We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

3

Control and Plant Modeling for Manufacturing Systems using Basic Statecharts

Raimundo Moura[1] and Luiz Affonso Guedes[2]
[1]*Federal University of Piaui – UFPI*
[2]*Federal University of Rio Grande do Norte – UFRN*
Brazil

1. Introduction

Based on the IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990, 1990), **"a system can be regarded as a collection of components organized to accomplish a specific function or set of functions"**. The key point in this definition is the interaction among system components. Cassandras & Lafortune (2008) discuss systems classification, especially for Discrete Event Systems (DES). In their definition, DES are systems that have discrete state space and an event-driven dynamic, i.e., the state can only change as a result of instantaneous events occurring asynchronously over time. In this context, state-based methods such as Finite State Machines (FSM) and Petri Nets have been traditionally used to describe these systems.

The automation area uses concepts of the theory of systems to control machines and industrial processes. Considering an industrial automation process based on *Programmable Logic Controllers (PLC)*, the **sensors** are installed in the plant and generate events that represent input variables to the PLC. The **actuators** are associated with the actions produced by the PLC program and represent output variables. Industrial controller programming is currently performed by qualified technicians using one of the five languages defined by IEC-61131-3 (1993) standard and who seldom have knowledge of modern software technologies. Furthermore, controllers are often reprogrammed during plant operation life-cycle to adapt them to new requirements. As a result, ***"for practically no implemented controller does a formal description exist"*** (Bani Younis & Frey, 2006). In general, PLC are still programmed by conventional "trial-and-error" methods and there is no written documentation on these systems.

On the other hand, *software* reusability and composability have been discussed since the 80's, with the use of object-oriented methods (Boehm, 2006). In the Industrial area, the IEC-61499 (2005) standard allows reuse of application parts (function block, sub-application) in different applications. Software reuse is a complicated problem and depends not only on the means provided by the modeling language, but also on the overall application structure.

In the Computer Science area, several models guide the software development process such as the **Waterfall Model** (Royce, 1970), a sequential software development model in which development is seen as sequence of phases; the **Spiral model** (Boehm, 1988), an iterative software development model which combines elements of software design and prototype stages; and **agile methods**, which emerged in the 1990. Examples of the latter are: *Adaptive*

Source: Programmable Logic Controller, Book edited by: Luiz Affonso Guedes,
ISBN 978-953-7619-63-3, pp. 170, January 2010, INTECH, Croatia, downloaded from SCIYO.COM

Software Development, *Crystal*, *Dynamic Systems Development*, *eXtreme Programming (XP)*, *Feature Driven Development*, and *Scrum*. B. Boehm (2006) presents an overview of the best *software* engineer practices used since 1950 (decade to decade) and he identifies the historical aspects of each tendency.

In short, an application life-cycle can be divided in three phases: **Modeling - Validation - Implementation** (see Figure 1). Modeling is phase that demands more time in application lifecycle. The "Modifications" arc represents multiple iterations that can occur in *software* modeling processes. The "Reengineering" arc represents the research area, which investigates the generation of a model from legacy code. Our focus is in forward engineering, which investigate the model generation from requirements specified by users.

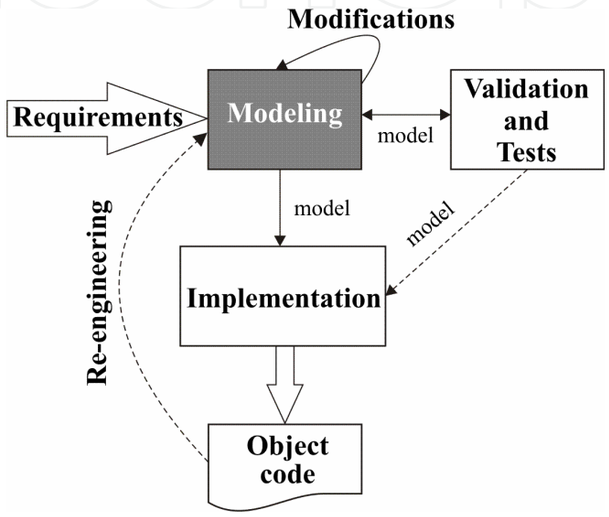

Fig. 1. Application life-cycle: overview.

In literature, there are several approaches that present methodologies, languages, and patterns for modeling industrial applications, especially for *Discrete Event Systems (DES)* (Cassandras & Lafortune, 2008). The two most common approaches are *Finite State Machines (FSM)* and *Petri nets*; both allow for formal verification of the correctness of a control system. However, despite significant research advances in recent years, these formal techniques have not been widely employed in industry (Endsley et al., 2006). We believe that such approaches are still low-level formalisms, resulting in large and unwieldy systems. The *Statecharts* formalism, described by David Harel (1987), makes the specification and design of complex DES easier. It extends conventional finite state machine with notions of **hierarchy**, **concurrency**, and **communication**.

Owing to the aforementioned problems, this work discusses a methodology for plant and control modeling and validating of the manufacturing systems that include sequential, parallel and timed operations, using a formalism based on *Statecharts*, denominated ***Basic Statechart (BSC)***. For the validation phase, simulations were executed through the execution environment developed by the *Jakarta Commons SCXML Project* (SCXML, 2006), and, as the control software model does not represent the controller itself, a translation from this model into a programming language accepted by the PLC was also carried out. In this study, *Ladder diagrams* were used because it is one of the languages defined by international IEC-61131-3 standard most widely used in industry. However, these models can be translated into any IEC-61131-3 standard language.

The remainder this work is organized as follows: Section 2 discusses about the main aspects of the *Statecharts* in modeling of automation systems and we introduce the semantic of the BSC using only characteristics relevant to the industrial area. Section 3 describes in general the methodology proposed by this contribution. In Section 4, we discuss an algorithm for translating the control model described in *Basic Statecharts* into *Ladder diagrams*, thereby enabling tests with actual PLCs. In Section 5, one typical example of application in the manufacturing area is discussed as case study to illustrate our ideas. In the last section, we conclude with a discussion about future projects.

2. Basic statecharts

Automata-based methods have been widely used to model DES, especially by the *Supervisory Control Theory* (Ramadge & Wonham, 1989). Automata represent mathematical abstractions that explicitly enumerate all the states of the system. To construct complex systems, the Automata are formally composed through systematic operations such as product and parallel composition. Moreover, they facilitate the analysis of system properties related to the validation and verification processes. However, the main drawback of the approach is inherent in the graphic representation of the model, due to the exponential growth of the number of states in the composition operations (Cassandras & Lafortune, 2008).

Statecharts formalism was described by David Harel in the 1980s and it extends conventional automata with notions of hierarchy, concurrency, and broadcast communication. Thus, *Statecharts* facilitate the specification and design of complex DES. Hierarchy and concurrency are represented through **OR-decomposition** and **AND-decomposition**, respectively. It is worth mentioning that *Statecharts* do not explicitly enumerate all the system states. Therefore, an implicit combination of the parallel states must be performed to obtain the real configuration of the model; that is, the real state of the system. Moreover, *Statecharts* have a compact graphic representation that can be translated into automata, according to the description in (Drusinsky & Harel, 1989).

The absence of a formal semantic of the original *Statecharts* makes the verification of these models very complex to carry out. In an attempt to minimize this problem, several *Statechart* variants were defined. Michael von der Beeck (1994) makes a comparison between 20 variants, and discusses a number of problems related to the original *Statecharts*. In addition, the broadcast communication of the *Statecharts* allows a triggered event in one state to affect another state that has no dependent relation with the former. Another drawback of the original *Statecharts* is that they allow interlevel transitions without imposing any constraints, a situation that can generate unstructured models.

To incorporate the advantages of the original *Statecharts* and to avoid the aforementioned problems, we propose a formalism to model DES based on *UML/Statechart* diagrams, but with a more limited syntax and semantic, denominated ***Basic Statechart (BSC)***.

The *Basic Statecharts* use the syntax of *UML/Statecharts* with some variations; for example: i) absence of history connectors; ii) inclusion of input/output data channels to allow explicit communication between the components and to avoid broadcast messages in the system; and iii) the transitions are represented by the expression "*[condition]/action*", where the conditions are composed using variables, data channels and the logical operators AND, OR and NOT; and, the actions allow one to change the value of these variables. The semantic of *Basic Statecharts* is more restrictive than that of *UML/Statecharts* to avoid conflict and

inconsistency in model evolution. We believe that this semantic is more appropriate for modeling industrial systems.

A **BSC** is composed of a collection of components and a **BSC component** is a structure used to model the behavior of a system element. A component can contain states, input/output channels, internal variables, and other components, which can be called subcomponents. A **data channel** is a resource used to communicate between system components. The **input data channels** are implicitly associated with internal variables and thus their values are maintained during the entire execution cycle. They can be used to change the value of guard condition from the component or external entity, such as control software or a simulation environment. The **output data channels** are also associated with internal variables; however, their values are updated only at the end of the execution cycle. They are used to publish the status of internal elements from one component to another.

The conceptual model describing the relationship between the elements that make up a BSC diagram is shown in Figure 2.

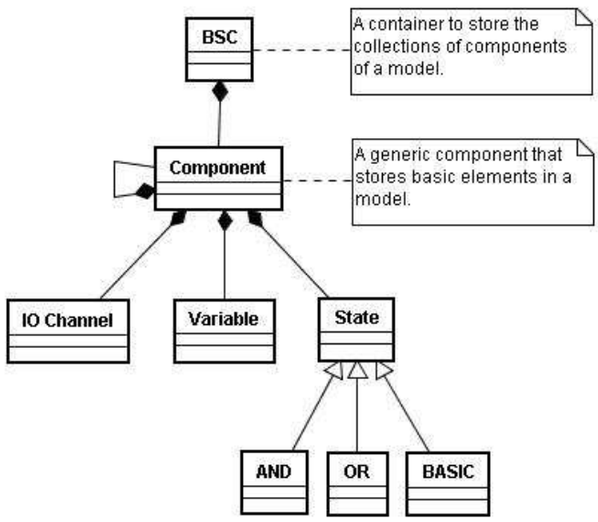

Fig. 2. Basic Statecharts: conceptual model.

The evolution of the BSC dynamic behavior is performed by sequential steps, called the *execution cycle* or *macrostep*. One constraint that is ensured by the BSC is that ***a component composed of basic states can only trigger one transition in each execution cycle (macrostep)***. As with original *Statecharts*, each macrostep in BSC can be divided into several microsteps; however, the actions performed when one transition is triggered only update the variables defined in the component data area. Moreover, the BSC run accordance with definition order of the components. Thus, in an execution cycle only one component can affect the components subsequently defined in the model. This point represents a difference between the proposed approach and the Harel diagrams specified by UML. *Basic Statecharts* make the definition of validation techniques more practical, because their syntax and semantic are more constrained than those of the original *Statecharts*.

A macrostep of a BSC execution is finished when all the components have been analyzed. The BSC communication mechanism follows a *publish/subscribe pattern*: the variables associated to output channels are published in a global area, and the variables associated to input channels are consumers of these data. It is important to note that a component can be

both publisher and subscriber of a same data item. However, the published value in one step is only consumed in the next step. It is also valid for different components. Moreover, one published value can be consumed by several components in a same step, but the value of all components is guaranteed to be the same.

3. Plant and control: modeling and validation

In industrial applications, normally the controller software is verified in conjunction with a model of the plant in which it operates. So, it is necessary to obtain an accurate model to maintain fidelity with the real plant (relation one-to-one).

3.1 Plant: modeling

For plant modeling, our methodology is based on the *hybrid approach - bottom-up and top-down*. More specifically, it proposes to model the basic elements, grouping them into larger structures. This process is repeated until it generates the correct model of application. The methodology consists of three phases described as follows:

1. Modeling the basic application elements or using models already defined in a component repository;
2. Decomposing the basic states in substates, if necessary;
3. Representing all automation plant components as parallel states;

Phases 1 and 2 consist of modeling and refinements of the basic elements which compose the application. They can be run several times as an iterative process. In each iteration, we work with components which are more and more complex. Further, these components can be grouped in a repository. The third phase determines that all application components must be executed at the same time, in a parallel way, where the communication between them is made by input/output channels.

We will present how our methodology works below.

3.1.1 Basic components: patterns

For automation systems, many components follow an *On/Off pattern*, for example, valves and sensors. Figure 3-a shows the dynamic behavior of this pattern, which can be in states: "Off" or "On", and two transitions to change from state: "[g1]" from "Off" to "On" and "[g2]" from state "On" to "Off". Other components require adjustment in modeling to include new characteristics. For example: a temporary state (Wait) between the states "On" and "Off" (see Figure 3-b).

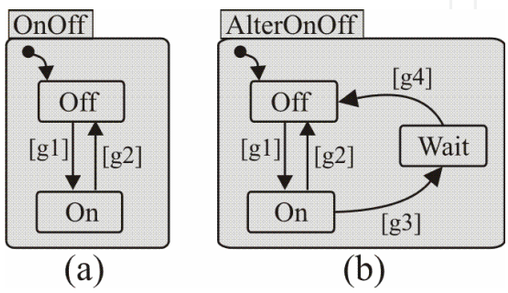

Fig. 3. On/Off patterns: basic model.

3.1.2 Cylinder component

In the manufacturing field, one of the most common components is the pneumatic cylinder that can be composed of more simple components (valves, arms and sensors) and can have displacement sensors/end-position initiators.

Figure 4 depicts a single-action cylinder with advancing controlled by the valve, return carried through springs, and one end-position sensor which is triggered when the cylinder arm gets the full advance. The generic notation "*[g]/A*" in a transition means that: when a guard condition g is true, the action A will be executed. Therefore, if an action in a component X1 updating one variable used in guard condition of a component X2, then we will say that: *X2 depends on component X1*. According to figure, the transition "*[ch]/v1=1*" and "*[v1]/tm1=1*" indicate that: the cylinder arm depends on the valve, i.e., the arm advances while the valve remains open. When the valve is closed through the action "ch=0", the cylinder arm gets "Returned", in function of transitions "*[¬v1]/tm1=0*" or "*[¬v1]/v2=0*".

The cylinder arm has the following behavior: when the variable v1 gets true, the arm gets to "Advancing" in a specified time, which depends on technical characteristics and it is represented by "*" in the figure. If the valve is closed before this specified time (event *tm1.tm*), the cylinder arm gets to "Returned" and nothing happens to the sensor. If the event *tm1.tm* occurs, then the arm gets to "Advanced" and the active state of the sensor passes from "False" to "True", implicitly. So, when the valve is closed, the arm gets "Returned" and the sensor passes from "True" to "False".

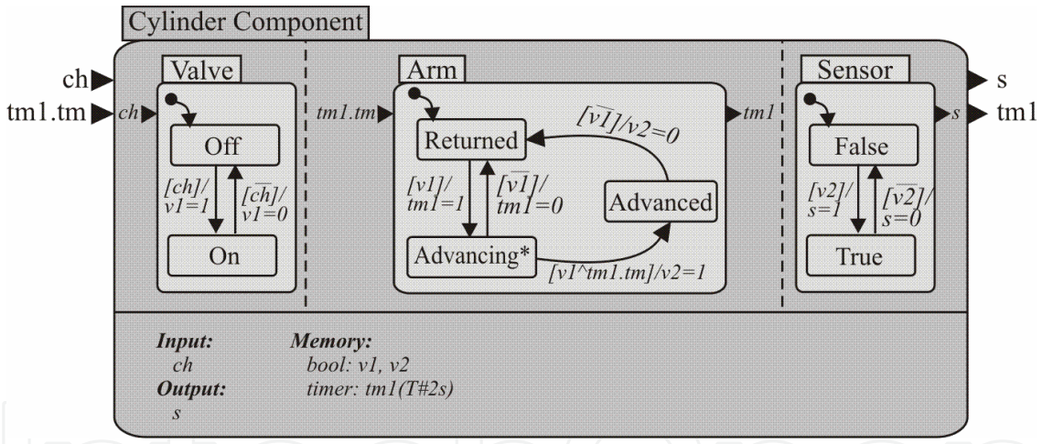

Fig. 4. Single-action cylinder: basic model.

The scenario that describes the desired operation of the cylinder is very simple: one external event allows the opening of the valve when the channel gets equal 1 (*ch=1*); then the transition "*[ch]/v1=1*" is run; and after the sensor detects the total advance of the cylinder-arm, the valve must be closed (data channel equals 0, i.e., *ch=0*); then the transition "*[¬ch]/v1=0*" is run. The events to open/close the valve represent the control police that is run by the model and define the dynamic cylinder.

3.2 Control software: modeling

In the manufacturing area, actuator components are controlled through events that are triggered by devices, such as *buttons, sensors,* and *timers*, which are defined in the control model using temporary variables. The controller is modeled through the composition of components; i.e., complex models are constructed from simpler models. The basic

components are: a) **actuators** that are modeled using components with two states: OFF and ON; b) **timers** that are modeled using components with three states: OFF, START and ON - the state "START" starts the timer and the transition "*[tm1.tm]*" from state "START" to state "ON" triggers the end of the timer event; and c) **variables** that are associated with sensors and temporary elements. Figure 5 shows the basic model for these elements. In this figure, g1, g2, and g3 are guard conditions. The data model area in Figure 5-c defines two Boolean variables (s1 and s2), both with the "false" value, using the syntax of the SCXML specification that was implemented by the *Jakarta Project Commons SCXML* (SCXML, 2006). This project provides a generic event-driven state machine based on the execution environment, borrowing the semantics defined by SCXML, which represents the *Statechart* diagrams by a XML file.

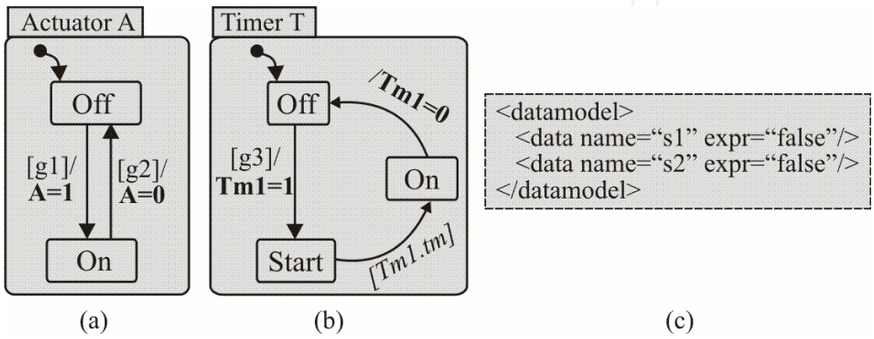

Fig. 5. Actuators: basic model.

Operational requirements of the actuators are inserted into the model as transitions between the states, in the following general form: "*[guard condition] / action*". The guard conditions are Boolean expressions composed of data channel and internal variables, interconnected through logical connectors ¬ (negation), || (disjunction) and & (conjunction). The actions can be, for example, an assignment statement to set a value in the variable and/or data channel. Therefore, operational requirements are constraints in the model to implement dependencies and/or interactions between the components. Such constraints allow us to define sequential and parallel behavior in the model; this will be described in the next subsections.

3.2.1 Sequential operation

Consider a plant composed of two actuators (A_i and A_j) that run sequentially one after the other, i.e., A_i;A_j. This sequence is run continuously in a cyclical way until user intervention. The sequential behavior of A_i and A_j is obtained through the execution of actions in actuator A_i, which generates internal event triggers in actuator A_j. In general, an action in an actuator can cause state changes in other actuators.

Figure 6 shows the *Basic Statechart* diagram for modeling the sequential behavior between actuators A_i and A_j discussed above. In this figure, ch_1, ch_2 and ch_3 are input data channels; ch_1, A_i and A_j are output data channels, and "ev" is an internal variable. Note that a same channel can be both input and output channel in a model. This is possible because the channels are associated implicitly with internal variables. These elements are used to generate the desired model behavior. In this case, the "ev" variable is used as an action by actuator A_i, which indicates the end of its actuation. It is perceived by actuator A_j, which starts its operation, generating the sequential behavior between them. Note that the data

model area is not represented in the figure. At the end of A_j actuation, data channel ch_1 is updated, generating the cyclical behavior of the model. In its initial configuration, all the actuators of the model are set to "Off". The system starts its operation when data channel ch_1 is equal to 1 (Boolean value "true"), a situation that can be simulated when the operator pushes a "start" button on the Interface Human-Machine (IHM), for example.

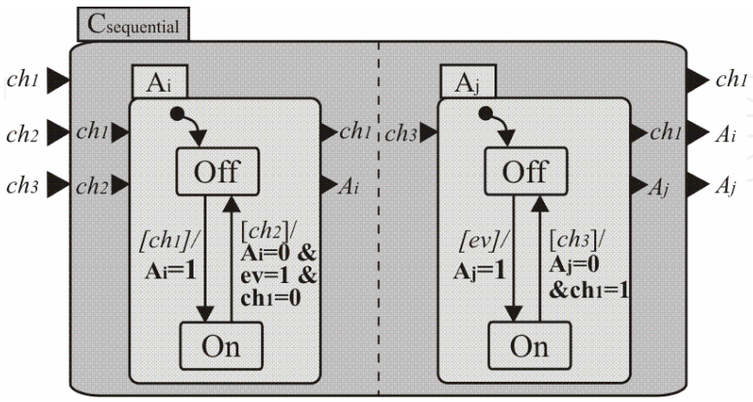

Fig. 6. Control model: sequential operation.

3.2.2 Parallel operation

Parallelism, an inherent characteristic of original *Statecharts*, is accomplished through AND-decomposition. However, the component synchronism demands additional mechanisms. Consider a plant composed of three actuators (A_i, A_j and A_k), where A_i and A_j run in parallel, but A_k can only run after the execution of the two first components, i.e., $(A_i||A_j);A_k$. This sequence is run continuously in a cyclical way until operator intervention. The parallel behavior of A_i and A_j is obtained naturally; however, internal variables must be used to generate internal event triggers in actuator A_k to indicate the end of execution in other actuators. Thus, A_k must wait for these updates to start its operation. After the A_k run, these internal variables must be updated to allow the execution of a new cycle in the system. Figure 7 shows the *Basic Statechart* diagram for modeling the parallel behavior between the aforementioned actuators. In this figure, $ch_i(i = 1..5)$ are input data channels, A_i, A_j and A_k are output data channels, ev_i and ev_j are internal variables. These elements are used to generate the desired application behavior. In this case, the variable ev_i is updated as an action by actuator A_i, indicating the end of its actuation, and the variable ev_j is updated to indicate the end of A_j actuation. These updates are perceived by actuator A_k, which starts its operation, generating the synchronism between them. At the end of A_k actuation, the ev_i and ev_j must be "reset" to generate the cyclical behavior of the model. In its initial configuration, the model must have all actuators set to "Off".

3.2.3 Timed operation

Timers and counters are quite common in industrial applications; for example: i) an actuator must execute for a specific time; ii) an actuator must execute only after a specific time; iii) the system must execute k times before triggering an alarm; and so on. Timers and counters are modeled through basic components and their current values can be used to set the guard conditions of the transitions in BSC. Furthermore, they can be started and/or reset by some action of the model.

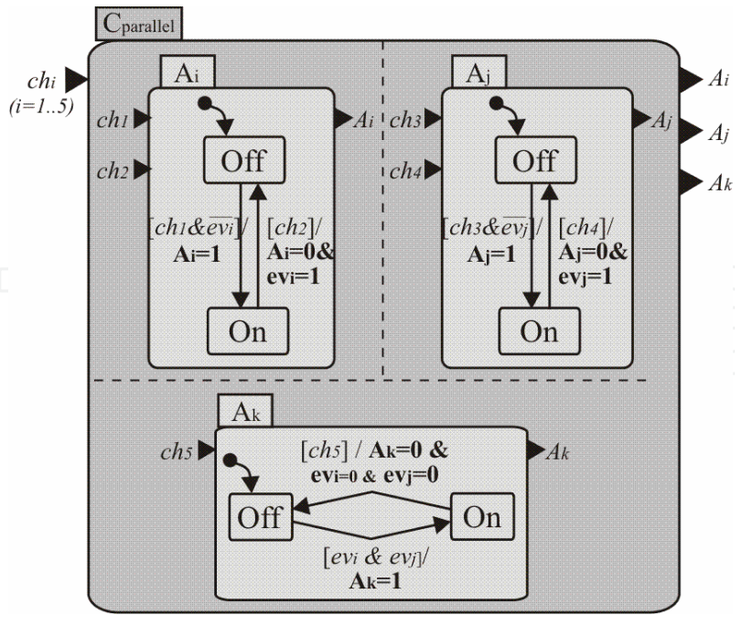

Fig. 7. Control model: parallel operation.

Timers are controlled by a global real-time clock that executes in parallel to the system model, and they are updated only at the beginning of each execution cycle. Thus, when a timer is enabled in a component, the timing process is initiated in the next execution cycle. When the timer reaches or surpasses its specified limit, an internal variable tm is made true (tm = true) to indicate end of timing. In the timer, creating must define the time limit value in time units.

Consider a plant composed of an actuator A_i and a timer T_k, where A_i must act for t seconds before turning off. Figure 8 shows the *Basic Statechart* for modeling the temporal behavior of actuator A_i, controlled by timer T_k. In this figure, ch_1 and ch_2 are input data channels used to start the operation of actuator A_i and of timer T_k, respectively, and tk.tm is an input data channel used to indicate the timeout of T_k. It is important to mention that the timers are updated as a global action of the model, and the timer T_k is started when action tk = 1 is executed.

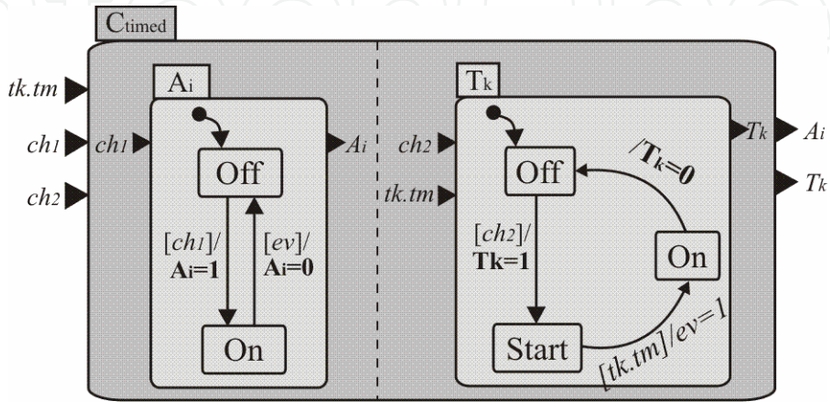

Fig. 8. Control model: timed operation.

The guard condition "ev" used to turn off actuator A_i becomes true when timer T_k reaches or surpasses the specified limit (condition *tk.tm*). Thus, the constraint that defines that actuator A_i must execute for a specific time is ensured.

3.3 Control software: validation

The approach for modeling the control software discussed in Section 3.2 maintains the description and specification aspects built into the *Basic Statechart* model. Transitions, guard conditions, and implicit actions are used to describe system constraints. Thus, the approach allows us to analyze some controller properties using the reachability tree of the formal model. Moreover, simulated environments can be used to validate the control model along with the plant model.

The reachability tree of the model allows us to analyze a number of properties, such as: i) **reinitiability** – for each cfg_i state configuration reached from the initial cfg_0 configuration, is it possible to return to cfg_0 by a sequence of events? ii) **vivacity** – does the controller act in all of the components in the model? iii) **deadlock** – is there a cfg_i state configuration in which progress cannot be made because no transition can be triggered?

Masiero et al. (1994) propose an algorithm to create a reachability tree for *Statecharts*. Here, we briefly discuss an adaptation of this algorithm to analyze the aforementioned structural properties. This algorithm was implemented using Java language and the SCXML execution environment, with the following modifications:

- The set that contains all possible transitions for a given configuration includes only the transitions with events controlled by an external agent, and with timed events triggered automatically by the components.
- To obtain a new configuration of the model by triggering a transition, the internal variables are implicitly updated and, therefore, can trigger other transitions automatically in the model. This characteristic decreases the number of states produced in the reachability tree.
- The part of the algorithm that describes the history connectors is completely excluded, because *Basic Statecharts* do not include such characteristics.

The use of this algorithm allows a formal analysis of system behavior (control + plant) to verify and validate a number of properties. It is important to note that a plant model is required, and it may be represented in a given formalism; for example, automata, Petri net or *Statecharts*. Moura et al. (2008) propose a systematic procedure for modeling complex plants using *Statecharts* and discuss some aspects of control modeling. However, they presented only a descriptive view of that process.

In this work, we chose *Basic Statecharts* to model plant behavior, without losing generality. Therefore, the system (control + plant) can be described as parallel composition between the controller and plant. The main advantage of this approach is that sensor and actuator characteristics become internal events of the system. Thus, the intrinsic properties of the system, such as **reachability**, **deadlock**, and **reinitiability** become intrinsic and extrinsic properties of the controller.

Another advantage of this approach owes to the fact that it maintains controller and plant functionality explicitly separated. Here, unlike other approaches, such as the R & W approach (Supervisory control), the controller synthesis produces more compact models. In the next section we present an algorithm for translating the control model described in Basic *Statecharts* into one PLC language (in this case, *Ladder diagram*).

4. Control software: implementation

Given that the control model does not represent the controller software itself, the translation from this model into a programming language accepted by the PLC must also be performed. *Ladder* diagrams were used because it is one of the languages defined by international IEC-61131-3 standard most widely used in industry. The translation is performed systematically by a method that analyzes one component at a time, according to its type (**actuator** or **timer**).

The states ("OFF" and "ON") in the actuators are represented in the *Ladder* through auxiliary contacts (*flip-flop Reset* and *flip-flop Set*), respectively. Each control model transition results in a "rung" of the *Ladder*, as follows: the source state must be added to the condition, and the target state represents the action that must be executed. Let A be the generic actuator shown in Figure 5-a, where transitions "*[g1]/A=1*" and "*[g2]/A=0*" generate lines 3 and 4, respectively, of the *Ladder diagram*, as shown in Figure 9. In this figure, c1, c2, and c3 are auxiliary variables that are computed from the guard conditions of the model (i.e., g1, g2, and g3, respectively). This mapping is made because the guard conditions can be complex.

The timers were translated as follows: one "rung" to transition from the "OFF" to "START" state, which allows us to start up the timing; one "rung" to specify the timer itself, with one element that indicates the end of the specified time, which can be used in other *Ladder* lines,

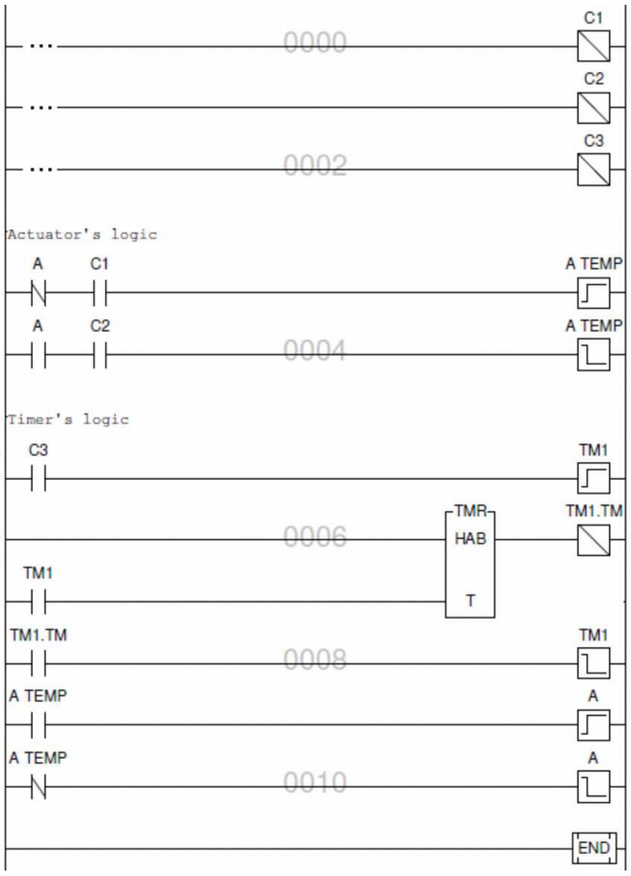

Fig. 9. Actuators: Ladder diagram.

according to the application; and another "rung" to reset the timer. The generic timer shown in Figure 5-b generates lines 5 to 8 of the *Ladder diagram* (see Figure 9). In this figure, the parameters "HAB" and "T" of the block TMR represent identifiers used to set up as follows: HAB lets it enable/disable, and T lets us define the time limit value of this block. The variables that represent the sensors and or auxiliary contacts can be freely used in the guard conditions and actions of the *Ladder* code, according to the transitions of the model. However, as the guard conditions of the transitions (in each *Ladder* line) must be guaranteed by at least one *PLC-scan cycle*, all conditions must be evaluated and stored in auxiliary variables at the beginning of each *PLC-scan cycle* (see lines 0, 1 and 2 in Figure 9).
Moreover, it is important to note that to avoid non-determinism in the system, the guard conditions for a same source state must be mutually exclusive. This constraint can be established during model building and the user can be notified by warning messages. But, as the conditions must be mutually exclusive to a same source state, these *Ladder* lines specifically cannot be generated in any order, because inconsistencies can occur in one *PLC-scan cycle*; for example, turning on/turning off an actuator. To avoid such inconsistencies, the temporary state of the actuators must be stored in auxiliary variables, and at the end of the cycle, these variables must be updated for the corresponding outputs (see lines 9 and 10 in the Figure 9).

Algorithm 1. Translation from the control model into a Ladder diagram

```
{Let there be n actuators, m timers, t transitions}
{Guard conditions analysis}
for i = 1 to t do
    Compute guard(i) {Guard condition of the i-th transition}
end for
{Actuator's logic}
for i = 1 to n do
    for j = 1 to T[Ai] do
        if target( j) = Ai.ON then
            AiTemp.set := source( j) AND guard( j)
        else
            AiTemp.reset := source( j) AND guard( j)
        end if
    end for
end for
{Timer's logic}
for i = 1 to m do
    Tmi.set := guard(enableTimer(Tmi))
    CreateTimer(Tmi, limit(Tmi)) {Function block: Timer}
    tmi.tm := Tmi.enable() AND Tmi.timeout()
    Tmi.reset := tmi.tm
end for
{Update actuators from temporary variables}
for i = 1 to n do
    Ai.set := AiTemp
    Ai.reset := ¬AiTemp
end for
```

The complete algorithm used to translate the control model into *Ladder code* is presented in Algorithm 1. In this algorithm, some terms have been used to facilitate the understanding, such as:

- **guard(t)** is the guard condition of the t-th transition;
- **source(t)** is the source state of the t-th transition;
- **target(t)** is the target state of the t-th transition;
- T[Ai] is number of transitions of actuator Ai;
- Ai.ON is a constant to represent the 'ON' state of actuator Ai;
- **enableTimer(Tmi)** is the transition that allows us to start up the timing of the i-th timer;
- **Tmi.limit(<value>)** is the time limit of the i-th timer;
- **Tmi.enable()** is a function to indicate if the i-th timer is enabled;
- **Tmi.timeout()** is a function to indicate when the i-th timer reaches the end of the specified time.

5. Case study: manufacturing cell

This section presents a case study that realizes a simulation of a manufacturing cell (see Figure 10-a), which is a typical example of the manufacturing sector where the devices can run in a simultaneous mode. This example is well explored in *Supervisory Control Theory* by Queiroz & Cury (2002). The problem with to these systems is the need for synchronization points between parallel blocks.

The execution flow, with a possible operation of the devices for this system, is shown in Figure 10-b. It is interesting to note that the four device actuators can run simultaneously and that the table must be run only after the execution of these devices. Thus, a synchronization point between devices and the table must be created to enable proper system operation.

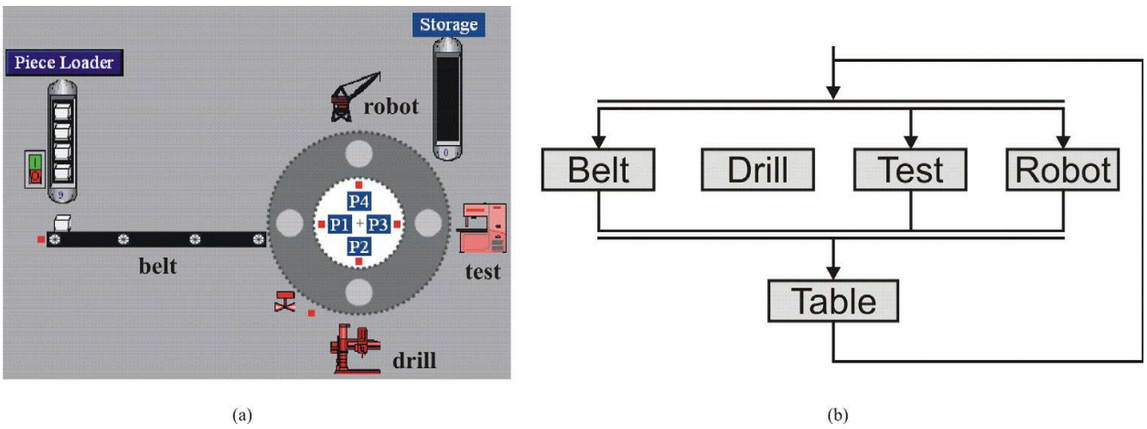

Fig. 10. Manufacturing cell: simulation environment.

Consider the run scenario described below:

- **BELT**: If there is a piece in the input buffer (initial position of the belt) and none in position P1, the belt must be turned on; later, when the piece is at position P1 the belt must be turned off. The ***if ... then*** *clauses* of this specification are:
  - **If** inputbuffer & ¬P1 **then** BeltOn;

  - **If** P1 **then** BeltOff;
- **DRILL**: If there is a piece in position P2, the drill and a timer component timerT1 must be turned on; at the end of timeout, the drill must be turned off. The ***if ... then*** *clauses* of this specification are:
  - **If** P2 **then** DrillOn & tm1On;
  - **If** tm1.tm **then** DrillOff;
- **TEST**: If there is a piece in position P3, the test and a timer component timerT2 must be turned on; at the end of timeout, the test must be turned off. The ***if ... then*** *clauses* of this specification are:
  - **If** P3 **then** TestOn & tm2On;
  - **If** tm2.tm **then** TestOff;
- **ROBOT**: The robot removes a piece from position P4, and stores it. If there is a piece in position P4, the robot and a timer component timerT3 must be turned on; at the end of timeout, the robot must be turned off. The ***if ... then*** *clauses* of this specification are:
  - **If** P4 **then** RobotOn & tm3On;
  - **If** tm3.tm **then** RobotOff;
- **TABLE**: The table rotation is controlled by the single-action cylinder and the total advance of the cylinder arm generates a 90 degree turn. Thus, after the execution of the four devices, the cylinder must be activated to obtain a new system configuration. The return of the cylinder-arm should occur when the sensor detects the total advance of the cylinder-arm. The ***if ... then*** *clauses* of this specification are:
  - **If** BeltEnd & DrillEnd & TestEnd & RobotEnd **then** ValveOn;
  - **If** SensorOn **then** ValveOff;

The belt model follow the *Alter On/Off pattern* (see Figure 3-b), whereas the drill, the test, and the robot models follow the *On/Off pattern* (see Figure 3-a). The table behavior is modeled through of single-action cylinder (see Figure 4). In each table position, there is one sensor for simulating piece in the place. Thus, the complete plant model is generated by representation, in parallel way, of the four devices and the cylinder, as can be shown in Figure 11.

Other constraints imposed on the model are:

1. Each device must execute only once before a table rotation;
2. If in a configuration there is no piece in the input buffer or in positions P2, P3, and P4, then the belt, the drill, the test, and the robot must not be turned on;
3. The table rotation must only be performed if there is at least one piece in positions P1, P2, or P3.

The inclusion of these constraints in the controller model is carried out by determining new transitions between states and/or changes in the guard conditions of the existing transitions. Initially, to create the control model for this case study, extra variables must be included to ensure synchronism between the devices and, therefore, the constraint imposed on table rotation, i.e., the table cannot rotate while the devices are running. In this case, the variables E1, E2, E3, and E4 indicate the "*end-of-operation*" of the belt, drill, test, and robot, respectively. These variables must be set to "true" for each of the devices. According to cell operation, the table must only be rotated when all of devices have concluded their operations, i.e., when the variables Ei = true (i = 1,...,4). After the table rotates 90 degrees, these variables must be reset to allow new operations in the system. These variables are also used in the transitions to turning on/turning off the actuators; for example, the drill must only be turned on if the E2 control variable is equal to "false".

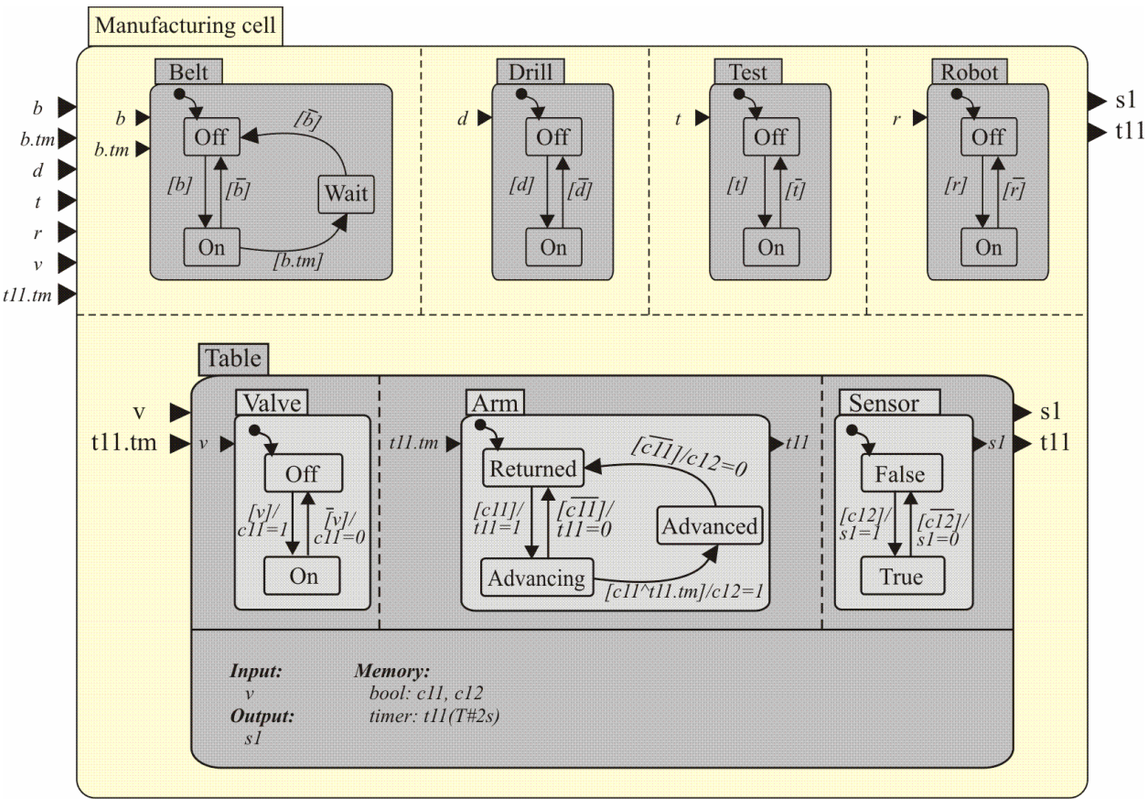

Fig. 11. Manufacturing cell: plant model.

Extra transitions to ensure constraints 2 and 3 must be included in the model. For example, if there is no piece in position P2, then the drill must not be turned on, but the E2 variable must be set to "true" to indicate end-of-operation of the phase. Similar ideas are applied to other actuator devices. In the table model, if there is no piece in positions P1, P2 or P3, then the table must not rotate (constraint 3); however, variables E1, E2, E3, and E4 must be set to "false" to allow new operations in the devices. Thus, if there is no piece in the manufacturing cell, the model will continually alternate the value of E1,...,E4 between

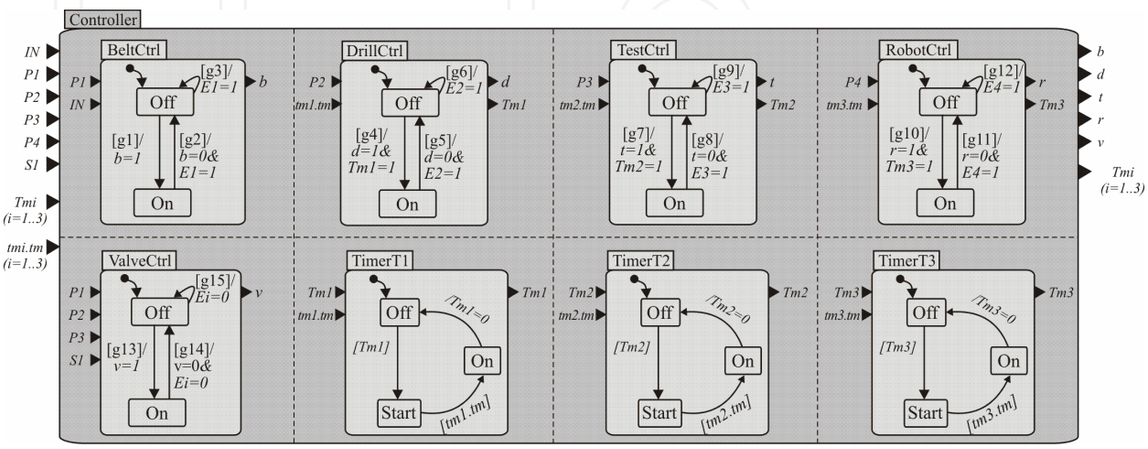

Fig. 12. Manufacturing cell: control plant.

"false" and "true". The complete BSC model of the controller software is shown in Figure 12. Guard conditions g1, g2,...,g15 are presented in Table 1, where the variable IN indicates the presence or absence of a component in the input buffer. Note that the data area is not represented in the figure, but the IO channels can be easily identified; Ei (i = 1,...,4) are internal variables, and P1,...,P4, IN, S1 represent sensors installed in the plant.

g1	¬P1 & ¬E1 & IN
g2	P1
g3	¬P1 & ¬E1 & ¬IN
g4	P2 & ¬E2
g5	tm1.tm
g6	¬P2 & ¬E2
g7	P3 & ¬E3
g8	tm2.tm
g9	¬P3 & ¬E3
g10	P4 & ¬E4
g11	tm3.tm
g12	¬P4 & ¬E4
g13	E1&E2&E3&E4 & (P1 || P2 || P3)
g14	S1
g15	E1&E2&E3&E4 & ¬P1 & ¬P2 & ¬P3

Table 1. Controller: guard conditions.

This example is composed of the belt with three possible states, three devices with two states each, and one cylinder linked to the table, which also has three states. The model with no control has 72 states, i.e., 3 x 2 x 2 x 2 x 3 = 72 distinct configurations, and the controlled model (control + plant) has 210 different states, in function of three timers included in the control model for simulating the processes of drilling, testing and moving the piece to storage. However, these 210 configurations act only in 26 distinct configurations, where: i) 24 possibilities of actuation of devices: 3 x 2 x 2 x 2 = 24 with the table in position stop (cylinder in configuration *[Off, Returned, False]*); and ii) 2 possibilities for rotating the table with the four devices in state Off. The reachability tree analysis has shown that the model ensures the properties of ***reinitiability***, ***vivacity***, and that there is no ***deadlock***. But, this analysis is out of scope of this work.

6. Conclusion

In this work we presented a methodology for systematizing the process of plant and control modeling of manufacturing systems. Our proposal uses a formalism based on *Statecharts* diagrams, called **Basic Statecharts (BSC)**. The plant modeling has three phases which can be executed as many times as necessary. In general, this methodology represents a hybrid approach - *bottom-up and top-down*, allowing components reuse and keeping a one-to-one relation between plant and model (i.e., it is faithful to the actual system). The control model is generated also using **Basic Statecharts**. Thus, the main contributions of this work are the following:

- A methodology to model plants and industrial control logics using **Basic Statecharts**;

- A procedure to integrate plant and control models in order to analyze and/or validate several structural proprieties of the modeled system, such as **deadlock absence**, **vivacity**, and **reinitiability**. This is very important in the project phase of every industrial controller;
- An algorithm to translate the control logics described in **Basic Statecharts** into *Ladder diagrams*.

One typical example of the manufacturing application was described as a case study to illustrate our proposal.

A prototype using Java language is currently being developed to create and simulate models generated by our methodology. The aim is to test how much easier and natural the creation of industrial applications will become, as well as to produce more "user-friendly" documentation for the designers, giving more autonomy to the development and maintenance teams.

7. References

Bani Younis, M. & Frey, G. (2006). UML-Based Approach for the Reengineering of PLC Programs, *in Proceedings of 32nd Annual Conference of the IEEE Industrial Electronics Society (IECON'06)*, pp. 3691–3696, Paris, France, November, 2006.

Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement, *Computer*, Vol. 21, Issue 5 (May 1988), pp. 61 – 72, ISSN:0018-9162

Boehm, B. (2006). A View of 20th and 21st Century Software Engineering, *in Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pp. 12–29, New York, NY, USA: ACM Press, 2006.

Cassandras, C. & Lafortune, S. (2008). *Introduction to Discrete Event Systems - Second Edition*, Springer Science, ISBN-13: 978-0-387-33332-8, New York, (USA).

Drusinsky, D. & Harel, D. (1989). Using Statecharts for Hardware Description and Synthesis, *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 7, pp. 798–807.

Endsley, E.; Almeida, E. & Tilbury, D. (2006). Modular Finite State Machines: Development and Application to Reconfigurable Manufacturing Cell Controller Generation, *Control Engineering Practice*, Vol. 14, No. 10 (October 2006), pp. 1127–1142.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, Vol. 8, No. 3 (June 1987), pp. 231–274.

IEC-61131-3 (1993). International Eletrotechnical Commission. Programmable Controllers Part 3, Programming Languages.

IEC-61499-1 (2005). International Eletrotechnical Commission. Functions Blocks Part 1, Architecture, Geneva: IEC, 2005.

IEEE Std 610.12-1990 (1990). Standard Glossary of Software Engineering Terminology, http://ieeexplore.ieee.org/ISOL/standardstoc.jsp?punumber=2238.

Masiero, P.; Maldonado, J. & Boaventura, I. (1994). A reachability Tree for Statecharts and Analysis of Some Properties, *Information and Software Technology*, Vol. 36, No. 10, pp. 615–624.

Moura, R.; Couto, F. & Guedes, L. (2008). Control and Plant Modeling for Manufacturing Systems using Statecharts, *in IEEE International Symposium on Industrial Electronics (ISIE 2008)*, Cambridge, UK, July 2008, pp. 1831–1836.

Queiroz, M. & Cury, J. (2002). Synthesis and Implementation of Local Modular Supervisory Control for a Manufacturing Cell, *In Proceedings of the 6th International Workshop on Discrete Event Systems*, IEEE Computer Society, pp. 377-382.

Ramadge, P. & Wonham, W. (1989). The Control of Discrete Event Systems, *in Proceedings of the IEEE*, Vol. 77(January, 1989), pp. 81–98.

Royce, W. W. (1970). Managing the Development of Large Software Systems, *in Proceedings. of IEEE WESCON*, pp. 1–9.

SCXML (2006). The Jakarta Project Commons SCXML, http://jakarta.apache.org/commons/scxml/.

Von der Beeck, M. (1994). A Comparison of Statecharts Variants, *in ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*. London, UK: Springer-Verlag, pp. 128–148.

Programmable Logic Controller

Edited by Luiz Affonso Guedes

ISBN 978-953-7619-63-3
Hard cover, 170 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

Despite the great technological advancement experienced in recent years, Programmable Logic Controllers (PLC) are still used in many applications from the real world and still play a central role in infrastructure of industrial automation. PLC operate in the factory-floor level and are responsible typically for implementing logical control, regulatory control strategies, such as PID and fuzzy-based algorithms, and safety logics. Usually PLC are interconnected with the supervision level through communication network, such as Ethernet networks, in order to work in an integrated form. In this context, this book was written by professionals that work and research in automation area and it has two major objectives. The first objective is present some advances in methodologies and techniques for development of industrial programs based on PLC. The second objective is present some PLC-based real applications from various areas such as manufacturing system, robotics, power system, communication system, and education.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Raimundo Moura and Luiz Affonso Guedes (2010). Control and Plant Modeling for Manufacturing Systems using Basic Statecharts, Programmable Logic Controller, Luiz Affonso Guedes (Ed.), ISBN: 978-953-7619-63-3, InTech, Available from: http://www.intechopen.com/books/programmable-logic-controller/control-and-plant-modeling-for-manufacturing-systems-using-basic-statecharts

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen