Int J Adv Manuf Technol (2008) 36:373–385
DOI 10.1007/s00170-006-0843-5

ORIGINAL ARTICLE

An implementation methodology for supervisory control theory

İ. Tolga Hasdemir · Salman Kurtulan · Leyla Gören

Received: 7 April 2006 / Accepted: 16 October 2006 / Published online: 9 September 2007

Abstract In this study, a methodology for PLC implementation of Supervisory Control Theory is introduced and realized on a pneumatic manufacturing system. The implementation methodology resolves the problem of avalanche effect and enhances program readability. We use local modular approach, which exploits modular structure of the plant and of the specifications. Local modular approach, together with the implementation methodology presented in this study provides an effective way for synthesizing and realizing supervisors for Discrete Event Systems (DES) control problems. The resulting PLC program is also modular in structure, making it handable for modification and error detection.

Keywords Discrete event systems · Supervisory control theory · Programmable logic controllers · Supervisor design

1 Introduction

An important portion of today's manufacturing systems can be classified as event driven systems or discrete event systems (DESs). Although it is possible to analyse and control this class of industrial systems formally, general methods applied in the industry are intuitive rather than being formal. For small-sized DES control problems, intuitive methods may yield practical solutions, but as the controlled system gets larger and complex, formal methods need to be applied.

The supervisory control theory (SCT) introduced by Ramadge and Wonham [1] provides a powerful framework for control of discrete event systems. The theory enables synthesis of closed loop control systems for DESs. Although the SCT has received a wide acceptance in the academy and some applications of SCT have been reported in the literature [2–6, 15], it has not been employed in the industry yet. This is mainly due to the difficulties arising in physical implementation of SCT [4, 7].

Programmable Logic Controllers (PLCs) have been used widely in industrial applications for more than 25 years, and today most of the automated manufacturing systems use PLCs as control units. Therefore, PLCs can be a potential physical platform to realize supervisors in the industry.

Finite state automata are used for representing the plant model and the supervisors in SCT. Implementation of SCT necessitates an appropriate method for developing a PLC program corresponding to the automaton that represents the theoretical supervisor. Therefore implementing the SCT is a matter of developing an appropriate PLC program, which will oblige the PLC to behave as an automaton. The methods for developing PLC codes to fulfill this purpose and the problems that may arise in doing so are discussed in [4, 7, 15]. One of the problems frequently encountered in programming an automaton as a PLC code is related to undesirable state transitions occurring in the PLC program which correspond to the theoretical automaton. This problem is called "avalanche effect" [7]. In this paper, in order to overcome this problem, we formalize a methodol-

İ. T. Hasdemir (✉) · S. Kurtulan · L. Gören
Department of Electrical Engineering,
Istanbul Technical University,
34390 Maslak,
Istanbul, Turkey
e-mail: hasdemir@elk.itu.edu.tr

S. Kurtulan
e-mail: kurtulan@elk.itu.edu.tr

L. Gören
e-mail: goren@elk.itu.edu.tr

ogy which was introduced and used in a previous work of authors [6].

One of the main difficulties arising in SCT implementations is state-space explosion. System models are generally built as a combination of subsystems and the number of states of the global system grows exponentially with the number of subsystems. Computational complexity caused by large number of states (may be more than 10^{20}) can make it impossible to design and implement a supervisor for a given DES control problem. Therefore some formal and informal techniques are used to reduce computational complexity either by reducing sizes of the automata or using different control structures comprising more than one supervisor. Informal techniques generally depend on the designer's experience and preferences such as ignoring some of the details (i.e. some of the states of the physical system) that are not necessary to appear for the given control problem [4]. Modular control, as a formal method, allows designing multiple supervisors for which control action combination is equivalent to a monolithic supervisor for a given system [12]. In this way, memory used for storing states of the supervisor can be reduced. Another formal method, called Local Modular Supervisory Control, extending the results of [12], reduces computational complexity and provides significant memory savings by using multiple supervisors and local models of the plant [13, 14]. A necessary and sufficient condition for local modularity is given in this method.

In this study, following some preliminaries of Discrete Event Systems in Section 2, a methodology for generating PLC codes corresponding to a given automaton is introduced in Section 3. In Section 4, an implementation of local modular supervisory control for an educational pneumatic manufacturing system based on the proposed methodology is presented and a hierarchical control structure for modular supervisory control is given. It is shown that modularity condition holds for the problem at hand. Finally, conclusions are given in Section 5.

2 Preliminaries and notation

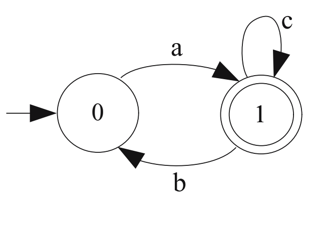
Discrete event systems evolve on spontaneously occurring events. Let $\sum$ be finite set of the events that drive the system. The set of all finite concatenations of events in $\sum$ is denoted as $\sum^*$. An element in this set is called a string. The number of events gives the length of the string. The string with zero length is denoted as ε. A subset $L \subseteq \sum^*$ is called a language over $\sum$. For a string $s \in \sum^*$, $\overline{s}$ denotes the prefixes of s and is defined as $\overline{s} = \left\{ s_p \in \Sigma^* : \exists t \in \Sigma^* \left(s_p t = s \right) \right\}$. Extending this definition to languages we get prefix closure of a language L denoted as $\overline{L}$. A language L satisfying the condition $L = \overline{L}$ is said to be prefix closed [8].

An automaton, denoted by G, is a six tuple G=(Q, $\sum$, f, Γ, q_0, Q_m) where Q is the set of states; $\sum$ is the finite set of events. f: $Q \times \sum \rightarrow Q$ is the partial transition function on its domain. $\Gamma : Q \rightarrow 2^{\Sigma}$ is the active event function. Γ(q) is the set defined for every state of G and represents the feasible events of q. q_0 is the initial state and $Q_m \subseteq Q$ is the set of marked states representing a completion of a given task or operation. Automata are generally represented by state diagrams. Figure 1 shows a simple automaton with two states. State 0 with a directed line pointing it is the initial state. State 1 with double circle is the marked state of the automaton. Directed lines (transitions) represent transition functions of the automaton. Labels of the transitions correspond to events. The events associated with all the transitions from a state give the active event set of that state.

The language generated by G is denoted by L(G) and defined as $L(G)=\{s \in \sum^*: f(q_0,s) \text{ is defined}\}$. The language marked by G is denoted by L_m(G) and defined as $L_m(G)=\{s \in \sum^*: f(q_0,s) \in Q_m\}$. A DES G is said to be nonblocking if $L(G) = \overline{L_m(G)}$. Product and parallel composition operations are defined for expressing two forms of joint behaviour of multiple automata that operate concurrently. Product of two automata, say G_1 and G_2 is denoted as $G_1 X G_2$ and represents the synchronous behaviour of the two automata. Thus, an event occurs in the resulting automata if and only if it occurs in both automata. The generated and marked languages of the resulting automaton will be $L(G_1 \text{ X } G_2)=L(G_1) \cap L(G_2)$ and $L_m(G_1 \text{ X } G_2)=L_m(G_1) \cap L_m(G_2)$, respectively. Parallel composition of automata G1 and G2 is denoted as $G_1||G_2$. In the resulting automaton common events occur synchronously, while the other events occur asynchronously. Therefore, if the event sets $\sum_1$ and $\sum_2$ of the automata are equal then parallel composition of G_1 and G_2 will be equivalent to the product of G_1 and G_2 [8].

The supervisory control theory (SCT) uses formal languages to model the uncontrolled behaviour of discrete event systems and specifications for the controlled behaviour. The objective is to restrict the behaviour of the system to a desired behaviour, which is represented by the specifications. This is done by disabling some events to prevent the occurrence of some undesired strings in the system. The decision of which event will be disabled is made by another simultaneously executing system, called the supervisor which is also represented by an automaton.

Fig. 1 A simple automaton



The active event set associated with a state of the supervisor includes the events that are allowed to occur in the corresponding state of the controlled system.

In SCT, events are divided into two classes as controllable events and uncontrollable events. The set of controllable events is denoted by $\sum_c$, while $\sum_{uc}$ represents the uncontrollable event set. The supervisor has no effect on uncontrollable events, which means that the supervisor cannot prevent an event of $\sum_{uc}$ from happening. The existence of a supervisor is guaranteed if the desired language satisfies a condition. This condition is called as the controllability condition and defined as $\overline{K}\sum_{uc} \cap M \subseteq \overline{K}$ where $\overline{K}$ is the language that will be generated under the control of the supervisor and M is the language generated by the uncontrolled system.

The behaviour of a DES G, under the control of supervisor S is denoted by L(S/G). A language K is called L_m(G)-closed when the language satisfies the condition $K = \overline{K} \cap L_m(G)$. If the desired behaviour $K \subseteq L_m$(G) is controllable and L_m(G)-closed, then $\overline{K} = L(S/G) = \overline{L_m(S/G)}$, and this means that the controlled behaviour is nonblocking.

Modular Supervisory Control guarantees nonblocking of DES controlled by multiple supervisors if the modularity condition $\overline{\bigcap_i L_m(S_i/G)} = \bigcap_i \overline{L_m(S_i/G)}$, holds. Local Modular Control approach extends this result to exploit the modularity of the controlled system G. When a DES G is given as a composition of $G_{loc,j}$, j=1,...,m, local modularity condition which is given as $\overline{\|_{j=1}^{m} L_m(S_{loc,j}/G_{loc,j})} = \|_{j=1}^{m} \overline{L_m(S_{loc,j}/G_{loc,j})}$ is equivalent to global modularity [13, 14]. While modelling a system as a combination of subsystems, local modular approach uses product system representation (PSR) of the global system [12, 14]. In the PSR, subsystems are modelled by asynchronous automata.

To help reading, a summary of notations is given in Table 1.

In the next section, we develop a methodology which enables expressing a given automaton in logical domain, in particular as a PLC program.

Table 1 Summary of notations

Symbol	Description
G=(Q, ∑, f, Γ, q_0, Q_m)	An automaton G with state set Q, event set ∑ set of state transition functions f, set of active events Γ initial state q_0, set of marked events Q_m
L(G)	Language generated by automaton G
L_m(G)	Language marked by automaton G
$\overline{s}$	Prefixes of string s
$\overline{L}$	Prefix closure of a language L
$G_1 \times G_2$	Product of automata G_1 and G_2
$G_1 \| G_2$	Parallel composition of automata G_1 and G_2
$L(G_1) \cap L(G_2)$	Language obtained by product of automata G_1 and G_2
$L(G_1) \| L(G_2)$	Language obtained by parallel composition of automata G_1 and G_2
L(S/G)	Language generated by a system G under the supervision of supervisor S
L_m(S/G)	Language marked by a system G under the supervision of supervisor S

3 Expressing automata in logical domain

In order to realize a given automaton using PLCs, states, events and state transition functions of the automaton should be defined in PLC language. IEC 61131-3 standard defines the graphical and textual languages for PLCs [9]. Among these, Ladder Diagram (LAD) is a graphical language which is traditionally preferred by the practitioners. Therefore, we will use LAD to program a given state machine. For detailed information about PLCs and Ladder Diagram, the interested reader is referred to [10, 11].

States, events and state transition functions of an automaton are simply mathematical sets, and a PLC is a device performing programmed logical expressions of Boolean valued signals in a cyclic manner. Therefore, a convenient method to represent these sets of the automaton in the PLC environment is needed. States can be represented by memory bits. If a memory bit corresponding to a state of the automaton is set, this means that the realized automaton is in that state. Events are assumed to be momentary in DES theory. Intuitively, this assumption can be met by using rising and falling edges of sensor signals. Falling and rising edges of a signal is detected by comparing the logical value of the signal between previous and current scan cycle. If a change from logical false to true (true to false), rising edge (falling edge) will output a value of logical true for one scan cycle time, which is the shortest possible period in the PLC environment. The next step is to define transition functions in the PLC language. This can be done using Boolean AND operations and SET/RESET functions. Boolean AND operation is used for memory bits representing the current state and memory bits representing the events which cause a transition from this state. The result of AND operation determines whether the transition condition is satisfied. If satisfied, current state is reset and next state is set by the use of SET/RESET functions. An example is given in Fig. 2. In the LAD program of Fig. 2b, q_1, q_2 and q_3 are memory bits representing the states of the automaton in Fig. 2a, while σ_1, σ_2, σ_3 and σ_4 correspond to signals whose rising edge represents the events of the automaton.

3.1 The avalanche effect problem

Although the above approach is straightforward and easy to apply, it may yield an incorrect realization depending upon the structure of the given automaton. This incorrect

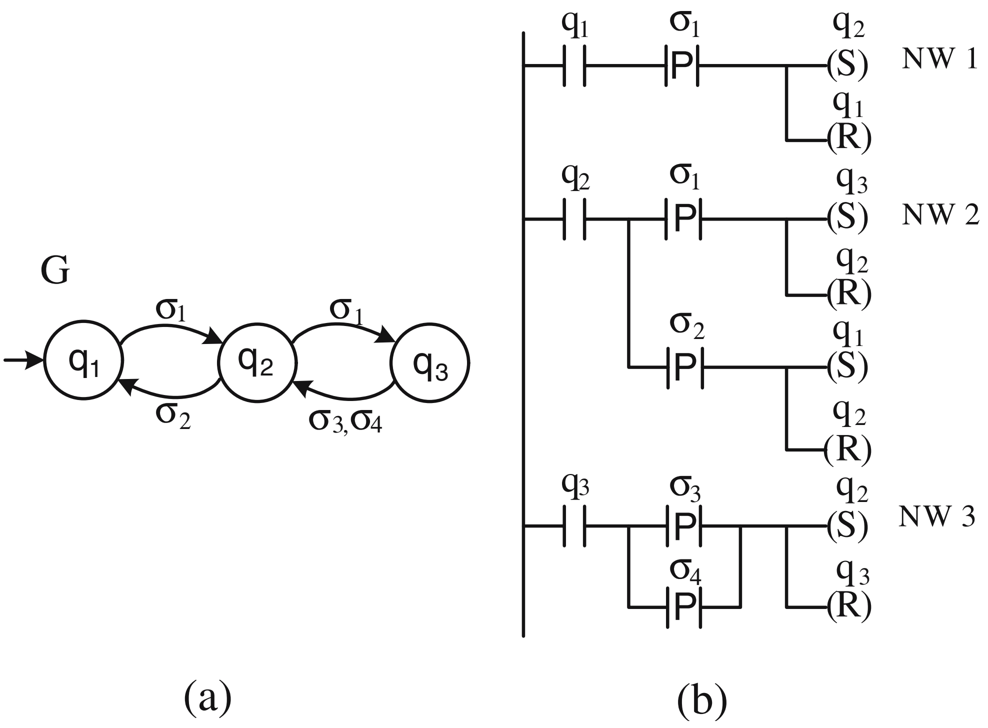


Fig. 2 An example automaton (**a**), and LAD realization of this automaton (**b**)

behavior is due to the occurrence of undesirable state transitions of the realized automaton and is called avalanche effect problem [7]. Consider the automaton and its LAD realization in Fig. 2 and suppose that the automaton in Fig. 2a is in state q_1; therefore bit q_1 is set in its LAD realization. On the occurrence of event σ_1, the automaton transits to state q_2 and PLC program sets q_2, resets q_1. Normally, the automaton stays in state q_2 until another σ_1 event occurs. However, when the second network is executed just after the first network, the PLC program of Fig. 2b will immediately jump to q_3, since q_2 is set and rising edge value of signal σ_1 is still true. Therefore, the behavior of the PLC realization in Fig. 2b is incorrect. We can conclude from this example that, for a given automaton, if successive occurrence of an event causes successive state transitions, the avalanche effect problem is inevitable with the programming approach described above.

In [7] a methodology based on reversing the order of ladder diagram networks for solving this problem is discussed. However, this methodology is highly dependent upon the problem at hand, and may fail in some cases. As an example, a simple automaton which may be used for detecting event pairs of σ_1 is given in Fig. 3. Obviously, σ_1 causes the avalanche effect problem: when state q_1 is active, an occurrence of σ_1 causes a transition to q_2 and then back to q_1 in the same PLC cycle. If the method described above is utilized, arranging the networks in reverse order prevents a transition back to q_1, but in this case the problem occurs when state q_2 is active: A $\sigma 1$ event causes a transition to q_1 and then back to q_2. This example shows that the method based on reversing the networks fails when there exist transitions between two states of an automaton in both directions with the same event.

In Sect. 3.2, we introduce a methodology for realizing a given automaton which constitutes a general solution for the avalanche effect problem in a systematic way and provides an advantage in program readability and memory saving in many cases.

3.2 Methodology for PLC realization

When SET and RESET functions are used, these functions appear in more than one network to program all transition conditions related to a single state. For example, in Fig. 2b, transition conditions of state q_2 necessitate using SET or RESET functions four times. Apparently, this reduces program readability. Furthermore, depending upon the PLC being used, SET and RESET functions may occupy a considerable amount of program memory in the PLC as compared to the standard bit logic operations. In this subsection we will develop a programming technique which does not use SET and RESET functions and renders a PLC program without avalanche effect problem. In the

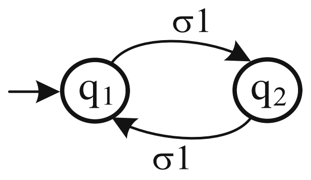


Fig. 3 An automaton with avalanche effect problem

following, we will introduce the methodology in a formal way. The methodology will also be considered from a practical point of view within the examples provided in this section.

As a first step toward the development of the methodology, we will try to realize SET and RESET functions by standard bit logic operations. Let us denote the SET condition of a logic variable q by S, and RESET condition by R. Representing current and next values of q by q(k) and q(k+1), respectively, it can easily be shown that the logic function

$$q(k+1) = S + q(k).R' \tag{1}$$

can be used to set q by S, and to reset it by R, where "+" corresponds to OR, "." corresponds to AND operations, and R' means inverse of R. This logic function can be used for realization of the rules defined by state transition function of a given automaton. In order to translate state transition function into logical domain in terms of SET (S) and RESET (R) conditions of Eq. 1, some definitions will be given below. In these definitions, we use the index sets I_q= {1,2,...,N_q} and $I_\sigma = \{1, 2, ..., N_\sigma\}$ for states and events of the automaton, respectively.

Definition 1 Given an automaton with state transition function f: $Q\times\sum\rightarrow Q$, where $Q = \{q_1, q_2, ...q_{N_q}\}$ is the set of states and $\sum = \{\sigma_1, \sigma_2, ..., \sigma_{N_\sigma}\}$ is the set of events. We define State Index Set for Transition Conditions to State q_i by

$$I_{Sq}(i) = \{j \in I_q | \exists k \in I_\sigma, f(q_j, \sigma_k) = q_i\}, \forall i \in I_q \tag{2}$$

Note that, I_{Sq}(i) is the set of indices of the states from which a transition to state q_i is feasible.

Definition 2 Given an automaton with state transition function f: $Q\times\sum\rightarrow Q$, where $Q = \{q_1, q_2, ..., q_{N_q}\}$ is the set of states and $\sum = \{\sigma_1, \sigma_2, ..., \sigma_{N_\sigma}\}$ is the set of events. We define Event Index Set for Transition Conditions to State q_i by

$$I_{S\sigma}(i) = \{j \in I_\sigma | \exists k \in I_q, f(q_k, \sigma_j) = q_i\}, \forall i \in I_q \tag{3}$$

Similar to Definition 1, $I_{S\sigma}(i)$ represents the set of indices of events by which a transition to state q_i is possible.

Definition 3 Given an automaton with state transition function f: $Q\times\sum\rightarrow Q$, and active event function $\Gamma : Q \rightarrow 2^\Sigma$, where $Q = \{q_1, q_2, ..., q_{N_q}\}$ is the set of states and $\sum = \{\sigma_1, \sigma_2, ..., \sigma_{N_\sigma}\}$ is the set of events. The following index set corresponds to the Index Set of Events That Cause a Transition from State q_j to State q_i and is defined by

$$I_{T\sigma}(i,j) = \{k \in I_\sigma | \sigma_k \in \Gamma(q_j) \wedge k \in I_{S\sigma}(i)\}, \forall (i,j) \in I_{Sq} \times I_{Sq} \tag{4}$$

Notice that, $I_{T\sigma}(i,j)$ may be empty for some combinations of i and j.

Definition 4 Given an automaton with state transition function f: $Q\times\sum\rightarrow Q$, and active event function $\Gamma : Q \rightarrow 2^\Sigma$, where $Q = \{q_1, q_2, ..., q_{N_q}\}$ is the set of states and $\sum = \{\sigma_1, \sigma_2, ..., \sigma_{N_\sigma}\}$ is the set of events. Event Index Set for Transition Conditions from State q_i is defined by

$$I_{R\sigma}(i) = \{j \in I_\sigma | \sigma_j \in \Gamma(q_i)\}, \forall i \in I_q \tag{5}$$

In the following, we give an example to clear the meaning of the definitions.

3.2.1 Example 1

Consider again the automaton in Fig. 2a. State and event index sets of the automaton can be picked as I_q={1,2,3} and $I_\sigma = \{1, 2, 3, 4\}$, respectively. By using Definitions 1 through 4 for state q_2, the following index sets can be obtained:

State Index Set for Transition Conditions to State q_2: $I_{Sq}(2)$={1,3},

Event Index Set for Transition Conditions to State $q_2 : I_{S\sigma}(2) = \{1, 3, 4\}$, Index Set of Events That Cause a Transition from State $q_j (\forall j \in I_q)$ to State $q_2 : I_{T\sigma}(2, 1) = \{1\}$, $I_{T\sigma}(2, 2) = \phi$, $I_{T\sigma}(2, 3) = \{3, 4\}$,

Event Index Set for Transition Conditions from State $q_2 : I_{R\sigma}(2) = \{1, 2\}$.

It has to be noted that, although the definition of an automaton includes the initial state and the set of marked states, we have not taken them into consideration in the above definitions, since we are primarily focused on state transitions. However, for a complete realization of a given automaton, initial state should also be taken into account. As we will discuss shortly, initial state can easily be programmed by setting a memory bit at the start-up of the PLC. For the case of marked states, there is nothing to do in the realization stage, since marked states are used for the validation purposes in the design procedure rather than realization.

Note also that, for simplicity, we use the same notations for the elements (i.e. events and states) of a given automaton and their corresponding logic variable representations as in the example of Fig. 2. The meaning of the notation can be distinguished from the context or from the domain of the mathematical expression in which it appears. In the following, we will be working in logical domain. Therefore all of the notational symbols correspond to logic variables and "+" (or the summation operator $\sum$) represents logical OR operation, and "." (or the multiplication operator $\prod$) represents logical AND operation as in Eq. 1. In the following two definitions, we will derive the expressions of SET and RESET conditions for a state of a given automaton.

Definition 5 The Logical Expression of SET Condition for State q_i is given by

$$S_i(k) = \sum_{j \in I_{Sq}(i)} q_j(k).\sigma(i,j) \tag{6}$$

where k stands for the kth time instant and,

$$\sigma(i,j) = \sum_{n \in I_{T\sigma}(i,j)} \sigma_n \tag{7}$$

is the event representing the equivalent effect of events that cause a transition from state q_j to state q_i.

Definition 6 The Logical Expression of RESET Condition for State q_i is defined as

$$R_i(k) = \sum_{j \in I_{R\sigma}(i)} \sigma_j(k) \tag{8}$$

Having defined logical expressions of SET and RESET conditions for state q_i, it is now possible to use Eq. 1 in order to obtain a logic function of state q_i:

$$q_i(k+1) = S_i(k) + q_i(k).R_i'(k) \tag{9}$$

The logic function given by Eq. 9 can easily be programmed using LAD language. Representing the logic variables $q_i(k+1)$ and $q_i(k)$ by the memory bits Q_i and q_i, respectively, the PLC program in Fig. 4 can be used to realize Eq. 9. The memory bits q_i and Q_i can be considered as the logic representation values of state q_i for the current and next PLC scan cycles, respectively. Notice from Fig. 4 that, memory bits Q_i and q_i are synchronized at the end of the LAD, in order to transfer the value of Q_i to q_i.

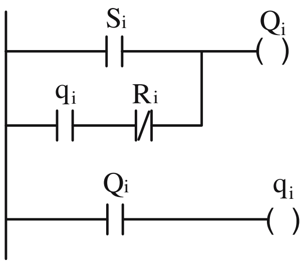


Fig. 4 LAD corresponding to logic function given by Eq. 10

3.2.2 Example 2

Recall the automaton given in Example 1 and consider again state q_2. The following can be obtained according to Definitions 5 and 6:

$$\begin{aligned} S_2(k) &= \sum_{j \in I_{Sq}(2)} q_j(k).\sigma(2,j) \\ &= q_1(k).\sigma_1 + q_3(k).(\sigma_3 + \sigma_4) \end{aligned} \tag{10}$$

$$R_2(k) = \sum_{j \in I_{R\sigma}(2)} \sigma_j(k) = \sigma_1 + \sigma_2 \tag{11}$$

Using Eqs. 10 and 11 together with Eq. 9, the logic function for state q_2 is obtained as

$$\begin{aligned} q_2(k+1) = {} & q_1(k).\sigma_1 + q_3(k).(\sigma_3 + \sigma_4) \\ & + q_2(k).(\sigma_1 + \sigma_2)' \end{aligned} \tag{12}$$

or,

$$\begin{aligned} q_2(k+1) = {} & q_1(k).\sigma_1 + q_3(k).(\sigma_3 + \sigma_4) \\ & + q_2(k).\sigma_1'.\sigma_2' \end{aligned} \tag{13}$$

Similarly, the logic functions for state q_1 and state q_3 can be obtained as

$$q_1(k+1) = q_2(k).\sigma_2 + q_1(k).\sigma_1' \tag{14}$$

and

$$q_3(k+1) = q_2(k).\sigma_1 + q_3(k).\sigma_3'.\sigma_4' \tag{15}$$

Using Eqs. 13, 14 and 15, the PLC program corresponding to the given automaton can be developed as in Fig. 5a. The memory bits representing the events in the PLC programs of Fig. 5 are assumed to be obtained by falling or rising edges of corresponding sensor signals.

It has to be noted that, from a practical point of view, it is straightforward to apply Definition 5 and Definition 6 together with the Eqs. 9, 10 and 11 in order to obtain the PLC program of the given automaton. Let us interpret the PLC program in Fig. 5a which is obtained as a realization of the automaton given in Fig. 2a. It is apparent from the automaton that a transition to state q_1 takes place if "q_2 is active and σ_2 occurs". Clearly, the first two series contacts (q_2 AND σ_2) in NW1 correspond to this expression. In fact, the same logical expression is obtained if the SET condition given by Definition 5 is applied. Again, it is apparent from the automaton that, state q_1 stays active as long as "the automaton is in state q_1 and σ_1 does not occur". The other two series contacts (q_1 AND σ_1') in NW1 realizes this expression. This expression corresponds to the one that can be obtained from Definition 6 and the logic function given in Eq. 9. Naturally, these two series contact groups (q_2

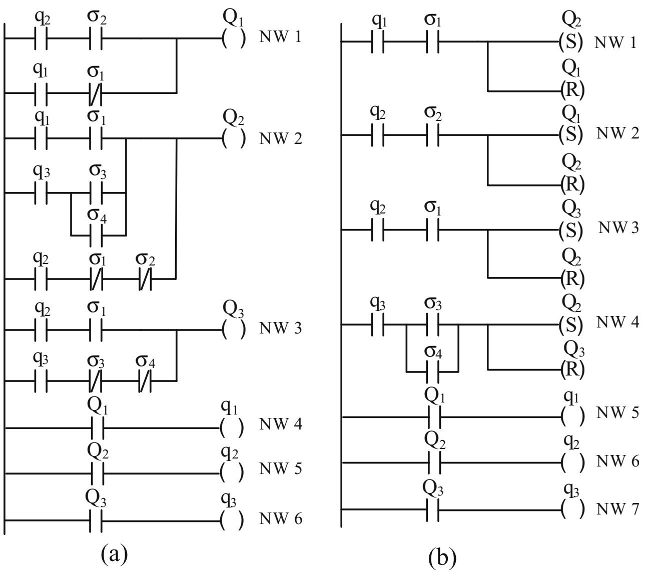


Fig. 5 LADs with (**a**) and without (**b**) SET, RESET functions for the automaton in Example 2

AND σ_2 & q_1 AND σ_1') are connected in parallel to transit to state q_1 and to keep it active. Similarly, NW2 expresses the fact that (1) "a transition to q_2 takes place if q_1 is active and σ_1 occurs, or q_3 is active and σ_3 or σ_4 occurs"; (2) "q_2 stays active if the automaton is in state q_2 and neither σ_1 nor σ_2 occurs". Similar considerations can be made for state q_3. To summarize, the main idea of the methodology is to obtain the conditions of transition to a state and keep it active. This practical interpretation may be used for programming the conditions defined formally above and may allow the programmer to apply the methodology in a straight-forward manner.

Recall that, the automaton we have been considering in Examples 1 and 2 has the problem of avalanche effect with the programming approach discussed in Section 3.1. Now, let us reconsider the PLC program given in Fig. 5a corresponding to this automaton. Assume that the automaton is in state q_1, and therefore, logical value of memory bit q_1 is true. On the occurrence of event σ_1, Q_2 becomes true, but this does not effect the value of Q_3 in the next network, since Q_3 is effected by q_2, not by Q_2. In the next PLC cycle, q_2 becomes true, but now σ_1 is false, because it is obtained by rising or falling edge functions.

The discussion above shows that, this methodology guarantees a PLC program free of avalanche effect problem. The methodology can successfully be adapted to allow SET and RESET functions as in Fig. 5b. However, as mentioned previously, using SET and RESET functions reduces program readability and may necessitate a larger amount of program memory. Indeed, Q_2 is SET or RESET in four different networks in the PLC program of Fig. 5b, while in Fig. 5a, a single network corresponds to each of the memory bits Q_1, Q_2 and Q_3. Furthermore, for Siemens S7-200 PLCs as an example, the sizes of PLC programs in Fig. 3a and b are 73 bytes and 103 bytes, respectively.

The PLC program of Fig. 5a corresponds to a complete realization of the automaton in Fig. 2, except for the initial state. Initial state can be realized in two different ways. One method is to make use of special memories signaling the startup of the PLC. For example, a special memory bit which has the value of logical true for the first PLC scan cycle only is addressed by SM0.1 in Siemens S7 200 PLCs. Like S7-200, many PLCs support special memories or program bocks that indicate the first PLC scan. However, if the PLC does not support this kind of facility, the PLC program part given in Fig. 6 using memory bits M0, M1 and M2 can be used to detect the PLC initialization. Though the given program detects the second PLC scan, practically it can be assumed that the memory bit M2 signs the startup of the PLC program by issuing a pulse for one scan time. It should be noted that the memory bits used in Fig. 6 must not be of retentive type, since in that case it might be impossible to detect a change to logic 1 in the value of bit M1.

In the other method, we propose to use a simple idea expressed as "the automaton is in initial state if it is not in any of the states". Formally, this expression can be written in logical domain as

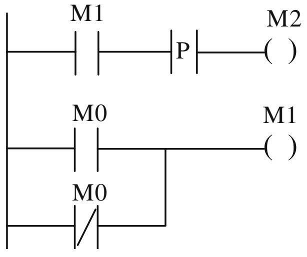
$$q_0(k+1) = \prod_{i \in I_{qr}} q_i'(k) \tag{16}$$

where $I_{qr} = \{i | i \in I_q \backslash \{j\}, q_j = q_0\}$, q_0 represents the initial state of the given automaton G=(∑, Q, f, Γ, q_0, Q_m), and q_i, i ∈ I_q={1,2,...,N_q} are the states of the automaton indexed by the index set I_q. We will see in the following example that this method can not only be used for realizing the initial state, but also for programming all of the transitions regarding the initial state.

3.2.3 Example 3

Figure 7a shows the network which can be replaced by NW1 of Fig. 5a in order to realize initial state q_1 when a Siemens S7-200 PLC is utilized. Notice that, the only difference from NW1 of Fig. 5a is the addition of special memory bit SM0.1 as a parallel contact.

Fig. 6 PLC program used for detecting PLC initialization



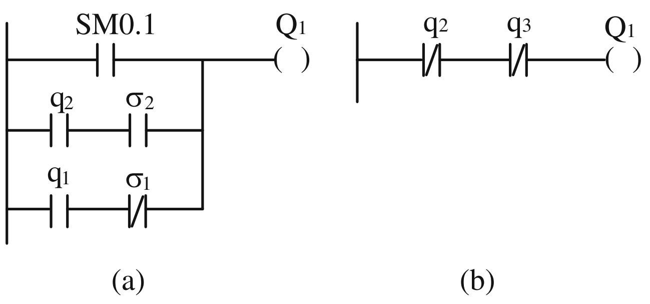


Fig. 7 Initial state realizations of Example 3

The program in Fig. 7b is the LAD realization of the following logical expression obtained by using Eq. 16.

$$q_1(k+1) = q_2'(k).q_3'(k) \tag{17}$$

Again, logic variables $q_i(k+1)$, and $q_i(k)$ are represented by Q_i and q_i, respectively. Note that, the network of Fig. 7b can not only be used for realizing the initial state q_1, but also for state transitions regarding q_1. Indeed, replacing NW 1 of the PLC program in Fig. 5a with the NW in Fig. 7b will cause no harm to the realization of state transition rules, since the new network sets Q_1 whenever none of the states are set. This means that, Q_1 will be set when a transition to q_1 is necessary. Therefore, applying the realization rule of Eq. 16 for the initial state provides additional simplification on the PLC realization.

Having defined the realization rules of state transition and initial state of a given automaton, the formal summary of the proposed methodology can now be given through the following steps:

i. Determine the state and event index sets I_q and I_σ for the given automaton,
ii. Apply Definitions 1 to 4 in order to derive the index sets necessary for transition conditions,
iii. Obtain the logical expressions of SET and RESET conditions by using Definitions 5 and 6,
iv. Program the initial state either by using PLC's memory bit signaling the start up or by using Eq. 16,
v. Program state transition rules of the automaton obtained in Step (iii) by using the LAD program given in Fig. 4.

As considered previously in Example 2, practical interpretations of the methodology may be used to apply the procedure summarized above in a straight-forward way for manual programming. However, the formal setting of the methodology above may enable developing algorithms or computer programs that can be used to generate the corresponding PLC program automatically for a given automaton.

Besides being a general solution for the avalanche effect problem, the proposed methodology provides a systematic way which allows the programmer to develop the PLC code without having to make "intelligent decisions" regarding the avalanche effect as opposed to the standard direct transition method mentioned in Section 3.1 [7]. In fact, the proposed methodology can be applied without caring for the avalanche effect.

It has to be noted that, for the cases without avalanche effect problem, it is not necessary to use an additional memory bit for each state. The proposed methodology can effectively be used with one memory bit for each state in these cases. Moreover, for the cases with avalanche effect problem, it is possible to use one memory bit for the states which are not related to the problem. As an example, for the PLC realization of the automaton given in Fig. 2, two memory bits are necessary for state q_2 (namely q_2 and Q_2 bits) but one memory bit is enough for states q_1 and q_3. Therefore, once the avalanche effect problem is detected, it is possible to decrease the number of memory bits by using one memory bit for the states which are not related with the problem. However, this necessitates making "intelligent" decisions.

Another important property of the proposed methodology is that, the network structure of the obtained PLC program corresponds to the realized automaton state by state. This means that, each network corresponds to a unique state, and is enough to completely define the transition conditions of that state. As an example, in Fig. 5a, NW1, NW2 and NW3 completely and uniquely define the transition rules of states q_1, q_2 and q_3, respectively. This feature is important to detect errors and realize modifications. In standard transition method using SET and RESET functions, in order to understand the transition conditions of a given state, one must detect all the SET and RESET functions regarding that state and then combine the obtained information in order to reveal the transition conditions of the state.

Moreover, in standard direct transition method, the number of the SET and RESET functions is twice of the number of transitions, since each transition must use a SET function and a RESET function. For some PLCs, SET and RESET functions occupy a considerable amount of program memory as compared to standard bit logic operations. For example, in Siemens S7-200 PLCs, one SET or RESET function occupies 7 bytes of program memory while a standard bit logic operation occupies 2 bytes. PLC programmers often experience the problem of program memory shortage and generally the only solution is to use an upper model PLC, which costs higher. One of the main motivations of the proposed methodology is to avoid using SET and RESET functions, hence to provide memory economy. Up to our experience in various cases, especially depending on the number of transitions in the given automaton, comparisons of the two methods show that the proposed methodology provides memory savings up to 30%.

In the following section we apply the methodology obtained in this section for the supervisory control of a pneumatic manufacturing system.

4 The pneumatic system

The pneumatic system (SMC-MAP 205) shown in Fig. 8 is a flexible automation mini-cell built for educational purposes comprising pneumatic manipulators and a PLC (Siemens S7-200). It demonstrates the assembly of a rotary mechanism composed of a base, a bearing, a shaft and a cover. Assembly of the rotary mechanism is carried out by means of three manipulators and four cylinders. We will call the manipulators as Manipulator E, H and K. Manipulator E is used for picking up the bearing and inserting it into the base. Manipulator H carries out the process necessary for the assembly of the shaft by picking up and placing it inside the previously inserted bearing. The task of Manipulator K is to pick up and place the cover onto the base after the shaft is inserted into the bearing. The four cylinders are called Cylinder A, B, C and D.

Cylinder A is used for feeding the base to the test point. Cylinder C tests a possible incorrect position of the base. The task of Cylinder B is to move the tested base to the assembly point. Finally, Cylinder D is used for ejecting the assembled mechanism. If the base fails the test carried out by Cylinder C, then Cylinder D ejects the base before the assembly process begins.

Figure 8b gives a schematic representation of the system. The supply mechanisms of the components are beyond the scope of this study and we assume that the components are supplied by a mechanism not modelled here.

The order of assembly is as follows. First a base arrives at the assembly point. Then a bearing is inserted into the base. Following insertion of the bearing a shaft is inserted into the bearing. And finally, a cover is placed onto the base.

The original control algorithm provided by the vendor allows only one rotary mechanism in an assembly process. Also, picking up and insertion operations of the components are done sequentially causing additional decrease in efficiency. The purpose of this study is to minimize the average time spent during the manufacturing of an assembled mechanism by an efficient operation. There are two steps to increase efficiency: One is to allow the system to work on more than one product simultaneously, and the other is picking up a supplied component immediately, therefore making the component ready for assembly before insertion/placement operation of the component begins. It is also necessary to prevent a possible collision of the product components, i.e. bases.

4.1 Modelling and control

The model of the uncontrolled system can be built in different levels depending upon the control purpose. A low level model gives detailed information of the system such as sensor readings, actuator status and deals with operational procedures [5, 15]. However, this kind of modelling is not suitable for large-scale control problems, since the model may become extremely large. Therefore, in this study modelling is done at a higher level, which defines start and end operations of subsystems. Each of the lower level control algorithms responsible for operating its corresponding subsystem is also developed using formal methods [5].

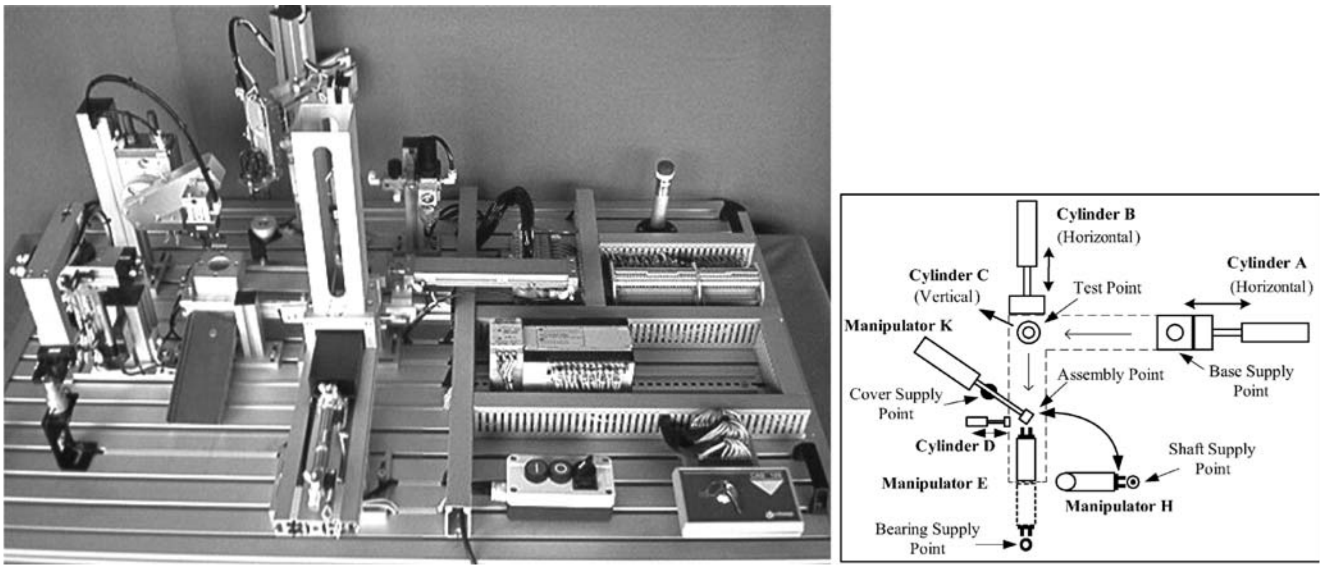


Fig. 8 The pneumatic system (**a**), schematic representation (**b**)

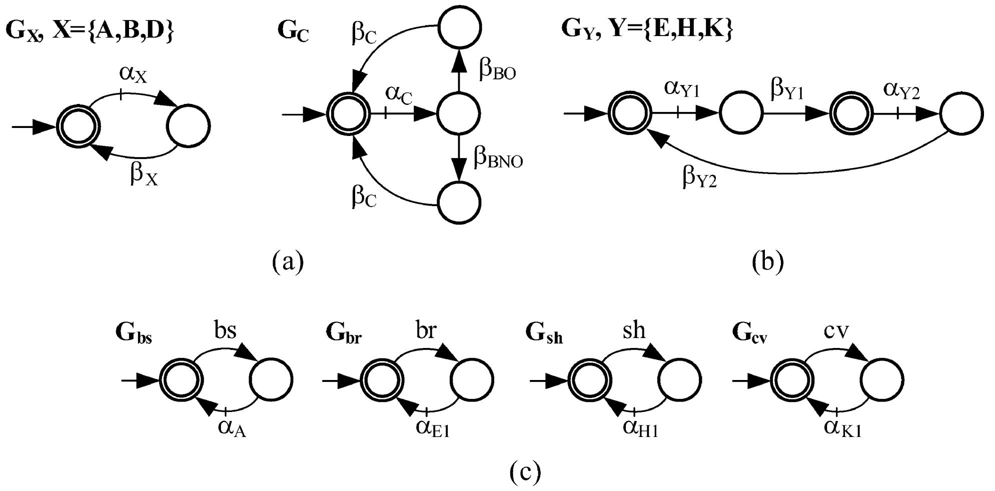


Fig. 9 (**a**) Cylinder, (**b**) Manipulator, (**c**) Component existence models

In the modelling stage, decomposition of the global system into subsystems exploits physical decomposition of the system. Therefore, there are seven models corresponding to the four cylinders and the three manipulators. Also there are four more models, which describe existence and absence of the supplied components.

Figure 9a shows the cylinder models. For x∈{A,B,C,D}, the controllable event α_X represents the start operation, while the uncontrollable event β_X represents the end operation of the corresponding subsystem. The uncontrollable events β_{BO} and β_{BNO} appearing in the model of cylinder C represent the test results of the base. If the base position is correct then event β_{BO} is generated, if not event β_{BNO} is generated.

Manipulator models are given in Fig. 9b. There are two different start and end operations defined for each manipulator. The first start operation is denoted by α_{Y1}, Y∈{E,H,K}, and this matches to start of picking up operation for the corresponding assembly component. The second start operation is denoted by α_{Y2} and represents start of insertion/placement operation of the component. β_{Y1} and β_{Y2} are used to denote the end of picking up and insertion/placement operations, respectively. Figure 9c shows the automaton models of component existence and absence. The uncontrollable events br, sh, bs and cv correspond to the existence of bearing, shaft, base and cover, respectively.

Cylinder models and manipulator models together with component existence/absence models represent the Composed System Representation [14] of the global system. In the Product System Representation, the following subsystem models should be obtained by parallel compositions.

$$G_{Abs} = G_A \| G_{bs}, G_{Ebr} = G_E \| G_{br}$$
$$G_{Hsh} = G_H \| G_{sh}, G_{Kcv} = G_K \| G_{cv}$$

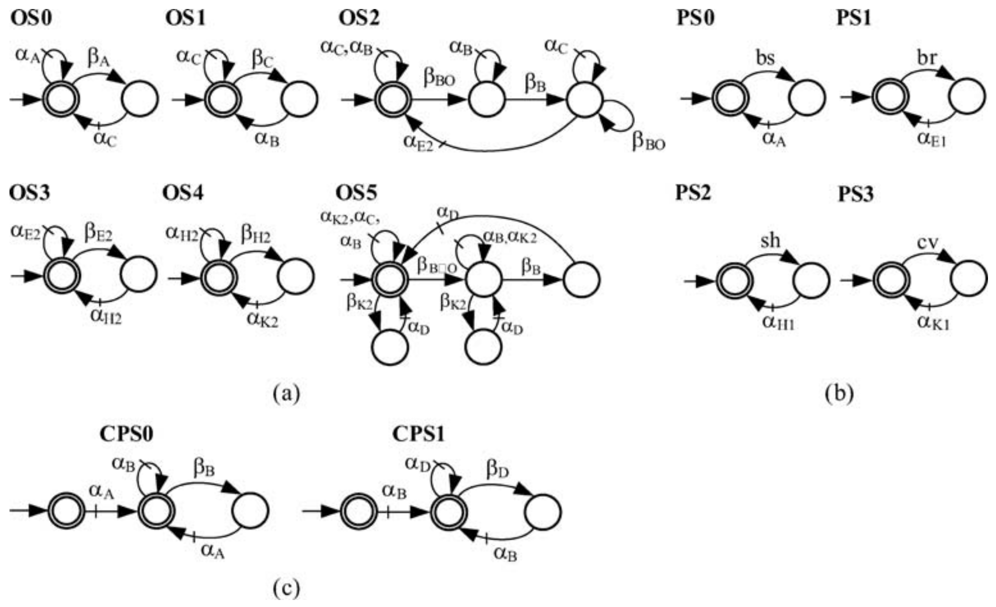


Fig. 10 (**a**) Ordering, (**b**) Preparation, (**c**) Collision protection specifications

Table 2 Local systems and local specifications

Local System Model	Local Specification
$G_{loc,a0}=G_{Abs}\|\|G_C$	$H_{loc,a0}=OS0\|\|G_{loc,a0}$
$G_{loc,a1}=G_B\|\|G_C$	$H_{loc,a1}=OS1\|\|G_{loc,a1}$
$G_{loc,a2}=G_B\|\|G_C\|\|G_{Ebr}$	$H_{loc,a2}=OS2\|\|G_{loc,a2}$
$G_{loc,a3}=G_{Ebr}\|\|G_{Hsh}$	$H_{loc,a3}=OS3\|\|G_{loc,a3}$
$G_{loc,a4}=G_{Hsh}\|\|G_{Kcv}$	$H_{loc,a4}=OS4\|\|G_{loc,a4}$
$G_{loc,a5}=G_B\|\|G_C\|\|G_D\|\|G_{Kcv}$	$H_{loc,a5}=OS5\|\|G_{loc,a5}$
$G_{loc,b0}=G_{Abs}$	$H_{loc,b0}=PS0\|\|G_{loc,b0}$
$G_{loc,b1}=G_{Ebr}$	$H_{loc,b1}=PS1\|\|G_{loc,b1}$
$G_{loc,b2}=G_{Hsh}$	$H_{loc,b2}=PS2\|\|G_{loc,b2}$
$G_{loc,b3}=G_{Kcv}$	$H_{loc,b3}=PS3\|\|G_{loc,b3}$
$G_{loc,c0}=G_{Abs}\|\|G_B$	$H_{loc,c0}=CPS0\|\|G_{loc,c0}$
$G_{loc,c1}=G_B\|\|G_D$	$H_{loc,c1}=CPS1\|\|G_{loc,c1}$

There are three groups of generic specifications corresponding to the desired behaviour of the system. The first group of specifications, shown in Fig. 10a, are called ordering specifications (OS) and manages the operation of subsystems in desired order. Depending upon the test result obtained by Cylinder C, Manipulator E or Cylinder D may be started following the end operation of Cylinder B. This choice is done by OS2 and OS5. The second group of specifications forces the manipulators to pick up the supplied component immediately and make the components ready for the next mode of operation, which is called insertion/placement operation. These specifications are called preparation specifications (PS) and are given in Fig. 10b. The last group of specifications shown in Fig. 10c prevents possible collision of two bases, which may occur in the test point or assembly point. These specifications are called collision protection specifications (CPS).

As a next step in the local modular control approach, local models and the automata that mark the local specifications of the system are built as in Table 2. The computations necessary for Table 2 and verification of controllability and modularity have been performed by UMDES-LIB which is a software package developed at the University of Michigan for performing operations on finite state machines (available at http://www.eecs.umich.edu/umdes).

It can be shown that each local specification is controllable with respect to its corresponding local system. Therefore, supervisors $S_{loc,x}$, x={a0,a1,a2,a3,a4,a5,a6,b0,b1,b2,c0,c1}, can be computed directly from their respective local specifications.

Local specifications are obtained by parallel composition of generic specifications and local system models. Therefore, in the realization stage, when generic specifications are considered as local supervisors and are operated in parallel with the corresponding local systems, it is guaranteed that the resulting behaviour will be equivalent to the behaviour determined by local specifications. Figure 11 shows three of the 12 local supervisors as an example. The labels appearing on some of the states of the supervisors correspond to enabling signals of the controllable events as defined in Section 4.2.

At this point, it might be interesting to calculate the supervisor that would be realized if a monolithic supervisor design approach had been taken. This monolithic supervisor would be obtained by calculating the parallel composition of the 12 local supervisors. Calculations by the software tool show that this monolithic supervisor would have 20992 states, and approximately 200000 transitions. This calculation shows that in many cases it might be impossible to design and realize a supervisory control application if a monolithic approach is taken.

To show that local modularity condition $\overline{\|_{j=1}^{m} L_m(S_{loc,j}/G_{loc,j})} = \|_{j=1}^{m} \overline{L_m(S_{loc,j}/G_{loc,j})}$ holds for the problem at hand, it is necessary to perform several operations on

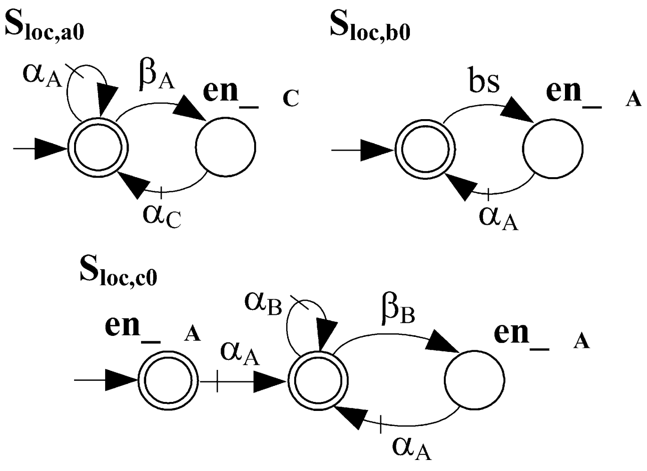


Fig. 11 Three of local supervisors

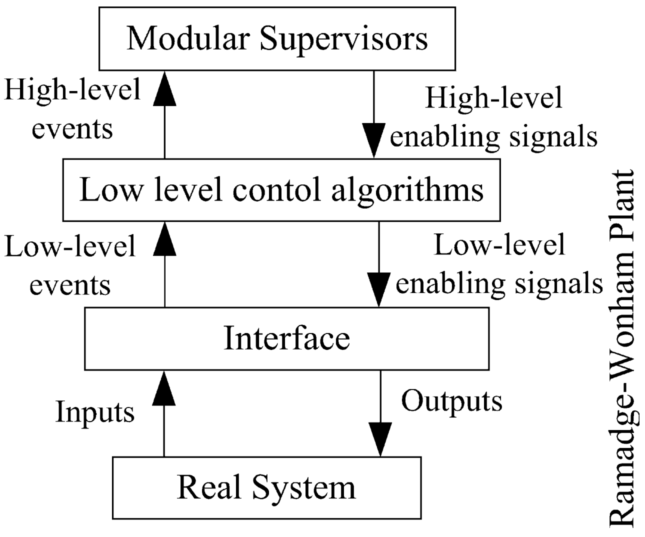


Fig. 12 Control system structure

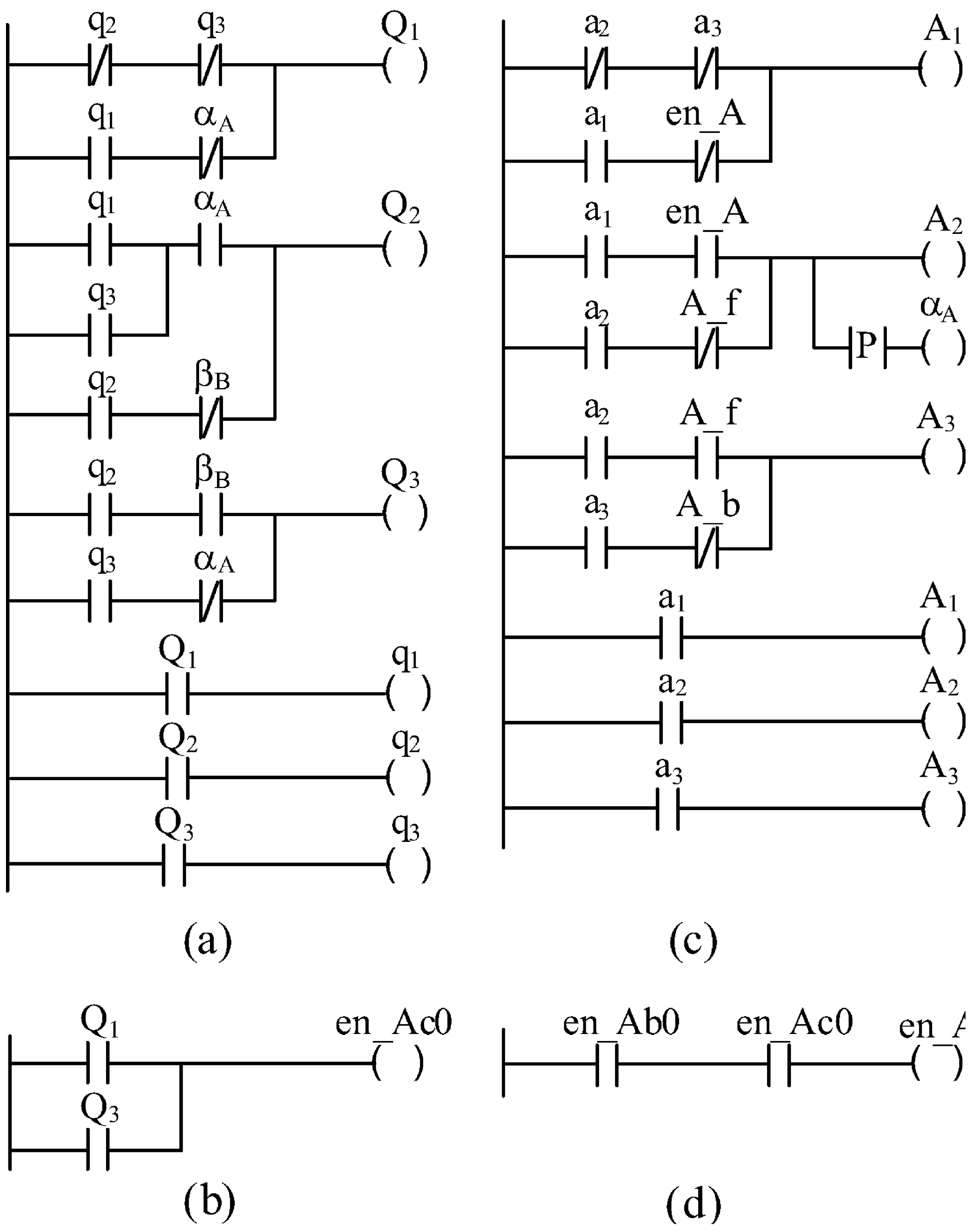


Fig. 13 PLC programs for modular supervisor $S_{loc,c0}$ and low-level control code for Cylinder A

automata corresponding to local supervisors and local system models. Local modularity condition is expressed based on marked languages. Therefore, all the calculations have been performed using the automata corresponding to these languages. For the computation of automata that correspond to local languages under the control of supervisors, the executable "par_comp.exe" of UMDES_LIB has been used.

For a nonblocking automaton, the prefix closure of the language marked by the automaton is equal to the language generated by the automaton. And for a coaccessible automaton, prefix closure of the marked language always equals the generated one, since a coaccessible automaton is always nonblocking [8]. This fact has been applied to calculate the prefix closures of the marked languages given in the modularity condition by using executable "co_acc.exe". The executable "par_comp.exe" is used to calculate the parallel compositions appearing in the modularity condition. As a last step "equiv.exe" of UMDES_LIB has been used to check if the modularity condition holds.

4.2 Implementation

In the implementation stage, we use the structure shown in Fig. 12. In this structure, modular supervisors constitute the highest level of hierarchy and they interact with the low-level control algorithms, which define the operational procedures of the real system. The task of low-level interface is to convert responses obtained as sensor signals into events and to convert low-level enabling signals into physical outputs that drive the real system.

Modular supervisors update their states when the high-level events occur. When modular supervisors update their states, they also update the enabling signal information sent to low-level control algorithms. Low-level control algorithms send high-level events corresponding to these enabling signals back to modular supervisors as soon as the enabling signals are processed by the plant. This feed back structure for high-level events ensures that modular supervisors update their states only if the plant processes the enabled signals. This is also compatible with SCT, which assumes that events are generated by the plant.

We will explain the realization methodology on supervisor $S_{loc,c0}$ shown in Fig. 11 using the methodology introduced in Section 3.2. Figure 13 shows the ladder diagrams related with the supervisor $S_{loc,c0}$. In Fig. 13a, state transition of $S_{loc,c0}$ is programmed. Notice that it is not necessary to program the self loop of the automaton. Figure 13b shows the LAD program which generates $S_{loc,c0}$'s enabling signal for α_A. The LAD given Fig. 13d is programmed to enable α_A if both of $S_{loc,b0}$ and $S_{loc,c0}$ enables it. Figure 13c shows the state transition program code of the low-level control algorithm for Cylinder A. As can be seen from the code, when transition to A_2 occurs by the enabling signal en_A, α_A is generated. Note that the events A_f and A_b correspond to forward and backward positions of cylinder A and are obtained in the interface by rising edges of sensor signals.

5 Conclusion

In this paper, a formal methodology which enables expressing a given state machine in logical domain has been introduced and applied for supervisory control of a pneumatic manufacturing system. We have used local modular approach, which exploits modular structure of the system and of the specifications. A hierarchical control structure using the presented implementation methodology is introduced. The introduced methodology constitutes a general solution for the problem of avalanche effect, enhances program readability and provides considerable memory savings in many cases. In particular, the approach used for the implementation methodology could be considered as a new definition of automata in a logical domain. Such a definition of automata would be very useful as it would skip the need of a translation to logical domain where realization generally takes place.

Acknowledgments This work was partially supported by SMC-ENTEK Company. The authors also thank Neslihan S. Sengör for her comments on the manuscript.

References

1. Ramadge PJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25 (1):206–230
2. Balemi S, Hoffmann GJ, Gyugyi P, Wong-Toi H, Franklin GF (1993) Supervisory control of a rapid thermal multiprocessor. IEEE Trans Automat Contr 38(7):1040–1059
3. Brandin BA (1996) The real-time supervisory control of an experimental manufacturing cell. IEEE Trans Robot Autom 12 (1):1–14
4. Leduc RJ (1996) PLC implementation of a DES supervisor for a manufacturing testbed: an implementation perspective. Master's thesis, University of Toronto, Toronto, Canada
5. Hasdemir İT, Kurtulan S, Gören L (2003) Supervisory control of a pneumatic system using PLC. Proc. third international conference on electrical and electronics engineering, ELECO, Bursa, Turkey
6. Hasdemir İT, Kurtulan S, Gören L (2004) Implementation of local modular supervisory control for a pneumatic system using PLC. Proc.7th Int. workshop on discrete event systems (WODES), Reims, France
7. Fabian M, Hellgren A (1998) PLC-based implementation of supervisory control for discrete event systems. Proc. 37th IEEE conference on decision & control, Tampa, Florida, USA
8. Cassandras GC, Lafortune S (1999) Introduction to discrete event systems. Kluwer Academic Publishers, Massachusetts, USA
9. IEC 61131-3 (2003) Programmable controllers-part 3: programming languages
10. Bolton W (1997) Programmable logic controllers: an introduction. Butterworth-Heinemann
11. Kurtulan S (2006) Industrial automated systems (In Turkish). Istanbul Technical University Press, Istanbul
12. Wonham WM, Ramadge PJ (1988) Modular supervisory control of DESs. Mathematics of control of discrete event systems 1 (1):13–30
13. Queiroz MH de, Cury JER (2000) Modular control of composed systems. Proc. ACC. Chicago, USA
14. Queiroz MH de, Cury JER (2000) Modular supervisory control of large scale discrete-event systems. Proc.5th Int. Workshop on Discrete Event Systems (WODES), Ghent, Belgium
15. Queiroz MH de, Cury JER (2002) Synthesis and implementation of local modular supervisory control for a manufacturing cell. Proc.6th Int. Workshop on Discrete Event Systems (WODES), Zaragoza, Spain