Modeling a Manufacturing System using Object-Oriented approach

V. CARCHIOLO, A. LONGHEU, M. MALGERI
Istituto di informatica e Telecomunicazioni
Università di Catania
V.le A. Doria, 6 – 95100 Catania
ITALY

Abstract: Current manufacturing systems have a very structured production flow, expecially when high complexity and precision is required, such as in semiconductor devices manifacturing. On the other hand, rapid changes in production and in market requirements may occur, hence great flexibility is also essential. So, a major challenge is the development of a logical model for such systems, satisfying these needs and also balancing between high abstraction and "hard-wiring" to the real production environment.
Here we propose such a model applying object oriented databases techniques, which are a powerful technique, well suited to manufacturing system environment, thanks to hierarchies, versioning and evolution management. We first define objects characteristics, presenting different hierarchies (aggregation and constitutional), then we consider core objects, process and product, also introducing several classes of operations used to describe production flow.

Keywords: Databases, Manufacturing systems, Information systems Object-oriented programming

1 Introduction.

Modern industrial processes are highly structured and automated, so when managing them it is necessary to have a ***logical production model*** which may be quite complex, depending on the number and type of requirements of the product being manufactured. A classical example is the integrated circuits (IC) area: due to both increasing miniaturization and rapid obsolescence of devices, it is required both a sophisticated and an open (flexible) model, capable of rapidly adjusting production, also in reply to market demand, in terms of both quantity of products and new technologies and product development (Just in Time production).
A generally accepted reference model for Computer Integrated Manufacturing (CIM) systems models a factory as a hierarchy of controls, with different responsibilities at each level [1]. Inside CIM different activities can be located, from CAE to CAQ (computer aided quality), to CAD, CAM and CAPP (computer aided process planning) [2], in which operations sequence for manufacturing tasks is established.
In this context, it is important to consider the connection between the model and the available ***production environment***, i.e. both architecture and applications, since the model is implemented using them (directly linked with machines executing the sequence); this interaction has to fully exploit architecture-applications pair features, but without being too deep, as the model may be significantly influenced (hence, limited) by the pair.
A solution is to implement the model with extremely general pairs, assuring greater model expandibility; though efficiency problems would arise due to the "distance" between the pair and the model. Alternatively, pair can be completely customized for the model; this assures high efficiency and compatibility, but also mean limited expandibility and high costs (e.g. software needs to be created ad-hoc), so a trade-off between generality-expandibility and specificity-efficiency has to be found when choosing the pair.
In addition, the model must be designed by ***abstracting*** from the pair, representing production process without constraints it imposes; adopting an object-oriented approach assures this, also remaining close to real system, generally hierarchical.
In this paper, we present a model for a manufacturing system applying object-oriented approach. We use OO database techniques to model several aspects of CIM, such as aggregation and constitutional hierarchies, inheritance and versioning. The model presented is a logical one, as opposed to a real model which is an implementation of the former by means of architecture-applications pair. This dichotomy may significantly alter correct use of the model, i.e., coherence problems arise; however, here we will not consider implementation issues.
In section 2 we describe general object modeling and its characteristics, while in section 3 we mention off-line and on-line issues. In section 4 we introduce hierarchies, then in section 5 we focus on process and product objects, then considering their states in section 6 and finally presenting conclusions in section 7.

2 Object Modeling and Characteristics.

The first step when creating a model is to identify ***objects***, each of which can be defined as an abstraction of a part of the reality being considered. Being this definition highly general, a generic object can represent any entity (e.g., human resources), it may partially overlap with others (several objects may have common features) and may or not have a physical counterpart (this becomes more evident as model description descends from a high abstraction level towards real objects). An object can also be used to group others together; this situation, which may also include classical OO inheritance [3], determines aggregation hierarchy. Another case is that of complex objects [4], i.e. consisting of other objects; such links creates constitutional hierarchies (also known as aggregation hierarchies [3][4]). Finally, several objects can make up a single logical entity, leading to composite objects [4] (constitutional hierarchies may also include these objects). Aggregation, complex and composite objects, all present in the CIM context, are essential to capture production cycle features.

Identifying an object also means choosing which production aspects must be considered as objects. Simple criteria are to select the most significant entities, those that require an identity of their own, and those that cannot be associated with any other objects. Otherwise, entities can be simply modeled as attribute of others objects.
Then objects have to be analyzed separately to establish their features, properties (attributes), states and constraints (determining criteria for changing in state and properties).
More specifically, considering an object's ***attribute*** means to define its meaning (semantics), type and allowed values, as well as any rules, i.e. conditions on existence (whether attribute is optional or mandatory), on values (e.g. max length in characters for text type) or involving other fields (e.g. "IF *field1* exists AND *field2*='x' then *field3*= 70") [11].
The last two rules may require more complex mechanisms (e.g., conditions involving more attributes may need a language to define and test rules).
We also note that some of object's attributes are merely used to support implementation (e.g. a "last access date" attribute); we do not consider such attributes here.
Considering ***states***, we have mainly:
- creation, i.e. how an object is created (from scratch or by copying, even from a different class with specific casting rules), and what are inheritance constraints (from parent object) and/or integrity constraints (when the object is copied);
- updating, i.e. if and in what conditions it is possible, and what this means for the object and all its descendants (change propagation);
- deletion (feasibility, implications for descendants).

Some objects may also have other states, so each will have its own appropriate finite state machine (FSM).
It should also be pointed out that if an object ***X*** is complex or composite, its state is determined both by its attributes and by states of objects it contains or is associated with.
Finally, an object can have associated ***methods***. However, we consider them as part of the object implementation, so here they will not be taken into further consideration.

3 Off-Line vs On-Line Issues.

Both in objects and their corresponding relations, there are aspects that can be classified as static, regarding production flow definition, and others dynamic, related to flow execution. "Dynamic" property crosses through hierarchies, although it becomes more evident at lower levels of abstraction, i.e. for objects representing basic production units, since they undergo real flow operations, leading to end product.
Dynamic nature does not depend on the modifications an object undergoes, i.e. it is not more dynamic whether it undergoes variations more frequently than others. An example of an on-line aspect (lot sensitivity) is given in section 6.2.

4 Hierarchies.

4.1 Aggregation hierarchies.

The root node is the highest level of abstraction required in the model. It may, for instance, indicate the plant where production takes place, or the head office, the various sites being its descendants. In general, there can be any number of both levels and nodes in a level. The model considered here, however, concerns the production process, so the hierarchy should represent just features strictly inherent: e.g. it will not include nodes for economic description of the plant, or nodes to model the hierarchy of personnel.
A possible elementary aggregation hierarchy is shown in Fig.1. At the top level we have the ***facility*** object, indicating the plant where the production takes place.
To better understanding following objects, it is convenient first of all to consider the ***product***, core of the production process. This object represents the sequence of operations to be given to real machines in order to make end products. This final product is instead modeled by the leaf object in the tree.
Its parent, named ***lot***, groups a set of final products viewed as a single production unit, e.g. for economic reasons (products are sold in lots), or because of the type of product (a food product could be a snack, and the packet would be the lot).
The father node for product is the ***process***, which groups all products having the same sequence of operations, and higher up still we have the ***process family***, a set of processes sharing some feature. For example, if the model refers to IC production, a family could contain all electrical devices with "similar" manufacturing process (e.g. MOS or BJT) or, in motor vehicle production, families could represent cars, vans, and trucks.

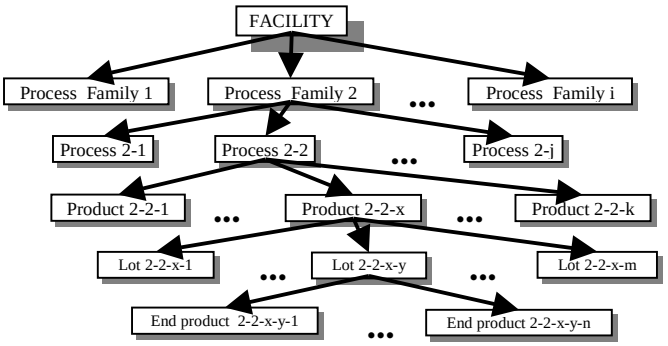

Fig. 1

The hierarchy we have presented may vary (as the grouping criteria may change), there may be different number of levels between the root and the leaves, and the root and leaves may express a varying degree of abstraction (the root may coincide with the product or with a group of production plants, or the highest level of detail required may be just the lot); in the following, we refer to Fig. 1.
As far as inheritance is concerned, it is only present in the process-product case (see section 5.5), or at most with lot-end product; with facility-process family, process family-process, and product-lot, a weaker semantics is adopted, i.e. these are considered simple logical aggregations (similar to association present in [4]).
There are other inheritance mechanisms [5],[6], as well as multiple inheritance [4], which will not be used here as we wish to limit model complexity in favour of flexibility and ease of management.

4.2 Constitutional Hierarchies.

A product consists of a sequence of ***operations***. More precisely, a product has a series of attributes, some of which are simple (integer, string, or even array or set of basic types), and others are expressed in terms of objects. One of these is the sequence (***OpSeq***) attribute, which is composite in that it consists of an ordered set of "operation" objects. A product is

thus a complex object defined in terms of composite objects.
In addition, at least one more level of detail may be generally required, e.g. because the operation is too abstract, hence we introduce (Fig.2) the sub-operation (***Step***) which is more closely linked to the real machines and/or is more atomic; therefore the operation is modeled as a sequence of steps (***StepSeq***). In particularly complex production systems, we may have more levels of detail.

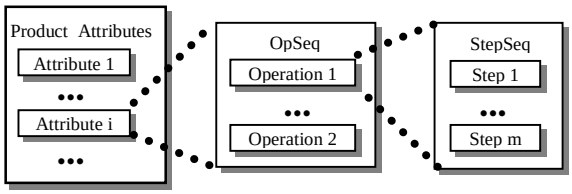

Fig.2

Besides, an operation (or step), representing a physical action, could appear in several OpSeqs (or StepSeqs), hence a distinction must be made when using separate copies of the same object (multiple instances) or not (single instance).
Having separate instances for an object X, means that they have common features (inherited from the parent object), and others specific to each instance. On the other hand, using a single instance object implies the presence of a *library* of such objects, creating a "link" any time we need that object. So, controls and behaviour required will change depending on the relation needed (e.g. when deleting, see section 6.3).
These concepts are similar to the "composite reference" introduced in [7], in which single instance coincides with the *shared* case, and multiple with the *exclusive* case. The same paper also introduces the concept of dependence independence which, combined with the previous property, leads to four possibilities. In this context, single instance is considered dependent, while deleting an instance of multiple instances object does not involve the others.
Finally, operations and steps could be subclasses of a single class; inheritance hierarchy is not generally related with constitutional or aggregation hierarchies, rather it is transversal.

5 Process and Product.

In this paper we give a precise role to process object and to its link with the product. When, indeed, several products only differ in the values of certain parameters, or for some operation of sequence, it is desirable to avoid the tedious and error-prone work of re-creating what is common among them.
A process meets this need, since it is a general schema for such similar products: creating it means to set all common features for this group of products, so they will automatically inherit all that has been defined inside their father process. Then, to define completely each product, it will be required just to specify some operations and/or missing parameters.
Hence, since creating a process implies inserting common operations, it is important to identify all possible categories of operations, in order to establish all that can be set inside a process and what must be defined inside its products children.
Since a process (or product) sequence is somewhat similar to a flow diagram, we located some ***operations classes***, each having a corresponding block shape in flow diagram context.

5.1 Production Operations.

The first class, named ***production operations***, represents all operations containing "instructions" (i.e., steps) about how to work lots in order to make end products. This operation is specified in several definition steps (levels).
First level is operation ***type***, which groups all operations with some precisely defined steps, essential for that "type".
To give a clearer idea, let us consider a mask operation, used in chip production and consisting in creating device's geometry by means of partial photo-exposure of the silicon (covered by a "mask") and then removing unexposed material. This operation can be assumed to consist of at least two fixed steps, i.e. the photo-exposure (the mask will represent a simple parameter) followed by a measurement to test it.
Up to this definition level, this operation can be represented as in Fig. 3 a), where a certain number of possible steps (1...x, unknown at this stage) appear before the masking step, followed by the latter, in which the mask does not appear (so it is a "missing parameter" in the previously defined sense, and has to be specified separately for each product). There then may be other steps, up to the measurement (whose completion e.g. depends on the value of the parameter (*Func (mask)*), and finally, another potential group of steps.
In the second phase the number and type of all the groups of steps 1…x, 1…y and 1…z are chosen, together with the exact form of *Func*, but all the parameters of the steps are not yet specified (thus passing from Fig. 3 a) to Fig. 3 b)). This level is called ***template***, since it completely defines the structure of each operation. Finally, it is possible to set any remaining parameter and calculate any related expression (*Func*), so we get to the last level (Fig. 3 c)), called ***default*** (since it represents a possible set of values for a template), at which an operation is completely specified.

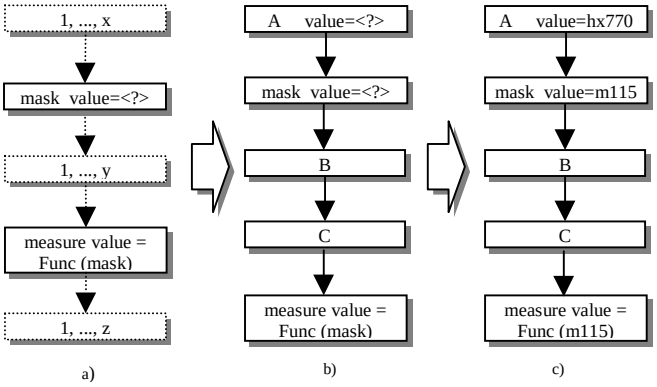

Fig.3

For each operation type (or template) a group of available templates (respectively, defaults) can be defined.
Each operation is defined following these three steps in order. Here we distinguish "process" operations, where these phases are all completed inside the process (such operations are then independent from the product), and "product" operations, where the third phase is specified when creating a product.
This cannot be done before, as parameters values depend on the products, nor can it be done afterwards: a product has indeed to be specified completely since lots will be physically processed following its operation sequence (this is referred to as product "instantiability").
What has been set at the first two levels thus only needs to be specified once in the process and can then be inherited by all children products. This makes it possible to create new products quickly, thus reducing time-to-market and then cost.
All presented up to now can also be extended to the step level (hence, *step a* in Fig. 3 b) is a template, and in Fig. 3 c) there

is a corresponding default); a template step can only appear in a template operation, whereas a default step can appear both in a template operation and in a default operation, (a template operation must have at least one template step, while a default operation consists of default steps only).

5.2 Optionality operations. Expressions.

The second class of operations has the same role of control flow statements in a flow diagram.

In a sequence of operations, indeed, it is necessary to specify one or more alternative paths, thus having a decision block that, depending on a condition ***C***, based on one or more expressions, executes different sets of operations (paths) or assigns different parameters to the same operation.

The decision block represents the class of ***optionality operations***, which do not imply any physical operations (no steps inside), as it only contains a condition.

We have two operations levels for this class, one in which the structure of the condition is set (logically equivalent to a template), and one in which all the current values of the variables in the conditions are known (similar to a default), so conditions can be calculated and a path is chosen.

Like the previous class, optionality operations can generally be "process-dependent", i.e. completed inside the process, or parameters values will only be known in product, so only the structure of the condition is set in the process.

A generic condition can be viewed as an *if…then…else* construct or a *case…switch*, as it can contain one or more ***expressions***, each of which returns one of a set of values (so an expression is not necessarily binary).

To make up an expression, classical operators (+, -, *, /, AND, OR, NOT, as well as <, >, <>, =) are available and act on ***environment variables***, i.e. on all information about current hierarchical sub-tree (e.g. if the condition is set in a product, the environment consists of all the attributes of the parent process, and grandparent process family). Environment can also include information from the whole hierarchy, as well as user-defined variables.

Besides, an expression can be ***off-line*** or ***on-line***. In the former case, its evaluation depends only on definition of objects in Fig. 1, hence is independent of the production flow, while on-line conditions can be evaluated just when production is being executed, e.g. because they depend on the number of lots being processed at a certain time.

In an OpSeq defined in a process there may be both on-line and off-line conditions, whereas in a product only on-line can "survive" (anyway evaluated during production): all off-line must be solved and replaced in the OpSeq for that product with the path associated with result value.

Off-line conditions represent a powerful way to generalise, since in a process we can specify different sub-sequences in order to group more products; products then will differ not only for parameter values, but even for these sub-sequences.

Finally, a condition can be ***non-deterministic*** when the choice is human-dependent. This is modeled by introducing ***users*** and ***user classes*** (groups of users with specific rights and/or limitations). Non-determinism (which may be both off-line and on-line) is implemented by adding users and classes to the environment variables, so an expression like "*@admin-class*" means that the path will be chosen by any users with *administrator* privileges during the creation of the object.

An expression can also appear in a production operation (as the *Func* above). Lastly we can have also optionality steps, which can only appear in production operations (optionality operations contain no steps).

5.3 Goto and Loop.

It is also possible to have jumps in an OpSeq or in a StepSeq. A jump can be conditional or not, using optionality operations. When going to previous operations, a loop is created. Composing all these options, we may have *repeat-until*, *for,* and *while-do* or simple *goto* constructs (*goto* can be used to change a sequence temporarily).

5.4 Monitoring Operations.

This class of operations is used to check or provide, during production (execution of the sequence), the status of parameters of interest, or an expression based on them. As optionality class, these operations contain no steps, while monitoring steps can be defined.

5.5 Putting it all together.

A comprehensive example of a process is shown in Fig. 4; in the OpSeq, entirely created during process definition, we have a portion containing an optionality operation with a condition *C1*, here supposed deterministic, binary and on-line.

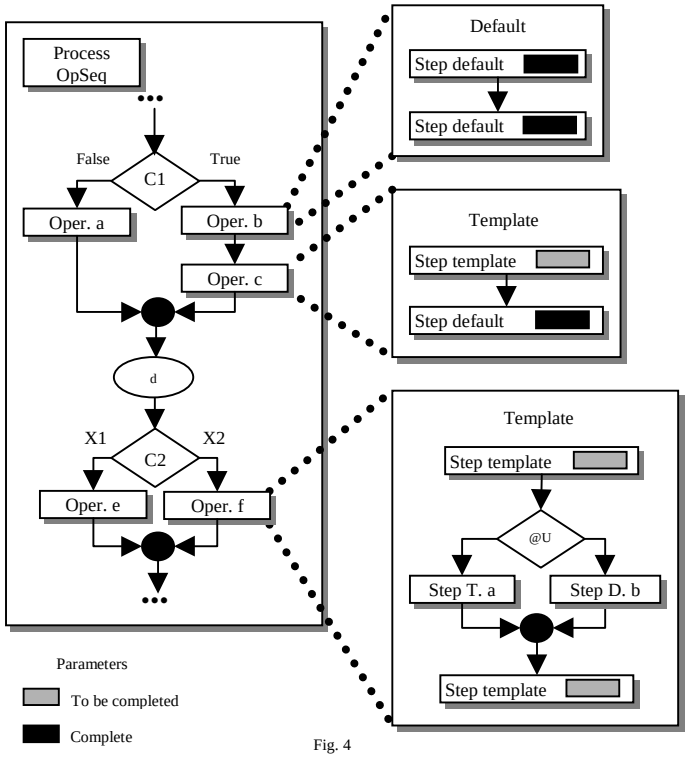

Fig. 4

From this operation we have two different paths, one with operation *a*, and the second with *b*, assumed to be process-dependent, and *c*, product-dependent. This means that type, template and then default for *b* are all specified in the process context, (in Fig. 4, *b* is directly linked to its default, e.g. containing only production steps). As operation *c* is product-dependent (i.e. the default level is specified in the product), it has a template, which possess at least one template step. *d* is a monitoring operation. The subsequent optionality operation contains a binary, deterministic, off-line condition *C2* with two paths, operations *e* and *f*, the latter being product-dependent, with a template containing a non-deterministic, off-line optionality step (hence the condition ***@U***). whose choices are a template, *a*, and a default, *b*.

When a product child of this process is created, inheritance mechanism offers a partially complete product, with at most

template production operations (product-dependent) to be specified at default level, with completed default operations (process-dependent) that must not be altered (otherwise, inheritance would be violated), and options and/or monitoring operations containing expressions. Inside expressions (which may also be present in production operations), there will be environment variables already known (e.g. a process attribute), variables known in product context (e.g. product name), and variables to be completed manually inside product. Having finished these three steps, all off-line expressions will have to return a value such that the templates can be completed, or the path to follow in an optionality operation can be chosen, or the desired check (in the monitoring operation case) can be made. In the on-line expressions, missing parameters will only be known when production will run.

By this hypothesis, a possible product child of the process defined above is shown in Fig. 5, where it is assumed that *C2* returns *X2* and that the user *@U* has created the product choosing the right-hand side branch in the relative condition.

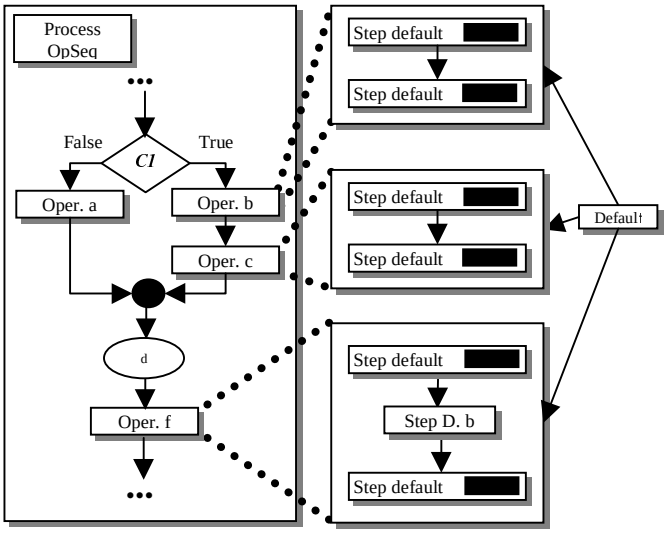

Fig. 5

6 States and FSM.

6.1 Creation.

In the following, we consider states for process and product. As stated in section 2, the first state is ***creation***, in which a new instance for an object is created. This can be done from scratch or by copying from other objects, having some conflicts in the second case.

Indeed, when a product is copied, pasting the copied product inside the same process, a conflict that arises concerns the logical identity of the new product (it will be the logically the same as the original until it is modified). If the copy is made from another process (from the same or different process family), the copied product is invalid (cannot be used) until it meets inheritance constraints of the destination process. Finally, we can copy a process and pasting it as a product. In this case, the process also has to become a product, following a procedure like that leading from Fig. 4 to Fig. 5.

If a process is to be created by copying, problems are similar, and in some cases even simpler, due to the weakness here established for process family semantics [11]. Copy issues are somewhat similar to instance migration in OO databases [8].

6.2 – Updating.

The second state is ***updating***, in which it is possible to modify the object, i.e. its attributes hence its associated objects, as specified in the constitutional hierarchy; updating a process includes the insertion, deletion and modification of each single operation (and/or step), as well as alteration of their order. It is therefore necessary to analyze the modifications that can be made at the lowest level (step), propagating them up to the operation level and then to the process or product.

Considering different steps classes, i.e. production, optionality and monitoring, allowed variations for steps are *template→ template*, where the internal structure is varied, *default→ default* where just parameters are altered, and finally *template→ default* and *default→ template*, the former being a specification and the latter a generalization task. Modifying options or monitoring steps means altering their expressions and/or conditions, eventually changing determinism and/or off-line/on-line properties.

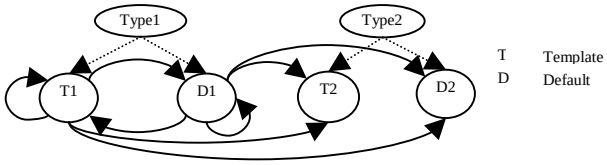

Fig. 6

Starting from these step modifications, adding insertion and elimination, allowed variations in production operations (according to definition levels introduced in section 5.1) can only concern the default level (i.e. in Fig. 6, ***D1→D1***), can also involve template (as in ***T1→ D1*** and vice versa) or even affect the type (i.e. ***D1→ T2*** or ***T1→ T2***). Modifications to optionality and monitoring operations is the same as steps.

At the upper level, updating a process means modifying (as shown), inserting or deleting its operations.

A process modification has to be propagated to its products; this make them not instantiable (invalid) in ***D1→ T1*** and ***D1→ T2*** or ***T1→ T1*** and ***T1→ T2***, i.e. when changes leads to products that have to be completed (as from default to template).

Besides, when a process is updated, lots (belonging to its products) at a certain stage of processing may not undergo all the modifications. A lot in progress, indeed, undergoes in a sequential manner the OpSeq of a product, so modificating an operation will not affect it if the lot has already undergone that operation; we therefore introduce lot *sensitivity.* For instance, lot 1 in Fig. 7 is sensitive to operation b updating, while 4 and 5 are insensitive; for lots 2 and 3 we have to decide the granularity level of an updating action. If we set it at the operation level, only lots before a modified operation will be sensitive to it, so 2 and 3 will be processed using the old version for b. If updating invalidates a product, the production of sensitive lots will be halted until the product is ok.

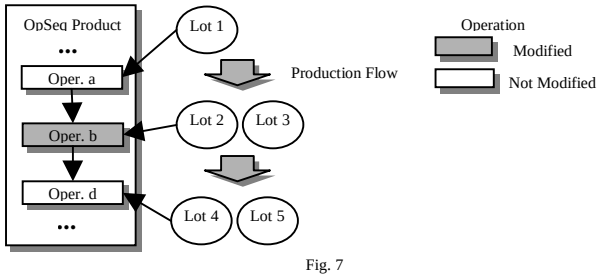

Fig. 7

Finally, when we want to modify just a product, the only modifications that can be made are those which do not violate

inheritance rules (e.g. in Fig. 6, only ***D1→ D1*** is allowed, since a product structure cannot be modified, so changes involving templates and/or types are not allowed).
In addition, when modifying an object, changes can directly affect it or lead to a new instance or to a new object's version (a snapshot of an object during its life cycle [4]).
Really, a version is considered as any other object; the distinction between creating a new version or a new object depends then on the modifications (in general, a new process is created by *major* changes, while a new version represents a *lightweight* creation, due to *minor* changes). Versioning is a powerful and complex tool, as it includes different versions of an object (alternatives), as well as simultaneously active versions and extensions for complex or composite objects, where versioning for their component is introduced [9].
Finally, updating can be extended by introducing modification on an object's class, as well as on aggregation and/or constitutional hierarchies [10]. It is also possible to modify the structure and/or behaviour of an object, in addition to changing its state, i.e. by introducing the concept of object migration [8]. Since we want a strong semantics and a simple model, such updating will not be considered here.

6.3 Deleting.

For large objects like process or product, as opposite to lower-level objects (operations, steps), deletion is replaced with ***obsolescence***, which prevents the object from being used in production, as well as any updating, without, however, eliminating it. A historical archive of objects (even data warehousing oriented) can thus be created. Besides, when an object becomes obsolete, obsolescence is propagated to all objects in its sub-tree (so, if a process becomes obsolescent, no more lots will be worked in any of its products).
When considering smaller objects, such as operations or steps, deleting is allowed, but the real effect depends on wheter the object is multiple or single istance. In the former case delete really means "throw away" an object, while for single instance case we may have local removing (deleting just the link in the object where action is invoked, e.g. remove the presence of an operation in a given OpSeq), global (e.g. removing an operation from any OpSeq it appears), or total (i.e., deleting an operation also from the library it belongs). In some cases referential integrity problems may arise [4]. Finally, invoking a deleting action may involve some on-line questions [11].

6.4 FSM.

A Finite State Machine for a process or product is shown in Fig. 8.

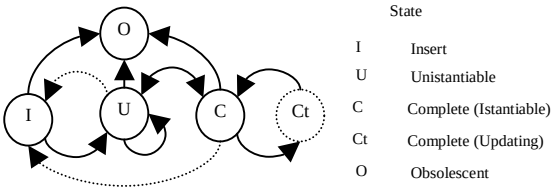

Fig. 8

The object will be in state I when just created, while when completing it (if starting from scratch) or making it different from origin object (if copied), we move in U state. The C state is as U, but with *instantiability* property [11] (that is the possibility of creating products children for a process, and of operating on lots for a product).
Transitions U→C and viceversa are allowed, while U→I and C→I cover new object or new version creation. Ct is a transitory state, representing an istantiable process (or product) when it is modified (C→ Ct), used in order to split lots that will be worked with modified object from those left under original object, choosing lots from sensitive list (section 6.2). After all lots under original object have been worked, the "modified" object overwrites original (Ct→I). Influence of some transitions made on and received form other object states may also occur [11]. The FSM presented is quite simple. In [11], we consider all other objects, in particular operations (and steps), which have the most complex behaviour, since its several states depends on attributes completion (mandatory and optional), and on the possibility of using that operation in creating product and if it will be updatable inside a product.

7 Conclusions.

We presented a logical model for production flow in a manufacturing system environment, adopting object-oriented databases techniques. We first defined general objects characteristics, then we illustrated hierarchies used for object aggregation and composition.
We then considered in detail process and product, which represents core objects. We introduced three classes of operations and steps (production, optionality and monitoring) used inside these objects, and their characteristics (template, default, expression), presenting a comprehensive example for a process and a product child. Finally, we considered states for an object (creation, updating, deleting) with related questions, giving an example of FSM for process and product.
The model presented is a logical, high-level way to describe manufacturing process sequences, achieving both complexity, abstraction and flexibility requirements.

References:

[1] McLean C. et al. - *A computer architecture for small batch manufacturing* - IEEE Spectrum, 20(5):59-64 - 1983
[2] Gupta U.G. – *Management information systems: A managerial perspective* – West publishing company, 1996
[3] Kim W. - *Modern Database Systems* - Acm Press 1995
[4] Bertino E., Martino L.R. – *Object-Oriented Database Systems – Concepts and Architecture* – Addison Wesley 1992
[5] Maier D., Zdonik S. - *Fundamentals of Object-Oriented Databases* – in Readings in OODBMS, Morgan Kauffman 1989
[6] Atkinson M. et al. - *The object-oriented database system manifesto* - in Proc. of First intl. conf. on DOOD, Kyoto 1989
[7] Kim W. et al. - *Composite object revisited* - in Acm Sigmod - Portland 1989
[8] Zdonik S. - *Object-oriented type evolution* - in Advances in database programming languages - Addison Wesley 1990
[9] Katz R. - *Towards a unified framework for version modeling in engineering databases* - in Acm Computing Surveys, 22 No.4 1990
[10] Banerjee et al. - Semantics and implementation of schema evolution in object-oriented databases - in Proc. of Acm-Sigmod - San Francisco 1987
[11] Longheu A. - *Manufacturing models* – Technical internal Report No. IIT-1999-43