29

MODELING DISTRIBUTED PRODUCTION ENTERPRISES WITH XML

Tomasz Janowski
The United Nations University
International Institute for Software Technology
MACAU
tj@iist.unu.edu

Extensible Markup Language emerges as the leading notation to represent and exchange structured information. A distributed enterprise is a new paradigm for organizational design, based upon independent business entities which partly cooperate, by sharing their resources, skills and knowledge, and partly compete with each other. This paper presents an XML-based language to represent models of distributed production enterprises – the enterprises that coordinate production and delivery of products to its customers by means of several independent production cells. The models are minimal (built with the smallest possible number of concepts), formal (assigned formal semantics), technology-independent (semantics is expressed in abstract mathematical terms), and wide-spectrum (supporting descriptive and prescriptive modeling). The paper presents Document Type Definitions for representing the resources, operations, processes, customer and purchase orders, etc. that comprise the language, and describes their intended semantics using formal specifications.

1. INTRODUCTION

Industrial enterprises and distributed industrial enterprises in particular require various kinds of software, hardware and communication technologies to support their operations. Given the mission-critical role of enterprise systems and their potential to help but also disrupt business operations, putting such technologies together is a complex and difficult task involving numerous technical, financial and managerial challenges (Davenport, 2000). Yet technology that does not support the business objectives, enforces changes to the time-proven business practices, and constraints the responses of the enterprise to dynamic changes in its environment, is of little value, or worse. The challenge for developing and operating enterprise systems is to ensure their integration with organizational goals and processes.

Developing such systems from unstructured, informal requirements is clearly unsatisfactory. Apart from the usual complaints that such requirements are often ambiguous or even contradictory, this would not support the objective of integration, with business processes and with other systems supporting them. Neither would it help to validate the requirements before the system is built and deployed. An alternative is to formulate requirements based on enterprise models. Fortunately,

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: 10.1007/978-0-387-35585-6_68

L. M. Camarinha-Matos (ed.), *Collaborative Business Ecosystems and Virtual Enterprises*

several public domain enterprise frameworks like ARIS (Scheer, 1998), CIMOSA (AMICE, 1993), IMEE (Kim, Weston and Woo, 2001), PERA (Williams, 1994) and others (see (Vernadat, 1996) for an overview), offer the choice of languages, methods and tools to help design such models. Although such frameworks are primarily aimed at analyzing and improving enterprise business operations, not at capturing requirements and then developing software to satisfy such requirements, some of them can be linked directly or indirectly to software development. IMEE, for instance, includes modeling with UML (Booch, Rumbaugh and Jacobson, 1999). There are also examples of concrete implementations based on enterprise frameworks, for instance ARIS and the R/3 software from SAP (Appelrath and Ritter, 2000). However, software development from enterprise models will remain a largely informal and therefore mostly manual and unsupported task as long as enterprise models are themselves informal.

Formalization of enterprise frameworks is a major challenge and there is currently little experience how to approach it. This is despite the fact that techniques to assign modeling languages with formal semantics are well known. The purpose of this paper is to gather some experience in formal definition of a prototype language for modeling distributed production enterprises. Its syntax is defined in XML - Extensible Markup Language (WWW Consortium, 2000). Its semantics defines a mapping from XML to formal specifications expressed with mathematical functions, sets, lists, maps, etc. Different semantic definitions correspond to different kinds of entities comprising enterprise models. For instance, semantics of a production operation is a function from workstations to triples of products, numbers and maps from products to numbers: the product and volume the workstation should manufacture and also which part of this volume should be forwarded to which workstation for further processing. The language is small by definition - built with the smallest possible number of modeling concepts - but is meant to be extensible. For instance, the basic language does not model the maintenance of workstations, but we show how to add this feature later, by extending syntax and semantics. Also by definition, the language includes redundancy, meaning more than one way to define its entities. An example is a process defined descriptively, as a sequence of operations, versus a process defined prescriptively, as a production goal. Due to its formal semantics, the language allows to verify that the former satisfies the latter.

The rest of this paper is as follows. Section 2 explains the concepts and terminology. Section 3 defines the syntax of the language using XML. Section 4 shows how to define formal semantics for the language, using a formal specification language RSL (RAISE Language Group, 1992). Section 4 demonstrates how the languages could be extended. Section 5 contains a discussion and draws some conclusions.

2. CONCEPTS AND TERMINOLOGY

Before presenting the syntax of the language, we explain the entities the language describes, and what such entities are meant to represent.

By an enterprise we mean an independent business entity that contains a number of resources to manufacture and subsequently deliver to its customers various kinds of products. We call this entity a production cell. Products are tangible goods like phones or shovels, built from various quantities of sub-products like handsets or shafts, all of which are products in their own right. Each product is described with two attributes: a number encoding its storage requirements and a function from products to numbers, representing all sub-products with quantities. The resources represent how the cell is able to store (warehouse), deliver (stocks) and manufacture (shopfloor) products. The shopfloor consists of workstations, each able to manufacture various quantities of products within a shift, connected by a transportation system. A production cell accepts customer orders to deliver various quantities of products within a deadline, and implements them with production processes. A process is a sequence of operations. An operation allocates each workstation with a task to manufacture a number of products of one kind and then forward parts of this volume to other workstations for further processing. The workstations process their assignments concurrently within a shift. A cell can handle several customer orders simultaneously. Processes that implement these orders execute concurrently, using a priority-based scheduler to resolve conflicts for the shared resources. When a cell lacks the resources to implement a given order, it may delegate part of the order to other cells, forming a distributed production system.

3. SYNTAX OF ENTERPRISE MODELS

Below is an XML Document Type Definition to represent product information. A valid document must contain a non-empty list of product elements. A product has an attribute that holds its unique identifier, a textual description and two elements: size and a list of sub-products. A sub-product element has one attribute, a product identifier, and holds a number of items as plain text.

```
<!DOCTYPE bill[
  <!ELEMENT bill(product+)>
  <!ELEMENT product(#PCDATA, size, sub*)>
  <!ATTLIST product id ID #REQUIRED>
  <!ELEMENT size(#PCDATA)>
  <!ELEMENT sub(#PCDATA)>
  <!ATTLIST sub product IDREF #IMPLIED>
]>
```

Here is a fragment of the XML file representing a shovel product (Vollmann, 1992). A shovel contains a top handle and four rivets, a top handle contains a handle, etc.

```
<bill>
  <product id="p1"> shovel <size> 151 </size>
    <sub product="p2"> 1 </sub>
    <sub product="p3"> 4 </sub> … </product>
  <product id="p2"> top_handle <size> 18 </size>
    <sub product="p4"> 1 </sub> … </product>
  <product id="p3"> rivet <size> 2 </size> </product>
</bill>
```

A production cell contains product stocks (which change during production), production resources (which enable production but do not change), processes which carry out production using the resources to satisfy customer orders, and the orders received but not yet assigned any process. Each cell is assigned a unique identifier.

```
<!DOCTYPE cell[
  <!ELEMENT cell(stocks, resources, process*, order*)>
  <!ATTLIST cell id ID #REQUIRED> …
]>
```

Product stocks describe how many products of different kinds the cell has available. Resources include: storage space, the maximum number of products weighted by their storage requirements; shopfloor, a set of workstations each able to assemble a number of products of different kinds within a shift; and transport, the maximum number of products to be transferred between any two workstations within a shift.

```
<!ELEMENT stocks(stock*)>
<!ELEMENT stock(#PCDATA)>
<!ATTLIST stock product IDREF #REQUIRED>
<!ELEMENT resources(storage, shopfloor, transport)>
<!ELEMENT storage(#PCDATA)>
<!ELEMENT shopfloor(workstation*)>
<!ELEMENT workstation(assembly+)>
<!ATTLIST workstation id ID #REQUIRED>
<!ELEMENT assembly(#PCDATA)>
<!ATTLIST assembly product IDREF #REQUIRED>
<!ELEMENT transport(from-to*)>
<!ELEMENT from-to(#PCDATA)>
<!ATTLIST from-to from IDREF #REQUIRED to IDREF #REQUIRED>
```

A customer order describes how may products of different kinds the customer needs and optionally, the latest time (number of shifts) to receive them.

```
<!ELEMENT order(item*, deadline)>
<!ELEMENT item(#PCDATA)> <!ATTLIST item product IDREF #REQUIRED>
<!ELEMENT deadline(#PCDATA)>
```

Each process is assigned a priority, an implementation (sequence of operations), an "insource" element and an "outsource" element. An operation describes the allocation of work to workstations: the product and volume it should manufacture and which part of this volume should be forwarded to which other workstation for further processing. The "insource" element describes the customer order the process implements and the customer who issued this order. The "outsource" element contains a number of auxiliary orders that are necessary for the process to satisfy its original order, but cannot be fulfilled within the cell itself. Such orders are outsourced for implementation in external production cells. Here are the definitions:

```
<!ELEMENT process(priority, insrc, outsrc*, operation*)>
<!ELEMENT priority(#PCDATA)>
<!ELEMENT insrc(#PCDATA)>
<!ATTLIST insrc customer IDREF #REQUIRED product IDREF #REQUIRED>
<!ELEMENT outsrc(#PCDATA)>
<!ATTLIST outsrc supplier IDREF #REQUIRED product IDREF #REQUIRED>
<!ELEMENT operation(work*)>
<!ELEMENT work(vol, forward*)>
```

```
<!ATTLIST work workstation IDREF #REQUIRED>
<!ATTLIST work product IDREF #REQUIRED>
<!ELEMENT vol(#PCDATA)>
<!ELEMENT forward(vol)>
<!ATTLIST forward workstation IDREF #REQUIRED>
```

The following file describes a cell `c1` with stocks and resources to produce a shovel product. The cell contains two workstations: `w1` can assemble `150` shovels per shift, and `w2` can assemble `100` shovels or `150` top handles per shift. The transportation system can move up to `5000` storage units from `w2` to `w1` per shift. The cell executes a single process that implements a customer order for `500` shovels (without deadline) issued by the customer `c2`, provided the supplier `c3` can deliver `2000` rivets. We only present the first operation which assigns `w1` to assemble `150` shovels and send them all to stock, and `w2` to assemble `150` top handles and send half to `w1` and half to stock.

```
<cell id="c1">
  <stocks>
    <stock product="p1"> 0 </stock>
    <stock product="p2"> 500 </stock> … </stocks>
  <resources >
    <storage> 200000 </storage>
    <shopfloor>
      <workstation id="w1">
        <assembly product="p1"> 150 </assembly> </workstation>
      <workstation id="w2">
        <assembly product="p1"> 100 </assembly>
        <assembly product="p2"> 150 </assembly> </workstation>
    </shopfloor>
    <transport>
      <from-to from="w2" to="w1"> 5000 </from-to> </transport>
  </resources>
  <process>
    <priority> 1 </priority>
    <insrc customer="c2" product="p1"> 500 <\insrc>
    <outsrc supplier="c3" product="p3"> 2000 <\outsrc>
    <operation>
      <work workstation="w1" product="p1"> <vol> 150 </vol> </work>
      <work workstation="w2" product="p2">
        <vol> 150 </vol> <forward to="w1"> 75 </forward> </work>
    </operation> … </process>
</cell>
```

Here is a schematic definition of the supplier cell `c3`. The cell contains a process that implements the order from `c1` (now a customer cell) to deliver `2000` rivets. As `c3` can deliver the order directly from its stock, the process contains no operations.

```
<cell id="c3">
  <stocks> <stock product="p3"> 10000 </stock> </stocks>
  <resources> … </resources>
  <process> …
    <insrc customer="c1" product="p3"> 2000 <\insrc>
  </process>
</cell>
```

`c1` and `c3` constitute together a distributed production system.

4. SEMANTICS OF ENTERPRISE MODELS

The definitions in Section 3 determine what kind of expressions the language allows to write. Now we demonstrate how to assign such expressions with formal semantics, using the RAISE Specification Language.

Products are described using the type `Product` and two functions on this type: `size` (returns a number) and `sub` (returns a map of products). We define three axioms to ensure the functions are "reasonable": `size` never returns zero (`size(p)>0`), the sub-product relation is acyclic (`¬issub(p,p)`, `issub` decides if one product is a sub-product of another) and `sub` never returns a map with a zero value (`size(p)>0`).

```
type
 Product,
 Size = Nat,
 Quantity = Nat,
 Products =
  Product -m->Quantity
```

```
value
 size: Product → Size,
 sub: Product → Products
axiom
 ∀ p:Product • size(p)>0 ∧ ¬issub(p,p),
 ∀ p,q:Product • q ∈ bill(p) ⇒ bill(p)(q)>0
```

A cell contains product stocks and the resources to store, assemble and transport products, represented as four functions on the type `Cell`. An axiom requires that the number of products in the cell (weighted by their size) does not exceed its capacity.

```
type
 Cell, Workstation
value
 stocks: Cell → Products,
 storage: Cell → Size,
 shopfloor: Cell × Workstation → Products,
 transport: Cell → Workstation × Workstation → Size
axiom
 ∀ c: Cell • storage(c) ≥
   sum([p → size(p)*stocks(c)(p) | p:Product :- p ∈ sub(p)])
```

A cell also contains a list of processes and a set of received customer orders. Each process contains an original customer order, a set of purchase orders (if any) and a list of operations. Each order contains the name of the customer/supplier cell, the items ordered with product types and volumes, and the deadline (number of shifts).

```
type
 Process::
  insrc: Order
  outsrc: Order-set
  impl: Operation-list
```

```
type
 Order::
  cell: Cell
  item: Products
  deadline: Nat
```

```
value
 processes:
   Cell → Process-list,
 orders: Cell → Order-set
```

An operation is a map from workstations to work assignments, which decide what product should be produced, how many items and how to partition the result among other workstations. A workstation `ws` can only forward its products to `ws'` provided `product(ws')` is a sub-product of `product(ws)`. The execution of an operation, if

the cell has enough resources, decreases the stocks for the products consumed and increases the stocks for the products produced but not forwarded.

```
type
 Work::
  vol: Quantity
  product: Product
  forward:
   Workstation -m-> Quantity
value
 iswf: Work → Bool
 iswf(w) ≡ vol(w)>0 ∧
  0 ∉rng forward(w) ∧
  sum(forward(w))≤vol(w)
```

```
type
 Operation = Workstation -m-> Work
value
 iswf: Operation → Bool
 iswf(op) ≡ ∀ ws:Workstation •
  ws ∈ dom op ⇒
  dom forward(op(ws)) ⊆ dom op ∧ … ,
 exec: Operation × Cell → Cell
 exec(op,c) ≡ deliver(fromstock(op),
              receive(tostock(op),c))
  pre enough(op,c)
```

A process is executed on a cell by executing all its operations in the `impl` list, provided the cell has enough resources for them (taking into account how they change the stocks). A process is correct for a cell if it satisfies its deadline and, after receiving all the purchase orders in `outsrc`, the cell has enough resources for its execution and this execution creates enough products to satisfy the `insrc` order.

```
value
 correct: Process × Cell → Bool
 correct(p,c) ≡ enough(impl(p),receive(outsrc(p),c)) ∧
   len impl(p) ≤ deadline(insrc(p)) ∧
   (∀ p:Product • p ∈ dom items(insrc(p)) ⇒
     stocks(exec(impl(p),c))(p) ≥ items(insrc(p))(p))
```

5. EXTENSIONS OF SYNTAX/SEMANTICS

The language would likely require extensions to accommodate other aspects of production than those included till now. For example, suppose each workstation has to be maintained after it has carried out production for a certain number of shifts. To represent this, we extend the XML syntax of the workstation with two numbers, `max` and `done`. We also extend the semantics with two corresponding functions and an axiom: if `enough(op,c)` then `done < max` for any workstation in the operation `op`, and subsequent execution of `op` on `c` increments the value of `done` by one.

```
<!ELEMENT workstation
  (assembly+,max,done)>
<!ELEMENT max(#PCDATA)>
<!ELEMENT done(#PCDATA)>
```

```
value
 max,done: Workstation × Cell → Nat
axiom
 ∀ op:Operation, c:Cell, w:Workstation •
  enough(op,c) ∧ w ∈ dom op ⇒
  done(w,c) < max(w,c) ∧
  done(w,exec(op,c)) = done(w,c) + 1
```

6. CONCLUSIONS

Few people would argue today about the need to formalize programming languages. While such a language describes the end-result of development, concrete enough to describe a computation, a modeling language is a vehicle for expressing design decisions. It must be able to describe its entities at multiple levels, and needs formal semantics to make sure that an abstract description (e.g. an order) and a concrete one (e.g. a process) refer to the same entity (that the process satisfies the order).

The paper presents our experience in defining and formalizing a prototype language for modeling distributed production systems. Based on a small number of concepts, the syntax of the language was defined in XML and semantics described with formal specifications in RAISE. For a full definition of semantics, without XML syntax or extensions, see (Janowski and Ojo, 2001). Our approach compares to Business Process Modeling Language (BPMI, 2001) or Electronic Business XML (ebXML). The main differences are: the language in this paper is concrete (not a meta-language), includes both abstract and concrete process descriptions (ebXML describes public interface and BPML private implementation) and is assigned formal semantics. On the other hand, Process Specification Language (Schlenoff and others, 2001) is also assigned formal semantics, but using Knowledge Interchange Format (KIF). The advantage of RAISE to express semantics is to facilitate rigorous development of software, in particular development of the engineering tools for the language. We also demonstrated how the language could be extended, to address the issues not predicted at the time of its design. Fortunately, XML is extensible by definition, although semantic extensions are more involved. Our current work is to validate and implement the language, and develop a method for its extension.

6. REFERENCES

1. AMICE Consortium. "CIMOSA: Open System Architecture for CIM". Springer Verlag, 1993.
2. H.J. Appelrath and J. Ritter. "SAP R/3 Implementation, Methods and Tools". Springer, 2000.
3. G. Booch, J. Rumbaugh and I. Jacobson. "The Unified Modelling Language User Guide". Addison Wesley, 1999.
4. Business Process Management Initiative. "Business Process Modeling Language Specification". http://www.bpmi.org, 2001.
5. T.H. Davenport. "Mission Critical, Realizing the Promise of Enterprises Systems". Harvard Business School Press, 2000.
6. T. Janowski and A. Ojo. "Formalizing Feasibility and Correctness of Distributed Business Processes". 2nd Workshop on Conceptual Modeling Approaches to E-Business. Springer, 2002.
7. C. Kim, R. Weston, H. Woo. "Development of an integrated methodology for enterprise engineering". International Journal of Computer Integrated Engineering, vol. 14, no. 5, 2001.
8. The RAISE Language Group. "The RAISE Specification Language". Prentice Hall, 1992.
9. A.W. Scheer. "ARIS – Business Process Frameworks". Springer Verlag, 1998.
10. C. Schlenoff and others. "The Process Specification Language, Overview and Version 1 Specification". http://www.mel.nist.gov/psl/pubs/PSL1.0/paper.doc, 2000.
11. F. Vernadat. "Enterprise Modeling and Integration". Chapman and Hall. 1996.
12. T.E. Vollmann. "Manufacturing, Planning and Control Systems". Irwin, 1992.
13. T.J. Williams. "The Purdue Enterprise Reference Architecture". Computers in Industry, vol. 24, no. 2-3, 1994.
14. WWW Consortium. "Extensible Markup Language (XML), 1.0, Second Edition". http://www.w3c.org/TR/2000/REC-xml-20001006. W3C Recommendation, 2000.