PROTOCOL-BASED COOPERATION IN A VIRTUAL MANUFACTURING ORGANIZATION

Tomasz Janowski and Phan Cong Vinh*
The United Nations University
International Institute for Software Technology
P.O. Box 3058, MACAU
{tj,pcv}@iist.unu.edu

Cooperation between the members of a virtual manufacturing organization can be described as following a certain protocol, similar to communication protocols in distributed systems. We investigate how it is possible to model such protocols explicitly, for an organization modeled as a client-server system where one member (a client) can request another (a server) to fulfill a certain production goal. The goal is given in an abstract way: the type of the product and its required quantity. It is up to the server to decide if and how to implement this goal. We present several protocols how the servers can react to such requests and the clients receive their responses. We also discuss how the choice of a protocol affects the behavior of the whole organization.

1. INTRODUCTION

One of the main aspects of a virtual organization is how its members interact with each other to fulfill a larger corporate goal. In the absence of central control, as in all distributed systems, this interaction can take place according to a protocol followed by each member. The choice of a protocol crucially affects the performance of the whole organization and how it is able to satisfy its goals. It is useful, therefore, if such protocols can be described explicitly and their behavior analyzed/simulated.

The purpose of this paper is to study protocol-based interactions between the members of a virtual manufacturing organization. We base this study on the client-server model of the organization, where one member (a client) can request another (a server) to fulfill a certain production goal. The goal is given in an abstract way: the type of the product and the volume required (number of items). It is up to the server to decide if and how to implement this goal. We present several protocols how the servers can react to such requests and the clients receive their responses. As it turns out, there are many possible decisions one can make to construct such protocols, with important consequences on the behavior of the whole organization.

* Vietnam Posts and Telecommunication Institute of Technology, Ho Chi Minh City, Vietnam.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35399-9_52

For instance, the server may decide to only accept the goals that it can fulfill immediately, from the products already on stock. A more eager server may decide to manufacture the missing products from the existing stocks, rejecting the goal if the product is not manufacturable on the shopfloor or there are not enough sub-products. A step further is to also manufacture those missing sub-products, as much as the stock allows, before the product itself. With each decision the server is able to fulfill more of its clients' requests but also requires them to wait longer. Instead, we may decide that the goal includes the deadline -- the maximum time from placing the order to its final delivery. In this case the server rejects the goal if it lacks the resources to implement it or the implementation exceeds the deadline. In all those cases, the client always requests a certain production goal and the server responds with a yes/no answer. Given a positive answer production can start immediately, as the client has no more decisions to make; communication takes place in two phases.

A different approach is to let the server reply with the maximum time to complete the goal (it at all). In this case the client has still to decide. Perhaps it has inquired with two different servers and is going to decide towards the one that can fulfill the order in the shortest time. However done, the server has still to wait for the client's final decision, if it replied with a positive answer (timing). This decision may be just a final yes/no. On the other, based on the responses received from two or more servers, the client may not be able to decide in favor of any of them in particular. Instead, it may decide that the goal be divided between them, to let them work concurrently on its parts. In that case the positive reply from the client would carry the quantity the client wants to allocate to the server, typically smaller than the original quantity. This way we would complete the negotiation in three phases (client inquires, server responds, client confirms).

One problem with the protocol above is the timing to deliver the reduced goal, which the client may be unable to calculate based on the original goal and its timing. One solution is having the server reply with a quantity-to-time function instead of just time for a given quantity. Another solution is to continue the negotiation: the server inquires with the new goal (reduced quantity), server responds, client confirms etc. This solution is also more appropriate when the server negotiates with several clients, applying some criteria to accept or reject the goals offered to it. Normally, it cannot accept all offered goals because of the deadlines and limited resources. For instance, it may decide to reject the goals with the quantities below a certain minimum. So we enter the fourth negotiation phase, etc.

Those are some of the design choices one can make to decide on the cooperation protocol. The paper presents a formal model where such decisions can be captured, discussed and analyzed. The model is written in a high-level formal specification language (The RAISE Method Group, 1992). The rest of this paper is as follows. Section 2 introduces the notation and provides the basic concepts for modeling production. Section 3 describes, through modeling and discussion, several protocols for cooperation between the members of a virtual manufacturing organization. Each protocol is motivated, discussed, and linked with the next, more complex protocol. Section 4 presents some conclusions.

2. MODELING PRODUCTION

Suppose the abstract type *Product* represents all kinds of products. Two functions are defined on this type: *size* returns a number which encodes the storage requirements of a product and *bill* returns a sub-product relation, all immediate sub-products with their quantities (to obtain a single item of the product). Both functions are values of the corresponding functional types, the return type of *size* is ***Nat*** (natural numbers) and of *bill* is *Product* $\xrightarrow{m}$ ***Nat*** (maps from *Product* to ***Nat***).

```
type
  Product
```

```
value
  size: Product → Nat,
  bill: Product → (Product $\xrightarrow{m}$ Nat)
```

We require that *size* never returns a zero, *bill* never returns a map with a zero value and no product is a sub-product of itself (according to *bill*). The latter involves an auxiliary function *issub* to determine if one product is an immediate or non-immediate sub-product of another. Below, ***dom*** applied to a map returns the set of all its arguments, × is a Cartesian product and ***Bool*** is the type of Boolean values.

```
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) )
```

We carry out production within a production cell, subject to the constraints on: the maximum number of products in the warehouse (weighted by their "size"), how many products are in the warehouse (stocks), which products can be manufactured from their sub-products and how many items during a shift (shopfloor). Formally, we define an abstract type *Cell* and three corresponding functions on this type: *space*, *stock* and *shop*, such that the warehouse occupancy is not greater than its capacity, each manufacturable product is non-atomic and the quantity is at least one.

```
type
  Cell
value
  space: Cell → Nat,
  stock: Cell × Product → Nat,
  shop: Cell → (Product $\xrightarrow{m}$ Nat)
```

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

Production changes the stocks within a cell: function *store* increments the stock of a product, *deliver* decrements the stock and *manufacture* increments the stock of a product and decrements the stocks for all its sub-products. They have the same type:

```
value
  store, deliver, manufacture: Product × Nat × Cell $\xrightarrow{\sim}$ Cell
```

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)
```

A process describes the low-level implementation of a certain production goal. Suppose the goal describes a product and its quantity (type *Goal*) which should be present in the cell after the process finished its execution (function *sat*). If the cell has not enough resources for the process to execute then the result of *sat* is underspecified. Function *issat* decides if a goal is implementable for a cell – there exists a process which can be executed on the resources present in the cell and which satisfies the goal. It is a precondition to the function *gen*, defined implicitly, which returns such a process for a given cell and a goal.

```
type
 Goal = Product × Nat
value
 sat: Process × Cell × Goal -~-> Bool
 sat(p,c,(q,n)) ≡
   stock(exec(p,c),q) ≥ n
   pre enough(p,c),
```

```
issat: Cell × Goal → Bool
issat(c,g) ≡
 (∃ p:Process • enough(p,c) ∧ sat(p,c,g)),
gen: Cell × Goal -~-> Process
gen(c,g) as p
  post enough(p,c) ∧  sat(p,c,g)
  pre issat(c,g)
```

3. PROTOCOLS FOR DISTRIBUTED PRODUCTION

Suppose a distributed production system consists of several cells that carry out their own production activities but also interact with each other. Interaction occurs by one cell (a client) requesting another (a server) to implement a certain production goal. In this section we describe several protocols for communication between clients and servers, negotiating the implementation of a given production goal. The main difference is how many phases it takes for the two parties to reach an agreement. Protocols use a network for communication between the cells.

3.1 Communication Network

Let the type *Msg* describe all messages that can be communicated between the cells. A network is a map from cells to sequences of messages (type *Net*), where a sequence contains all messages sent to but not yet received by the cell, first-in-first-out. There are two functions defined on the network: *snd* sends a message to a given cell, provided the cell exists in the network, *rcv* removes the first message for a given cell, provided the cell exists and its message queue is non-empty.

```
type
  Msg,
  Net = Cell $\xrightarrow{m}$ $Msg^*$
```

```
value
  snd: Msg × Cell × Net $\xrightarrow{\sim}$ Net
  snd(m,c,n) as n' post n'(c) = n(c) ^ <m> ∧ ...
    pre c ∈ dom n,
  rcv: Cell × Net $\xrightarrow{\sim}$ Net
  rcv(c,n) as n' post n'(c) = tl n(c) ∧ ...
    pre c ∈ dom n ∧ n(c) ≠ < >
```

The following sections describe how clients and servers communicate via this network. The protocols involve an increasing number of phases to complete the negotiation, starting from the simplest one-phase protocol.

3.2 One-Phase Protocol

Suppose the client requests the server to implement a given production goal, by sending the goal over the network. Then it carries out with its own business without waiting or expecting any acknowledgment. The server, on the other hand, either implements the request by constructing and executing the corresponding process (which satisfies the goal) or decides to reject the request and does nothing.

```
type
  Msg = Goal
value
  request: Msg × Cell × Net $\xrightarrow{\sim}$ Net
  request(m,c,n) ≡ snd(m,c,n)
    pre c ∈ dom n,
```

```
reply: Cell × Net $\xrightarrow{\sim}$ Cell × Net
reply(c,n) ≡
  let g = hd n(c)
  in if accept(c,g)
      then (exec(gen(c,g),c),rcv(c,n))
      else (c,rcv(c,n)) end
  end pre c ∈ dom n ∧ n(c) ≠ < >
```

Decision about acceptance/rejection is done by the function *accept*, given a goal and a cell. The server may decide to only accept the request if: (1) able to fulfill it directly from the stock or (2) the missing quantity can be manufactured from the stock or (3) the product and its sub-products can be obtained from the stock. The last is the weakest, we only require the corresponding process to exist.

```
value
  accept: Cell × Goal → Bool
  accept(c,(p,n)) ≡
   1. stock(c,p) ≥ n
   2. p ∈ dom shop(c) ∧
      (∀ q:Product •
        q ∈ dom bill(p) ⇒
        stock(c,q) ≥ bill(p)(q)*(n-stock(c,p))
      )
   3. issat(c,(p,n))
```

There is only one communication in the protocol above, from the client to the server to request implementation of a goal. Afterwards, the client has no idea if the server decided to accept or reject the request (lacking the resources). To allow clients to follow-up on unsuccessful requests we introduce a two-phase protocol.

3.3 Two-Phase Protocol

In a two-phase protocol the server sends back a reply to the client, to inform about the outcome of its request. Consider the simplest kind of reply: *accept* or *reject*. We have to extend the message type with two kinds of messages: *req* is a request message, includes the goal and the client's name (to know where to send a reply), *rep* is a reply message, includes the decision by the server.

```
type
 Reply ==
   reject | accept,
 Msg = =
   req(from:Cell, go:Goal) |
   rep(Reply)
```

```
value
 reply: Cell × Net ~→ Cell × Net
 reply(c,n) ≡
   let g=go(hd n(c)), d=from(hd n(c)), n'=rcv(c,n)
   in if accept(c,g)
      then (exec(gen(c,g),c),snd(rep(accept),d,n'))
      else (c,snd(rep(reject),d,n') end
   end pre c ∈ dom n ∧ n(c) ≠ < >
```

The more goals the server is willing to accept the longer it takes to complete them. The client may wish to take more control over the decision, including in the request the maximum time to complete the goal, in terms of the number of shifts. The server will only accept the goal if it can construct a process that satisfies this goal and completes before the deadline. We assume *gen* returns the fastest process for a given goal and *time* calculates the number of shifts for a process to complete.

```
type
 Time = Nat,
 Msg = =
    req(from:Cell, go:Goal, dn: Time) |
    rep(Reply)
```

```
value
 reply: Cell × Net ~→ Cell × Net
 reply(c,n) ≡ ...m = hd n(c)...
  if issat(c,go(m)) ∧
     time(gen(c,go(m)))≤ dn(m)
  then ...accept... else ...reject... end
```

3.4 Three-Phase Protocol

The two-phase protocol does not allow the client to choose between the servers, as they all produce a yes/no answer. A different approach is letting a server reply with the minimum time to get the goal completed, if at all. This allows the client to choose the server that is able to implement the goal fastest. Function *reply* rejects the goal if not implementable, otherwise accepts the goal with the time-to-complete of the fastest process generated for the goal. Function *confirm* takes as an argument a map from cells to numbers, representing their proposed completion times. It sends the positive confirmation (*cnf(go)*) to the server which proposed the minimum time (*min(m)*) and the negative confirmation (*cnf(forget)*) to all other servers (*sndall*).

```
type
 Reply ==
   reject |
   accept(Time),
 Confirm ==
   forget | go,
 Msg ==
   req(Cell,Goal) |
   rep(Reply) |
   cnf(Confirm)
```

```
value
 reply: Cell × Net -~-> Cell × Net
 reply(c,n) ≡ ...g = go(hd n(c))...
   if issat(c,g) then ...accept(time(gen(c,g)))...
   else ...reject... end ...
 confirm: Cell × Net × Goal × (Cell -m-> Nat) -~-> Net
 confirm(c,n,g,m) ≡
   n'=snd(cnf(go),min(m),n)...
   sndall(cnf(forget), dom m \ {min(m)},n')...
   pre {c} ∪ dom m ⊆ dom n ∧ card dom m > 0
```

It remains possible that the promised completion time of the fastest server is still behind the client's deadline. In this case the client may decide to let two or more servers work concurrently on the parts of the goal. For instance, it may choose two servers *s1* and *s2* with the smallest proposed times and divide the volume of the original goal (*q=q1+q2*) proportionally to those times (*q1*m(s2)=q2*m(s1)*), according to the function *divide*. Then it sends the positive confirmations to *s1* (*go(q1)*) and *s2* (*go(q2)*) and negative (*forget*) to all other servers.

```
type
 Confirm ==
   forget | go(Nat),
 Msg ==
   req(Cell,Goal) |
   rep(Reply) |
   cnf(Confirm)
```

```
value
 confirm: Cell × Net × Goal × (Cell -m-> Nat) -~-> Net
 confirm(c,n,(p,q),m) ≡ (s1,s2,q1,q2) = divide(m,q)...
   n'=snd(cnf(go(q1)),s1,n)...
   n''=snd(cnf(go(q2)),s2,n')...
   sndall(cnf(forget), dom m \ {s1,s2},n'')...
   pre c ∪ dom m ⊆ dom n ∧ card dom m > 1
```

This protocol may still be considered unsatisfactory. First, the time-to-completion of a given production goal in general does not depend on the quantity in a linear way, therefore dividing the quantities proportionally is not optimal. A solution is for the server to reply with a quantity-to-time map (instead of time for a given quantity), then base the client's decision on such functions. Second, the server may not like to accept the reduced goal. Perhaps it is negotiating with several clients and decides not to accept the goals below a certain minimum quantity. A solution is to introduce the fourth negotiation phase, letting the server accept the reduced goal. And so on.

4. CONCLUSIONS

The paper is about protocols for cooperation in a virtual manufacturing organization, how its members negotiate the implementation of a production goal. The goal describes the type of the product and its volume. The organization is a set of production cells: clients (they request implementation of a goal), servers (they carry out the implementation) or both. The cells communicate over a network with messages that contain: production goals, deadlines, calculated production times, positive and negative replies and confirmations etc. The exchange of messages takes places according to a protocol followed by each member. We described how to model such protocols explicitly, presented a number of possible design decisions and discussed how such decisions affect the behavior/performance of the whole organization. The models are described in a formal notation.

The production models in this paper follow (Janowski, Lugo and Zheng, 1999). From the design point of view, how to build an extended organization, related work includes (Vernadat, 1996) and (Schonsleben and Buchel,1998). From the operational point of view, how information technology can be used to support the extended enterprise, we refer to (Camarinha-Matos and others, 1997). From the practical point of view, how virtual organizations support and implement supply chains, we point to (Handfield and Nichols, 1999). The technical scope of this work is based on the protocols used for communication in distributed systems (Tannenbaum, 1998) and their formal models.

We have several plans to continue this work. First, we plan to implement the protocols described here in a prototype tool, for demonstration and further research. Second, we plan to carry out analysis of their behavior in a formal rather than informal way. Third, we intend to design an application-specific language with formal semantics, where such protocols can be conveniently described, analyzed, and translated into software. Fourth, we want to integrate the generator program for distributed production processes (Janowski, 2000) with one or more of the protocols described here. Finally, we would like to see how the protocols can also support competition (marketing) between members of a virtual manufacturing organization.

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. Handfield R and Nichols E. Supply Chain Management. Prentice Hall, 1999.
3. Janowski T. Distributed Production with Specification-Generated Processes. BASYS'2000, Berlin, Kluwer.
4. 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.
5. Schonsleben P and Buchel A. Organizing the Extended Enterprise. Chapman and Hall, 1998.
6. Tannenbaum A. Communication Networks, Prentice Hall, 1998.
7. The RAISE Method Group. The RAISE Specification Language. Prentice Hall, 1992.
8. Vernadat F. Enterprise Modeling and Integration. Chapman and Hall, 1996.