Control Engineering Practice 21 (2013) 1608–1619

Contents lists available at ScienceDirect

Control Engineering Practice

journal homepage: www.elsevier.com/locate/conengprac

Design patterns for model-based automation software design and implementation

Marcello Bonfè [a,*], Cesare Fantuzzi [b], Cristian Secchi [b]

[a] ENDIF – University of Ferrara, Via Saragat 1, 44122 Ferrara, Italy
[b] DISMI – University of Modena and Reggio Emilia, Via Amendola 2, 42122 Reggio Emilia, Italy

ARTICLE INFO

Article history:
Received 10 May 2011
Accepted 20 March 2012
Available online 20 April 2012

Keywords:
Industry automation
Programmable logic controllers
Object modeling techniques
Object-oriented programming

ABSTRACT

The paper presents the application of object-oriented modeling techniques to control software development for complex manufacturing systems, with particular focus on case studies taken from the packaging industry and design patterns that can be abstracted from such case studies. The proposed methodology for control software modeling and implementation is based on a practical approach refined on the basis of *on-the-field* experience and interactions with control engineers involved in the development projects.

The final objective of the paper is to review and analyze patterns for the solution of design and implementation issues that typically arise in the considered application domain.

1. Introduction

The adaptation and application of Object-Oriented (O-O) concepts to modeling and implementation of industrial control software is a hot topic, in both industry and academia. The number of scientific and technical publications covering at least one aspect among logic control (i.e. software for Programmable Logic Controllers, PLCs) modeling with O-O languages, O-O extensions of PLC programming tools or simply automatic code generation from O-O models to existing PLC languages has been constantly growing in the last 15 years. In a broader perspective, the *mechatronic* approach (Tomizuka, 2002) to production systems design provides further motivations for the application of O-O and *component* based design methodologies to increase synergetic integration of information technology, control and physical systems, promoting modular thinking and artifacts reuse.

Object-orientation is perceived as a key factor to manage software complexity, which is an increasing issue also in the manufacturing industry. In fact, encapsulation features of objects and classes prevents from incorrect use or manipulation of software parts and data, while inheritance enables software reuse and so-called *design by extension* (Bonfè, Fantuzzi, & Secchi, 2006). O-O design increases reuse possibilities also from a mechatronic point of view, as pointed out by several authors (Lüder, Hundt, Foehr, Wagner, & Zaddach, 2010; Thramboulidis, 2008), provided that mechatronic system development is supported by modeling and design specification tools allowing to capture conceptual peculiarities of the application domain.

In the context of O-O modeling of software and systems, UML (Unified Modeling Language) and UML *dialects* (Object Management Group, 2009, 2010) are currently the dominant standards. Many authors propose the use of UML or, at least, of Statecharts (included in UML for state-based behavioral specification) in industrial applications. Most of these proposals, that will be further discussed in Section 2, define a domain-specific adaptation of UML, by means of its native extension mechanism (i.e. *stereotypes* and *profiles*), and then either formalize the semantics of the adapted language for verification purposes (i.e. Model Checking, Bonfè & Fantuzzi, 2003) or develop an automatic model-to-code generator (Vogel-Heuser, Witsch, & Katzke, 2005) for industrial targets (i.e. supporting IEC 61131-3 programming languages, International Electrotechnical Commission, 2003).

However, control engineers addressing real-world applications do not need only modeling languages and design methodologies, but also reference examples and design patterns solving common issues in manufacturing systems control design (e.g. alarm handling, coordinated motion, production reconfiguration and recipe management). Moreover, the final implementation of PLC and motion control software, if a model-based code generator is not available, should be supported by rules and patterns that can be easily applied even by manual coding.

The aim of this paper is to describe design patterns for industrial control software development, that have been collected during years of collaboration with companies developing automated machines and packaging solutions. The O-O paradigm is applied by means of UML modeling, while the implementation phase refers to IEC 61131-3 compatible targets. However, the paper does not address fully automatic translation of UML models to PLC code, but provides practical rules thought to be useful even if manual translation is required, with a focus on the

* Corresponding author. Tel.: +39 0532 974839; fax: +39 0532 974870.
E-mail address: marcello.bonfe@unife.it (M. Bonfè).

0967-0661/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.conengprac.2012.03.017

implementation of hierarchial state machines as specified by UML State Diagrams.

The rest of the paper is organized as follows: Section 2 provides an overview of related literature, while Section 3 recalls the model-based approach to industrial control software design previously proposed by the authors. Then, Section 4 presents relevant case studies and Section 5 shows proposed solutions to common issues in industrial control software design, while Section 6 contains a comparative analysis of different solutions for IEC 61131-3 code implementation. The paper ends with some concluding remarks and proposal for future works.

2. Related work

The industrial control software domain is almost dominated by the programming models defined in the historical IEC 61131-3 standard and its more recent counterpart IEC 61499 (International Electrotechnical Commission, 2005) focused on distributed systems. Both IEC 61131-3 and IEC 61499 support modularity and reusability by means of the *Function Block* (FB) concept, similar to the *object* concept of modern programming languages. More precisely, the two IEC standards can be defined *Object-Based*: FBs are defined as types and used as instances, they encapsulate both private algorithms and data and they communicate with other software modules through well-defined *interfaces*, composed of *input* and *output* signals. However, neither IEC 61131-3 nor IEC 61499, in their current form, is Object-Oriented because they do not support *inheritance*.

This lackness has not prevented many authors from applying O-O design techniques and modeling languages like UML, introducing a proper domain-specific interpretation of abstract design concepts and workarounds to address limitations of the implementation framework. UML extensions, integrating IEC Function Blocks, have been proposed in Storr, Lewek, and Lutz (1997), Heverhagen, Tracht, and Hirschfeld (2003), Thramboulidis (2004), Katzke and Vogel-Heuser (2005), Dubinin, Vyatkin, and Pfeiffer (2005), Younis and Frey (2006), and Ritala and Kuikka (2007). In general, the most notable domain-specific aspect in the context of manufacturing system is the interpretation of *objects*. A manufacturing machine module would be reusable if designed as a tight aggregate of mechanical parts, sensors, actuators and control software routines, specifically related to a given part of the manufacturing process recurring in different projects. This aggregate can be thought as a *mechatronic object*, as introduced in Bonfè and Fantuzzi (2003). A very similar conceptual element has been proposed in Thramboulidis (2008), in which *MTCs* (Mechatronic Components) are described by means of a UML extension. Interfaces for MTCs, called *MTConnectors*, might define material, energy or information exchange channels, while interfaces for *mechatronic objects* as defined in Bonfè and Fantuzzi (2003) are actually more software-oriented. Other proposals for UML extensions, including physical parts in the model, can be found in Burmester, Giese, and Tichi (2005) and Secchi, Bonfè, and Fantuzzi (2007). Using a unified language for control and plant modeling, even if the latter is abstracted at pure logic level, is also a viable approach for closed-loop formal verification, as discussed in Lobov, Lastra, and Tuokko (2005) and Klotz, Fordran, Straube, and Haufe (2009). Many of the concepts elaborated in cited works are embedded in SysML (Object Management Group, 2010), a language harmonizing UML with Systems Engineering methods, whose application to the mechatronic domain is described in Thramboulidis (2010) and Bassi, Secchi, Bonfè, and Fantuzzi (2011). However, SysML is less mature than UML and less supported by off-the-shelf tools, so that UML with domain-specific extensions remains the favorite choice for industrial software design.

Another recurring approach that can be found in the literature is related to architectural specifications for complex manufacturing systems. Most of the papers that focus on practical applications of O-O or model-based design (Cengic, Ljungkrantz, & Akesson, 2006; Ferrarini & Veber, 2005; Hametner, Zoitl, & Semo, 2010; Vyatkin, Karras, & Pfeiffer, 2005) address the complexity problem using a top-down approach and an architectural pattern compatible with the so-called *Hierarchical Control* described in Douglass (2006) and Gomaa (2011). According to this pattern, each component of a control system operates on a given part of the system, by conceptually executing a state machine. In addition, a high-level component provides overall system (or sub-system, if multiple hierarchical levels are defined) control, by coordinating its subordinate components. A rigorous interpretation of this design pattern may increase reusability of low-level components by avoiding direct interactions between *parts* in a *whole*, delegating to the *coordinator* or *supervisor* of each hierarchical level the execution of behavioral specifications for the *whole*. In fact, as discussed in Pazzi (2000) that proposes a modeling framework called part–whole Statecharts for explicit composition of state machines, self-containment and encapsulation are enforced by prohibiting communication and mutual knowledge among different parts, thus allowing their reuse in different contexts. The fact that the Hierarchical Control pattern is applied even to distributed systems, as supported by IEC 61499, demonstrates that it is the most reasonable design approach for complex manufacturing systems. Section 5 will present the application of this pattern to real-world case studies (described in Section 4).

To conclude this literature review, it is interesting to remark that the comparison or, in a certain sense, the dispute between IEC 61131-3 and IEC 61499 has been addressed by many authors (Zoitl, Strasser, Sünder, & Baier, 2009). On one hand, as remarked in Thramboulidis (2009) and Zoitl and Vyatkin (2009), IEC 61499 has not yet changed dramatically the industrial practice, because it is supported by too few mature reference implementations and Integrated Development Environments (IDE). On the other hand, academic research has been quite active on IEC 61499 development, as demonstrated by the number of IDE prototypes or runtime environments mentioned by Thramboulidis (2009) and IEC 61131-3, instead, has weaknesses deriving from the fact that the standard is based on software engineering concepts that are not *state-of-the-art* (Zoitl & Vyatkin, 2009). Moreover, despite the efforts of the PLCOpen organization and its work on additional standards promoting portability of IEC 61131-3 code (PLCOpen Technical Committee 6, 2009) and motion control applications (PLCOpen Technical Committee 2, 2010), straightforward migration of industrial control software from a PLC or Motion Control vendor to another one is simply not possible today. Even though this fact does not depend on IEC 61131-3 itself and is not likely to change in the near future, there is a promising trend towards the revision of the standard in its third release, in which substantial improvements are expected. In particular, O-O extensions of IEC 61131-3 have been discussed in the last years (Gonzalez, Diaz, Fernandez, Junquera, & Bayon, 2010; Schünemann, 2007; Werner, 2009) and there is even a tool already on the market supporting such features (3S Smart Software Solutions, 2010), which will certainly facilitate the use of UML for industrial applications (Witsch & Vogel-Heuser, 2009).

3. Model-based design of industrial control software

The concept of *mechatronic object* introduced in Bonfè and Fantuzzi (2003) guides control software design towards appropriate modular thinking of manufacturing systems. Putting the

focus on software aspects highlights that O-O industrial control modules, as generally suggested in the field of embedded systems design (Douglass, 1999; Selic, Gullekson, & Ward, 1994), should have:

- an interface of synchronization and configuration input/output signals and data;
- an interface of sensory information and actuator commands, connected to physical components;
- private data (i.e. the internal status);
- a control algorithm.

The features described above are fulfilled by IEC 61131-3 and IEC 61499 FBs. Even if the two documents propose a different execution semantics, the generic interpretation of FBs as *mechatronic object controllers* can be schematized as shown in Fig. 1.

A subset of the UML composed of Class Diagrams and State Diagrams would allow to completely specify respectively *structure* and *behavior* of an industrial control system. In particular, Class Diagrams show structural relationships among classes with graphical links called (*simple*) *association*, *aggregation/composition* (part–whole relationship) and *generalization* (a class *derives* from another), which involves inheritance. An important property that can be associated to a class is its *stereotype*, which defines its conceptual role in a domain-specific model. In industrial control applications, structural aspects should be specified from a *mechatronic perspective*. A UML stereotype allowing to describe classifiers for mechatronic objects could be labeled as 《mechatronic》, for classes having an interface of publicly visible properties in their turn stereotyped as 《input》 or 《output》, as required by IEC FBs. A 《mechatronic》 class should not have public operations, while private operations may be used to model complex data-processing activities.

The part of a 《mechatronic》 class related to the connection with physical components is specified with the help of classes stereotyped as 《hardware》. From a mechatronic perspective, physical elements associated to a given machine module are actually *part* of the mechatronic class. Therefore, 《hardware》 classes should be related by means of a *composition* link with a 《mechatronic》 class. Focusing on control software design, physical parts reduce to sensors/actuators I/O signals, which means that 《hardware》 classes and their 《input》 and 《output》 properties represent the hardware I/O ports as a private part of the 《mechatronic》 class. Fig. 2 shows the graphical representation of the proposed stereotypes in a UML Class Diagram. Notice that inputs of a 《hardware》 class are outputs of the controller and vice versa. 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 encapsulation 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).

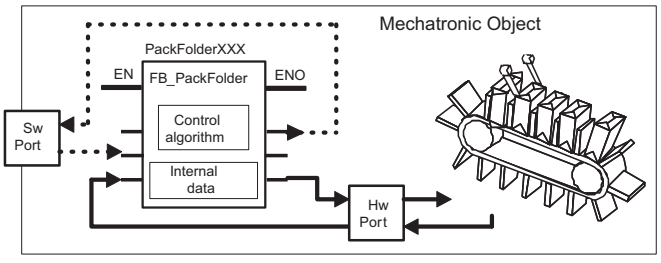

Fig. 1. Schematic representation of the mechatronic object concept.

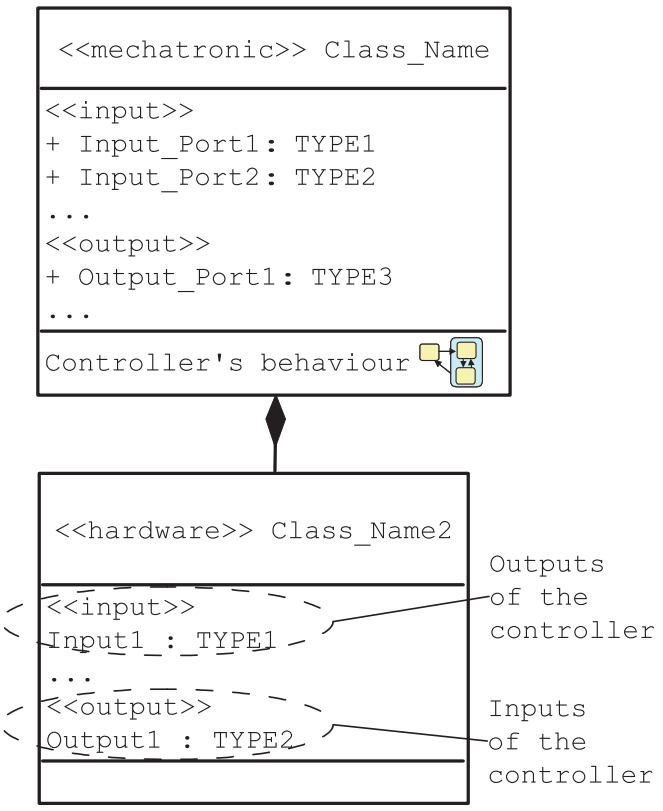

Fig. 2. UML stereotypes for mechatronic models.

The behavioral specification of the system is defined describing the internal behavior of each class with a UML *State Diagram*. This kind of model is strictly derived from Harel's (1987) Statecharts, well-known to embedded system designers thanks to its appealing features: state hierarchy, concurrency and inter-level transitions (i.e. transitions crossing the boundaries of states hierarchy). Even if not formally described, the UML State Diagram semantics is rather different from the one originally proposed by Harel and Naamad (1996) and Chan et al. (1998), the so-called *Statemate* semantics. In particular, the UML semantics requires events to be queued, so that they are processed one at a time to trigger transitions and update accordingly the state configuration, while the Statemate semantics allows simultaneous events processing. Since in both languages multiple conflicting transitions can be enabled in the same execution step, determinism must be ensured by a priority scheme based on the state hierarchy. In UML, higher priority is given to transitions sourcing from states in the lower levels of the hierarchy, while in the Statemate semantics the priority scheme is reversed. The rationale behind the different priority schemes is not well-documented (Crane & Dingel, 2007), but the UML one looks more inspired by the O-O approach: substates override superstates just like subclass operations override those of their superclass. On the other hand, exception handling by means of state-based logic (i.e. interrupt the behavior of substates by exiting their superstate) is more intuitive adopting the Statemate semantics, which is in fact very similar to the one implemented by Stateflow® (Mathworks Inc., 2010b), a tool well-known by embedded systems engineers. Some authors suggest that, if UML is used, exceptions should be treated as higher priority events by a specific event-queue management algorithms (Pintánd Majziker, 2005). However, simultaneous events processing, rather than events queuing, is inherently

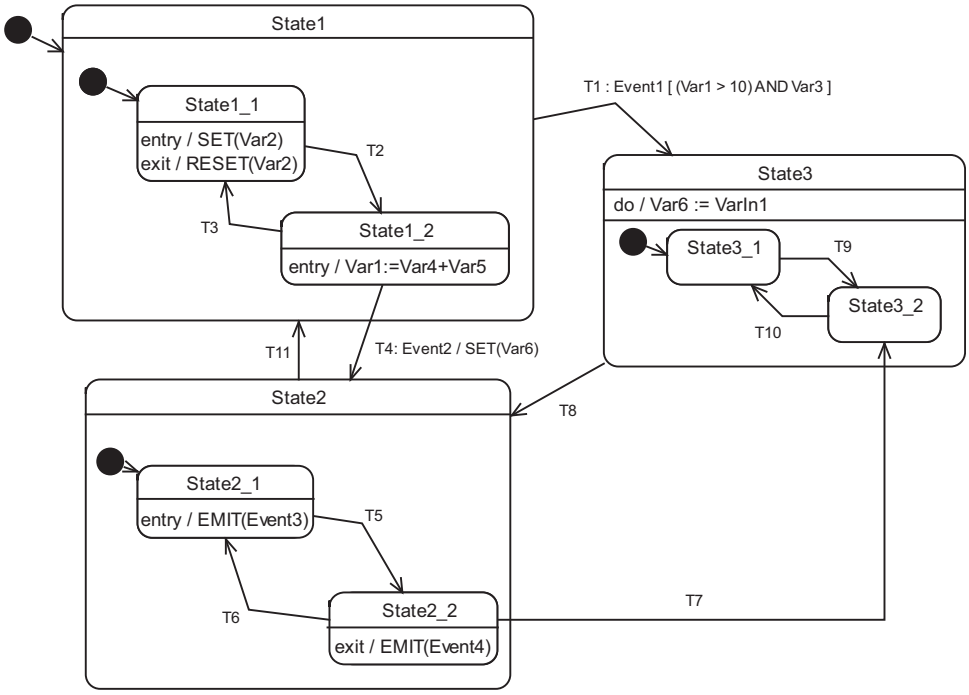

Fig. 3. Example of a generic UML State Diagram for industrial control design.

compatible with the classical scan-based execution model of PLC programs. For these reasons, a Statemate-like semantics, giving higher priority to transitions exiting superstates and requiring user-defined priorities for transitions exiting the same state, is adopted in the proposed UML extension (see Bonfè, Fantuzzi, & Secchi, 2005 for formal details). Similar approaches, including user-defined transition priorities, are adopted by other authors (Klotz et al., 2009; Witsch et al., 2010) among others. The latter reference aims even to formalize a brand new language called PLC-Statecharts. However, PLC-Statecharts enforce user-defined priorities for *any* transition in the model, independently from the hierarchy, which may burden the design process and does not exploit the graphical expressiveness of Statecharts.

From the syntactical point of view, few modifications related to textual expressions (i.e. transition labels and state actions) are proposed. In particular, their specification with an IEC 61131-3 compatible syntax would ease code generation for industrial controllers. Therefore, we require to label transitions with strings having the format:

```
trigger[guard]/actions
```

where events in the `trigger` can be inputs of the stereotyped class or outputs of its contained instances, explicitly typed as `EVENT`. The `guard` must also be a valid Boolean expression, and `actions` will follow the same rules of the similar string included in a state action, which is specified by a textual expression like:

```
when/actions IF[guard]
```

Here, `when` is a qualifier that can be `entry`, `exit` or `do`, associating the actions execution respectively to the state activation, deactivation or persistency, `guard` is an optional Boolean expression that may prevent the action from being executed, if it evaluates to false, and `actions` is an ordered list of operations that can: *set* or *reset* a Boolean variable, *assign* the value of an expression to a variable of either Boolean or non-Boolean data-type, or *emit* an attribute typed as `EVENT`. A generic example of a UML State Diagram with the proposed domain-specific textual expressions is shown in Fig. 3.

4. Packaging industry case studies

Model-based design methodologies, adopting UML as the main specification language, have been applied in recent years to control software development for Tetra Brik Aseptic® packaging machines, developed by Tetra Pak Packaging Solutions S.p.A., and PET/glass bottles filling and labeling machines developed by Sidel S.p.A., which are companies specialized in the food and beverage industry.

Tetra Pak packaging machines adopt the so-called *vertical filling&forming* process, as schematized in Fig. 4. 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 from above a tube of packaging material, obtained by folding and transversally sealing the material, then cutting and longitudinally sealing the package. 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 vapors, and then external and internal cleaning are performed.

Another interesting manufacturing system is the flexible and reconfigurable Sidel labeling machine, whose structure is schematized in Fig. 5. The machine is composed by a main carousel that allows, with its continuous rotation, the processing of bottles by means of several (possibly heterogenous) labeling stations mounted at the side of the carousel.

Thanks to modular design, the machine can be reconfigured on-site with some simple manual operations, in order to modify the setup of the labeling stations and, therefore, change the labeling pattern or the labeling technology (cold glue, self-adhesive, etc.).

In both kinds of manufacturing systems, coordinated motion control of multiple electrical drives is mandatory, in order to allow perfect synchronization of the processing stations manipulating the packaging material or the bottles. In general, motion profiles of mechanical parts in such stations are defined in terms

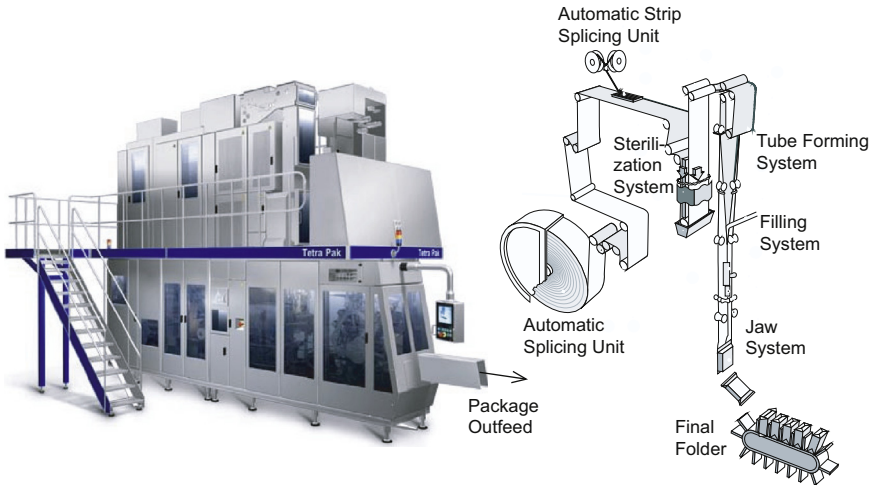

Fig. 4. Tetra Brik Aseptic® machine and detail of packaging material path.

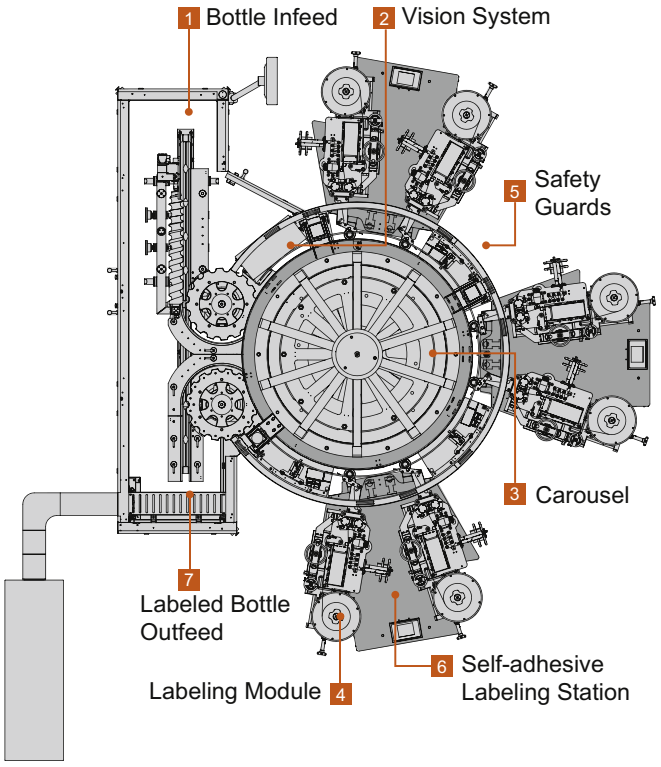

Fig. 5. Sidel SL 90 bottle labeling machine.

of master–slave relationships with the position of a main machine axis, which can be a real motor (e.g. the drive moving the carousel in the labeling machine) or a virtual reference. In addition to the master/slave synchronization, which is performed in real-time by the servo control drives, these manufacturing systems require a minimal interlocking logic, but complex alarm handling logic. Strategies to cope with this complexity, applying appropriate design patterns, are described in next section.

5. Design patterns for complex manufacturing systems

Intensive use of design patterns is a development approach suggested by many papers and books on software engineering (Douglass, 2006; Sanz & Zalewski, 2003), especially when safety-critical or high-reliability requirements are specified. Design patterns are also quite useful in the mechatronic domain, as pointed out also in Burmester et al. (2005) and Maurmaier and Göhner (2009). On the other hand, it is necessary to remark that pure software engineering patterns must be adapted to the domain and, in particular, their usability must be validated on-the-field. This section presents the application of three design patterns, strictly related to each other, to the previously described case studies.

5.1. Hierarchical Control pattern

As discussed in Section 2, the Hierarchical Control pattern is acknowledged as an approach that simplifies reconfigurability of a system, since a part can be changed with another one having the same interface with its supervisor, and that supports incremental testing and validation (see also Fantuzzi, Bonfè, & Secchi, 2009). More precisely, the supervisor executes the behavioral specification for a whole by applying *commands* to its subordinates and receiving back information on their *status*. In addition, a reconfigurable manufacturing system requires each component to set up according to a production *recipe* and to promptly detect *alarms*, that the supervisor will present to human operators. Lowest-level components will directly interact with the physical plant by means of hardware I/Os. Therefore, the application of the Hierarchical Control pattern to manufacturing systems control can be schematized as shown by the UML Collaboration Diagram of Fig. 6.

In the specific case of a Tetra Pak filling machine, the control software structure can be modeled as shown in Fig. 7. Diamond-head arrows show the *aggregation* relationship between the machine supervisor and each module, while simple arrows show *generalization* of some modules, which can have several variants inheriting and, possibly, refining the base behavioral specification (Bonfè et al., 2005). Therefore, the model represents a family of machines, each one differing from the others in the *instantiation* of modules at configuration time.

As can be seen, the UML model defines a three-level hierarchy, whose lowest level is shown explicitly only for the module called *AutomaticStripSplicingUnit* (ASSU). In general, the role of a splicing unit is to guarantee the continuous flow of a laminar material stored in reels. The module will change automatically the reel that feeds the machine when it is almost empty, cutting and joining the material from another full reel. During changeover, the

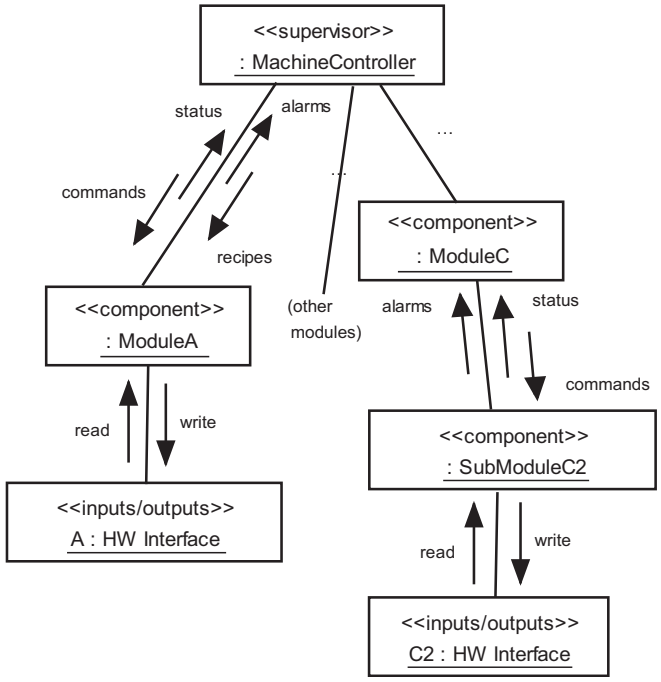

Fig. 6. Example of mechatronic objects collaboration in a Hierarchical Control pattern.

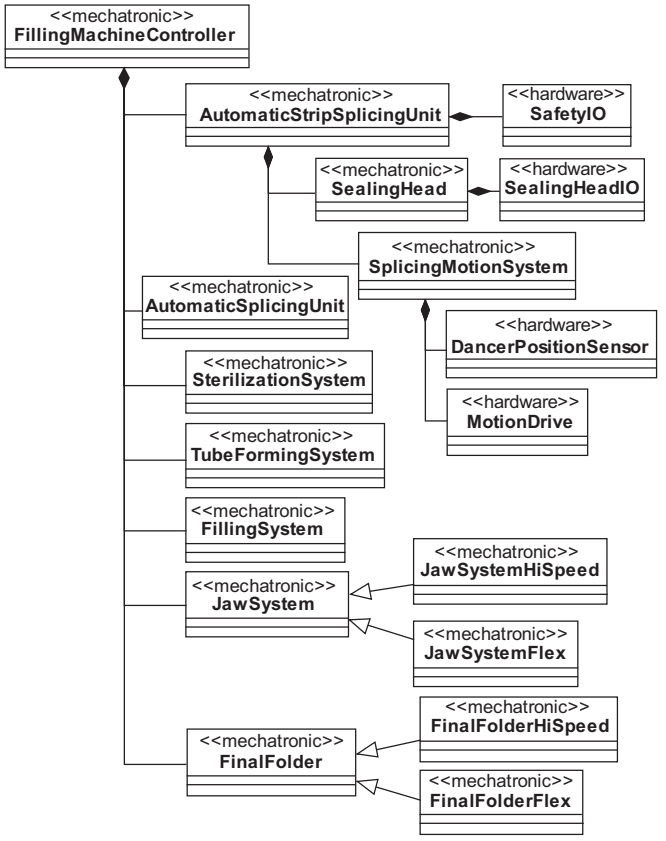

Fig. 7. UML Class Diagram of a Tetra Brik Aseptic® filling machine.

machine is fed by the material buffered in a so-called *dancer* mechanism. In a Tetra Brik Aseptic® filling machine there is always a splicing unit for the packaging material (*AutomaticSplicingUnit*, ASU) and another one for the plastic strip that is melt for transversal sealing. An automatic splicing module can be found also in Sidel bottle labeling machines, as part of some types of labeling stations. Labeling stations are also the mechatronic objects that, applying the Hierarchical Control pattern, would require the specification of a base generic class and several derived classes, according to the specific labeling technology.

The supervisor of such complex systems should take care of synchronization of the modules, which is mostly depending on master/slave coordinated motion rather than interlocking logic, and manage the interface with human operators or packaging line supervision systems. In particular, these interactions drive the machine along a sequence of operating conditions, including not only the active production phase, but also configuration and other procedures (i.e. sterilization of the package filling chamber in which food is processed) preceding and following production. A simplified version of the behavioral specification for the filling machine supervisor is shown in Fig. 8.

The proposed supervisor State Diagram presents many similarities with the state model described by the PackML standard, first proposed in OMAC Users Group (2006), which is going to be included in the ISA-88 (International Society of Automation, 2008) as an extension to the packaging domain of its well-known batch process control model. The PackML objective is to bring operational consistency to all machines making up a complete packaging line, especially if they are produced by different vendors. However, the PackML state model is proposed as a *template*, rather than a design pattern. The line interface logic of a given machine must include all the states prescribed by the standard, even they are not meaningful for that specific equipment. Moreover, line supervision logic for complex packaging plants may require to know at any time if a machine is actually receiving (delivering) products from (to) subsequent machines in the line or not, which happens if its *infeed* (*outfeed*) part is blocked. In this case, the line controller should take corrective action, for example by switching products to different conveyor lines. This aspect is not covered by the PackML standard, but is instead included in the State Diagram of Fig. 8.

5.2. Alarm handling pattern

Alarm handling is a critical issue for industrial control engineers. In the packaging domain, in which motion control systems play a dominant role, machine alarms are generally classified and according to safety regulations into conditions for Normal Stop, Fast Stop and Safety Stop. In any of the three cases, mechanical parts in the machine must be stopped with a given deceleration, which is the highest possible (i.e. electromechanical brakes are applied) in the case of Safety Stop. Of course, these alarms classification defines implicitly a priority order. Within the Hierarchical Control pattern described before, the supervisor of the hierarchical level (i.e. machine, group or module level) should manage the alarms related to that specific part of the machine and coordinate stopping and recovery procedures for its components. The alarm handling logic can be specified as shown by the Statechart of Fig. 9, assuming the transition priority order discussed in Section 3. The collaboration between multi-level alarm handlers can also be specified, according to the Hierarchical Control pattern and the part/whole interaction rules discussed in Section 2, as shown in Fig. 10. The code implementing alarm handling logic for each supervisor component should be executed concurrently with the one performing management of *nominal* operations (i.e. implementing a state machine like the one of Fig. 8), possibly within a higher priority task.

5.3. Motion control pattern

Logic control of motion systems is another massive portion of a packaging machine software. Design issues are further complicated

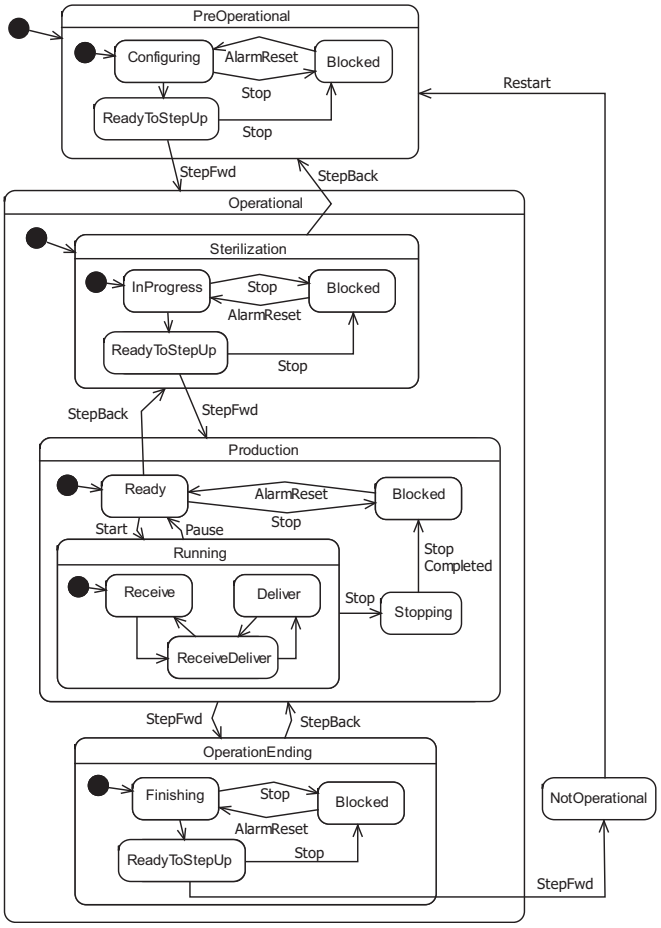

Fig. 8. UML State Diagram of the machine supervision logic for a Tetra Brik Aseptic® filler.

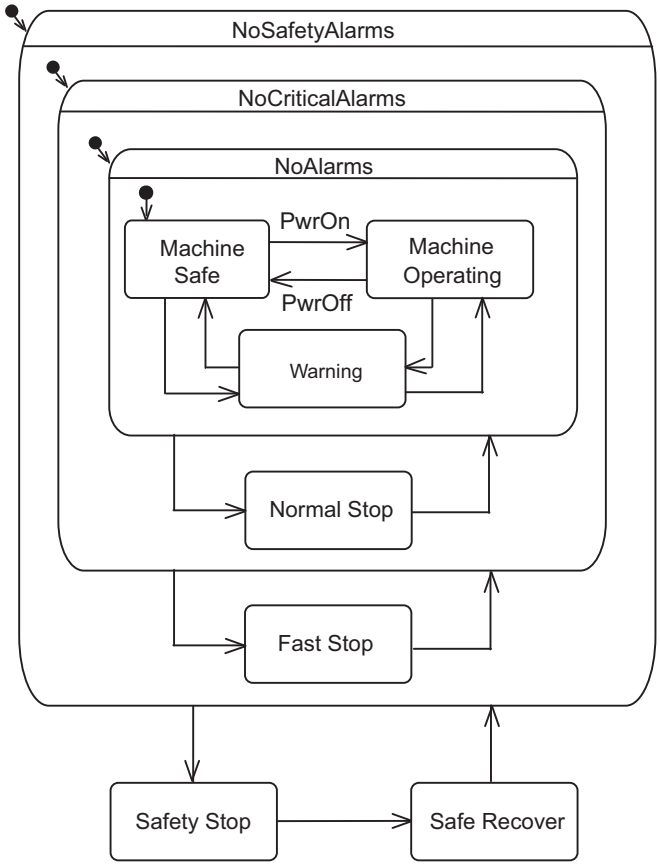

Fig. 9. Alarm handler for a typical packaging or labeling machine.

by the peculiarities of programming tools and functionalities implemented by different vendors. In order to mitigate the absence of standardization, PLCOpen has published a set of documents trying to provide a common *look-and-feel* for motion control Function Blocks, compatibly with IEC 61131-3 languages (PLCOpen Technical Committee 2, 2010). This set of standard FBs defines also a state machine describing the behavior of both single and coordinated axes. However, the PLCOpen state machine is not fully consistent with respect to alarm handling. Therefore, a more precise Statechart specification, shown in Fig. 11, for motion control management has been defined during the design of Tetra Pak and Sidel manufacturing systems.

In particular, the control logic can be described as follows. Upon start-up the control system closes the contactors between drives and power supply. Then the system waits for a command to energize the motors, after which contactors between drives and motors are closed and servo control loops are activated. If there are no faults in the drives, the motors can be moved in order to set up a *home* position or, if homing was already performed in the past, it will go directly to the `Ready to Run` state. From `Ready to Run` the system can start repetitive cycles of motion profiles, required by the standard production process.

As is common for a multi-axis motion control system, the main issues to address are related to error handling. In particular, complex recovery procedures may be required after an emergency stop, which causes all the motors to decelerate in minimum time and, therefore, possibly lose synchronism with the coordinated motion profile. The complete sequence of recovery operations, embedded in the `Synchronization` state of the Statechart in Fig. 11 can be described as follows:

1. register the current position of each motor, which may differ from the position reached during the emergency stop because of manual movements performed by human operators during machine recovery;
2. identify the most suitable segment of the production running motion profile for subsequent restart;
3. calculate a motion profile to move the motors at the initial point of the segment identified at previous step, considering as a constraint to avoid mechanical interferences;
4. hook up the axes to the production running motion profile and set the index of the current segment to the one identified in step 2.

At the end of this sub-sequence, the motors are ready to execute the trajectories required by the motion profiler, since they are correctly synchronized in order to avoid mechanical interference.

6. Industrial control software implementation

Modern Computer-Aided Software Engineering (CASE) tools supports developers from requirements specification to coding and testing, by means of UML diagrams, especially Class and State Diagrams, that map directly into constructs of O-O programming languages. In fact, automatic code generation is a common practice for users of advanced CASE tools. Industrial control engineers working with IEC 61131-3 compatible systems are

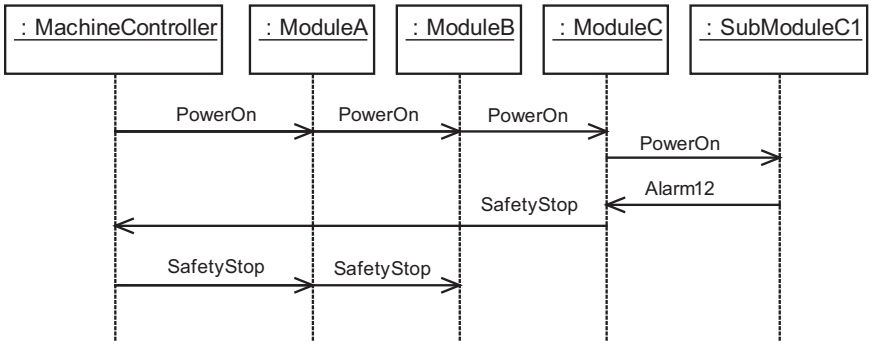

Fig. 10. UML Sequence Diagram for a generic alarm handling procedure.

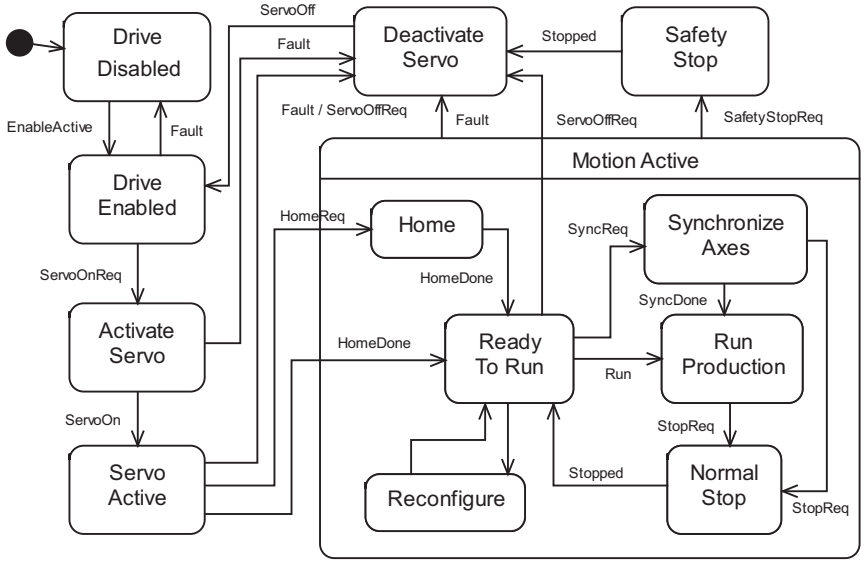

Fig. 11. Design pattern for a multi-axis motion control application.

instead less used to such a practice, mainly because of the lackness of commercially available model-based tools for PLC programming (being Simulink® PLC Coder a notable exception Mathworks Inc., 2010a). Of course, UML and Statecharts can be manually translated into PLC code, but this approach requires an implementation strategy preserving maintainability and readability of the code, especially considering modification of the control logic during field-test of the automation system. In the authors opinion, the most critical part of UML model-to-code translation is the implementation of Statecharts. Other authors (Vogel-Heuser et al., 2005) have focused more on mapping structural properties of Class Diagrams into IEC FBs, assuming that Statecharts can be mapped into IEC 61131-3 Sequential Function Charts (SFCs). However, some PLC vendors provide programming structures differing from IEC FBs (e.g. the equivalent concept of *Add-on Instructions* was introduced by Rockwell Automation Inc., 2010 only since 2007), that may require specific model-to-code mapping rules. Moreover, SFCs are not equivalent to Statecharts (Bauer & Engell, 2002), especially considering state hierarchy and inter-level transitions. The aim of this paper is instead to describe solution patterns for Statecharts implementation using PLC languages, such that:

- the highest number of Statechart features is supported, starting from hierarchy and inter-level transitions (i.e. direct mapping of Statecharts into SFCs is therefore not considered);
- the number of operations required to the developer to modify the resulting code directly at the PLC level is minimized;
- the resulting code is *human-readable*, especially considering the possibility to easily identify the active state when the PLC program is running, exploiting the debug mode features of PLC programming tools;
- the resulting code is computationally efficient.

Needless to say, the implementation of Statecharts using high-level and O-O programming languages (C/C++, Java, etc.) has a long history. An interesting review of the most common strategies, namely the *Nested Switch* pattern, the *State Table* pattern and the *O-O State* pattern, can be found in Chapter 3 of Samek (2008). Another pattern, based on *Equivalent Boolean Equations*, is included in the present discussion, because it can be easily implemented with classical Ladder Diagram (LD, International Electrotechnical Commission, 2002) code.

6.1. Equivalent Boolean equations pattern

The activation conditions for states in a hierarchial state machine can be re-written as Boolean equations. In fact, a state must be activated if it is in the *default completion* of the destination state of a firable transition, and deactivated if it is part of the *reflexive-transitive closure* of the *children* of the source state of a firable transition. More details, including formalization of these

concepts, can be found in Harel and Naamad (1996) and Chan et al. (1998). Boolean equations for the selection of firable transition, update of the active state configuration and execution of actions for the Statechart example of Fig. 3 can be easily translated in LD as follows[1]:

1. Transitions are firable if their trigger and enabling conditions evaluate to true and no higher-priority transition is simultaneously enabled (recalling that a Statemate-like semantics is adopted):

```
| State1      Condition                T1
+---| |--------| |------------------( )
|
| State1_1   Condition     T1          T2
+---| |--------| |-------|/|--------( )
|
```

2. A state is active if it was active in the previous PLC cycle, OR if any of its entering transitions is firable transition, AND its superstate is still active AND there are no firable exiting transitions. The set of entering and exiting transitions for each state is defined according to the default completion and reflexive-transitive closure rules mentioned before.

```
|    T3         T2   State1  State1_1
+---| |----+---|/|---| |------( )
|   T11    |
+---| |----+
| State1_1 |
+---| |----+
|
```

3. An action is executed if it is a *do* action and its related state is active, or if it is an *entry* action and its related state has been activated in the latest execution cycle, or if it is an *exit* action and its related state has been deactivated in the latest execution cycle.

```
| State1_1  (* entry *)       Var2
+----|P|----------------------(S)
|
| State1_1  (* exit  *)       Var2
+----|N|----------------------(R)
|
...
| State3  (* do *) VarIn1     Var6
+----| |------------| |--------( )
|
```

Benefits of the pattern:

- the translation in LD is straightforward, since the language is specifically designed to evaluate Boolean expressions;
- LD is well-known in the PLC domain and its on-line debugging is strongly supported by visual features of PLC programming tools (i.e. typically showing virtual current that flows in contacts and coils);

Drawbacks of the pattern:

- it is not based on state-of-the-art software engineering approaches (i.e. it does not promote code reuse);
- manual computation of entering and exiting transitions for each state is error-prone, if the Statechart is complex.

6.2. Nested Switch pattern

The most popular pattern for the implementation of hierarchical state machines with textual high-level languages is the Nested Switch pattern. Each level in the hierarchy is associated to a scalar state variable used as a discriminator of a `switch` (i.e. `CASE .. OF` in IEC 61131-3 Structured Text or ST language) statement. State machine levels nesting is therefore mapped into nesting of the `switch` statements. Transitions are evaluated in the `switch` case of their source state by means of `if-then` constructs and, if a transition is firable, the new state configuration is assigned. Since inter-level transitions inherently break the hierarchy of `switch` statements, auxiliary flags and variable are necessary to preserve the behavioral specification. For example, an additional scalar variable can be introduced for the assignment of next state values, whose usefulness is twofold: to ensure the transition priority scheme of the Statemate-like semantics and to enable *exit* actions of higher level states if an inter-level transition (e.g. `T4` in Fig. 3) is fired. Assuming that the following data-type is defined:

```
TYPE STATELEVEL:
STRUCT
  CurrentState: DINT;
  NextState: DINT;
  Entry: BOOL;
END_STRUCT
END_TYPE
```

and that it is instantiated as an array called `States`, with an element for each hierarchical level, the implementation of Fig. 3 example can be schematized as follows:

```
CASE States[0].CurrentState OF
State1:
IF States[0].Entry THEN
(* Entry actions *)
END_IF
(* Do actions *) ...
(* Transitions *)
IF (Event1 AND
((Var1 > 10)ANDVar3)(*T1*)
THEN
States[0].NextState := State3;
States[1].NextState := State31;
END_IF
(* Substates *)
CASE States[1].CurrentState OF
State1_1: (* Entry actions *)...
(* Do actions *) ...
(* Transitions *)
IF States[1].CurrentState
=States[1].NextState
THEN (* No conflicting transitions *)
IF ...(* T2, T3 *)
IF Event2 (* T4 *)
THEN
States[0].NextState := State2;
States[1].NextState := State21;
```

[1] A LD *rung* assigns the result of a Boolean expression, in which AND and OR operations are programmed respectively by series of contacts or parallel contacts, to a virtual coil in the rightmost part of the rung.

```
Var6 := TRUE; (*Transitionaction*)
END_IF
END_IF
IF States[1].CurrentState
< > States[1].NextState
THEN (* Exit actions/set Entry flag *)
...
END_IF
...
END_CASE
IF States[0].CurrentState
< > States[0].NextState
THEN (* Exit actions/set Entry flag *)
...
END_IF
...
END_CASE
```

Benefits of the pattern:

- the hierarchy of the Statechart is preserved in the hierarchy of the `switch` constructs;
- textual code can be easily generated automatically with modern CASE tools (in fact Simulink PLC Coder implements this strategy);
- textual code is more portable than graphical code (e.g. LD).

Drawbacks of the pattern:

- inter-level transitions requires slightly obscure workarounds, that limit code readability;
- it does not promote code reuse;
- manual coding is difficult to maintain;
- on-line debugging of textual code is not intuitive as it is for graphical languages.

6.3. State Table pattern

Another popular approach is to use multi-dimensional tables containing arrays of transitions and related source and destination states. For example, Table 1 shows a possible solution mapping state configurations of Fig. 3 example.

This table could be implemented in IEC 61131-3 by defining the following data-type:

```
TYPE TRANSITIONS:
STRUCT
  (*A source and a destination for each hierarchical
level *)
  Source: ARRAY[1..LEVELS] OF DINT;
  Destination: ARRAY[1..LEVELS] OF DINT;
  Condition : BOOL;
END_STRUCT
END_TYPE
```

Table 1
State Table for the Statechart example.

Current state configuration		Transition	Next state configuration	
Level 1	Level 2		Level 1	Level 2
State 1	ANY	T1	State 3	State 3_1
State 1	State 1_1	T2	State 1	State 1_2
State 1	State 1_2	T3	State 1	State 1_1
State 1	State 1_2	T4	State 2	State 2_1
⋮	⋮	⋮	⋮	⋮
State 2	State 2_2	T7	State 3	State 3_2

embedding source/destination state configurations and the Boolean result of triggering/enabling conditions for each transition. The configuration table is then mapped into an array of the previous data-type, which requires a proper static initialization according to the Statechart structure. At run-time, the state configuration is stored into an array, encoding the currently active set of states:

```
(* Transitions table *)
TRXs : ARRAY[1..N_TRANS] OF TRANSITIONS;
(* State array: a DINT for each
  hierarchical level *)
STATE_CONFIG : ARRAY[1..LEVELS] OF DINT;
```

Finally, once that each transition firing condition has been evaluated, processing of the table can be implemented as a reusable parametric function that scans the table with nested `FOR..DO` loops and if a transition is firable, assigns to `STATE_CONFIG` the newest value.

Benefits of the pattern:

- State Tables can be easily generated automatically with modern CASE tools;
- it promotes code reuse, especially for the table processing part;
- on-line debugging requires only visualization of the data structures.

Drawbacks of the pattern:

- the hierarchy of the Statechart is not preserved explicitly;
- requires careful initialization of the State Table.

6.4. O-O State pattern

To conclude this review of patterns for Statecharts implementation, it is briefly recalled the O-O State pattern, which has been intensively elaborated by Samek (2008) and many others. The pattern is based on inheritance and polymorphism and can be schematized by the Class Diagram of Fig. 12. Transitions (i.e. events) handling is initiated by the methods of the `HSM` class, which is composed of instances of the `State` class, and delegated to the methods of the latter. The hierarchy of the Statechart is mapped into nested aggregations of the same `State` class.

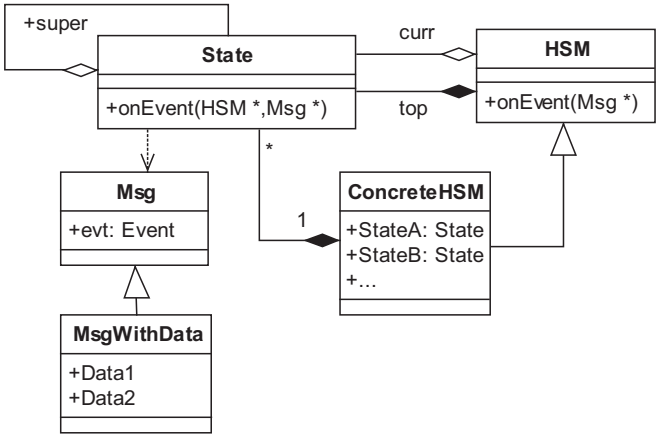

Fig. 12. O-O pattern for hierarchical state machines (Samek, 2008).

In addition to O-O features, the possibility to use pointers to functions further increases code reusability.

Needless to say, the main drawback of this pattern is that IEC 61131-3, in its current form, does not allow its implementation, so that in this paper it will not be discussed in more detail. Further investigations are in progress to verify if, at least partially, the O-O State pattern can be implemented with the only known PLC programming tool supporting O-O IEC 61131-3 extensions (i.e. Codesys V3).

Remarks. Code reusability is strongly promoted by both State Table and O-O State patterns. However, the O-O State pattern promotes reusability mainly because of inheritance features. Moreover, the O-O State pattern facilitates the application of the UML semantics priority scheme, since code in low-level states is evaluated first (see Samek, 2008), instead of the Statemate-like one adopted in the proposed applications. Therefore, if code reuse is strongly needed, the most reasonable choice for PLC code implementation of Statecharts is the State Table pattern. On the other hand, the whole lifecycle of complex manufacturing systems, like those described in this paper, includes several maintenance and updates to the PLC code that may be applied by field-test technicians. These technicians commonly have a strong background on Ladder Diagram programming and, according to the authors experience, their preferred implementation pattern is the Equivalent Boolean Equations one, mainly because it promotes the use of LD (even though the pattern itself, like the others, can be implemented in any language) and it is based on traditional *logic-based* PLC programming solutions (e.g. *self-retentive* coils for state bits management).

7. Conclusion

The paper presented a practical approach to 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.) and reusable design patterns into the machine design. The patterns presented in this paper have driven control software programmers in the development stage of real-world systems, in order to cope with the increasing system complexity, preserving efficiency and reusability of *mechatronic* components. The experience acquired during the development of the proposed case studies teaches that PLC programming is slightly different from other software design tasks, in which proposals for methodologies and implementation guidelines must cope with the features of industrial tools, the background of control engineers or technicians and some sort of conservativism, though the latter is primarily justified by safety and reliability requirements.

On the other hand, new standards for industrial control programming, like IEC 61499 and the revised IEC 61131-3, to be published in the near future, will push further the application of the O-O paradigm and ease the adaptation of software engineering patterns, like for example the O-O State pattern previously described, to the manufacturing domain.

Acknowledgments

The authors fully acknowledge Tetra Pak Packaging Solutions S.p.A. and Sidel S.p.A. for their support and cooperation during the research work described in the paper.

References

3S Smart Software Solutions (2010). *Codesys V3—The IEC 61131-3 development system* 〈http://www.3s-software.com〉.

Bassi, L., Secchi, C., Bonfè, M., & Fantuzzi, C. (2011). A SysML-based methodology for manufacturing machinery modeling and design. *IEEE/ASME Transactions on Mechatronics*, *16*, 1049–1062.

Bauer, N., & Engell, S. (2002). A comparison of sequential function charts and Statecharts and an approach towards integration. In: H. Ehrig, & M. Grosse-Rhode (Eds.), *Proceedings of the 2nd international workshop on integration of specification techniques for applications in engineering* (pp. 58–69). Berlin: Technische Universität.

Bonfè, M., & Fantuzzi, C. (2003). Design and verification of industrial logic controllers with UML and Statecharts. In *IEEE conference on control applications*, Istanbul, Turkey.

Bonfè, M., Fantuzzi, C., & Secchi, C. (2005). Verification of behavioral substitutability in object-oriented models for industrial controllers. In *Proceedings of the IEEE international conference on robotics and automation* (*ICRA*), Barcelona, Spain.

Bonfè, M., Fantuzzi, C., & Secchi, C. (2006). Behavioural inheritance in object-oriented models for mechatronic systems. *International Journal of Manufacturing Research*, *1*, 421–441.

Burmester, S., Giese, H., & Tichi, M. (2005). Model-driven development of reconfigurable mechatronic systems with Mechatronic UML. In *Lecture notes on computer science* (Vol. 3599/2005).

Cengic, G., Ljungkrantz, O., & Akesson, K. (2006). A framework for component based distributed control software development using IEC 61499. In *Proceedings of the IEEE international conference on emerging technologies and factory automation* (*ETFA*). Prague, Czech Republic.

Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., et al. (1998). Model checking large software specifications. *IEEE Transactions on Software Engineering*, *24*, 498–520.

Crane, M., & Dingel, J. (2007). UML vs. classical vs. Rhapsody Statecharts: Not all models are created equal. *Software and Systems Modeling*, *6*, 415–435.

Douglass, B. P. (1999). *Doing hard time: Developing real-time systems with UML, objects, frameworks and patterns*. Addison Wesley Longman.

Douglass, B. P. (2006). *Real time design patterns*. Addison Wesley.

Dubinin, V., Vyatkin, V., & Pfeiffer, T. (2005). Engineering of validatable automation systems based on an extension of UML combined with Function Blocks of IEC 61499. In *IEEE international conference on robotics and automation* (pp. 3996–4001), Barcelona, Spain.

Fantuzzi, C., Bonfè, M., & Secchi, C. (2009). A design pattern for model based software development for automatic machinery. In *Proceedings of the 13th IFAC symposium on information control problems in manufacturing* (*INCOM*), Moscow, Russia.

Ferrarini, L., & Veber, C. (2005). Design and implementation of distributed hierarchical automation and control systems with IEC 61499. In *Proceedings of the IEEE international conference on industrial informatics* (*INDIN*), Perth, Australia.

Gomaa, H. (2011). *Software modeling and design: UML, use cases patterns and software architectures*. Cambridge University Press.

Gonzalez, V., Diaz, A., Fernandez, P., Junquera, A., & Bayon, R. (2010). MIOOP, an object-oriented programming paradigm approach on the IEC 61131 standard. In *IEEE conference on emerging technologies and factory automation*, Bilbao, Spain.

Hametner, R., Zoitl, A., & Semo, M. (2010). Automation component architecture for the efficient development of industrial automation systems. In *Proceedings of the 6th IEEE conference on automation science and engineering* (*CASE*), Toronto, Canada.

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

Harel, D., & Naamad, A. (1996). The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodologies*, *5*(4), 293–333.

Heverhagen, T., Tracht, R., Hirschfeld, R., 2003. A profile for integrating function blocks into the unified modeling language. In *SVERTS: Specification and validation of UML models for real time and embedded systems. Sixth international conference on the unified modeling language*.

International Electrotechnical Commission (2002). *IEC 61131-3. Programmable controllers—Part 3: Programming languages* (2nd ed.). Final Draft International Standard (FDIS).

International Electrotechnical Commission (2003). *IEC 61131-3. Programmable controllers—Part 3: Programming languages* (2nd ed.). International Standard.

International Electrotechnical Commission (2005). *IEC 61499-1. Function blocks for industrial process measurement and control—Part 1: Architecture.* International Standard.

International Society of Automation (2008). *Machine and unit states: An implementation example of ISA-88* 〈www.isa.org〉. ISA-TR88.00.02-2008.

Katzke, U., & Vogel-Heuser, B. (2005). UML-PA as an engineering model for distributed process automation. In *IFAC World Congress* (Vol. 16). Prague, Czech Republic.

Klotz, T., Fordran, E., Straube, B., & Haufe, J. (2009). Formal verification of UML-modeled machine controls. In *IEEE international conference on emerging technologies and factory automation*, Mallorca, Spain.

Lobov, A., Lastra, J., & Tuokko, R. (2005). Application of UML in plant modeling for model-based verification: UML translation to TNCES. In *IEEE International Conference on Industrial Informatics*, Perth, Australia.

Lüder, A., Hundt, L., Foehr, M., Wagner, T., & Zaddach, J. -J. (2010). Manufacturing systems engineering with mechatronical units. In *Proceedings of the IEEE international conference on emerging technologies and factory automation* (*ETFA*), Bilbao, Spain.

Mathworks Inc. (2010a). Simulink® PLC coder 〈http://www.mathworks.com〉.

Mathworks Inc. (2010b). Stateflow® 〈http://www.mathworks.com〉.

Maurmaier, M., & Göhner, P. (2009). Model-driven development in industrial automation—automating the development of industrial automation systems using model transformations. In *ICINCO-RA* (pp. 244–249).

Object Management Group (2009). *UML v. 2.2 superstructure specification*. Document N. formal/2009-02-02 〈http://www.omg.org/spec/UML/2.2/〉.

Object Management Group (2010). *System modeling language (SysML) specification. v. 1.2.* 〈http://www.sysml.org/specs〉.

OMAC Users Group (2006). *Guidelines for packaging machinery automation V3.1* 〈http://www.omac.org〉.

Pazzi, L. (2000). Part–whole Statecharts for the explicit representation of compound behaviours. In A. Evans, S. Kent, & B. Selic (Eds.), *Proceedings of UML 2000. Lecture notes on computer science* (Vol. 1939, pp. 541–555). Springer-Verlag.

Pintér, G., & Majzik, I. (2005). Modeling and analysis of exception handling by using UML Statecharts. In *Scientific engineering of distributed java applications. Lecture notes on computer science* (Vol. 3049/2005, pp. 58–67).

PLCOpen Technical Committee 2 (2010). *Technical specification—function blocks for motion control—Version 2.0* 〈http://www.plcopen.org〉.

PLCOpen Technical Committee 6 (2009). *Technical paper—XML formats for IEC 61131-3—Version 2.01* 〈http://www.plcopen.org〉.

Ritala, T., & Kuikka, S. (2007). UML automation profile: Enhancing the efficiency of software development in the automation industry. In *IEEE international conference on industrial informatics*. Vienna, Austria.

Rockwell Automation Inc. (2010). RSLogix 5000® 〈http://www.rockwellautomation.com〉.

Samek, M. (2008). *Practical statecharts in C/C++*. (2nd ed.). Newnes.

Sanz, R., & Zalewski, J. (2003). Pattern-based control systems engineering—Using design patterns to document, transfer, and exploit design knowledge. *IEEE Control Systems Magazine*, 42–60.

Schünemann, U. (2007). Programming PLCs with an object-oriented approach. *Automation Technology in Practice*, 59–63.

Secchi, C., Bonfè, M., & Fantuzzi, C. (2007). On the use of UML for modeling mechatronic systems. *IEEE Transactions on Automation Science and Engineering*, *4*, 105–113.

Selic, B., Gullekson, G., & Ward, P. (1994). *Real-time object-oriented modeling*. John Wiley & Sons.

Storr, A., Lewek, J., & Lutz, R. (1997). Modelling and reuse of object-oriented machine software. In *Proceedings of the European conference on integration in manufacturing* (pp. 475–484), Dresden, Germany.

Thramboulidis, K. (2004). Using UML in control and automation: A model driven approach. In *IEEE International conference on industrial informatics* (pp. 587–593), Berlin, Germany.

Thramboulidis, K. (2008). Challenges in the development of mechatronic systems: The mechatronic component. In *IEEE conference on emerging technologies and factory automation*, Hamburg, Germany.

Thramboulidis, K. (2009). Different perspectives. face to face; IEC 61499 function block model: Facts and fallacies. *IEEE Industrial Electronics Magazine*, *3*, 7–26.

Thramboulidis, K. (2010). The 3+1 SysML view-model in model integrated mechatronics. *Journal of Software Engineering and Applications*, *3*, 109–118.

Tomizuka, M. (2002). Mechatronics: From the 20th to 21st century. *Control Engineering Practice*, *10*, 877–886.

Vogel-Heuser, B., Witsch, D., & Katzke, U. (2005). Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In *International conference on control and automation* (Vol. 16), Budapest, Hungary.

Vyatkin, V., Karras, S., & Pfeiffer, T. (2005). Architecture for automation system development based on IEC 61499 standard. In *Proceedings of the IEEE international conference on industrial informatics* (*INDIN*), Perth, Australia.

Werner, B. (2009). Object-oriented extensions for IEC 61131-3. *IEEE Industrial Electronics Magazine*, *3*, 36–39.

Witsch, D., Ricken, M., Kormann, B., & Vogel-Heuser, B. (2010). PLC-Statecharts: An approach to integrate UML-Statecharts in open-loop control engineering. In *IEEE international conference on emerging technologies and factory automation*, Osaka, Japan.

Witsch, D., & Vogel-Heuser, B. (2009). Close integration between UML and IEC 61131-3: New possibilities through object-oriented extensions. In *IEEE international conference on emerging technologies and factory automation*, Mallorca, Spain.

Younis, M., & Frey, G. (2006). UML-based approach for the re-engineering of PLC programs. In *IECON 2006—32nd annual conference on IEEE industrial electronics* (pp. 3691–3696).

Zoitl, A., Strasser, T., Sünder, C., & Baier, T. (2009). Is IEC 61499 in harmony with IEC 61131-3?. *IEEE Industrial Electronics Magazine*, *3*, 49–55.

Zoitl, A., & Vyatkin, V. (2009). Different perspectives. Face to face; IEC 61499 architecture for distributed automation: the "glass half full" view. *IEEE Industrial Electronics Magazine*, *3*, 7–26.