenuePerEachRecipeStep; in this case set to 21). The content of the file is:

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

[16] A time unit here is a day, a shift, … not the base unit of the time as intended introducing recipes (seconds, minutes, hours, days, …); a day, a shifts etc. can be subdivided in any number of ticks of a programmable clock, with the parameter ticksInATimeUnit; in this example ticksInATimeUnit = 1.

[17] More exactly, it starts with the first tick of the first day or shift or any other denomination of the unit of time we are considering.

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

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

BENEFIT

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
```

Some test without end units and some simple recipes, with jesframe.scm reporting: useOrderDistiller #t, totalUnitNumber 3, maxStepNumber 4, maxStepLength 2, useWarehouses #f, useNewses #f.

A problem: what if those parameters are not consistent with the unit and recipe files when we are using the OrderDistiller?

Case 1[19] (reported also into the file jSE Case1 Case2.xls)

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

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 cost of the order]; multiplicity
1 0 recipeB 0 1 0 0 (1) 1 { 1 } [11.0] 1
4 0 recipeA 0 2 1 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
5 0 recipeA 0 3 1 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
6 0 recipeA 0 4 2 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
7 0 recipeA 0 5 2 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
8 0 recipeA 0 6 3 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
```

[19] Look at the file jesframe.scm.Case1_OrderDistiller and to the content of the directories unitData.Case1_OrderDistiller and recipeData.Case1_OrderDistiller in testCases directory.

```
9 0 recipeA 0 7 3 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
```

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 is following the schedules:

recipeData/orderStartingSequence.xls file:

```
1 100 * 1 ;
```

recipeData/orderSequence.xls file:

```
1 101 * 2 ;
```

COSTS

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
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
+3.3000000000000000e+01
```

REVENUES

The file Revenues/dailyRevenues.txt reports the revenues from finished orders. 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 of semimanufactured orders at a specific date (evaluated using revenuePerEachRecipeStep; in this case set to 21). 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
```

BENEFIT

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
```

Case 2[20] (reported into the file jSE Case1 Case2.xls); in the testCases directory we have also Case 2b[21] data and recipes (this case is not reported here; it used nested procurement, to check the recursive application of routines when dealing with orders)

Test with one end unit and some simple recipes, with jesframe.scm reporting: 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 whole 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
```

[20] Look at the file jesframe.scm.Case2_OrderDistiller and to the content of the directories unitData.Case2_OrderDistiller and recipeData.Case2_OrderDistiller in testCases directory.

[21] Look at the file jesframe.scm.Case2b_OrderDistiller and to the content of the directories unitData.Case2b_OrderDistiller and recipeData.Case2b_OrderDistiller in testCases directory.

```
10
```

The concluded order log (file log/ concludedOrderLog.txt) is[22]:

```
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 cost of the order]; multiplicity
4 0 recipeA 0 2 1 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
6 0 recipeA 0 4 2 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
8 0 recipeA 0 6 3 0 (3) 1 2 3 { 1 2 3 } [33.0] 1
```

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 and an ‘e’ key introducing the endUnit number 10.

The simulation is following the schedules:

recipeData/orderStartingSequence.xls file:

1	100	*	1	;

recipeData/orderSequence.xls file:

1	101	*	1	100	*	1	;

COSTS

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
```

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

REVENUES

The file Revenues/dailyRevenues.txt reports the revenues from finished orders. 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 of semimanufactured orders at a specific date (evaluated using revenuePerEachRecipeStep; in this case set to 21). 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
```

BENEFIT

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
```

Deepening Case 2

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 "Order in Unit"s graph at the end of the previous day, being the orders propagated in the second part of each day

prod. = production within the day

(**) 100a (and below 100b, 100c, 100d, ...) disappears here because it is immdiately kept 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. Deepening Case 2.

Case 2 accounting

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.

REFERENCES

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