Design and Verification of Industrial Logic Controllers with UML and Statecharts

Marcello Bonfè‡, Cesare Fantuzzi§
‡Università di Ferrara, Dip. Ingegneria, Via Saragat 1, Ferrara - Italy.
§Università di Modena e Reggio Emilia, DISMI, Via Allegri 15, Reggio Emilia - Italy.
E-mail: mbonfe@ing.unife.it
E-mail: cesare.fantuzzi@ieee.org

Abstract—The paper describes a methodological framework that aims to apply formal design and verification techniques to the domain of Logic Control and Supervision for Manufacturing Systems. The methodology is based on an Object-Oriented approach, supported by a syntactical and semantical adaptation of the semi-formal software specification languages UML and Statecharts. The modeling languages have been subsequently formalized, according to a semantics that take into account the concepts described in the IEC 61131-3 Standard for industrial controllers programming, in order to prove correctness properties expressed in the temporal logic CTL. The verification process is performed by means of the model checking tool SMV.

I. INTRODUCTION

The task of control design for Manufacturing Systems represents a challenging and interesting problem, since the application domain presents heterogeneous characteristics and requires engineering efforts that include different kind of technological skills (i.e. mechanical engineering, electrical/electronic engineering, systems theory and computer science). In fact, a manufacturing machine is a real mechatronic system, which means that an efficient design methodology for the complete control system should take care of all the aspects involved in the project, especially those related to the integration between mechanical/electrical parts and the control software.

A manufacturing machine control system includes, in general, tasks for continuous control (mechanical motion, temperature regulation, etc.) and logic control (sequencing, interlocking and alarms handling, typically realized with Programmable Logic Controllers, PLCs). However, while formal analysis and design techniques for continuous control systems are well-known, even in the industry, and are supported by several widespread engineering tools, systematic and rigorous approaches to the solution of logic control problems are actually uncommon in the industrial practice. Despite the number of academical projects devoted to the application of formal methods in PLC programming [1], logic controllers for Manufacturing Systems are, in general, directly programmed from informal requirements and validated with intensive “on-line” tests.

There are many factors limiting the application of formal methods to logic control design in the manufacturing industry. First of all, the description and modeling languages, derived from Discrete Events Systems theory, typically adopted in the academic community are complex and obscure for the most part of industrial control engineers. Moreover, models for real-world systems may become complicated because of size, number of alarms and exceptions, etc., so that their graphical representation may become impractical. This problem can be solved adopting “modular” approaches and visual languages with features oriented to cope with the complexity of DES models for “real” systems, like, for example, Statecharts [2]. Another important issue is related to the definition of adequate techniques to solve the control problem. The approaches oriented to the automatic synthesis of a controller (or a supervisor [3]), given a model of the uncontrolled system and of the expected behaviour, seem quite difficult to apply in most practical cases of industrial interest. A more suitable approach would be to map directly functional specifications, even if given with

informal descriptions, into an operational input/output model of the controller, assuming that the plant’s reactions are always consistent with the expectation of the controller, which must therefore include explicitly alarms monitoring and exception handling (direct design approach). Finally, the implementation of the control system as a software program may also result a complex task, therefore it is important to consider the peculiarities of industrial control devices and their programming tool from a software development point of view, taking into account aspects related to efficiency, readability and maintainability of the application. In this sense, concepts and principles derived from the Software Engineering discipline, particularly those related to Object-Oriented (OO) methods, should be applied from the beginning of the development process.

The aim of the paper is to describe a design and verification methodology that can overcome the limiting factors described above, thanks to a syntactical adaptation and semantical formalization of some appealing software specification languages, while preserving rigorosity of the formal approach. In particular, Section II presents the approach and cardinal principles of the proposed methodology. Section III presents the modeling and specification languages adopted during the control design phase and Section IV shows how those semi-formal languages can be formalized in order to apply verification procedures. The paper ends with an illustrative example and some concluding remarks.

II. THE CONTROL DESIGN METHODOLOGY FOR MANUFACTURING SYSTEMS

In order to identify a practical and efficient methodology for the design and specification of industrial control systems, it is necessary to discuss which is the correct approach to the analysis and design phase, considering the peculiarities and critical factors of the application domain, which modeling and specification languages should be used, which development process (i.e. ordered set of activities) should be followed and, finally, which tools can support the designer during that process.

First of all, it is important to consider logic control design as part of a wider set of activities related to the global project of a machine. In fact, the design of a manufacturing machine is very often a composition of sub-processing solutions developed in previous projects. These “components” are related to a specific functionality of the global production process and are typically realized with a well-defined set of mechanical parts, sensors and actuators. Of course, this means that the physical modules are also related to a well-defined set of control specifications, which could be mapped in a few software routines. It is evident that the efficiency of machine development would be greatly enhanced if mechanical/electrical design and logic control design were strictly related with each other and if modularity and reusability of the control software were cardinal principles of a methodological approach to the control problem. Therefore, the control design activity should start from the very beginning of machine project and its result must be a model of the software
application that is modular, precise but easy to understand, especially for engineers with different backgrounds (i.e. mechanical, electrical, etc.), verifiable with adequate formal tools and, finally, easy to implement in the programming platform adopted. In this way, the design specification model would be an efficient support also for the testing, documentation and maintenance phases.

The concepts of modularity and reusability are the basis of the Object-Oriented approach and, in fact, several researchers have tried to apply OO principles to industrial control design and implementation (see [4, 5]). However, practical applications of the methodologies reported are still limited to academic projects, mainly because of the difficulties to adapt the OO programming framework to languages and tools used in industrial control programming. Moreover, few research works are concerned with formal verification of OO design models.

In fact, an important aspect that must be discussed is the correct interpretation of “objects” in the domain of manufacturing machines control. In “business” software, an object is the encapsulation of a complex data structure with a large set of operations devoted to its manipulation. For manufacturing machines and their control software, instead, an “object” has a much more physical interpretation: a machine module would have reusability features if it consists of a tight aggregate of mechanical parts, sensors, actuators and control software routines, specifically related to a given part of the manufacturing process. In this context, it is valuable to define a module of that kind as a Mechatronic Object.

Moreover, several authors ([6], [7]) suggest that in real-time and reactive systems the most suitable interface for a software module should be based on signals and events, rather than explicit operation invocation. Therefore, a “mechatronic object” should have a signal-based interface, in order to send and receive synchronization and interlocking information, rather than an operation-based interface, like “pure” software objects. In practice, the software part of a “mechatronic object” would be a reusable module of the machine control program that has:

- an input/output interface of signals and data parameters that permits the interactions and synchronization with the rest of the control system;
- an input/output interface of sensory information and actuator commands required to control the physical part of the mechatronic object; this interface should be directly mapped into the I/O connections of the controller (i.e. PLC) and should also be considered as a “private” part of the mechatronic object;
- a private data set to store status information;
- a control algorithm, which determines the dynamic behaviour of the mechatronic ensemble, in order to match the functional requirements.

The features described above are actually fulfilled by the Function Block (FB) software structure described in the IEC 61131-3 document [8], which is currently the most important International Standard regulating programming languages for industrial control devices. The interpretation of Function Blocks as “mechatronic object” controllers, schematized in Figure 1, has been already described by the authors in [9].

Given these preliminaries, the task of analysis and design of logic control system for a complex manufacturing machine can be approached with an OO perspective, performing the following activities:

1) **Identification of objects** and their abstraction into a hierarchy of classes: in this phase both functional requirements and mechanical/electrical design should be analyzed, highlighting the entities that can be considered as related to independent control modules. The hierarchical relationship between composite and component modules would make necessary to define “supervisor” control routines, related to the composite object, to coordinate the behaviour of components, according to a nested encapsulation principle. The result of this activity is a structural model of the control system architecture.

2) **Identification of interfaces** for the objects identified in the previous step, with regard to both the connections between the controller and the plant and between modules inside the controller. The result of this activity is a data-flow model.

3) **Specification of controller’s behaviour**: this phase corresponds to the direct design of logic control modules, given the operational requirements upon the manufacturing machine; of course, the principal issue of this task is to decompose also the model (either formal or informal) of the machine into the control modules identified in previous phases. The result of this activity is a dynamic model of the controller, which should be expressed in an intuitive and powerful language for DES modeling (i.e. Statecharts).

4) **Formal verification**: the correctness of the model must be proved with formal methods, in order to guarantee safety and reachability properties, plus system-specific expected properties, expressed in an adequate specification language. Since the complexity of formal methods and languages is the factor that most limits their application in the manufacturing industry, it is important to make the verification procedure transparent to the designer, so that he can cope only with more intuitive graphical languages during the design phase. This would be possible by means of a subsequent translation of the design model into the input language of a formal verification tool.

5) **Implementation** into the programming languages and structures of the target platform. An important issue in this phase is to maintain the structural properties of the design model, in order to ensure modularity and reusability of the final software, and to respect behavioral specifications, in order to preserve correctness properties. In the methodology proposed by the authors, these objectives are achieved thanks to the strict relationship between modeling elements and the concepts and features of IEC 61131-3. In particular, techniques to implement Statecharts with the most used PLC programming languages (i.e. Ladder Diagram and Structured Text) have been developed. For sake of brevity, the latter aspect is not covered by the present paper.

### III. Modeling and Specification Languages

The most well-known OO specification language for software systems is currently the Unified Modeling Language (UML) [10], which defines a set of graphical notation to describe both architectural and behavioral aspects. In particular, structural views of an OO
A complete and realizable structural model for a machine control systems should be defined by Class Diagrams in which the hierarchical architecture is modeled by adequate composition links between mechatronic classes. Well-formed rules prescribe that there must be a single “top-level” class (i.e. the machine) and composition links must be qualified with fixed and unambiguous multiplicity, since dynamic creation of objects is not consistent neither with their physical interpretation nor with the features of IEC 61131-3. The latter would not permit to model also generalization links, which involve inheritance mechanisms. However, rules to include this concept in the abstract model are currently being defined, on the base of formal procedures to ensure that two mechatronic classes are substitutable by interface and behaviour.

With regards to the specification of mechatronic objects interactions, the concepts described in [7], which have been the basis for the definition of a UML profile for Real-Time systems, seems more adequate than the standard language. In practice, Collaboration Diagrams, based on methods invocation, should be replaced with a description of the signal/data-flow interconnections between object instances, with a notation that emphasize the interpretation of <<input>> and <<output>> attributes as “ports” of the system components. The notation of stereotyped Collaboration Diagrams have been called Mechatronic Data-Flow Diagram, of which an example is shown in Figure 3.

The graphical notation of Statecharts is perfectly adequate for the specification of logic control systems, especially with regard to industrial applications. In fact, since the first presentation of that language in [2], it have been universally considered as an efficient way to describe models for complex reactive systems, thanks to the features of concurrency and hierarchy. Moreover, the possibility
to define inter-level transitions permit to include very easily in the specification alarms and exceptions, which is a critical aspect for real manufacturing machines. In this sense, Statecharts are more powerful than other DES models widely used in the industry, like Grafcet [11] or Sequential Function Charts [8].

With regard to textual expressions, instead, some elements of the Statecharts language can be adapted to the application domain by means of a redefinition in terms of IEC 61131-3 syntax. In particular, transitions may have a label that contains an expression of events, a guard condition and a list of actions, expressed in the traditional form proposed by Harel:

trigger[guard]/actions

but events in the trigger may be only: the rising edge (denoted with R_TRIG(Expression)) of the result of a boolean expression (compatible with IEC 61131-3), that may include attributes of the ≪mechatronic≫ class and elements of a related ≪hardware≫ class; the falling edge of the result of a boolean expression (denoted with F_TRIG(Expression)). The guard condition must be a boolean expression defined with the same syntactical rules prescribed by IEC 61131-3.

With regard to actions, they can be associated to either a transition or a state. In the latter case, the action would be specified by a textual expression like:

when / action IF[guard]

where when is a qualifier, that can be entry, exit or do, and guard is an optional boolean expression that may prevent the action from being executed, if it evaluates to false.

The kind of actions that can be executed are restricted to a set of operations that derives from the concepts of the IEC 611313-3 programming languages. In particular, an action can set or reset persistently a boolean variable (with SET(Var) or RESET(Var)), enable a boolean variable as long as the action is executed (EN(Var)), or assign the value of an expression to a variable of a non-boolean data-type, denoted with Var_name := Expression.

To conclude, the behaviour modeling language adapted to IEC 61131-3 can be extended to include timing specifications, by means of the use of the standard Function Blocks TON (on-delay timer), TOF (off-delay timer) and TP (pulse timer) [8]. In fact, the only input variables for each one of these FBs are the counting enable IN(boolean) and the preset value PT, a variable of TYPE data-type, which can be modified by Statecharts actions according to the rules previously described, while the timer output Q is a boolean variable that can be easily included in guard expressions.

IV. FORMALIZATION AND VERIFICATION OF THE DESIGN MODEL

The design methodology proposed in Section II and the specification languages previously described are suitable for the practical application to logic control problems of industrial interest. In particular, the modeling notations are intuitive and understandable even for designers without a strong background on formal methods, and the design models can be easily translated, preserving both architectural and behavioral properties, into programs written with the most used programming languages for industrial control devices. However, the techniques proposed so far are not “formal” in a strict sense. Correctness of the design model still depends from the quality and preciseness of the initial functional requirements and from the skillness of the designer. A possible solution to this problem would be to define a formalization of the design model, according to an adequate semantical interpretation, in order to obtain a description that can be understood by a formal verification tool. One of the most efficient verification technique for finite state system is, up to now, Symbolic Model Checking [12], which is supported by an efficient tool called SMV, developed at Carnegie Mellon University. The SMV tool verifies properties expressed in the temporal logic CTL [13] for finite state models described as Kripke structures. A Kripke structure is a tuple \( M = (S, R, L) \), where \( S \) is a set of states, \( R \) is the transition relation \( \subseteq S \times S \), that gives the set of all pairs \((s, t)\) such that \( t \) is an immediate successor of \( s \), and \( L : S \rightarrow 2^{AP} \), in which \( AP \) is a set of atomic propositions, is a valuation or labeling that associates to each state \( s \) a set \( L(s) \) of atomic propositions defined in \( AP \) that are true at state \( s \).

In order to formalize and describe with the SMV language the Kripke structure equivalent to a given mechatronic objects model, it is necessary to define a semantical interpretation consistent with some assumptions about the execution of the final control software. First of all, the static instantiation of the “top-level” class of the model results in a mechatronic system \( M_1 = (M_1, M_2, \ldots, M_n) \), where each \( M_i = (S, T, V, I, O) \), is a mechatronic module composed in its turn by a set \( S \) of states, a set \( T \) of transitions, a set \( V \) of internal variables, including also inputs and outputs related to the hardware part (i.e. ≪hardware≫ objects are modeled simply as signals, without an explicit structural entity), a set \( I \) of (software) input and a set \( O \) of (software) output. It is assumed that the software modules related to the mechatronic system are executed sequentially, according to the hierarchy and to a predefined order between modules at the same hierarchical level, within the same task, which is a typical choice in real PLC systems. The execution of a module modifies its state configuration according to an interpretation algorithm derived from the one described in [14] (the so-called Step Algorithm), with the synchronous time model. In fact, the Run-To-Completion algorithm (based on event queues) proposed in the UML standard is not, in our opinion, a suitable Statecharts semantics for the IEC 61131-3 software model. Moreover, also the priority scheme adopted in UML to solve transition conflicts, rising from simultaneous enabling of a transition exiting a super-state and a transition exiting one of its sub-state, is not adequate to implement state-based exception handling and preemption mechanisms. In this case, higher priority should be given to the transition exiting the highest level state, while priority of transitions exiting the same state should be defined explicitly, in order to ensure determinism.

Given these preliminaries, it is possible to describe the Kripke structure equivalent to a mechatronic system with a SMV program composed of modules defined as:

```plaintext
MODULE Mech_Class(EN, Inputs, READ_HW_INPUTS) VAR Exec : {IDLE, ACTIVE, EXECUTING1, EXECUTING2, ..., FINISHED};

where EN is a boolean input parameter that is true when the module is currently executing and READ_HW_INPUTS is a variable that permit the synchronous update of all the hardware inputs of the complete controller, as is common in PLC systems. The state variable Exec takes the enumerated value according to the sequence of execution followed by the controller: the state ACTIVE is the one in which the given module is performing its own “step”, while the states EXECUTING1 are related to the execution of sub-modules. The other state variables will be defined according to the Statechart specification, encoding the hierarchy as follows:

VAR Root : {State1, State2, ..., StateN}; SUBState1 : {S1_State1, ..., S1_StateN};

In order to define the transition relation, it is necessary to introduce the following predicates, that permit to evaluate the current state configuration and the set of enabled and actually firable transitions:

DEFINE IN-StateX := (Root = StateX); IN-StateY := IN-StateX & (SUBStateX = StateY); EN-Trans1 := IN-SourceState & Trigger & Guard; CONFLICT-Trans1 := EN-Trans2 | EN-Trans3 | ...;
```

Given these preliminaries, it is possible to describe the Kripke structure equivalent to a mechatronic system with a SMV program composed of modules defined as:

```plaintext
MODULE Mech_Class(EN, Inputs, READ_HW_INPUTS) VAR Exec : {IDLE, ACTIVE, EXECUTING1, EXECUTING2, ..., FINISHED};

where EN is a boolean input parameter that is true when the module is currently executing and READ_HW_INPUTS is a variable that permit the synchronous update of all the hardware inputs of the complete controller, as is common in PLC systems. The state variable Exec takes the enumerated value according to the sequence of execution followed by the controller: the state ACTIVE is the one in which the given module is performing its own “step”, while the states EXECUTING1 are related to the execution of sub-modules. The other state variables will be defined according to the Statechart specification, encoding the hierarchy as follows:

VAR Root : {State1, State2, ..., StateN}; SUBState1 : {S1_State1, ..., S1_StateN};

In order to define the transition relation, it is necessary to introduce the following predicates, that permit to evaluate the current state configuration and the set of enabled and actually firable transitions:

DEFINE IN-StateX := (Root = StateX); IN-StateY := IN-StateX & (SUBStateX = StateY); EN-Trans1 := IN-SourceState & Trigger & Guard; CONFLICT-Trans1 := EN-Trans2 | EN-Trans3 | ...;
```
Finally, the transition relation is encoded in a set of assignments, that initialize and update properly the state variables and the other internal attributes of the module:

\[
\begin{align*}
\text{ASSIGN} \\
\text{init(Exec)} & := \text{IDLE}; \\
\text{init(Root)} & := \text{Statel}; \quad \text{Default} \\
\text{next(Root)} & := \text{case} \\
& \quad \text{FIRABLE-Trans1} := \text{State2}; \\
& \quad \text{...} \\
& \quad 1 : \text{Root}; \\
& \quad \text{esac}; \\
\text{next(Hw_In1)} & := \text{case} \\
& \quad \text{READ_HW_INPUTS} := \{0,1\}; \\
& \quad 1 : \text{Hw_Input1}; \\
& \quad \text{esac};
\end{align*}
\]

Notice that the value of hardware inputs is updated non-deterministically only when the controller is in the “read inputs” phase. The private variables and outputs of each module are updated according to the actions specified in the Statechart diagram.

Once that the design model has been formalized in a SMV program, it is possible to verify the desired correctness properties expressing them with CTL formulas. The model checking tool will prove that either the formula is true (i.e. the system holds the property) or that it is false, showing, if possible, a counterexample (i.e. a computational path in which the system violates the property). For example, state reachability can be expressed with:

\[
\text{SPEC AG ModuleX.IN-StateY, which means that there exists a computational path in which the system reaches somehow the given state. To prove that a given state is not a deadlock, it is necessary to verify: SPEC AG (ModuleX.IN-StateY -> EF (!ModuleX.IN-StateY)).}
\]

Other system-specific properties, for example, mutual exclusion of undesired outputs configuration, cause-effect relationships and so on, can be expressed with the help of the numerous patterns and templates that can be found in the literature [15]. Methods to obtain CTL formulas from graphical specifications (i.e. timing diagrams) are currently under investigation.

V. A CASE OF STUDY

The proposed design and verification methodology has been applied to practical problems, especially regarding machines for goods packaging. An example of a very common module in a packaging machine is the system schematically shown in Figure 4. This module, that can be generically named Splicing Unit (SU), is in charge of unreeeling packaging material (i.e. plastic film) bobbins and manage the automatic splice, when necessary, of the film in a new bobbin to that of the one currently “running”. The design specification that will be described in the following is purely illustrative, therefore many details are not be discussed in depth.

The packaging material flowing from one of the bobbins runs over a set of pulleys, called dancer, whose position is dependent from the difference between the speed of the reel’s motion and the speed of the machine components that “consume” the packaging material. The packaging material of the other bobbin, not flowing in the dancer, must be prepared as shown by the figure, so that the film of the two reels can be spliced, by simply pushing forward and heating the sealing head. The splicing operation ensure continuity in the packaging material flow when the bobbin currently running gets empty. The motion of the reels is actuated by two independent induction AC motor, both controlled by the same single frequency converter drive. Two switches, that must not be simultaneously closed, permit to disconnect each motor from the power line. The speed of the motor must be controlled in order to keep the position of the dancer within a specified range (the speed control algorithm will not be considered). When the splicing operation is required, the running motor is stopped and the sealing head is activated. During the splicing, the machine does not stop, so that the packaging material stored in the dancer is consumed. Once that splicing is completed, the status of the two switches connecting the motors to the power line is reversed and the speed control algorithm is enabled again, in order to restore the dancer’s nominal position.

After an analysis of the machine module, considering the functional requirements and the physical components (mechanical or electrical), it is possible to identify the following objects: the “Sealing Head”, the “Dancer” (related to a position control algorithm executed by a module that will be simply enabled or disabled by the logic controller), the “Motion System” and the related “Frequency Converter” and “Switches”. The composition of “Sealing Head” and “Motion System” control modules, forming the complete “Splicing Unit”, would require also a logic control algorithm to supervise their interactions. In particular, this algorithm may include monitoring functions (i.e. alarms detection) and management of the global status of the Splicing Unit.

The architectural specification of the control system would be therefore described by the Class Diagram shown in Figure 5. Notice that the “Sealing Head” class and the “Motion System” class are associated, which means that there is a signal exchange that permit to stop and restart the bobbins’ motion accordingly to the splicing phases. Apart from this case, all the other signal flows are directed in the intuatable manner, therefore, the Mechatronic Data-Flow Diagram can be omitted for shortness.

The behavioral specification of the controller is divided into three statecharts: one for the “Splicing Unit” composite class (omitted), one for the “Sealing Head” (omitted) and one for the “Motion System” (Figure 6).

The design model can be verified with SMV in order to prove that all the states are reachable and there are no deadlocks. Another important property that must be satisfied can be expressed as: SPEC AG !(MotionSys.Bobbin1Running & MotionSys.Bobbin2Running), which states that the two motor switches are never simultaneously closed. To conclude, the possibility of SMV (as every model checker) to generate counter-examples for negative answers, can be exploited to obtain test sequences that can be used as validation patterns on the real system. For example, state reachability properties can be expressed in the negated form SPEC !EF ModuleX.IN-StateY, for which SMV generates a computational path, in which StateY is actually reached as expected, corresponding to an I/O trace that should be observed during “online” tests.

\[\text{FIRABLE-Trans1} := \text{EN-Trans1 & !CONFLICT-Trans1 & (Exec = ACTIVE)};\]
VI. Conclusion

The paper has described a design and verification methodology for the development of complex manufacturing systems logic controllers. The design models specified within the proposed methodology, that relies on concepts derived from object-oriented software engineering techniques, hold formality and expressiveness, avoiding at the same time obscurity and complexity typical of many formal approaches. Technicians and engineers involved in the machine project can understand the modeling notations without a background in formal languages, while verification can be applied easily by automating some translation and properties formalization procedures. Furthermore, design specifications can be easily implemented in a modular and efficient control software architecture, thanks to the adaptation of the modeling language to the application domain.

REFERENCES


Fig. 5
CLASS DIAGRAM OF THE SPICING UNIT CONTROL SYSTEM

Fig. 6
STATECHART OF THE MOTION SYSTEM LOGIC CONTROLLER