OBJECT-ORIENTED APPROACH IN RE-ENGINEERING: THE STMICROELECTRONICS MANUFACTURING MODEL

V. Carchiolo[(1)]
S. D'ambra[(2)]
A. Longheu[(1)]
M. Malgeri[(1)]

[(1)]Istituto di informatica e Telecomunicazioni
Università di Catania
V.le A. Doria, 6 – 95100 Catania – ITALY

[(2)]CAM Support Group - STMicroelectronics
Str. Primosole 50 – 95100 Catania – ITALY

Current manufacturing systems have a very structured production model, expecially when high complexity and precision is required, as in semiconductor devices manifacturing. In addition, rapid changes in both production and market requirements may occur, hence great flexibility is also essential. These goals, together with the requirement of both adding new features as well as removing some model limitations, often impose to re-engineer periodically existing models, reducing as much as possible the time-to-market of a new product.

Re-engineering means first to create an abstract representation of the system (i.e. the abstract model) through a reverse engineering process, then this model is re-designed, keeping as much as possible a high abstraction, so great expandability can be assured, as well as fully exploiting personnel know-how, so it is possible to take advantage of their experience about the model. Updating a model generally also requires to improve existing applications, both re-writing software components as well as adding new features to existing components.

In this paper, the current model used inside STMicroelectronics facilities to define production flow (the sequence of operations to be performed in order to make products) is considered. First, the model and its limitations are described, then introducing an updated, enhanced, object-oriented model, with aggregational and constitutional hierarchies used to model and classify all entities and with a flexible inheritance mechanism used to speed-up and improve the definition of a production flow.

It is also considered which improvements are required on the existing applications, introducing an enhanced environment capable of supporting the enhanced model while preserving at the same time both technical and economic investments through avoiding radical changes in existing environment.

1. Introduction

Modern manufacturing processes [A. Cichocki et al., 1998; F. Leymann, 1999] are highly structured and automated, so in order to manage them it is necessary a production model which may be quite complex, depending on the requirements of the product being manufactured.

Besides, during the use of a model over several years, some questions can occur:

- new marketing scenarios may arise, so new kinds of products could be required , or new technologies (e.g. chips with a lower consumption) can be developed;
- some model limitations may be discovered, even for well-designed models, e.g. a shorter procedure for products manufacturing can be found or repetitive tasks can be discovered;
- changes in the production area of the model may occur, for instance a new generation of production machine with advanced capabilities could allow manufacturing operations improvements, so time and costs of existing products manufacturing can be reduced.

Such situations often requires a periodic re-engineering of existing production model in order to keep it updated. Re-engineering [E. Byrne, 1992] first requires to analyze the currently adopted manufacturing system, identifying its components and their relationships, creating an abstract representation of the system (reverse engineering process [E. Byrne, 1992; Elliot J. Chikofsky and James H. Cross II, 1990], then the model obtained in this way is re-designed through removing its limitations (whenever possible) as well as adding new desired capabilities.

General guidelines that should be followed during the re-design phase are:

- to keep as much as possible a high-abstraction, achieving expandability and also acting as a base for future re-engineering. For instance, introducing a "production machine" with general characteristics comprehensive of all real machines is better than just modelling all categories of current real machines, so it will be likely that a new type of machine can be modelled easier. Such an abstraction can be achieved using a suitable methodology, e.g. adopting an object-oriented approach [S. Gossain et al, 1998], in which the concept of class, inheritance and versioning, as well as aggregational and constitutional hierarchies [S. Gossain et al, 1998; Bertino E. and Martino L.R., 1992], allow to model real environments with high abstraction;
- to fully exploit the know-how of personnel, taking advantage of their experience about the model. Indeed, when working with a model, on one hand its limitations sometimes leads personnel to resort to expedients (for instance performing manually some unsupported coherence checking on the manufacturing process), in order to bridge the gap between the model and new needs; on the other hand, the way the personnel works may evolve, e.g. reducing the time for setting up a new product using a new shorter sequence of actions. In order to allow the re-engineered model to take into account this experience, it is necessary a complete support for new needs, e.g. providing a tool to define internal rules to automatically guarantee coherence (so users are relieved from using current expedients), and also providing some mechanism (e.g. a customizable wizard) that allows experienced users to define working sequences that can be used by inexperienced users during the specification of the manufacturing process, reducing training time.

A significant example is the Integrated Circuits (IC) area, where the increasing miniaturization and rapid obsolescence of devices as well as rapid changes in market demand, impose to re-engineer periodically

existing production models in order to get a model which is both sophisticated, so more and more complex devices can be manufactured, and also flexible, so it is open towards new technologies developement and it can rapidly adjust production in reply to market demand (just in time production).

We addressed all question described so far considering the STMicroelectronics (ST) [ST Microelectronics – http://www.st.com] environment, in particular analyzing Catania facilities manufacturing model [Longheu A, 1999; Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 1999; Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 2000] for production flow definition and management.

The *production flow* is the sequence of logical operations that must be performed in order to make products; we note that such operations just include low-level (i.e. machine devoted) instructions, so an operation like "*take X quantity of raw material form repository Y*" will not belong to production flow (rather it belongs to a higher-level flow, on which we do not focus here [Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 1999]).

In this case-study paper we first consider the reverse engineering task of the ST model, i.e. we outline the current model, describing any entity together with its links, which are a set of relationships used both to group entities according to a hierarchical classification and to show how different entities form the structure of a larger one. Then we re-design the ST model adopting an object-oriented approach, that is modeling real entities as object classes, placed together into one or more hierarchies, both aggregational (when objects must be simply grouped) or constitutional (when an object consists of others). During re-design we follow the previously described guidelines, crafting an "enhanced" object-oriented model that removes the limitations of current model, in particular:

- we give all objects the role suitable for new needs arisen inside ST, also providing each object with a finite state machine to completely describe its behaviour;
- we provide an inheritance mechanism for products development, assuring coherence, correctness and reducing setup time for new products, since they will inherit all their common characteristics from a specific "template" (named *process*). We also provide a three step specification when developing a product, so it is possible to decide what must be inherited from a process and what must be specified for each single product, allowing to set the *degree* of inheritance.
- we introduce a set of operation classes in order to improve the definition and management of production flows, e.g. with the optional operation class we can establish which subsequence of operation must be followed based on the result of an expression. The optional operation can also be used to drive the inheritance between operations belonging to a process and operations belonging to its products children, providing a *conditional* inheritance (see sec. 3).

We use object-oriented techniques in the IC manufacturing models context, making it easier to improve the model. Indeed, current manufacturing models are generally "hard-wired" on to the specific production environment, so when changes occur, e.g. when a new kind of technology is developed, it is difficult to modify the model; on the other hand, object-oriented techniques offers a high abstraction through which improvements can be easily added.

An object-oriented approach reengineering does not represent a solution for any possible future needs, as when radical changes are required on the core of the model, but it is a way to generalize the specific characteristics of the model, so when new needs arise the abstraction of the model should allow to find a solution as in many cases as possible.

Another question that must be considered after an existing model is re-engineered is whether or not the existing software applications are enough to implement the new "enhanced" model.

We addressed this problem in the ST case, where existing software is not completely suitable for new needs, that are:

- to provide a visual environment for a more user friendly production flow definition and management (the existing software works using a character-based interface); this environment should also be cross-platform, since ST intranet presents different operating systems and hardware;
- to provide support for defining automatic coherence and correctness rules to check the production flow (with existing software all such controls are manual, hence the training for inexperienced personnel is harder);
- to manage several user classes, since the model representation, as well as permissions on production flow specification, must be different depending on the specific user group (e.g. design, CAD/CAM, marketing…); with existing software, all users share the same model representation and have the same rights;
- to allow a complete sharing of all information about production flow specification inside the ST intranet, i.e. in a web-based environment (existing software does not support HTML or XML [XML – http://www.w3.org/XML; Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 2000] representation of the model).

Given these new needs, on one hand we want to fully exploit existing applications potentiality in order to preserve as much as possible related technical and economic investments, but at the same time complete we want a complete support for all model improvements, hence if some limit is present in the software, it has not to be propagated to the model, influencing the model itself and/or the way it is used (as currently occurs).

A compromise between enhanced model requirements and these software applications issues can be reached by *enhancing* the existing software, without re-writing it whenever possible (e.g. adding a level placed on top of existing software, supporting new required capabilites and acting as an interface to users). Such a goal can be achieved either using general purpose tools, assuring great expandability but with less efficiency, or implementing customized tools, with high efficiency but limited expandability and high costs, since software has to be created ad-hoc; hence, a trade-off between generality-expandibility and specificity-efficiency must be found when enhancing the software.

Here we consider such questions, introducing an "enhanced" environment capable of supporting the "enhanced" model, satisfying the requirements described above. In particular, we developed software enhancements using Java platform [Java – http://www.javasoft.com], that fits well in the web-based, cross-platform ST environment, thanks to its networking support, applets, security management, HTML and XML support.

The rest of paper is organized as follows: in section 2 we introduce the current model. Then, in section 3 we present the enhanced, object-oriented model, describing objects and their hierarchies, then focusing on process family, product and operation objects, where major improvements are made, and finally describing finite state machine for process families and product objects. In section 4 we consider the existing environment, i.e. software applications used to implement the model, defining improvements needed to support the enhanced model, and finally we present our conclusions in section 5.

2. Overview of the STMicroelectronics Current Model.

ST Microelectronics Catania M5 facilities produces several families of memories, in particular Flash memories for smartcards, and Eprom and EEprom memories. Capacity for such memories currently

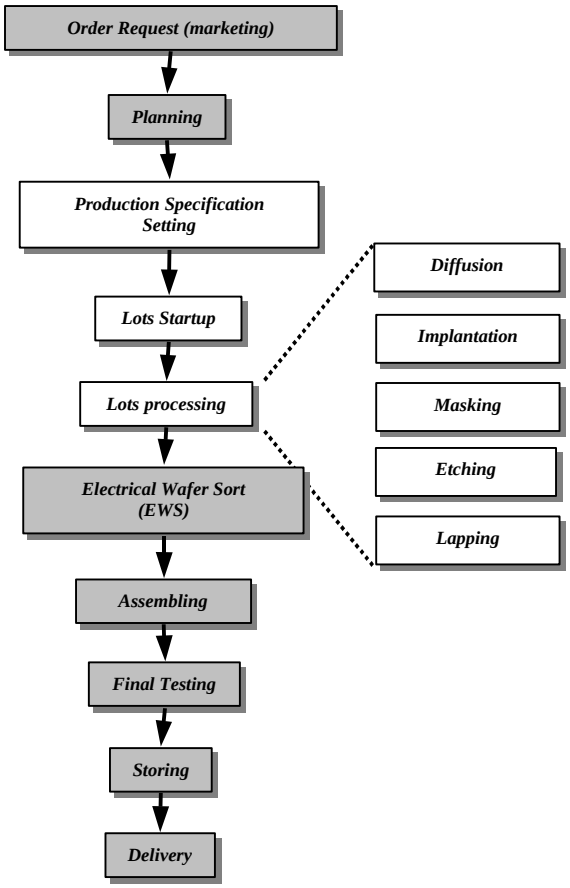

Fig. 1 ST Production model.

ranges from 256Kb to 32Mb, while minimum working dimension is less than 0.5mm. In the following we describe the model for the specification of the production flow used for such ICs manufacturing; the model is placed in the context of the entire ST production model [Longheu A, 1999], so we briefly present the latter to better describe the context in which we work.

The ST production model can be viewed as a set of macro-operations (Fig.1) starting from the client order, evolving through product manufacturing, to delivering of final required products. In the first step order requests coming from clients are examined by marketing area, where an agreement about quantity, price and delivery time is achieved with clients. Then, in the planning phase, quantity of products are expressed in terms of technical parameters, as the number of slices to manufacture, cycle time constraints and so on. The following steps, represented as white boxes, are performed using the production flow specification model, on which we will focus.

The production flow specification setting phase consists in definition of the run-sheet, a document associated with *lots* (a lot is a group of 25 silicon slices, viewed as a single production unit).

The run-sheet contains the sequence of *operations* that must be carried out on lots in order to make ICs. In the lots startup step we establish which lots must be associated with a given product, based on their crystallographic orientation, electrical resistivity and so on. Then we move to lots processing, the core of

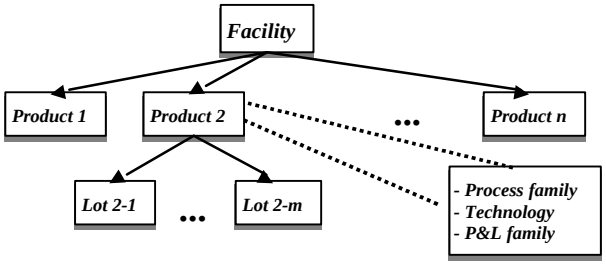

Fig. 2 Products grouped inside a facility.

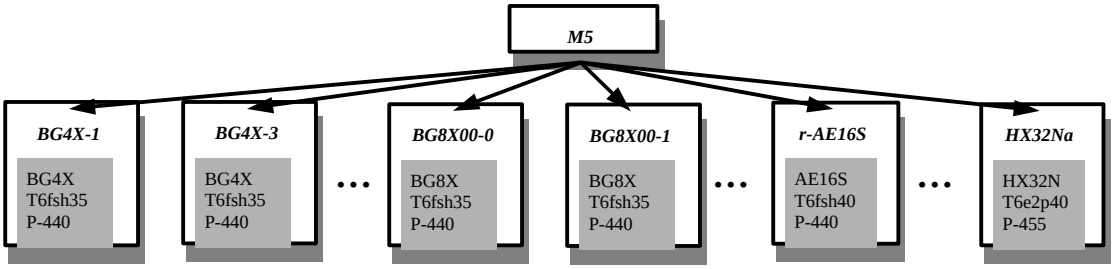

Fig. 3 A set of products inside a facility.

production process, in which several physical and chemical operations (from diffusion to lapping) are performed in order to turn each silicon slice into a set of ICs according to run-sheet specification.

After lots processing, we must check if anything went ok performing the Electrical Wafer Sort (EWS), a test in which all faulty chips are marked and discarded. In the assembling phase we separate each single chip from others, packaging it in a plastic or ceramic package, performing a final test on each IC. Finally, storing and delivery phases complete the production process.

The production flow specification model currently used to accomplish white boxes phases shown in Fig. 1 is a *product-centric* model, where *product* is the logical entity used to represent a real product, and where all products are grouped into the same *facility* (Fig. 2), that is the plant where production takes place.

Each product is characterized by a set of attributes [Longheu A, 1999], e.g. its *name*, or the *material* (silicon or germanium) used to make the product, and by an associated production flow (also named *route*), that is sequence of physical *operations* that must be carried out by production machines to make ICs belonging to that product (hence, the route is the electronic form of the run-sheet document described previously). Example of production machines are furnaces, ionic planter, etching machines, and photolithographic machine, all used to create silicon wafers (slices), *dope* them (i.e. increase their electrical conductivity through introducing doping elements as phosphorus or arsenic into silicon), and manifacture IC on doped silicon wafers.

To logically classify products, three of its attributes (highlighted in Fig.2) are used: *Process family*, *Technology*, *Profict and Losses* (*P&L*) *family*.

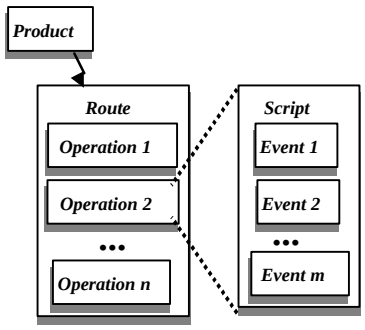

Fig. 4 A production flow (route) and its script.

The *Process family* entity is introduced to group products having "similar" routes, i.e. differing for some operation of the sequence or for the value of some parameter belonging to a given operation. Several process families characterized by the same manufacturing process (even if they may have different sequences) are grouped into a *technology*. Finally, technologies are classified according to the *P&L family*, an economic parameter.

To better explain these parameters, in Fig. 3 we show an example of current hierarchy (lots are not represented), where a set of products manufactured inside *M5* facilities is considered.

Starting from the bottom left, products *BG4X-1* and *BG4X-3* represent two flash memories with a capacity of 4Mb, belonging to the same process family, the *BG4X*. Such products differ for a specific parameter of an operation of their (identical) production flow, indeed it is required a different thickness of the silicon wafer on which they are manufactured, since the *BG4X-1* must be assembled on a chip for smartcards (hence, its dimension must be reduced), while *BG4X-3* will be assembled on plastic package (where more space is available).

Products *BG8X00-0* and *BG8X00-1* represent two versions of the same logical product, the *BG8X00* (a flash memory with 8Mb capacity), belonging to the *BG8X* process family. Such products have different operations in their production flow (differently from the previous case), indeed some etching phases are made with one operation in the *BG8X00-0*, while this operation is splitted into two distinct operations in *BG8X00-1*.

Both *BG4X* and *BG8X* flash memories process families belong to the same technology *T6fsh35*, where the *T6* prefix indicates the *technology generation*, in which guidelines for the device geometry are fixed together with the starting minimum working dimension, that is 0.35mm (*fsh* stands for *flash*). *Starting* means that new improved process families created inside this technology may also be manufactured operating at a lower working dimension. Moreover, any time a major change is made on the geometry, a new techology generation is created.

We note that production flows for distinct process families (e.g., *BG4X* and *BG8X*) may have completely different operations, or they may share some subsequence of operations, or they may have the same operations with different parameters values. So, the same criterias both distinguish process families and separate distinct products inside a process family. The choice on whether a new product is placed into an existing process family or it determines the creation of a new process family is not currently rigorously defined since it is related to corporate internal decisions (e.g., the product may represent a *significant* evolution of an existing one, since it derives from the application of new technologies, or its production flow is *radically* different from previous ones, i.e. the most significant operations (core) have been changed).

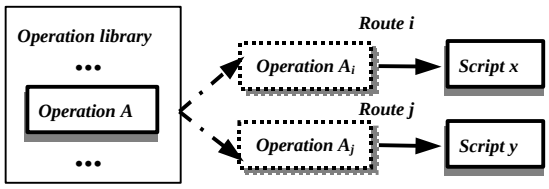

Fig. 5 Operation Library.

Finally, in Fig. 3 we also have the 16Mb flash memory *r-AE16S*, belonging to *AE16S* process family, placed into *T6fsh40* technology, whose starting minimum working dimension is $0.40\,mm$. Both *T6fsh35* and *T6fsh40* are grouped in the same P&L family, *P-440*, while the $0.40\,mm$ EEProm memories technology *T6e2p40*, which includes the product *HX32Na* shown in Fig. 3, is placed below a different P&L family (*P-455*), according to corporate (internal) economic and marketing considerations.

Considering operations belonging to a route (Fig.4) mentioned before, they represent abstract actions that must be further specified through a second level of detail, hence, each operation has an associated sub-sequence (*script*) of *events*, each of which is an "atomic" action directly linked to production machines.

This further specification for operations is needed since e.g. we have *diffusion* operations, but while this simply means that during the execution of the operation doping elements will be introduced into silicon, all physical and chemical parameters needed to really perform the diffusion phase (i.e. to control furnace machines), as the time of diffusion or the quantity of doping element, will be specified just at low (event) level.

Each operation also has a set of attributes in addition to its script [Longheu A, 1999], e.g. *short description*, used for comments, or *workarea*, specifying the production area where the operation can be used.

Besides, in the current model an operation is used in different routes (i.e. different products), maintaining the same set of values for attributes but with a different associated script, so operations belong to a unique library (Fig. 5) from which we get desidered operation any time a new route is created, then associating the related script (scripts are also stored in a library).

The current model described up to now presents several limitations:

- the *product-centric* feature implies that the product is directly placed below the facility entity, while P&L family, technology and process family are viewed as simple product parameters, even if they play a significant role as products classifiers, hence the hierarchy should be better defined, explicitly classifying products;
- no mechanism is present to speed-up definition for products having similar operations, so the specification of a new product is a time-consuming and error-prone task, since it must be created from scratch or, at most, by first copying from a similar existing product, then making changes according to specifications;
- no support for operations setup is provided; indeed, several operations may share a subset of events inside their script, or may have the same script structure with different values for event parameters, so it would be useful some mechanism for operation management;

Solutions for these limitations, mentioned in sec.1, impose the re-engineering task leading to the enhanced model shown in the next section. Table 1 provides a summary for all terms introduced so far.

Table 1 Description of the terms introduced.

Entity	Description
Product	The logical entity representing a real product, i.e. all integrated circuits (ICs) manufactured with the same identical production flow.
Facility	The plant where production takes place.
Profict and Losses (P&L) family	A group of technologies, collected together according to corporate (internal) economic and marketing considerations.
Technology	A group of process families sharing the same manufacturing process (i.e. the same device's geometry guidelines).
Process family	A group of products with similar production flows (i.e. differing for some opertion or for some operation parameter).
Lot	A group of 25 silicon slices viewed as a single unit of work.
Production flow (Route)	the sequence of operations to be performed in order to make products.
Operation	an abstract action (e.g. a measurement of the dimension of a chip) performed by a production machine. To be concretely defined, an operation must be further specified through its script.
Script	A set of events associated to an operation.
Event	An atomic action (the lowest level of detail of specification) directly linked to production machines.

3. The Enhanced Model.

3.1. Overview.

As mentioned in sec.1, the solution we propose for existing model limitations is the re-engineering through the use of an object-oriented (OO) approach. The first step when creating an OO 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 entities) [S. Gossain et al, 1998].

An object can also be used to group others together; this situation, which may also include classical OO inheritance, determines *aggregation* hierarchy. Another case is that of *complex* objects [Bertino E. and Martino L.R., 1992], i.e. consisting of other objects; such links creates *constitutional* hierarchies (also known as aggregation hierarchies [Bertino E. and Martino L.R., 1992]). Finally, several objects can make up a single logical entity, leading to *composite* objects [Kim W. et al., 1989] (constitutional hierarchies may also include these objects). Aggregation, complex and composite objects, all present in a manufacturing environment, 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

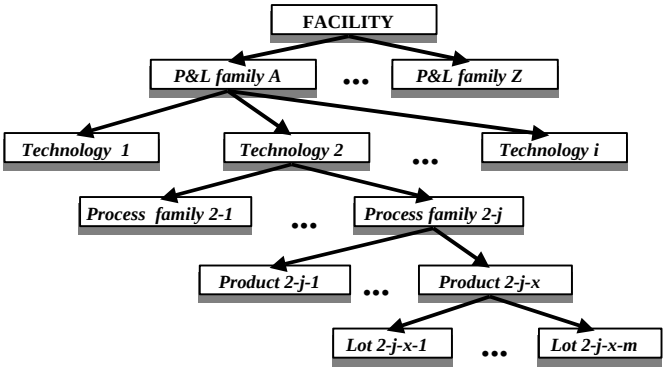

Fig. 6 Logical hierarchy of the new model.

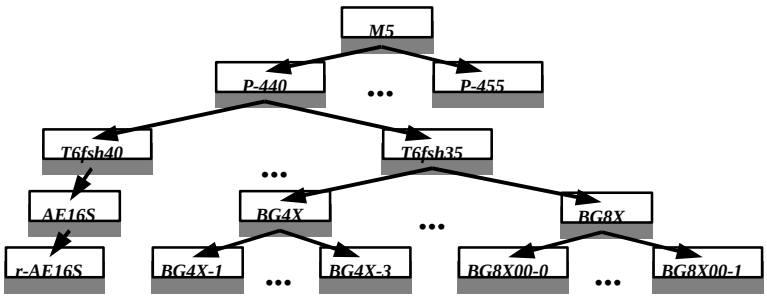

Fig. 7 The hierarchy of Fig.3 revisited.

those that cannot be associated with any other objects. Otherwise, entities can be modeled as attribute of others objects. Finally, after objects identification, it is necessary to determine each object's attribute and methods.

Based on all these considerations, we reconsider existing model, modelling each previously described entity as an object. In particular, we model process family, technology and P&L family, which are currently simple product parameters, as distinct objects placed into a hierarchy based on their role (as defined in sec. 2), since we want to give such entities a more central role and also to classify products more explicitly. Hence, we get to Fig. 6 where a new hierarchy is introduced; we note that such hierarchy is aggregational since it is just used to group objects. According to this organization, the flat hierarchy shown in Fig. 3 can be better represented as in Fig. 7.

Besides, in the re-engineered (enhanced) model we adopt the existing product definition (see Fig. 4), hence a product has an associated sequence of operations, being a complex object defined in terms of a composite object (its route). In this way, we form a constitutional hierarchy. Really, this definition is extended by introducing two new categories of operations (see section 3.2), in addition to the unique category currently supported, providing in this way major abstraction and flexibility during route definition, also reducing time and cost in product development.

In the enhanced model we also make the script for each operation as one of its attributes, since it is required to avoid the current separation between the script and the operation object. In this way operations

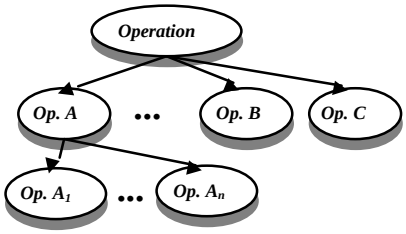

Fig. 8 Subclassing operations in the new model.

are modelled in a slightly different manner than current operation. Indeed, in the existing model (see Fig. 5), an operation A, identified by its parameters (that do not include the script) is used in different routes, say as operation A_1, ..., A_n, associating a different script for each route, i.e. for each operation A_i, so operations A_i differ just for the script.

In the new model, this behaviour is implemented (using OO terminology) first subclassing the operation class (Fig. 8), creating the A operation class, in which its attributes, except for the script, are specified as *static* (they do not change across instances). Then we instance the A class, creating A_1, ..., A_n operations, which can only differ for the script, which is an *instance* attribute. Any time a new set of attributes (except for the script) is needed, a new subclass of operation class (B, C, ...) is created and then instanced in order to specify the script.

We finally associate attributes to each object [Longheu A, 1999] (e.g. also for technology and process family objects). Attributes collect all significant information about an object, even those dispersed throughout the model, i.e. when an attribute is related to another entity or is represented as an entity (the latter is the case of *script* described above). Since this is not a very significant task, we do not specify here the complete list of attributes for each object.

Considering methods, they include object management methods (e.g. constructor), as well as specific methods used to provide each object with its desired behaviour [S. Gossain et al, 1998]. Focusing in particular on object management, or object's *states*, when an object is instanced it will be in a *created* state, then it moves through several states during its life cycle, reaching the final state (*obsolescence*, see sec.3.2.5) when it will be removed from the production model. All state changes are accomplished using specific methods; hovewer, rather than describe a list of such methods, we introduce (sec. 3.2.5) the most significant states of Finite State Machine (FSM) for process family and product classes. Since no other methods are worth of mentioning (they can be considered as implementation details), we do not consider methods throughout this paper.

3.2. Enhanced model core: Process Family and Product.

The major improvement with respect to the existing model is the use of inheritance mechanism to model dependences between process family and related products. As mentioned in sec.2, when 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. The process family definition is extended in order to meet this needs, since in the new model it is not just a simple classifier, but it is a general schema for similar products: creating a process family 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 family. Then, to define completely each product, it is required just to specify some operations and/or parameters.

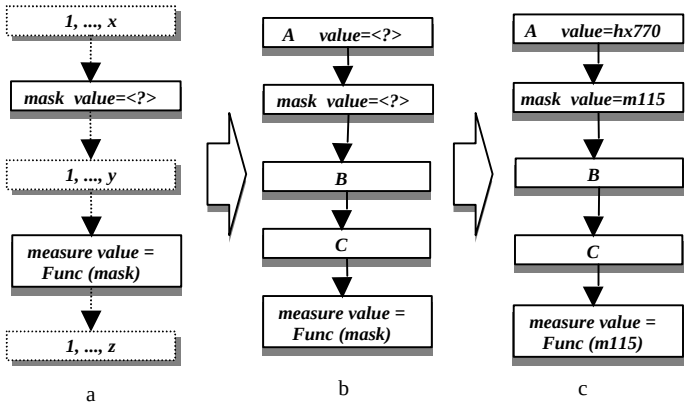

Fig. 9 The definition of an operation through the three steps *type, template* and *default.*

Going into details, the inheritance mechanism we introduce is slightly different from the classical inheritance, indeed it does not provide that *all* operations are *completely* inherited by products children of a given process, rather it provides two mechanisms through which users can establish what must be inherited from a given process family, and what must be further specified inside its children. Such mechanisms are:

- the *type-template-default* definition path (sec. 3.2.1), followed during production operations specification, that allows to decide what portion of an operation must be inherited, allowing to set the *degree* of inheritance;
- the *off-line* conditions inside optionality operations (sec. 3.2.2), through which users establish what subsequence of operations must be inherited, providing a *conditional* inheritance.

These mechanisms, which extend simple inheritance adding more flexibility, are introduced since they fits better to our manufacturing context than other inheritance mechanisms (e.g. multiple inheritance [S. Gossain et al, 1998]).

In order to describe how such mechanisms work, we introduce three categories of operations, i.e *production* operations, *optionality* operations, and *monitoring* operations, used to create a process family route, i.e. when common operations are specified. In the following these categories will be each represented with a specific shape, i.e. classical rectangle for production operations, rhomb for optionality and circle for monitoring operations, since a route can be viewed as a logical flow diagram of the manufacturing process.

3.2.1. Production Operations.

Production operations, which is the only category available in the existing model, contain "instructions" (i.e., *events*) about how to work lots in order to make ICs. In the new model they are specified in three phases: type, template, and default.

Type groups all operations performing the same physical action. For instance, we have *diffusion* type, mentioned in sec.2, or *mask* type, that is the set of operations used to create the geometry of the device by means of partial photo-exposure of the silicon covered by a *mask* (a plate with transparent and opaque areas) followed by removal of unexposed material (distinct masking operations differ for the mask parameter), or *measure* type, collecting all operations where a physical or electrical parameter is traced.

An operation belongs to a given type if its script contains some specific events, characteristic for that type. Referring to the mask operation example, any operation of this type contains a script with at least two fixed events, i.e. the photo-exposure (with the mask as a parameter) followed by a measurement to test the photo-exposure.

Up to this stage (type specification phase), this operation script can be represented as in Fig. 9-a, where a certain number of possible events (1…x, unknown at this stage) appear before the masking event, where the value of mask is a parameter not yet specified. Then there may be other events (1…y), up to the measurement event (whose specification depends on the value of the parameter according to a function *F(mask)*), and finally, another possible group of events (1…z).

The second phase is the *template* definition, where the number and type of all the groups of events 1…x, 1…y and 1…z are chosen, together with the exact form of *F*, but all events parameters are not yet specified (thus moving from Fig. 9-a to Fig. 9-b). This phase is called template since the structure of each operation is completely defined.

Last phase is the *default*, where it is possible to set any remaining parameter and calculate any related expression (*F*), as shown in Fig. 9-c. This step, at which an operation is completely specified, is called default since it represents a possible set of values for a template.

All operations are classified based on their type, and for each type (or template) a group of available corresponding templates (respectively, defaults) can be defined, in order to create a library for future use.

As an example, in Fig. 10 we show a type-template-default aggregational hierarchy, in particular we consider mask operations, whose core events specified inside type are the *MASK* and the *VIS-Insp* event, where the latter represent a visual ispection performed with an electronic microscope to measure physical parameter of interest. At template level, we show two different templates, the *Mask-MS-01*, with the *MS-full* additional event, and *Mask-MS-02*, with *MS-full* and *MS-partial* additional events. *MS-full* and *MS-partial* events are used to collect data coming from the previous measure *VIS-Insp*; when all data have to be collected, the *MS-full* is performed, otherwise the *MS-partial* is used. These *MS* events are anyway completely defined at this level.

Indeed, inside operation defaults shown in Fig. 9, such events are left as they were at template level, while *MASK* and *VIS-Insp* events move from an abstract to a concrete specification, e.g. we have current values for masks (this is indicated using a specific code, as *BG8X00-H-560-A*, *BG4X-H-680-E*, and *BG8X00-H-60-r*), toghether with current, mask-related values for measure events, where relationship is highlighted by naming codes, e.g. the *Vis-BG8X00-H-560* is a measure depending on *BG8X00-H-560-A* mask event (the *Func* shown in Fig. 9 is contained inside *Vis-BG8X00-H-560* event).

Each operation is defined following the type-template-default definition sequence. We distinguish "process family" operations, where these phases are all specified inside the process family (such operations are then independent from the product, i.e. they will be the same for all products belonging to that process family), and "product" operations, where some phase has to be specified when creating a product.

We note that in the current release of re-engineered model, such operations are those for which just the default can be set, so type and template are currently specified anyway inside process family.

The keypoint is that all what is set inside a process family is specified just once and will be automatically inherited by all products children; choosing which phases are specified in the process family context allow users to establish what part of the operation must be inherited, extending inheritance by allowing to set its *degree*.

Clearly, phases specified inside process family represent the part (of that operation) which is common across products children. We also note that the use of type-template-default definition sequence reduces anyway the specification time for a single product, since the use of type-template-default libraries avoids to specify each single event or parameter.

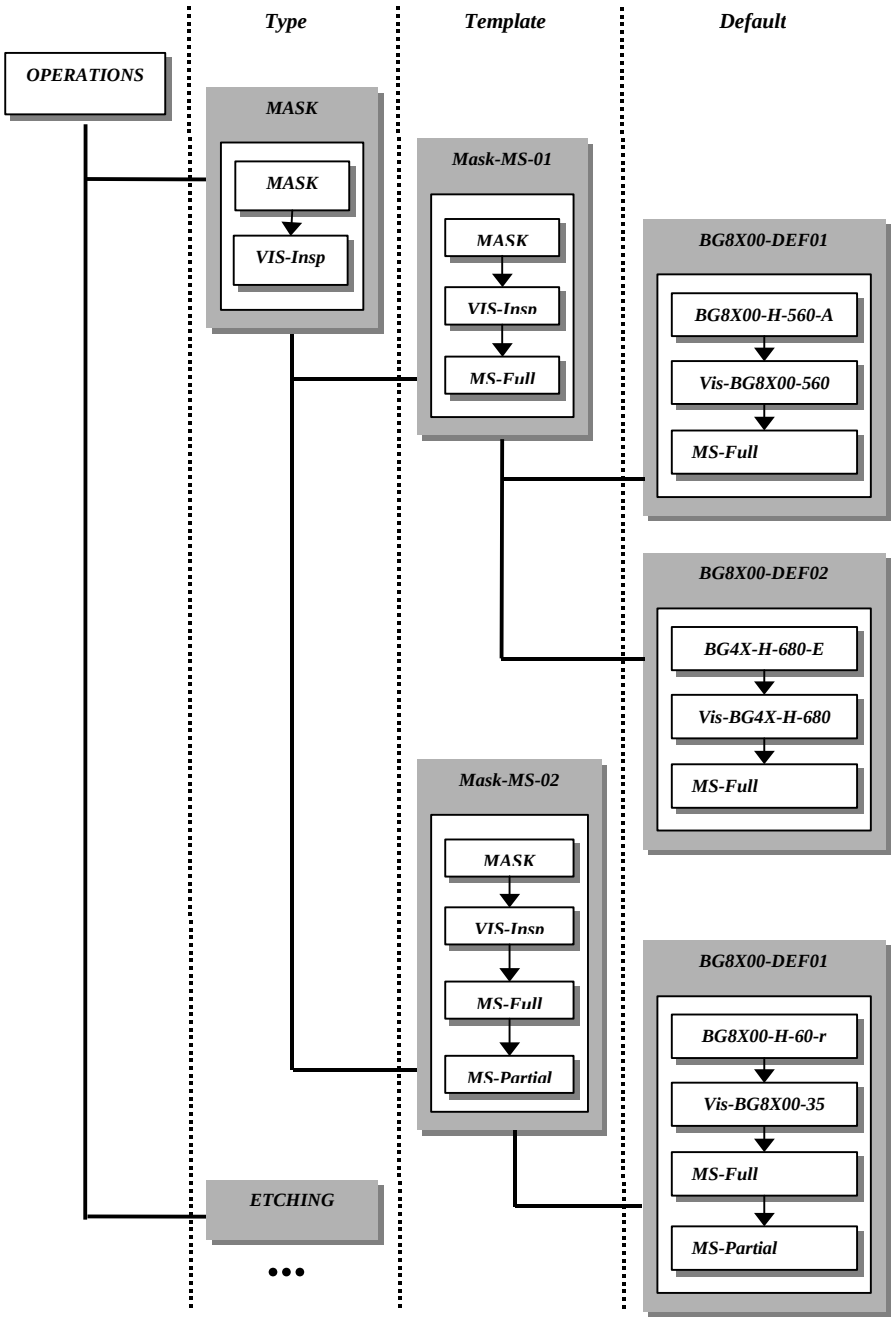

Fig. 10 An example of type-template-default hierarchy.

As an example, we consider the hierarchy shown in Fig. 3, in particular products *BG4X-1* and *BG4X-3*. Such products differ for a parameter (the chip thickness), so their production flow can be defined just once, using a proper template where lapping operation is performed (*lapping* is a mechanical operation which reduces the thickness of silicon wafer). This template contains a currently undefined parameter which will be replaced with two distinct defaults (each with the proper value for thickness) inside each product. Similarly, this mechanism can also be used to represent two process families differing for some parameter. When greater abstraction is required, we move up to the type level, so even more different products or process families can be grouped.

The type-template-default definition path can also be extended to the event level. In particular, in the current model, an event consists of a group of actions named *steps*, so it is easy to define for an event the *type* as the level in which core steps are chosen, while in the *template* all steps but their parameters are specified and finally in the *default* parameters are specified. Although, we want that an event is *atomic* (sec. 2), i.e. we do not want that steps are considered, since they concern low level details (related to manufacturing), hence we extend just template and default levels to events by viewing parameters for steps as a whole. In this way, an event template is a complete event with no parameters, whereas in an event default all its parameters are specified, so for instance event A in Fig. 9-b is an event template, (in Fig. 9-c there is a corresponding default), while *MS* events in Fig.10 are default events. A template event can only appear in a template operation, whereas a default event can appear both in a template operation and in a default operation, (a template operation must have at least one template event, while a default operation consists of default events only since it must be completely specified).

We note that in current model abstraction support, as the type-template-default phases, is not provided at all.

3.2.2. Optionality operations Expressions

The second category of operations has the role of control flow statements. 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 is represented with the optionality operations category, and it does not imply any physical action (no events inside), as it only contains a condition. This is also the reason why in the current model only production operation exists (in the current model an operation must always contain events while a production flow does not support any alternative paths).

We follow two phases when defining this category of operations, one in which the structure of the condition is set (as in a template), and one in which all the current values of the variables in the conditions are provided (as in a default), so conditions can be evaluated and a path is chosen. We note that a phase equivalent to *type* for production operations should allow to select a specific category of condition from a library. We are currently developing such classification.

Like the previous category, optionality operations can generally be “process family dependent”, i.e. completed inside the process family, or parameters values will only be known inside products, so only the structure of the condition is set in the process. The inheritance mechanism, hovewer, is slightly different from production operations. In order to describe such mechanism, we consider in detail the structure of conditions and expressions.

A condition can be viewed as an *if…then…else* or a *case…switch* construct, containing 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.

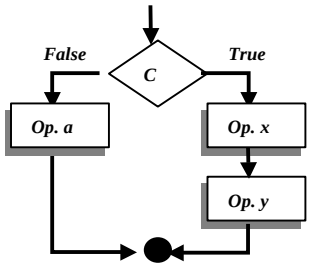

Fig. 11 Example of off-line condition.

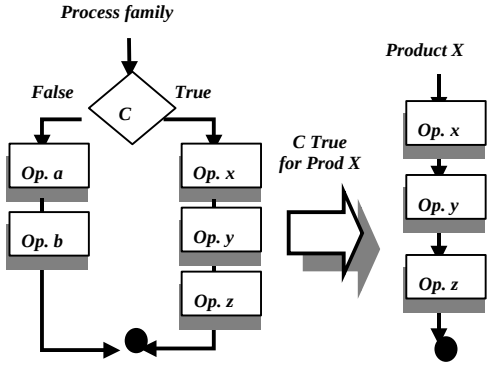

Fig. 12 - Inheritance of off-line condition.

if the condition is set in a product, the environment consists of all product attributes plus those belonging to parent family process and technology). Environment can also include other information, e.g. user-defined variables.

Expressions can be *off-line* or *on-line*. Off-line expression evaluation depends on definition of objects in Fig. 6 (it does not depend from the production flow), while on-line conditions are evaluated when production is running.

Off-line expression can be used to improve process families and product specification through a greater abstraction. Considering the example hierarchy shown in Fig. 3, products *BG8X00-0* and *BG8X00-1* differ for a subsequence, i.e. the *BG8X00-0* contains some etching phase performed with a single operation, while the *BG8X00-1* requires a pair of operations. Such a situation can be easily modelled through a proper off-line expression (*C* in Fig. 11), e.g. based on product's version (the last character of products name), so that the path with one operation is chosen for *BG8X00-0*, while *BG8X00-1* will use the path with two operations.

Off-line condition can also be used in the same way to represent process families differing for some subsequence, given that a proper condition (e.g. based on some object's attribute) must be established to choose the right path. We note that in the current model there is no support for grouping such similar sequences differing just for some subsequence, hence each production flow must be defined separately (generally copying from an existing one and then making changes according to specifications, as mentioned in sec.2).

Table 2 Operations categories.

Operation Category	Meaning
Production	Represent physical operations, i.e. it contains a sequence of events (script). It is defined following three steps: type (where core events are established), template (where the script structure is completely defined), and default (where parameters are specified). Such steps also allow to set the degree of inheritance between process family and product production flows.
Optionality	Represent control flow operations, i.e. it contains a condition used to select different subsequences.Expression inside a condition can be on-line or off-line, depending on whether expression depends or not on some production-related quantity.Off-line expressions provide conditional inheritance between process family and product production flows.
Monitoring	Represent checking operations, i.e. it contains expressions used during execution of the production to check the status of parameters of interest.

On-line expressions are also used to provide a greater abstraction, but they model conditions depending on some production-related quantity. An example is the *fault check* operation, used to control lots belonging to a given product in order to evaluate the rate of faulty chips, an important parameter used to discover how manufacturing process can be improved (for instance, this rate is used to test manufacturing process during new technologies development, where production flow must be refined until faulty chips rate decrease below a given threshold). Since checking each single lot is a very time-consuming task, lots are checked with a statistical frequency, i.e. each time a certain number of lots have been processed, the next lot must perfom a specific test operation to evaluate faulty chips rate. In order to model this, we create an optionality operation whose condition is based on the number of lots being processed, so when the condition is true the faulty check operation will be perfomed on the current lot.

In a process family route there can be both on-line and off-line conditions, whereas in a product only on-line can "survive" (they will be evaluated during production): all off-line must be solved and replaced in the route for that product with the path associated with result value (Fig. 12).

While on-line conditions are inherited by products "as they are", together with all their operation subsequences, off-line expression are used to establish which subsequence of operations a product inherits from a process, acting as a tool to drive the inheritance, i.e. achieving a *conditional* inheritance.

In addition to on-line/off-line characteristic, a condition can also be *human-dependent*. This is modeled by introducing users and user classes, i.e. groups of users with specific rights and/or limitations. Such conditions, which may be both off-line and on-line, are 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 example of human dependent condition is when the subsequence of operation to be performed depends on the human operator supervising production machines, e.g. he has to decide what machine can or must be used according to its state, an information which affects the subsequence of operations to be executed and which is not always automatically detectable, since some machines must be managed manually.

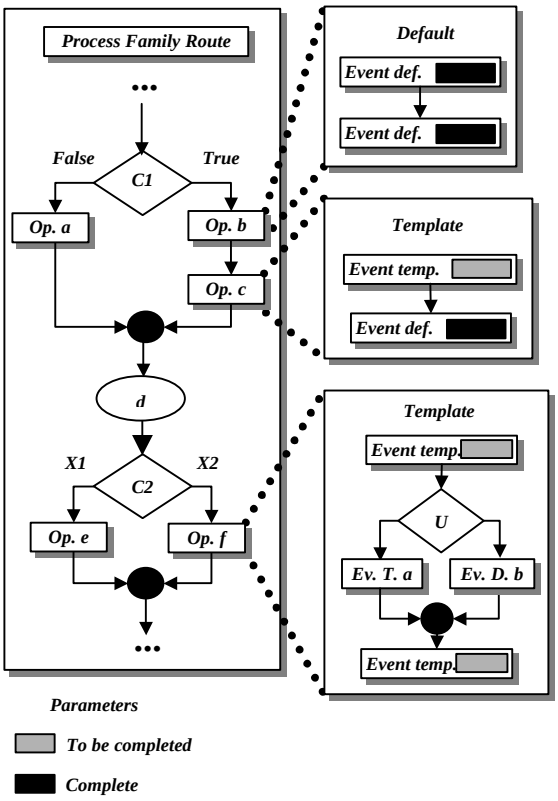

Fig. 13 Example of process family definition.

Finally, we note that an expression can also appear in a production operation (as the *Func* shown in Fig. 9), and that it is possible to have also optionality events, which can only appear in production operations (optionality operations contain no events).

3.2.3. Monitoring Operations.

This category of operations is used during execution of the production sequence to check the status of parameters of interest (e.g. lots throughput), to track alerts when missing given deadlines or to trace the history of flow execution. Monitoring operations play the role of supervisoring operations [A. Cichocki et al., 1998; F. Leymann, 1999]. As in the case of optionality class, these operations contain no events, hence do not exist in the current model; moreover, monitoring events can be defined.

3.2.4. Putting it all together.

Table 2 shows the three categories of operations we introduced, together with their main features, while a comprehensive example of a process family is shown in Fig. 13; in the route, entirely created during process family definition, we have an optionality operation with a condition C1, here supposed binary, human-independent and on-line. From this operation we have two different paths, one with operation a, and the second with b, assumed to be process family-dependent, and c, product-dependent. This means

that type, template and then default for b are all specified in the process family context, while the default level for operation c is specified in the product. Then, d is a monitoring operation, followed by an optionality operation containing a binary, human-independent, off-line condition C2 with two paths, operations e and f. This is product-dependent,with a template containing a human-dependent, off-line optionality event (condition @U), whose choices are template a and default b.

We note that generally it is also possible to have jumps in a route or in a script. A jump can be conditional or not, using optionality operations, and can be a loop when it is directed to previous operations. Hence, we can create *repeat-until*, *for*, and *while-do* or simple *goto* constructs.

When a product child of such a process family is created, the inheritance mechanism allows to create a partially complete product, with at most template production operations (product-dependent) to be specified at default level, with completed default operations (process family - dependent) that must not be altered, otherwise inheritance would be violated, and with optionality and/or monitoring operations containing expressions. Inside expressions there will be environment variables already defined during process family definition, variables known in product definition context (e.g. product name), and variables to be completed manually.

Having finished such steps, all off-line expressions must 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 on-line expressions, missing parameters will be known when production will run.

All these steps are needed since in order to work lots according to a production flow of a given product, the product must be completely specified at a concrete level (e.g. templates are no more allowed).

By this hypothesis, a product child of the process defined above is shown in Fig. 14, where we assume that C2 returns X2 and that the user @U created the product choosing the right-hand side branch in the relative condition.

The high abstraction provided with gradual and conditional inheritance mechanism allow to reduce significantly time for production flows specification, e.g. referring to Fig. 3 we can define just a production flow for *BG4X-1* and *BG4X-3* products (using the type-template-default mechanism), and just a production flow for *BG8X00-0* and *BG8X00-1* (using optionality operations). Similarly, we may also group (if there is a significant set of common operations) production flows for *BG4X-1* / *BG4X-3* and for *BG8X00-0* / *BG8X00-1* into a single production flow, saving time and reducing error-prone situations.

3.2.5. Process Family and Product States.

To manage each object, a Finite State Machine (FSM) is provided, as mentioned in sec. 3.1. This models objects life-cycle, both with classical states as creation and deletion, as well as states specifically devoted to model production [A. Cichocki et al., 1998; F. Leymann, 1999; Longheu A, 1999]. Considering just process family and product objects, in Fig. 15 we show their FSM, examining in the following each single state.

The first state we consider is *Creation*, in which a new instance for an object is created. This can be done from scratch, or using the inheritance mechanism (if a product is being created, as no template for process families is defined in our context), or finally copying from other objects.

In particular, when a product is copied, pasting the copied product inside the same process family, leads to a conflict concerning the logical identity of the new product, since it will be logically the same as the original until it is modified. If the copy is made from another process family the copied product is *invalid* (i.e. lots cannot be worked following the production flow of that product) until it will be changed in order to meet inheritance constraints of the destination process family.

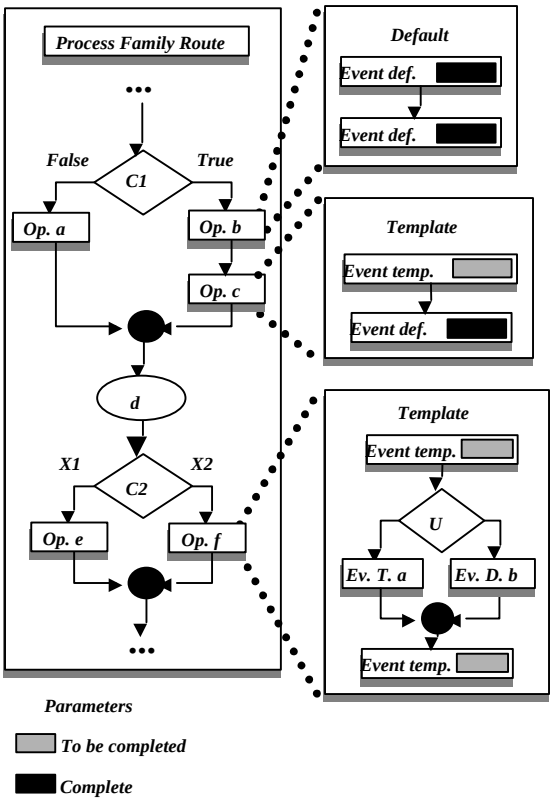

Fig. 14 Example of product definition.

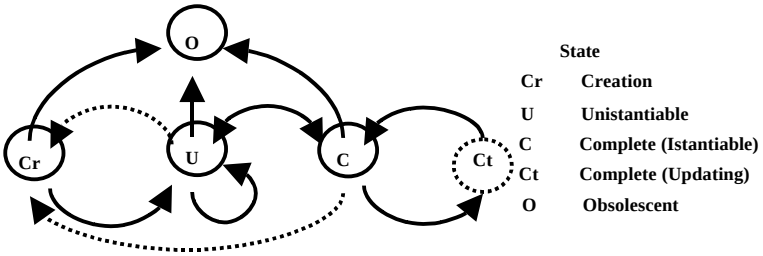

Fig. 15 FSM for process and product.

Copying process families can result in a logical conflict only (if the copied process family is pasted inside the same technology), while no inheritance constraints can be violated when pasting a process family out of its technology, since technologies simply group process families sharing the same geometry, where a device's geometry is not currently defined through an inheritance mechanism in our context. Finally, we can copy a process family and pasting it as a product. In this case, the process family also must become a product, i.e. all of its template operations must be completely specified at default level, and any off-line expression must be evaluated and replaced with the subsequence associated to the result value

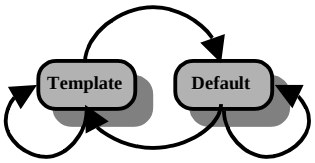

Fig. 16 Event changes.

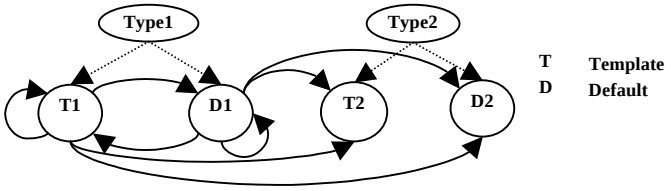

Fig. 17 Operation changes.

(sec. 3.2.2), hence following a procedure like that leading from process family represented in Fig. 13 to product shown in Fig. 14.

We note that copy issues described here are somewhat similar to instance migration in OO databases [6].

The transition $Cr \rightarrow U$ models the resolution of conflicts made by users after an object has just been created, moving from the *Creation* state to the *Unistantiable* state. Hence, the *Unistantiable* state represents the initial state for an object which has been made though clearly distinguishable from the object from which it has been copied.

The object (process family or product) is then completed, i.e. we insert all values for its attributes, also adding operations together with their scripts. The transition modeling the completion phase is the $U \rightarrow U$, while when the object is completed, it reachs the *Complete* (*Instatiable*) state (transition $U \rightarrow C$), which means that products children can be created (if the object was a process family), or that lots can be worked according to the object's production flow (if the object was a product). Really, users must explicit declare if an object is instantiable, even if it is completed, since we are requested to model internal corporate decision procedures. According to this possibility, istantiability can also be revoked (transition $C \rightarrow U$).

Now we consider updating, in which it is possible to modify the object, i.e. its attributes and/or its associated objects, as in the product constitutional hierarchy; updating process family and product includes the insertion, deletion and modification of each single operation (and/or event), as well as the change of their order. It is therefore necessary to analyze the modifications that can be made starting from the lowest level (event), considering how they affect the operation level finally moving up to the process family or product objects.

Considering different events categories, i.e. production, optionality and monitoring, allowed variations (Fig. 16) for events (for which just template and default are defined) are template $\rightarrow$ template, where the internal structure is varied, default $\rightarrow$ default where just parameters are altered, and finally template $\rightarrow$ default and default $\rightarrow$ template, the former being a specification and the latter a generalization task. Modifying optionality and monitoring events means altering their expressions and/or conditions.

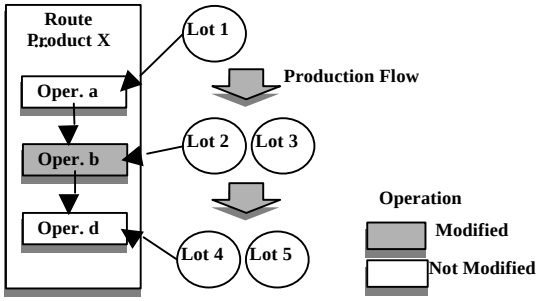

Fig. 18 Lots sensitivity.

Considering a production operation, it can be modified acting on its attributes, as well as on its script. In the latter case, we can add, remove and/or modify its events (according to events modifications described before), hence referring to Fig. 17, where *T1* is a template and *D1* is a default belonging to the same type *Type1* (but *D1* does not necessarily belong to *T1*), allowed variations in production operations are:

- $D1 \rightarrow D1$, which includes both the case in which just parameter values are changed inside a given default and the case in which the given default is changed into a new template and this template is completed as a new default (i.e. $D1 \rightarrow T1 \rightarrow D1$), provided that the destination default must belong to the same type of current default, i.e. core events must not be changed at all;
- $T1 \rightarrow T1$, where the structure is changed acting on all events but core events, so the destination template remains in the same type;
- $D1 \rightarrow T1$ and viceversa ($T1 \rightarrow D1$), where the given default is changed into a template belonging to the same type, either by simply removing values from parameters, or by changing the structure; $T1 \rightarrow D1$ represent the modification that leads from a template to a default by providing a set of values;
- $T1 \rightarrow T2$, $T1 \rightarrow D2$, $D1 \rightarrow T2$ and $D1 \rightarrow D2$ that represent situations in which a given template or default is modified into a new template or in a new default by acting on all its events, including core events, so that the type is changed;

In a few words, modifications on production operation can only concern the default level (e.g. $D1 \rightarrow D1$), can also involve template level (as in $T1 \rightarrow D1$) or can even affect the type (e.g. $T1 \rightarrow T2$).

Modifications to optionality and monitoring operations are the same described for events.

At the upper level, updating a process family means modifying (as shown above), inserting or deleting its operations. A process family modification has to be propagated to its products, in order to satisfy inheritance constraints; this make such products not *instantiable* (i.e. such products cannot be used for manufacturing ICs) in $D1 \rightarrow T1$ and $D1 \rightarrow T2$ or $T1 \rightarrow T1$ and $T1 \rightarrow T2$ cases, i.e. any time changes lead to products that have to be completed. For instance, the $D1 \rightarrow T1$ modification performed on a given process family means that one of its operation has been changed in such a way that some parameter value is missing (since it has been changed from a default to a template); all products belonging to that process

family must also reflect these changes, so that template is copied into each product, overwriting the previous default, hence all products have to be completed by specifying parameters in order to use them for ICs manufacturing. We note that old parameter values cannot be used, since the destination template *T1* may have a completely different structure (e.g. more events) with respect to *D1*, given that *T1* only has to be of the same type of *D1*.

Besides, when modifying directly a product, the only modifications that can be made are those which do not violate inheritance constraints (e.g. in Fig. 17, only D1 → D1 is allowed, since a product structure cannot be modified, so changes involving templates and/or types are not allowed).

Now we must consider some on-line related issues that arise when a process family or a single product is updated. Indeed, lots belonging to a given product (or those belonging to its products, if considering a process family) are at a certain stage of processing when modifications are made, so they may or not undergo all the modifications.

In particular, a lot in progress undergoes in a sequential manner the route of a product, so modifying 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. 18 is sensitive to operation *b* updating, while 4 and 5 are insensitive; lots 2 and 3 will be processed using the old version for b, i.e. lots are not worked using the newer version of the operation if they are currently undergoing that operation.

Really, modifications do not automatically overwrite the existing production flow, as not all lots must be worked according to these modifications. This means that modification leads to a temporary state in which is made a copy of the current object with modified production flow. In this state users can select which lots must move to the changed production flow, provided that such lots must be chosen them from the *sensitive* lots list, i.e. lots which must undergone unmodified operations cannot be moved. When all sensitive lots that was not moved are finished, the existing production flow is no more needed, and so it will be overwritten with the modified production flow.

Referring to the FSM in Fig. 15, all these issues are modeled with $C \rightarrow C_t$ and $C_t \rightarrow C$ transactions, where the *Complete (Updating)* is the temporary state mentioned above, and $C \rightarrow C_t$ models the modifications phase, while $C_t \rightarrow C$ transaction is performed when the existing production flow is overwritten.

We finally note that if an updating invalidates a product (makes it unistantiable), the production of sensitive lots will be halted until the product is completely specified.

In addition, when modifying an object, changes can directly affect it or can lead to a new instance or to a new object's version (a snapshot of an object during its life cycle [Bertino E. and Martino L.R., 1992]). The distinction between creating a new version for a given object or a new distinct object depends on the modifications, i.e. a new distinct object is created if we apply *major* changes to an existing object, while a new version is a lightweight creation, due to *minor* changes (a more rigorous criteria is not currently defined); hovewer, in our model a *version* is treated as any other object.

We also note that when an object is modified, the choice whether such modifications must overwrite the existing object (using the temporary state C_t) or must create a new version/istance depends on several considerations, e.g. if the production flow of existing object must be optimized, it is overwritten.

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/composite objects, where versioning for their component is introduced [Katz R., 1990]. Each time a new version of the current object or a new object is created, we move from the *Unistantiable* or *Complete* (*Istantiable*) state towards the *Creation* state of the new object (a new version is treated as a new distinct object), hence we have transition $C \rightarrow I$ and $U \rightarrow I$ in Fig. 15, where dotted lines indicate that the *Creation* state reached belongs to the FSM of the new (not current) object.

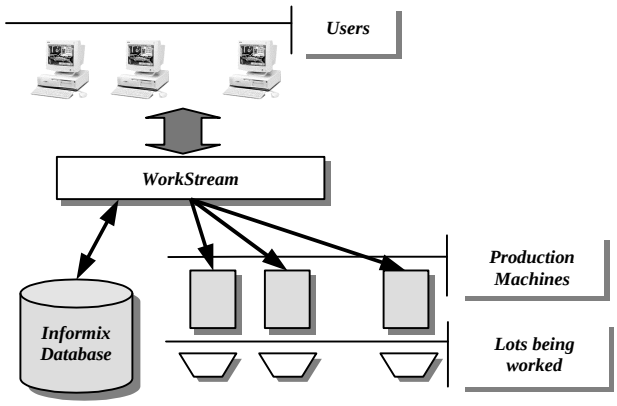

Fig. 19 Current environment.

We finally note that updating can be extended by introducing modification on an object's class, as well as on aggregation and/or constitutional hierarchies [Bertino E. and Martino L.R., 1992]. 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 [Bertino E. and Martino L.R., 1992]. Since a simple model is required, such updating will not be considered here.

The last state we consider is *Obsolescence*, which models the deletion of an object. Indeed, for large objects like process family or product, as opposite to low-level, simple objects (operations, events), simple deletion is replaced with a lighter *obsolescence*, which prevents the object from being used in production, as well as any updating, but without eliminating it, since a historical archive of objects (even data warehousing oriented) has been required during our model development. Besides, when an object becomes obsolete, obsolescence is propagated to all objects in its aggregational sub-tree (so, if a process family becomes obsolescent, no more lots will be worked in any of its products). In Fig. 15, *Obsolescence* can be reached from any state (but the temporary C_t).

4. Environment Enhancements

The enhanced model introduced so far provide significant and useful changes in production flow specification, but we also have to consider whether existing environment can implement it, or how environment must be changed in order to support the new model.

The STMicroelectronics M5 facility environment consists (Fig. 19) of an Informix database containing all data and a proprietary production flow manager application, WorkStream (WST) [Workstream - http://www.consilium.com/products/ws.htm], used to create and manage production process. Besides, WST controls production machines, both manual (sending text commands to operators that supervise the machine) and automatic (using specific communication buses and protocols).

This environment has several limitations, mainly due to WST, since it was initially developed as a completely general purpose tool for ICs manufacturing.

First of all, WST is product-centric, i.e. it does not support neither process-family nor technology or P&L family as ancestor objects (Fig. 6), rather they can be at most defined as simple attributes for

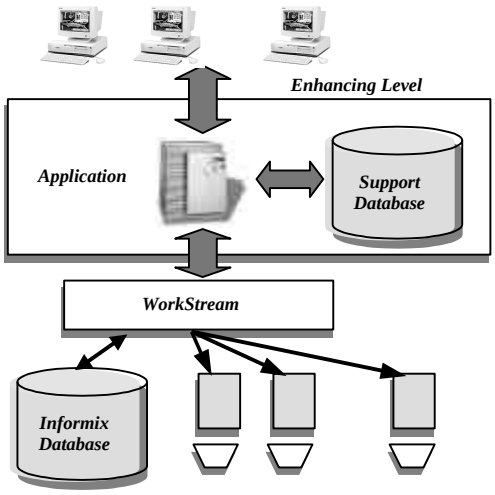

Fig. 20 Enhanced environment.

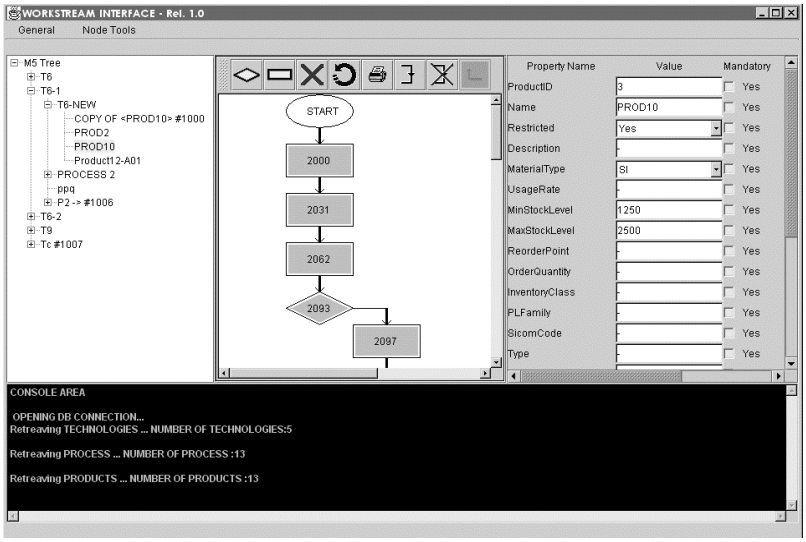

Fig. 21 The new Interface for production flow management.

product. As a consequence, WST cannot support any inheritance mechanism for products, hence any new product must be created from scratch, usually copying an existing product (with a “similar” sequence as defined in sec.2) and then adding, deleting and/or modifying its operations in order to get to the new desired product (as mentioned in sec.3).

Another limit is that WST manages objects separately, e.g. a route must be explicitly linked to a product; this may be a problem since coherence must be assured manually acting on internal user defined tables where such relations are stored; in general, WST does not offer any automatic validation tool for

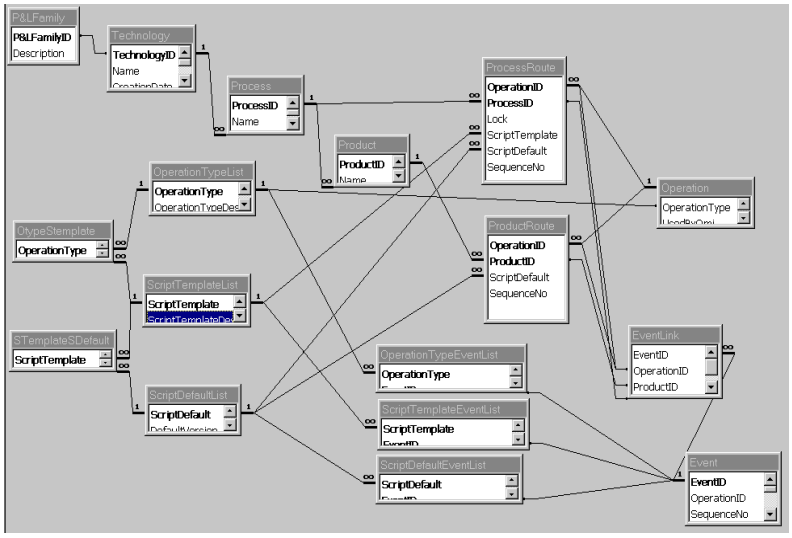

Fig. 22 Support database schema.

such purpose. Besides, many production settings are repetitive but cannot be automated. Finally, WST uses a vt100-terminal interface, which is unsuitable to represent hierarchies or sequences in an user friendly way.

All these environment limitations require an enhancing in order to support our model, providing that any improvement should not consist in a complete migration, in order to both avoid to stop the production and preserve technical and economic investements.

The simple strategy we adopted (Fig. 20) was to create an intermediate level, placed between existing environment and users, consisting of an application together with a database, which aims to maintain the old architecture hiding it to users, while providing a complete support for the enhanced model. Users will interact with new tools, even if they can continue to work as usual, so a gradual migration can be achieved.

The *application* is Java-based and offers an object-oriented view of the entire production, according to the model introduced in sec.3. Referring to the snapshot shown in Fig. 21, the left panel offers a tree representation of *aggregational* hierarchies, allowing to explore and manage P&L families, technologies, process families and products. The center panel provides *constitutional* hierarchy management, showing the production flow for a given process family or product. Double-clicking on an operation will show its script, allowing an user to manage it. The right side is a property panel, used to set objects attributes, while the bottom panel is a text area providing control messages during current session. We note that since in our model the script of an operation is viewed as an attribute (sec.3.1), the value for script type, template and default will be specified in the right side panel.

This user-friendly interface provides process family - product inheritance with type, templates and defaults management for both operations and events (sec.3.1), as well as engines for expressions management.

Morever, it assures a flexible and robust object management with an embedded finite state machine (FSM) associated to each object, and taking into account on-line interaction with lots mentioned in sec. 3.2.5 [Longheu A, 1999].

The second component of the enhancing level is the *support database* (whose snapshot is shown in Fig. 22), which must be introduced in order to store all data related to new model. These includes process family and technology objects, type-template-default support, as well as optionality and monitoring operations (only production operations indeed exist in current environment).

We note that such data could be placed directly into the existing Informix database by extending its schema, but this corporate database is shared across ST Intranet, i.e. it is accessed by different working groups, each of which use database data according to its internal model. Our model concern just the CAM/Automation support group, so modifications needed to implement our model cannot modify the Informix database schema, which must be viwed as a *core* database where the common base view of all models is stored. Additional objects defined inside our model, e.g. templates, are then stored on a separate database (the support database) that works in cooperation with Informix database in order to provide with users the complete model. In this way, support database is not a replica of existing database, hence we do not have to assure data coherency between databases; although, we must manage relationships between databases, e.g. creating and maintaining links between a process family (stored in the support database) and all its children products (stored in existing database). Such links are guaranteed by our application, which communicates with both databases. Really, our application does not communicate directly with Informix database, since this would violate WST internal schema coherence [Longheu A, 1999], rather it establishes a connection with WST, using it as a mediator towards Informix database. This communication is implemented through a Java-based driver emulating a telnet session with WST (the most conservative solution was required).

Referring to Fig. 22, where all support database main tables (together with their relationships) are shown, we have the aggregational hierarchy P&L family – Technology - Process family – Product, then both process families and products are linked to their routes (respectively, through *ProcessRoute* and *ProductRoute* tables). Such routes contains links both to *Operation* table and to the set of tables used to implement the type-template-default specification. This is needed since when the same operation is placed into different routes (Fig. 5), it is shared (together with its type) across such routes, but the template and default must be specified each time this instance is used in a given route, so e.g the *ProcessRoute* table contains the list of *OperationID*s each of them having a *ScriptTemplate* and *ScriptDefault* fields. We note that *ProcessRoute* and *ProductRoute* tables, together with their links, represent constitutional hierarchies, even if the inheritance mechanism is managed by the Java application. The aggregational hierarchy for type, template and default is implemented through tables *OperationTypeList*, *ScriptTemplateList*, *ScriptDefaultList*, *OtypeSTemplate* and *StemplateSDefault*, while the constitutional hierarchy operation-events is represented by *OperationTypeEventList*, *ScriptTemplateEventList* and *ScriptDefaultEventList*, all linked to *Event* table.

We note that support database is currently a relational database; we are planning to develop an object-oriented database in future releases, so inheritance and hierarchies (aggregational and constitutional) support can be easily managed directly by database thus reducing external coherence management and having the Java application acting just a simple database interface.

Both application and support databases defined so far are merged with existing environment using widespread protocols and languages, e.g. the Java application communicates with WST using vt-100 terminal protocol, while communication with support databases uses JDBC-ODBC databases driver. In addition, as a future extension, such environment will be made CORBA [18] compliant, achieving the trade off between generality and specificity mentioned in sec.1.

Finally, our application can also run as an applet into a browser, making production process information available inside the whole STMicroelectronics Intranet (providing an adequate security management), improving data exchange activities (e.g. production reporting), essential to increase productivity.

5. Conclusions

In this paper, we presented the re-engineering of the production flow model used inside STMicroelectronics facilities using an object-oriented approach, also introducing related environment enhancements.

We first introduced current model used for specification of the production flow for IC manufacturing, starting from an overview of the entire ST production model, then focusing on specific phases concerning with the production flow specification, introducing the *facility-products-lots* aggregational hierarchy, as well as products classifiers (P&Lfamily, technology and process families) and *product-operation-events* constitutional hierarchy.

We then highlighted main limitations of current model, moving towards an object-oriented, "enhanced" model, where a new aggregational hierarchy with explicit products classification is introduced, together with an object-oriented operation management, in which script object management is improved through removing the current separation between operation and script.

Then core objects, i.e. process family and product, have been considered in detail. In particular, we introduced a powerful inheritance mechanism between process family and product through which product setup time and costs are significantly reduced.

We also outlined three categories of operations, *production*, *optionality* and *monitoring*, in order to provide the production flow specification with a more powerful expressiveness (e.g. *if…then…else* construct can be defined).

A three steps specification (*type*, *template*, *default*) for production operation was introduced, improving the flexibility of the inheritance between process families and products, since the *degree of inheritance* for production operation can be set (i.e. what part must be inherited). Type, template and default also significantly improve operations setup, since events have not to be separately specified each time a new operation is defined, as it is currently required, rather event groups can be selected from type-template-default libraries and then used to rapidly specify a script inside an operation.

We then introduced *conditions* and *expressions* support for optionality operation, defining *off-line*, *on-line*, and *human-dependent* expressions. Such expression types are needed to model existing environment (current model does not support such features). Moreover, a *conditional inheritance* mechanism between process families and products is provided through off-line expressions, allowing to define more and more production flows in a single design phase.

The description of the enhanced model was completed by examining finite state machines for process family and product objects to show their management. In particular, conflicts issues are considered when a new object is created by copy, then the updating state was analyzed, starting from defining modifications from low-level (events) to high-level (process families and products) objects, also considering on-line questions, introducing lots *sensitivity* and temporary state needed to manage updating actions while production is running.

We then introduced the architecture of an enhanced environment, placed on top of the existing one, and consisting of a Java-based application and a support database. Such framework is capable of supporting the enhanced model, offering an user-friendly interface, and preserving at the same time both technical and economic investments since no radical changes are required in existing environment.

Our re-engineering project is currently being implemented in Catania STMicroelectronics facility and we are collecting response data in order to quantify how productivity is improved through relieving users of current error-prone and wasting-time works. Evaluation of productivity improvement requires a medium - long term analysis, since a relevant factor lies in personnel training, so migration towards new model may require a certain time to reach a stable state in which time and costs are really reduced.

Finally, future works we planned concern:

- the creation of a model for evaluation of productivity improvements, in order to quantify time and cost reduction;
- the use of java application as an applet in a multiuser environment, where a proper transactions management [A. Cichocki et al., 1998;F. Leymann, 1999] is required;
- the re-examination of the enhanced model through a workflow perspective [A. Cichocki et al., 1998;F. Leymann, 1999]];
- extending the object-oriented approach used here to model the entire production model shown in Fig. 1;
- the description of enhanced model with XML [13], to improve model management inside ST intranet, e.g. providing different views of enhanced model through XSL [19], in order to provide each working group with its specific model representation.

6. References

A. Cichocki et al., 1998, Workflow and process automation: Concepts and Technology – Kluwer.

F. Leymann, 1999, Production Workflow: Concepts and Techniques – Prentice Hall.

E. Byrne, 1992, A conceptual foundation for software re-engineering – in Proc. of the Int. Conference on Software Maintenance.

Elliot J. Chikofsky and James H. Cross II, 1990, Reverse Engineering and Design Recovery: A Taxonomy - IEEE Software.

S. Gossain et al, 1998, Object Modeling and design strategies - Cambridge Press, 1998.

Bertino E. and Martino L.R., 1992, Object-Oriented Database Systems – Concepts and Architecture – Addison Wesley.

ST Microelectronics – http://www.st.com.

Longheu A, 1999, ST Microelectronics Manufacturing model - Technical internal Report No. IIT-1999-43.

Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 1999, Modeling a Manufacturing Model using Object-Oriented approach – Proc. of CSCC '99.

Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 2000, Re-engineering the ST Microelectronics manufacturing model – Proc. Of CSMR 2000.

Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 1999, Object-oriented approach in production flow modeling – Proc. Of Workflow management 99.

XML – http://www.w3.org/XML.

Carchiolo V., Longheu A., Malgeri M. and D'Ambra S., 2000, Using XML to model an object-oriented manufacturing process – submitted to XOT.

Java – http://www.javasoft.com.

Kim W. et al., 1989, Composite object revisited - in Acm Sigmod.

Katz R., 1990, Towards a unified framework for version modeling in engineering databases - in Acm Computing Surveys, 22 No.4.

Workstream - http://www.consilium.com/products/ws.htm.

CORBA - http://www.omg.org/corba.

XSL – http://www.w3.org/XSL.