A Design Pattern for Model Based Software Development for Automatic Machinery

Cesare Fantuzzi * **Marcello Bonfè** ** **Cristian Secchi** *

* *Università di Modena e Reggio Emilia, DISMI, Via Amendola, 2, 42100 Reggio Emilia - Italy, E-mail: cesare.fantuzzi@unimore.it.*
** *Università di Ferrara, Dip. Ingegneria, Via Saragat 1, 40100 Ferrara - Italy. E-mail: marcello.bonfe@unife.it*

Abstract:
The paper presents the results of the application of object-oriented modeling techniques to the control software design of complex manufacturing systems, with particular focus on automatic machineries for production and packaging of food stuff, as milk, snacks, etc.
In this application fields there are some peculiar problems to tackle in order to develop effective software control solutions, as for example the exception handling caused by product or packaging material jam, the Human Machine Interface, the recipe production management etc.
The goal of this paper is to introduce design patterns developed in the framework of UML applied to the development of automatic machineries software, aiming to define a set of predefined modeling solutions to some class of recurrent design problems.

Keywords: Industrial control, PLC, UML, Object-oriented

1. INTRODUCTION

In recent years, the *Object–oriented* (O–O) modeling techniques Rumbaugh et al. (1991) gained wide attention both by the scientists and the practitioners as a design methodology that allows to tackle complexity in the development process of mechatronic systems (i.e. mechanical systems tightly coupled with their controllers), as automatic machineries.

The reasons of such a interest is in the effectiveness of O–O basic concepts (abstraction, encapsulation, inheritance) in managing complexity in many *business software* applications. Thus it was a popular matter of research in the past years to find if introducing O–O principles and modeling would carry the same advantage in the field of industrial control software development (see Storr et al. (1997); Benitez Pina et al. (1999); Maffezzoni et al. (1999); Bonfatti et al. (2001); Frey and Litz (2000); K. Thramboulidis (2008)).

Unified Modeling Language (UML) Object Management Group (2007) published by the Object Management Group (OMG), became a standard for O–O methodology deployment, because it is well known and accepted in many application fields, and, moreover, several software tools for UML diagrams development are available ready on the shelf, some of them also for free.

UML defines a set of graphical notations (nine different kind of diagrams) to describe both architectural and behavioral aspects of systems of any kind, but is not necessarily associated to a specific methodology. In fact, UML has been developed to support OO analysis and design methods with a common language that can be used in any phase of the development process, from requirements specification to detailed design. For example, system requirements can be expressed with the help of *Use Case Diagrams*, structural views of the system can be described by means of *Class Diagrams*, interaction views by means of *Collaboration Diagrams* or *Sequence Diagrams* and the event-oriented dynamic behaviour of each software component can be specified with a *Statechart*. Moreover, UML itself is defined by means of an extensible *metamodel*, which allows the designer to adapt the syntax and semantics of the language to a specific application domain, by developing *stereotyped* elements and modeling constraints expressed with *OCL* (Object Constraint Language, Object Management Group (2007)), extending the basic concepts of classes, objects, attributes and so on, in order to represent more precisely domain-specific features.

The main issue in object oriented modeling have been focused on *software static structure*, described by means of UML Class and Object Diagrams, and *software dynamic behavior*, developed by means of UML Statechart and Sequence Diagrams. In particular, attempt to describe mechatronic system dynamic behavior by means of state machine started well in advanced with respect to the introduction of Object-Oriented approach.

State machine languages as Grafcet David and Alla (1992) and Sequential Function Chart (SFC) International Electrotechnical Commission (2002) are both derived from Petri Nets David and Alla (1992), in particular from the subset of *safe* Petri Nets, but they have been enhanced with extensions for input/output interpretation and data/timing processing, which are necessary for practical software implementation.

Moreover, control design models must be graphically represented in modular and hierarchical forms, in order to be of practical use in real cases. In fact, both the languages mentioned above and, among others, the one described in Frey (2001), admit hierarchy between sub-nets, while some authors have formalized modular extensions for traditional Finite State Machines Shah et al. (2002) or adopted specification languages derived from Object-Oriented (O-O) Software Engineering methods Bonfatti et al. (2001); Storr et al. (1997).

Another key issue, in the industrial practice, is the design and representation of exceptions handling mechanisms. In

that sense, it is worth to remark that the visual language of Statecharts Harel (1987), which has been embodied in UML specification which also supports hierarchy and concurrency, have a significant advantage over Grafcet or SFC, which derives from the possibility to define explicitly inter-level transitions (i.e. transitions crossing the boundaries of states hierarchy), very useful to model preemption of subsequences and recovery from faults or alarms.

Some authors have tried to improve the expressiveness of Grafcet/SFC-like diagrams Johnsson (1999), or to integrate SFC and Statecharts Bauer and Engell (2002), in order to fill this gap, while some others have developed translation mechanisms to design PLC programs directly with Statecharts Machado et al. (2001); Chester et al. (1998).

Currently, the programming languages adopted by PLC producers are those described in the international standard IEC 61131-3 International Electrotechnical Commission (2002), of which the most used is Ladder Diagram, a low-level language that, without a proper definition of modular structures, would make even programs for small machines quite complicated.

However, as reported by several authors, the straightforward translation of O-O techniques from Business domain into Industrial domain is not possible, mainly because the following reasons:

- *It allows to take into account the peculiarity of the* ***mechatronic*** *context.* To manage system complexity, the object decomposition must be carefully shaped, that means that objects have little interactions or these interactions can be precisely defined. Here is the point where the O–O paradigm comes in hand, as it is helpful to identify the physical objects candidate to represent a reusable module and to describe how their interactions can be modeled in the control program. With this approach, the electronic and mechanical parts of the machine modules becomes so tightly coupled together and with their control software, that are referred as *mechatronic objects* Bonfè et al. (2001).
- *The language implementation constraints.* As matter of fact, devices like Industrial Controllers (i.e. PLCs) are historically programmed with low-level languages, with poor modularization and information hiding facilities and, especially, strictly tied to the hardware producer.
- *Practical approaches to software developments.* The lost of the encapsulation bounds for the designed software modules is in the application domain a disadvantage, since many technicians working on manufacturing machine test and maintenance *on the field* are able to read and modify only the PLC code directly running on the installed system, either because of cultural background or because of practical reasons.

Starting from the above notes, the work described in this paper introduces proper *design patterns* (a unified solution to common design problems) based on O–O paradigms which can support the design engineer in the whole development and implementation process.

The entire work came from a strict collaboration between academic and industrial partners, in which has been considered a complete development process, from the conceptual thinking and preliminary requirement capture, to final code generation. The process targets to describe a generic control structure (e.g. compliant to IEC61131-3 standard for the development of industrial controllers), which is based on an "Industrial Automation" UML profile (introduced in Sec. 2). Some details of the implementation in an industrial packaging machine platform are presented in Sec. 3. Sec. 4 present some conclusive remarks.

2. AN OBJECT-ORIENTED DESIGN PATTERN DESIGN METHODOLOGY FOR MANUFACTURING SYSTEMS CONTROL

The efficiency of machine engineering process would be greatly enhanced by systematic approaches that consider modularity and reusability of the control software as cardinal principles. Therefore, the control design activity should produce a model of the software application that is *modular*, *precise* but *easy to understand* (especially for engineers with different backgrounds), *correct* and, finally, *easy to implement* in the target programming platform. At the end, but not less important, the design specification model would be an efficient support also for the testing, documentation and maintenance phases.

An important aspect that must be discussed is the correct interpretation of "objects" in the domain of manufacturing systems. In fact, a manufacturing machine module would have reusability features if it consists of a tight aggregate of mechanical parts, sensors, actuators and control software routines, specifically related to a given part of the manufacturing process.

In this context, it is valuable to define a module of that kind as a *Mechatronic Object*. With regards to objects interactions, several authors (e.g. Selic et al. (1994)) suggest that the most suitable interface for a real-time software module should be based on signals and events, rather than explicit operation invocation. Therefore, a "mechatronic object" should have a signal-based interface, in order to send and receive synchronization and interlocking information, rather than an operation-based interface, like remote procedure calls, typically used in busness software environments.

From the above considerations, the software part of a "mechatronic object" should fatures the following characteristics:

- an input/output interface of signals and data parameters that permits the interactions and synchronization with the rest of the control system.
- an input/output interface of sensory information and actuator commands required to control the physical part of the mechatronic object; this interface should be considered as a "private" part of the mechatronic object.
- a private data set to store status information.
- a control algorithm, which determines the dynamic behaviour of the mechatronic ensemble, in order to match the functional requirements.

The features described above are fulfilled by the Function Block (FB) software structure, as described primarily by IEC 61131-3 and extended in IEC 61499-1 for distributed architectures. Even if the two documents actually propose a different execution semantics Bonfè and Fantuzzi (2003), the generic interpretation of FBs as "mechatronic object" controllers can be schematized as shown in Figure 1.

The notation of UML Class Diagrams can be adapted to mechatronic objects modeling by means of the UML class stereotypes described in the following. The software component that controls a mechatronic object is defined as a class stereotyped as ≪`mechatronic`≫, which is subject to the following rules:

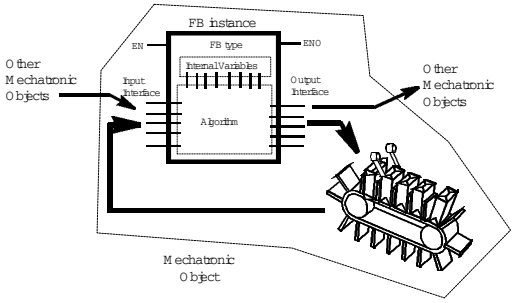

Fig. 1. Schematic representation of the Mechatronic Object concept

- the public attributes of a ≪mechatronic≫ class must be stereotyped as ≪`input`≫, in which case they will be modifiable exclusively from outside the class, or ≪`output`≫, which are instead modifiable only by the internal behaviour of the class;
- attributes not stereotyped as described before cannot have public visibility;
- operations, which may be used to model internal data-processing activities, cannot have public visibility;

The choice to define the external interface of a ≪mechatronic≫ class exclusively in terms of I/O signals is similar to the one adopted in the Real-Time profile for UML described in Selic and Rumbaugh (1998), that introduces the concept of *capsule*, a highly de-coupled and reusable software component, communicating with its surroundings through *ports*. In our approach, each I/O attribute can be viewed as an independent "port", that can be connected with any consistent signal source or sink.

Another extension of the modeling language is necessary to model the I/O connections between the physical part and the software part of a mechatronic object. As said before, these signals should be considered a private part of the object. In order to specify this concept in UML models, it is necessary to introduce another class stereotype, that we have called ≪`hardware`≫. A ≪hardware≫ class should always be related by means of a *composition* link (also called "strong aggregation" or "include by value" relationship) with a ≪mechatronic≫ class and its stereotyped attributes (≪input≫ and ≪output≫) represent the hardware-related I/O ports of the mechatronic object. The notation of the proposed class stereotypes is shown by the UML Class Diagram of Figure 2. Notice that inputs of a ≪hardware≫ class are outputs of the controller and viceversa. The definition of separate class stereotypes for the software part and the hardware part of the mechatronic object, leaves to the implementation phase the definition of an efficient protection mechanism for the physical I/Os of a control module, according to the features of the execution platform (e.g. `VAR_CONFIG` variables, with instance-specific location assignment, in IEC 61131-3 or *Service Interface Function Blocks* in IEC 61499-1).

A complete and realizable structural model for a machine control system should be defined by Class Diagrams in which the hierarchical architecture derived by machine modularization is modeled with composition links between mechatronic classes. Well-formedness rules for mechatronic object models should prescribe that there must be a single *"top-level"* class (i.e. the machine) and composition links must be qualified with fixed and unambiguous multiplicity, since dynamic creation of mechatronic objects is not consistent with their physical interpretation. In this way, the set of control modules for a given machine is

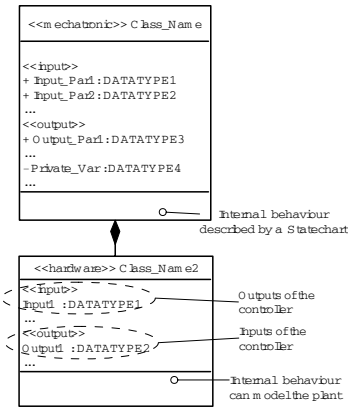

Fig. 2. UML stereotypes for mechatronic object models

exactly determined by the instantiation of the top-level class.

With regards to the specification of mechatronic objects interactions, the syntax of UML Collaboration Diagrams, based on methods invocation, should be modified in order to describe the signal/data-flow interconnections between object instances, with a graphical syntax that emphasize the interpretation of ≪input≫ and ≪output≫ attributes as "ports" of the system components. The notation of stereotyped Collaboration Diagrams, that we have called *Mechatronic Data-Flow Diagram*, should permit to specify the association between inputs and outputs of two objects as textual expressions upon the interaction link, as is briefly described by the simple example shown in Figure 3.

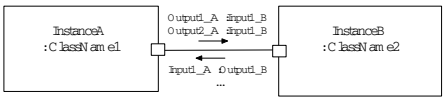

Fig. 3. Interaction between two mechatronic objects

Considering then the development of a generic machine control application, it is important to understand that a manufacturing machine is very often a composition of mechanical parts that handle and process the incoming product to achieve the desidered functionality (e.g. product packaging), acting like a "chain process": those parts act in a sequential manner on the products, and, generally, it's possible to extract a modular decomposition of the overall system from this functional view. This approach leads to develop well defined processing modules (e.g. a cutting module) that can be reused in subsequent projects. So that, the design of a manufacturing machine can be very often connected to the composition of sub-processing solutions developed in previous projects. These modules consist of a well-defined set of mechanical parts, sensors and actuators, which are related to a well-defined set of control specifications.

The general design pattern of module composition can be described by means of a hierachy of a *Machine Controller* which acts as a system global supervisor, that coordinates the behaviour of the physical functional modules described using *mechatronic objects* (Figure 4. The Machine Controller is a virtual entity which describes the macro behaviour of the whole machine, and it acts as a orchestra director which coordinates the action of each performer. In particular, to assure system modularity and interchangability, only communication between Machine controller and each mechatronic object are allowed.

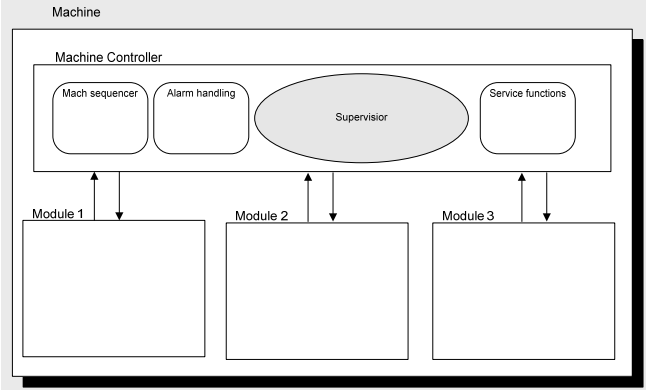

Fig. 4. Machine Controller supervises Mechatronics Objects.

Given the architectural specification of the control system, the last essential part of the model is the behavioural specification. With the proposed approach, the functional requirements of the manufacturing machine are mapped into an operational input/output model of each module of the machine controller. Since these modules are highly reactive, the language of Statecharts, introduced in UML as a variation of the original language of Harel Harel (1987), is perfectly adequate for the behavioural specification of mechatronic objects: each ≪mechatronic≫ class will be associated with a Statechart diagram representing the control action necessary for that module, while the behaviour of ≪hardware≫ classes, which can also be specified with a Statechart, will represent the behaviour of the uncontrolled plant.

The features of concurrency and hierarchy provided by Statecharts are very important for an efficient and intuitive description of complex event-driven dynamics. Moreover, the possibility to define inter-level transitions permit to include very easily in the specification alarms and exceptions (as shown by Figure 5), which is a critical aspect for real manufacturing machines.

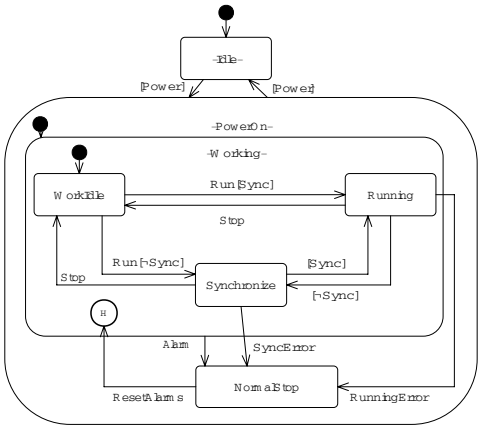

Fig. 5. Example of Statechart

Textual expressions, like transition labels or state actions, needs a more detailed discussion, since they are the elements of the model that have major impact on implementation details. In particular, Statechart's transition labels are, for the most generic case, in the form:

`trigger[guard]/actions`,

where `trigger` is *an expression of events*, `guard` is *a boolean expression* and `actions` is a list of data-processing activities or events "generated" at the firing instant. State actions are instead expressed in the form:

`when / action`

in which `when` is a qualifier that can be `entry`, `exit`, `do` or a specified event, and `action` can be only a data-processing activity. The correct semantics of these textual expressions must take into account the features of IEC 61131-3 and IEC 61499-1, in which the concept of "event" and data-processing activities should be interpreted as boolean equation.

3. APPLICATION

The methodology described in previous section, has been applied for the definition of a design model for the supervisory control of Tetra Brik Aseptic ®packaging machines developed at Tetra Pak Carton Ambient S.p.A., which are complex manufacturing systems whose purpose is to fill special carton packages with liquid products, like milk, fruit juices and so on. The machines we have considered belong to the aseptic type, which means that the packaging material has to be sterilized before filling it with product, and the machine part where the package is actually formed must be kept in sterile conditions. The so-called "vertical filling&forming" packaging process is schematized in Figure 6.

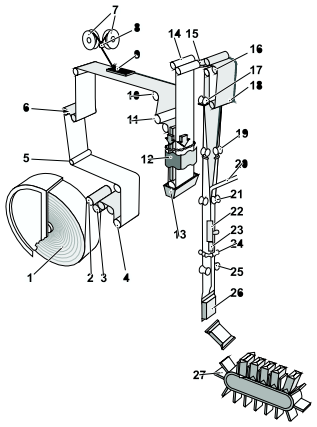

Fig. 6. Packaging material path in a Tetra Brik Aseptic ®machine.

According to the decomposition guidelines described above, the filling machine can be modularized as is described in Figure 7, which shows the top-level hierarchical decomposition according to UML Class Diagram notation. In detail, diamond-head arrows show the *aggregation* relationship between the machine supervisor and each module, while simple arrows shows *generalization* of some modules, which can have several possible variants. The model represents then a family of machines, which differs in the "instantiation" of modules at configuration time.

A very critical set of operations for this manufacturing system regards sterilization and preparation of the packaging material and the filling system, because the packaging process requires filling a tube of packaging material from above and then cutting the package around the product. When production ends, some operations are still necessary before leaving the machine inactive, in particular the aseptic chamber has to be ventilated in order to eliminate dangerous peroxide vapours, and then external and internal cleaning are performed. To describe accurately the machine global behaviour, a statechart for the machine

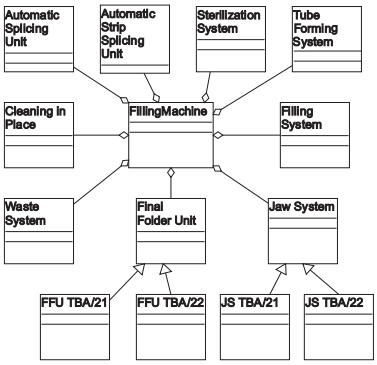

Fig. 7. UML Class Diagram of a Tetra Brik Aseptic ®filling machine

supervisor module has been defined. Figure 8 shows a simplified version of the statechart diagram.

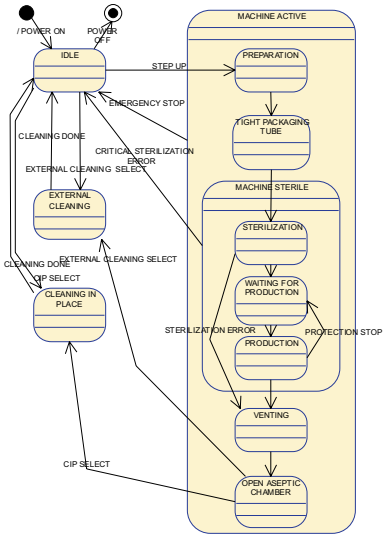

Fig. 8. Global behaviour of the machine considered

The machine modules are activated or deactivated according to synchronization signals generated opportunely by the supervisor module. For example, if the machine is in the "Sterilization" phase, the parts of the machine devoted to cutting and forming packages, namely the "Jaw System" and the "Final Folder Unit", should remain inactive, while during production the "Sterilization System" behaviour is different from that of the initial sterilization sequence. In this way, modules can be considered as indipendent entities, relying only on the supervision module to perform correctly their contribution to the packaging process.

As an example of detailed design for a control module, we can focus on the part of the machine named "Automatic Strip Splicing Unit" (ASSU), whose job is to attach a plastic strip to the packaging material, which is needed to seal the material in order to make the packaging tube. The ASSU must also guarantee continuity of the plastic strip flow, which is made possible by an automatic splicing of the strip contained in two different reels, when the one currently attached to the packaging material is over.

The ASSU module can be further decomposed in three other components, each one with a different role in the strip control functionality. One entity is the sealing head that attaches the strip to the packaging material, one is the motion system that permits a correct unrolling of the strip reel, and another is the sealing head that performs the strip splicing. Internal hierarchy of the ASSU is described by a UML Class Diagram that describes all the entities aggregated to the module, together with their sensors and actuators, as shown in Figure 9.

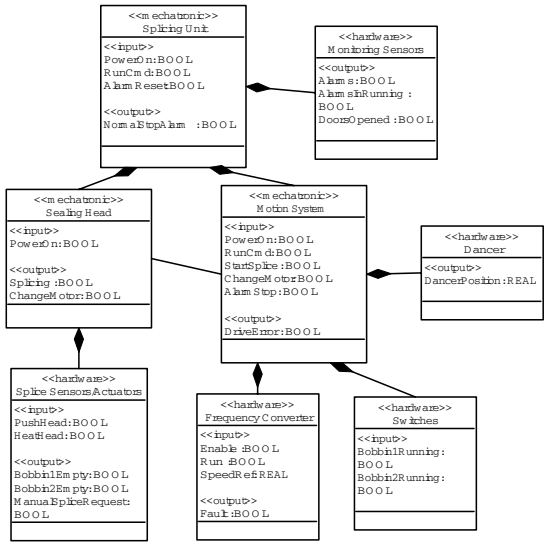

Fig. 9. UML Description of the ASSU Module

To completely specify the behaviour of this particular part of the machine, four statecharts have been defined: one for the module supervision, one for the strip splicing sequence, one for the activation steps of the strip applicator and one to describe the behaviour of the motion control system (the latter three omitted because paper length constraints).

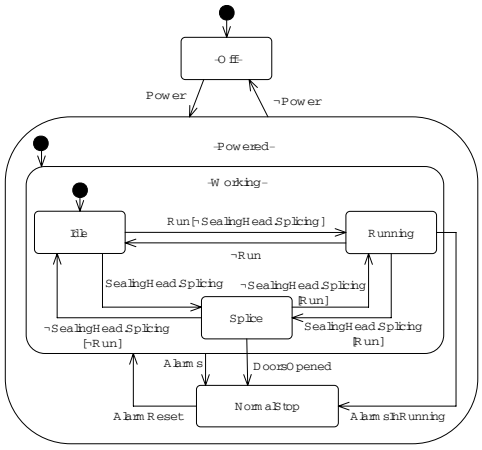

Fig. 10. UML Statechart for the ASSU supervisor

Similar considerations have been applied for the definition of the other modules not described here, obtaining a global description of system architecture and behaviour strongly oriented to modularity and flexibility.

4. CONCLUSION

The paper presented a viable methodology for the application of advanced software engineering methodologies for the design and realization of automatic machineries for smart manufacturing processes.

The requirement for system flexibility, cost reduction and performance improvement has pushed for massive introduction of intelligent components (motion control, smart

sensors, etc.) into the machine design. The methodologies presented in this paper had the objective of defining an advanced techniques to drive the control software programmer in the development stage, to cope with the increasing system complexity, preserving efficiency under real-time constraints.

REFERENCES

Bauer, N. and Engell, S. (2002). A comparison of Sequential Function Charts and Statecharts and an approach towards integration. In H. Ehrig and M. Grosse-Rhode (eds.), *Proc. 2nd Int. Workshop on Integration of Specification Techniques for Applications in Engineering*, 58–69. Technische Universität Berlin.

Benitez Pina, I., Vazquez Seisdedos, L., and Villafruela Loperana, L. (1999). Including object–oriented properties in the plc's programming languages. In IEEE (ed.), *ETFA '99*, volume 2, 1029–1043.

Bonfatti, F., Gadda, G., and Monari, P. (2001). PLC software modularity and co-operative development. In *Proc. of AIM2001*, volume 2, 775–780. IEEE/ASME.

Bonfè, M. and Fantuzzi, C. (2003). Design and verification of mechatronic object-oriented models for industrial control systems. In *Proc. of 9th IEEE Int. Conf. on Emerging Technologies and Factory Automation.*

Bonfè, M., Fantuzzi, C., and Poretti, L. (2001). PLC object-oriented programming using IEC 61131-3 norm languages: an application to manufacture machinery. In *Proc. of the European Control Conference 2001*, 3235–3240.

Chester, E., Bates, I., and Kinniment, D. (1998). Automatic generation of IEC 1131 PLC code from Statemate Statecharts. In *4th International Workshop on Discrete Event Systems, Cagliari, Italy.*

David, R. and Alla, H. (1992). *Petri Nets and Grafcet: Tools for modelling discrete events systems.* Prentice–Hall.

Frey, G. (2001). SIPN, hierarchical SIPN and extensions. Technical Report I19/2001, Institute of Automatic Control, University of Kaiserslautern.

Frey, G. and Litz, L. (2000). Formal methods in PLC programming. In *Proc. IEEE Conf. on Systems Man and Cybernetics (SMC) 2000*, 2431–2436.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 231–274.

International Electrotechnical Commission (2002). IEC 61131-3. Programmable Controllers - Part 3: Programming Languages (2nd Edition). Final Draft International Standard (FDIS).

Johnsson, C. (1999). *A Graphical Language for Batch Control.* Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology.

Machado, J., Louni, F., Faure, J., Lesage, J., Ferreira Da Silva, J., and Roussel, J. (2001). Modelling and implementing the control of automated production systems using Statecharts and PLC programming languages. In *Proc. of the ECC 2001*, 1019–1024. Porto, Portugal.

Maffezzoni, C., Ferrarini, L., and Carpanzano, E. (1999). Object–oriented models for advanced automation engineering. *Control Engineering Practice*, 7(8), 957–968.

Object Management Group (2007). UML, v. 2.1.2, OMG. Document N. formal/2007-11-04. Http://www.omg.org/spec/UML/2.1.2/.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object–Oriented Modeling and Design.* Prentice–Hall.

Selic, B., Gullekson, G., and Ward, P. (1994). *Real-Time Object-Oriented Modeling.* John Wiley & Sons.

Selic, B. and Rumbaugh, J. (1998). Using UML for complex real-time systems. ObjecTime Limited, www.rational.com/media/whitepapers/umlrt.pdf.

Shah, S., Endsley, E., Lucas, M., and Tilbury, D. (2002). Reconfigurable logic control using modular FSMs: design, verification, implementation, and integrated error handling. In *Proc. of ACC 2002.* Anchorage, USA.

Storr, A., Lewek, J., and Lutz, R. (1997). Modeling and reuse of object-oriented machine software. In *Proc. of European Conference in Integration in Manufacturing*, 475–484.

K. Thramboulidis. (2008) Challenges in the Development of Mechatronic Systems: The Mechatronic Component. In *Porc. of 13th IEEE Int. Conf. on Emerging Technologies and Factory Automation.*