50

Dynamic Behavior of Objects in Modelling Manufacturing Processes

José Barata; L.M. Camarinha-Matos
Universidade Nova de Lisboa - Dep. de Engenharia Electrotécnica
Quinta da Torre, 2825 - Monte Caparica, Portugal
Tel +351-1-3500224 / 2953213 Fax +351-1-2957786
E-mail: {jab,cam}@fct.unl.pt

Abstract

Various modelling perspectives for manufacturing systems, both from the structural and dynamic behaviour points of view, are discussed. The utilisation of object-oriented and frame-based paradigms in this modelling context is discussed as well as the connection of models to the real device controllers. The synthesis of control programs from a Petri net model is also presented in this general modelling framework. Finally the concept of object migration is introduced as an approach to deal, in a flexible way, with moving objects in manufacturing systems.

Keywords

Modelling, Object Behaviour, Flexible Manufacturing Systems, Object Persistency, Views.

1. INTRODUCTION

Object Oriented and Frame based (OO&F) techniques have been intensively used in modelling manufacturing systems and processes. OO&F provides a structured modelling approach, allowing for multiple levels of abstraction, a convenient approach for complex systems modelling. However an extended view of objects is necessary in order to capture the dynamic behaviour of such systems.

Particularly in the area of shop floor control, OO&F can be used to model the various manufacturing agents -- robots, NC machines, transportation systems or even continuous processes equipment. An adequate combination of reactive programming and client-server architecture allows for an effective link between the OO model and the local controllers, therefore capturing the dynamic behaviour of the modelled devices and providing a kind of dynamic persistence. Classical real-time aspects, like asynchronous events / interrupts, device drivers, etc., may be adequately modelled/abstracted using such extended OO&F approach.

In this paper various modelling situations are described and experimental results discussed.

Q. Sun et al. (eds.), *Computer Applications in Production Engineering*

2. MODELLING ASPECTS

The Object Oriented paradigm or its "mate" Frame-based/Reactive Programming represent a convenient tool to model the inherent complexity of a manufacturing system. This complexity comes from the amount of relationships among components in association with the diversity of components. On the other hand, the topic of modelling is a pre-requisite for systems integration in CIM. Speaking specifically about system controllers there is a need for a model that supports the interaction between high and low level controllers, and, at the same time, supports the configuration of new systems.

The model should emphasise the relationships among the various components in the cell and hide the specificity's of the hardware. This last item can be easily achieved with the Object Oriented/Frame based paradigm using methods or demons. Methods associated to the component can hide the underlying hardware infrastructure. Another important aspect is the "relation" concept which can provide a flexible way to describe inter-components relationships. This concept has different semantic meanings in the Object and Frame paradigms, which makes modelling a little bit different in these two frameworks.

Before going further in the modelling discussion, it is important to say a few words about which paradigm should be used to model - Frames or Object Oriented. It isn't an easy choice because both have advantages and disadvantages. The fact that a Frame deals with objects as prototypes, allowing dynamic change of the structure of those objects is a good point, especially during the research phase. But this could be also a disadvantage, especially for software engineering production. For instance, a programming environment that doesn't provide strong type checking can be very error prone and lead to software difficult to maintain. On the contrary, these could be the virtues of the Object Oriented paradigm. On the other side, OO systems are quite limited in terms of definition of new relations with customised inheritance mechanisms.

2.1 Structural aspects

The various examples to be discussed in order to introduce the main modelling concepts will use a cell as the basic modelling unit. A cell is a composite entity that is capable of making some transformation, movement or storage related to some product or part. In structural terms, each cell (C) has components to support the input of parts (I), an agent to perform the transforming actions (A) and components to support the output of products/processed parts (O). Therefore, a cell is the tuple: C = (I, A, O).

Parts input and output and the agent will be supported by manufacturing components. Some components can only support one function but there are others components which can support more than one function. Components adapt themselves to the roles they can perform. Some components are more adaptable than others. For instance, the Conveyor is very flexible because it can perform an input, output or agent role, but a CNC machine only can play an agent role.

Components	*Input*	*Output*	*Agent*
Vibrator Feeders	√		
Buffers	√	√	
Indexed Table	√	√	
Gravity Feeder	√		
Conveyor	√	√	√
Robot			√
CNC machine			√
AGV			√
Positioning device	√	√	

Table 1 Components and their possible roles

The generic cell concept can be specialised by activity. There can be cells specialised in assembly, painting, welding, storage, machining, transportation, etc.. A shop floor is just a set of specialised cells.

Metaknowledge should be associated with each specialised cell to represent the specificities of its application domain. For each domain the specific cell has the same structure as the generalised Cell concept (Input Agent, Processing Agent, Output Agent) but the domain and carnality of the implementing components is different in each specialisation. For example, in a Painting or Welding Cell, a vibrator feeder is not a valid Input item, but this component is valid in an Assembly Cell. The Metaknowledge seems to be a very important element at the configuration phase, assuring the validity of cells.

```
FRAME  CELL
name:
base_coordination_system:
processable_products:
input_parts:
connected from:
processor:
connected to:

FRAME  ASSEMBLY-CELL
is-a: CELL
val-inp-ag: vibratory_feeder,
            buffer,
            gravitic_feeder,
            Index_Table, agv,
            conveyor
val-out-ag: conveyor, agv, buffer,
            index_table
val-proc-ag: robot
```

```
FRAME  ROBOT_COMPONENT
is-a: manufacturing_component
Base_coordinate_system:
Controlled by:
Applications: assembly, gluing, ..
DOF: 6
Working_area:
Load:
Repeatability:
Current_position:
Cost:
Cycle_Time:
Next_maintenance:
N_working_hours:
Weight:
Max_speed_by_axes:
```

Figure 1 Example concepts of cell, assembly cell and component.

At this stage it is convenient to clarify the concepts of agent, input and output, and their relation with the components/manufacturing resources.

Components are entities which participates in the productive process with a specific function and can be controlled by a computational entity. Components models are context independent description of its static and dynamic characteristics. A robot component model, for instance, includes all the characteristics which completely characterise its structural and dynamic aspects.

A robot **agent** (Figure 2) is a model of a robot and associated resources, like tools or auxiliary sensors, when inserted in a particular context. A robot can play different **roles** in different **contexts**. The (expected) behaviour of a robot in an Assembly context is different from its behaviour in a spot welding context.

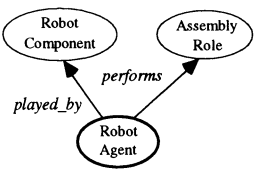

Figure 2 Structure of a robot agent.

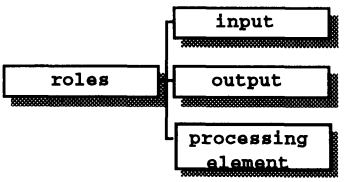

Figure 3 Role taxonomy: main level.

On the other hand, when a robot is performing a given role, it may resort to auxiliary resources, like tools, sensors, buffers, etc., that extend the robot functionality in order to fulfil the functionality required by this role. A robot agent is, therefore, a model of the robot when playing a particular role and extended by selected attributes inherited from the auxiliary resources.

The entity that effectively participates as an assembly robot, for instance, is one which has those characteristics from the robot component model to perform the assembly role.

The agent entity ASSEMBLY_ROBOT is a structure which is supported by two relations: *performs* and *played_by*. The relation *performs* assures the inheritance of the role characteristics to the structure while *played_by* assures the inheritance of those agent relevant aspects, from the component. *Main_attributes* and *component_attrib* are attributes to be used by *played_by* and *performs* relations.

```
FRAME  ASSEMBLY_ROBOT
is-a: agent
performs: ASSEMBLY_ROLE
played by: ROBOT_COMPONENT
main_attributes:force_sensor,
                current_tool,
                available_tools,
               available_resources
component_attrib:Base_coord_system,
Controlled_by,
Working_area, load,
Current_position
```

```
FRAME  AG_ROBOT_ASSEMBLY_ROLE
is-a: role
tools_domain:(grippers,
              screwdriver)
aux_res_domain: (buffers)
force sensor:
current tool:
available_tools: gr1, gr2, sd2
aux_resource: buf1, buf2
assembly_device: fixture1
```

Figure 4 Example concept of an agent

The slots, *tools_domain* and *aux_res_domain* represent domain-knowledge that is important during configuration time. The slot *current_tool* is a relation that associates the main player of this role (robot component) to a particular tool. It could be defined as:

new_relation(current_tool, transitive, inclusion(tool_operations, attached_to)

where *attached_to* is the inverse relation. By the "inclusion" restriction, only tool_operations will be inherited by the ag_robot_assembly_role.

Assembly_device is an attribute describing where assembly operations are really done. *Fixture1* is an instance of a component specialized in holding parts.

```
RELATION  PERFORMS
type: intransitive
inherit_slot: main_attributes
inverse_relation: performed_by
```

```
RELATION  PLAYED_BY
is-a: relation
type: intransitive
inherit_slot: component_attributes
inverse_relation: play
```

Figure 5 Definition of relations *performs* and *played_by*.

A cell is made of entities that are playing different roles.

This modular approach to cell representation facilitates the creation of complex systems by simple "concatenation" of cells. A particular manufacturing unit is made of several subsystems (Transportation Cells, Painting Cell, Assembly Cell, ...). A manufacturing unit could be modelled by a SYSTEM entity, which has access to all characteristics and functionality of all subsystems involved in the Unit.

The way applications see the unit varies with their needs. An application concerned with maintenance activities has different needs from SYSTEM than an application concerned with supervision activities. These differences could be easily supported using the view concept. Using this concept an application only sees the relevant information for its activity.

This is a very convenient concept because it supports information structuring and consistency. Thinking of a robot, an attribute that accumulates its number of working hours is important for a maintenance application, but it could be irrelevant for a direct control application.

We can even think that applications in the same activity area, i.e., accessing the same view, could have different requirements. In this case the access is determined not only by the role of the client but also by its status. Applications may have different status to access a view entity, having conditioned access determined by their status.

These concepts are not easily implemented with current OO&F technologies. The views implementation is different whether it is implemented by frames or by objects. The UNL Robotics group developed some implementations using frames that support this concept. The frame implementations are based on the inheritance and relation mechanisms.

Object Oriented languages should be extended in order to include a create_view constructor that could be related with another new construct VIEW. Using an example in EIFFEL the result could be:

```
CLASSE  robot
interface
maintenance VIEW
 total_hours()
 hardhome()
 ...
END maitenance
operative VIEW
 hardhome()
 move()
 ...
END operative
configuration VIEW
 load()
 max_speed()
 ...
END configuration
END robot
...
r1,maint_view,op_view: robot;
...

r1 := create(robot);
maint_view :=
   create_view(robot,
                maintenance);
oper_view :=
    create_view(robot,
                operative);
```

2.2 Dynamic aspects

Dynamic aspects are related to the components internal state changes. The dynamism presented by components is achieved through controller actions. Every component with dynamics must have a controller associated with it. The main discussion is not centered in the aspects related with the physical components changes, i.e., it isn't important to know what are the inertial conditions associated, for instance, with a robot movement, but it is important to discuss the functional behavihour of the physical component being modelled, i.e., it is important to know what actions should be done in order to move the robot in the most flexible way. The way the model reflects component physical changes and the way physical component reflects model changes is the most important point when discussing dynamic aspects.

Dynamic aspects can also be discussed with two different views: (1) considering the components as isolated entities or (2) considering complex structures, like cells, made of components. In the first view the key point is how components are actuated, without any concerns about their interrelationships. In the second view, aspects related with synchronisation are the most important ones (it will be analysed in the PETRI NETs chapter). In this point the concern is with the first view.

Every component model with behaviour should have a controller model. This model should be like an image of the real controller. Using a frame oriented paradigm, the controllers functionalities could be defined by methods. In this way most of the controller's model is a list of methods, a method for each functionality.

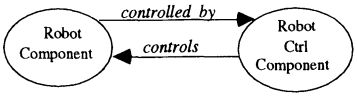

Figure 6 Controlled_by and Controls relations.

A component is related to its controller by a *controlled_by* relation while the controller relates to its component by a **controls** relation.

```
RELATION  CONTROLLED_BY
is-a: relation
type: intransitive
inherits: inclusion (move_wc,
                     move_jc,
                     hardhome,
                     acceleration,
                     speed)
inverse_relation: controls
inverse_relation: controls
```

```
FRAME  ROBOT_CTRL_COMPONENT
is-a: controller
move_wc: method move_wc_fn(x,y,z,q)
move_jc: method
          move_jc_fn(m1,m2,m3,m4)
hardhome: method hardhome_fn
acceleration:demon if write accel_dem
speed: demon if write speed_dem
input: byte demon if needed input_dem
output:byte demon if write output_dem
```

Figure 7 Model of a component and its controller.

One of the most important points in this discussion is the way a controller model is connected to the physical controller. This connection is sometimes not easy because it involves the cooperation of two different computational worlds: the computational world where the model runs and the real controller. To make things even more difficult, sometimes, real controllers have closed architectures. From our experience a lot of effort has usually to be put in trying to open real controllers architecture, and implies the production of an interpreter that runs on it. This interpreter accepts commands from an image that runs in the other world.

The methods of a Controller model implement the actions that are needed to send the right commands to the real controller. The real controller image should be developed using a client-server approach. In this way, implementation methods can ask this server to perform the needed actions. These methods hide the underlying hardware structure from the application, i.e., any application using a robot component doesn't need to know anything about the real robot controller and its image or server. The applications only know what functionalities are provided by the robot component model. This approach could be very suitable to integrate existing controllers, making the integration of legacy systems an easier task.

3. DYNAMIC PERSISTENCE OF OBJECTS

In a manufacturing environment, many information sources -- sensors, state variables of local controllers , etc. -- have their own "life", independent of the computer that is running the general controller model, because they have local processing power. This may lead to the concept of dynamic persistence, that will be introduced and exemplified in modelling manufacturing systems.

Object Persistence is the property of extending the life of an object beyond the running session of the application software that created or changed it. This characteristic is important for applications that may interact with long lifetime objects.

The traditional way of dealing with Object Persistence is storing the objects in secondary memory. In some approaches, classical Database Management Technology has been integrated with OOP languages in order to manage the flow from main to secondary memory and vice versa.

The concept of Dynamic Persistence of Objects is not very different from normal persistence. The basic difference comes from the way persistence is supported: by the local memory of devices' controllers. The use of reactive programming (demons) and methods to "link" the object model to the real cell controllers allows for a permanent update of the dynamic object's model. In this way, a special kind of persistence is achieved - **dynamic persistence**. It is dynamic because the object model reflects, at every time, the status of the physical object. The persistence is assured by the "memory" present in the device controller. There is a tight connection between the object "living" in main memory and the physical controller. We can say that the object virtualizes de physical controller. The physical entity description (object) is connected to the physical entity via demons associated to object attributes. These demons establish a communication link to the physical entity controller.

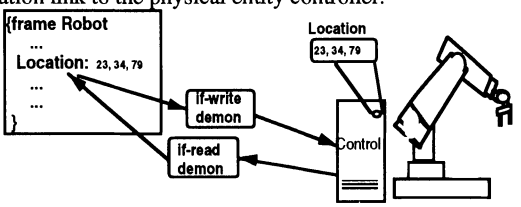

Figure 8 Use of reactive programming to support dynamic persistence

In our work an assembly cell composed by a SONY Scara robot, a fixture, a force sensor, an automatic tool exchanger and two gravity feeders was integrated in a UNIX environment using this approach. However, due to SONY robot's closed architecture a big amount of effort was needed in order to allow this integration. It implied "breaking" the protocol used to down and upload programs, and also the development of an interpreter that runs on top of SONY's own controller. To make error recovery possible (guarded moves) it was also necessary to "break" the teach pendant protocol and to replace it by a PC controller. A controller was also included to drive the force/torque sensor.To connect these external controllers into the UNIX environment it was necessary to develop their "images" on the UNIX side. Each of these "images" are accessible to applications according to a client-server basis.

The same methodology is being used to integrate the several agents belonging to a complex manufacturing cell installed in our facilities. Due to cell's complexity this work is being done by phases. A BOSCH scara robot, a conveyor belt system and an automatic warehouse have been already integrated using this approach.

4. PETRI NETS

Petri Nets are important tools to model the structure and behaviour of controllers and application programs. Complex system dynamics can be described and analysed in a structured way (mathematical methods). There are several types of Petri Nets, which can be used to model distinct types of systems, but Predicate Transition Nets (PTN) seem to be very suitable to model logic controllers. A PTN has predicates associated with transitions which only fire when all input places has marks and the predicate returns true.

The benefits of using a high level modelling tool, like a PTN, shouldn't end in their descriptive characteristics but there should be a direct connection between the description and the real controllers. This means that the model could drive directly its associated physical controller. To assure this connection two different approaches could be used: (1) using PTN to directly program the physical controller, which implies a support by its manufacturer or (2) using a PTN translator which converts PTN to the own language of the physical controller.

The first approach is unrealistic at current stage because the concept of PTN is not well disseminated among controllers' manufacturers, which makes the second approach a better one.

The PTN description should be compiled in order to generate a program which interact with the real system in the way described by the PTN.

The application program or High Level Control Program (HLCP) is generated from a PTN which describes components behaviour and interactions. The HLCP interacts with the execution infrastructure, mainly with the components models. These models are described using an Object/Frame paradigm. As mentioned before, the components behaviour is implemented by methods, which interact with the component's physical controller through a server which supports an image of the physical controller functionalities.

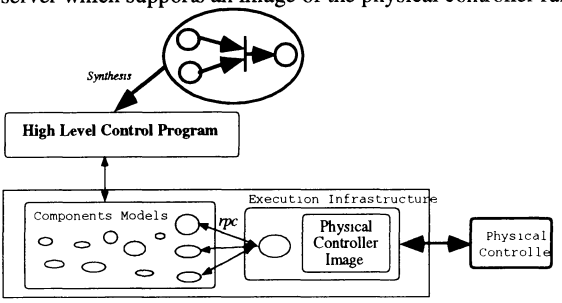

Figure 9 - Interactions between HLCP and Execution Infrastructure.

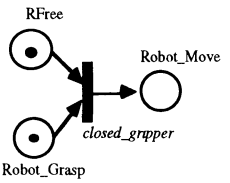

Figure 10 A Petri Net example.

The components model associated to a HLCP include only those models relevant for the program that is running in the HLCP. Saying it by other words, only those components which appear in the PTN will be included in the components model associated with HLCP. It should be noted that a Component Model can interact with more than one server, depending on the number of needed controllers to control the components being used. For instance there can be different servers to control de robot, the gripper, gravity feeders, etc.

In order to generate the HLCP from a PTN some considerations should be made about PTN. Components actions can only be done in places with marks. Predicates associated to transitions specify conditions. In order to match places to components actions, places' names include components' name and action name separated by an underscore (for instance, the name for the place that describes the grasp action of a robot is named **robot_grasp**).

The HLCP program generated is a simulator of the PTN being modelled. Different PTNs may have the same kind of simulator, differing only in which order transitions will be fired and which actions will be done.

Program generation was developed using Prolog, and the generated program is also described in Prolog with a frame extension developed in our group- Golog. The generated program can be seen below.

The first section of the generated program is concerned with place definition; every place is defined by an object/frame whose main attribute is the slot mark to store the place's mark value. During this phase the program that contains the components model is consulted.

```
:- consult('models.pl').
:- new_frame(places), new_slot(places, mark).
:- new_frame('RFree'), new_slot('RFree', isa, places), new_slot('RFree', mark, 1).
:- new_frame(robot_grasp),new_slot(robot_grasp,isa,places), new_slot(robot_grasp,mark,1).
:- new_frame(robot_move),new_slot(robot_move,isa,places), new_slot(robot_move,mark,0).
```

After this, transitions are defined. Each transition is defined by checking its enabling condition. When this occurs, input places are updated and transition fires with output place updating and the corresponding method activation: *call_method(robot, move, [true])*. This method's code will send a message to the server which will react by sending the command "move" to the robot.

```
t1 :- get_value('RFree', mark, X0), X0 > 0,
      get_value(robot_grasp, mark, X1), X1 > 0,
      NVal0 is X0 - 1, new_value('RFree', mark, NVal0),
      NVal1 is X1 - 1, new_value(robot_grasp, mark, NVal1),
      call_method(robot, move, [true]),
      get_value(robot_move, mark, VOal0), NVOal0 is VOal0 + 1,
      new_value(robot_move, mark, NVOal0).
```

The main program consists of a forever cycle that continuously apply the existing transition names and randomly choose one which will be checked for its enabling condition.

```
rep_run([]).
rep_run(List) :- length(List, Tam), Pos is ip(rand(Tam)), position(Pos, List, Tr),
                 remove(Tr, List, RList), call(Tr, Success), !, fail == Success, rep_run(RList).
run :- repeat, rep_run([t1]).
```

This generated program run with a similar behaviour as the PTN shown in figure 10.

5. OBJECT MIGRATION

In a FMS/FAS System several distinct physical "worlds" may be considered. Assembly cells, transportation systems and automatic warehouses are examples of existing "worlds" in a shop floor environment that have their own controllers (i.e. distinct computational worlds !). These "worlds" are strongly interconnected requiring information exchange, which could be achieved by sharing a centralised repository, by messages or by moving data among "worlds".

Viewing the associated computational "world" as a set of objects which model the physical entities participating in the process, the concept of **object migration** becomes relevant. A computational "world" can include objects which belong intrinsically to that "world". For instance, the object robot belongs intrinsically to the assembly cell "world", but the object pallet doesn't. A pallet migrates between worlds. This object doesn't belong to any specific "world" and can "enter" different "worlds" at different time slots. Taking into account the need of these objects in a FMS/FAS system it would be necessary to develop an infrastructure to support object migration.

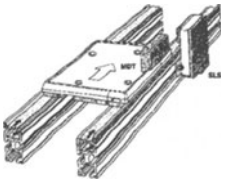

Figure 11 SLS and MDT.

```
FRAME   MDT_OBJECT
is-a: migration_object
pallet_id:
kind_of_pallet:( cnc_raw_material,
                 cnc_finished,
            assembly_raw_material,
               assembly_finished )
materials_list:
path:
stop_places:
```

Figure 12 Object stored on pallet's memory.

As in our pilot manufacturing system, the BOSCH pallets include an attached memory device - MDT, that can be read/written by various other devices - SLS, located in special places of the system, we have a "physical" support to this object migration (figure 11, 12).

Regarding again the object pallet, it could be seen that while it moves through the "worlds", it can be modified. The modification in the object's structure reflects the changing conditions in the physical "world". At each time slot the object's state reflects the operations done by physical entities, over the moving physical entity, represented by these migration objects.

The attribute *materials_list* includes identifications of those objects carried on the pallet. Pallet's path movement within the cell is controlled by the attribute *path*, which includes a list of conveyor names. The attribute *stop_places* tells the system where the pallet should be stopped.

When a pallet passes in front of an SLS, it reads the contents of the MDT structure in order to determine which action to be done. SLSs are controlled by a PC server which is directly connected to the PLC that controls cell's conveyors. Depending on the MDT's memory contents, the SLS sends commands to the PLC. For instance, if there is a stopper nearby the SLS and the *stop_places* attribute has the name of this SLS, the server commands the PLC to stop the pallet. This is a highly dynamic system, either at the spatial or at the internal structure levels. However, these aspects of migrating objects are still on a developing phase in our system, needing a deeper evaluation.

6. CONCLUSIONS

In this paper we discussed various aspects of modelling manufacturing systems, both from the structural and the dynamic perspectives, resorting to the object/frame based paradigms.

In particular, our experimental results have shown that objects' dynamic behaviour (by means of reactive programming and methods) combined with a client-server approach, provide an effective way to link models with local controllers of manufacturing devices. Therefore this can be a suitable approach for migrating from legacy systems to more integrated high level control systems. The generation (synthesis) of application control programs - directly linked to the above methods from a Petri Net description was also discussed. Current work is addressing the aspects of object migration as a flexible way to "deal" with moving objects in a manufacturing environment.

Acknowledgements
This work has been funded in part by the European Community (Esprit project B-Learn and FlexSys) and JNICT (projects SARPIC and CIM-CASE). We also thank Mr. João Carlos Silva and the students Eduardo Bras, Sandra Gadanho, Luis Fernandes and Nuno Chagas for their contribution to the experimental setup.

REFERENCES

Barata, J.; Camarinha-Matos, L.M.; Rojas Chavarria, J.F. (1994) — Modelling, Dynamic Persistence and Active Images for Manufacturing Processes, *Studies on Informatics and Control* , vol. 3, nº 2-3.

Camarinha-Matos, L.M. (1989) — *Sistema de programação e controle de estações robóticas - Uma arquitectura baseada em conhecimento*, PhD Thesis, Universidade Nova de Lisboa, 12 Jun. 1989.

Camarinha-Matos, L.M.; Negreto, U.; Meijer, G.R.; Moura-Pires, J.; Rabelo, R. (1989) — Information Integration for Assembly Cell Programming and Monitoring in CIM, *21st ISATA*, Wiesbaden.

Camarinha-Matos, L.M.; Osório, A.L. (1990) — Monitoring and Error Recovery in Assembly Tasks, *23ª ISATA*, Viena.

Camarinha-Matos, L.M.; Pinheiro-Pita, H.J. (1990) — Interactive Planning of Motion and Assembly Operations, *Proc. IEEE Int. Workshop on Intelligent Motion Control* , Instambul, Turkey.

Camarinha-Matos, L.M., L. Seabra Lopes, J. Barata (1994) — Execution Monitoring in Assembly with Learning Capabilities, *Proc. of the 1994 IEEE Int'l Conf. on Robotics and Automation*, San Diego.