DISTRIBUTED PRODUCTION WITH SPECIFICATION-GENERATED PROCESSES

Tomasz Janowski
The United Nations University
International Institute for Software Technology
P.O.Box 3058, Macau, China
tj@iist.unu.edu

We study design support for a virtual organization: how to organize production among the set of distributed cells to meet a given production goal. Each cell has a certain capacity to store, manufacture and deliver products, and interacts with other cells by means of information and logistics networks. The goal specifies the product and the volume required. To solve this problem we propose a protocol which given a goal and a cell attempts to generate a sequence of operations on the cell to implement this goal. Lacking the resources locally the protocol generates some auxiliary goals which it communicates to other cells, involving generating (recursively) more processes and goals etc. We formalize the problem and its solution, offer correctness arguments and discuss an email-based implementation.

1. INTRODUCTION

Industrial virtual organizations are no longer just a subject of academic study, they are a fact of life (Handfield and Nichols, 1999). They emerged as soon as the technological progress, in particular information and communication technology made them possible. Yet behind this application-driven progress, we have seen rather less progress in methodologies (methods, notations and tools) to guide the design of such organizations (Vernadat, 1996). Without such a methodology it is hard to realize what kind of objectives the organization is supposed to meet, does it meet those objectives, what design decisions can be made, what are the tradeoffs attached with each etc. One research direction is trying to automate as much as possible of the design effort which comes to building a virtual organization, by: formulating relevant problems, proposing algorithmic solutions, arguing (formally or informally) about their correctness, finally providing implementations.

In this paper we investigate the problem of distributed production planning to meet a certain production goal. The goal includes the type of the product and the volume required (number of items). As usual, to meet this goal the virtual organization is not built from the scratch but based on the set of existing organizations, each modeled as a single production cell. A cell has a certain capacity to store, manufacture and deliver products. The shopfloor (manufacture) capacity is static while the product stocks are dynamic. The cell carries out production internally, based on the existing stocks/shopfloor, and also interacts with other cells by means of information and logistics networks. We provide the formalization of this problem, its constraints and objectives, and propose an algorithmic solution.

The solution consists of a process-generator, which given a cell and a goal attempts to built a process (a sequence of operations) to be carried out by the cell to meet this goal. Lacking the resources locally the generator may also output some auxiliary goals. Then in order for the cell to meet the original goal, other cells must satisfy those auxiliary goals. The protocol communicates such goals over the information network, trying to find a match between them and the cells. This in turn involves the work of the process-generator in each individual cell, therefore producing more processes and possibly more auxiliary goals. This procedure is recursive in nature, however with each turn the goals become generally "simpler", in terms of the volumes (smaller) or products (lower in the bill-of-products hierarchy) involved, until a cell is found which implements them entirely on its own. The protocol fails to find the right match between the cells and the goals when it lacks the resources among all of the cells (globally). We formalize this solution, justify why it correctly solves the problem and propose an email-based implementation.

The rest of this paper is as follows. Section 2 introduces the notation and provides the basic concepts for production modeling. The notation is a fragment of the RAISE Specification Language (The RAISE Method Group, 1992). Section 3 formulates the problem of distributed production planning (Section 3.1), proposes a solution (Section 3.2), argues about its correctness (Section 3.3) and describes an email-based implementation (Section 3.4). Section 4 presents some conclusions.

2. MODELING PRODUCTION, SYMBOLICALLY

Modeling production starts with modeling products. The type *Product* represents all products. We have two functions on this type: *size* returns a number representing the storage requirements of a product and *bill* returns all sub-products with their quantities (to obtain a single item of the product). Both are declared as values of the corresponding functional types, the return type of *size* is ***Nat*** (natural numbers), the return type of *bill* is *Product* $\xrightarrow{m}$ ***Nat*** (finite maps from *Product* to ***Nat***).

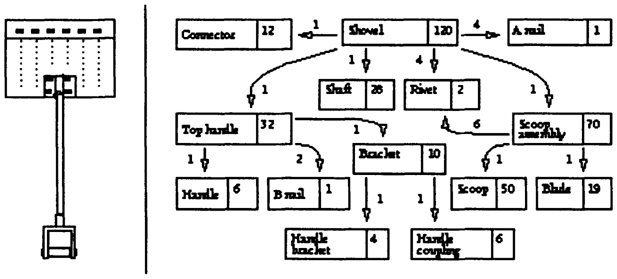
type	**value**
Product	size: Product $\rightarrow$ **Nat**,
	bill: Product $\rightarrow$ (Product $\xrightarrow{m}$ **Nat**)

We also introduce some axioms that constrain the values of those functions: *size* never returns zero, *bill* never returns a map with a zero value and no product is a sub-product of itself. The latter involves an auxiliary function *issub* to determine if one product is a sub-product (immediate or not) of another. Below, ***dom*** applied to a map returns the set of all its arguments, $\times$ is a Cartesian product and ***Bool*** is the type of the Boolean values with the usual logical operations.

```
axiom
  (∀ p,q: Product •
    size(p) > 0 ∧ ¬issub(p,p) ∧
    q ∈ dom bill(p) ⇒ bill(p)(q)>0 )
```

```
value
  issub: Product × Product → Bool
  issub(q,p) ≡ q ∈ dom bill(p) ∨
    (∃ r:Product • issub(q,r) ∧ issub(r,p) )
```

Consider an example product, a snow shovel (Vollmann, 1992) (Janowski, Moreira and Sousa, 2000). The picture below shows the product design on the left and the sub-product graph on the right, which includes the values of functions *size* (numbers in boxes) and *bill* (arrows and labels).



Production is carried out within a cell subject to the constraints on: the maximum number of products the cell can store in the warehouse (weighted by their storage requirements), the numbers of products in the warehouse (stocks), the set of products we can manufacture from their sub-products and the number of items manufacturable within a shift (shopfloor). Formally, we introduce an abstract type *Cell* and three corresponding functions *space*, *stock* and *shop*, such that the current occupancy of the warehouse is not greater than its capacity, a product that can be manufactured is non-atomic and the manufacturable quantity is at least one.

```
type
  Cell
value
  space: Cell → Nat,
  stock: Cell × Product → Nat,
  shop: Cell → (Product ─m→ Nat)
```

```
axiom
  (∀ c:Cell •
    occupancy(c) ≤ space(c) ∧
    (∀ p:Product • p ∈ dom shop(c) ⇒
      shop(c)(p)≠0 ∧ bill(p)≠[])
  )
```

Function *occupancy* adds all stocks in a cell, weighted by their storage requirements. We assume that *space* and *shop* are static attributes which only constrain production. There are three operations to change stocks: *store* increments the stock of a given product, *deliver* decrements the stock and *manufacture* increments the stock for the product and decrements the stocks for all its sub-products. They have the same type:

```
value
  store, deliver, manufacture: Product × Nat × Cell ─~→ Cell
```

For instance, *manufacture* is defined using a pair of logical conditions: a pre-condition defines the range of argument values, a post-condition defines the relation between the arguments and the results, provided they satisfy a pre-condition.

```
manufacture(p,n,c) as c'
post stock(c',p) = stock(c,p)+n ∧ ...
  (∀ q:Product • q ∈ dom bill(p) ⇒ stock(c',q)=stock(c,q)-n*bill(p)(q))
pre p ∈ dom shop(c) ∧ n ≤ shop(c)(p) ∧ ...
```

The type *Operation* includes all operations together with their arguments. Two functions are defined on this type: *enough* determines if a given cell has enough resources to execute an operation (the corresponding pre-condition holds) and *exec* executes the operation on the cell and returns a modified cell.

```
type
 Operation == store(Product, Nat) |
    deliver(Product, Nat) | manufacture(Product,Nat)
```

```
value
 enough: Operation × Cell → Bool
 enough(op,c) ≡
   if op=deliver(p,n)
   then stock(c,p) ≥ n else ... end,
```

```
exec: Operation × Cell ~→ Cell
exec(op,c) ≡
  if op=store(p,n)
  then store(p,n,c) else ... end
  pre enough(op,c)
```

The actual production is carried out by a sequence of such operations. We introduce the type *Process* and the functions *enough* and *exec*, to check if a cell has enough resources for a process and to execute a process, respectively. ***hd*** returns the first element of a non-empty list and ***tl*** returns the list with the first element removed.

```
type
 Process = Operation*
value
 enough: Process × Cell → Bool
 enough(p,c) ≡ p = < > ∨
   enough(hd p,c) ∧ enough(tl p,exec(hd p,c)),
```

```
exec: Process × Cell ~→ Cell
exec(p,c) ≡
  if p= < > then c
  else exec(tl p,exec(hd p,c))
  end pre enough(p,c)
```

Consider a cell *c* to produce snow shovels (Janowski, Moreira and Sousa, 2000). The following tables describe the values of the functions *space*, *stock* and *shop* when applied to this cell. In particular, the cell has the storage capacity of 200000, contains the stock of 150 shovels and can manufacture 100 items during a shift.

space
200000

shop	
shovel	100
scoop_assembly	260
top_handle	250
bracket	300

stock	
shovel	150
scoop_assembly	300
rivet	1600
connector	550
top_handle	400
shaft	150
a_nail	1200
blade	300
scoop	250

3. PLANNING DISTRIBUTED PRODUCTION

Equipped with the notation and the basic concepts for modeling production, we can approach the main problem of this paper: planning distributed production. In the following sections we formulate the problem, propose a solution, analyze it informally and discuss a possible implementation.

3.1 Problem Formulation

The problem is to implement a given production goal using the resources offered by a set of cells. Each goal describes the product and the number of items to deliver, formally a value of the record type *Goal* with two corresponding fields *prod* and *vol* and the function *revol* which modifies a goal by changing its volume field.

```
type
  Goal:: prod: Product  vol: Nat ↔ revol
```

Consider first a restricted case of the problem, for a single cell *c*. Given a goal *g*, the problem is to find a process *p* such that *c* has enough resources to execute *p* and the execution results in the stock for *prod(g)* no less than *vol(g)*. The function *satisfies* formalizes this property while *generate* (defined implicitly) produces the required process, if one exists. Together, they formulate the problem for the single-cell case.

```
value
  satisfies: Process × Goal × Cell → Bool
  satisfies(p,g,c) ≡
    enough(p,c) ∧ stock(exec(p,c),prod(g)) ≥ vol(g),
  generate: Goal × Cell $\xrightarrow{\sim}$ Process
  generate(g,c) as p post satisfies(p,g,c) pre (∃ p:Process • satisfies(p,g,c))
```

Consider a goal *g*, a set *cs* of production cells and a distinguished cell *c∈cs*. We seek a process *p* for *c* that implements *g*. For all *store* operations in *p, s*uch an operation defines a sub-goal *g'*, we also look for their implementation in *cs,* a cell *c'∈cs* and a process *p'* which implements *g'* when executed on *c'* . Like *p*, *p'* may also contain sub-goals (sub-sub-goals with respect to *p*). To represent such executions involving multiple cells we introduce a "protocol", a process which links all *store* operations with their possible implementations (a cell and a protocol, recursively). We say the protocol is well formed with respect to the cell-set *cs* (function *iswf*) if every cell occurs in this protocol (and recursively any sub-protocol) at most once and is in *cs*.

```
type
  Action ==
    deliver(Goal) |
    store(Goal, Protocol, Cell) ...,
  Protocol = Action*
value
  cells: Protocol → Cell-set,
```

```
iswf: Protocol × Cell-set → Bool
iswf(p,cs) ≡ p=< > ∨
  if hd p=store(g,p',c)
  then iswf(p',cs\{c}) ∧
          iswf(tl p, cs\(cells(p') ∪{c}))
  else iswf(tl p, cs) end
```

Then in order for the protocol *p* and the cell *c*∈*cs* to satisfy a goal *g* the following conditions must be satisfied: (1) *c* has enough resources to execute *p*, (2) the execution results in the stock for *prod(g)* greater than or equal to *vol(g)* and (3) every sub-goal *g'* in *p* is satisfied by the corresponding cell *c'*∈*cs* and the protocol *p'*. The function *satisfies* is defined recursively, as below. Function *generate* produces the required protocol, if one exists, defined using pre- and post-conditions.

```
value
 satisfies: Protocol × Goal × Cell × Cell-set $\xrightarrow{\sim}$ Bool
 satisfies(p,g,c,cs) ≡
   enough(p,c) ∧ stock(exec(p,c),prod(g)) ≥ vol(g) ∧
   (∀g':Goal, p':Protocol, c':Cell • store(g',p',c') ∈ elems p ⇒ satisfies(p',g',c',cs))
   pre c ∈ cs ∧ iswf(p,cs\{c}),
 generate: Goal × Cell × Cell-set $\xrightarrow{\sim}$ Protocol
 generate(g,c,cs) as p
   post iswf(p,cs\{c}) ∧ satisfies(p,g,c,cs) pre c ∈ cs ∧ satisfiable(g,c,cs)
```

3.2 Problem Solution

We provide an explicit definition for the function *generate*, showing how it can be actually calculated. Like before, we start with the single cell case.

Given a goal *g* and a cell *c* the function first checks if the stock in *c* already satisfies *g*. If so then we produce the empty process, otherwise we recalculate the volume by subtracting the existing stock. For the resulting goal *g'*, if the product is not manufacturable then we generate the process with an operation to store the missing quantity. Otherwise we first generate a process *p1* to obtain the needed sub-products (*gen_sub*) then *p2* to manufacture the final product (*gen_fin*) and finally concatenate such processes (*p1* ^ *p2*). Function *gen_fin* takes into account the cell *c* after it executed *p1* (*exec(c,p1)*). Here is the explicit definition of *generate*:

```
value
 generate: Goal × Cell $\xrightarrow{\sim}$ Process
 generate(g,c) ≡
   let p=prod(g), v=vol(g)
   in if stock(c,p) ≥ v then < >
      else let g'=revol(v-stock(c,p),g)
           in if p ∉ dom shop(c) then <store(p, v-stock(c,p))>
              else let p1=gen_sub(g',c,dom bill(p)), p2=gen_fin(g',exec(p1,c))
                   in p1 ^ p2 end end
   end end end pre (∃ p:Process • satisfies(p,g,c))
```

Function *gen_fin* generates a sequence of *manufacture* operations for *prod(g)* and either the maximum quantity allowed by the shopfloor (*shop(c)(prod(g))*) or *vol(g)* (if less than this quantity), until the whole required volume is manufactured. Function *gen_sub* considers each immediate sub-product of *prod(g)*, calculates a (sub-) goal for this product according to the needs of *g*, recursively calls *generate* to produce a process for this goal, and concatenates all such processes.

```
value
 gen_fin: Goal × Cell $\xrightarrow{\sim}$ Process
 gen_fin(g,c) ≡
   let p=prod(g), v=vol(g), n=shop(c)(p)
   in if v ≤ n then <manufacture(p,v)>
      else <manufacture(p,n)> ^ gen_fin(revol(v-n,g),c) end
   end pre ...,
 gen_sub: Goal × Cell × Product-set $\xrightarrow{\sim}$ Process
 gen_sub(g,c,ps) ≡
   if ps={} then < > else
     let p:Product • p ∈ ps in
       let v=vol(g)*bill(prod(g))(p),
           p1=generate(mk_Goal(p,v),c), p2=gen_sub(g,exec(p1,c),ps\{p})
       in p1 ^ p2 end
   end end pre ...
```

Consider the cell *c* to produce snow shovels, as in Section 2. Below, we show some results produced by the *generate* function, applied to deliver various quantities of the shovel product. If the quantity is 150 then *generate* returns an empty process, the amount is already on stock. If the quantity is 250 then we still lack 100 items, which can be manufactured within one shift from the existing subproducts. If the quantity is 300 then we already need two shifts, one to manufacture 100 items and another 50 items; there is still enough sub-products. Raising the quantity to 450, however, we miss 150 shafts which cannot be manufactured internally but must be obtained from outside. Finally, with the quantity raised to 500, we miss 250 shafts, 200 a-nails and 50 scoop-assemblies. The latter can be manufactured but the added demand on rivets (for producing scoop-assemblies as well as shovels) exceeds the stock by 100.

generate(shovel,150,c)

generate(shovel,250,c)
manufacture(shovel,100)

generate(shovel,300,c)
manufacture(shovel,100)
manufacture(shovel,50)

generate(shovel,450,c)
store(shaft,150)
manufacture(shovel,100)
manufacture(shovel,100)
manufacture(shovel,100)

generate(shovel,500,c)
store(shaft,250)
store(a_nail,200)
manufacture(scoop_assembly,50)
store(rivet,100)
manufacture(shovel,100)
manufacture(shovel,100)
manufacture(shovel,100)

In the general case, consider a set *cs* of cells, *c*∈*cs* and a goal *g*. The main difference from the single-cell case is how we deal with the lack of resources in *c* to satisfy *g*; when *c* has insufficient stock to satisfy the volume of *g* and it lacks the capacity to manufacture the missing number of *prod(g)*. First we assume (by a precondition on *generate*) that there exists at least one cell in *cs* for which *g* is satisfiable. Second, we take any cell *c'*∈*cs* with this property, recursively generate a protocol *p* for *c'* and produce a corresponding *store* action where *g*, *p* and *c'* occur explicitly. Otherwise, the *generate* function is similar as before:

```
value
 generate: Goal × Cell × Cell-set ~→ Protocol
 generate(g,c,cs) ≡
   if stock(c,prod(g))>=vol(g) then < >
   else let g'=revol(vol(g)-stock(c,prod(g)),g)
        in if prod(g) ∉ dom shop(c)
           then let c':Cell • c' ∈ cs ∧ satisfiable(g',c',cs\{c})
                 in <store(g',generate(g',c',cs\{c}),c')> end
           else let p1=gen_sub(g',c,dom bill(p),cs), p2=gen_fin(g',exec(p1,c))
                in p1 ^ p2 end end
   end end pre c ∈cs ∧ satisfiable(g,c,cs)
```

gen_fin is as before, it generates the *manufacture* operations for *c* after it has executed *p1* (*exec(p1,c)*) but ignoring the effect of *p1* on other cells. *gen_sub* is also similar, except it now returns a protocol (rather than a process) and takes the set *cs* of cells as one more argument; in order to pass it to the function *generate*. It also follows on the state-changes in *c* but not how other cells change: as soon as the cell is used in the protocol for a sub-product it is not considered for the remaining sub-products (the *iswf* condition for the well-defined protocol).

3.3 Protocol Correctness

We argue about the correctness of the solution: that the model is well defined, that it is internally consistent, and that it provides the correct solution to our problem.

The first part we can check fully automatically. The RSL type checker will parse all specifications, decide if they are syntactically correct and obey the typing constraints. For instance, it will detect that the precondition for the *manufacture* function applies *stock* in the wrong order of its arguments and the *generate* function tries to concatenate a protocol with a process. The latter requires explicit conversions from *Protocol* to *Process* (*flat*) and vice versa (*nest*), applied whenever we want to execute a protocol on a given cell (*exec(flat(p),c)*) or concatenate the two (*p1^nest(p2)*). Except for those two errors, the models are checked successfully.

The second part involves checking in particular that all functions are applied within their preconditions. This task cannot be fully automated as it would involve reasoning about the truth of arbitrary logical expressions. Instead, the type checker generates some confidence conditions to inspect. There are 47 conditions generated for the specifications in this paper, most of which are discarded immediately. For instance: *bill(p)(q)>0* requires checking that *q* is in the domain of *bill(p)*, *d*ecreasing the stock in the *manufacture* function requires checking that the result is positive, using the let expression to choose a cell satisfying the goal (in the *generate* function) requires checking that such a cell exists. And so on.

The third part is to argue that the proposed solution correctly solves the problem, that the explicit definition of *generate* satisfies the implicit definition; the post-condition produces a logically true statement: *satisfies(generate(g,c,cs),g,c,cs)* for any goal g, set of cells *cs* and *c* ∈*cs*, such that *g* is satisfiable by *c* and *cs*. Given that *satisfies* is defined recursively, we justify this by induction: if the statement is true for all operations *store(g',p',c')* in the generated protocol then it is true for the protocol itself. We demonstrate that: (1) *c* has enough resources to execute the generated protocol and (2) the resulting stock for *prod(g)* satisfies the required volume *vol(g)*. Clearly, the protocol is generated to satisfy both conditions: we first prepare the required quantities of all sub-products of *prod(g)* (*gen_sub*) then generate the sequence of *manufacture* operations (making sure the volumes do not exceed the shopfloor capacity) to produce *vol(g)* items of *prod(g)*. If the required product is not manufacturable and the stock does not satisfy the required volume then we delegate the goal (minus the existing stock) to some other cell. We can see that (1) is satisfied as we make sure the resources exist before we try to use them as well as (2) holds with the stock value exactly equal to the required volume.

3.4 Email-Based Implementation

The implementation is based on the environment for simulating production systems. Each cell has a concrete representation using text files: the *STOCKS* file describes product stocks, the *STATIC* file describes the warehouse/shopfloor constraints and the *PRODUCTS* file contains product information. Given such files, we implement the operations on the cell (*store*, *deliver* and *manufacture*) as text transformations, which read all three files above and modify the *STOCKS* file; we implemented them with the `Awk` language (Robbins, 1989). Similar text files are used to document a process, so the *exec* and *enough* functions can be implemented as simple shell scripts. In particular *exec* looks through the process file and for each entry calls the corresponding `Awk` program. We assume that each cell has an email address to communicate with other cells, with two additional files to facilitate communication: *aliases* defines the mail alias *cells* with the addresses of all active cells and *.forward* causes each received email is to be sent to the script *incoming* for further processing.

The implementation for the protocol is the program *generate*, available to run for each cell. Initially, the program is called from the command line, given a goal as input. Then it carries out processing of this goal, taking into account the local *STOCKS* and *STATIC* files and printing the generated operations on the screen. The processing is like described by the function *generate*. There are two obstacles, however, both related to the use of an implicit *let* expression (take an arbitrary product/cell satisfying a certain property). In *gen_sub,* the implementation looks through the *PRODUCTS* file to produce the first product satisfying the property.

In *generate*, the *let* expression is used to choose a cell which can satisfy a given goal. The implementation broadcasts the goal to all active cells. When a cell receives such a message it calls the local *generate* program, then sends back the result of this invocation. The replies are gathered in a file: either a process description that tells how to implement the goal or an indication that the goal is not satisfiable. When all replies have been received and all failed to produce a result then *generate* also outputs the failed message and quits. Otherwise it works as follows: (1) accept one of the received solutions, (2) broadcast a message to *cells* informing them about the choice, (3) calculate all cells in the solution and remove them from the *cells* alias, (4) print the choice on the screen and (5) continue as in *generate*. There are many ways to reach a decision (1): the first solution received, the "best" solution received, the solution coming from the top-priority cell (according to some ranking). The details of the implementation will be subject of a companion paper.

4. CONCLUSIONS

We focused in this paper on the specific problem related to the design of virtual organizations: how to distribute production among a set of cells to satisfy a given production goal. We formalized this problem, proposed a solution, argued about correctness and outlined an implementation. The model for production follows (Janowski, Lugo and Zheng, 1999) and the simulation environment have been described in (Janowski, Moreira and Sousa, 2000).

The concept of a virtual organization has received a lot of attention. We have seen reports on case studies (Molina and Flores, 1999), architectures (Camarinha-Matos and others, 1997), standards (Clements, 1997), protocols (Hardwick and others, 1997) and supporting technologies (Camarinha-Matos and Afsarmanesh, 1997). Also planning and design issues has been approached (Schonsleben and Buchel, 1998), albeit in an informal way. This paper is different in the sense of adopting a formal approach to modeling, used to capture a specific design-related problem, then propose and justify a solution to this problem.

We have several plans to continue this work. We want to document in more detail the implementation of the protocol, including its applications to some concrete problems. To gain the confidence in our design we intend to prove that this protocol is correct, with formal rather than informal arguments. We want to extend the problem with timing attached to production goals: the maximum number of shifts from placing the order to its final delivery. This, in turn, opens the possibility for accepting some imperfect (perhaps violating the deadline) but still satisfactory (producing the required volume of the product) solutions.

5. REFERENCES

1. Camarinha-Matos LM and Afsarmanesh H. Handbook of Life Cycle Engineering, Virtual Enterprise: Life Cycle Supporting Tools and Technologies. Chapman and Hall, 1997.
2. Camarinha-Matos LM, Afsarmanesh H, Garita C and Lima C. Towards an Architecture for Virtual Enterprise. World Congress on Intelligent Manufacturing, 1997.
3. Clements P. Standards Support for the Integration and Interoperation of the Virtual Enterprise. Enterprise Integration Modeling Technology, 1997.
4. Handfield R and Nichols E. Supply Chain Management. Prentice Hall, 1999.
5. Hardwick M, Rando T, Spooner DL and Morris KC. Data Protocols for the Industrial Virtual Enterprise. IEEE Journal of Internet Computing, 1997, vol. 1, no. 1.
6. Janowski T, Lugo G and Zheng H. Modeling an Extended/Virtual Enterprise by the Composition of Enterprise Models. Journal of Intelligent and Robotic Systems, 1999, vol. 26, no. 2-3, 303-324.
7. Janowski T, Moreira F and Sousa R. Production Modeling as Shell Programming: Concurrency and Delegation. IFAC Symposium on Manufacturing, Modeling, Management and Control, Rio, Greece, July 2000. Elsevier.
8. Molina A and Flores M. A Virtual Enterprise in Mexico: From Concepts to Practice. Journal of Intelligent and Robotic Systems, 1999, vol. 26, no. 2-3, 289-302.
9. Robbins AD. Effective Awk Programming. Free Software Foundation, 1989.
10. Schonsleben P and Buchel A. Organizing the Extended Enterprise. Chapman and Hall, 1998.
11. Vernadat F. Enterprise Modeling and Integration. Chapman and Hall, 1996.
12. Vollmann TE. Manufacturing, Planning and Control Systems. Irwin, 1992.
13. The RAISE Method Group. The RAISE Specification Language. Prentice Hall, 1992.