Mechatronic Objects Modeling and Realization with IEC61131-3 Software Structures

Marcello Bonfe and Cesare Fantuzzi

University of Ferrara, Dept. of Engineering
V. Saragat, 1, 44100 Ferrara - Italy.
E-mail: {mbonfe,cfantuzzi}@ing.unife.it

Abstract

This paper concerns with the definition and use of “mechatronic objects” for the design of complex manufacturing system as large machinery, which involves some concepts derived from Object–Oriented programming applied to industrial software development.

In particular, the programming languages and structures defined in the International Standard for Programmable Logic Controllers (PLC) IEC 61131-3 norm are considered, with regards to software encapsulation facilities they offer. The aim is to develop a software configuration in which both control and mechanical devices are considered as a single object module. We call this integrated electronic–mechanical structure a “mechatronic object”, meaning that this module should be designed, maintained and reused as a whole.

The paper describes a modular objects identification methodology and reports about a real application example of the proposed concepts, showing effectiveness of the object–oriented framework for industrial software applications.

Keywords: Programmable Controllers, Object–Oriented modeling, IEC 61131-3, software engineering, real–time control.

1 Introduction.

Large manufacturing machines designed and developed for goods production are complex systems with several mechanical and electronic parts. The control software they need is often designed and developed separately from the physical project, even if they are structurally close connected. In fact, the machine project is usually split in two tasks carried on separately by mechanical and electronic engineering teams, the last being in charge of machinery control design. These teams usually don’t work in parallel, but the electrical project is queued to the mechanical design and realization. This basically leads to inefficiency in machine time scheduling and poor integration between mechanical and electronic parts, in particular with respect to hardware and software for control.

On the other hand, the software development tools for programmable controllers are very primitive, especially when compared to those designed for high level languages such as C/C++ or Java. As matter of fact, industry standards PLC (Programmable Logic Controllers) relies on low level programming languages such as Ladder Diagram and Instruction List [Lewis, 1998] for control program implementation, with evident limits with respect to software structuring and modularization, not mentioning lack of parameterizable procedures.
This scenario leads to inefficiencies in the machinery development cycle, and often few components (e.g. mechanical, electrical and software parts) can be reused in subsequent projects “as it is”, increasing overall costs.

Principles of Object–Oriented philosophy has been applied by some authors to the industrial field, e.g. [Duran and Batocchio, 1994], [Bonfatti et al., 1995], [Maffezzoni et al., 1999], in order to increase software quality, trying to take advantage of object–oriented basic concepts (abstraction, encapsulation, inheritance), which showed effectiveness in increasing software modularity and reusability in many Personal Computer applications.

A leading common theme between these works is the use of object–oriented methodology in system modeling and designing phase, that leads to a very powerful description of the physical devices composing the overall system. Unfortunately, the description obtained is hard to translate into code, when developing control software in a PLC–based environment, without introducing limitations or loss of effectiveness. Some of the works cited ([Duran and Batocchio, 1994], [Bonfatti et al., 1995]) describe the development of a specific intermediate tool between the high–level objects model and the low–level programming environment for the controlling device chosen. However, even if the objects model in itself is general and abstract, its implementation requires an adaptation to the PLC language characteristics, which change dramatically among different producers and, sometimes, even from different series within a single producer. In particular, as said above, the basic PLC programming languages don’t support efficiently modularization, so the bounds between the encapsulated objects may definitively blur. Within these limitations it is very hard to take advantage of the object–oriented paradigm and really reuse program portions in different projects.

In order to improve the reusability and the portability of control applications for PLC environment, we should discuss on two main aspects:

- the necessity to treat a single portion of a complex machine as an object, composed by a whole of mechanical parts and electronic (hardware and software) parts that carry out a specified functionality in the productive process.
- the necessity to have ready “off-the-shelf” programming languages and tools, possibly platform independent, that easily allow the application of the object-oriented methodology.

The first topic leads to a modular approach that starts from the very early phase of the project, when the machine is just an idea. In this way the design of a complex system can be developed more efficiently, as the project can be divided into parallel tasks, each one in charge of the complete definition, from mechanical design to control software, of a single and separate module. Of course, this also means that the difficult problem of the entire machine project can be decomposed into easier sub–problems. However, to reach the goal of increasing efficiency in the project development, the modules must be carefully shaped, that means that they have little interactions or these interactions can be precisely defined. Here is the point where the object–oriented 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 in this discussion they are referred as mechatronic objects.

The second topic regards an aspect of machinery control that is outside the reach of developers, but is in charge of producers of programmable controllers and their software development tools. Until recent years, major companies in this field were used to carry out proprietary programming languages and software
structures. This scenario is completely different to that of Personal Computers software environments, for which standardized high level languages are now a must. However, this situation has started to change when the International Electrotechnical Commission (IEC), aiming to cope with PLC users and developers requests for standards in PLC programming, published the IEC 61131-3 International Standard [Iec 61131-3, 1993], that regards the main aspects of industrial controllers programming languages and structures. PLC producers are complying more and more to this norm and it seems that in the future years it will be an establishment.

This paper aims principally to explore how the languages and the structures of the cited norm can be exploited to develop industrial software object–oriented. Right before discussing this main topic, we should explain the definition of “mechatronic objects” in manufacturing systems, that is treated in the next Section. Then Section 3 gives a quick survey on the main concepts defined in part 3 (“Programming Languages”) of the IEC 61131 standard, that can be helpful for implementing a mechatronic object–oriented architecture. Subsequent section presents a practical application of the projecting methodology discussed with regards to a machine for tiles production.

2 Identifying “mechatronic objects”.

When the object–oriented paradigm was introduced, in the first 80’s, it changed dramatically the approach to software development. The main difference with traditional process of program construction was in shifting the focus point of the programmer. In fact, Meyer’s object motto states: “ask not first what the system does, but what it does it to” [Meyer, 1997]. In business and information systems software, this statement refers typically to data, in any possible shape (e.g. texts, numbers or images).

In practice, following this approach data structures cannot be considered separately from the operations that can be performed on them. In object-oriented languages this is possible thanks to peculiar software structures called classes, that encapsulate together data, called attributes, and operations, called methods. Normally, attributes should be accessed only by methods of the same class, so it is necessary to appoint some methods as interfaces with the rest of the system.

The term object refers to the runtime structure related to the instance of a class, which is instead the static structure abstracted from a specific program. In fact, object-oriented programs are developed introducing instances of various necessary classes and then performing invocation of objects interface methods, real means of data processing. An object is then somehow similar to a “services centre”, executing specific tasks when requested from the main program or from another object. Figure 1 schematize this viewpoint, describing also the encapsulation principle behind the object concept.

![Figure 1: Encapsulation of functionalities in an object](image)

Object-based programs are intrinsically modular, thinking to objects as modules, and this is helpful in program testing. Moreover, relationships between
modules are strictly under programmer’s control, as they rely only on well-defined objects collaboration, limiting the possibility of dangerous side-effects. Furthermore, thanks to the mechanism of classes definitions, it is easy to reuse large code parts, exploiting generic classes libraries and developing new classes as a composition of existing ones or specifying more general ones. In this case peculiar mechanism of object-oriented programming languages (such as inheritance and polymorphism [Meyer, 1997]) are applied.

2.1 Applying object-oriented modeling to real-time systems

While the popularity of this programming paradigm were growing, also new software engineering methods were developed in order to manage the complexity of large software systems. For example, advanced object-oriented modeling techniques developed by leading software development methodologists are Rumbaugh’s Object Modeling Technique (OMT) [Rumbaugh et al., 1991], Booch’s Method [Booch, 1994] and Jacobson’s Object Oriented Software Engineering (OOSE) [Jacobson, 1992]. Moreover, potentialities of object-oriented modeling for physical complex systems recalled the attention of control engineering specialists ([Joannis and Krieger, 1992], [Abou-Haider et al., 1994], [Andersson, 1994]). However, some aspects peculiar of real-time complex systems were difficult to describe without improving the capabilities of objects model, especially with regards to the operative features of physical devices. Recently, the principal concepts of modeling methods cited above have been unified into the formal description of an object-oriented language, called UML (Unified Modeling Language [Rumbaugh et al., 1999]), very powerful for expressing the constructs and relationships of complex physical architectures. The UML is intended to be a graphical and general modeling language, not a programming language, despite its efficacy for specifying and constructing a software system. However, UML models can be tightly mapped into almost any possible object-oriented programming language.

Thanks to its expressiveness and flexibility, UML has been widely recognized as a one of the most powerful software modeling language, and has also been extended with specific features for real-time systems. In UML models, these systems can be described by means of two kinds of analysis:

- the **object structure** analysis, from which is derived a model that describes all the objects, with their attributes, operations (or methods), state, identity and responsibilities, and relationships among them (association, aggregation, composition, generalization and dependency).

- the **object behaviour** analysis, that regards both internal behaviour, which in some cases can reveal a discrete events dynamic described in UML by means of Harel’s State-charts [Harel, 1987], and objects collaborations, that can be described by sequence diagrams (time traces of operation requests and answers) and collaboration diagrams (similar concept, without timing considerations).

With particular regards to UML applications for real-time systems, in [Douglass, 1999] is given a guideline of the key concepts for the definition of an efficient object structure:

- identification of sources of actions, events and messages and their targets;
- identification of physical devices, electronic or not;
- identification of transactions (e.g. bus messages and queued data)
Most of the times, all of these items should result in an object of the model.

A model obtained applying UML concepts to a control problem will be therefore a set of heterogeneous objects. For example, in Figure 2 is shown a possible model of an unspecified control application. Some objects are related to different kind of sensors, which are specification of a generalized “sensor” class that have an attribute related to the value sensed and methods related to acquisition or reset. Relationships between sensors are modeled in UML by the generalization arrow, while relationship between an A/D converter and a sensor is modeled with aggregation arrow (for details see [Rumbaugh et al., 1999]). Electric motors, valves, even simple buttons can also be described as objects. All of these physical devices are then associated to another object, more likely a “software” object, named “controller” (e.g. a PID regulator, or another kind of algorithm implemented on a digital device).

![Figure 2: Simple model of a control application.](image)

The objects model is then strictly related to the system physical architecture. Moreover, the analysis of system dynamics, object by object, permits to highlight state oriented behaviour of specific components, while with an overall analysis it is very difficult to isolate different working states. As said above, this kind of behaviour is described with Harel’s State–chart formalism [Harel, 1987], of which an example is given in Figure 3.

![Figure 3: Example of Harel’s Statechart](image)

### 2.2 Mechatronic objects in PLC software

The realization of the control software for a system modeled with UML, when considering a PLC–based platform, requires a higher level of abstraction. In fact, PLCs are devices with characteristics designed to simplify the realization of discrete time control, with particular regards to sensors and actuators interfacing.
The PLC operating system take charge of synchronous input signals acquisition and output signals actuation, so that these operations need not to be addressed by the programmer, who can concentrate only on the control algorithm.

The analysis and design approach for a machine controlled by a PLC should then be slightly different. First of all, the system should be split into physical-functional modules, even distributed onto several hierarchical levels, that can be considered as stand-alone components performing a specific functionality in the productive process. For example, a packaging machine will probably have a transport cell, which receives products from the preceding machine. This transport cell can be composed by one or more different conveyors, each one moved by a different motor. All of these items are candidate to be modeled as objects.

Then, great attention should be given to interactions between the electro-mechanical view and the control software view of the project, so that it can be achieved a software modularization that reflects the physical objects. Furthermore, from the control point of view, it is easy to discover which objects are autonomous and which are instead collaborating (e.g. with interlocks, synchronization, etc.), that means the related software modules must exchange informations. Back to the packaging machine example, from the controller side, a conveyor is identified by the connections with the driver of its motor. The control algorithm acts on these signals, performing different operations (enable driver, activate, change speed, etc.), typically with a well defined state sequence. The ensemble of the conveyor, the motor, the driver and the control logic can then represent an “object”, composed by mechanic, electronic and software elements: a mechatronic object.

The control algorithm can be considered as the main “method”, in an object-oriented terminology, of the object, meaning that it should be encapsulated in a software structure together with internal data, to store the object state, and an interface. This interface is the means to connect the control algorithm with its physical object, via PLC memory mapping of sensors and actuators, and also with other objects of the control program, for interlocks and synchronization. Recalling the parallelism with traditional software objects, the latter play the role of “interface methods”. It is evident that if a similar conveyor is used in another machine, or in different parts of the same machine, its control software module can be reused.

3 Object-oriented techniques and IEC61131-3 norm languages

The implementation problems in developing reusable control software for the mechatronic objects described above, regard mainly the modularization and encapsulation facilities that can be used in the programming tools.

The lack of program structuring is a typical weak point of the PLC world, mainly because of historical reasons. In fact, PLCs were introduced in late ’60s to substitute electro-mechanical control panels with smaller and flexible devices and the most common PLC program structure has been based, up to now, on a unique block written in Ladder Logic, a graphical language that resemble a relays and coils rack. For a long period of time, it was also the only method permitted by development tools. However, as people with different backgrounds approached PLC programming, they perceived its limitations, so new languages have been studied and formalized (e.g. the state-transition chart Grafcet [David and Alla, 1992]) and facilities to split large programs in smaller blocks and subroutines have been introduced. Unfortunately, major Programmable Controller producers has so far developed proprietary languages, pretty different from each other, with programming tools not effectively oriented to software reuse (e.g. symbolic and
parametric programming is poorly supported). In this confused situation, the request for standardization has grown up from PLC users until it has been recognized by the international industrial community. That is the main reason that brought the International Electrotechnical Commission (IEC) to publish the IEC 1131 document, a standard for Programmable Controller, whose part 3, devoted to Programming Languages, became and International Standard in 1993 with identifier IEC 61131-3. 1979, the IEC set up a

The main aspects covered by IEC 61131-3 are:

- the definition of four different programming languages and a graphic formalism for structuring the internal organization of software blocks;
- the definition of the software model of a control application.

The programming languages described in IEC 61131-3 can be divided in two families: textual and graphical. Textual languages are: Instruction List (IL), a low–level (assembler–like) language, suitable for generating optimized code; Structured Text (ST), a high–level language, with some resemblance to PAS-CAL, suitable for complex computations on numerical data and structured programming. Graphical languages are instead: Ladder Diagram (LD), based on most known PLC programming formalism, very much logic–oriented; Function Block Diagram (FBD), a powerful formalism for describing signal and data connections between Function Blocks, software elements described better later on.

Another graphical syntax is described in the definition of Sequential Function Chart (SFC), a state–transition diagram derived from Grafcet and Petri Nets [David and Alla, 1992], very expressive for describing, in a readable and intuitive way, sequential behavior of complex systems. Even if many people consider SFC as an effective programming language and several programming tools are equipped with a SFC editor, it cannot be considered as a full language, because some of its parts (actions and transitions) require an internal implementation with instructions from other languages.

Every automation problem can be resolved with one of these development. However, in this discussion, the most important concept proposed in the IEC standard regards the structure of a control application. The IEC 61131-3 software model allows the decomposition of the application into components that are defined Program Organization Units (POUs). POUs can be of one of the following three types:

- **Program**: it is the larger unit that contains all the programming elements (global variables, other POUs declaration, etc.) and constructs necessary for the control of a machine or a process.

- **Function Block (FB)**: it is a software element that have a generic definition based on two parts:
  - a data set, which describes the input/output parameters that permit connections to the other POUs (its external interface) and static or temporary internal variables; their value
  - an algorithm, which is the code executed when the function block is called, processing the data set described above. The algorithm can be expressed in any of the IEC 61131-3 languages.

- **Function**: it is a software element that differs from Function Block in the lack of static internal variables (so that it can’t maintain an internal state)
and in having a single output parameter. The choice of the programming language to implement the Function algorithm must exclude SFC, of course, as the diagram state cannot be stored.

The generic and parametric definition of POUs, in particular FBs, is the major point indicated in the IEC 61131-3 norm for software reuse. In fact, Function Block definition is simply a type declaration, like a class is the definition of an object type. A program can contain several instances of the same FB, each one independent from the others as their internal data are allocated in a reserved memory area. Furthermore, the same FB can be used in several different programs, reducing time spent on rewriting code. This means that a library of user-defined reliable software blocks is available when programs end their “on-the-field” test phase. Another notable aspect of the IEC 61131-3 definition of POUs, related to hierarchical decomposition, is that FB can contain internal instances of other FBs, whose scope limit is the container FB. In this way program can be constructed with higher level blocks that encapsulate and control the execution of other modules. As is better described in next section, this feature of IEC 61131-3 FBs plays a key role in the mechatronic objects encapsulation.

3.1 Mechatronic objects within Function Blocks

It is evident that the software component appointed to implement a mechatronic objects model as it is described in Section 2.2 is identifiable in the Function Block (FB). This one is in fact the structure that can encapsulate the control part related to a physical object, with interfacing features and internal data protection. As shown in figure 4, the FB Input/Output interface is not only the connection means to physical sensors and actuators of the mechatronic object (via PLC I/O), but also a means to exchange data with the rest of the program, as required for efficient machine modules synchronization. In this case, these signals can be viewed as other mechatronic objects messages or requests. It is interesting to note that, even if they have different semantics, physically related parameters or logical objects interactions are not formally separated in the FB definition of the IEC 61131-3, while in the IEC 61499 Document [Iec 1499, 1998], FBs are required to have distinct Data Interface and Event Interface. However, the latter is more related to execution control and scheduling of FBs, because the IEC 61499 addresses Distributed Control Systems, in which it is very important that software modules that resides on different devices follow a well predictable execution sequence. In this discussion, instead, the scenario is that of a hierarchical and centralized execution model, in which higher level objects manage the lower level objects and control their execution. For example, the PLC program controls directly the enabling of FB global instances and each FB can act similarly on internal FBs, managing directly their input/output parameters.

mechatronic and they represent communications to other objects.

The FB internal data section, instead, requires a different interpretation, as it is the equivalent of class attributes in the object-oriented paradigm. The local variables can be static, whose value is maintained between subsequent execution, or temporary, whose value is instead reset. Static variables represents practically the “state” of the object, meaning with this term not only the actual condition of a State–chart, for example, but also the condition of an object with continuous behaviour. For example, a PID control loop implementation in a Function Blocks, requires internal real variables to store the Integral term and preceding samples needed to calculate discrete Derivative term.

In traditional PLC programming methods, machine states are typically stored in public memory area. In this way, events and conditions can be easily detected by means of read accesses to this public memory, even if the program is block
structured. On the other hand, also dangerous write memory accesses are allowed. In an efficient IEC 61131-3 compliant programming method, instead, only block inputs and outputs should be used to exchange informations between different parts of the program, and static internal variables, protected from external accesses, to keep consistent objects state.

3.2 Mechatronic objects identification methodology

Before starting with the implementation of a machine control program on a IEC 61131-3 compliant PLC platform, following a object–oriented analysis like that proposed in this paper, an important issue to discuss is how much the IEC 61131-3 software model is really object–oriented. In fact, as is discussed in [?], a language can be considered object–oriented only if objects have an associated type (its class) and if types may inherit features from a supertype. In the IEC 61131-3 FB are actually defined as types, but it is not possible to define a FB type as a specification of a FB supertype. Without inheritance facilities it is also not possible to define polymorphic operators. However, according to the terminology introduced in [?], the IEC 61131-3 features described in the previous section permit to say that the Standard is at least “Object–Based”, even though not “Object–Oriented”. With regards to inheritance and polymorphism features, necessary to talk about object–orientation, it should be noted that in real-time systems, where predictability of control flow and timing is a must, having polymorphic operators can be dangerous, if they are not accurately defined and used by the programmer. Moreover, in object–oriented programming, when a class is defined from a superclass, typically their most peculiar operations are rewritten. In the mechatronic objects paradigm described in Section 2.2, the control algorithm is considered as the main operation of the object, so if there are similar objects with different control characteristics, their software parts are probably different. This means that, even if supported, inheritance in the definition of FBs would probably have limited benefits.

The hierarchical architecture of the IEC 61131-3 software model suggests therefore an approach that is not based on generalized superclasses and their specification, but on the aggregation of small objects into “container” objects acting as supervisors. This is possible my means of FBs instances encapsulated inside “higher–level” FBs, which are instead declared as global instances at Program level. It is obvious that, with this framework, a top–down analysis is probably more effective to identify the preliminary project structure. In practice,
the system should be decomposed step by step until elemental devices are recognized. For a manufacturing machine an effective decomposition methodology should determine the following levels, that can be named:

- **Cell**: this level corresponds to a machine module in which a substantial part of the global process is executed. For example, it could be a welding cell or a belt system between two neighboring work-cells.

- **Unit**: it is a subset of devices in a cell which are related to a particular function. For example, considering a filling system for a can machine, the set of devices (actuators and sensors) used to let the product flowing into the cans.

- **Device**: the bottom level corresponding to “atomic” devices, provided directly “off-the-shelf” by manufacturers, even if composed themselves by mechatronic components (e.g. pneumatic actuators with embedded end-stroke sensors).

Once identified mechatronic components, their relationships should be carefully determined. In fact, the external interface of the FBs related to control mechatronic objects is fixed by the input and output signals coming from machine parts, plus some interlocks conditions related with other mechatronic objects. The FB interface definition is very important, because it is the only means to make objects communicate. After the Analysis and Program structure Design phases, the FBs internal algorithm can be programmed. In this phase, it can be of help a user-defined library of FBs related to standard machine components defined in previous projects. Actually, common objects are typically of the lowest hierarchical levels, as the “Cell” level objects are likely to be adapted for specific project features. This means that the project development process for control application results in a combination of a top-down and bottom-up approach, the first one coming in hand principally to manage the complexity of the overall view of the project, while the second is better suited to deal with the low-level implementing phase, making easier maintainability and reuse of modules. Figure 5 shows the cyclic projecting steps of the methodology, highlighting which are their intermediate results.

Figure 5: The projecting methodology oriented to mechatronic objects.

### 4 A practical example

The projecting methodology discussed in previous section can be applied in various field of manufacturing automation. Of course, its effectiveness is related to the likelihood that many components recur on different machine projects developed by the same firm’s R&D department. Most of the times, manufacturers of manufacturing machines are specialized in a particular type of production, so the situation described above is pretty common and it is exactly the best condition to take advantage of previously implemented FBs related to those components.

In this section we report on a real application of the mechatronic decomposition to a machine for concrete tiles production. The control program has been implemented using Siemens Step7 programming tools for a Siemens S7 PLC [Sim, 1999]. This tools have been chosen because of their compliance with IEC
61131-3, even if some implementation details are peculiar. This means that at present time the software blocks library that result cannot be ported easily on different platforms, which is instead one of the IEC 61131-3 norm goals. However, the Analysis and Design phases of the methodology are implementation independent, as they rely on International Standards, so that even if a project needs to be ported on a different control platform, if it is IEC 61131-3 compliant only the Programming phase needs little adjustment, especially with regards to hardware specific instructions set, while Program structure and FBs interfaces remains the same.

The machine considered is in charge of tiles drying process. It receives fresh products from another machine, stacks them and pushes the stack in the drying store. This one consists of a rotating wheel divided into several sectors, each one is able to store a fixed amount of tiles stacks. The wheel turns around sector by sector, when the desired number of stacks is loaded, and when a specific sector has done a full round the tiles can be unloaded and unstacked from a machine module similar to the loading one.

In Figure 6 a UML diagram of the physical modules composing the machine is showed. Notice that it is exactly the FBs structure of the control program developed, as will be described later.

Figure 6: UML diagram of the machine considered

The nested architecture described in section 3.2 is modeled with the aggregation/composition relationship between objects, graphically shown as a line with a rhomboidal end. According to the identification methodology, cell–level objects contain unit–level ones and these contain device–level ones. For example, “Loader” cell object contains two “Conveyor” objects, an “Elevating Grid” object and a “Pushing Bar” object.

Another relationship between objects is generalization, shown as a line with closed arrowhead, that means that an object is a more specific kind of another object type. For example, the “Loader” cell contains a “Collapsing Conveyor” and a “Launching Conveyor”, which are specialized “Conveyor” objects. The two are actually different, even if they both act to transport products. As said above, without inheritance facilities a transparent generalization is not possible, but different FB types must be programmed for the specific objects. Anyway, the internal implementation can be a further composition of lower level generic
FBs, plus some instructions controlling object peculiarities. For example, the collapsing conveyor consists of variable speed conveyor and an hydraulic unit that extends it. The two parts can be controlled separately by two specific FBs.

The PLC Program is then composed by three “cell” FBs plus a simple Function that manages the operator panel, not considered because not strictly related to control aspects. The FBs correspondent with the three machine cells, working in parallel, are:

- Loading Cell;
- Storing and drying Cell;
- Unloading Cell.

The loading and the unloading cell are very similar, even though they are not the same. Each one is composed by one or more conveyors, a brushless motor driven elevating grid, and a pushing bar, for moving tiles stacks inside and outside the rotating store. The latter is instead very simple, because it requires only one or two bolt–bar systems hydraulically actuated to execute a step by step rotation. Each of these units and devices are controlled by a specific FB.

The program has then been developed programming each FB according to the nested encapsulation methodology, so higher level FBs contains simply lower level FBs instance declarations, FB calls and few “supervision” instructions, while lower level FBs contain the effective control code. This one has been written using S7-KOP, S7-Graph and S7-SCL, Siemens equivalents of LD, SFC and ST, choosing the most suitable according to the specific FB characteristics. The use of three different languages proves that the realization of FB internal algorithms, if correctly programmed, doesn’t affect external interactions with other FBs. For the descriptive aims of this paper, it is helpful to describe how blocks of a cell interacts with a Function Block Diagram, even if the language has not been used in the program. For example, the internal FBD of the loading cell control block is schematized in Figure 7. Notice that each FB is able to detect malfunctionings in the mechatronic object it controls and whenever one of the four components sets its “alarm” condition, also the others are stopped.

As said above, by approaching the machine project with different levels of detail, some objects reveal a state–oriented behaviour. In this case the SFC formalism is suitable to program objects control, taking advantage of its high level of expressiveness.

For example, in the machine considered, a well defined operational sequence regards the loader pushing bar. The bar moves along the horizontal axis, to push the tiles stack and return to its start position, and along the vertical axis. The vertical movement permits the immediate restart of the elevating grid as soon as it has been emptied, because lifting the pushing bar prevents interceptions, during its backward movement, with tiles elevated in the meantime. Of course, the end of the horizontal movement depends on the number of stacks loaded before in the same wheel sector, so that some state transitions are conditioned by proximity switches signals and stack counter comparison. A sketch for a SFC to control this part of the machine is described in Figure 8. It is worth to note, that an SFC implementation starting from a UML State-chart description, that may be available from the Analysis phase of the methodology, can be done with little additional efforts, as the two formalisms are very similar.

The program developed accomplishing the methodology proposed in the paper, presents several advantages with respect to a program written with traditional PLC software development methods. First of all, a sketch of the program structure can be drawn right from the first phase of the mechanical design, as the physical structure of the machine is already defined. Furthermore, time needed
Figure 7: Function Block Diagram of the Loader Machine Unit.

Figure 8: Sequential Function Chart of the Loading Pushing Bar.
for program testing, task executed typically with machine on work, can be shortened, because when the programmer detects malfunctionings in a specific part of the machine, he can concentrate just on the FBs that are supposed to need corrections.

Software reuse can be achieved basically through the lowest level FBs, because typically the smaller is the module the higher is its independence from other modules. In higher level FBs, instead, the code needed to make the elementary components work together according to the machine specific requirements is more likely different from other projects. Finally, the program is much more readable, because it is fully structured and fully documented right from the first phase of its development.

5 Conclusion

To conclude, this paper has dealt with the implementation of the “object–oriented” technology for industrial automation environment. In particular, the aspects strictly related to the peculiar characteristics of the control software for industrial manufacturing machines have been discussed, like the necessity of combining together mechanical and electrical devices, leading to “mechatronic” components, and some characteristics of the industry standard IEC 61131-3 norm permitting the implementation of some concepts of the object–oriented technology.

In particular the work’s focus is on the use of the Function Block software structure, that permits module encapsulation and reuse, complying to an object–based methodology.

The paper ends with a real example showing the effectiveness of the concepts discussed.

first citation and labelled by numbers labelled by authors’ names (long).

References


