Embedded Supervisory Control of Discrete-Event Systems

Yue Yang

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

October 2005

Your file Votre référence
ISBN: 0-494-14287-1
Our file Notre référence
ISBN: 0-494-14287-1

ABSTRACT

Embedded Supervisory Control of Discrete-Event Systems

Yue Yang

In this work we propose to implement supervisory control by embedding control in the plant Finite State Machine (FSM). Supervisory control is introduced by extending the plant with boolean variables, guard formulas and updating functions. Boolean variables are used to encode the supervisor's states. Event observation is captured by a set of boolean functions that update the values of boolean variables, and control is introduced by guarding events with boolean formulas. The resulting Extended Finite State Machine (EFSM) implements the supervisory control map in the sense that the languages closed and marked by the EFSM are equal to those of the supervised system. After studying embedded supervisory control under partial observation, centralized and decentralized control architectures are analyzed. It is shown that the coobservability condition remains necessary and sufficient for the existence of decentralized supervisors. An application of our approach in the synthesis of communication protocols is presented.

Acknowledgments

First and foremost I would like to thank my supervisor Dr. Peyman Gohari for everything he has done for me through the years. Without doubt, this work was motivated and benefited from his keen mathematical insight, guidance and financial support. Besides, I am most grateful for his counsel on things unrelated to this work.

Special thanks go to people at Control and Robotics Group for their friendship and creating an atmosphere conducive to innovative research. Dr. Shahin Hashtrudi Zad taught me DES course, which has been profoundly used in my later research.

At last but certainly not the least I would like to give my deepest appreciation to my parents for their love and support, and also many thanks to my friends in Montreal.

YUE YANG

Concordia University

October 2005

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 RW supervisory control of DES

This work proposes to implement Ramadge and Wonham supervisory control by embedding a control mechanism in the plant Finite State Machine (FSM). Therefore, it is appropriate to start with a brief tour of RW supervisory control theory.

Supervisory Control Theory (SCT) proposed by Ramadge and Wonham [RW87], [WR87] has provided a systematic approach to control a Discrete-Event System (DES). A DES, called *plant*, is modeled as an FSM. The behavior of a DES is represented by a formal language L, where a string of L is a sequence of events executable by the system. The control task is to restrict the plant behavior by a *supervisor* such that it satisfies the specification of some desired behavior.

The alphabet Σ denotes the set of events over which L is defined. For Σ we have the partition $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, where the disjoint subsets Σ_c and Σ_u comprise respectively the *controllable* and *uncontrollable* events. The way a supervisor exercises closed-loop control over the plant is by disabling some of the events in order that the plant under control may achieve a certain legal behavior. Supervisors that do not prevent uncontrollable events from happening are called *admissible*. Thus, the behavior of the controlled system is a restriction, or a sublanguage, of L.

Since under the closed-loop control of an admissible supervisor all uncontrollable events that can occur in the plant can also occur in the closed-loop system, there may

not always exist an admissible supervisor for a given plant such that the closed-loop system satisfies the desired behavior. A question then arises as to how to construct an admissible supervisor which is *minimally restrictive*, in the sense defined in the next paragraph. A general notion of *controllability* is introduced and it is shown that a sublanguage K of the specification E can be implemented by some supervisory control if and only if it is controllable with respect to L.

Formally, the desired closed-loop behavior of the system is represented by some sublanguage of L. By the result just quoted, the specification cannot be exactly met if E is not controllable with respect to L, which informally means that it is possible to exit E through uncontrollable events defined in L. In such a case, one could settle for a best approximation, namely, the largest controllable (thus implementable) sublanguage of E, if it ever exists. It has been proven that the class of controllable sublanguages of E forms a complete upper subsemilattice of $(\Sigma^*, \subseteq)$ and therefore the supremal controllable sublanguage of E indeed exists. An effective algorithm for computation of the supremal controllable sublanguage has been presented when the languages E and L are both regular, i.e. they can be generated by FSMs.

1.2 Motivation

In this section we discuss the motivation of this research. We want to show why EFSM is a good discrete-event model and a powerful tool for supervisory control.

Ramadge and Wonham's SCT provides a unifying framework for the control problem of discrete-event systems. Given a plant G and a specification E modeled as FSM, it is desired to find a supervisor (or a supervisory control map) S such that plant under supervision, denoted by S/G, satisfies the specification. Thus, as shown in Fig.1.1-a, from a control specialist's vantage point there is a clear separation between the plant and the controller (supervisor).

However, in reality, oftentimes the partition of the plant and the controller is blurred. Rather, the "closed-loop" or "controlled" system is designed at once, which is later tested against the specification of some desired behavior for compliance.

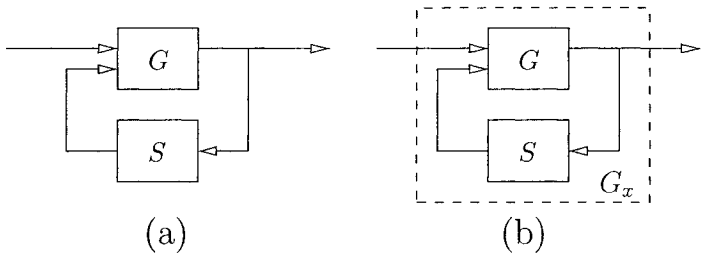

Figure 1.1: (a) Traditional supervisory control (b) Embedded supervisory control.

The implementation issues of SCT have not received the attention they deserve. Although several software tools such as TTCT [TT05] and UMDES [UM05] have been developed to automate supervisory control design procedures, their in-out interface are still unhandy and fallible. For example, when the transition structure is large, it would be a significant task for users to input the system parameters. This problem can be largely attributed to the underlying FSM model which does not allow an effective way to represent the control information that is enforced on the plant by the controller. A possible way to overcome this problem is to equip the supervisory control with boolean operations, which would largely facilitate computer implementation as it is much easier to code variables, update them and use their combination as a condition than to deal with automata directly. In addition, when a change in the system model becomes necessary (for example as a result of updates during the system maintenance), in many cases the underlying transition structure remains unchanged while only the guard formulas and updating functions need be changed.

In this work we attempt to bridge the gap between SCT and its computer implementation. While we use supervisory control theory to design a supervisor, we use a novel approach to implement the supervisor as an embedded part of the system to be controlled. The result, denoted by G_x and shown in Figure.1.1-b, is an economical way to represent the closed-loop system, and is readily identifiable with traditional designs to discrete-event control problems. We model G_x as an Extended Finite State Machine (EFSM). The main advantages of our framework are listed below.

1. Having equal expressive power with regular FSM (we will show this in Chapter

4), the EFSM offers a more compact representation of the closed-loop system. The controller has been embedded into the system to be controlled. This feature is commonly used in communication protocols, where control mechanism needs to be attached to each process since external controllers are costly and inefficient.

2. Compared with traditional plant-supervisor closed-loop system, the EFSM model is easier to program. Control information is encoded as formulas defined over boolean variables, therefore implementation becomes easier to some extent, and it is flexible to modify the code.

3. EFSM is closer to input/output models, in the sense that a controller "command is encoded by a guard formula, while a plant "response is encoded by a set of updating functions. Thus, for example, EFSM models can be readily translated to PLC implementations, which are used in many industrial applications (where boolean variables are used to encode supervisor states and events).

1.3 Related work

Holtzmann [HO91] has defined the EFSM model as an augmentation of the traditional finite state machine model. Variables are introduced to hold abstract objects (messages). They hold only one value at a time, selected from a finite range of possible values. Transition rules of an extended finite state machine have two parts: a condition and an effect. The former is generalized to include boolean expressions over the values of variables, and the latter (i.e. the actions) is generalized to include assignments to variables. Expressions are built from variables and constants with the usual arithmetic and relational operators, while a single assignment can change the value of only one variable. This work is at the basis of the EFSM model defined in this thesis.

Previous work with EFSM includes [CK96], [LW02], [HC98] and [CL00]. In the field of Application Specific Integrated Circuit (ASIC), [CK96] presents the notation

of a formal EFSM model as a 5-tuple, and uses a graph to represent an EFSM. A method is proposed to automatically transform the high-level description of a circuit in VHDL or C into an EFSM model that is used to generate functional vectors. [LW02] uses EFSM models introduced in [HC98] to extract a set of functional constraints, with which the constrained path classification is able to identify the set of functionally testable paths inside the ASIC design.

In the area of supervisory control of discrete-event systems, Chen and Lin [CL00] have presented their work on controller synthesis for Finite State Machines with Parameters (FSMwP) introduced in [YF00]. FSMwP is an extension of a regular FSM in which provisions have been made to capture the notions of event disablement and enforcement. Note that the definition of their FSMwP is similar to the EFSM described in [CK96]. Nevertheless, the EFSM mechanism in [CK96] was developed for verification of circuits, while the general discrete-event systems modeled in FSMwP framework can represent efficiently systems that cannot normally be represented by regular finite state machines without arbitrarily large state spaces.

One of the objectives of [CL00] is to use parameters to control the system efficiently. The authors have introduced guards that are predicates over parameters. In FSMwP models, transitions can be guarded by inequations, which largely mitigates the state explosion problem. In addition, [CL00] has also introduced an external parameter, called *global time* t, and presented a set of online safety control synthesis procedures based on the limited/variable lookahead policies to address the practical concerns of real-time implementation. However, controller design is complicated by the iterative nature of their algorithm. In contrast, we offer a simpler and more understandable approach to design a controller.

A study that motivated us to embark on the current research project is carried out by Gohari in [PG04]. EFSMs are used to formalize the age-old alternating bit protocol which is used in reliable transmission of files over half-duplex channels. In his model transitions are guarded and may result in taking actions. A set X of boolean variables are defined. A guard is specified as a boolean formula over X. A transition can be taken (enabled) if and only if its guard evaluates to *true* (1). Thus, guards

can be viewed as a mechanism for controlling DES. We attempt to formalize his ideas by formally defining notions of guards and updating actions. An exact formulation of the control problem in the supervisory control framework of Ramadge and Wonham [RW87] will be sought.

1.4 Organization of the thesis

The rest of this thesis is organized as follows: In Chapter 2 we study the mathematical preliminaries, and the syntax and semantics of DES are introduced. Chapter 3 briefly reviews the RW Supervisory Control Theory (SCT). A general model of EFSM is defined in Chapter 4 and their synchronous product is defined as well. In Chapter 5, an approach for implementing a supervisory control map by an EFSM is introduced, and two applications of our approach are presented. In Chapter 6, we show how the design can be generalized when the observation of plant by the supervisor is partial. Chapter 7 concludes the thesis.

Chapter 2

Mathematical Preliminaries

2.1 Introduction

In this chapter, mathematical preliminaries of our approach on embedded supervisory control by extended finite state machines will be reviewed. Since the variables used in controller are boolean, the concept and basic properties of boolean algebra play an important role in our framework. The fundamental elements of boolean logic are discussed Section 2.2. Section 2.3 reviews the basics of language and automata theory.

2.2 Boolean algebras

Let X be a set of variables over $\mathbb{B}$. A *literal* is a boolean variable or its complement. A *maxterm* is the disjunction (*or*) of a collection of literals. A *minterm* is the conjunction (*and*) of a collection of literals. A *boolean formula* is an expression in which boolean variables are combined by recursively applying boolean operations *and* (conjunction), *or* (disjunction), and *not* (complement). A *boolean function* is a function from domain $\mathbb{B}^k$ to codomain $\mathbb{B}$, where k is the number of variables in domain. It assigns a new boolean value to a variable based on the current values of all k variables. Thus we can also specify a boolean function by writing out a *truth table*, which is a table listing all possible assignments of truth values to the variables and the resulting output from the function.

Definition 2.2.1 *(n-Minterm, bit combination)* An *n-Minterm* is a minterm consisting of n literals. The corresponding bit combination for each n-Minterm is the only bit combination for which the minterm evaluates to 1. □

Example 2.1: If x_1, x_2 and x_3 are the boolean variables, the 3-Minterms are: $\bar{x}_1\bar{x}_2\bar{x}_3, \bar{x}_1\bar{x}_2x_3, \bar{x}_1x_2\bar{x}_3, \bar{x}_1x_2x_3, x_1\bar{x}_2\bar{x}_3, x_1\bar{x}_2x_3, x_1x_2\bar{x}_3, x_1x_2x_3$. The corresponding bit combinations are shown in Tab.2.1.

Table 2.1: 3-Minterm and bit combination.

3-Minterm	Designation	Bit Combination
$\bar{x}_1\bar{x}_2\bar{x}_3$	m0	000
$\bar{x}_1\bar{x}_2x_3$	m1	001
$\bar{x}_1x_2\bar{x}_3$	m2	010
$\bar{x}_1x_2x_3$	m3	011
$x_1\bar{x}_2\bar{x}_3$	m4	100
$x_1\bar{x}_2x_3$	m5	101
$x_1x_2\bar{x}_3$	m6	110
$x_1x_2x_3$	m7	111

◇

Any boolean function can be written using literals and boolean operations. There are two standard forms, called *Disjunctive Normal Form* (DNF) and *Conjunctive Normal Form* (CNF), which are particularly useful. If a boolean expression is a conjunction of maxterms then it is said to be in *conjunctive normal form*, and if it is a disjunction of minterms then it is said to be in *disjunctive normal form.*

Example 2.2: Let $X = \{x_1, x_2\}$. The boolean expression $\bar{x}_1x_2 + x_1\bar{x}_2$ is in disjunctive normal form (it is an *or* of two minterms); the boolean expression $(\bar{x}_1 + \bar{x}_2)(x_1 + x_2)$ is in conjunctive normal form (it is an *and* of two maxterms). ◇

Definition 2.2.2 *(An n-Minterm set)* An *n-Minterm set* is a set including all

n-Minterms, denoted by M_n. □

Example 2.3: The set of all 2-Minterms is M_2=$\{x_1x_2, \bar{x}_1x_2, x_1\bar{x}_2, \bar{x}_1\bar{x}_2\}$. ◇

2.3 Languages and automata

2.3.1 Languages

Let Σ be a finite set of distinct symbols. We refer to Σ as an *alphabet.* Let Σ^+ denote the set of all finite sequences and ϵ denote the empty sequence (sequence with no symbols), where $\epsilon \notin \Sigma$. We then write:

$$\Sigma^* := \{\epsilon\} \cup \Sigma^+$$

An element of Σ^* is a *string* or *word* over the alphabet Σ; ϵ is the *empty string.*

Let $s, t \in \Sigma^*$. The catenation operation between two strings

$$cat : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$$

is defined according to

1. $cat(\epsilon, s) = cat(s, \epsilon) = s$;
2. $cat(s, t) = st$.

Any string in Σ^* can be generated by catenating event labels from Σ. The *length* $\mid s \mid$ of a string $s \in \Sigma^*$ is defined according to

1. $\mid \epsilon \mid = 0$,
2. $\mid s \mid = k$, if $s = \sigma_1\sigma_2 \ldots \sigma_k$, $\sigma_i \in \Sigma$, $i = 1, 2, \ldots, k$.

Thus $\mid cat(s, t) \mid = \mid s \mid + \mid t \mid$, $s, t \in \Sigma^*$. We conclude this subsection with some terminology about strings. If $tuv = s$ with $t, u, v \in \Sigma^*$, then

- t is called a *prefix* of s,
- u is called a *substring* of s,
- v is called a *suffix* of s.

Observe that both ϵ and s are prefixes, substrings and suffixes of s.

2.3.2 Automata

In discrete-event systems, an automaton is a device that is capable of representing a language that describes the behavior of a system or its specification.

An automaton over the alphabet Σ is a 5-tuple,

$$G = (Q, \Sigma, \delta, q_0, Q_m)$$

where Q is a nonempty set of states, $q_0 \in Q$ is the *initial state*, $Q_m \subseteq Q$ is the subset of *marker states*, and δ is a *partial state transition function* $\delta : Q \times \Sigma \rightarrow Q$: $\delta(q, \sigma) = q'$ means that there is a transition labeled by event σ from state q to state q'. We extend δ from $Q \times \Sigma$ to $Q \times \Sigma^*$:

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q$$

in the following recursive manner. Let $q \in Q$, $\sigma \in \Sigma$ and $s \in \Sigma^*$,

- $\hat{\delta}(q, \epsilon) = q$,
- $\hat{\delta}(q, \sigma) = \delta(q, \sigma)$,
- $\hat{\delta}(q, s\sigma) = \hat{\delta}(\hat{\delta}(q, s), \sigma)$.

In the rest of this thesis, we omit the $\hat{}$ and write δ in place of $\hat{\delta}$. G is characterized by two subsets of Σ^* called the *closed behavior* of G, written as $L(G)$, and the *marked behavior* of G, written as $L_m(G)$. The closed language is defined as:

$$L(G) := \{s \in \Sigma^* \mid \delta(q_0, s) \in Q\}$$

and is interpreted to mean the set of all possible event sequences which the plant may generate. The marked language is defined as:

$$L_m(G) := \{s \in \Sigma^* \mid \delta(q_0, s) \in Q_m\}$$

and is intended to distinguish the subset of closed behavior that represents completed tasks.

Note that $\emptyset \subseteq L_m(G) \subseteq L(G)$, and always $\epsilon \in L(G)$ (provided $G \neq EMPTY$, the DES with empty state set). The *reachable (state) subset* of G is

$$Q_r = \{q \in Q | (\exists s \in \Sigma^*)\delta(q_0, s) = q\};$$

G is *reachable* if $Q_r = Q$. The *coreachable subset* is

$$Q_{cr} = \{q \in Q | (\exists s \in \Sigma^*)\delta(q, s) \in Q_m\};$$

G is *coreachable* if $Q_{cr} = Q$. G is *trim* if it is both reachable and coreachable.

2.4 Synchronous product on automata

Let $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, where in general $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$. Define a natural projection

$$P_i : \Sigma^* \to \Sigma_i^* \quad (i = 1, 2)$$

according to

1. $P_i(\epsilon) = \epsilon$;
2. $P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases}$;
3. $P_i(s\sigma) = P_i(s)P_i(\sigma)$, $s \in \Sigma^*$, $\sigma \in \Sigma$.

P_i is *catenative* since $P_i(st) = P_i(s)P_i(t)$. The action of P_i on a string s is just to erase all occurrences of σ in s where $\sigma \notin \Sigma_i$. P_i is called the *natural projection* of Σ^* onto Σ_i^*. Let

$$P_i^{-1} : Pwr(\Sigma_i^*) \to Pwr(\Sigma^*)$$

be the inverse image function of P_i, where for $K \subseteq \Sigma_i^*$,

$$P_i^{-1}(K) := \{s \in \Sigma^* \mid P_i(s) \in K\}$$

For $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the synchronous product $L_1 \parallel L_2 \subseteq \Sigma^*$ is defined according to

$$L_1 \parallel L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$$

Thus, $s \in L_1 \parallel L_2$ if and only if $P_1(s) \in L_1$ and $P_2(s) \in L_2$. If $G_1 = (Q_1, \Sigma_1, \delta_1, q_{10}, Q_{1m})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{20}, Q_{2m})$ are recognizers for L_1 and L_2, respectively, then the synchronous product of G_1 and G_2 is the automaton

$$G_1 \parallel G_2 := Ac(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{10}, q_{20}), Q_{1m} \times Q_{2m})$$

where for $(q_1, q_2) \in Q_1 \times Q_2$ and $\sigma \in \Sigma$,

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \delta_2(q_2, \sigma)! \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \sigma \notin \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \delta_2(q_2, \sigma)! \text{ and } \sigma \notin \Sigma_1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The automaton $G_1 \parallel G_2$ satisfies the following equations:

$$L(G_1 \parallel G_2) = L(G_1) \parallel L(G_2) \tag{2.1}$$

$$L_m(G_1 \parallel G_2) = L_m(G_1) \parallel L_m(G_2) \tag{2.2}$$

Chapter 3

Background Review

3.1 Introduction

This work is built on the the supervisory control framework for discrete-event systems developed by Ramadge and Wonham [RW87], [WR87], and decentralized supervisory control presented by Rudie and Wonham [RW92]. Therefore it would be appropriate to review the basics of RW supervisory control theory and the decentralized supervisory control theory. We first review Finite State Machines (FSM) by which plants and supervisors in this work are modeled. Then two principal control architectures, *centralized supervisory control* and *decentralized supervisory control*, are discussed. In centralized supervisory control the overall plant is supervised by one supervisor, while in the decentralized case local supervisors are synthesized for local plants to achieve a global control objective. The partial observation of plant events by a supervisor is also considered in the former.

3.2 FSM model

The plant to be controlled is modeled by an automaton $G = (Q, \Sigma, \delta, q_0, Q_m)$ where Q is a set of states, Σ is an alphabet of event labels, $q_0 \in Q$ is the initial state, $Q_m \in Q$ is the set of marker states, and $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function defined at each state in Q for a subset of Σ. For the case where Q is finite, G can

be represented by a state-transition diagram whose nodes are states and whose edges are transitions defined by δ. On the transition diagram, the initial state is identified by an incoming arrow and marker states are identified by outgoing arrows. The set Σ is the set of all edge labels on the diagram.

The *closure* $\overline{K}$ of a language $K \in \Sigma^*$ is the set of all prefixes of strings in K. K is *closed* if $K = \overline{K}$, and K is $L_m(G)$ – *closed* if $K = \overline{K} \cap L_m(G)$. By definition, $L_m(G) \subseteq L(G)$.

3.3 Supervisory control of DES

3.3.1 Centralized supervisory control

In RW supervisory control theory a plant to be controlled, as a generator of a formal language, may generate strings that are illegal, or may cause harm to the system operation. By adjoining a supervisor (controller), it will be possible to force the language generated by the plant to an acceptable region. The desired performance of such a controlled generator will be specified by requiring that its generated language must be contained in some specification languages. It is often possible to meet this specification in an 'optimal', that is, minimally restrictive, fashion. The control problem will be considered fully solved when a controller that forces the specification to be met has been shown to exist and to be constructible. The one-plant-one-supervisor (*centralized*) 'architecture' is sketched below in Fig. 3.1-(a), while Fig. 3.1-(b) shows when the plant consists of several parallel components.

Controllability and supervision

To impose supervision on the plant, we identify some events as *controllable* and some as *uncontrollable*, thereby partitioning Σ into the disjoint sets Σ_c, the set of controllable events, and Σ_u, the set of uncontrollable events. A particular subset of events to be enabled can be selected by specifying a subset of controllable events. It is convenient to adjoin with this all the uncontrollable events as they are never

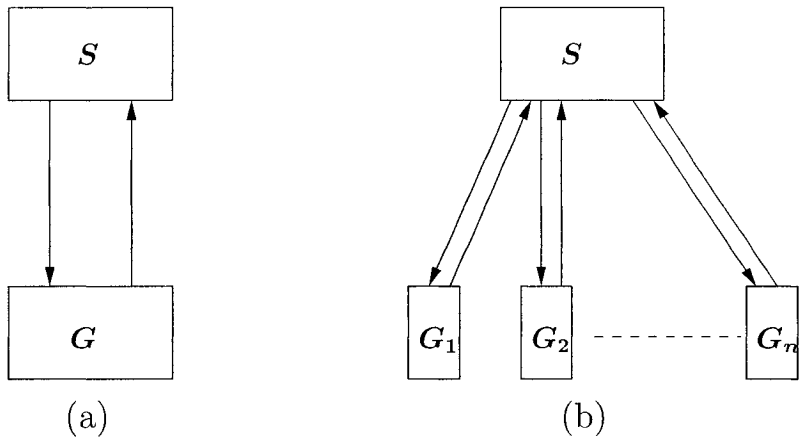

Figure 3.1: The architecture of centralized supervisory control.

disabled. Each such subset of events is a control pattern, and we introduce the set of all control patterns

$$\Gamma = \{\gamma \in Pwr(\Sigma) \mid \gamma \supseteq \Sigma_u\}.$$

We denote by G the overall plant to be controlled. A supervisor is an agent which observes a sequence of events as it is generated by G and enables or disables any of the controllable events at any point in time throughout its evolution. By performing such a manipulation on controllable events, the supervisor ensures that only a subset of $L(G)$ is permitted to be generated. Formally, a supervisor is a map $\mathcal{V} : L(G) \to \Gamma$. The behavior of the closed-loop system is represented by $\mathcal{V}/G$, to suggest 'G under the supervision of $\mathcal{V}$'. The closed behavior of $\mathcal{V}/G$ is defined to be the language $L(\mathcal{V}/G) \subseteq L(G)$ as follows:

1. $\epsilon \in L(\mathcal{V}/G)$;

2. If $s \in L(\mathcal{V}/G)$, $s\sigma \in L(G)$ and $\sigma \in \mathcal{V}(s)$, then $s\sigma \in L(\mathcal{V}/G)$;

3. No other strings belong to $L(\mathcal{V}/G)$.

Clearly $L(\mathcal{V}/G)$ is nonempty and closed. When the marker states of the controlled system are decided by the plant, i.e., all the states in the supervisor are marked, the marked behavior of $\mathcal{V}/G$ is

$$L_m(\mathcal{V}/G) = L(\mathcal{V}/G) \cap L_m(G).$$

In addition, when correctly designed, a supervisor must guarantee that the closed-loop system is nonblocking, i.e., that every string generated by the closed-loop system can be completed to a marker state. This requirement is expressed as follows: a supervisor $\mathcal{V}$ is *proper* for G if

$$\overline{L}_m(\mathcal{V}/G) = L(\mathcal{V}/G)$$

where the overbar notation denotes prefix closure.

To characterize those languages that qualify as the marked behavior of some supervisory control $\mathcal{V}$, a language $K \in L(G)$ is said to be *controllable* with respect to G if

$$\overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}.$$

For illustration, $\overline{K}$ might be viewed as a specification of some "legal behavior". Controllability requires that if s is legal, σ is uncontrollable, and $s\sigma$ is physically possible, then $s\sigma$ must be legal as well. In the rest of this thesis, we denote by S the automaton that implements $\mathcal{V}$.

An optimal (i.e. minimally restrictive) proper supervisor S for G subject to $L_m(S/G) \subseteq K$ can be obtained by the TTCT procedure `supcon` that computes a trim representation of supervisor S according to $S = \texttt{supcon}(G, K)$.

Observability

So far, we have assumed that a supervisor can observe and record all events generated by the plant. A more realistic problem of supervisory control is that only a subset of event labels generated by the plant can actually be observed by the supervisor. The events visible to S form a subset of *observable* events, denoted by Σ_o, of the alphabet Σ. Note that, S can potentially disable controllable events that are not observable, namely $\Sigma_c - \Sigma_o$ need not be empty. The subset Σ_o in general need not have any particular relation to the subset of controllable events Σ_c. However, in this chapter, we assume that each event is either controllable or observable, which

makes our formulation more realistic. All other events in the world—that can neither be controlled nor observed by the supervisor— are irrelevant and therefore will be implicitly self-looped in every state of the plant. The sets of uncontrollable and unobservable events are denoted by $\Sigma_{uc} = \Sigma - \Sigma_c$ and $\Sigma_{uo} = \Sigma - \Sigma_o$, respectively. To represent the fact that a supervisor has only a partial observation of strings in $L(G)$, $P_o : \Sigma^* \rightarrow \Sigma_o^*$, the natural projection operator defined in Section.2.4, is used. Thus the effect of P_o on a string s is just to erase from s the events that do not belong to Σ_o, leaving the order of Σ_o-events in s unchanged. A supervisor under partial observation is a map $S_p : P_o[L(G)] \rightarrow 2^{\Sigma}$ such that $S_p[P_o(s)] \supseteq \Sigma_{uc}$ for any $s \in L(G)$.

In their work [FW88], Lin and Wonham defined the observability of a language K as follows: Let $K \subseteq L_m(G)$. The language K is said to be observable with respect to $(L(G), P)$ if and only if, for all $s, s' \in \Sigma^*$ if $P_o(s) = P_o(s') \Rightarrow$

i) $(\forall \sigma \in \Sigma) s\sigma \in \overline{K} \wedge s' \in \overline{K} \wedge s'\sigma \in L(G) \Rightarrow s'\sigma \in \overline{K}$;

ii) $s \in K \wedge s' \in \overline{K} \cap L_m(G) \Rightarrow s' \in K$.

In words, observability requires that if two strings look the same to a supervisor, they must be consistent with respect to one-step continuations in K. The second condition ensures that the decision as to whether mark a string generated by K can be unambiguously made on the basis of an observer's view of a string in $L(G)$. The observability condition together with the controllability condition is necessary and sufficient for the existence of a supervisor under partial observation as shown in following theorem and corollary presented in [FW88].

Theorem 3.3.1 Let $K \in L_m(G)$ be a nonempty language. There exists a non-blocking supervisor S_p such that $L_m(S_p/G) = K$ if and only if the following three conditions are all satisfied.

1. K is controllable with respect to $L(G)$;
2. K is observable with respect to $L(G)$; and
3. K is $L_m(G)$-closed. □

If we are only interested in the closed behavior, then we have the following corollary.

Corollary 3.3.1 Let $K \in L(G)$ be a nonempty language. There exists a nonblocking supervisor S_p such that $L(S_p/G) = K$ if and only if the following three conditions are all satisfied.

1. K is controllable with respect to $L(G)$;
2. K is observable with respect to $L(G)$; and
3. K is closed. □

3.3.2 Decentralized supervisory control

The control problem modeled in Section. 3.3.1 describe those situations where an acceptable solution details the actions that a single supervisor must take. However, in distributed systems where plant components are geographically widely separated, a centralized supervisor satisfying the global control objectives cannot be designed; rather we need a *decentralized* solution, i.e., a set of local supervisors, each is designed to monitor and control a plant component. A supervisor acting on all controllable events in the entire event set is called a *global* supervisor; in contrast, a supervisor that can only monitor and control subsets of events pertaining to a component is said to be *local*. A decentralized solution prescribes the control action that each local supervisor must take. The architecture is sketched in Fig. 3.2.

In many practical fields, such as communication networks, local specifications are not suitable for modeling a control problem; rather, specifications are given as a global requirement. That is, a problem statement describes what goal the network as a whole must achieve without spelling out what each agent in the network must do to achieve the global control objective.

Two decentralized control problems, called *GP* (Global Problem) and *GPZT* (Global Problem with Zero Tolerance), are described by Rudie and Wonham in [RW92], in which a global specification is to be satisfied by a set of local controllers.

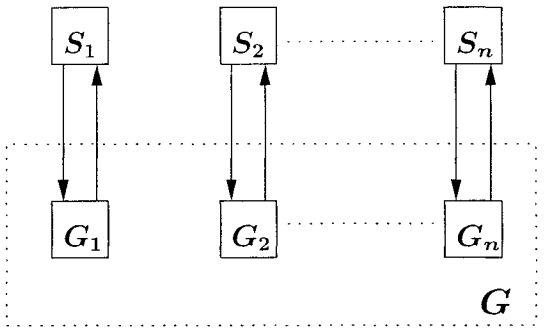

Figure 3.2: The architecture of decentralized supervisory control.

Global Problem (GP) Given a plant G over an alphabet Σ, a language K such that $\emptyset \neq K \subseteq L_m(G)$, a minimally adequate language $A \subseteq K$, and sets Σ_{c1}, Σ_{c2}, Σ_{o1}, $\Sigma_{o2} \subseteq \Sigma$, construct local admissible supervisors S_{p1} and S_{p2} such that $\tilde{S}_{p1} \wedge \tilde{S}_{p2}$ is a proper supervisor for G and

$$A \subseteq L(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G) \subseteq K.$$

Here, for $i = 1, 2$, supervisor S_{pi} can only observe events in Σ_{oi} and can only control events in Σ_{ci}. $\tilde{S}_{pi}$ denotes the supervisor which takes the same control decision as S_{pi} on an event in Σ_{ci}, enables all events in $\Sigma \backslash \Sigma_{ci}$, makes the same transitions as S_{pi} on Σ_{oi} and stays in the same state for events in $\Sigma \backslash \Sigma_{oi}$. $\tilde{S}_{pi}$, the global extension of S_{pi}, can be obtained from S_{pi} by adding $\Sigma \backslash \Sigma_i$ selfloops in its every state.

In the special case where $A = K$, GP reduces to GPZT.

Global Problem with Zero Tolerance (GPZT) Given a plant G over an alphabet Σ, a language K such that $\emptyset \neq K \subseteq L_m(G)$, and sets Σ_{c1}, Σ_{c2}, Σ_{o1}, $\Sigma_{o2} \subseteq \Sigma$, construct local admissible supervisors S_{p1} and S_{p2} such that $\tilde{S}_{p1} \wedge \tilde{S}_{p2}$ is a proper supervisor for G and

$$L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G) = K.$$

Notice that the language K need not be prefix-closed. The solvability of GPZT is related to the *coobservability* property of the specification language.

For simplicity, we present all decentralized problem with *two* supervisors, and all results can be generalized to any fixed number of supervisors.

The key to solving both GP and GPZT is a property called *coobservability*. Theorem 4.1 in [RW92] states that there exist supervisors S_{p1} and S_{p2} that solve GPZT if and only if K is controllable w.r.t. G and coobservable w.r.t. G, P_{o1}, and P_{o2}, where P_{oi}, $i = 1, 2$, stands for the projection from Σ^* to Σ_{oi}^*. In the rest of this work, we assume that the K is always controllable w.r.t. G.

The definition of coobservability in [RW92] is written as the conjuction of several instances of two relations. Fix an alphabet Σ, a plant G, and a language $K \subseteq L_m(G)$. The relation *nextact* is defined as follows:

$(\forall \sigma \in \Sigma, s, s' \in \Sigma^*)(s, \sigma, s') \in nextact_K$ if $s'\sigma \in \bar{K} \wedge s \in \bar{K} \wedge s\sigma \in L(G) \Rightarrow s\sigma \in \bar{K}$.

Informally, for any two strings s and s' and $\sigma \in \Sigma$, (s, σ, s') are related according to *nextact* if the supervisor's decision as to whether disable or enable σ after the occurrence of s' forces the same decision upon the occurrence of s, provided the plant permits such an action to be taken.

The relation *markact* is defined as follows:

$(\forall s, s' \in \Sigma^*)(s, s') \in markact_K$ if $s' \in K \wedge s \in \bar{K} \cap L_m(G) \Rightarrow s \in K$.

Informally, for any two strings s and s', (s, σ, s') are related according to *markact* if the supervisor's decision as to whether mark a permissible s is based on whether s' is marked, provided the plant permits the marking of s.

Now we define the notion of coobservability. A language K is said to be *coobservable* with respect to G, P_{o1} and P_{o2} if

$(\forall s, s', s'' \in \Sigma^*) P_{o1}(s) = P_{o1}(s') \wedge P_{o2}(s) = P_{o2}(s'') \Rightarrow$

$(\forall \sigma \in \Sigma_{c1} \cap \Sigma_{c2})[(s, \sigma, s') \in nextact_K \vee (s, \sigma, s'') \in nextact_K]$ **conjunct 1**

$\wedge (\forall \sigma \in \Sigma_{c1} \setminus \Sigma_{c2})[(s, \sigma, s') \in nextact_K]$ **conjunct 2**

$\wedge (\forall \sigma \in \Sigma_{c2} \setminus \Sigma_{c1})[(s, \sigma, s'') \in nextact_K]$ **conjunct 3**

$\wedge [(s, s') \in markact_K \vee (s, s'') \in markact_K]$ **conjunct 4**

In plain English coobservability states that if an event generated by the plant leads the system to illegal behavior, then at least one of the two supervisors must have enough information to disable it. On the other hand, if the event does not lead the sequence generated so far into illegal behavior, then neither of the supervisors should disable it. Finally, the decision as to whether or not mark an ambiguous string can

be determined by at least one of the supervisors.

Theorem 3.3.2 There exist supervisors S_{p1} and S_{p2} that solve GPZT if and only if

1. K is controllable w.r.t G;

2. K is coobservable w.r.t G, P_{o1}, P_{o2}. □

Chapter 4

Extended Finite State Machines

4.1 Introduction

This chapter describes a general model for Extended Finite State Machines (EFSMs) as well as their synchronous product.

An EFSM is an augmentation of a regular finite state machine. A set X of k boolean variables is defined. A transition is enabled if and only if its *guard formula*, which is a predicate defined as a boolean formula over X, is *true* (1). When a transition is taken, several *updating actions* may follow. An updating action is a boolean function that reassigns a new value to a variable based on the old values of all variables. Since there are k variables, each transition may trigger up to k actions.

An EFSM generates a closed language and a marked language, in the same way as a regular FSM does. In addition, we show that they have equal expressive power on representing languages. The synchronous product of two EFSMs is defined in this chapter, and a precondition under which the synchronous product exists is also discussed.

4.2 EFSM Model

In the following definition, let $\mathcal{G}$ denote the set of all boolean formulas over X, and $\mathcal{A}$ denote the set of all boolean functions $\mathbb{B}^k \rightarrow \mathbb{B}$.

Definition 4.2.1 (Boolean Extended finite-state machine) A boolean EFSM, denoted by G_x, is an eight-tuple

$$G_x = (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$$

where:

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function: $\delta(q, \sigma) = q'$, means that there is a transition labeled with event σ from state q to state q';
- q_0 is the initial state;
- $Q_m \subseteq Q$ is the set of marker states;
- X is a finite set of k boolean variables;
- $g : \Sigma \rightarrow \mathcal{G}$ is a *guard* formula;
- $A : \Sigma \rightarrow \mathcal{A}^k$ is a k-tuple of updating functions, where $k = |X|$. □

We make the following remarks about this definition.

- Employing a finite set of boolean variables does not enhance the expressive power of our models. Rather, it serves our ultimate goal of implementing supervisory control on systems modeled by FSM. For simplicity, we write EFSM in place of boolean EFSM in the rest of this thesis.
- We assume that all EFSMs in this thesis are non-empty.

- For the sake of convenience, δ is extended from domain $Q \times \Sigma$ to domain $Q \times \Sigma^*$ in the following recursive manner:

  1. $\delta(q, \epsilon) := q$;
  2. $\delta(q, s\sigma) := \delta(\delta(q, s), \sigma)$ for $s \in \Sigma^*$ and $\sigma \in \Sigma$.

- Boolean variable set $X=\{x_1, x_2, \ldots, x_k\}$ is a set of k boolean variables over which the guard formulas and updating functions are defined. All variables are initialized to *false* (0).

- Guard formula $g : \Sigma \rightarrow \mathcal{G}$. For $\alpha \in \Sigma$, g_α is a boolean formula with which all transitions labeled with α are guarded. In an EFSM, a transition labeled with an event is allowed to happen if and only if its guard formula evaluated at the current values of the boolean variables, returns *true*.

- Updating function $A : \Sigma \rightarrow \mathcal{A}^k$. For $\alpha \in \Sigma$, A_α is a k-tuple

$$A_\alpha = (a_\alpha^x)_{x \in X}$$

  where $a_\alpha^x : \mathbb{B}^k \rightarrow \mathbb{B}$ is a boolean function. When α is taken, it results in assignments $x := a_\alpha^x(\mathbf{v})$ for all $x \in X$, where $\mathbf{v} = (v_1, v_2, \ldots, v_k)$ are the current values of all variables in X. Notice that the updating function A_α updates boolean variables with values from domain $\mathbb{B}^k$ with values from codomain $\mathbb{B}^k$. We can extend the definition of the updating function from $A : \Sigma \rightarrow \mathcal{A}^k$ to $A : \Sigma^* \rightarrow \mathcal{A}^k$. The updating function

$$A_s = (a_s^x)_{x \in X},\ s \in \Sigma^*$$

  updates the values of boolean variables when the string s is taken, resulting in assignments $x := a_s^x(\mathbf{v})$ for all $x \in X$. A_s is defined in the following recursive manner:

  1. $A_\epsilon(\mathbf{v}) := \left(\underbrace{0, 0, \ldots, 0}_{k}\right)$;
  2. $A_{u\sigma} := A_\sigma(A_u(\mathbf{v}))$ for $u \in \Sigma^*$ and $\sigma \in \Sigma$.

4.2.1 Languages

In this section we formally define languages as a framework for studying the behavior of EFSM. A nonempty EFSM generates a closed language and a marked language over the alphabet Σ.

Given an EFSM G_x, we define the notions of the languages *generated* and *marked* by an EFSM by considering all paths that can be traversed while respecting guard formulas at all visiting states. To formalize the idea, we need to know in advance the values of variables after each prefix is generated. To this end, we define a string-values map.

Definition 4.2.2 (String-values map) Let $V : \Sigma^* \to \mathbb{B}^k$ be a map that assigns to a string $s \in \Sigma^*$ a tuple of boolean values assumed by variables in X at the state reached by s from the initial state of G_x. Thus, $V(s)$ is defined as follows:

$$V(s) = (v(s, x))_{x \in X}$$

where $v : \Sigma^* \times X \to \mathbb{B}$ is recursively defined as follows: for $s \in \Sigma^*$ and $x \in X$,

1. $v(\epsilon, x) := 0$;

2. $v(s\sigma, x) := a^x_\sigma(V(s))$, $\sigma \in \Sigma$. □

According to this definition, $v(s, x)$ is always tracking the value of $x \in X$ at the state $\delta(q_0, t)$ for all prefixes $t \leq s$. Therefore, the value of x at the state reached by s can be precalculated by $V(s)$. Using the notation, we define the closed and marked languages of an EFSM.

Definition 4.2.3 (Closed and marked languages of an EFSM)
The closed language of an EFSM $G_x = (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$, denoted by $L(G_x)$, is defined recursively as follows:

1. $\epsilon \in L(G_x)$.

2. $s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \Leftrightarrow s\sigma \in L(G_x)$.

The language marked by G_x, denoted by $L_m(G_x)$, is:

$$L_m(G_x) := \{s \in L(G_x) \mid \delta(q_0, s) \in Q_m\} \qquad \square$$

The language $L(G_x)$ contains the event sequences of all directed paths starting from the initial state that can be traversed along the state transition diagram while respecting guard formulas at all visiting states; the string corresponding to a path is the concatenation of the event labels of the constituting transitions in the path. Therefore, a string s is in $L(G_x)$ if and only if it corresponds to an admissible path in the state transition diagram; equivalently, if and only if at all states in the path the guard formula evaluates to *true* for a transition that is a part of the string s. Obviously, $L(G_x)$ is prefix-closed by definition.

The marked language represented by G_x, $L_m(G_x)$, is a subset of $L(G_x)$ consisting only of the strings s for which $\delta(q_0, s) \in Q_m$, that is, these strings correspond to paths that lead to one of the marker states in the state transition diagram. Since not all states need be marked, $L_m(G_x)$ need not be prefix-closed in general.

A state $q \in Q$ is *reachable* if $q = \delta(q_0, s)$ for some $s \in \Sigma^*$. G_x itself is reachable if q is reachable for all $q \in Q$. A state $q \in Q$ is *coreachable* if there exists an $s \in \Sigma^*$ such that $\delta(q, s) \in Q_m$, and G_x is coreachable if q is coreachable for all $q \in Q$. G_x is nonblocking if every reachable state is coreachable, or equivalently $L(G_x) = \overline{L_m(G_x)}$. In other words, any string that can be generated by G_x is a prefix of (i.e. can always be completed to) a marker state of G_x.

Before showing an example, we introduce the notation used for guards and updating actions in following figures. Each transition in the diagram is equipped with a guard and updating functions. A right arrow '$\rightarrow$' indicates that the formula g on the left is the guard formula for the event on the right, while '/' indicates that after the event occurs the function A on the right follows to update variables. When g is true, we simply write σ/A, while if there are no updates we write $g \rightarrow \sigma$.

Example 4.1 (Closed and marked languages of an EFSM)

As shown in Fig. 4.1, G_x is equipped with boolean variables, guard formulas and updating functions. At the initial state q_0, all boolean variables are initialized to 0. G_x is not empty, so $\epsilon \in L(G_x)$ as defined. Since the guard formula for event α is

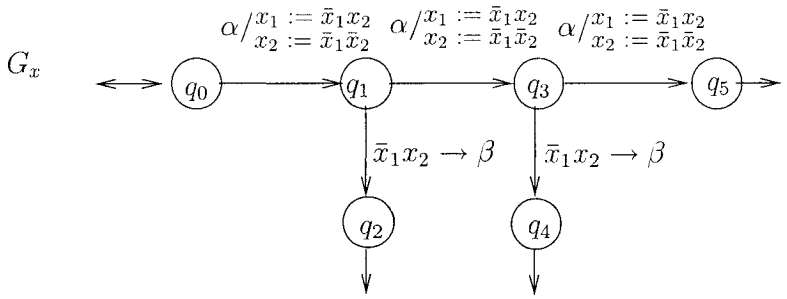

Figure 4.1: Language represented by an EFSM.

true, α is always enabled regardless of the values of boolean variables. Therefore, by inspecting the transition diagram, we know that $\{\epsilon, \alpha, \alpha\alpha, \alpha\alpha\alpha\} \subseteq L(G_x)$. At state q_1, X has been updated to $\{0,1\}$ by $a_\alpha^{x_1}$ and $a_\alpha^{x_2}$. Then $g_\beta(0,1) = 1$ implies that $\alpha\beta \in L(G_x)$. At state q_3, X has been updated to $\{1,0\}$. Then $g_\beta(1,0) = 0$ implies that $\alpha\alpha\beta \notin L(G_x)$. No other path is defined in the transition diagram of G_x, so the language generated by G_x is:

$$L(G_x) = \{\epsilon, \alpha, \alpha\alpha, \alpha\alpha\alpha, \alpha\beta\}$$

In $L(G_x)$, only the string α and $\alpha\alpha$ do not lead the system to a marker state, so according to the definition of marked language of an EFSM,

$$L_m(G_x) = \{\epsilon, \alpha\alpha\alpha, \alpha\beta\}$$

$\diamond$

4.2.2 Equivalent regular FSM

In this section we establish that EFSMs and regular FSMs have equal expressive powers. An EFSM and a regular FSM are said to be *equivalent* if and only if they generate and mark the same languages.

Definition 4.2.4 (Equivalent regular FSM of an EFSM)
The equivalent regular FSM G_{eq} of a given EFSM $G_x = (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$ is a five tuple:

$$G_{eq} = (R, \Sigma, f, r_0, R_m)$$

where:

- $R = Q \times \mathbb{B}^k$ is a finite set of states;

- Σ is the alphabet of events in G_x;

- $f : R \times \Sigma \rightarrow R$ is the partial transition function. For a state $r = (q, \mathbf{v}) \in R$ and $\sigma \in \Sigma$,

$$f(r, \sigma)! \text{ iff } \delta(q, \sigma)! \wedge g_\sigma(\mathbf{v}) = 1,$$

  in which case $f(r, \sigma) = (q', \mathbf{v}')$, where $q' = \delta(q, \sigma)$ and $\mathbf{v}' = (a^x_\sigma(\mathbf{v}))_{x \in X}$, where $\mathbf{v}$ and $\mathbf{v}'$ are the values of variables at states q and q', respectively.

- $r_0 = (q_0, \underbrace{0, 0, \ldots, 0}_{k})$ is the initial state.

- $R_m \subseteq R$ is the set of marker states where: $R_m = \{(q, \mathbf{v}) \in R | q \in Q_m\}$. □

We denote by Eq the operation that converts an EFSM to its equivalent regular FSM, i.e. $Eq(G_x) = G_{eq}$. For the sake of convenience, f is extended from domain $R \times \Sigma$ to domain $R \times \Sigma^*$ in the following recursive manner:

1. $f(r, \epsilon) := r$;

2. $f(r, s)! \wedge f(f(r, s), \sigma)! \Rightarrow f(r, s\sigma) = f(f(r, s), \sigma)$ for $s \in \Sigma^*$ and $\sigma \in \Sigma$.

The following theorem states that an EFSM and its equivalent regular FSM generate and mark the same languages. First we state and prove a preliminary result.

Lemma 1 For all $s \in L(G_x)$, $f(r_0, s) = (\delta(q_0, s), V(s))$.

Proof: The proof is by induction on the length of strings.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$. Then, $f(r_0, \epsilon) = (\delta(q_0, \epsilon), V(\epsilon))$ trivially holds since $r_0 = (q_0, \underbrace{0, 0, \ldots, 0}_{k})$ as defined.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since s is of length $n-1$, it follows from the inductive assumption that:

$$f(r_0, s) = (\delta(q_0, s), V(s)).$$

Next, we have

$$\begin{aligned} f(r_0, s') &= f(r_0, s\sigma) \wedge \delta(q_0, s\sigma)! \\ &= f(f(r_0, s), \sigma) \wedge \delta(q_0, s\sigma)! \\ &= f((\delta(q_0, s), V(s)), \sigma) \wedge \delta(q_0, s\sigma)! \\ &= (\delta(\delta(q_0, s), \sigma), A_\sigma(V(s))) \wedge \delta(q_0, s\sigma)! \qquad \text{(Def. 4.2.4)} \\ &= (\delta(q_0, s\sigma), V(s\sigma)). \end{aligned}$$

■

We now prove our main result which states that EFSMs and FSMs have equal expressive power.

Theorem 4.2.1 Given an EFSM $G_x = (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$, its equivalent regular FSM $G_{eq} = (R, \Sigma, f, r_0, R_m)$ generates and marks the same languages as G_x, i.e.

$$L(G_x) = L(G_{eq}) \tag{4.1}$$

$$L_m(G_x) = L_m(G_{eq}) \tag{4.2}$$

Proof: The proof of equation 4.1 is by induction on the length of strings. First, we will show that for all $s' \in L(G_x)$, it must be the case that $s' \in L(G_{eq})$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$. Then trivially $\epsilon \in L(G_{eq})$ since G_x is nonempty.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since $L(G)$ is prefix-closed and $s\sigma \in L(G_x)$, it implies by inductive assumption that:

$$s \in L(G_{eq}).$$

Furthermore, we know that

$$s \in L(G_{eq}) \Rightarrow f(r_0, s)!.$$

In addition, by the Definition 4.2.3 it follows that,

$$\begin{aligned}
& s\sigma \in L(G_x) \wedge f(r_0, s)! \\
\Rightarrow\ & \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \wedge f(r_0, s)! \\
\Rightarrow\ & \delta(\delta(q_0, s), \sigma)! \wedge g_\sigma(V(s)) = 1 \wedge f(r_0, s)! \\
\Rightarrow\ & f(f(r_0, s), \sigma)! && \text{(Lemma. 1 and Def. 4.2.4)} \\
\Rightarrow\ & f(r_0, s\sigma)! \\
\Rightarrow\ & s\sigma = s' \in L(G_{eq}).
\end{aligned}$$

Second, we will show that for all $s' \in L(G_{eq})$, it must be the case that $s' \in L(G_x)$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$, $\epsilon \in L(G_{eq})$ trivially since G_{eq} is nonempty.
- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since $L(G_{eq})$ is prefix-closed and $s\sigma \in L(G_{eq})$, it implies by the inductive assumption that:

$$s \in L(G_x).$$

Next, we have

$$\begin{aligned}
& s \in L(G_x) \wedge s' \in L(G_{eq}) \\
\Rightarrow\ & s \in L(G_x) \wedge f(r_0, s\sigma)! \\
\Rightarrow\ & s \in L(G_x) \wedge f(f(r_0, s), \sigma)!) \\
\Rightarrow\ & s \in L(G_x) \wedge f((\delta(q_0, s), V(s)), \sigma)! && \text{(Lem. 1)} \\
\Rightarrow\ & s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 && \text{(Def. 4.2.4)} \\
\Rightarrow\ & s' = s\sigma \in L(G_x) && \text{(Def. 4.2.3)}
\end{aligned}$$

We conclude that $L(G_x) = L(G_{eq})$.

By Definition 4.2.4, a state $r = f(r_0, s') = (q, \mathbf{v}) \in R$ is marked if and only if the corresponding state $q = \delta(q_0, s') \in Q$ is marked, so $L_m(G_x) = L_m(G_{eq})$. ■

So far, we have shown that each EFSM can be converted to its equivalent regular FSM. The following example illustrates the conversion.

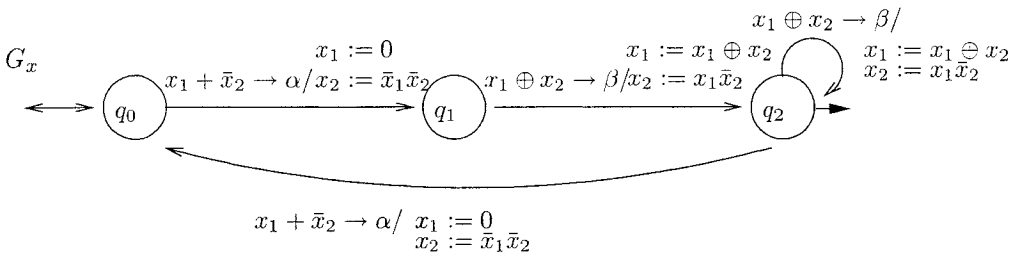

Figure 4.2: Converting an EFSM to its equivalent regular FSM (a).

Example 4.2 (Converting an EFSM to its equivalent regular FSM)
As shown in Fig. 4.2, G_x is equipped with two boolean variables, guard formulas and updating functions for events. Based on Definition 4.2.4, we convert G_x to its equivalent regular FSM G_{eq}. First of all, since G_{eq} is nonempty, we create a state $r_0 = (q_0, 0, 0)$ as the initial state of G_{eq}, shown in Fig. 4.3-1. At state q_0 only event α is generated in G_x with $g_\alpha(0,0) = 1$, so a transition labeled with α should be generated at the state $r_0 = (q_0, 0, 0)$ in G_{eq}. It leads G_{eq} from r_0 to a new state $r_1 = (\delta(q_0, \alpha), a_\alpha^{x_1}(0,0), a_\alpha^{x_2}(0,0)) = (q_1, 0, 1)$, shown in Fig. 4.3-2. At state q_1 only event β is generated in G_x with $g_\beta(0,1) = 1$, so a transition labeled with β should be generated at the state $r_1 = (q_1, 0, 1)$ in G_{eq}. It leads G_{eq} from r_1 to a new state $r_2 = (\delta(q_1, \beta), a_\beta^{x_1}(0,1), a_\beta^{x_2}(0,1)) = (q_2, 1, 0)$, shown in Fig. 4.3-3. At state q_2, both α and β are generated in G_x. Since $g_\alpha(1,0) = 1$ and $g_\beta(1,0) = 1$, both transitions labeled with α and β should be generated at state $r_2 = (q_2, 1, 0)$ in G_{eq}. Transition α leads G_{eq} from r_2 to state $(\delta(q_2, \alpha), a_\alpha^{x_1}(1,0), a_\alpha^{x_2}(1,0)) = (q_0, 0, 0) = r_0$, which is the initial state of G_{eq}. Transition β leads G_{eq} from r_2 to a new state $r_3 = (\delta(q_2, \beta), a_\beta^{x_1}(1,0), a_\beta^{x_2}(1,0)) = (q_2, 1, 1)$, shown in Fig. 4.3-4. At state q_2, both α and β are generated in G_x. However, for $x_1 = 1$ and $x_2 = 1$, only $g_\alpha(x_1, x_2) = 1$ while $g_\beta(x_1, x_2)$ evaluates *false*. So only transition α should be generated at state $r_3 = (q_2, 1, 1)$ in G_{eq}. Transition α leads G_{eq} to state $(\delta(q_2, \alpha), a_\alpha^{x_1}(1,1), a_\alpha^{x_2}(1,1)) = (q_0, 0, 0) = r_0$, which is the initial state of G_{eq}, shown in Fig. 4.3-5.

No other state or transition can possibly be generated in G_{eq}. Finally, we mark all states in G_{eq} whose corresponding state components are marked in G_x. Thus, $r_0 = (q_0, 0, 0)$, $r_2 = (q_2, 1, 0)$ and $r_3 = (q_2, 1, 1)$ should be marked, as shown in Fig.

4.3-6.

The resulting G_{eq} and G_x generate and mark the same language, i.e. $L(G_x) = L(G_{eq})$, $L_m(G_x) = L_m(G_{eq})$. ◇

4.2.3 Synchronous product

In this section we describe a way of combining two EFSMs into their synchronous product ($\|$). The technique will be used for the specification of control problems involving the coordination or synchronization of several EFSMs together. The commutation between synchronous product operation and the Eq operator will be discussed at the end of this section.

Note that not any two given EFSMs are compatible to work together. One undesirable case should be excluded: the updating functions of a common event change the values of a common variable differently in each machine. We limit the synchronous product operation on two EFSMs which satisfy a consistency condition, resulting in their concurrent opreation.

Definition 4.2.5 (Consistency condition) Given two EFSMs
$G_{xi} = (Q_i, \Sigma_i, \delta_i, q_{i0}, Q_{im}, X_i, g_i, A_i)$, $i = 1, 2$, they are consistent if

$\forall \sigma \in \Sigma_1 \cap \Sigma_2$ and $\forall x \in X_1 \cap X_2$, $a^x_{1\sigma} = a^x_{2\sigma}$. □

The consistency condition guarantees the synchronous product of EFSMs to work properly by preventing the above ambiguous and hence undesirable situation. The consistency condition requires that the updating functions triggered by a common event in the two machines set a common variable to the same value. In other words, the component EFSMs will not update a common variable to different values when they run in parallel.

Definition 4.2.6 (Synchronous product of two EFSMs)
Given two consistent EFSMs
$G_{xi} = (Q_i, \Sigma_i, \delta_i, q_{i0}, Q_{im}, X_i, g_i, A_i)$, $i = 1, 2$, their synchronous product is:

$$G_{x1} \parallel G_{x2} := (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$$

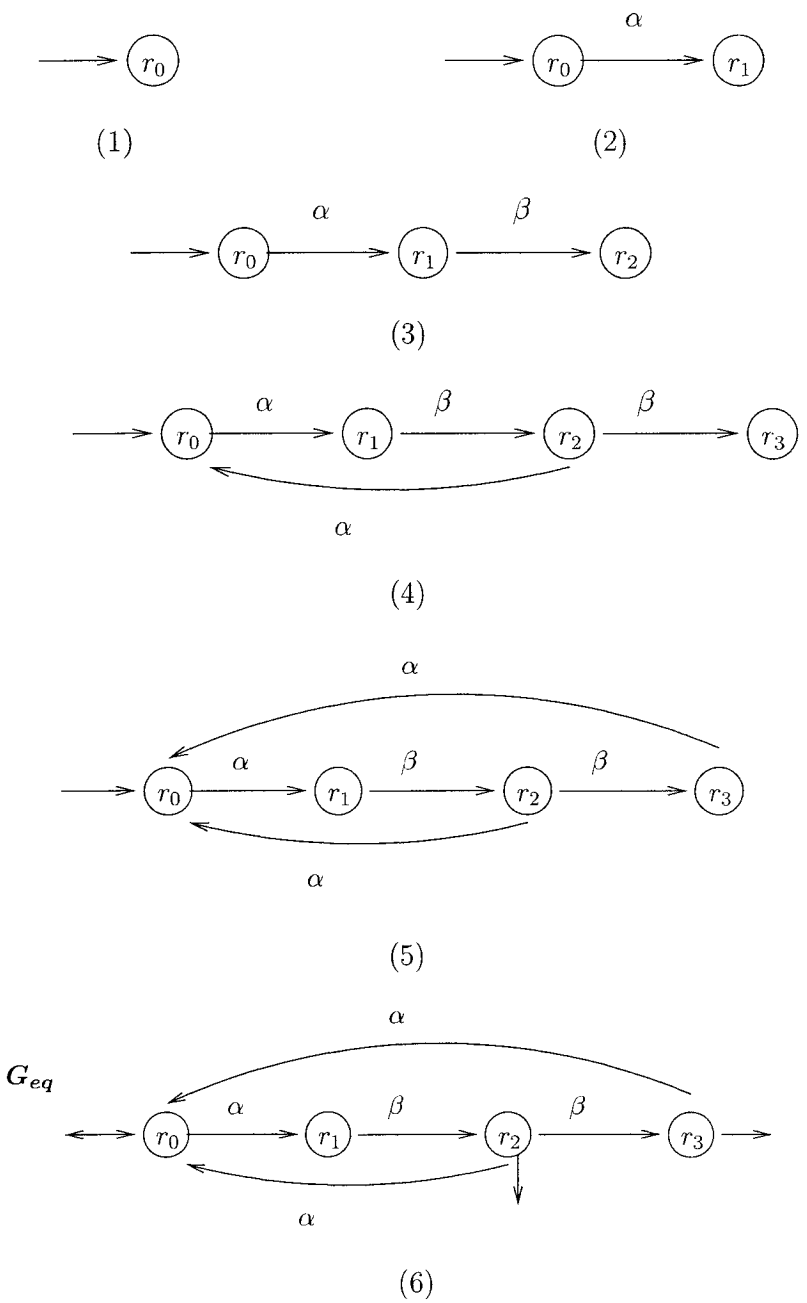

Figure 4.3: Converting an EFSM to its equivalent regular FSM (b).

where:

- $Q = Q_1 \times Q_2$;

- $\Sigma = \Sigma_1 \cup \Sigma_2$;

- For $q_1, q_2 \in Q$ and $\sigma \in \Sigma$,

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)); \\ \qquad \text{if } \delta_1(q_1, \sigma)! \wedge \delta_2(q_2, \sigma)! \\ (\delta_1(q_1, \sigma), q_2); \\ \qquad \text{if } \delta_1(q_1, \sigma)! \wedge \sigma \notin \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)); \\ \qquad \text{if } \delta_2(q_2, \sigma)! \wedge \sigma \notin \Sigma_1 \\ \text{undefined}; \\ \qquad \text{otherwise} \end{cases}$$

- $q_0 = (q_{10}, q_{20})$;

- $Q_m = Q_{1m} \times Q_{2m}$;

- $X = X_1 \cup X_2$;

- For $\sigma \in \Sigma$,

$$g_\sigma = \begin{cases} g_{1\sigma} \wedge g_{2\sigma}; & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \\ g_{1\sigma}; & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \\ g_{2\sigma}; & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

- $A_\sigma = (a_\sigma^x)_{x \in X}$, where for $\sigma \in \Sigma$ and $x \in X$,

$$a_\sigma^x = \begin{cases} a_{1\sigma}^x (= a_{2\sigma}^x); & \text{if } x \in X_1 \cap X_2 \wedge \\ & \qquad \sigma \in \Sigma_1 \cap \Sigma_2 \\ a_{1\sigma}^x; & \text{if } x \in X_1 \setminus X_2 \wedge \sigma \in \Sigma_1 \\ & \qquad \vee x \in X_1 \wedge \sigma \in \Sigma_1 \setminus \Sigma_2 \\ a_{2\sigma}^x; & \text{if } x \in X_2 \setminus X_1 \wedge \sigma \in \Sigma_2 \\ & \qquad \vee x \in X_2 \wedge \sigma \in \Sigma_2 \setminus \Sigma_1 \\ x; & \text{otherwise} \end{cases}$$

□

We make the following remarks about this definition:

- The consistency conditions can be easily checked by inspecting the updating functions of common events on common variables before taking the synchronous product.
- Without considering boolean variables, guard formulas and updating functions, the synchronous product of EFSMs is identical to the synchronous product of regular FSMs.
- $|X| = |X_1| + |X_2| - |X_1 \cap X_2|$, where $|X|$ denotes the number of boolean variables in set X.
- A transition with a common label in the composed EFSM should be guarded by both guard formulas in the component EFSMs, i.e. a common event is allowed to occur in composed EFSM if and only if both of its component guard formulas are *true*.
- The updating function of the composed EFSM updates the values of boolean variables in X. Since $A_{1\sigma}$ and $A_{2\sigma}$ always agree on updating common variables, we can think of common variables as being updated by either $A_{1\sigma}$ or $A_{2\sigma}$, if $\sigma \in \Sigma_1 \cap \Sigma_2$. Table. 4.1 summarizes the updating functions of the composed EFSM for nine possible combinations of $x \in X$ and $\sigma \in \Sigma$.

Table 4.1: Updating functions of the synchronous product of two EFSMs.

$\sigma \setminus x$	$X_1 \cap X_2$	$X_1 \setminus X_2$	$X_2 \setminus X_1$
$\Sigma_1 \cap \Sigma_2$	$a^x_{1\sigma}$ or $a^x_{2\sigma}$	$a^x_{1\sigma}$	$a^x_{2\sigma}$
$\Sigma_1 \setminus \Sigma_2$	$a^x_{1\sigma}$	$a^x_{1\sigma}$	x
$\Sigma_2 \setminus \Sigma_1$	$a^x_{2\sigma}$	x	$a^x_{2\sigma}$

The following result states that the synchronous product of two FSMs can be extended by taking the synchronous product of the extended components; in other words, the operations of 'extension' and 'synchronous product' commute.

Theorem 4.2.2 Let $G_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{im})$, $i = 1, 2$ be FSMs, and define $G := G_1 \parallel G_2 = (Q, \Sigma, \delta, q_0, Q_m)$. Assume that G is extended to an EFSM $G_x = (G; X, g, A)$. Then

$$L(G_x) = L(G_{x1} \parallel G_{x2}) \tag{4.3}$$

$$L_m(G_x) = L_m(G_{x1} \parallel G_{x2}) \tag{4.4}$$

where in $G_{xi} = (G_i; X, g_i, A_i)$, $g_i = g|_{\Sigma_i}$ and $A_i = A|_{\Sigma_i}$, $i = 1, 2$.

Proof: First note that by definition $G_{x1} \parallel G_{x2}$ has a transition diagram identical to G_x. Next, we show that guard formulas and updating functions of $G_{x1} \parallel G_{x2}$ and G_x are identical by denoting the guard formulas of G_x and $G_{x1} \parallel G_{x2}$ by g and g', respectively, and updating functions of of G_x and $G_{x1} \parallel G_{x2}$ by A and A', respectively.

For an event $\sigma \in \Sigma$, we discuss its guards, g_σ and g'_σ, and its updating functions, A_σ and A'_σ in three cases:

For $\sigma \in \Sigma_1 \backslash \Sigma_2$, by the definition of synchronous product of $G_{x1} \parallel G_{x2}$, we know that $g'_\sigma = g_{1\sigma}$. On the other hand, since $\sigma \in \Sigma_1 \backslash \Sigma_2$, $g_{1\sigma} = g_\sigma$, which in trun implies $g_\sigma = g'_\sigma$. By the definition of $G_{x1} \parallel G_{x2}$, for $x \in X$, $a'^x_\sigma = a^x_{1\sigma}$. On the other hand, since $\sigma \in \Sigma_1 \backslash \Sigma_2$, $a^x_\sigma = a^x_{1\sigma}$, which in turn implies $a'^x_\sigma = a^x_\sigma$. Since the variable sets of G_x and $G_{x1} \parallel G_{x2}$ are the same (X), we conclude that $A_\sigma = A'_\sigma$.

For $\sigma \in \Sigma_2 \backslash \Sigma_1$, by a similar argument, it follows that $g_\sigma = g'_\sigma$ and $A_\sigma = A'_\sigma$.

For $\sigma \in \Sigma_1 \cap \Sigma_2$, by the definition of synchronous product of $G_{x1} \parallel G_{x2}$, we know that $g'_\sigma = g_{1\sigma} \wedge g_{2\sigma}$. Since $\sigma \in \Sigma_1 \cap \Sigma_2$ and $g_i = g|_{\Sigma_i}$, $i = 1, 2$, $g_{1\sigma} = g_{2\sigma} = g_\sigma$, which in turn implies $g_\sigma = g'_\sigma$. By the definition of $G_{x1} \parallel G_{x2}$, for $x \in X$, $a'^x_\sigma = a^x_{1\sigma} = a^x_{2\sigma}$. On the other hand, since $\sigma \in \Sigma_2 \cap \Sigma_1$, $a^x_{1\sigma} = a^x_{2\sigma} = a^x_\sigma$, which in turn implies $a'^x_\sigma = a^x_\sigma$. Since the variable sets of G_x and $G_{x1} \parallel G_{x2}$ are same (X), we conclude that $A_\sigma = A'_\sigma$.

Finally, a state (q_1, q_2) in $G_{x1} \parallel G_{x2}$ is marked if $q_i \in Q_{im}$, $i = 1, 2$, which implies the state (q_1, q_2) in G is also marked. Since the operation 'extension' does not change

the marker states set Q_m of G_x, (q_1, q_2) in G_x is also marked. Thus, we can assert that the two EFSMs $G_{x1} \parallel G_{x2}$ and G_x are identical, and therefore they generate and mark the same languages as shown in Eq. 4.3 and Eq. 4.4. ■

The following theorem shows that the operations 'synchronous product' and 'equivalence regular FSM' commute given a condition that an updating function of a private event never changes a common variable's value. Before proving this, we show a preliminary result.

Lemma 2 Let G_{x1} and G_{x2} be two consistent EFSMs and $G_x = G_{x1} \parallel G_{x2}$. If an updating function of a private event never changes a common variable's value, i.e. for $x \in X_1 \cap X_2$ and $\alpha \in (\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)$, $a^x_{1\alpha} = a^x_{2\alpha} = x$, then for all $s \in L(G_x)$ and $\sigma \in \Sigma$,

$$g_\sigma(V(s)) = \begin{cases} g_{1\sigma}(V_1(P_1(s))) \wedge g_{2\sigma}(V_2(P_2(s))) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \\ g_{1\sigma}(V_1(P_1(s))) & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \\ g_{2\sigma}(V_2(P_2(s))) & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

where V, V_1 and V_2 are the string-values maps of G_x, G_{x1} and G_{x2}, respectively, and P_i, $i = 1, 2$, is the natural projection $\Sigma^* \rightarrow \Sigma_i^*$.

Proof: First, we consider $\sigma \in \Sigma_1 \setminus \Sigma_2$. According to Definition 4.2.6, we know that after s occurs in G_x from the initial state, $\sigma \in \Sigma_1 \setminus \Sigma_2$ is enabled if and only if $g_{1\sigma}$ evaluates *true* calculated at the current values of variables in X_1. Therefore, we need to show that for all $x \in X_1$, the values assigned to x by $V(s)$ and $V_1(P_1(s))$ are identical. By induction on the length of s, we show that

$$v(s, x) = v_1(P_1(s), x) \tag{4.5}$$

- Base: $length(s) = 0$, i.e. $s = \epsilon$. This trivially holds true.
- Inductive step: $length(s) = n$. Let $s = t\beta$, where $t \in \Sigma^*$. Since both t and $P_1(t)$ are of length smaller than n, it follows from the inductive assumption that:

$$v(t, x) = v_1(P_1(t), x).$$

Furthermore, since we are only concerned about $x \in X_1$, if $\beta \in \Sigma_1 \cap \Sigma_2 \wedge x \in X_1$, then

$$
\begin{aligned}
v(s,x) &= v(t\beta,x) \\
&= a^x_\beta\Big(V(t)\Big) \\
&= a^x_\beta\Big(v(t,y)_{y\in X}\Big) \\
&= a^x_{1\beta}\Big(v(t,y)_{y\in X_1}\Big) && \text{(Def. 4.2.6)} \\
&= a^x_{1\beta}\Big(v_1(P_1(t),y)_{y\in X_1}\Big) && \text{(inductive assumption)} \\
&= a^x_{1\beta}\Big(V_1(P_1(t))\Big) \\
&= v_1(P_1(t)\beta,x) && \text{(Def. 4.2.2)} \\
&= v_1(P_1(t\beta),x) && (\because \beta \in \Sigma_1) \\
&= v_1(P_1(s),x)
\end{aligned}
$$

Otherwise, $\beta \in (\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)$, then

$$
\begin{aligned}
v(s,x) &= a^x_\beta(V(t)) \\
&= v(t,x) && (\because a^x_\sigma = x) \\
&= v_1(P_1(t),x) && \text{(inductive assumption)} \\
&= v_1(P_1(t\beta),x) && (\because \beta \notin \Sigma_1) \\
&= v_1(P_1(s),x)
\end{aligned}
$$

So, the inductive case holds, and we conclude that

$$v(s,x) = v_1(P_1(s),x)$$

which implies $\forall\sigma \in \Sigma_1\setminus\Sigma_2$ and $s \in \Sigma^*$, $g_\sigma(V(s)) = g_{1\sigma}(V_1(P_1(s)))$.

Similarly, we conclude $\forall\sigma \in \Sigma_2\setminus\Sigma_1$ and $s \in \Sigma^*$, $g_\sigma(V(s)) = g_{2\sigma}(V_2(P_2(s)))$.

For the case $\sigma \in \Sigma_1 \cap \Sigma_2$, we know that after s occurs in G_x from the initial state, σ is enabled if and only if both $g_{1\sigma}$, calculated at the current values of $X_1 \subseteq X$, and $g_{2\sigma}$, calculated at the current values of $X_2 \subseteq X$, evaluate *true*. Since we have

proved that for all $x \in X_i$, the values assigned to x by $V(s)$ and $V_i(P_i(s))$, $i = 1, 2$, are identical, we can easily ascertain that

$$\begin{aligned} g_\sigma(V(s)) &= g_{1\sigma}(V(s)) \wedge g_{2\sigma}(V(s)) \\ &= g_{1\sigma}(V_1(P_1(s))) \wedge g_{2\sigma}(V_2(P_2(s))) \end{aligned}$$

■

Theorem 4.2.3 Given two consistent EFSMs $G_{xi} = (Q_i, \Sigma_i, \delta_i, q_{i0}, Q_{mi}, X_i, g_i, A_i)$, $i = 1, 2$, if for $x \in X_1 \cap X_2$ and $\sigma \in (\Sigma_1 \backslash \Sigma_2) \cup (\Sigma_2 \backslash \Sigma_1)$, $a^x_{1\sigma} = a^x_{2\sigma} = x$, the equivalent FSM of their parallel product is equivalent to the parallel product of their equivalent FSM, that is

$$L\Big(Eq(G_{x1} \parallel G_{x2})\Big) = L\Big(Eq(G_{x1}) \parallel Eq(G_{x2})\Big) \tag{4.6}$$

$$L_m\Big(Eq(G_{x1} \parallel G_{x2})\Big) = L_m\Big(Eq(G_{x1}) \parallel Eq(G_{x2})\Big) \tag{4.7}$$

Proof:

Let

$$\begin{gathered} G_x = G_{x1} \parallel G_{x2} = (Q, \Sigma, \delta, q_0, Q_m, X, g, A), \\ G_{eq1} = Eq(G_{x1}) = (R_1, \Sigma_1, f_1, r_{10}, R_{m1}), \\ G_{eq2} = Eq(G_{x2}) = (R_2, \Sigma_2, f_2, r_{20}, R_{m2}), \\ G_{eq} = Eq(G_x) = (R, \Sigma, f, r_0, R_m) \end{gathered}$$

and

$$G'_{eq} = G_{eq1} \parallel G_{eq2} = (R', \Sigma', f', r'_0, R'_m).$$

We need to prove:

$$L(G_{eq}) = L(G'_{eq}) \tag{4.8}$$

$$L_m(G_{eq}) = L_m(G'_{eq}) \tag{4.9}$$

The proof of equation 4.8 is by induction on the length of strings. First, we show that for all $s' \in L(G_{eq})$, it must be the case that $s' \in L(G'_{eq})$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$, then trivially $\epsilon \in L(G'_{eq})$ since it is nonempty.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since s is of length of $n-1$, if follows from the inductive assumption that:

$$s \in L(G_{eq}) \Rightarrow s \in L(G'_{eq}).$$

If $\sigma \in \Sigma_1 \cap \Sigma_2$, then we can write

$s' = s\sigma \in L(G_{eq})$

$s' = s\sigma \in L(G_x)$ (equivalence)

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (Def 4.2.3)

$\Rightarrow s \in L(G_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (equivalence)

$\Rightarrow s \in L(G'_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (inductive assumption)

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge \delta(q_0, s\sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = g_{2\sigma}(V_2(P_2(s))) = 1$

(Lemma. 2)

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge \delta_1(q_{10}, P_1(s\sigma))! \wedge \delta_2(q_{20}, P_2(s\sigma))!$
$\wedge g_{1\sigma}(V_1(P_1(s))) = g_{2\sigma}(V_2(P_2(s))) = 1$

($\because G_x = G_{x1} \parallel G_{x2}$ and $\sigma \in \Sigma_1 \cap \Sigma_2$)

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge \delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge \delta_2(\delta_2(q_{20}, P_2(s)), \sigma)!$
$\wedge g_{1\sigma}(V_1(P_1(s))) = g_{2\sigma}(V_2(P_2(s))) = 1$ ($\because \sigma \in \Sigma_1 \cap \Sigma_2$)

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge \left(\delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1\right)$
$\wedge \left(\delta_2(\delta_2(q_{20}, P_2(s)), \sigma)! \wedge g_{2\sigma}(V_2(P_2(s))) = 1\right)$

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge f_1(f_1(r_{10}, P_1(s)), \sigma)! \wedge f_2(f_2(r_{20}, P_2(s)), \sigma)!$ (Def.4.2.4)

$\Rightarrow s \in L(G_{eq1} \parallel G_{eq2}) \wedge f(f(r_0, s), \sigma)!$ (Def. 4.2.6)

$\Rightarrow s\sigma \in L(G_{eq1} \parallel G_{eq2})$ ($\because \sigma \in \Sigma_1 \cap \Sigma_2$)

$\Rightarrow s\sigma \in L(G'_{eq})$

If $\sigma \in \Sigma_1 \setminus \Sigma_2$, then we can write

$s' = s\sigma \in L(G_{eq})$

$\Rightarrow s' = s\sigma \in L(G_x)$ (equivalence)

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (Def. 4.2.3)

$\Rightarrow s \in L(G_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (equivalence)

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (inductive assumption)

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge \delta(q_0, s\sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1$ (Lemma. 2)

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge \delta_1(q_{10}, P_1(s\sigma))! \wedge g_{1\sigma}(V_1(P_1(s))) = 1$

$(\because G_x = G_{x1} \parallel G_{x2})$

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge \delta\_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1$

$(\because G_x = G_{x1} \parallel G_{x2}$ and $\sigma \in \Sigma_1 \setminus \Sigma_2)$

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge \left(\delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1\right)$

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge f_1(f_1(r_{10}, P_1(s)), \sigma)!$ (Def.4.2.4)

$\Rightarrow s\sigma \in L(G_{eq1}) \parallel L(G_{eq2})$ $(\because \sigma \in \Sigma_1 \setminus \Sigma_2)$

$\Rightarrow s\sigma \in L(G'_{eq})$.

Similarly, $s\sigma \in L(G_{eq}) \Rightarrow s\sigma \in L(G'_{eq})$ holds true for $\sigma \in \Sigma_2 \setminus \Sigma_1$. So we can conclude that:

$$L(G_{eq}) \subseteq L(G'_{eq}).$$

Second, we show that for all $s' \in L(G'_{eq})$, it must be the case that $s' \in L(G_{eq})$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$, then trivially $s' \in L(G_{eq})$ since $L(G_{eq})$ is nonempty and prefix-closed.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since s is of length of $n - 1$, it follows from the inductive assumption that:

$$s \in L(G'_{eq}) \Rightarrow s \in L(G_{eq}).$$

If $\sigma \in \Sigma_1 \cap \Sigma_2$, then we can write

$s\sigma \in L(G'_{eq})$

$\Rightarrow s\sigma \in L(G_{eq1}) \parallel L(G_{eq2})$

$\Rightarrow s \in L(G_{eq1}) \parallel L(G_{eq2}) \wedge f_1(f_1(r_{10}, P_1(s\sigma)))! \wedge f_2(f_2(r_{20}, P_2(s\sigma)))!$

$\Rightarrow s \in L(G'_{eq}) \wedge f_1(f_1(r_{10}, P_1(s)), \sigma)! \wedge f_2(f_2(r_{20}, P_2(s)), \sigma)!$ $(\because \sigma \in \Sigma_1 \cap \Sigma_2$)

$\Rightarrow s \in L(G'_{eq}) \wedge \Big(\delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1\Big)$

$\wedge \Big(\delta_2(\delta_2(q_{20}, P_2(s)), \sigma)! \wedge g_{2\sigma}(V_2(P_2(s))) = 1\Big)$ (Def.4.2.4)

$\Rightarrow s \in L(G'_{eq}) \wedge \Big(\delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge \delta_2(\delta_2(q_{20}, P_2(s)))!\Big)$

$\wedge \Big(g_{1\sigma}(V_1(P_1(s))) = 1 \wedge g_{2\sigma}(V_2(P_2(s))) = 1\Big)$

$\Rightarrow s \in L(G'_{eq}) \wedge \delta(q_0, s\sigma)! \wedge \Big(g_{1\sigma}(V_1(P_1(s))) = g_{2\sigma}(V_2(P_2(s))) = 1\Big)$

(Def. 4.2.6)

$\Rightarrow s \in L(G'_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ $(\because \sigma \in \Sigma_1 \cap \Sigma_2$ and Lemma. 2)

$\Rightarrow s \in L(G_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (inductive assumption)

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (equivalence)

$\Rightarrow s\sigma \in L(G_x)$ (Def 4.2.3)

$\Rightarrow s\sigma \in L(G_{eq})$ (equivalence)

If $\sigma \in \Sigma_1 \setminus \Sigma_2$, we can write

$s\sigma \in L(G'_{eq})$

$\Rightarrow s\sigma \in L(G_{eq1}) \parallel L(G_{eq2})$

$\Rightarrow s \in L(G'_{eq}) \wedge f_1(f_1(r_{10}, P_1(s\sigma)))!$ $(\because \sigma \in \Sigma_1 \setminus \Sigma_2)$

$\Rightarrow s \in L(G'_{eq}) \wedge f_1(f_1(r_{10}, P_1(s)), \sigma)!$ $(\because \sigma \in \Sigma_1)$

$\Rightarrow s \in L(G'_{eq}) \wedge \Big(\delta_1(\delta_1(q_{10}, P_1(s)), \sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1\Big)$ (Def.4.2.4)

$\Rightarrow s \in L(G'_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_{1\sigma}(V_1(P_1(s))) = 1$ (Def. 4.2.6)

$\Rightarrow s \in L(G'_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ $(\because \sigma \in \Sigma_1 \setminus \Sigma_2$ and Lemma. 2)

$\Rightarrow s \in L(G_{eq}) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (inductive assumption)

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1$ (equivalence)

$\Rightarrow s\sigma \in L(G_x)$ (Def 4.2.3)

$\Rightarrow s\sigma \in L(G_{eq})$ (equivalence)

Similarly, $s\sigma \in L(G'_{eq}) \Rightarrow s\sigma \in L(G_{eq})$ holds true for $\sigma \in \Sigma_2 \setminus \Sigma_1$. So we can conclude that:

$$L(G'_{eq}) \subseteq L(G_{eq}).$$

Based on the above two sub-conclusions, we have proved equation 4.8.

Finally, we conclude the proof of Theorem 4.2.3 by proving equation 4.9.

$s' \in L(G_{eq})$

$\Leftrightarrow s' \in L(G_x)$ (equivalence)

$\Leftrightarrow s' \in L(G_x\ ) \wedge \delta(q_0, s') \in Q_m$ (Def.4.2.3)

$\Leftrightarrow s' \in L(G'_{eq}) \wedge \delta(q_0, s') \in Q_m$ (equivalence and Eq.4.8)

$\Leftrightarrow s' \in L(G'_{eq}) \wedge P_1(s') \in L(G_{eq1})) \wedge P_2(s') \in L(G_{eq2})) \wedge \delta(q_0, s') \in Q_m$

$\Leftrightarrow s' \in L(G'_{eq}) \wedge P_1(s') \in L(G_{eq1})) \wedge P_2(s') \in L(G_{eq2}))$

$\quad \wedge \delta_1(q_{10}, P_1(s')) \in Q_{m1} \wedge \delta_2(q_{20}, P_2(s')) \in Q_{m2}$ (Def. 4.2.6)

$\Leftrightarrow s' \in L(G'_{eq}) \wedge \Big(P_1(s') \in L(G_{eq1})) \wedge \delta_1(q_{10}, P_1(s')) \in Q_{m1}\Big)$

$\quad \wedge \Big(P_2(s') \in L(G_{eq2})) \wedge \delta_2(q_{20}, P_2(s')) \in Q_{m2}\Big)$

$\Leftrightarrow s' \in L(G'_{eq}) \wedge \Big(f_1(r_{10}, P_1(s')) \in R_{m1} \wedge f_2(r_{20}, P_2(s')) \in R_{m2}\Big)$

(Def.4.2.4)

$\Leftrightarrow s' \in L(G'_{eq}) \wedge f'(r'_0, s') \in R'_m$

$\Leftrightarrow s' \in L_m(G'_{eq})$ ■

4.3 Example

In this section, we show how the foregoing model can be used to build up the specifications for a control problem, to be known as a *small factory*.

Shown in Fig. 4.4, the small factory operates as follows. The machines M_1 and M_2 are connected to each other through a buffer of capacity one. An item is fetched (α_i) and is then processed (β_i) by machine M_i, $i = 1, 2$. Machine M_1 fetches an item

from an input conveyor belt and processes it. After being processed by M_1, the item is placed in the buffer and is later fetched by M_2 for further processing. It is desired that the buffer neither overflows nor underflows.

The formal design procedure is explained in the next chapter; here we present an intuitive solution. To satisfy the overflow/underflow restriction, we introduce a boolean variable x which effectively counts the number of items in the buffer. Machine M_1 can fetch a new item if the buffer is empty ($x = 0$), while machine M_2 can fetch an item if there is an item already in the buffer ($x = 1$). When M_1 places an item in the buffer (β_1), it sets $x := 1$, and when M_2 fetches an item from the buffer (α_2), it sets $x := 0$. We call M_1 and M_2 extended as M_{x1} and M_{x2}, respectively. It is

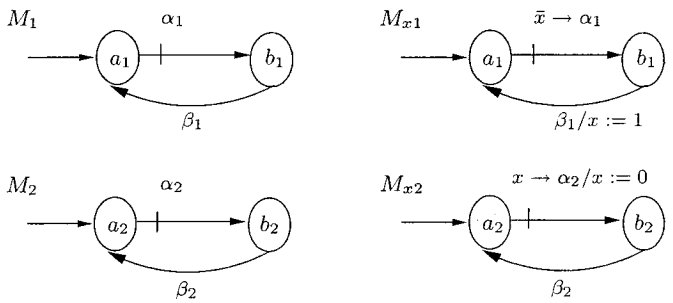

Figure 4.4: Machines M_1 and M_2, and extended machines M_{x1} and M_{x2}.

easy to see that no common event is shared by the two machines, so the synchronous product of M_{x_1} and M_{x_2}, shown in Fig. 4.5-(a), exists and models the behavior of the overall *controlled* system of machines and buffer. The equivalent regular FSM is shown in Fig. 4.5-(b) as well, from which it can be easily verified that the overflow and underflow specifications are both met.

We would like to point out that if we embedded our control in the synchronous product of M_1 and M_2 the result, as shown by Theorem 4.2.2, would also satisfy the control objectives.

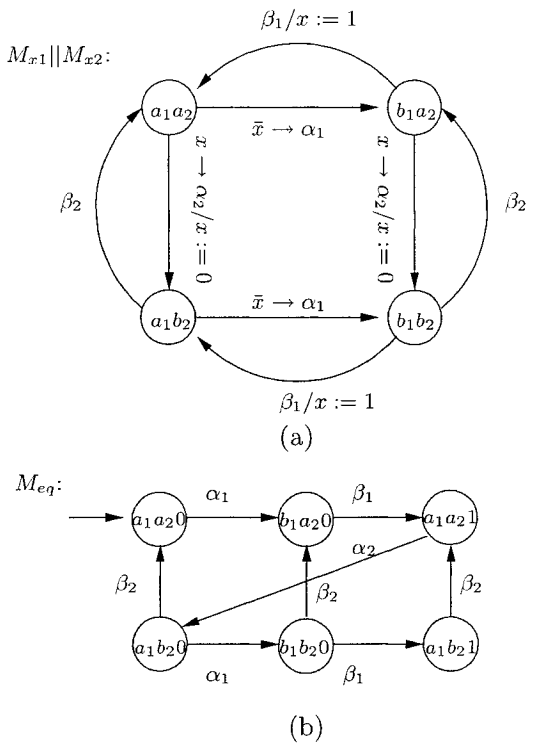

Figure 4.5: Controlled system.

4.4 Problem definition

In the next chapter we present a novel approach to implement the supervisiory control by an EFSM, in which the supervisor is embedded in the system to be controlled.

Suppose that we already have a plant G and a supervisor S modeled by regular FSM. We extend the plant by embedding the control mechanism in it. The result, denoted by G_x, is shown in Fig. 1.1-b.

Supervisory control is introduced by extending the plant with boolean variables, guard formulas and updating functions. Boolean variables are used to encode the supervisors states. Event observation is captured by a set of boolean functions that update the values of boolean variables and are triggered by the occurrence of events. Finally, control is introduced by guarding events with boolean formulas. The resulting EFSM implements the supervisory control map in the sense that the languages generated and marked by the EFSM are equal to those of the supervised system, that

is:

$$L(G_x) = L(S/G)$$
$$L_m(G_x) = L_m(S/G).$$

4.5 Conclusion

In this chapter, we have defined EFSM by augmenting the traditional FSM. We have shown that although EFSMs have equal expressive power as FSMs, they offer far more economical and realistic representations of physical systems. Languages represented by an EFSM are defined as well as the synchronous product of two consistent EFSMs. Two useful properties of the 'synchronous product' operation are shown. In addition, we use 'small factory' as an example to show how the EFSM can be used to model the specification of a control problem, and the main problem to be solved in this thesis is defined in the end.

Chapter 5

Embedded Supervisory Control by EFSM

5.1 Introduction

The objective of supervisory control is to synthesize controllers or supervisors such that the closed-loop system consisting of plant and controllers meets the specification of some desired behavior. The concept of supervisory control for regular FSM was presented by Ramadge and Wonham [RW87].

In this chapter, we discuss supervisory control implemented by extended finite state machines, where the supervisor is embedded in the plant. In our design, the plant and the supervisor are synchronized on the set of events, and the close-loop (controlled) system is designed in EFSM framework at once. The controller disables or enables a transition by defining a guard formula that evaluates to *false* whenever the transition leads to illegal behavior. We assume that a plant and a supervisor represented by nonempty deterministic finite automata, denoted by G and S respectively, have been already given, and an EFSM G_x is to be designed based on the information provided by G and S. Also, we will show G_x does enforce the supervision of S on G, in the sense that the languages generated and marked by the EFSM are equal to those of the supervised system, that is:

$$L(G_x) = L(S/G)$$
$$L_m(G_x) = L_m(S/G).$$

5.2 Supervisory control implemented by EFSM

5.2.1 Design overview

In this subsection the idea of supervisory control implemented by EFSM will be explained. Consider a plant to be controlled G over a set of events Σ, modeled by FSM. We design a boolean formula for each event in Σ, called a guard formula. The formulas are defined over a set of boolean variables. A transition labeled with an event is enabled if the guard formula associated with that event, calculated at the current values of variables, evaluates *true*. The values of variables are updated by updating functions, which are triggered by the occurrence of events. Notice that each event has an updating function for each boolean variable.

The supervision is thus enforced by the above control mechanism. Next, we present our design mathematically. Since in traditional supervisory control the supervisor is the machine that enables or disables controllable events which are to be generated by the plant, all the variables and formulas are derived from the automaton of the supervisor. Throughout this work we assume that all supervisors are controllable with respect to the plant to be controlled.

Let $S = (Y, \Sigma, \xi, y_0, Y_m)$ be the generating automaton of a controllable supervisor. Three steps are involved in the design of an EFSM G_x that implements the supervision of S on G:

1. Boolean variables design;

2. Guard formulas design;

3. Updating functions design.

5.2.2 Boolean variables design

In our design, boolean variables are introduced for two ends:

1. Encoding the states of the supervisor S;
2. Combined with binary operations, they form guard formulas and updating functions.

Before initializing the values of boolean variables, we have to decide on the number of boolean variables, i.e. how many boolean variables are sufficient for encoding the states of S.

Definition 5.2.1 (Boolean variable set X) X is a boolean variable set containing k boolean variables, and k, the required number of boolean variables in X for supervisory control, is equal to the next higher integer of the logarithm of N base 2, where N is the number of the supervisor's states:

$$k = \lceil \log_2 N \rceil \tag{5.1}$$

□

Notice that even when not all the bit combinations are used, still k boolean variables are necessary when $k - 1 < log_2 N < k$. Without loss of generality, all the boolean variables are initialized to *false* (0), i.e. $X := \{\underbrace{0, 0, \ldots, 0}_{k}\}$.

Next, we assign each state $y_i \in Y$ a *label* l_i, $0 \leq i \leq 2^k$, which is a unique bit combination of k-bit boolean variables. Since all the variables are initialized to 0, the initial state, y_0, is always labeled with '$\underbrace{00\ldots0}_{k}$'.

For convenience we define two injective maps: state-label map l and label-minterm map m.

Definition 5.2.2 (State-label map) A label-state map $l : Y \rightarrow \mathbb{B}^k$ is an injective map which assigns to a state its unique label. □

Definition 5.2.3 (Label-minterm map) A label-minterm map $m : \mathbb{B}^k \rightarrow M_k$ is a map in which a label is mapped to the k-minterm that is *true* only at that label. Thus, for $(v_1, v_2, \ldots, v_k) \in \mathbb{B}^k$, $m(v_1, v_2, \ldots, v_k) = z_1 z_2 \ldots z_k$, where

$$z_i = \begin{cases} x_i & , v_i = 1 \\ \bar{x}_i & , v_i = 0 \end{cases}$$

□

5.2.3 Guard formulas design

The guard formula of an event is designed to control the occurrence of that event. It is formed by boolean variables in X combined with binary operators. If the current values of variables evaluate the guard formula to *true* (1), the event is enabled; otherwise, the event is disabled. Since a controller should never disable an uncontrollable event, the guard for an uncontrollable event is always *true*. However, a guard for a controllable event may evaluate to *true* or *false* depending on the current values of variables.

For a general event α, its guard formula, denoted by g_α, is calculated as prescribed in the following definition.

Definition 5.2.4 (Guard formula) Let $L_{g(\alpha)}$ be the collection of labels of all supervisor states from which α is enabled, i.e.:

$$L_{g(\alpha)} = \{l(y) \mid \xi(y, \alpha)!, y \in Y\}$$

Then the guard formula for an uncontrollable event is always *true*; for a controllable event g_α is the boolean expression in DNF disjoining the minterms whose bit combinations are in $L_{g(\alpha)}$, that is:

$$g_\alpha = \begin{cases} 1 & , \alpha \in \Sigma_{uc} \\ \sum_{l \in L_{g(\alpha)}} m(l) & , \alpha \in \Sigma_c \end{cases} \tag{5.2}$$

□

5.2.4 Updating functions design

The updating functions are used to update the values of variables in X. They are triggered by the occurrence of events. Since the functions update the values of all variables, each event triggers k updating functions, each for a variable in X. The updating functions of event α, denoted by A_α, are calculated as prescribed in the following definition.

Definition 5.2.5 (Updating functions) Let $[\!|\ x\ |\!]_y$ denote the value of boolean variable x in state y, and let $L_{A(\alpha,x)}$ be the set of all state labels from which x becomes 1 after the occurrence of α:

$$L_{A(\alpha,x)} = \{l(y) \mid [\!|\ x\ |\!]_{\xi(y,\alpha)} = 1, y \in Y\ \}.$$

Then the updating function activated by α on x is the boolean expression in DNF disjoining the minterms whose bit combinations are in $L_{A(\alpha,x)}$, that is

$$a_\alpha^x := \sum_{l \in L_{A(\alpha,x)}} m(l) \tag{5.3}$$

and

$$A_\alpha = (a_\alpha^x)_{x \in X} \tag{5.4}$$

□

5.2.5 Embedded supervisor design

Before we formally introduce the design of an embedded supervisor, we show two preliminary results. First we show that the map V introduced in Section 4.2.2 indeed assigns to a string s the values of the boolean variables reached by s.

Lemma 3 We have: $v(s,x) = [\!|\ x\ |\!]_{\xi(y_0,s)}$.

Proof: The proof is by the induction on the length of s.

- Base: $|s| = 0$, i.e. $s = \epsilon$. By Definition 4.2.2

$$\begin{aligned} v(\epsilon, x) &= 0 \\ &= [\!| x |\!]_{\xi(y_0,\epsilon)} \end{aligned}$$

So, the base case holds.

- Inductive step: $|s| = n$. Let $s = u\sigma$, where $u \in \Sigma^*$ and $\sigma \in \Sigma$. Since u is of length $n-1$, it follows from the inductive assumption that:

$$v(u, x) = [\!| x |\!]_{\xi(y_0,u)}.$$

Furthermore, by Definition 4.2.2

$$\begin{aligned} v(s, x) &= a^x_\sigma(v(u, z))_{z \in X} \\ &= a^x_\sigma([\!| z |\!]_{\xi(y_0,u)})_{z \in X} \end{aligned}$$

According to the definition of updating function a^x_σ, we know that

$$a^x_\sigma([\!| z |\!]_{\xi(y_0,u)})_{z \in X} = 1 \Leftrightarrow l(\xi(y_0, u)) \in L_A(\sigma, x)$$

i.e.

$$\begin{aligned} v(s, x) = 1 &\Leftrightarrow l(\xi(y_0, u)) \in L_A(\sigma, x) \\ &\Leftrightarrow [\!| x |\!]_{\xi(y_0,u\sigma)} = 1 \\ &\Leftrightarrow [\!| x |\!]_{\xi(y_0,s)} = 1 \end{aligned}$$

So, $v(s, x) = [\!| x |\!]_{\xi(y_0,s)}$ holds true for the inductive step. ■

Second, we show at a reachable supervisor state the guard formula of a generated event evaluates to *true* with the values of variables at that state.

Lemma 4 For $s \in \Sigma^*$ and $\sigma \in \Sigma$, $s\sigma \in L(S)$ iff $s \in L(S)$ and $g_\sigma(V(s)) = 1$.

Proof:

$s\sigma \in L(S) \Leftrightarrow \xi(y_0, s\sigma)!$

$\Leftrightarrow \xi(y_0, s)! \wedge l(\xi(y_0, s)) \in L_{g(\sigma)}$ (Def. 5.2.4)

$\Leftrightarrow s \in L(S) \wedge g_\sigma([\!| x |\!]_{\xi(y_0,s)})_{x \in X} = 1$

$\Leftrightarrow s \in L(S) \wedge g_\sigma(v(s,x))_{x\in X} = 1$ (Lem. 3)

$\Leftrightarrow s \in L(S) \wedge g_\sigma(V(s)) = 1$ ■

Given the automaton of an admissible supervisor $S = (Y, \Sigma, \xi, y_0, Y_m)$ over a plant $G = (Q, \Sigma, \delta, q_0, Q_m)$ with $\Sigma_c \subseteq \Sigma$, we implement the supervisory control map by extending G to an EFSM $G_x = (Q, \Sigma, \delta, q_0, Q_m, X, g, A)$. The EFSM G_x can be regarded as the closed-loop system satisfying the control objectives. The first five components of G_x are identical to those of G, while X, g and A are derived from S as described in details in the previous subsections, and summarized below for convenience.

- $X = \{x_1, x_2, \ldots, x_k\}$, where $k = \lceil \log_2 |Y| \rceil$.

- For $\sigma \in \Sigma$,

$$g_\sigma = \begin{cases} 1 & , \sigma \in \Sigma_{uc} \\ \sum_{l\in L_{g(\sigma)}} m(l) & , \sigma \in \Sigma_c \end{cases}$$

where

$$L_{g(\sigma)} = \{l(y) \mid y \in Y \wedge \xi(y,\sigma)!.$$

- For $\sigma \in \Sigma$, $A_\alpha = (a^x_\alpha)_{x\in X}$, where $a^x_\alpha := \sum_{l\in L_{A(\alpha,x)}} m(l)$ and

$$L_{A(\sigma,x_i)} = \{l(y) \mid y \in Y \wedge [\, x_i \,]_{\xi(y,\sigma)} = 1\}.$$

□

Now, we show that the EFSM G_x designed as prescribed above will in effect implement the supervisory control map enforced by S.

Theorem 5.2.1 For the EFSM G_x designed as above we have $L(G_x) = L(G) \cap L(S) = L(S/G)$. In addition, $L_m(G_x) = L_m(G) \cap L_m(S) = L_m(S/G)$ if $L_m(S)$ is $L_m(G)$-closed.

Proof: If N is 1, each event is always enabled or disabled at the only state of the supervisor, so no boolean variable is needed. If N is greater than 1, we assign a bit

combination to every state, resulting in a minimum number of $\lceil \log_2 N \rceil$ variables for encoding N states.

The proof is by induction on the length of strings.

First, we will show that for all $s' \in L(G_x)$, it must be the case that $s' \in L(G) \cap L(S)$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$. This trivially holds true since $L(G) \cap L(S)$ is nonempty and prefix-closed.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since s is of length $n-1$, it follows from inductive assumption that:

$$s \in L(G_x) \Rightarrow s \in L(G) \cap L(S).$$

Furthermore, we know that:

$$\begin{aligned}
& s' = s\sigma \in L(G_x) \\
& \Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 && \text{(Def. 4.2.3)} \\
& \Rightarrow s \in L(G) \wedge s \in L(S) \wedge \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \\
& \Rightarrow \Big(s \in L(G) \wedge \delta(q_0, s\sigma)! \Big) \wedge \Big(s \in L(S) \wedge g_\sigma(V(s)) = 1 \Big) \\
& \Rightarrow s\sigma \in L(G) \wedge s\sigma \in L(S) && \text{(Lemma 4)} \\
& \Rightarrow s' = s\sigma \in L() \cap L(S)
\end{aligned}$$

We conclude that:

$$L(G_x) \subseteq L(G) \cap L(S).$$

Next, we will show that for all $s' \in L(G) \cap L(S)$, it must be the case that $s' \in L(G_x)$.

- Base: $length(s') = 0$, i.e. $s' = \epsilon$. This trivially holds true since $L(G_x)$ is nonempty.

- Inductive step: $length(s') = n \geq 1$. Let $s' = s\sigma$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Since s is of length $n-1$, it follows that:

$$s \in L(G) \cap L(S) \Rightarrow s \in L(G_x)$$

Furthermore, we know that:

$s' = s\sigma \in L(G) \cap L(S)$

$\Rightarrow s \in L(G) \cap L(S) \wedge s\sigma \in L(G) \wedge s\sigma \in L(S)$

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge s \in L(S) \wedge g(V(s)) = 1$ (Lemma 4)

$\Rightarrow s \in L(G_x) \wedge \delta(q_0, s\sigma)! \wedge g(V(s)) = 1$

$\Rightarrow s\sigma \in L(G_x)$

$\Rightarrow s' \in L(G_x)$

We conclude that:

$$L(G) \cap L(S) \subseteq L(G_x).$$

and therefore,

$$L(G_x) = L(G) \cap L(S).$$

For the marked language,

$L_m(G_x)$

$= L(G_x) \cap L_m(G)$

$= L(S/G) \cap L_m(G)$

$= L(G) \cap L(S) \cap L_m(G)$

$= L_m(S) \cap L_m(G)$ ($L_m(S)$ is $L_m(G)$-closed)

$= L_m(S/G).$ ■

Notice that if we assume that the supervisor is a nonmarking supervisor, i.e., the marker states of the controlled system are decided by the plant only, the condition requiring $L_m(S)$ being $L_m(G)$-closed would not be necessary in order to obtain $L_m(G_x) = L_m(S/G)$ since in this case

$L_m(G_x)$

$= L(G_x) \cap L_m(G)$

$= L(S/G) \cap L_m(G)$

$= L_m(S/G)$.

The following theorem states a sufficient condition for the controlled system to be nonblocking.

Theorem 5.2.2 Given nonblocking G and S, if $L_m(G)$ and $L_m(S)$ are nonconflicting and S is a nonmarking supervisor or $L_m(S)$ is $L_m(G)$-closed, G_x is nonblocking.

Proof:

$\overline{L_m(G_x)}$

$= \overline{L_m(S/G)}$ ($\because$ S is a nonmarking supervisor or $L_m(S)$ is $L_m(G)$-closed)

$= \overline{L_m(G) \cap L_m(S)}$

$= \overline{L_m(G)} \cap \overline{L_m(S)}$ (nonconflicting)

$= L(G) \cap L(S)$ (G and S are nonblocking)

$= L(S/G)$

$= L(G_x)$ ■

5.3 Examples

We present three examples to illustrate the procedure of controller design. In the first example, we simply present a plant and an admissible supervisor.

5.3.1 Example 1

With $\Sigma = \Sigma_c = \{\alpha, \beta, \gamma\}$, a plant G and an admissible supervisor $S = \{Y, \Sigma, \xi, y_0, Y_m\}$ are given in Fig. 5.1.

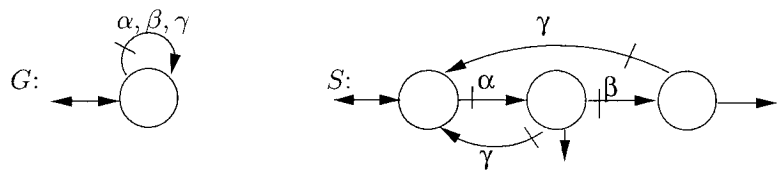

Figure 5.1: Example: embedded supervisory control design.

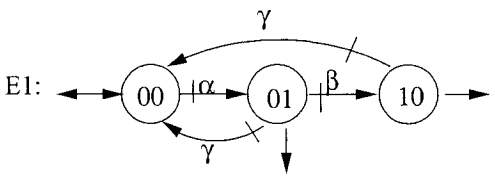

Figure 5.2: Labeled supervisor states.

First, we determine the value of k, the required number of boolean variables for encoding supervisor states. There are three states in supervisor S, i.e. $N = 3$. Therefore, according to equation 5.1 we choose $k = \lceil \log_2 3 \rceil = 2$, so the set X contains two boolean variables: $X = \{x_1, x_2\}$. The encoded (labeled) supervisor states are shown in Figure 5.2. For instance, when the system transitions lead the supervisor to the state labeled with '01', the current values of x_1 and x_2 are 0 and 1, respectively. Notice that since both the variables are initialized to 0, the initial state of the supervisor is labeled with '00'.

Second, we design the guard formula for each event as a boolean formula over X.

- Guard formula for α, denoted by g_α. The set of labels that label states at which event α is enabled in S is:

$$\begin{aligned} L_{g(\alpha)} &= \{l(y) | \xi(y, \alpha)!, y \in Y\} \\ &= \{00\} \end{aligned} \tag{5.5}$$

  So, the event α is enabled only after the occurrence of those strings which lead the system to the state labeled with '00' in S. According to the equation 5.2, we have

$$\begin{aligned} g_\alpha &= \sum_{l \in L_{g(\alpha)}} m(l) \\ &= m(00) \\ &= \bar{x}_1 \bar{x}_2 \end{aligned} \tag{5.6}$$

- Guard formula for β, denoted by g_β. We follow the same procedure as that in

designing g_α.

$$\begin{aligned} L_{g(\beta)} &= \{l(y)|\xi(y,\beta)!, y \in Y\} \\ &= \{01\} \end{aligned} \tag{5.7}$$

$$\begin{aligned} g_\beta &= \sum_{l \in L_{g(\beta)}} m(l) \\ &= m(01) \\ &= \bar{x}_1 x_2 \end{aligned} \tag{5.8}$$

- Guard formula for γ, denoted by g_γ.

$$\begin{aligned} L_{g(\gamma)} &= \{l(y)|\xi(y,\gamma)!, y \in Y\} \\ &= \{01, 10\} \end{aligned} \tag{5.9}$$

$$\begin{aligned} g_\gamma &= \sum_{l \in L_{g(\gamma)}} m(l) \\ &= m(01) + m(10) \\ &= \bar{x}_1 x_2 + x_1 \bar{x}_2 \\ &= x_1 \oplus x_2 \end{aligned} \tag{5.10}$$

Finally, we design the updating functions triggered by each event.

- The updating functions triggered by α, denoted by A_α, is a pair

$$A_\alpha = (a_\alpha^{x_1}, a_\alpha^{x_2}).$$

For x_1, the set of labels labeling the states from which x_1 becomes 1 after the occurrence of α is:

$$\begin{aligned} L_{A(\alpha,x_1)} &= \{l(y)|[\, x_1 \,]_{\xi(y,\alpha)} = 1, y \in Y\} \\ &= \emptyset \end{aligned} \tag{5.11}$$

According to the equation 5.3, we have

$$\begin{aligned} a_{\alpha}^{x_1} &:= \sum_{l \in L_{A(\alpha,x_1)}} m(l) \\ &= 0 \end{aligned} \tag{5.12}$$

Note that 0 is a boolean function that always returns the value *false* for all values of (x_1, x_2) in $\mathbb{B}^2$.

For x_2,

$$\begin{aligned} L_{A(\alpha,x_2)} &= \{l(y) | [\, x_2 \,]_{\xi(y,\alpha)} = 1, y \in Y\} \\ &= \{00\} \end{aligned} \tag{5.13}$$

So,

$$\begin{aligned} a_{\alpha}^{x_2} &:= \sum_{l \in L_{A(\alpha,x_2)}} m(l) \\ &= m(00) \\ &= \bar{x}_1 \bar{x}_2 \end{aligned} \tag{5.14}$$

- A_β is a pair. We follow the same procedure as that in designing A_α.

$$A_\beta = (a_\beta^{x_1}, a_\beta^{x_2})$$

For x_1,

$$\begin{aligned} L_{A(\beta,x_1)} &= \{l(y) | [\, x_1 \,]_{\xi(y,\beta)} = 1, y \in Y\} \\ &= \{01\} \end{aligned} \tag{5.15}$$

So,

$$\begin{aligned} a_{\beta}^{x_1} &= \sum_{l \in L_{A(\beta,x_1)}} m(l) \\ &= m(01) \\ &= \bar{x}_1 x_2 \end{aligned} \tag{5.16}$$

For x_2,

$$\begin{aligned} L_{A(\beta,x_2)} &= \{l(y)|[\neg x_2]_{\xi(y,\beta)} = 1, y \in Y\} \\ &= \emptyset \end{aligned} \tag{5.17}$$

So, the occurrence of β never updates the value of x_2 to *true* (1), and hence:

$$a_\beta^{x_2} = 0 \tag{5.18}$$

- A_γ is a pair

$$A_\gamma = (a_\gamma^{x_1}, a_\gamma^{x_2})$$

From Figure 5.2 we see that by taking the transitions labeled with γ the system always returns to its initial state, which means that γ always sets the variables x_1 and x_2 to *false*, i.e., $L_{A(\gamma,x_1)} = L_{A(\gamma,x_2)} = \emptyset$. So the updating functions of event γ are:

$$a_\gamma^{x_1} := 0 \tag{5.19}$$

$$a_\gamma^{x_2} := 0 \tag{5.20}$$

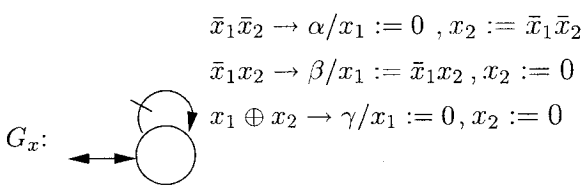

Figure 5.3: Extended system of Example 1.

The extended system G_x is shown in Fig. 5.3, and the equivalent FSM of G_x, denoted by G_{eq}, is shown in Fig. 5.4, from which it can be easily checked that $L(G_x) = L(G_{eq}) = L(S/G)$ and $L_m(G_x) = L_m(G_{eq}) = L_m(S/G)$. ◇

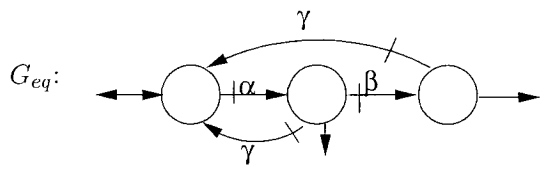

Figure 5.4: The equivalent regular FSM of the extended system.

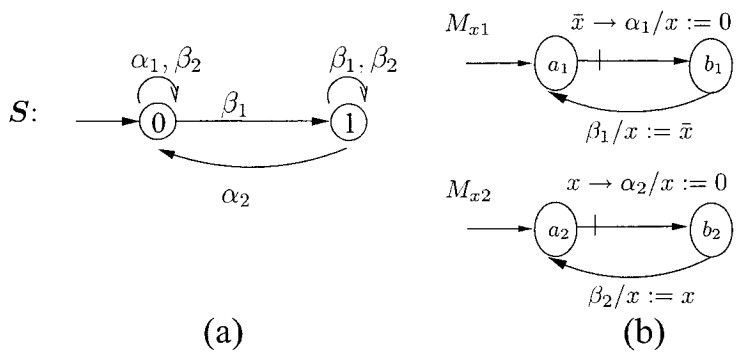

Figure 5.5: (a) Supervisor for small factory (b) Extended plants.

5.3.2 Example 2: Small factory

We formalize the supervisory control design for the small factory example described in Section 4.3. An admissible supervisor, shown in Fig. 5.5-a, implements the desired specification requiring that the buffer neither overflows nor underflows.

We note that one boolean variable is sufficient to encode the states of S. We denote this variable by x and initialize it to 0. We then label the initial state of S with '0' and the other state with '1'.

Since β_1 and β_2 are uncontrollable events, their guards are always *true*. For the events α_1 and α_2 we have $L_{g(\alpha_1)} = \{0\}$ and $L_{g(\alpha_2)} = \{1\}$, so the guards are $g_{\alpha_1} = \bar{x}$ and $g_{\alpha_2} = x$.

As for the updating functions, we have $L_{A(\alpha_1,x)} = L_{A(\alpha_2,x)} = \emptyset$, $L_{A(\beta_1,x)} = \{0\}$ and $L_{A(\beta_2,x)} = \{1\}$, so the updating functions are $a^x_{\alpha_1} = a^x_{\alpha_2} = 0$, $a^x_{\beta_1} = \bar{x}$ and $a^x_{\beta_2} = x$. Extended (controlled) plants are shown in Fig. 5.5-b.

It is worthwhile to observe that the designs in Fig. 5.5 and Fig. 4.4 are identical: the updating function $x := 0$ of α_1 can be dropped since α_1 is enabled only when

$x = 0$. Also, as is the case with β_2, when the value of a variable is not updated by the occurrence of an event, the updating function can be either left out as in Fig. 4.4, or set equal to the identity function as in Fig. 5.5. ◇

5.3.3 Example 3: Alternating bit protocol

Alternating Bit Protocol (ABP) [WC68, KR69] is used for reliable data transmission over half-duplex channels, and supervisory control in EFSM framework has firstly been applied to ABP in [PG04]. We formalize the design in this section. As shown in Fig. 5.6, two processes A and B communicate over a channel ch. Process A fetches a message and sends it to the channel. Then process B receives the message from the channel and if it is error-free, accepts it. The control objective states that *every message fetched by A should be accepted by B exactly once.*

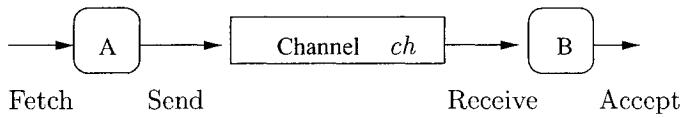

Figure 5.6: Two processes A and B communicating over a channel

In this example we first model the plant and will show how it fails to satisfy the specification of the desired behavior. Then embedded controllers for sender and receiver are synthesized by introducing boolean variables and designing guard formulas and updating functions so that the system under control satisfies the desired specification.

Plant and specification

A schematic of the plant is shown in Fig. 5.7, where a transmission error is depicted by a broken arrow. The system events are defined in Table 5.1. Fig. 5.8 shows FSM models for sender A, receiver B and channel ch. Finally, the specification [1] is

[1] Observe that since this specification is controllable with respect to the plant, it can serve as an admissible supervisor to achieve the control objective. For eliminating confusion, we write S in place of E.

Table 5.1: System events.

Event	Description
df	**d**ata **f**etched by A
ds	**d**ata **s**ent by A
dr	**d**ata **r**eceived by B
de	**d**ata received by B **e**rroneous
da	**d**ata **a**ccepted by B
cs	**c**ontrol (acknowledgement) **s**ent by B
cr	**c**ontrol **r**eceived by A
ce	**c**ontrol received by A **e**rroneous

formalized in Fig. 5.9, in which states are labeled with the value of the only boolean variable, explained in Section 5.2.2.

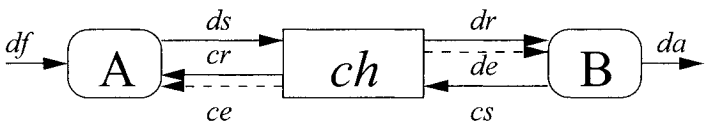

Figure 5.7: Schematic of the ABP plant.

We make two assumptions:

1. The channel can pass data messages only in one direction (from A to B), while the control information can flow bidirectionally.

2. Messages or control acknowledgements never get lost in the channel; rather, they can only get corrupted, which will always be detected by the receiving process.

Based on these assumptions, we briefly describe each plant component in Fig. 5.8. The sender A initially sends a data message to the channel, or fetches a new data message and sends it to the channel. After receiving an acknowledgement from the

channel, the sender A returns to its initial state. As far as the channel is concerned, any type of message received by the channel ch from one party (data ds or control cs) will be sent to the other party (dr or cr, respectively), or it will be delivered corrupted (de or ce, respectively). After receiving a data message from the channel, the receiver B nondeterministically sends an acknowledgement to the channel, or accepts the message and sends an acknowledgement to the channel. Note that for simplicity our models overapproximate the behavior of the actual system. Our embedded controller, to be designed later, will remove all unreasonable as well as illegal behavior.

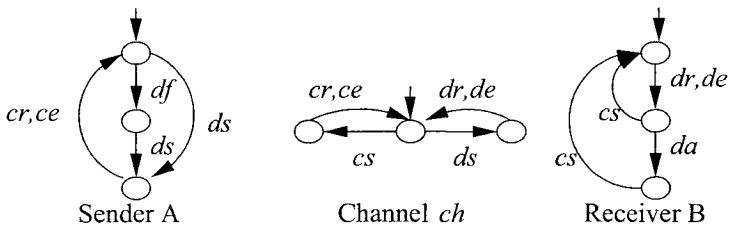

Figure 5.8: Parallel FSM of the plant. $\Sigma = \{df, ds, dr, de, da, cs, cr, ce\}$.

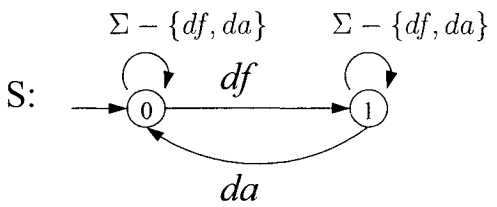

Figure 5.9: Requirement specification.

It is easy to see that the plants in Fig. 5.8 do not satisfy the specification in Fig. 5.9. For example, the string '***df***;*ds;dr;****da***;*cs;ce;ds;dr;****da***' is accepted by the plant but not by the specification. The problem is that the receiver B accepts a data message that has already been accepted. Thus, some form of control is required to prevent a duplicate copy of a data message from being accepted. The alternating bit protocol provides a standard solution to achieve the control objective. A description of the protocol is given by W. C. Lynch in [WC68].

Formalizing the alternating bit protocol by EFSM

In this subsection we formalize alternating bit protocol in the EFSM framework using the approach presented in Section 5.2.5. We extend A and B automata so that they can serve as embedded local supervisors at the sender and receiver sites.

To implement S by extending the FSMs of the plants, we note that one boolean variable is sufficient to encode the states of S. We denote this variable by x and initialize it to *0*. Then we label the initial state of S with '0' and the other state with '1', as shown in Fig. 5.9.

Since an event in the set $\Sigma - \{df, da\}$ is self-looped at all states of S, the guard formula for such an event is always *true*, while its occurrence does not change the value of x.

For the events *df* and *da* we have:

$$L_{g(df)} = \{0\}, \quad L_{A(df,x)} = \{0\}$$

$$L_{g(da)} = \{1\}, \quad L_{A(da,x)} = \emptyset$$

We conclude that $g_{df} = \bar{x}$, $a^x_{df} = \bar{x}$, $g_{da} = x$, and $a^x_{da} = 0$[2]. The extended plant components are shown in Fig. 5.10. The synchronous product of plant EFSMs is shown in Fig. 5.11, and its equivalent regular FSM is shown in Fig. 5.12.

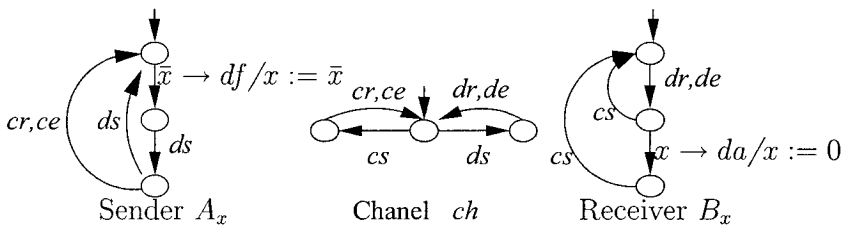

Figure 5.10: Alternating bit protocol in EFSM framework.

In the foregoing design of the protocol, one boolean variable x is introduced and is initialized to 0. The variable x is set to $\bar{x}$ and 0 by *fetch* and *accept* operations,

[2]Note that since *da* is enabled only when $x = 1$, the updating action of *da* can also be written as $a^x_{da} = \bar{x}$, and hence the name *alternating* bit protocol: when a message is either fetched or accepted, the variable x is toggled.

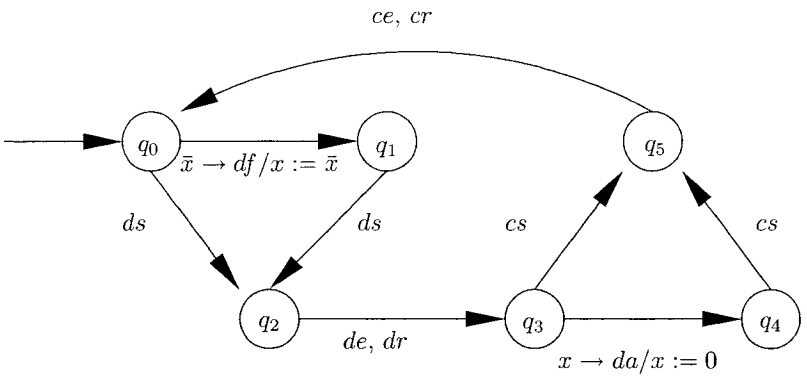

Figure 5.11: Synchronous product of sender, receiver and channel EFSMs.

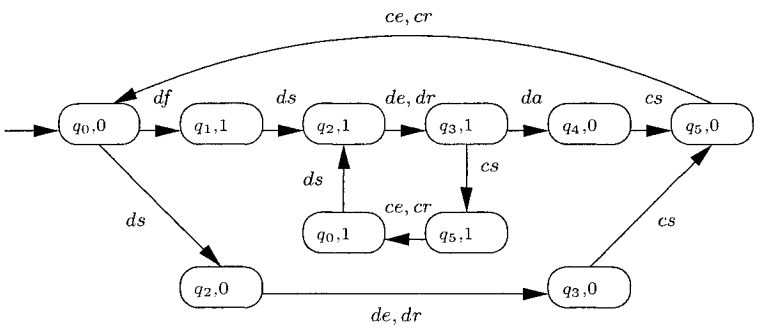

Figure 5.12: Equivalent FSM of Fig. 5.11.

respectively. Initially, we have $x = 0$ and therefore df is enabled while da is disabled. When df is eventually taken, the variable x becomes equal to 1 and as a result the event da is enabled while df is disabled. When da is finally taken, the variable x becomes equal to 0 again, and the cycle '$df; da$' repeats alternately.

The design presented in this section does not reflect the decentralized nature of alternating bit protocol. The variable x is regarded as a global variable, whose value is assumed to be instantly available at both processes[3]. However, this is not the case in practice as the sender and receiver are usually geographically widely separated. Thus, the value of variable x needs to be communicated to the second process in real-time when it is updated in the first process.

[3]In other words, control decisions are made *globally* while control actions are taken *locally*.

We have seen that the supervisory control of a small factory and ABP are examples in which control *decisions* are made *globally* while control *actions* are taken *locally*: given a global specification, local supervisors with full observation of the plant are embedded into local plants. In next chapter we discuss the case where the local supervisors have only a partial view of the plant.

5.4 Conclusion

In this chapter, we have presented a new approach to implement supervisory control by EFSM as an embedded part of the system to be controlled. The control mechanism is embedded by extending the plant to be controlled with boolean variables, guard formulas and updating functions. The resulting EFSM is shown to generate the same behavior as the system under supervision. Furthermore, we have presented a sufficient condition under which the controlled system is nonblocking. Examples to illustrate the design method are presented as well.

Chapter 6

Embedded Supervisory Control Under Partial Observation

6.1 Introduction

In this chapter, we apply our method described in the previous chapter to supervisory control problem under partial observation. Supervisors with partial observation are synthesized for the plant components, which are represented in the EFSM framework. Supervisory control problem under partial observation has been studied in [CD88], [FW90], [RW92], [SF00] and [GS00].

Controllability and observability of events in G_x, the controlled system, are interpreted as follows: G_x owns a set of boolean variables, denoted by X, whose values are fully known to G_x. The embedded controller guards controllable events with boolean formulas over X, and updates the values of variables in X after the occurrence of an observable event.

Both centralized and decentralized issues are discussed in this chapter. In the former case, we assume that each event is either controllable or observable, which makes our formulation more realistic (we simply do not care about an event that we can neither see nor control). For simplicity, we present all decentralized problems for *two* supervisors, which can be readily extended to any finite number of supervisors. We assume that a local event $\sigma \in \Sigma_i$, $i = 1, 2$, is either controllable or observable by

the local supervisor S_i, $i = 1, 2$, where Σ_i is the alphabet over which the local plant G_i is defined. In addition, S_i cannot see events in $\Sigma \setminus \Sigma_i$, where $\Sigma = \Sigma_1 \cup \Sigma_2$.

6.2 Centralized embedded control under partial observation

In this section we implement centralized supervisory control under partial observation in EFSM framework. We assume that the event set Σ is partitioned into disjoint sets Σ_o of observable events and Σ_{uo} of unobservable events. From the supervisor's view of the strings in Σ^*, a natural projection $P_o : \Sigma^* \rightarrow \Sigma_o^*$ erases all unobservable events.

Supervisory Control and Observation Problem with Zero Tolerance (SCOPZT) is defined in [RW92] as follows: Given a plant G over an alphabet $\Sigma = \Sigma_c \cup \Sigma_o$ and a nonempty language $K \subseteq L_m(G)$, construct a supervisor S for G, which observes only events in Σ_o and controls only events in Σ_c, such that $L_m(S/G) = K$.

The SCOPZT is solvable if and only if K is both *controllable* and *observable.*

Since the control information in the extended plant is derived from the supervisor, before the EFSM implementation in what follows we first present the construction of S.

Construction of S: Let $H = (Y, \Sigma, \xi, y_0, Y_m)$ be a recognizer for a controllable and observable specification language K, i.e. $L_m(H) = K$ and $L(H) = \bar{K}$. Let Y' be the set of nonempty subsets of Y. Since H is finite, Y' is guaranteed to be finite. The supervisor

$$S = (Y', \Sigma, \xi', y_0', Y_m')$$

is given by:

$\forall \sigma \in \Sigma_o, y' \in Y'$,

$$\xi'(y', \sigma) = \begin{cases} \{\xi(y, s) \mid s \in \Sigma^*, y \in y', P_o(s) = \sigma\} \\ \qquad\qquad\qquad\qquad \text{if this is nonempty} \\ undefined \qquad\qquad\qquad \text{otherwise} \end{cases}$$

$\forall \sigma \in \Sigma \setminus \Sigma_o, y' \in Y'$,

$$\xi'(y', \sigma) = \begin{cases} y' & \text{if } \exists y \in y', \xi(y, \sigma)! \\ undefined & \text{otherwise} \end{cases}$$

$$y'_0 := \{\xi(y_0, s) \mid s \in \Sigma^*, P_o(s) = \epsilon\}$$
$$Y'_m := \{y' \in Y' \mid \exists y \in y', y \in Y_m\}.$$

The following result states that the closed and marked behaviors of the supervised system are equal to $\bar{K}$ and K, respectively.

Proposition 6.2.1 If a nonempty language $K \subseteq L_m(G)$ is controllable and observable, then S constructed as above is a supervisor for G such that $L(S/G) = \bar{K}$ and $L_m(S/G) = K$.

Proof: First we show by induction on the length of strings that $L(S/G) = \bar{K}$.

- Base: $|s| = 0$. Since both $L(S/G)$ and $\bar{K}$ are nonempty and prefix-closed, $\epsilon \in L(S/G) \Leftrightarrow \epsilon \in \bar{K}$ trivially holds.

- Inductive step: assume for all $s \in \Sigma^*$, $|s| = n$ $(n \geq 0)$, that $s \in L(S/G) \Leftrightarrow s \in \bar{K}$. First we show that for all $\sigma \in \Sigma$, $s\sigma \in \bar{K} \Rightarrow s\sigma \in L(S/G)$.

  $s\sigma \in \bar{K} \Rightarrow s\sigma \in L(H) \wedge s \in \bar{K} \wedge s\sigma \in \overline{L_m(G)}$
  $\Rightarrow s\sigma \in L(H) \wedge s \in L(S/G) \wedge s\sigma \in L(G)$ (inductive assumption)
  $\Rightarrow s\sigma \in L(H) \wedge s \in L(S) \wedge s\sigma \in L(G)$
  $\Rightarrow \xi(y_0, s\sigma)! \wedge \xi'(y'_0, s)! \wedge s\sigma \in L(G)$

  CASE 1: $\sigma \in \Sigma_o$
  $\Rightarrow \xi(y_0, s\sigma)! \wedge \xi(y_0, s) \in \xi'(y'_0, s) \wedge \sigma \in \Sigma_o \wedge s\sigma \in L(G)$
  $\Rightarrow \xi'(y'_0, s\sigma)! \wedge s\sigma \in L(G)$ (Def. of $\xi'(y', \sigma)$)
  $\Rightarrow s\sigma \in L(S) \wedge s\sigma \in L(G)$
  $\Rightarrow s\sigma \in L(S/G)$

  CASE 2: $\sigma \in (\Sigma - \Sigma_o)$
  $\Rightarrow \xi(\xi(y_0, s), \sigma)! \wedge \xi(y_0, s) \in \xi'(y'_0. s) \wedge s\sigma \in L(G)$

$\Rightarrow \xi'(y'_0, s\sigma)! \wedge s\sigma \in L(G)$ (Def. of $\xi'(y', \sigma)$)

$\Rightarrow s\sigma \in L(S) \wedge s\sigma \in L(G)$

$\Rightarrow s\sigma \in L(S/G)$

Next we show that for all $\sigma \in \Sigma$, $s\sigma \in L(S/G) \Rightarrow s\sigma \in \bar{K}$.

CASE 1: $\sigma \in \Sigma_c$

$s\sigma \in L(S/G)$

$\Rightarrow s\sigma \in L(G) \wedge s \in L(S/G) \wedge s\sigma \in L(S)$

$\Rightarrow s\sigma \in L(G) \wedge s \in L(S/G) \wedge \xi'(y'_0, s)! \wedge \xi'(\xi'(y'_0, s), \sigma)!$ (Def. of $\xi'(y', \sigma)$)

$\Rightarrow s\sigma \in L(G) \wedge s \in L(S/G) \wedge \exists y \in Y, y \in \xi'(y'_0, s) \wedge \xi(y, \sigma)!$

$\Rightarrow s\sigma \in L(G) \wedge s \in L(S/G) \wedge \exists s' \in \Sigma^*, P(s') = P(s), \xi(y_0, s') = y \wedge \xi(y, \sigma)!$

$\Rightarrow s\sigma \in L(G) \wedge s \in \bar{K} \wedge \exists s' \in \Sigma^*.\ P(s') = P(s) \wedge \xi(y_0, s'\sigma)!$

$\Rightarrow s\sigma \in L(G) \wedge s \in \bar{K} \wedge \exists s' \in \Sigma^*.\ P(s') = P(s) \wedge s'\sigma \in \bar{K}$

$\Rightarrow s\sigma \in \bar{K}$ ($\because K$ is observable)

CASE 2: $\sigma \in \Sigma - \Sigma_c$

$s\sigma \in L(S/G) \Rightarrow s \in L(S/G) \wedge s\sigma \in L(G)$

$\Rightarrow s \in \bar{K} \wedge s\sigma \in L(G)$

$\Rightarrow s\sigma \in \bar{K}\Sigma_{uc} \cap L(G)$

$\Rightarrow s\sigma \in \bar{K}$ ($\because K$ is controllable)

Next we prove that $L_m(S/G) = K$. First, we show $\forall s \in K$, it must be the case that $s \in L_m(S/G)$.

$s \in K \Rightarrow s \in \bar{K} \wedge s \in L_m(G) \wedge s \in K$ ($\because K \subseteq L_m(G)$)

$\Rightarrow s \in L(S/G) \cap s \in L_m(G) \wedge \xi(y_0, s) \in Y_m$

$\Rightarrow s \in L(S) \wedge s \in L_m(G) \wedge \xi(y_0, s) \in Y_m$

$\Rightarrow s \in L_m(G) \wedge \xi'(y'_0, s) \in Y'_m$

$\Rightarrow s \in L_m(G) \wedge s \in L_m(S)$

$\Rightarrow s \in L_m(S/G)$

Conversely, if $s \in L_m(S/G)$,

$\Rightarrow s \in L(S/G) \cap s \in L_m(G) \wedge s \in L_m(S)$

$\Rightarrow s \in L(S/G) \cap s \in L_m(G) \wedge \xi'(y'_0, s) \in Y'_m$

$\Rightarrow s \in \bar{K} \cap L_m(G) \wedge \exists s' \in \Sigma^*, P(s') = P(s) \wedge \xi(y_0, s') \in Y_m$ (Def. of $\xi'(y', \sigma)$)

$\Rightarrow s \in \bar{K} \cap L_m(G) \wedge s' \in K$

$\Rightarrow s \in K$ ($\because K$ is observable)

■

We can employ the design approach presented in Section 5.2.5 to get an EFSM G_x enforcing the supervision S/G if $L_m(S)$ is $L_m(G)$-closed, or S is a nonmarking supervisor, in which case $L_m(G_x) = K$.

Example 6.1: The foregoing construction is illustrated by mutual exclusion problem under partial observation described below. Consider agents A1 and A2 as shown in Fig. 6.1. The state names refer to a single shared resource, so simultaneous occupancy

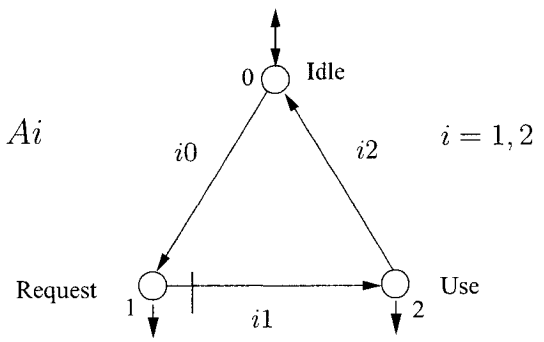

Figure 6.1: Agents subject to mutual exclusion.

of the state pair $(2, 2)$ is prohibited. An additional specification requires that resource usage be 'fair' in the sense of 'first-request-first-use', implemented by means of a queue. It is assumed that the events 11 and 21 (transitions from Request to Use) are unobservable. To find a solution, we start by constructing $A = sync(A1, A2)$ and specification $ASPEC$, shown in Fig. 6.2. The resulting supervisor $ASUPER = supcon(A, ASPEC)$ is displayed in Fig. 6.3.

$ASUPER$ is a nonmarking supervisor. With events 11 and 21 unobservable, application of the construction to $ASUPER$ yields $PASUPER$, with state set $y_0 = \{0\}$; $y_1 = \{1, 3\}$; $y_2 = \{2, 6\}$; $y_3 = \{4, 7\}$ and $y_4 = \{5, 8\}$. The five states are encoded by three boolean variables x_1, x_2 and x_3, shown in Fig. 6.4.

The guard formula and updating functions for each event are listed in Table 6.1.

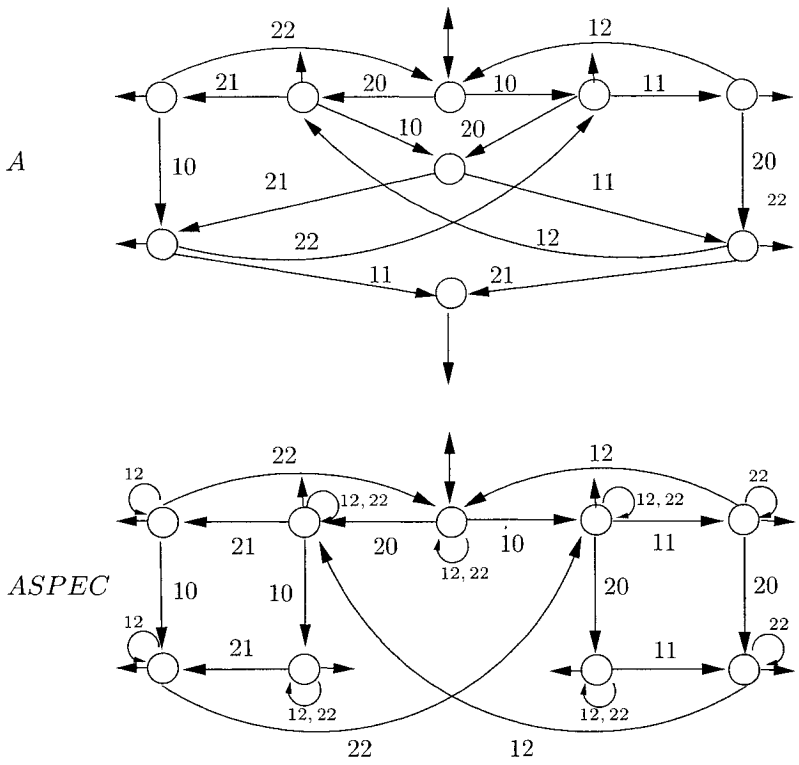

Figure 6.2: Mutual exclusion specification.

The updating function $a_\alpha^{x_i}$, $\alpha \in \{10, 11, 12, 20, 21, 22\}$ and $i = 1, 2, 3$, is listed in (row α , column i) of Table 6.2.

Table 6.1: Guard formulas.

Event	Guard formula
11	$\bar{x}_1 x_3$
21	$(x_1 \oplus x_2)\bar{x}_3$
10, 20, 12, 22	1

It can be verified that the specification is indeed satisfied by the embedded closed-loop system. As an example, consider the string "10, 11, 20, 21". It can be generated in the plant A, but it is illegal since while in use by one process the resource cannot be accessed by the other until it is released. From Table 6.2, we find that after the string

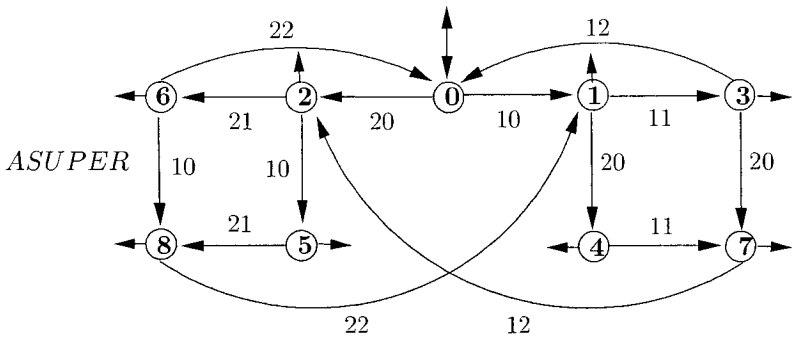

Figure 6.3: Supervisor implementing to mutual exclusion.

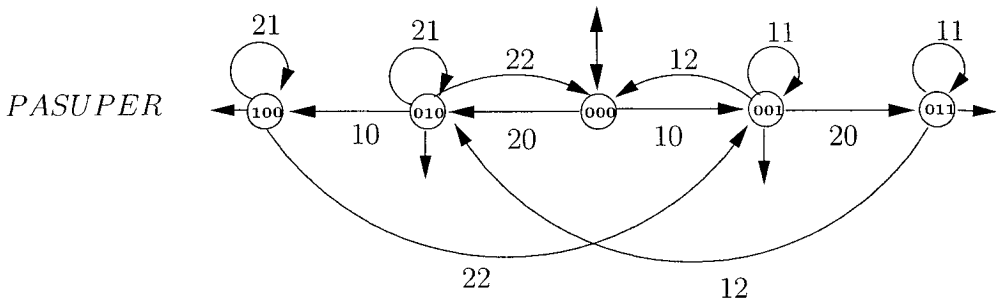

Figure 6.4: Constructed supervisor under partial observation.

"$10, 11, 20$" occurs, the variables are set to $x_1 = 0$, $x_2 = 1$ and $x_3 = 1$. Then, the guard formula of event 21, $(x_1 \oplus x_2)\bar{x}_3$, evaluates to *false*. Therefore, the transition labeled with 21 is disabled by the supervisor after the string "$10, 11, 20$". $\diamond$

6.3 Decentralized embedded control

In decentralized embedded supervisory control problems, we design a set of local, embedded supervisors to monitor and control plant components in order to achieve a global control objective. The main control problem we discuss in this section is related to GPZT defined in [RW92]. Rudie and Wonham have presented a sufficient and necessary condition under which GPZT is solvable. Their original idea is to construct local supervisors S_{pi} with feedback maps ψ_i. Each closed-loop system S_{pi}/G works in parallel to satisfy a global specification. Based on their work, we propose the following

Table 6.2: Updating functions.

event \ *variable*	x_1	x_2	x_3
$11, 21$	x_1	x_2	x_3
10	$\bar{x}_1 x_2 \bar{x}_3$	0	$\bar{x}_1 \bar{x}_2 \bar{x}_3$
12	0	$\bar{x}_1 x_2 x_3$	0
20	0	$\bar{x}_1 \bar{x}_2$	$\bar{x}_1 \bar{x}_2 x_3$
22	0	0	$x_1 \bar{x}_2 \bar{x}_3$

control problem: local plant components are geographically widely separated, while a global specification is given. Assuming the control problem is solvable, we construct a supervisor for each local plant, and extend the plant by the approach presented in Section 5.2.5 to get plant components with control mechanism embedded.

6.3.1 Notations and problem definition

Throughout this section the following notations are used: G_i denotes a local plant over Σ_i, and G denotes the overall plant, which is the synchronous product of local plants. G is defined over $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$. Let $\Sigma_{ci} \subseteq \Sigma_i$ and $\Sigma_{oi} \subseteq \Sigma_i$ denote the local controllable and observable event sets for local supervisor S_{pi}, respectively, such that $\Sigma_{ci} \cup \Sigma_{oi} = \Sigma_i$. We selfloop each event in $\Sigma \setminus \Sigma_i$ at every state in G_i to get $\tilde{G}_i$. The local supervisor S_{pi}, is given by a map $S_{pi} : P_{oi}[L(G_i)] \rightarrow 2^{\Sigma_i}$ that satisfies $S_{pi}[P_{oi}(s)] \supseteq \Sigma_i \setminus \Sigma_{ci}$ for all $s \in L(G_i)$, where $P_{oi} : \Sigma_i^* \rightarrow \Sigma_{oi}^*$ is a natural projection. $\tilde{S}_{pi}$ denotes the supervisor which takes the same control decision as S_{pi} on an event in Σ_{ci}, enables all events in $\Sigma \setminus \Sigma_{ci}$, makes the same transitions as S_{pi} on Σ_{oi} and stays in the same state for events in $\Sigma \setminus \Sigma_{oi}$. $\tilde{S}_{pi}$ can be obtained from S_{pi} by adding $\Sigma \setminus \Sigma_i$ selfloops in its every state. G_{xi} is the closed-loop system enforcing the supervision of $\tilde{S}_{pi}/\tilde{G}_i$. Again, we choose two supervisors for simplicity although the results can be generalized to any fixed number of supervisors.

In this section, the main decentralized control problem we study is called *GPZT*

(Global Problem with Zero Tolerance) described by Rudie and Wonham in [RW92], in which a global specification is satisfied by a set of local controls. Given a plant G over an alphabet Σ, a language K such that $\emptyset \neq K \subseteq L_m(G)$, and sets Σ_{c1}, Σ_{c2}, Σ_{o1}, $\Sigma_{o2} \subseteq \Sigma$, construct local admissible supervisors S_{p1} and S_{p2} such that $\tilde{S}_{p1} \wedge \tilde{S}_{p2}$ is a proper supervisor for G and

$$L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G) = K.$$

Notice that the language K need not be prefix-closed. The solvability of GPZT is related to the *coobservability* property of the specification language.

Motivated by [RW92], we solve GPZT with embedded controllers. We refer to the problem as *EGPZT*, which is short for Embedded Global Problem with Zero Tolerance.

EGPZT: Given distributed plants G_1 and G_2 over an alphabet Σ, a language $\emptyset \neq K \subseteq L_m(G_1 \parallel G_2)$, and sets Σ_{c1}, Σ_{c2}, Σ_{o1}, $\Sigma_{o2} \subseteq \Sigma$, construct local embedded systems G_{x1} and G_{x2} such that

$$L_m(G_{x1} \parallel G_{x2}) = K.$$

Below, we find the EGPZT solution when K is coobservable.

6.3.2 Coobservable specification and decentralized supervisors

In [RW92] S_{pi} are constructed as an automaton with a feedback map provided that K is coobservable, while we present a method to construct $\tilde{S}_{pi}$, which facilitates the implementation of supervisory control map by EFSM.

Construction of $\tilde{S}_{pi}$: Let $M = (Y, \Sigma, \xi, y_0, Y_m)$ be a recognizer for a controllable specification language. For $i = 1, 2$, let Y_i be the set of nonempty subsets of Y. Since M is finite, Y_i is guaranteed to be finite. The supervisor:

$$\tilde{S}_{pi} = (Y_i, \Sigma, \xi_i, y_{0i}, Y_{mi})$$

is given by:

$\forall \sigma \in \Sigma_{oi}, y_i \in Y_i,$

$$\xi_i(y_i, \sigma) = \begin{cases} \{\xi(y.\, s) \mid s \in \Sigma^*, y \in y_i.\, P_{oi}(s) = \sigma\} & \\ & \text{if this is nonempty} \\ undefined & \text{otherwise} \end{cases}$$

$\forall \sigma \in (\Sigma - \Sigma_{oi}) \cap \Sigma_{ci}, y_i \in Y_i,$

$$\xi_i(y_i, \sigma) = \begin{cases} y_i & \text{if } \exists y \in y_i.\, \xi(y, \sigma)! \\ undefined & \text{otherwise} \end{cases}$$

$\forall \sigma \in \Sigma \setminus \Sigma_i,\ y_i \in Y_i,\ \xi_i(y_i, \sigma) = y_i,$

$$y_{0i} := \{\xi(y_0, s) \mid s \in \Sigma^*, P_{io}(s) = \epsilon\}$$
$$Y_{mi} := \{y_i \in Y_i \mid (\exists y \in y_i) y \in Y_m\}$$

Next, we extend $\tilde{G}_i$ to G_{xi} to enforce the supervision $\tilde{S}_{pi}$ on $\tilde{G}_i$. Since $\tilde{S}_{pi}$ is constructed over Σ, like $\tilde{G}_i$ the extended machine G_{xi} is defined over Σ. In Rudie's GPZT the uncontrolled system is a single plant, while in our EGPZT plant consists of several parallel components. To prove $L_m(G_{x1} \parallel G_{x2}) = K$, first we show two preliminary results.

Lemma 5 If G_{x1} and G_{x2} are defined over the same alphabet, $L_m(G_{x1} \parallel G_{x2}) = L_m(G_{x1}) \parallel L_m(G_{x2})$.

Proof: First, we show $L(G_{x1} \parallel G_{x2}) = L(G_{x1}) \parallel L(G_{x2})$. The proof is by induction on the length of strings. First we show that for all $s' \in L(G_{x1} \parallel G_{x2})$, it must be the case that $s' \in L(G_{x1}) \parallel L(G_{x2})$.

- Base: $s' = \epsilon$ trivially holds since G_{x1}, G_{x2} and $G_{x1} \parallel G_{x2}$ are nonempty.
- Inductive step: Let $s' = s\sigma \in L(G_{x1} \parallel G_{x2})$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. Then $s \in L(G_{x1} \parallel G_{x2})$ implies $s \in L(G_{x1}) \parallel L(G_{x2})$ by the inductive assumption. Furthermore, since G_{x1} and G_{x2} are defined over the same alphabet, σ is a common event.

$s' = s\sigma \in L(G_{x1} \parallel G_{x2}) \wedge s \in L(G_{x1}) \parallel L(G_{x2})$

$\Rightarrow \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \wedge s \in L(G_{x1}) \parallel L(G_{x2})$

$\Rightarrow \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \wedge s \in L(G_{x1}) \cap L(G_{x2})$ (inductive assumption)

$\Rightarrow \delta(q_{01}, s\sigma)! \wedge \delta(q_{02}, s\sigma)! \wedge g_{\sigma 1}(V(s)) = 1 \wedge g_{\sigma 2}(V(s)) = 1 \wedge s \in L(G_{x1}) \cap L(G_{x2})$ (Def. 4.2.6)

$\Rightarrow [\delta(q_{01}, s\sigma)! \wedge g_{\sigma 1}(V(s)) = 1 \wedge s \in L(G_{x1})] \wedge [\delta(q_{02}, s\sigma)! \wedge g_{\sigma 2}(V(s)) = 1 \wedge s \in L(G_{x2})]$

$\Rightarrow s\sigma \in L(G_{x1}) \wedge s\sigma \in L(G_{x2})$

$\Rightarrow s' \in L(G_{x1}) \cap L(G_{x2})$

$\Rightarrow s' \in L(G_{x1}) \parallel L(G_{x2})$

We conclude that $L(G_{x1} \parallel G_{x2}) \subseteq L(G_{x1}) \parallel L(G_{x2})$. Next, we show for all $s' \in L(G_{x1}) \parallel L(G_{x2})$, it must be the case that $s' \in L(G_{x1} \parallel G_{x2})$.

- Base: $s' = \epsilon$ trivially holds since G_{x1}, G_{x2} and $G_{x1} \parallel G_{x2}$ are nonempty.

- Inductive step: Let $s' = s\sigma \in L(G_{x1}) \parallel L(G_{x2})$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$. It follows that $s \in L(G_{x1}) \parallel L(G_{x2})$ which implies by the inductive assumption $s \in L(G_{x1} \parallel G_{x2})$. Furthermore,

  $s\sigma \in L(G_{x1}) \parallel L(G_{x2}) \wedge s \in L(G_{x1} \parallel G_{x2})$

  $\Rightarrow s\sigma \in L(G_{x1}) \cap L(G_{x2}) \wedge s \in L(G_{x1} \parallel G_{x2})$

  $\Rightarrow [\delta(q_{01}, s\sigma)! \wedge g_{\sigma 1}(V(s)) = 1 \wedge s \in L(G_{x1})] \wedge [\delta(q_{02}, s\sigma)! \wedge g_{\sigma 2}(V(s)) = 1 \wedge s \in L(G_{x2})] \wedge s \in L(G_{x1} \parallel G_{x2})$

  $\Rightarrow [\delta(q_{01}, s\sigma)! \wedge \delta(q_{02}, s\sigma)!] \wedge [g_{\sigma 1}(V(s)) = 1 \wedge g_{\sigma 2}(V(s)) = 1] \wedge s \in L(G_{x1} \parallel G_{x2})$

  $\Rightarrow \delta(q_0, s\sigma)! \wedge g_\sigma(V(s)) = 1 \wedge s \in L(G_{x1} \parallel G_{x2})$

  $\Rightarrow s' \in L(G_{x1} \parallel G_{x2})$

We get $L(G_{x1}) \parallel L(G_{x2}) \subseteq L(G_{x1} \parallel G_{x2})$. Therefore, we conclude that $L(G_{x1} \parallel G_{x2}) = L(G_{x1}) \parallel L(G_{x2})$. Next we show $L_m(G_{x1} \parallel G_{x2}) = L_m(G_{x1}) \parallel L_m(G_{x2})$.

$s \in L_m(G_{x1} \parallel G_{x2})$

$\Leftrightarrow s \in L(G_{x1} \parallel G_{x2}) \wedge \delta(q_0, s) \in Q_m$

$\Leftrightarrow s \in L(G_{x1}) \cap L(G_{x2}) \wedge \delta_1(q_{01}, s) \in Q_{m1} \wedge \delta_2(q_{02}, s) \in Q_{m2}$ (Def. 4.2.6)

$\Leftrightarrow s \in L_m(G_{x1}) \cap L_m(G_{x2})$

$\Leftrightarrow s \in L_m(G_{x1}) \parallel L_m(G_{x2})$. ■

Next, we show the synchronous product of extended plants do enforce the local supervision on the plant.

Lemma 6 We have: $L_m(G_{x1} \parallel G_{x2}) = L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G)$.

Proof: $L_m(G_{x1} \parallel G_{x2}) = L_m(G_{x1}) \parallel L_m(G_{x2})$ (Lem. 5)

$= L_m(\tilde{S}_{p1}/\tilde{G}_1) \cap L_m(\tilde{S}_{p2}/\tilde{G}_2)$

$= L_m(\tilde{S}_{p1}) \cap L_m(\tilde{G}_1) \cap L_m(\tilde{S}_{p2}) \cap L_m(\tilde{G}_2)$ ($\because \tilde{S}_{pi}$ is a marking supervisor)

$= [L_m(\tilde{S}_{p1}) \cap L_m(\tilde{S}_{p2})] \cap [L_m(\tilde{G}_1) \cap L_m(\tilde{G}_2)]$

$= L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}) \cap L_m(G)$

$= L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G)$. ■

A proof for $L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G) = K$ is provided in [RW92].

We have the following main result.

Theorem 6.3.1 The marked language of overall controlled system is equal to K, i.e.,

$$L_m(G_{x1} \parallel G_{x2}) = K.$$

Proof:

$L_m(G_{x1} \parallel G_{x2})$

$= L_m(\tilde{S}_{p1} \wedge \tilde{S}_{p2}/G)$ (Lem. 6)

$= K$ ■

We end this subsection by a simple example to show the above scheme works when the specification language is controllable and coobservable, and it fails when it is not.

Example 6.2: Plant G_1 defined over $\Sigma_1 = \{\alpha, \gamma\}$, G_2 defined over $\Sigma_2 = \{\beta, \gamma\}$ and the recognizer of specification language M are shown in Fig. 6.5 with $\Sigma_{o1} = \{\alpha\}$, $\Sigma_{o2} = \{\beta\}$ and $\Sigma_{c1} = \Sigma_{c2} = \{\gamma\}$. The overall plant G is the synchronous product of G_1 and G_2, shown in Fig. 6.6 together with $\tilde{G}_1$ and $\tilde{G}_2$. We see that $L_m(G) = \gamma^* + \gamma^*\alpha\gamma^* + \gamma^*\alpha\gamma^*\beta\gamma^* + \gamma^*\beta\gamma^* + \gamma^*\beta\gamma^*\alpha\gamma^*$, while the legal language

$K = L_m(M) = \epsilon + \alpha + \beta + \alpha\beta\gamma^* + \beta\alpha\gamma^*$, which is coobservable w.r.t. G, P_{o1} and P_{o2}. Therefore, by Theorem 4.1 in [RW92] and Theorem 6.3.1 we can find $\tilde{S}_{p1}$ and $\tilde{S}_{p2}$ supervising the local plants to ensure that only language K is marked by the closed-loop system.

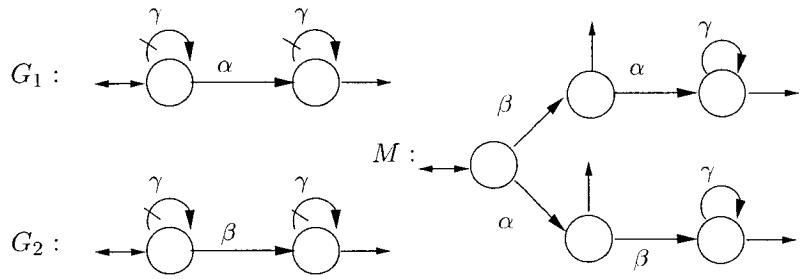

Figure 6.5: Decentralized embedded supervisory control with a coobservable specification.

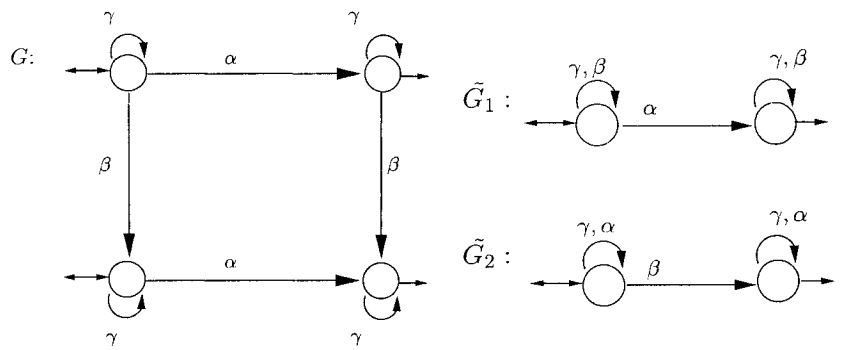

Figure 6.6: Plants G, $\tilde{G}_1$ and $\tilde{G}_2$.

Applying the foregoing constructive procedure yields supervisors $\tilde{S}_{p1}$ and $\tilde{S}_{p2}$ shown in Fig. 6.7. The supervision of $\tilde{S}_{pi}$, $i = 1, 2$, on $\tilde{G}_i$, yields the embedded closed-loop system G_{xi}, shown in Fig. 6.8. The overall controlled system, shown in Fig. 6.9, marks the language $L_m(G_{x1} \parallel G_{x2}) = \epsilon + \alpha + \beta + \alpha\beta\gamma^* + \beta\alpha\gamma^* = K$.

However, in the absence of communication a decentralized solution does not exist when the legal language is not coobservable. For instance, it is not possible to implement $L_m(M')$ shown in Fig. 6.10 if the machines do not "talk" to each other:

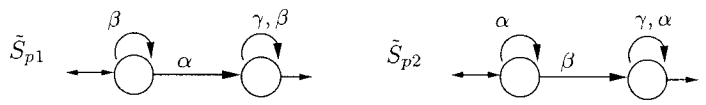

Figure 6.7: Supervisors $\tilde{S}_{p1}$ and $\tilde{S}_{p2}$.

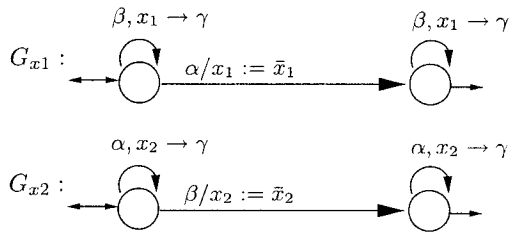

Figure 6.8: Embedded closed-loop system components G_{x1} and G_{x2}.

neither of the local supervisors can be sure whether to enable or disable the event γ in the right-hand side states because, based on its own observation, it cannot distinguish between $\alpha\beta$ and $\beta\alpha$. $\diamond$

6.4 Conclusion

In this chapter we apply the method developed in the previous chapter to supervisory control problem under partial observation. Construction of supervisors in EFSM framework in centralized and decentralized cases are discussed. In the former case, we solve a variant of the SCOPZT presented in [RW92], while we define and investigate EGPZT in the latter case, which is motivated by GPZT in [RW92]. To illustrate the main ideas of this chapter an example is presented.

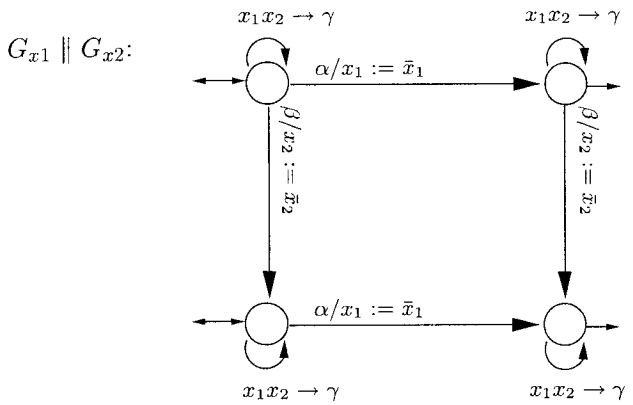

Figure 6.9: Overall controlled system $G_{x1} \parallel G_{x2}$.

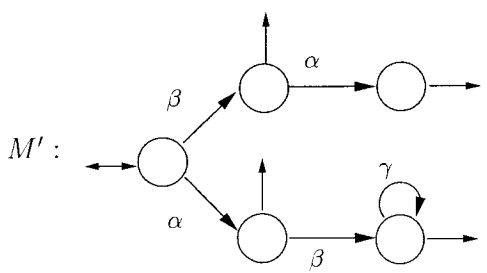

Figure 6.10: A specification that is not coobservable.

Chapter 7

Conclusions and Future Research

7.1 Original contribution

In this thesis we have presented our work on a new approach to implement supervisory control of DES by EFSM as an embedded part of the system to be controlled. In our work we assume that supervisory control theory of Ramadge and Wonham [RW87], [WR87] has been utilized to design an external admissible supervisor in the form of an automaton. The main idea is to abstract control information from the given supervisor and then apply it directly to the plant. The extended system, in EFSM framework, enforces the supervision in the sense that it generates and marks the same languages as the closed-loop system. We also explore the construction of supervisors under partial observation. By implementing the supervisory control with embedded control mechanism, EFSM offers more economical and realistic representations of physical systems.

In conclusion, the original contributions of this thesis are:

1. Modeling a closed-loop DES by an EFSM.

2. Designing embedded supervisors by EFSM.

3. Designing supervisors by EFSM under partial observation.

7.1.1 EFSM modeling

We have built an EFSM G_x by augmenting a regular FSM with a finite set of boolean variables, guard formulas and updating functions. This will later enable us to introduce supervisory control as an embedded part of the original FSM plant. Boolean variables are used to encode the supervisor's states. Event observation is captured by a set of boolean functions that update the values of boolean variables, and control is introduced by guarding events with boolean formulas. A transition labeled with an event is allowed to happen if and only if its guard formula returns *true* (1), and after its occurrence the values of all variables are accordingly updated.

We have defined the closed and marked languages represented by an EFSM in Definition 4.2.3. The language $L(G_x)$ contains the sequence of labels of all directed paths from the initial state that can be traversed along the state transition diagram while respecting guard formulas at all visiting states. The marked language $L_m(G_x)$ is a subset of $L(G_x)$ consisting of strings corresponding to paths that lead to a marker state in the state transition diagram.

We have established that EFSM and regular FSM have equal expressive powers by defining the equivalent regular FSM of an EFSM. They generate and mark the same languages. An example showing the conversion is presented.

The synchronous product of two EFSMs is defined in Section 4.2.3. We have explained that not any two EFSMs can work in parallel and a consistency condition is formulated in Definition 4.2.5. The consistency condition can easily be checked by inspecting updating functions of common events on common variables. The synchronous product of two consistent EFSMs is given by Definition 4.2.6. We have shown two useful properties of synchronous product: that the operations 'extension' and 'synchronous product' commute; and that the operations 'synchronous product' and 'equivalent FSM' commute if an updating function of a private event never changes the value of a common variable.

7.1.2 Design of embedded supervisor by EFSM

The main contribution of this work is to implement a supervisory control map by an EFSM. In our design, the plant and the supervisor are synchronized on the set of events, and the closed-loop (controlled) system is designed in EFSM framework at once. The controller disables or enables a transition by defining a guard formula that evaluates to *false* whenever the transition leads to illegal behavior.

The design consists of three steps. In the first step, we find the required number of boolean variables in Definition 5.2.1. Even when not all bit combinations are used, still k boolean variables are necessary when $k-1 < log_2 N \leq k$, where N is the number of supervisor's states. Without loss of generality, all boolean variables are initialized to *false* (0). Then we label each state in supervisor's automaton with a unique bit combination of k-bit boolean variables, in particular, all bits used in encoding the initial state are 0.

In the second step, guard formula for an event is given in Definition 5.2.4. Since a supervisor can never disable an uncontrollable event, the guard formula for an uncontrollable event always evaluates to *true* regardless of the current values of variables.

In the third step, an updating function triggered by the occurrence of an event for each variable is calculated as prescribed in Definition 5.2.5. The design is completed by extending the original plant with guard formulas and updating functions for each transition.

We have shown that the design prescribed above in effect implements the supervisory control map enforced by the supervisor. The closed language of the resulting EFSM is equal to that of the closed-loop system, and the marked languages are equal if $L_m(S)$ is $L_m(G)$-closed. We have shown that if the supervisor is nonmarking, which is often the case, the $L_m(G)$-closedness condition can be relaxed. Furthermore, we have presented a sufficient condition under which the controlled system is nonblocking.

7.1.3 Supervisory control by EFSM under partial observation

We apply our method to the supervisory control problem under partial observation. Supervisor construction in EFSM framework for centralized and decentralized cases are discussed. In the former case we solve a variant of SCOPZT [RW92], while similar to GPZT [RW92], we introduce and solve EGPZT to design local controllers for plant components to achieve a global control objective.

7.2 Future research

We envisage extending this work in several directions. In what follows we list some problems that may be addressed in future. We believe these interesting issues will lead to more insightful results.

- Currently a guard formula evaluates to *true* if the boolean variables are *equal* to a given set of values. However, in some applications it would be more realistic to enable a transition when the values of boolean variables satisfy a set of inequalities.

- Decentralized supervisory control problem with a general specification that is not necessarily coobservable, where communication between local supervisors may become necessary.

- It will be desirable to develop a software tool to implement the methods presented in this work.

Bibliography

[CD88] Randy Cieslak, C. Desclaux, Ayman S. Fawaz and Pravin Varaiya, "Supervisory Control of Discrete-Event processes with Partial Observations", IEEE Transactions on Automatic Control, vol.33, No.3, March 1988.

[CK96] Kwang-Ting Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model". *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Volume 1 Issue 1, January 1996.

[CL00] Yi-Liang Chen and Feng Lin, Modeling of Discrete Event Systems Using Finite State Machines with Parameters". *Proceedings of the 2000 IEEE International Conference on Control Applications Anchorage*, Alaska, USA September 25-27,2000.

[FW88] F.Lin and W.M.Wonham, On observability of discrete-event systems, Inf.Sci., vol. 44, pp. 173-198, 1988.

[FW90] Feng Lin and W.M.Wonham, "Decentralized control and coordination of decrete-event systems with partial observation", IEEE Transaction on Automatice Control, Vol. 35, No.12, December 1990.

[FW95] F.Lin and W.M.Wonham, Supervisory Control of Timed Discrete-Event Systems under Partial Observation. IEEE Transactions on Automatic Control, vol.40, No.3, March 1995.

[GS00] George Barrett and Stephane Lafortune, "Decentralized Supervisory Con-

trol with Communicating Controllers", IEEE Transaction on Automatic Control, Vol. 45, No. 9, SEPTEMBER 2000.

[HC98] R. C.-Y. Huang and K.-T. Cheng, "A new extended finite state machine (efsm) model for rtl design verification", International HLDVT Workshop, pp. 47C53, November 1998.

[HO91] HOLTZMANN, G. 1991. Design and Validation of Computer Protocols. Prentice-Hall, New York, NY.

[KR69] K.A.Bartlett, R.A.Scantlebury, and P.T.Wilkinson, A note on reliable fullduplex transmission over half-duplex links, Communications of the ACM, vol. 12, pp. 260-261, 1969.

[LW02] W. C. Lai, "Embedded software-based self-test for system-on-a-chip design", Ph.D. dissertation, University of California, Santa Barbara, March 2002.

[PG04] Peyman Gohari, "Alternating Bit Protocol: A Supervisory Control Perspective". To appear in the *proceedings of Decentralized Discrete Event Systems: Structure, Communication and Control*, Banff International Research Station, May 2004.

[RW87] P.J.Ramadge and W.M.Wonham, "Supervisory control of a class of discrete event processes". *SIAM J. Control and Optimization*, 25(1):206-230,1987.

[RW92] K. Rudie and W. M.Wonham, "Think globally, act locally: Decentralized supervisory control, IEEE Transaction on Automatic Control, vol. 37, pp. 1692- 1708, Nov. 1992.

[SF00] Stephane Lafortune, "Centralized and Decentralized Control of Partially-Observed Discrete-Event Systems: The State of the Art and Some New Results", ARO Workshop 2000 on Intelligent Systems Sponsored by Army Research Office 8-9 December 2000.

[TT05] Design Software: XPTCT (Version 862 for Windows 95/98/2000/XP, updated 2005.07.01).

[UM05] UMDES-LIB software library for Windows.

[WC68] W.C.Lynch, "Computer system: Reliable full-duplex file tranmisstion over half-duplex telephone line", Commun. ACM, vol. 11, No. 6, pp. 407C410, 1968.

[Won04] W.M.Wonham, "Supervisory control of discrete-event systems". http://www.control.toronto.edu/people/profs/wonham/wonham.html, 2004.

[WR87] W.M.Wonham and P.J.Ramadge, "On the supermal controllable sublanguage of a given language". *SIAM J. Control and Optimization*, 25(3):637-659, May 1987.

[YF00] Yi-Liang Chen and Feng Lin, "Modeling of discrete event systems using finite state machines with parameters", Proc. 9th IEEE Intl. Conf. on Control Applications, September 2000.