Basic Statechart: A Formalism to Model Industrial Systems

Raimundo Santos Moura and Luiz Affonso Guedes

Abstract— **Despite the many studies that have attempted to develop "friendly" methods for industrial controller programming, they are still programmed by conventional "trial-and-error" methods and there is little written documentation on these systems. Accordingly, this paper proposes a methodology to model control programs for manufacturing systems that include sequential, parallel and timed operations, using a formalism based on *Statechart diagrams*, denominated Basic Statechart (BSC). One typical example of application in the manufacturing area is presented as case study to illustrate the proposed methodology.**

I. Introduction

The automation area uses concepts of system theory 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. In generic applications, there can be k sensors and n actuators, in addition to m timers and counting devices implemented in the PLCs through auxiliary memories. The values k, n, and m are non-negative integers. Thus, software-based technologies play an important role.

Industrial controller programming is currently performed by qualified technicians using one of the five languages defined by IEC-61131-3 [1] standard and who seldom have knowledge of modern software technologies. Furthermore, controllers are often reprogrammed during the plant operation life-cycle to adapt them to new requirements. As a result, *"for practically no implemented controller does a formal description exist"* [2]. In general, PLCs are still programmed by conventional "trial-and-error" methods and there is no written documentation on these systems.

On the other hand, in the academic field there are several proposals for the systematic synthesis of controllers, such as *Supervisory Control Theory* developed by Ramadge and Wonham in the 1980's[3]. This theory is based on formal aspects of that automata for automate controller synthesis. However, despite advances in this area, the scientific community has not managed to convince industrial engineers to migrate from traditional programming methods to the new approaches. Skoldstam et. al. [4] indicate two main causes for the failure of Supervisory Control Theory to implement real world controllers: i) the discrepancy between reality based on the signal and the event-based automata framework, and ii) the lack of a compact representation of large models. Gourcuff et al. [5] present three reasons for not adopting formal methods in industry: i) formal specification of properties in temporal logic or in the form of timed automata is an extremely difficult task for most engineers; ii) model-checkers provide, in the case of negative proof, counterexamples that are difficult to interpret; and iii) PLC manufacturers do not provide commercial software capable of automatically translating PLC programs into formal models. Therefore, the use of higher-level methodologies could be useful in developing controller programming. However, it poses a great challenge to industry. In addition, to remain focused on the formal aspect is very important for analysis, validation, and verification of the models, but it must be abstracted for the designers to fully understand the system.

R. Moura is with the Department of Informatics and Statistics / UFPI; (e-mail: rmoura@dca.ufrn.br)

L. Affonso is with the Department of Computer Engineering and Automation / UFRN; (e-mail: affonso@dca.ufrn.br)

In the Computer Science area, several models guide the software development process, such as the **Waterfall Model** [6], a sequential software development model in which development is seen as a sequence of phases; the **Spiral model** [7], an iterative software development model that combines elements of software design and prototype stages; and **agile methods**, which emerged in the 1990s. Examples of the latter are: *Adaptive Software Development, Crystal, Dynamic Systems Development, eXtreme Programming (XP), Feature Driven Development, and Scrum*. Boehm, in [8], 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 several software development models, the life-cycle used can be divided into three phases: **Modeling - Validation - Implementation** (see Fig. 1). The "Modifications" arc represents multiple iterations that can occur in software modeling processes. The "Reengineering" arc represents the research area that investigates the generation of a model from the legacy code. Our focus is on forward engineering, which investigates model generation from user-specified requirements.

Owing to the aforementioned problems, we propose a method for modeling and validating the control software for manufacturing systems that involve sequential, parallel and timed operations. For the modeling phase, we use the *Basic Statecharts (BSC)*, which represent a reduced variant of the *Statecharts formalism* [9] described by David Harel in the 1980s. For the validation phase, simulations have been used through the execution environment developed by *the Jakarta Commons SCXML Project* [10]. Moura & Guedes, in [11], describe a general schema of the industrial automation process, using the SCXML environment. As the control software model does not represent the controller itself, a translation from this model to a programming language accepted by PLC has been also carried out. In our studies, *Ladder diagrams* are used because it is one of the languages defined by international IEC-61131-3 standard most widely used in industry. However,

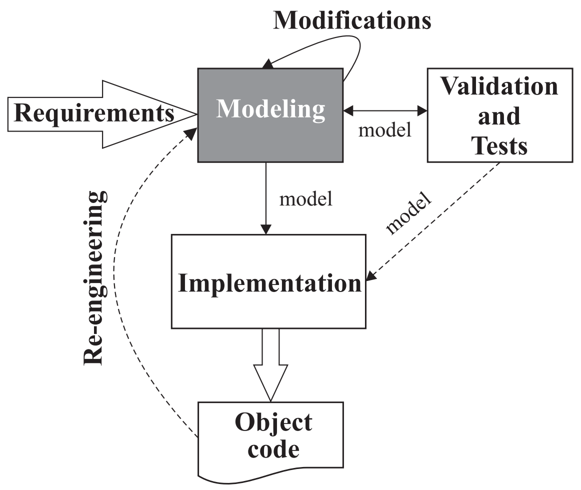

Fig. 1. Application life-cycle: overview

these models can be translated to any IEC-61131-3 standard language.

In this paper specifically we will discuss the main aspects of the *Basic Statecharts*. It is organized as follows: Section II presents the formal definition of *Basic Statechart* formalism. Section III discuss the main aspects involved in modeling of industrial automation systems, based on the PLC. In Section IV, a case study is presented showing the modeling process of the controller for a manufacturing cell. Finally, in Section V we present the conclusions and future works.

II. Basic Statechart

Although there are several formalisms for *Discrete Event Systems (DES)* modeling, we decide to propose a formalism based on *Statecharts*, named as *Basic Statecharts*, for three main reasons: i) to allow validation of the models; ii) to be a high-level formalism, making controller design easier for industrial engineers; and iii) to use *Statecharts*' properties to model complex systems, which is very hard in approaches based on automata or Petri nets.

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"*. The *conditions* are composed using variables, data channels and the logical operators AND, OR and NOT. The *actions* allow one to change the value of these variables. The semantic of *Basic Statechart* 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.

The formalism is formally defined as follows:

Definition 1: A Basic Statechart BSC is a 7-tuple

$$BSC = (S, p, s_0, \Psi, \delta, T, \mathcal{M})$$

where

- S is a finite set of states and parallel states;
- $p : S \rightarrow 2^S$ defines a tree structure with substate $descendants$, i.e., direct children or children of descendants. Functions p^* and p^+ are extensions of p, defined by

$$p^*(x) = \bigcup_{i \geq 0} p^i(x)$$

and

$$p^+(x) = \bigcup_{i \geq 1} p^i(x)$$

where $p^i(x)$ is the set of direct descendants of state "x" in hierarchy i. By the definition, $p^0(x) = \{x\}$. Therefore, $p^*(x)$ is the set of states containing "x" and all of the descendants in the hierarchy, and $p^+(x)$ is the set containing all of the descendants of "x", except itself;
- $s_0 \in S$ is the initial state (root of the tree of the *Basic Statechart*), where $\forall x \in S, s_0 \notin p(x)$;
- $\Psi : S \rightarrow \{BASIC, AND, OR\}$ is a function that describes the type of state decomposition. For example, $x \in S$ is "BASIC" if $p(x) = \emptyset$, i.e., a state without substates;
- $\delta : S \rightarrow 2^S$ is a *default function* that defines a set of initial states for state $x \in S$, if x is not "BASIC". If x is an "OR" state, it has only one *default state*; on the other hand, if x is an "AND" state, then $\delta(x) = \delta(p(x))$;
- T is a finite set of transitions (X, c, A_i, Y), where $X \in S$ is the source state; c is a guard condition used to enable/disable the transition. It is described by one combination of variables and/or data defined in the data model area; A_i is a set of actions that must be executed when the transition is taken; $Y \in S$ is the target state. Informally, a transition can be represented by the expression *"[c]/a"*, where these elements are optional;
- $\mathcal{M} : \mathcal{S} \rightarrow \{$ input/output channels and variables $\}$ is a function that defines the data model area of a state $x \in S$, if x is not "BASIC".

III. Discussion about BSC

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

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. It 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

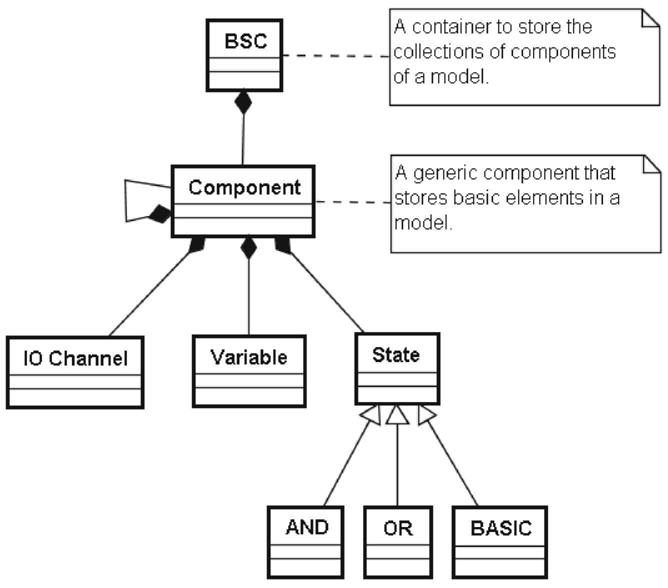
BSC
A container to store the collections of components of a model.
Component
A generic component that stores basic elements in a model.
IO Channel
Variable
State
AND
OR
BASIC

Fig. 2. *Basic Statecharts*: conceptual model.

to publish the *status* of internal elements from one component to another.

The evolution of the BSC dynamic behavior is performed by sequential steps, called the *execution cycle*. 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 in 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 BSC 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*.

Below we discuss the modeling of control software from basic actuators, sensors, and timers. 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. Operational requirements of the actuators are inserted in the model as transitions between the states in the following general form: "[guard condition] / action". The guard conditions are Boolean expressions composed of data channels and internal variables, interconnected through logical connectors $\neg$ (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.

A. 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.

Fig. 3-a 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.

B. 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's 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 update 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.

Fig. 3-b shows the *Basic Statechart* diagram for modeling the parallel behavior between the aforementioned. In this figure, $ch_i(i = 1 \dots 5)$ 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".

C. *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. Furthermore, they can be started and/or reset by some action of the model.

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. Fig. 3-c 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. The timers are updated as a global action of the model. Timer T_k is started when action $tk = 1$ is executed.

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.

IV. Case Study

This section presents a case study that realizes a simulation of a manufacturing cell (see Fig. 4), 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 and Cury in [12]. 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 Fig. 5. 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.

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;

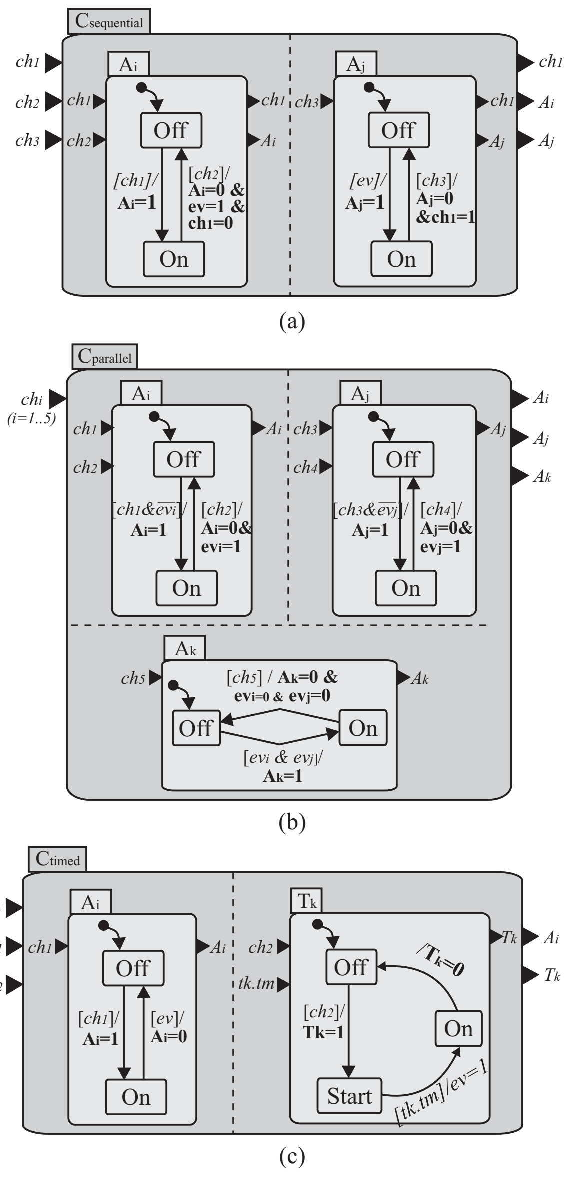

Fig. 3. Control model: operations

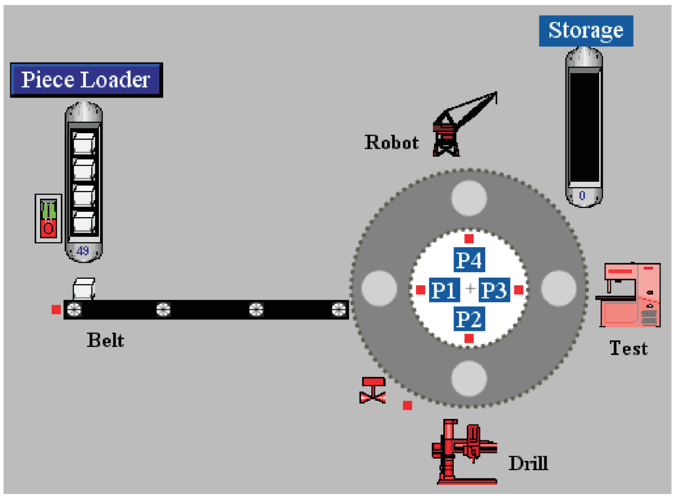

Fig. 4. Manufacturing cell: simulation

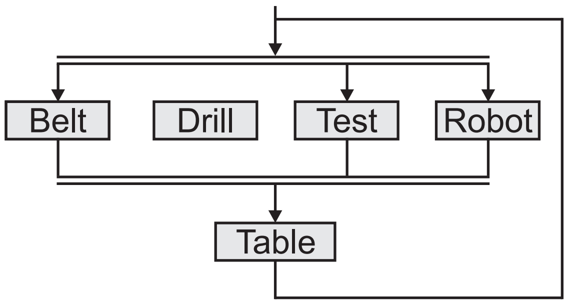

Fig. 5. Manufacturing cell: execution flow

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;

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 $E_i = true(i = 1, \ldots, 4)$. 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".

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. A similar idea is 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 "false" and "true". The complete BSC model of the controller software is shown in Fig. 6. Guard conditions g1, g2, . . . , g15 are presented in Table I, 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; $E_i(i = 1, \ldots, 4)$ are internal variables, and P1, . . . , P4, IN, S1 represent sensors installed in the plant.

TABLE I
CONTROLLER: GUARD CONDITIONS

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

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,

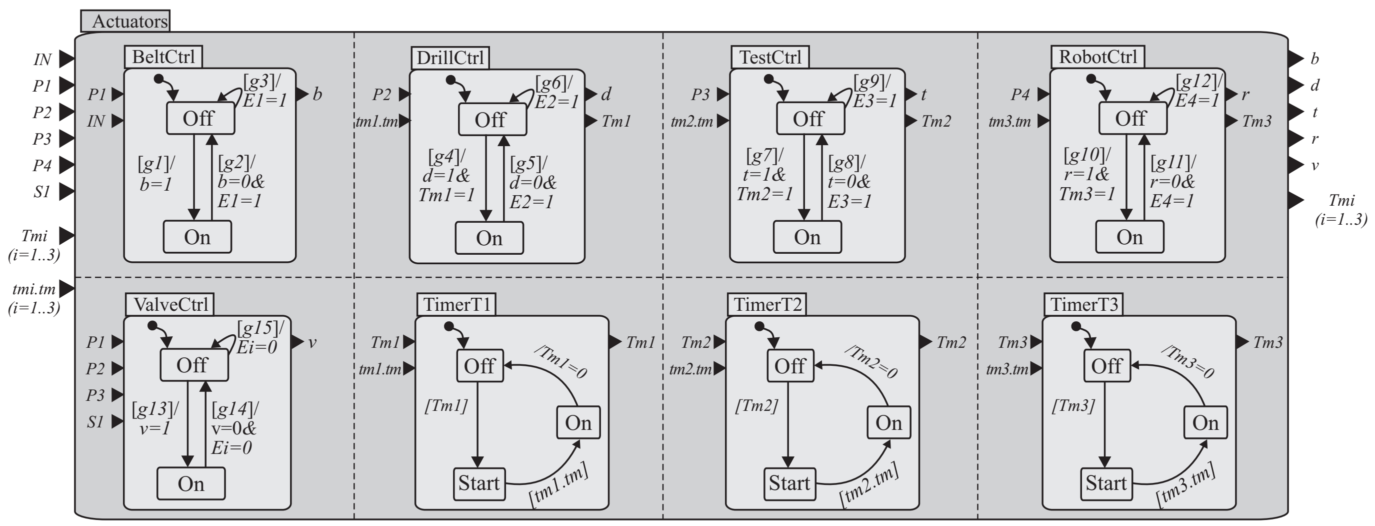

Fig. 6. Manufacturing cell: control model

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**. Due to space limitations, this analysis does not presented here.

V. Conclusion

Currently, our researches are focused in the definition of a methodology to systematize modeling, tests, and implementation phases of industrial controllers. In this paper, we presented and discussed a methodology to systematize the modeling phase, using the formalism *Basic Statecharts*.

One typical example of application in the manufacturing sector was presented as case study to illustrate the proposed methodology. From the control model presented in Fig. 6, *Ladder diagrams* have been generated by an algorithm and transcribed to the programming environment of an actual PLC in order to validate it.

Based on our experience, we believe that *Basic Statecharts* are a very interesting formalism for modeling the dynamic behavior of complex DES. Some advantages are: i) since *Basic Statecharts* are a high-level language, it allows us to model control logic and document it in an easier to understand manner for automation engineers than do low-level languages such as *Ladder diagram*; ii) as *Statecharts* are incorporated in UML 2.0, they are widespread both in academic and industrial areas; iii) because BSC are in line with original *Statecharts*, we believe that their will be easily understood bt academic and industrial professionals; iv) BSC use channels to communicate explicitly between components. Thus, the broadcast approach used by original *Statecharts* is avoided; v) because BSC work with a condition/action paradigm instead of an event-based mechanism, we believe that BSC are more appropriate than the automata-based approach (for example: Supervisory Control Theory) for modeling industrial applications in the real world; and vi) as *Basic Statecharts* retain formal properties, they can be used to verify 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.

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.

References

[1] IEC, "International eletrotechnical commission. programmable controllers part 3, programming languages," IEC61131-3, 1993.

[2] M. Bani Younis and G. Frey, "UML-based approach for the re-engineering of PLC programs," in *32nd Annual Conference of the IEEE Industrial Electronics Society (IECON'06)*, 2006, pp. 3691–3696.

[3] P. Ramadge and W. Wonham, "The control of discrete event systems," in *Proceedings of the IEEE*, vol. 77(1), 1989, pp. 81–98.

[4] M. Skoldstam, K. Akesson, and M. Fabian, "Supervisory control applied to automata extended with variables - revised," Goteborg: Chalmers University of Technology, Tech. Rep., 2008.

[5] V. Gourcuff, O. D. Smet, and J.-M. Faure, "Efficient representation for formal verification of plc programs," in *8th International Workshop on Discrete Event Systems (WODES 2006)*, Ann Arbor, Michigan, USA, 2006.

[6] W. W. Royce, "Managing the development of large software systems," in *Proc. of IEEE WESCON*, 1970, pp. 1–9.

[7] B. W. Boehm, "A spiral model of software development and enhancement," pp. 61–72, May 1988.

[8] B. Boehm, "A view of 20th and 21st century software engineering," in *ICSE '06: Proceeding of the 28th international conference on Software engineering*. New York, NY, USA: ACM Press, 2006, pp. 12–29.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[10] SCXML, "The jakarta project commons SCXML," http://jakarta.apache.org/commons/scxml/, 2006.

[11] R. Moura and L. Guedes, "Simulation of industrial applications using the execution environment SCXML," in *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, 2007, pp. 255–260.

[12] M. Queiroz and J. Cury, "Synthesis and implementation of local modular supervisory control for a manufacturing cell," in *6th International Workshop on Discrete Event Systems (WODES 2002)*, Zaragoza, Spain, 2002.