Task Scheduling Using Priority-Based Supervisory Control of DES

Abdollah Saffar

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
(Electrical & Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

September 2004

Your file Votre référence
ISBN: 0-612-94709-2
Our file Notre référence
ISBN: 0-612-94709-2

Abstract

Task Scheduling Using Priority-Based Supervisory Control of DES

Abdollah Saffar

With supervisory control, it is possible to restrict the behavior of a system by disabling a subset of controllable events. Our objective is to develop a well-established formal method for preemptive and non-preemptive task scheduling based on supervisory control. We also examine how formal methods can help address issues such as priority-based scheduling. To achieve these objectives we first introduce a model for each tasks and desired requirements. To address the priority-based scheduling we define a priority relation as a specification, which assigns a priority to each task. The task and specification models are then combined separately into a campsite model. We finally obtain a supervisory control that guaranties all desires requirements are met by using the supervisory control theory.

Acknowledgment

I would like to express my deep gratitude towards Professor Peyman Gohari, my supervisor, for his invaluable assistance and constant guidance throughout this research, and for his advice and constructive criticism during the preparation of this thesis.

I devote this work to my parents for their love.

List of Figures

List of Tables

List of Acronyms and Abbreviations

DES	Discrete-Event Systems.
DTA	Discrete Timed Automata.
EDF	Earliest Deadline First.
NPRE	Non-preemptive.
NPRI	Non-priority.
PRE	Preemptive.
PRI	Priority.
RM	Rate Monotonic.
SCH	Schedule.
SPEC	Specification.
SUP	Supremal Supervisory Control.
SUPSYSPE	Priority-based Supremal Supervisory Control.

Chapter 1

Discrete-Event Systems Preliminaries

1.1 Introduction

A Discrete-Event System (DES) is a dynamic system that evolves in accordance with instantaneous occurrence of physical events. An event can be either *controllable* or *uncontrollable*. An external agent called *supervisor*, which can disable a subset of controllable events, represents the control feature. The general problem of control theory is to find a supervisor such that the closed-loop behavior of environment and supervisor meets the specification of some desired behavior. In this chapter we will study the essentials of DES and the supervisory control theory of DES.

1.2 Supervisory Control Theory

In this section we summarize basic concepts of discrete-event system and supervisory control theories, the latter introduced by Ramadge and Wonham. For more information see [1], [2].

1.2.1 Languages

An alphabet is a finite set of symbols. For an alphabet Σ, let Σ^* denote the set of all finite strings (or words) of the form $\sigma_1\sigma_2...\sigma_k$, where $\sigma_i \in \Sigma$ for $1 \le i \le k$. When $k = 0$ we have an empty string, denoted by ε. A language over Σ is any subset $L \subseteq \Sigma^*$.

For $s \in \Sigma^*$, we say $r \in \Sigma^*$ is a prefix of s, and write $r \leq s$, if $s = ru$ for some $u \in \Sigma^*$. The prefix-closure $\overline{L}$ of a language $L \subseteq \Sigma$ is the set of all prefixes of strings in L, i.e.:

$$\overline{L} = \{r \in \Sigma^* | r \leq s \text{ for some } s \in L\}.$$

The left quotient of a language $L \subseteq \Sigma^*$ by a word $s \in \Sigma^*$ is defined by $L/s = \{r \in \Sigma^* | sr \in L\}$. The left quotient describes possible continuations of a word in a language.

1.2.2 Discrete-Event Systems

When the state space of a system is naturally described by a discrete set such as any subset of natural numbers, and state transitions are only observed at discrete points in time, we associate state transitions with 'events' and talk about a 'discrete event system'.

A discrete event system (DES) is a dynamic system whose state space is discrete and whose state can only change in response to instantaneous occurrence of events over time.

Sample paths of DES are typically piecewise constant functions of time. Conventional differential equations are not suitable for describing such 'discontinuous' behavior. The main elements of a DES are (a) a discrete state space we will usually denote by X, and (b) an event set we will usually denote by Σ.

Examples of DES, which we frequently encounter, include computer systems, communication networks, manufacturing systems, and traffic control systems. A 'queuing system' is often used as a building block for modeling many types of DES. The abstract behavior of complex dynamic systems with continuous variables is often modeled as a DES for the purpose of supervisory control, monitoring, and diagnostics [3].

A discrete-event system (DES) is modeled as a generator of two formal languages over an alphabet. Formally, a DES is a tuple $D = (\Sigma, L, L_m)$ where Σ is an alphabet of

events, L is a prefix-closed language over Σ, and $L_m \subseteq L$ is another language over Σ, called *marked language*. The language L describes all possible behaviors of D, while the marked language of a DES is used to describe completed tasks. A system is said to be *blocking* if it can execute a word that cannot be completed to a string of the marked behavior. Formally, a DES $D = (\Sigma, L, L_m)$ is said to be *non-blocking* if the prefix closure of the marked behavior is equal to L; otherwise D is said to be *blocking*. In other words, we say D is non-blocking if for all $s \in L$ there exists $r \in \Sigma^*$ such that $sr \in L_m$.

A DES can also be expressed as the generator of a formal language. Formally this is a tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is a set of states (at most countable), Σ is a finite alphabet of events, $q_0 \in Q$ is the initial state, $Q_m \subseteq Q$ is the subset of marked states, and $\delta : Q \times \Sigma \to Q$ is a partial transition function. Denote by $\delta(q,\sigma)!$ the fact that δ is defined on $(q,\sigma) \in Q \times \Sigma$. The transition function is extended to $Q \times \Sigma^* \to Q$ in the following recursive manner: For $s \in \Sigma^*, \sigma \in \Sigma$ and $q \in Q$ we define:

- $\delta(q, \varepsilon) = q$
- $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$, whenever both are defined.

The languages associated with G are $L(G)$ and $L_m(G)$. The language $L(G)$ is defined as the set of all strings of events corresponding to sequences of state transitions leading from the initial state to any state of G:

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

The marked language $L_m(G)$ is the set of all strings of events corresponding to sequences of state transitions leading from the initial state to a marked state of G:

$$L_m(G) = \{s \in L(G) \mid \delta(q_0, s) \in Q_m\}.$$

The automaton G can be used as an alternative representation for the DES $D = (\Sigma, L(G), L_m(G))$. A transition diagram may be used to graphically represent an automaton. The nodes of such a graph represent the states of the automaton, and the arcs labeled with event symbols represent transitions.

1.2.3 Operations on Automata

In order to analyze DES modeled as automata, we need to have a set of operations that allows us to combine two or more automata, as well as operations on a single automaton in order to modify its state transition diagram appropriately.

In this section we define reachable *part* and coreachable *part*, and one of the important operations on DES (or its language) called *synchronous composition.*

- **Reachable Part**

From the definition of $L(G)$ and $L_m(G)$, we see that we can delete from G all states that are not *accessible* or *reachable* from q_0 by some string in $L(G)$, without affecting the languages generated or marked by G. The resulting automaton is called the 'reachable' part of G.

- **Coreachable Part**

A state q of G is said to be *coreachable* to Q_m, or simply 'coreachable', if there is a string in $L_m(G)$ that goes through q; in other words there is a path in the state transition diagram of G from state q to a marked state. The 'coreachable' part of G can be obtained by deleting all states of G that are not 'coreachable'.

- **Synchronous composition**

Next we define synchronous composition, denoted by an operator *sync:* $Ð \times Ð \rightarrow Ð$ where Ð is the class of all discrete-event systems defined in section 1.2.2.

Let $G_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{mi})$, $i = 1,2$, be two DES. The synchronous composition of G_1 and G_2 is an automaton: $sync(G_1, G_2) := (Q, \Sigma, \delta, q_0, Q_m)_{rch}$ [1], where

- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $Q = Q_1 \times Q_2$
- $q_0 = (q_{01}, q_{02})$
- $Q_m = Q_{1m} \times Q_{2m}$

For $q_1, q_2 \in Q$ and $\sigma \in \Sigma$ we define $\delta((q_1, q_2), \sigma)$ as follows:

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

In synchronous composition, a common event (that is, an event in $\Sigma_1 \cap \Sigma_2$) can be executed only if the two automata can both execute it simultaneously. Thus, the two automata are 'synchronized' on common events. The other 'local' events, that is, those in $(\Sigma_1 - \Sigma_2) \cup (\Sigma_2 - \Sigma_1)$, are not subject to such a constraint and can be executed independently whenever possible. It can be verified that synchronous composition is commutative and associative [3].

In order to precisely characterize the languages generated and marked by $sync(G_1, G_2)$ in terms of those of G_1 and G_2, let us define natural projection maps:

$$P_i : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_i^* \quad \text{for} \quad i = 1,2$$

as follows:

$$P_i(\varepsilon) := \varepsilon$$

[1] rch: reachable part.

$$P_i(e) := \begin{cases} e & \text{if } e \in \Sigma_i \\ \varepsilon & \text{if } e \notin \Sigma_i \end{cases}$$

$$P_i(se) := P_i(s)P_i(e) \text{ for } s \in (\Sigma_1 \cup \Sigma_2)^*, e \in (\Sigma_1 \cup \Sigma_2).$$

Given two alphabets where one is a subset of the other, namely $\Sigma_1 \cup \Sigma_2$ and Σ_i in this case, natural projection erases events in a string formed from the larger alphabet $(\Sigma_1 \cup \Sigma_2)$ that do not belong to the smaller alphabet (either Σ_1 or Σ_2). We will also be working with the corresponding inverse maps:

$$P_i^{-1} : \Sigma_i^* \to 2^{(\Sigma_1 \cup \Sigma_2)^*}$$

defined as follows:

$$P_i^{-1}(t) := \{ s \in (\Sigma_1 \cup \Sigma_2)^* : P_i(s) = t \}.$$

Given a string formed over the smaller alphabet (Σ_i), the inverse projection returns the set of all strings formed over the larger alphabet $(\Sigma_1 \cup \Sigma_2)$ that project, under P_i, to the given string.

The projections P_i and their inverses P_i^{-1} are extended to languages by simply applying them to all the strings in a language. For $L \subseteq (\Sigma_1 \cup \Sigma_2)^*$,

$$P_i(L) := \{ t \in \Sigma_i^* : \exists s \in L. P_i(s) = t \}.$$

and for $L_i \subseteq \Sigma_i^*$,

$$P_i^{-1}(L_i) := \{ s \in (\Sigma_1 \cup \Sigma_2)^* : \exists t \in L_i . P_i(s) = t \}.$$

Note that $P_i\left[P_i^{-1}(L)\right] = L$, but in general $L \subseteq P_i^{-1}\left[P_i(L)\right]$ [3].

Using the above-defined projections, we can characterize the languages resulting from a parallel composition:

1. $L(sync(G_1, G_2)) = P_1^{-1}[L(G_1)] \cap P_2^{-1}[L(G_2)]$.
2. $L_m(sync(G_1, G_2)) = P_1^{-1}[L_m(G_1)] \cap P_2^{-1}[L_m(G_2)]$.

1.2.4 Supervisors

The general problem of supervisory control of DES consists of finding a supervisor to restrict the behavior of a given DES (plant) in such a way that it meets the control objectives (specification).

We assume that an event can be either *controllable* or *uncontrollable*. A supervisor can disable a transition if and only if it is labeled with a controllable event. Formally, let $D = (\Sigma, L, L_m)$ be a DES, and let $\Sigma = \Sigma_c \cup \Sigma_u$ be a partition of Σ into controllable and uncontrollable events, respectively. A supervisor S for D is a function $S: L \rightarrow 2^{\Sigma_c}$ from the language of D to the power set of Σ_c. The supervisor maps each word of the language to a subset of controllable events, which are to be enabled after the occurrence of that word.

The closed-loop controlled system is denoted by S/D (D under supervision of S) and it is defined as $(\Sigma, S/L, S/L_m)$, where S/L is the smallest language such that:

- $\varepsilon \in S/L$, and
- $s\sigma \in S/L$ if and only if $s \in S/L$, $s\sigma \in L$, and $\sigma \in \Sigma_u \cup S(s)$.

The marked language is defined as $S/L_m = (S/L) \cap L_m$. A supervisor *restricts* the behavior of a given DES. Therefore, the language of the controlled system is contained in the language of the uncontrolled system.

An additional requirement for the controlled system is to be *non-blocking*. A supervisor S for D is said to be *non-blocking* if the closed-loop system S/D is non-blocking. Ramadge and Wonham show that if there exists a non-blocking supervisor satisfying a given specification then there must exist a least restrictive supervisor satisfying the specification as well. The least restrictive supervisor S disables as few events as possible, that is to say, if there is a supervisor S' which also meets the desired specification and enables the set $S'(s)$ after the occurrence of a word $s \in L$, it must be the case that $S'(s) \subseteq S(s)$. Ramadge and Wonham also show that in case of regular languages the least restrictive supervisor is computable through a fix-point algorithm in polynomial time [1], [2].

Chapter 2

Priority-Based Supervisory Control in Discrete-Event Systems

2.1 Introduction

Sometimes it is desirable to design a controller that not only disables controllable events, but also selects an event with the highest priority among the enabled events. For instance, a CPU may be required to give preference to a task with the shortest response time. This requirement can be addressed by defining a priority relation that grants the highest priority to such a task. Part of our mission is to model task priorities in DES framework.

This chapter is organized as follows: The notion of a priority relation is introduced in Section 2. Priority-based supervisory control of DES is discussed in Section 3, and a practical application is considered in Section 4 through an example.

2.2 Priority Relation

In practice, it is sometimes desirable to have a controller that not only disables a subset of controllable events, but also selects exactly one with the highest "priority" among the set of enabled controllable events that are physically possible in the plant. This is useful, for instance, when controllable events selected by the controller are interpreted as commands given to the plant. Then arbitrary priorities must be assigned to commands to ensure that they are executed one at a time. Another example where priorities are not arbitrary is when the event triggering of a component with a faster response time is given a higher priority.

In such a case, we would like to design a supremal supervisory control such that the supervisor grants the chance of occurrence to events with 'higher priorities' when they compete with other events.

In order to formalize the problem we define a priority relation P on the set of events. If an event σ' can overtake σ when the plant gives them both a chance to occur, we say the event σ' has a *higher priority* than the event σ, denoted by $(\sigma',\sigma)\in P$. To present a formal definition, let us assume that Σ is the event set of a given system. A priority relation P is any subset of $\Sigma\times\Sigma$, with the interpretation that:

$$\forall\sigma,\sigma'\in\Sigma.(\sigma',\sigma)\in P \Leftrightarrow \sigma' \text{ has a higher priority than } \sigma .$$

Observe that priority of σ' is not higher than σ, i.e. $\neg(\sigma',\sigma)\in P$ iff σ has a higher priority than σ', or the two priorities are not comparable. It is noticeable that $[(\sigma',\sigma)\in P\wedge(\sigma,\sigma'')\in P]$ does not necessarily imply $(\sigma',\sigma'')\in P$, i.e. we do not require that P be transitive.

Finally, we introduce the definition of *conformance* of a priority relation P with respect to a language L. Let $P\subseteq\Sigma\times\Sigma$ be a priority relation on Σ and $K\subseteq\Sigma^*$. We say K *conforms* with P with respect to L, denoted by $K \models_L P$, iff:

$$\forall s\in\Sigma^*,\forall\sigma,\sigma'\in\Sigma.(\sigma',\sigma)\in P\wedge s\sigma\in K\Rightarrow s\sigma'\notin L.^{2}$$

From now on, we assume that L is fixed and therefore it may be dropped in our future references.

Note that if $K \models P$ then given $s\in\Sigma^*$ and $\sigma\in\Sigma$ we have:

$$\begin{aligned}
&\forall\sigma'\in\Sigma.(\sigma',\sigma)\in P\wedge s\sigma\in K\Rightarrow s\sigma'\notin L\\
&\Leftrightarrow\forall\sigma'\in\Sigma.\neg(\sigma',\sigma)\in P\vee s\sigma\notin K\vee s\sigma'\notin L\\
&\Leftrightarrow\forall\sigma'\in\Sigma.s\sigma\notin K\vee\neg[(\sigma',\sigma)\in P\wedge s\sigma'\in L]\\
&\Leftrightarrow s\sigma\in K\Rightarrow\forall\sigma'\in\Sigma.\neg[(\sigma',\sigma)\in P\wedge s\sigma'\in L].
\end{aligned}$$

[2] Appendix A explains why conformance is defined in this manner.

The last statement simply requires that an event σ can happen at a string s of the specification only if all events with a higher priority than σ are not eligible in the plant.

One of the interesting properties of conformance with a priority relation is that if a language conforms with a priority relation, then all its sublanguages conform with the priority relation as well. In other words, conformance is inherited downwards.

Lemma 2.2.1

Let $P \subseteq \Sigma \times \Sigma$ be a priority relation on Σ, L and M be languages over Σ, and M conforms with P with respect to L. Then any sublanguage of M conforms with P with respect to L.

Proof:

Let us assume that $K \subseteq M$. We must show that $K \models P$, i.e.:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in K \Rightarrow s\sigma' \notin L.$$

We show that the truth of the right-hand side follows from the truth of the left-hand side. Since $K \subseteq M$ we have:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. [(\sigma', \sigma) \in P \wedge s\sigma \in K \Rightarrow (\sigma', \sigma) \in P \wedge s\sigma \in M]. \text{ (i)}$$

By definition $M \models P$ is equivalent to:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in M \Rightarrow s\sigma' \notin L. \text{ (ii)}$$

By transitivity of $(\Rightarrow)$ it follows from (i) and (ii) that:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in K \Rightarrow s\sigma' \notin L. \quad \therefore$$

2.3 Priority-Based Supervisory Control

We assume that plant $\underline{L}$ and specification $\underline{E}$ are modeled as generators of formal languages L and E, respectively, where:

$$\underline{L} = (X, \Sigma, \xi, x_0)$$
$$\underline{E} = (Y, \Sigma, \eta, y_0)$$

We use 4-tuple models because we assume that all states are marked, i.e. $X = X_m$. It is important to note that this assumption does not affect the generality of our approach and it has been considered for the sake of convenience. The blocking problem can be addressed separately.

Let $P \subseteq \Sigma \times \Sigma$ be a priority relation, and the languages generated by $\underline{L}$ and $\underline{E}$ be denoted by L and E, respectively. Our interest is to design a supremal supervisory control such that plant under supervision satisfies the specification $\underline{E}$ while conforming with the priority relation P.

Assume for now that all events are controllable. When $L \cap E$ conforms with the priority relation, $\underline{L} \times \underline{E}$ can serve as a supervisor to control the system in such a way that all desired objectives are met.

When $L \cap E$ does not conform with the priority relation, one seeks one of its sublanguages that does exactly that. Then it is natural to ask whether a largest such sublanguage exists. To answer this question let us consider the class of sublanguages of $L \cap E$ that confirm with P, denoted by K, that is:

$$\mathsf{K} = \{K \subseteq L \cap E | K \models P\}.$$

As the following proposition suggests, K has a largest element.

Preposition 2.3.1

K is nonempty and $\mathsf{K}\uparrow = \bigcup_{K \in \mathsf{K}} K \in \mathsf{K}$, where $\mathsf{K}\uparrow$ is the supremal sublanguage of $L \cap E$ conforming with P.

Proof:

1. We must show K is nonempty.
   Since $\varnothing \subseteq L \cap E$ and $\varnothing \models P$ it follows that $\varnothing \in \mathsf{K}$, which means that K is nonempty.

2. We must show that $\mathsf{K}\uparrow$ is the supremal sublanguage of $L \cap E$ conforming with P, that is, 1) $\mathsf{K}\uparrow \subseteq L \cap E$ which is true since $L \cap E$ is an upper bound for K, and 2) $\mathsf{K}\uparrow \models P$, i.e.:

   $$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in \mathsf{K}\uparrow \Rightarrow s\sigma' \notin L.$$

Let $s \in \Sigma^*$ and $\sigma, \sigma' \in \Sigma$ be such that $(\sigma', \sigma) \in P$ and $s\sigma \in \mathsf{K}\uparrow$. Then $s\sigma \in \bigcup K$, i.e. for some $K \in \mathsf{K}$ we must have $s\sigma \in K$. But since $K \models P$, this implies $s\sigma' \notin L$, as desired. $\therefore$

To compute $\mathsf{K}\uparrow$, we first obtain $\underline{L}_p$ by *applying priorities* to L, that is, whenever several events are eligible to occur at a state, the ones with maximal priorities are kept while the others are removed. The following definition formalizes the procedure.

Definition 2.3.1

The *prioritized system*, denoted by $\underline{L}_p$, is a four tuple $\underline{L}_p = (X, \Sigma, \xi_p, x_0)$, where X, x_0, and Σ are as in $\underline{L}$, and the partial transition function ξ_p is determined according to:

$$\forall x \in X, \sigma \in \Sigma . \xi_p(x,\sigma)! \text{ iff } \xi(x,\sigma)\,! \wedge \forall \sigma' \in \Sigma \neg [\xi(x,\sigma')\,! \wedge (\sigma',\sigma) \in P],$$

in which case $\xi_p(x,\sigma) := \xi(x,\sigma)$. ∴

In other words $\underline{L}_p$ is obtained from $\underline{L}$ by simply removing transitions labeled with low priority events whenever transitions labeled with higher priority events are present.

Note that since the empty string is not reached by any event, it follows from the definition that:

$$\varepsilon \in L \Rightarrow \varepsilon \in \underline{L}_p.$$

Denote by $\underline{L}_p \times \underline{E}$ the synchronous product of $\underline{L}_p$ and $\underline{E}$. We propose $\underline{L}_p \times \underline{E}$ as a candidate for a final specification that can be used in order to calculate the supremal supervisory control such that plant under supervision conforms with the priority relation P and satisfies the original specification $\underline{E}$.

The following theorem states that $\underline{L}_p \times \underline{E}$ generates the supremal sublanguage of $L \cap E$ that conforms with the priority relation P (i.e. $K \uparrow$). Thus, $\underline{L}_p \times \underline{E}$ may indeed be used as a 'final specification' to compute a supervisory control that satisfies the specification and conforms with the priority relation.

Theorem 2.3.1

The language generated by $\underline{L}_p \times \underline{E}$ is equal to the supremal sublanguage of $L \cap E$ that conforms with P.

Proof:

We show that $L_p \bigcap E = \mathsf{K}\uparrow$, i.e.:

1. $L_p \bigcap E \in \mathsf{K}$ (which implies $L_p \bigcap E \subseteq \mathsf{K}\uparrow$) and
2. $L_p \bigcap E$ is an upper bound of K (which implies $\mathsf{K}\uparrow \subseteq L_p \bigcap E$).

1.i. We first show that $L_p \bigcap E \subseteq L \bigcap E$. Note that:

$$\forall x \in X, \sigma \in \Sigma . \xi_p(x,\sigma)! \Rightarrow \xi(x,\sigma)!$$
$$\Rightarrow \forall s \in \Sigma^* . \xi_p(x_0,s)! \Rightarrow \xi(x_0,s)!$$
$$\Leftrightarrow \forall s \in \Sigma^* . s \in L_p \Rightarrow s \in L$$
$$\Leftrightarrow L_p \subseteq L$$
$$\Rightarrow L_p \bigcap E \subseteq L \bigcap E.$$

1.ii. We must show $L_p \bigcap E \models P$, i.e.

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma . (\sigma', \sigma) \in P \wedge s\sigma \in L_p \bigcap E \Rightarrow s\sigma' \notin L.$$

Let $(\sigma', \sigma) \in P$ and $s\sigma \in L_p \bigcap E$. It follows that $s\sigma \in L_p$. Denote $x := \xi_p(x_0, s) = \xi(x_0, s)$. Since $(\sigma', \sigma) \in P$ and $\xi_p(x,\sigma)!$ from the definition of L_p it follows that $\neg \xi(x,\sigma')$! , i.e. $s\sigma' \notin L$.

From (1.i.) & (1.ii.) it follows that $L_p \bigcap E \in \mathsf{K}$. (1)

2. Next we show that $L_p \bigcap E$ is an upper bound of K.

Let $K \in \mathsf{K}$, i.e. $K \subseteq L \bigcap E$ and $K \models P$. We must show $K \subseteq L_p \bigcap E$, i.e.

$$\forall s \in \Sigma^* . s \in K \Rightarrow \mathrm{s} \in L_p \bigcap E.$$

We prove this by induction on the length of s.

i. Base $s = \varepsilon$:

$$\varepsilon \in K$$
$$\Rightarrow \varepsilon \in L \cap E$$
$$\Rightarrow (\varepsilon \in L \wedge \varepsilon \in E)$$
$$\Rightarrow (\varepsilon \in L_p \wedge \varepsilon \in E)$$
$$\Rightarrow \varepsilon \in L_p \cap E.$$

ii. Inductive step:

Suppose we have established for any $s \in \Sigma^*$ of length n that:

$$s \in K \Rightarrow \mathrm{s} \in L_p \cap E$$

A string of length $n+1$ is of the form $s\sigma$ for some $\sigma \in \Sigma$, thus we must show:

$$s\sigma \in K \Rightarrow \mathrm{s}\sigma \in L_p \cap E.$$

Let $s\sigma \in K$. It follows that $\mathrm{s}\sigma \in L \cap E$. Since K is prefix-closed, we must have $s \in K$, and therefore by the induction assumption $s \in L_p \cap E$. Denote $x := \xi_p(x_0, s) = \xi(x_0, s)$. Since $K \models P$ and $s\sigma \in K$, we have:

$$\forall \sigma' \neg [(\sigma', \sigma) \in P \wedge s\sigma' \in L]$$
$$\Leftrightarrow \forall \sigma' \neg [(\sigma, \sigma') \in P \wedge \xi(x, \sigma')!]. \quad (1)$$

Also,

$$\mathrm{s}\sigma \in L \Leftrightarrow \xi(x, \sigma)!. \quad (2)$$

From (1) and (2) it follows that $\xi_p(x,\sigma)!$, i.e. $s\sigma \in L_p$. This, together with $s\sigma \in E$ establishes that $s\sigma \in L_p \cap E$. Hence we have proved that $L_p \cap E$ is an upper bound of K, as desired. ∴

We have proved that $L_p \cap E = K\uparrow$, which implies that $\underline{L_p} \times \underline{E}$ could be used as the final specification in order to calculate a supremal priority-based supervisory control using the supcon procedure in 'TTCT'. [4]

Recall from [2] that output of supcon is a sublanguage of plant and specification. Since our new specification $L_p \cap E$ conforms with P, it follows from Lemma 2.2.1 that any of its sublanguages, including that returned by supcon, conforms with P, i.e. we have the following result:

Corollary 2.3.1:

Given a pair of plant and specification *(L, E)* and a priority relation P, the supremal controllable sublanguage of E conforming with P exists, and is equal to $\underline{S}$, where:

$\underline{S} = \text{supcon}(\underline{L}, \underline{E} \times \underline{L_P})$.

2.4 Priority-Based Supervisory Control: An Application

In this section, we will see an application of priority-based supervisory control discussed earlier. We illustrate this through a practical example, which deals with a small part of a transportation system for a mining site [5]. The plant is composed of one shovel and three trucks, all of 100 tons capacities, located at the same ore mine. In the actual setup, a coordinator performs the dispatch of trucks to the site. The dispatch rules consist of a mixture of the dispatcher's experience and a set of heuristics. Clearly, the possibility of various scenarios can exceed the dispatcher's capability to handle fine details of the overall process. It is natural to seek a systematic procedure for making decisions that will ensure effective use of the existing recourses (e.g. trucks). The DES representing the

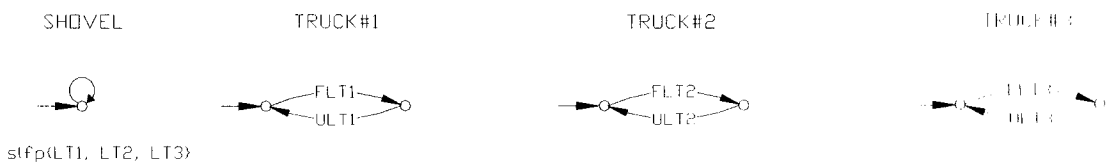
plant is composed of Shovel, Truck #1, Truck #2 and Truck #3, as shown in Figure 2.1. The system events are summarized in Table 2.1. The resulting plant SYS shown in Figure 2.2 is the synchronous product of all finite-state machines in Figure 2.1. DES model of SYS can be obtained using the sync procedure in 'TTCT' software as follows.

T12 = sync (TRUCK #1, TRUCK #2)

T123 = sync (T12, TRUCK #3)

SYS = sync (T123, SHOVEL).

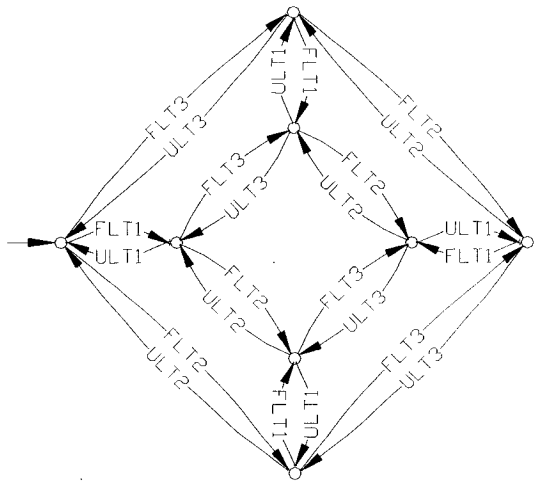
Assume that all states are marked, all events are controllable, and we let 'slfp{}' denote self-loop at every state of the DES with events between brackets.



Figue 2.1: The DES models of the shovel and trucks.

Event	Description	Event	Description
LT1	Loading of Truck #1	FLT3	Full loading of Truck #3
LT2	Loading of Truck #2	ULT1	Unloading of Truck #1
LT3	Loading of Truck #3	ULT2	Unloading of Truck #2
FLT1	Full loading of Truck #1	ULT3	Unloading of Truck #3
FLT2	Full loading of Truck #2		

Table 2.1: Event descriptions for the combined system SYS.



slfp ⟨LT1, LT2, LT3⟩

Figure 2.2: Automaton model for the combined system (SYS).

The required specifications are stated below:

1. Only one truck is loaded at a time.
2. The trucks must not underflow or overflow.
3. Since Truck #3 is faster than Truck #1 and Truck #2, unloading of Truck #3 has a higher priority than unloading of Truck #1 and Truck #2.

As illustrated in Figure 2.3, the specifications for the loading capacity, and the overflow and underflow of truck # i are represented by automaton. E_i , $i = 1,2,3$.

Take $E1$ for example. Referring to Figure 2.3, since the filled state has not been self-looped with events LT2 (Loading of Truck #2) and LT3 (Loading of Truck #3), $E1$ disallows the loading of Trucks #2 and #3 while Truck #1 is being loaded. In addition, $E1$ disables the event ULT1 (Unloading of Truck #1) from occurring before the event FLT1 (Full Loading of Truck #1) to prevent underflow. After Truck #1 is fully loaded, it also disables the event LT1 (Loading of Truck #1) from occurring before the event ULT1 (Unloading of Truck #1) to prevent overflow.

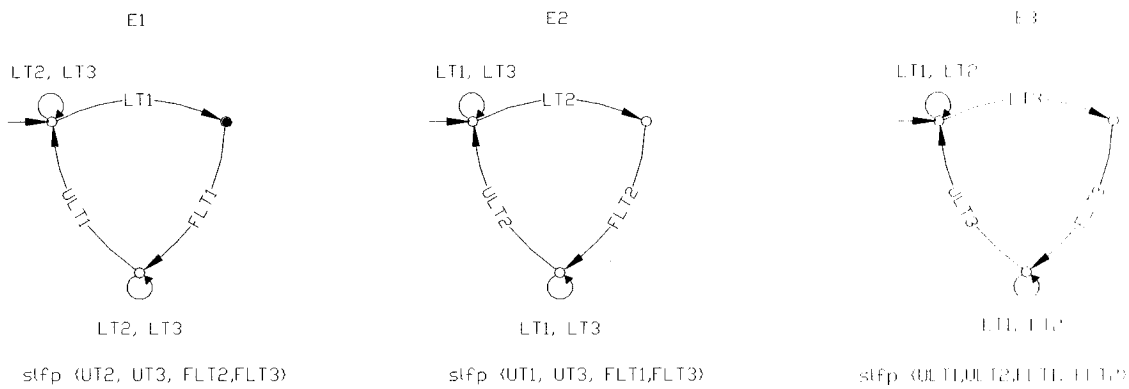


Figure 2.3: Ei represents the capacity, overflow and underflow specifications for truck #i, i = 1,2,3.

The last requirement specification deals with priorities and can be represented by a priority relation P ={(ULT3, ULT1), (ULT3, ULT2)}, which means that unloading of Truck #3 has a higher priority than unloading of Trucks #1 and #2, because Truck #3 is faster and hence its service is more valuable than the other two.

To obtain a supervisory control satisfying all specifications we first obtain SYS_P by applying priorities to the open loop system according to the constructive procedure described in Definition 2.3.1. The result is shown in Figure 2.4. Then a centralized supervisor SUPSYSPE is obtained by applying supcon to the pair (SYS, SYSPE).

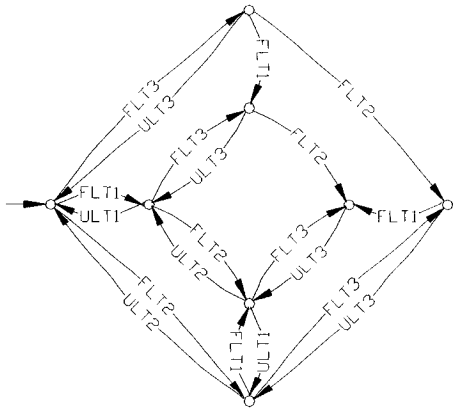
E12 = sync (E1, E2)

E = sync (E12, E3)

SYSPE = sync (SYS_P , E)

SUPSYSPE = supcon (SYS, SYSPE).

The supervisor SUPSYSPE is illustrated in Figure 2.5.



slfp ⟨LT1, LT2, LT3⟩

Figure 2.4: Automaton model for the SYSp.

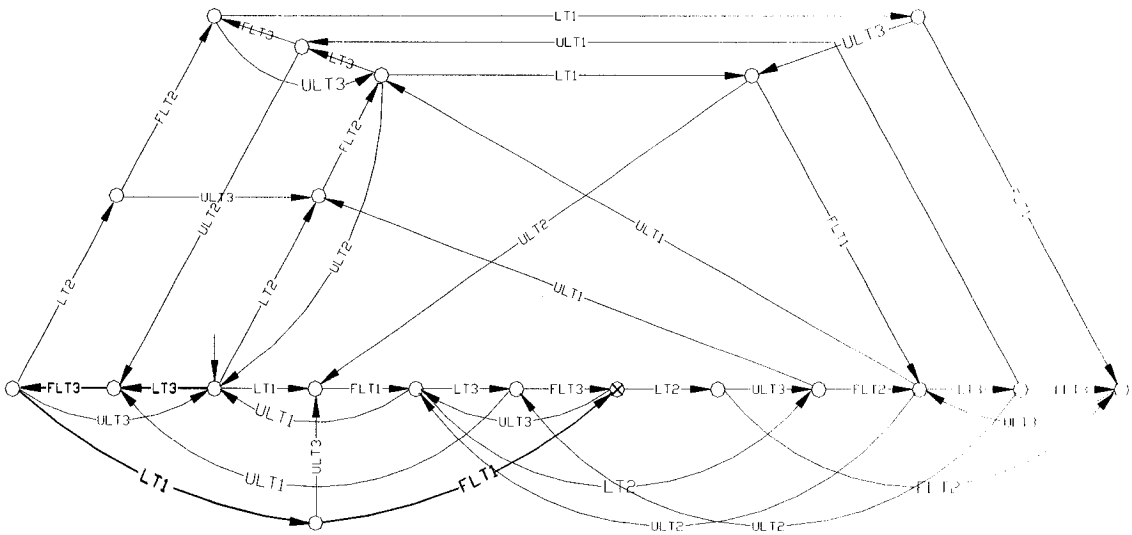


Figure 2.5: Automaton model for the supervised system.

One can verify that, for instance, at the state reached by the string 'LT3;FLT3;LT1;FLT1', the event UTL1 is disabled because it has a lower priority than UTL3. In other words, when both Trucks #1 and #3 are fully loaded, only Truck #3 is allowed to unload.

Another candidate that can be considered as a final specification for determining a priority-based supremal supervisory control such that plant under supervision satisfies the specification E while conforming with the priority relation P is $(\underline{L} \times \underline{E})_p$. $(\underline{L} \times \underline{E})_p$ is

obtained by applying priorities to the synchronous product of L and E as described in Definition 2.3.1. However, it can be shown that $(\underline{L}_p \times \underline{E})$ and $(\underline{L} \times \underline{E})_p$ are in fact isomorphic, and therefore it will be computationally more efficient to apply priorities to $\underline{L}$, and then take the synchronous product with $\underline{E}$.

Note that E_p alone may not be used as a specification to capture the priority relation P. For instance, let us consider the same system SYS, and let us assume that the only required specification is 'Truck #3 is allowed to be loaded only after Truck #1 has been loaded'. The automaton NE in Figure 2.6 translates the required specification. Given the same priority relation as before, to obtain NE_p, we apply priorities to NE as described in Definition 2.3.1. The result is shown in Figure 2.6.

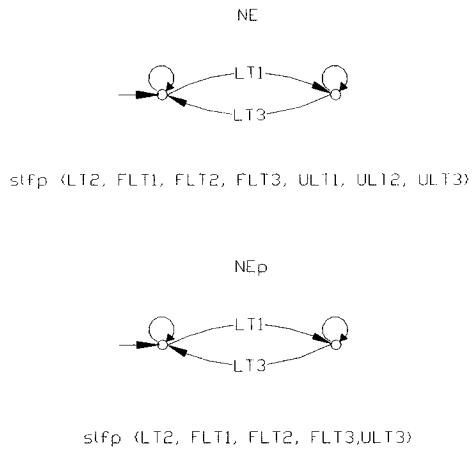


Figure 2.6 : The DES models of the NE and NEp.

Consider the string 'LT1;FLT1;ULT1'. It is admissible in the plant and its only specification, and it conforms with the priority relation P, yet it is not admissible in NE_p. We conclude that E_p is too restrictive to be used as a final specification.

Chapter 3

Discrete Timed Automata

3.1 Introduction

In this chapter, we augment the DES framework with a timing feature. Timing adds a new dimension to DES modeling and control, which is of considerable power and applied interest, but also of significant complexity [2]. In this way execution of a task can be modeled, which enables us to capture timing issues in a wide range of applications such as control of hard real-time systems.

In this chapter, we introduce discrete timed automata and languages, which are used in the rest of this work to model real-time tasks and their behaviors, respectively. In the process the notion of continuity in time is defined, and finally a question of whether the arbitrary union of timed languages is itself a timed language is addressed.

3.2 Discrete Timed Automata

We model a real-time task by a Discrete Timed Automaton (DTA), which is a 4-tuple of the form $G = (X, \Sigma_t, \delta, x_0)$ where:

- X is a finite set of states.
- Σ_t is the finite set of event labels associated with transitions in G, where

$$\Sigma_t := \Sigma \dot{\cup} \{t\}.$$

Here Σ is a finite alphabet of system events and *tick* represents one *tick* of the global clock.

$-\delta : X \times \Sigma_t \to X$ is a partial transition function. $\delta(x,\sigma) = y$ means that there is a transition labeled by event σ from state x to state y. For the sake of convenience, δ is extended from domain $X \times \Sigma_t$ to domain $X \times \Sigma_t^*$ in the following recursive manner:

$$\delta(x,\varepsilon) := x$$

$$\delta(x,s\sigma) := \delta(\delta(x,s),\sigma) \text{ iff } \delta(x,s)! \wedge \delta(\delta(x,s),\sigma)!.$$

$-x_0$ is the initial state.

Before defining the behavioral semantics of G in detail, we provide an informal summary. As is customary with DES, events are thought of as instantaneous and occur at quasi-random moments in real time $R^+ = \{t | 0 \le t \le \infty\}$. However, we assume that time is measured with a global digital clock with output *tickcount*: $R^+ \to N$, defined as:

$$tickcount(t) := n, \qquad n \le t < n+1.$$

Temporal conditions will always be specified in terms of this digital clock; real-valued time as such, and the clock function *tickcount*, will play no formal role in the model. The event *tick* occurs exactly at real time moments $t = n\,(n \in N)$. As usual, G is thought of as a generator of strings over Σ_t^*; intuitively G incorporates the digital clock, and thus its 'generating action' extends to the event *tick* [2]. In addition, we require a notion of *'continuity in time'*, in the sense that starting from any state of G, there exists at least a path in which the event *tick* has always a chance to occur (i.e. at each state, time always moves forward along at least one path).

Events are generated as follows. G starts from q_0 at $t = 0$ and executes state transitions in accordance with its transition function δ. The partial transition function δ is defined at a pair (x,σ), written $\delta(x,\sigma)!$, provided that:

$$\exists s \in \Sigma^* . \delta(x,\sigma s t)!.$$

which reflects the requirement of *'continuity in time'* in our models. It is worthwhile to mention that unlike [2], in our models we allow a *tick* transition to be preempted indefinitely by system events as long as the *'continuity in time'* is respected. In other words we require that a *tick* occur infinitely often along at *least* one path, as opposed to [2] where *tick* is required to happen infinitely often along *every* path.

With the definition of G complete, we now describe the behavior of G as it evolves over time. At the initial state x_0 and upon the occurrence of an event $\sigma \in \Gamma(x_0)$[3] $\subseteq \Sigma_t$, G will make a transition to state $\delta(x_0,\sigma) \in X$. This process then continues in accordance with the transitions for which δ is defined.
The language generated by G is defined as:

$$L_t(G) = \left\{ s \in \Sigma_t^* \mid \delta(x_0,s)! \right\}.$$

Since a path can be generated only if all its prefixes can be generated, it follows from the definition that $L_t(G)$ is prefix-closed.

As our interest is in analyzing the behavior of a DTA through its language, it is important to identify the class of languages generated by DTA, which we call *discrete timed languages*. Let us assume that L is a language over Σ_t, i.e. an element of the

[3] $\Gamma : X \to 2^{\Sigma_t}$ is *the active event set* function; $\Gamma(x)$ is the set of all events σ for which $\delta(x,\sigma)$ is defined and is called *the active event set* of G at x.

power set $2^{\Sigma_t^*}$ (thus, the definition includes both the empty language $\varnothing$ and Σ_t^*). We say that L is a *discrete timed language* if it satisfies the following properties:

1. L is prefix-closed, i.e. $L = \overline{L}$.
2. $\forall s \in L. \exists s' \in \Sigma^*. ss't \in L$.

Thus, in a timed language as a string evolves over the alphabet of events, time always moves forward along at least one path.

An important question about timed languages is whether the arbitrary union of timed languages is itself a timed language. This property is essential when we want to analyze a system whose behavior is not a timed language. Then such a property guarantees that a *largest* timed sublanguage of the behavior exists.

In order to analyze such a system, let us assume that M is a language over Σ_t, which is not necessarily a timed language. We define the class of timed sublanguages of M as:

$$\mathfrak{T} = \{T \subseteq M \mid T \text{ is a timed language}\}.$$

Lemma 3.2.1

$\mathfrak{T}$ is nonempty and $\mathfrak{T}\uparrow = \bigcup_{T \in \mathfrak{T}} T \in \mathfrak{T}$, where $\mathfrak{T}\uparrow$ is the supremal timed sublanguage of M. In other words, the supremum of $\mathfrak{T}$ with respect to the partial ordering $\subseteq$ exists and belongs to $\mathfrak{T}$.

Proof:

Denote the supremum of $\mathfrak{T}$ by $\mathfrak{T}\uparrow$. First we note that $\varnothing \in \mathfrak{T}$, and therefore $\mathfrak{T}$ is nonempty. It follows that $\mathfrak{T}\uparrow = \bigcup_{T \in \mathfrak{T}} T$.

In order to show $\tau\uparrow \in \tau$, we must show that:

1. $\tau\uparrow \subseteq M$.
2. $\overline{\tau\uparrow} = \tau\uparrow$.
3. $\forall s \in \tau\uparrow . \exists s' \in \Sigma^*.\ ss't \in \tau\uparrow$.

1. We show that $\tau\uparrow \subseteq M$.

Since M is an upper bound of τ and $\tau\uparrow$ is the least upper bound of τ, it follows that $\tau\uparrow \subseteq M$.

2. We show that $\overline{\tau\uparrow} = \tau\uparrow$.

Since $\tau\uparrow$ is the union of timed sublanguages, it follows that $\tau\uparrow$ is the union of prefix-closed sublanguages. In addition, we know that the union of prefix-closed languages is itself a prefix-closed language. Therefore $\tau\uparrow$ is a prefix-closed language.

3. We show that $\forall s \in \tau\uparrow . \exists s' \in \Sigma_t^*. ss't \in \tau\uparrow$.

Let $s \in \tau\uparrow = \bigcup_{T \in \tau} T$. It follows that $s \in T$ for some $T \in \tau$. T is a timed language, which implies that:

$$\forall s \in T . \exists s' \in \Sigma^*.\ ss't \in T. \quad \text{(i)}$$

In addition, since $T \subseteq \tau\uparrow$ it follows from (i) that $ss't \in \tau\uparrow$. Thus, we have shown that:

$$\forall s \in \tau\uparrow . \exists s' \in \Sigma^*. ss't \in \tau\uparrow .$$

From (1), (2) and (3) we conclude that $\tau\uparrow \in \tau$, and therefore $\tau\uparrow$ is the supremal timed sublanguage of M. $\therefore$

To illustrate this property we consider an example.

Example 3.1

Let us consider a real-time system described by the automaton OLS_1, shown in Figure 3.1.

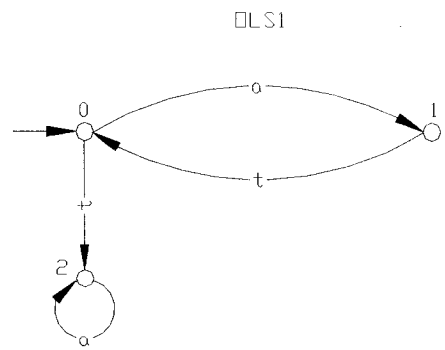


Figure 3.1: A real-time system OLS1.

It can be seen that at state #2 of OLS_1, there is no path in which the event *tick* has a chance to occur. Thus, according to the definition of 'continuity in time' the language generated by OLS_1 is not a timed language. By removing those states of OLS_1 from which time cannot progress, we will obtain a DTA that generates the supremal timed sublanguage of the language generated by OLS_1. The resulting DTA is shown in Figure 3.2.

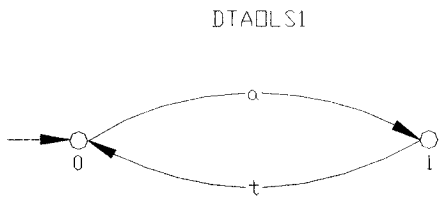


Figure 3.2: A DTA representing the supremal timed sublanguage of $\mathrm{IL}(A)$.

Note that although it is technically impossible to disable tick at state 0, we assume that the event 'a' is forced to preempt tick, so 'a' is required to be a forcible event [6]. We make this assumption implicitly throughout the rest of the thesis: *tick* can be disabled whenever there are other (forcible) events present in the system.

It is worthwhile to note that the class of timed sublanguages of a given language forms a 'complete lattice' that has a largest element. To illustrate this property we consider the next example.

Example 3.2

Let us consider a real-time system described by the automaton OLS_2 shown in Figure 3.3. It can be seen that at state #2 of OLS_2, there is no path in which the event *tick* has a chance to occur. Thus, according to the definition of 'continuity in time' the language generated by OLS_2 is not a timed language.

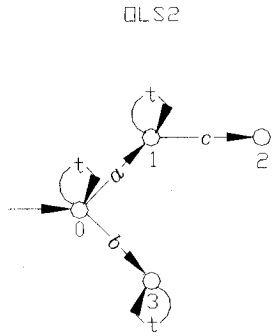


Figure 3.3: A real-time system OLS2.

The language generated by OLS_2 has five timed sublanguages. It can be verified that these timed sublanguages are partially ordered with respect to set inclusion. The Hasse diagram of timed sublanguages of $OLS_2$is shown in Figure 3.4.

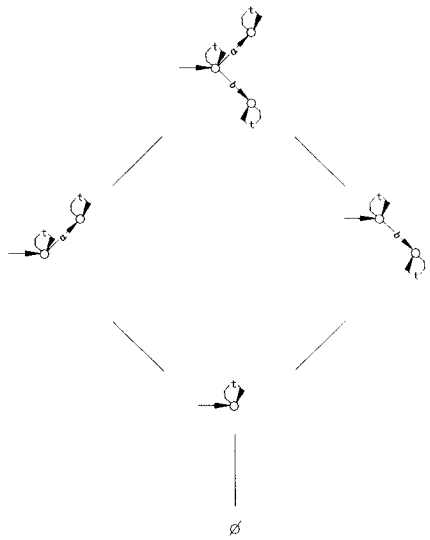


Figure3.4: The Hasse Diagram of the Timed Sublanguages of OLS2.

It can be verified that the partial ordering represented by the Hasse diagram of Figure 3.4 is in fact a complete lattice. Every pair of the timed sublanguages of the language generated by OLS_2 has both a least upper bound and a greatest lower bound in

the lattice. It follows that the language generated by OLS_2 has a supremal timed sublanguage. The supremal timed sublanguage of the language generated by OLS_2 is the supremal element of the lattice, and is shown in Figure 3.5 for convenience.

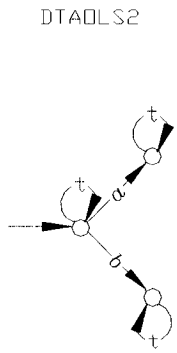


Figure 3.5: The DTA Model of the Supremal Timed Sublanguage of $L(S_2)$

Chapter 4

Tasks Scheduling in Supervisory Control

4.1 Introduction

Task scheduling is used in a wide range of applications. For instance, real-time scheduling has been a fundamental issue in the development of hard real-time systems. Most research in this area has been conducted mainly in the context of computer operating systems. Schedulers coordinate the execution of a set of tasks, so that all desired deadlines are met.

Real-time environment requires a very disciplined approach based on worst case assumptions and mathematical techniques. In a real-time control system this means establishing well-defined limits for managing computing resources and using advanced modeling techniques to guarantee their performance.

However, task scheduling has potentially far wider applications in other areas such as manufacturing systems [6]. Just-in-time idea has being used in manufacturing processes. The basic idea in Just in time (JIT) is to reduce wasting of material. Requesting the right amount of raw material and producing the products as much as it is required achieve this goal. It is worthwhile to mention that in JIT material right processing time and right processing place are two separate concerns. Considering the principle of JIT it seems that applying task scheduling algorithms could be desirable.

Well-established theory of real-time scheduling has been developed, that should provide a direction to the concepts such as validation of real-time system [7]. Yet, the literature traditionally introduces informal methods in which proofs are not precise. It is worthwhile to mention that informal results do not provide necessary tools to generalize and improve the existing results. There are potential errors in existing models of real-time

scheduling. Very precise approaches are necessary to capture these errors. In addition, the correctness of scheduling algorithms must be proven before applying to safety-critical real-time applications and it should be compatible with different complex systems.

We propose a general formal algorithm that introduces a model for real-time schedulers such that all processes satisfy their critical requirements. It also covers the scheduling of different types of tasks. We examine how formal methods can help address these issues. Formal methods can be used to develop very precise, complete, and mathematically sound proofs. Formal modeling and verification insures that an RTOS satisfies critical scheduling and synchronization properties.

This chapter is organized as follows. Section 2 introduces preemptive/non-preemptive as well as priority/non-priority-based scheduling. In addition, dynamic and static (off-line) scheduling paradigms are introduced and their strengths and weaknesses are discussed. Section 3 proposes how to model a task as a DTA. Supervisory control of tasks is discussed in Section 4 and some examples are presented to show how supervisory control of DTA can be used as a unified approach to address different types of scheduling problems.

4.2 Scheduling Paradigms

Depending on time instances when requests for execution are made, tasks can be classified as *periodic, sporadic,* and *aperiodic*. A *periodic* task repeats at regular time intervals and its request times are known *a priori*. A *sporadic* task request times are not known *a priori*, but it is assumed that a minimum interval exists between two successive requests. Finally, an *aperiodic* task has no such constraint on its request times.

Tasks can be independent or have a precedence relation defined among them. Tasks can be scheduled in preemptive/non-preemptive as well as priority/non-priority-based fashions.

- **Non-preemptive Schedule:** In a *non-preemptive* schedule, the execution of a task cannot be interrupted once it starts. The task must be executed to completion, and

then may another task be initiated. Non-preemptive schedules incur fewer overheads due to the lack of context switching, and are generally easier to implement compared to preemptive ones.

- **Preemptive Schedule:** In a *preemptive* scheduling algorithm, it is possible to interrupt a task during its execution. This type of scheduling algorithms is usually driven by the notion of prioritized computation, where a task with the highest priority should always be the one that is presently under execution. If a task is currently running and a new task with a higher priority arrives, the task with the lower priority should be suspended and it should wait until it once again becomes the highest-priority task in the system.

- **Priority-Based Schedule:** A *priority-based* schedule executes the high-priority tasks first, and allows the controller to give preference to 'important ' tasks. In priority-based scheduling, the order of task execution is determined by their priorities as specified by a priority relation. The key to timely performance of a priority-based schedule is in choosing a sound priority relation among tasks.

- **Non-Priority-Based Schedule:** In contrast, in a *non-priority-based* schedule, no priority relation is defined over the set of tasks and therefore the order of task execution is determined by their arrivals only.

With preemptive scheduling, the running task can be interrupted at any time by the controller to perform another task, whereas with non-preemptive scheduling, once started, a task has to run to completion before it relinquishes the processor.

It is worthwhile to note that in a non-preemptive/priority-based schedule, execution of a task cannot be preempted no matter how low its priority is. A task either runs to completion, or else it has to make way for another higher priority task *before* it starts execution.

A schedule in which all tasks meet their time constraints is called *feasible*. A *dynamic* scheduler makes its scheduling decisions at run time based on requests for

system services. After the occurrence of a significant event such as a service request, the scheduler determines which task in the set of 'ready' tasks should be executed next based on some task priorities, which is statically or dynamically assigned.

A well-known static priority-based algorithm for scheduling independent and periodic tasks is due to Liu and Layland [8] and is called *Rate Monotonic (RM)*. The rate monotonic algorithm sets task priorities according to their periodicity: The task with the shortest period is assigned the highest priority and is thus executed first. Next is the task having the second shortest period; followed by the one with the third shortest period, and so on. Liu and Layland offer a simple schedulability test based on resource utilization by the tasks [8].

Liu and Layland also studied a preemptive algorithm called *Earliest Deadline First (EDF)* that dynamically assigns task priorities based on their deadlines. The closer is a task's deadline, the higher its priority is [8]. When tasks are released, the one whose deadline is coming first is assigned the highest priority and will be executed first. After each time unit, the operating system selects, among all ready tasks, the one whose deadline is earliest for executing next [9].

Dynamic scheduling algorithms such as EDF are flexible and can be extended to handle aperiodic and sporadic task requests [10]. Static scheduling is suited for periodic, embedded control systems such as automotive control because of its predictability of behavior and high resource utilization. A common way to increase the flexibility of static scheduling is called *Mode-change* execution [11]. All possible operating and emergency modes are identified during system design and a static schedule is calculated for each mode. When a mode change is requested at run time, the appropriate schedule is activated.

4.3 DTA Task Mode

In this section, we illustrate how a DTA can be used to model non-preemptive as well as preemptive tasks in general, and periodic ones in particular. We evaluate the DTA task model through a practical example concerning a sampled-data system.

4.3.1 Preemptive Task Model

In this section, we model the execution of a preemptive task by a DTA. In the following development, as mentioned before, time is assumed to be discrete and is incremented by *ticks* of a global clock, which is indexed by natural numbers. The instant at which a task has first arrived is referred to as the *release time* of a task. Once released, the first instance of the task can be executed at any discrete time. Similarly, other instances of the task can be executed at any discrete time after their arrivals.

An instance of a task is divided into *segments.* Each segment has duration equal to one *tick.* Breaking an instance of a task into segments enables the designer to include in his models the possibility of interrupting a task.

A task is arrived when its existence is made known to the system with an event called *arrival of the task.* The event 'arrival of the task' can occur either at initial state (for the first instance) or after the completion of the task (for other instances). A task is completed when all its segments have been successfully executed by the system. It is assumed that the task execution time c_t is an integer, implying that the total number of segments for such a task is equal to c_t.

We introduce a reasonable assumption that each *segment* is non-preemptive, which means that once the execution of a segment has started, it cannot be interrupted until its completion. This is consistent with our assumption about *tick* as the unbreakable unit of time. Whenever preemption is allowed, it is possible to interrupt a task *between* its different segments, but not during execution of a segment.

With the definition of a task now complete, we present in Figure 4.1 a model for a preemptive task generically named PRE.TJ with an execution time of c_j. All events of task PRE.TJ are summarized in Table 4.1.

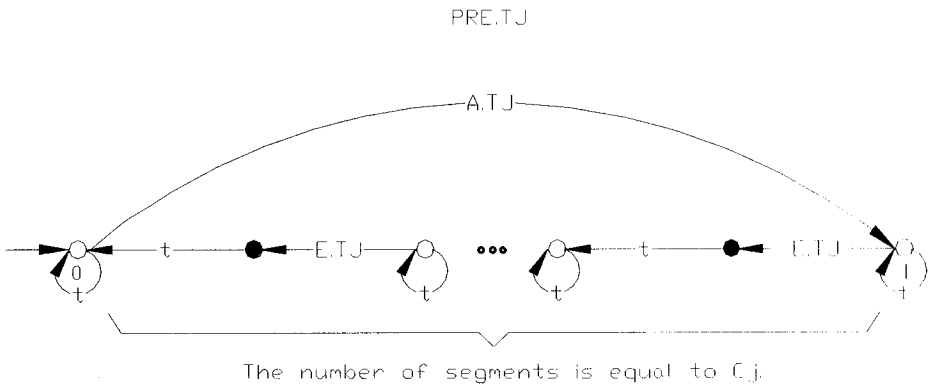


Figure 4.1: The DTA model of the preemptive task PRE.TJ.

Event	Description
A.TJ	Arrival of the task PRE.TJ
E.TJ	Commencing execution of one segment of the task PRE.TJ
t	*Tick* of the global clock

Table 4.1: Event descriptions for the task PRE.TJ.

By stepping through the transition graph of Figure 4.1 it can be verified that since the initial state is self-looped by the *tick* event, the task PRE.TJ may arrive at any time. The task PRE.TJ makes itself known to the system with the occurrence of event A.TJ which takes the model to state #1. Since state #1 is self-looped by the *tick* event, it follows that the task PRE.TJ may be executed any time after its arrival. It is important to note that at all filled states, where a segment of the task is under execution, the *tick* event is the only event that can occur, meaning that each segment is non-preemptive and takes one unit of time to complete. However, all other states are self-looped with the *tick* event to allow for preemption. It is worthwhile to mention that the self-loops of 'irrelevant'

events, i.e. those events belonging to the alphabet of another task, can be added at states #0 and #1 of PRE.TJ to reflect that it is admissible for other tasks to occur before starting or after arrival of the task PRE.TJ. These self-loops are added at all unfilled states between states #0 and #1 as well, to indicate that PRE.TJ is assumed to be preemptive.

In fact, this is an essential property that gives a designer the ability to model preemptive tasks by allowing execution of a task to be interrupted while the system attends to other tasks.

4.3.2 Non-preemptive Task Model

In this section, we model the execution of a non-preemptive task using a DTA. As noted earlier the execution of a non-preemptive task cannot be interrupted once started; the execution of an instance of the task must be completed before another execution can be initiated.

To present a DTA model for a non-preemptive task, we modify the DTA of PRE.TJ by removing the *tick* self-loop from all states in between the first and the last segments, if there is any. Let us denote the modified model by generic name NPRE.TJ, illustrated in Figure 4.2.

All events of the task NPRE.TJ are the same as in PRE.TJ and are summarized in Table 4.1. Note that as illustrated in Figure 4.2, we keep self-loops at state #0 and state #1 since the preemption of a task is not an issue before starting or after its arrival.

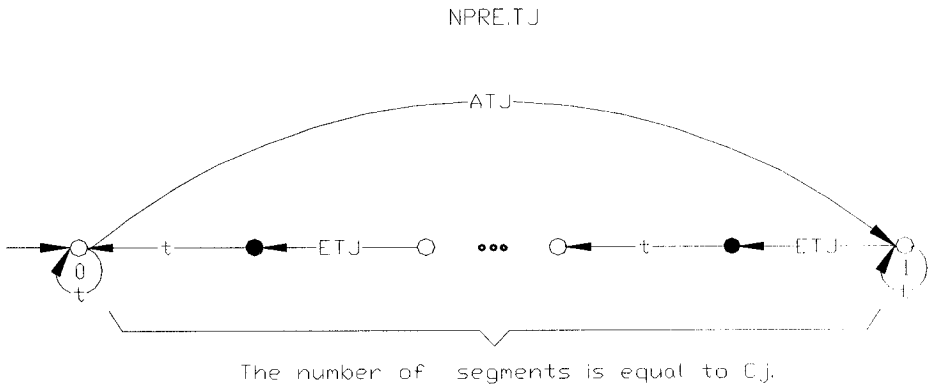


Figure 4.2: The DTA model of the non-preemptive task NPRE.TJ

4.3.3 Periodic Task Model

We continue this section with a formal definition of a periodic task. A *periodic* task is one that arrives repetitively at equidistance moments in time. The time interval between two consecutive arrivals is referred to as *task period.* For a given task *J*, let $c_j \in \mathrm{N}$ be the task execution time, $T_j \in \mathrm{N}$ be the task period, and t_k be the time of the k^{th} arrival of the task. A task is *periodic* if $t_{k+1} = t_k + T_j$. Since the execution of a task may be initiated only after the task has arrived, it is normally required that $c_j \leq T_j$.

A periodic task could be either non-preemptive or preemptive. We model periodicity of a task by a *specification* DTA. We use the automaton SPECJ illustrated in Figure 4.3 to specify that task J arrives once every T_j *ticks.* We assume that the first instance of the task arrives at time 0, i.e. $t_1 = 0$.

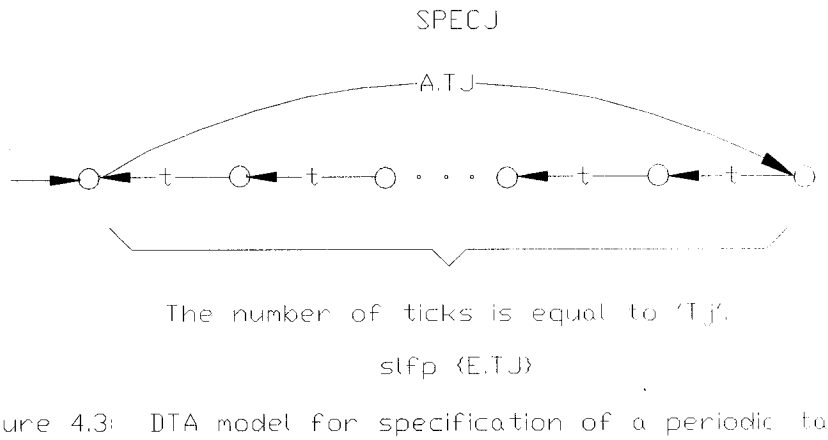


Figure 4.3: DTA model for specification of a periodic task J.

The specification of Figure 4.3 imposes the restriction that the event 'arrival of the task' can occur once every T_j *ticks*, which implies that task J is periodic with period T_j.

The synchronous product of the DTA models of task J and specification provides a model for a periodic task with execution time and period equal to c_j and T_j, respectively.

Example 4.3.1

A computer-controlled system is a real-time application where control law must be calculated within a specified deadline. In a sampled-data system, deadline may simply be defined to be equal to the sampling period, as the control calculation is required to be complete before the next sampling instant at the latest [9].

In this example, we illustrate how the DTA model of task execution can be used to obtain different safe task executions with a guarantee that all task deadlines are met.

Let us consider a dosing unit used in a chemical batch plant to supply a defined amount of liquid to a subsequent processing unit [12]. A dosing unit consists of a tank, an inlet valve A, an outlet valve B, and two sensors sending data indicating the level of the fluid in the tank to a controller unit which controls the operation of both valves. Let us assume that the inlet valve A and the outlet valve B require sampling intervals of 2 and 4

msec, respectively, while the controller unit needs 1 and 2 *msec*, respectively, to generate control signals for each update of valves A and B.

A simple network between valves and controller is illustrated in Figure 4.4.

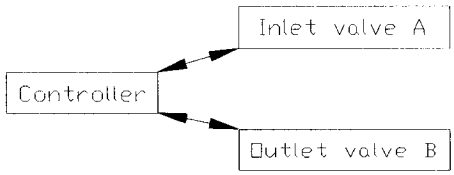


Figure 4.4: The Valves and the controller network.

Computation times required for generating control signals by the controller, and the required sampling periods for the valves are summarized in Table 4.2.

Required computation times		Required sampling periods	
Valve A	Valve B	Valve A	Valve B
1 *msec*	2 *msec*	2 *msec*	4 *msec*

Table 4.2: Computation times and sampling intervals required by the controller and the valves A and B.

As mentioned earlier the controller must perform two tasks. These tasks are responsible for generating signals needed to control the operation of valves A and B. Let us denote these tasks by PRE.TA and PRE.TB, respectively, as shown in Figure 4.5.

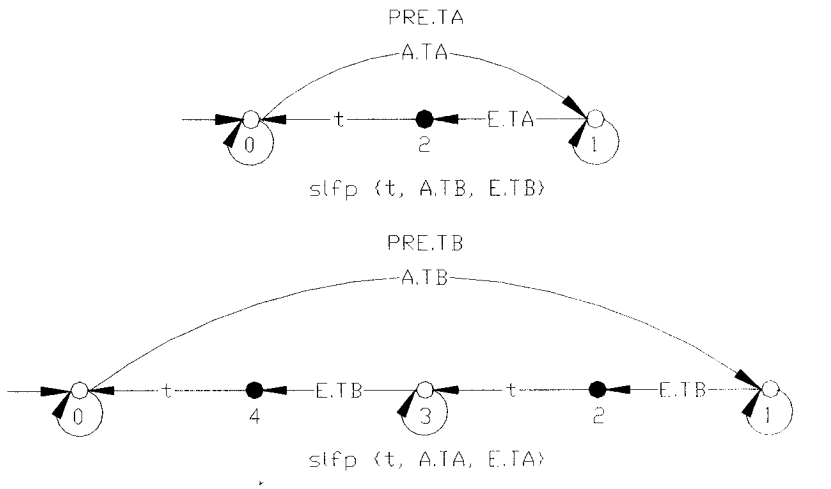


Figure 4.5:The DTA models of the controller preemptive tasks.

The events for each task are summarized in Table 4.3.

Event	Description	Event	Description
A.TA	Arrival of the task PRE.TA	E.TB	Commencing execution of one segment of the task PRE.TB
A.TB	Arrival of the task PRE.TB	T	*Tick* of the global clock
E.TA	Commencing execution of one segment of the task PRE.TA		

Table 4.3: Event descriptions for the controller.

As discussed in Section 4.3.1, we add self-loops at states #0 and #1 of PRE.TA and PRE.TB to reflect that it is admissible for each task to occur before starting or after completion of the other. We add self-loop at state #3 of PRE.TB to illustrate that PRE.TB, i.e. the control-generating task for valve B, is assumed to be preemptible.

It is important to note that in PRE.TA the *tick* transition could not be preempted indefinitely by *irrelevant* events (i.e. those events from PRE.TB that do not belong to the

alphabet of PRE.TA). Such a behavior is simply disallowed by the transition structure of PRE.TB. A similar observation can be made about PRE.TB.

For instance, at the initial state of PRE.TA, the event E.TB cannot indefinitely occur because the transition structure of PRE.TB dictates that a *tick* event must occur between subsequent occurrences of E.TB. This can be verified by stepping through the transition graph of PRE.TB. We introduce two specifications named SPEC.TA and SPEC.TB to guarantee that events A.TA and A.TB occur once every 2 and 4 *ticks*, enforcing sampling periods of 2 and 4 *msec* for valve A and B, respectively. The task specifications, SPEC.TA and SPEC.TB are illustrated in Figure 4.6.

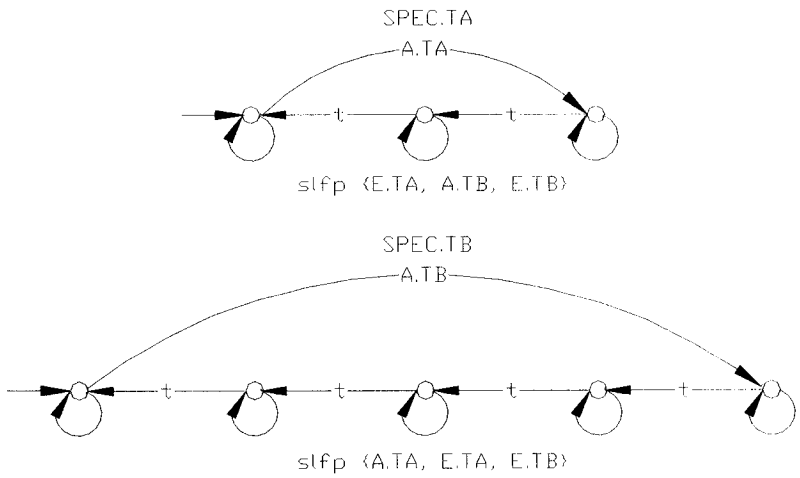


Figure 4.6: The DTA models of the controller task specifications.

In order to model the combined execution of control-generating tasks, we take the synchronous product of PRE.TA and PRE.TB to obtain a single DTA named PRE.TAB.

PRE.TAB = sync (PRE.TA, PRE.TB).

We also combine the two specifications SPEC.TA and SPEC.TB into a single DTA by taking their synchronous product in order to enforce the periodic execution of tasks PRE.TA and PRE.TB by a single specification.

SPEC.TAB = sync (SPEC.TA, SPEC.TB).

The supremal supervisory control, which contains all allowable sequences of task executions, can then be obtained by taking the supcon of plant and specification.

PRE.NPRI.SUP = supcon (PRE.TAB, SPEC.TAB).

The supremal supervisory control PRE.NPRI.SUP is illustrated in Figure 4.7.

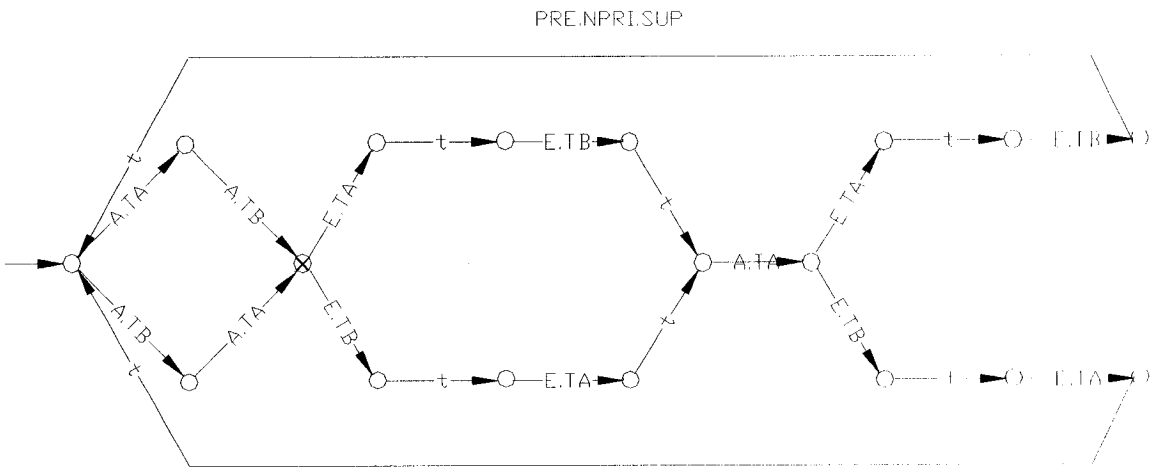


Figure 4.7: The preemptive/non-priority-based supremal supervisor for controlling the operations of valves A and B.

It is worthwhile to note that for the sake of convenience all events are assumed to be controllable. This assumption does not affect the generality of our proposed approach, since supcon procedure results in a supremal controllable sublanguage in case of some of the events are uncontrollable. In fact, we would like to separate our concerns about controllability and schedulability.

By stepping through the transition graph of PRE.NPRI.SUP it can be verified that:

- There is one occurrence of E.TA between two successive occurrences of A.TA.

- There are two occurrences of E.TB between two successive occurrences of A.TB.
- A.TA occurs once every two *ticks*.
- A.TB occurs once every four *ticks*.

These are in fact the timing requirements specified for tasks earlier. The supervisor PRE.NPRI.SUP provides all possible schedules guaranteed to meet the desired specifications.

As illustrated in Figure 4.8 there are four possible 'main' offline schedules, i.e. those schedules of minimal length that are repetitive from cycle to cycle. The supervisor for controlling both valves may select each of these schedules. They guarantee that the operation of valves A and B is in the right order as specified by SPEC.TA and SPEC.TB.

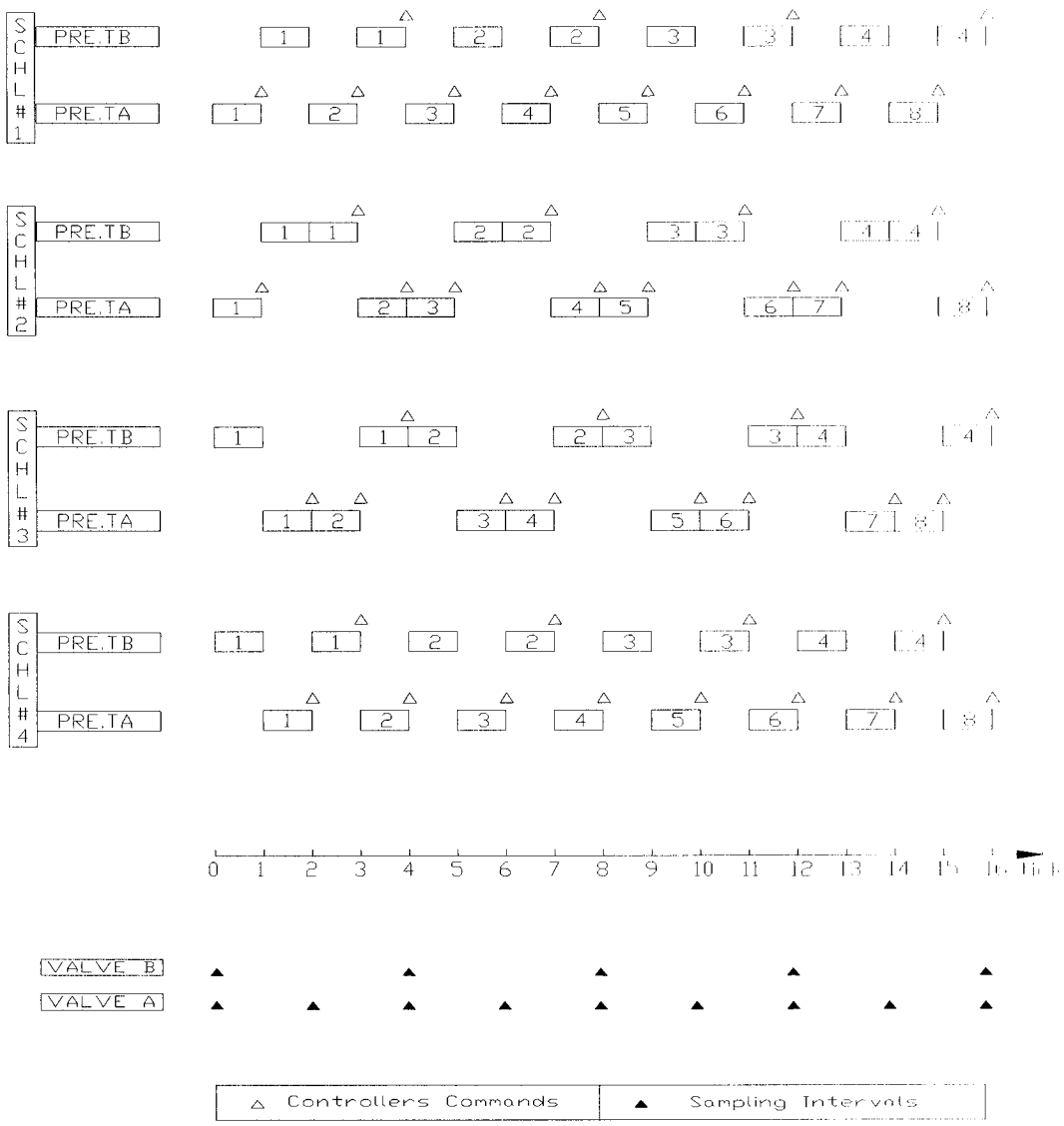


Figure 4.8: The preemptive/non-priority-based task schedules for controlling the operations of valves A and B.

4.4 Task Scheduling in Supervisory Control

The problem of scheduling a set of tasks is usually solved by two approaches. In the first approach, the problem is considered as two sub-problems:

i. Determining whether a set of tasks is schedulable by using *utilization* technique, discussed in Appendix B.

ii. Finding an algorithm to generate a sequence by which the tasks are to be executed.

The other approach works in the reverse order; namely, a candidate schedule is first generated, which is then checked to see whether it meets all the deadlines.

Both approaches involve solving two separate problems. In this section, we present a formal approach, which is based on supervisory control of discrete timed automata.

To investigate the problem of task scheduling, as illustrated in Example 4.3.1, we first model the execution of tasks using DTA. We then obtain the supremal controllable sublanguage of the desired specification with respect to task DTA. To determine whether a set of tasks is schedulable, we simply check whether the supremal controllable sublanguage is *nonempty*. If so, it follows that it contains all *safe* sequences of task executions, which guarantees that all deadlines are met.

Our approach differs from the conventional scheduling methods in the following ways:

1. A nonempty supremal controllable sublanguage corresponding to DTA models of plant and desired specifications, if it exists, contains task execution sequences that meet all the deadlines. It follows that we do not need to treat checking for schedulability and finding a scheduling algorithm as two separate problems. Neither do we need to verify the timing characteristics of a schedule once it is computed, as is customary in conventional scheduling approaches.

2. A nonempty supremal controllable sublanguage provides *all* possible task execution sequences, while guaranteeing that all desired specifications are met. The issue of completeness does not appear to have been addressed in the conventional scheduling approaches, resulting in a scheduling algorithm that produces *only one* sequence of task executions for a given set of tasks.

3. It is a formal approach that can be used to develop concise, complete, and rigorous scheduling methods, which are easily extensible to more general problems.

The other advantage of our approach is that different types of schedules, including priority-based, non-priority-based, preemptive and non-preemptive scheduling schemes can all be modeled in DTA framework simply by changing the desired specifications.

Many scheduling strategies have been devised and put into service [13]. Figure 4.9 shows a general categorization among the various scheduling policies.

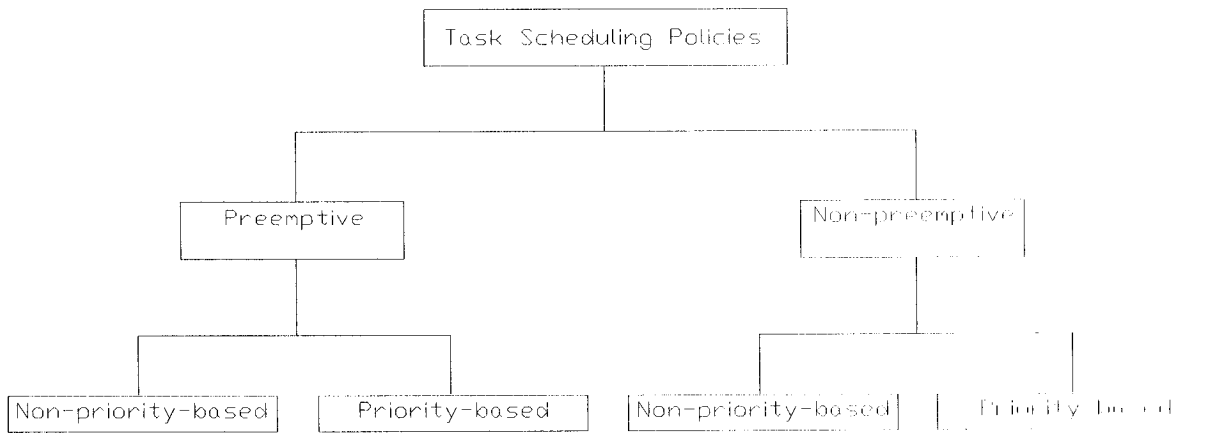


Figure 4.9: Scheduling policies.

To obtain scheduling of a set of tasks in supervisory control, the first step is to model each task as a DTA. It is noticeable that every state of a preemptive task model must be self-looped by irrelevant events which occur in other tasks. These DTA are then combined (e.g. by using the synchronous product procedure in 'TTCT') into a *composite task execution* model.

The desired requirements include the following:

1. Whether tasks are preemptive.
2. Whether a priority relation among tasks is given.
3. Timing constraints.

These specifications are expressed by DTA and combined (using synchronous product) into a *composite specification.* We finally obtain the supremal controllable sublanguage by applying the supcon procedure to the composite task and specification models.

As mentioned before a nonempty outcome of the supcon procedure generates all task execution sequences that satisfy all desired specifications. The overall procedure is illustrated in Figure 4.10.

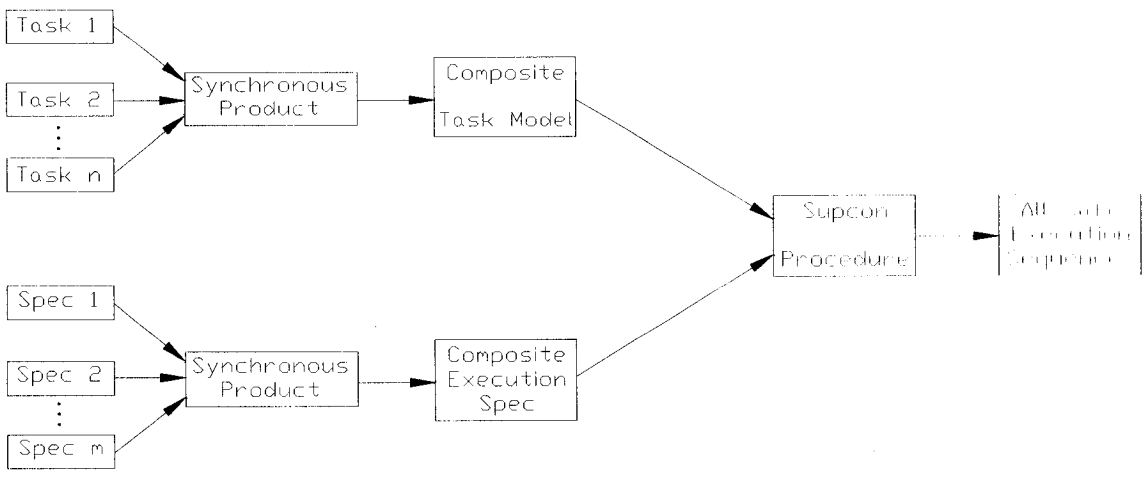


Figure 4.10: DTA tasks scheduling procedure.

In the next section, we show how supervisory control can be employed to generate different scheduling policies.

4.4.1 Preemptive/Non-priority-based Scheduling

As discussed earlier in preemptive scheduling algorithms it is possible to interrupt a task during its execution. In non-priority-based scheduling, no priority relation is defined over the set of tasks to restrict the order of task executions.

The preemptive/non-priority-based scheduling satisfies both of the above properties. The supervisor of example 4.3.1 is in fact a preemptive/non-priority-based scheduler. This scheduling policy imposes the least restriction on the system behavior. In other words, such a scheduling policy is a *universal schedule*. The universal schedule generates all possible execution sequences that meet all the deadlines.

4.4.2 Preemptive/Priority-based Scheduling

As discussed earlier, in preemptive scheduling algorithms it is possible to interrupt a task during its execution. Preemptive algorithms are usually driven by the notion of prioritized computation. If a task is currently executing and a new task with a higher priority arrives, the task with the lower priority is suspended until it is once again the highest priority task in the system. In a priority-based algorithm, the order of task executions is determined by their priorities as specified by a priority relation.

To see an example, let us consider the dosing unit described in Example 4.3.1. Let us assume that execution of task PRE.TA has a higher priority than the corresponding event in task PRE.TB. To address this restriction we define a priority relation P as:

$$P=\{(\text{E.TA}, \text{E.TB})\}.$$

To obtain a schedule that conforms with the priority relation P we first obtain PRE.TAB_P by applying priorities to the open loop system according to the constructive

procedure described in Definition 2.3.1. The final specification will be obtained after taking the synchronous product of SPEC and $PRE.TAB_P$.

A centralized supervisor PRE.PRI.SUP is then obtained in 'TTCT' by taking the supcon of plant PRE.TAB and the final specification PRE.PRI.SPEC.

$$PRE.PRI.SPEC = \underline{sync}\ (PRE.TAB_P, SPEC.TAB)$$
$$PRE.PRI.SUP = \underline{supcon}\ (PRE.TAB, PRE.PRI.SPEC).$$

The supremal supervisory control PRE.PRI.SUP is illustrated in Figure 4.11.

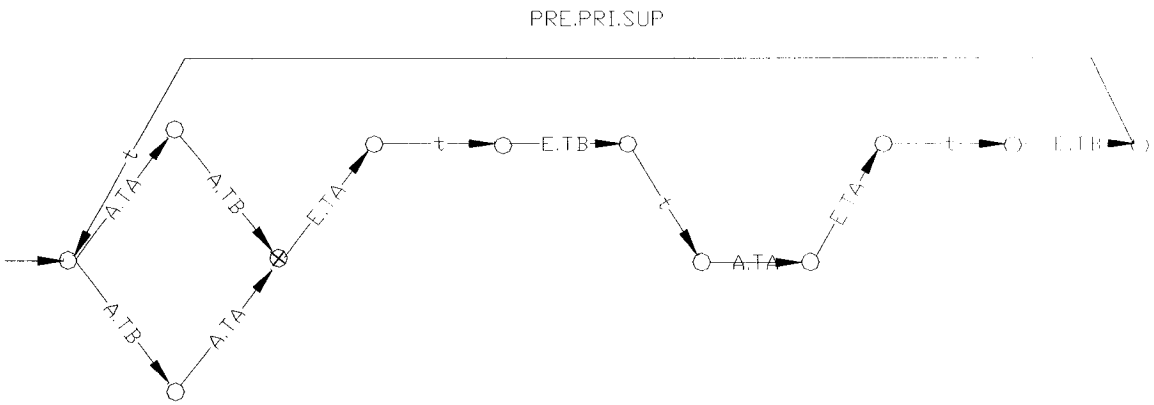


Figure 4.11: The preemptive/priority-based supremal supervisor for controlling the operations of valves A and B.

By stepping through the transition graph of PRE.PRI.SUP it can be verified that the major cyclic sequence generated by PRE.PRI.SUP is:

$$[(A.TA;A.TB + A.TB;A.TA);E.TA;t;E.TB;t;A.TA;E.TA;t;E.TB;t]^*.$$

This sequence is one of the safe sequences generated by the automaton of Figure 4.7, Page 44, which contains all safe execution sequences for tasks PRE.TA and PRE.TB. As illustrated in Figure 4.7, PRE.NPRI.SUP allows both E.TA and E.TB to occur at the

state marked by a cross sign. Since (E.TA, E.TB)$\in P$ and E.TA is defined in the plant, it follows that supervisor PRE.PRI.SUP should not allow E.TB to occur at the same state. We can see that the supervisor PRE.PRI.SUP indeed offers only one choice at the state marked by a cross sign in Figure 4.11, namely, E.TA.

Figure 4.12 illustrates the only possible preemptive/priority-based schedule corresponding to the priority relation P that can be selected by the supervisor for controlling both valves A and B. This schedule guarantees that the operation of valves A and B is in the right order as specified by SPEC.TA and SPEC.TB.

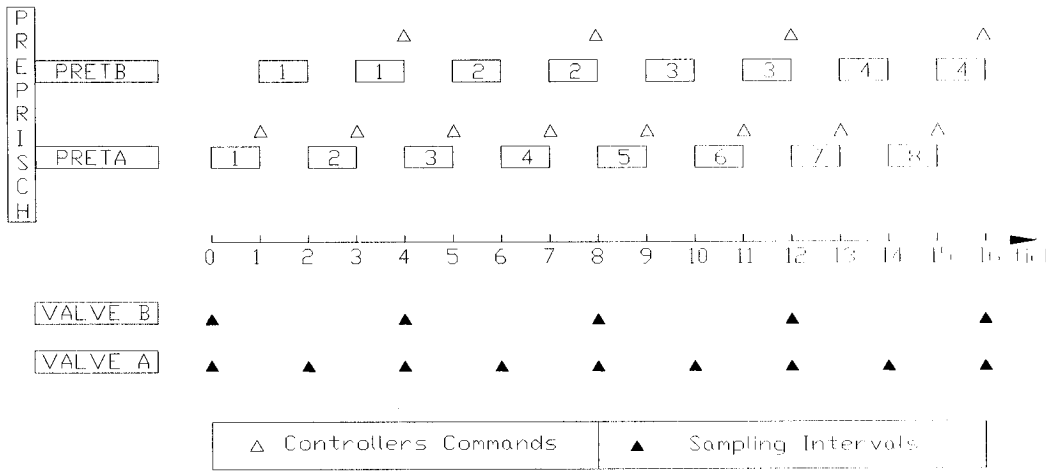


Figure 4.12: The preemptive/priority-based schedule for controlling the operations of valves A and B.

It can be verified that the schedule of Figure 4.12 is in fact rate monotonic.

4.4.3 Non-preemptive/Non-priority-based Scheduling

As discussed earlier in a non-preemptive schedule, the execution of a task cannot be interrupted once it is started. In a non-priority-based schedule, no priority relation is defined over the set of tasks to restrict the order of task executions.

To see how such a schedule can be generated using DTA supervisory control, let us consider the dosing unit described in Example 4.3.1, where tasks PRE.TA and PRE.TB were originally assumed to be preemptible. Now we impose an additional restriction on the behavior of the dosing unit described so far. The new restriction states that both tasks should be non-preemptible, i.e. the execution of a task must be completed before another execution can be initiated.

To accommodate this requirement in our setting, it is more convenient to modify the DTA of PRE.TA and PRE.TB by removing the event *tick* from the self-loop of states in between the first and the last segments, if there is any. Let us denote the modified models by NPRE.TA and NPRE.TB. Note that NPRE.TA and PRE.TA are identical, since the execution time of PRE.TA is equal to one time unit; in other words, it has only one segment. Figure 4.13 shows DTA models of NPRE.TA and NPRE.TB.

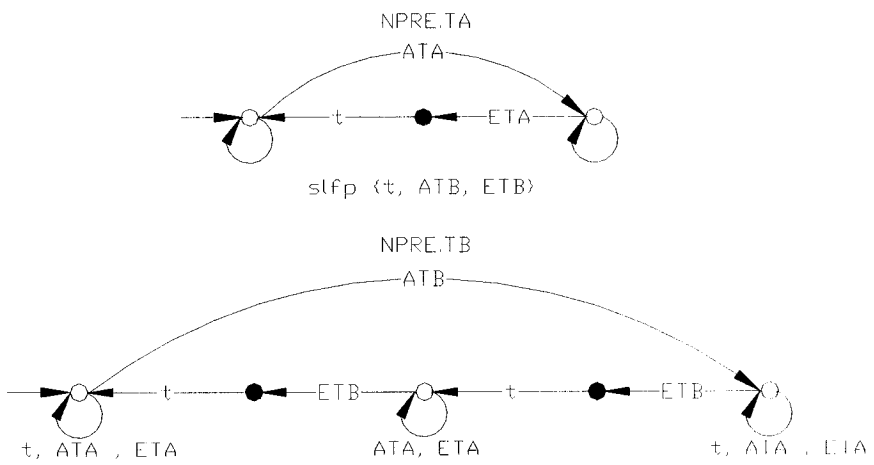


Figure 4.13: DTA models of non-preemptive tasks for controllers A and B.

To model the execution of control-generating tasks, we combine the two modified DTA, namely NPRE.TA and NPRE.TB, into a single DTA by taking their synchronous product. To enforce the periodic execution of both tasks, we use the same DTA model as presented in Example 4.3.1, i.e., SPEC.TAB.

A centralized supervisor NPRE.NPRI is obtained in 'TTCT' as follows:

NPRE.TAB = sync (NPRE.TA, NPRE.TB)

NPRE.NPRI = supcon (NPRE.TAB, SPEC.TAB).

It turns out that the language generated by NPRE.NPRI is not a timed language. We obtain its largest timed sublanguage by removing all paths in which 'continuity in time' is not respected. The result, denoted by NPRE.NPRI.SUP, is shown in Figure 4.14.

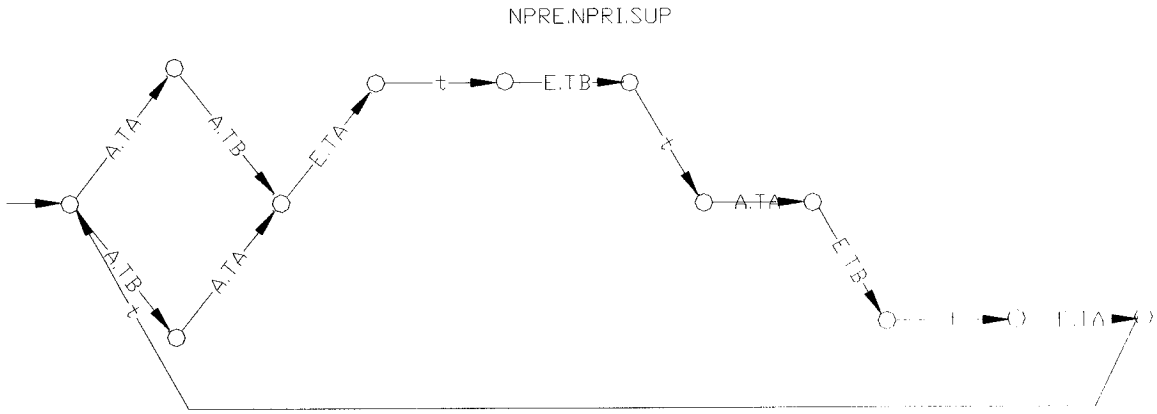


Figure 4.14: The non-preemptive/non-priority-based supremal supervisor for controlling the operations of valves A and B.

By stepping through the transition graph of NPRE.NPRI.SUP it can be verified that the cyclic sequence generated by NPRE.NPRI.SUP is:

$$[(A.TA;A.TB + A.TB;A.TA);E.TA;t;E.TB;t;A.TA;E.TB;t;E.TA;t]^{*}.$$

It can be seen that once the task NPRE.TB has started, it cannot be preempted by the task NPRE.TA. Note that the event A.TA is the only event that occurs between two consecutive segments of the task NPRE.TB, but it does not suspend the execution of NPRE.TB as arrival events are instantaneous and occupy no time.

In fact, this sequence is one of the safe sequences generated by the automaton of Figure 4.7, which contains all safe execution sequences for preemptive tasks PRE.TA and PRE.TB.

Figure 4.15 illustrates the only possible non-preemptive/non-priority-based schedule that can be selected by the supervisor for controlling both valves A and B. This schedule guarantees that the operation of valves A and B is in the right order as specified by SPEC.TA and SPEC.TB.

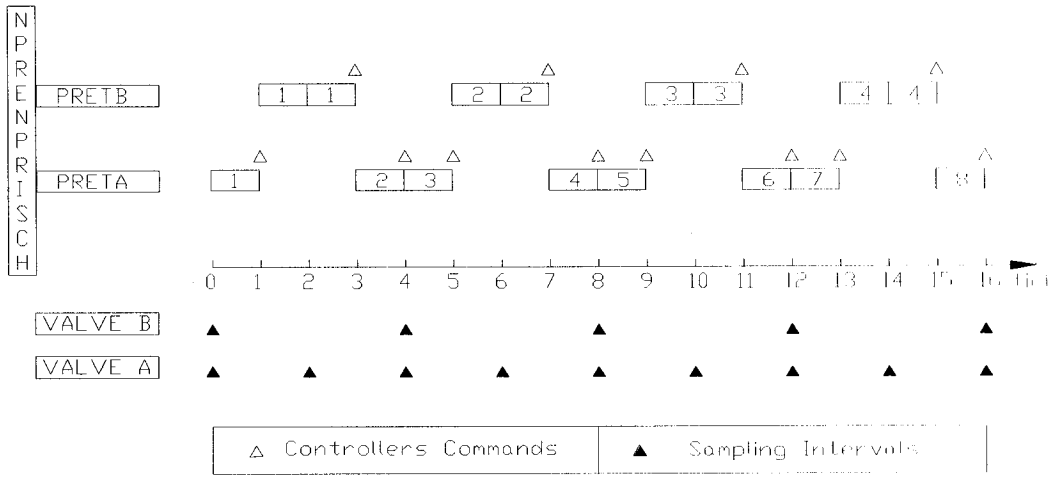


Figure 4.15: The non-preemptive/non-priority-based schedule for controlling the operations of valves A and B.

4.4.4 Non-preemptive/Priority-based Scheduling

As discussed earlier, in a non-preemptive schedule the execution of a task cannot be interrupted once started. A priority-based scheduling algorithm executes the highest-priority task first and allows the controller to give preference to important tasks.

To see how such a schedule can be generated using DTA supervisory control, let us consider the same priority relation as discussed earlier, i.e.:

$$P_1=\{(E.TA, E.TB)\}.$$

A centralized supervisor NPRE.PRI$_1$ is obtained using 'TTCT' software as follows.

$$NPRE.PRI.SPEC_1 = \underline{sync}\ (NPRE.TAB_{P1}, SPEC.TAB)$$

$$NPRE.PRI_1 = \underline{supcon}\ (NPRE.TAB, NPRE.PRI.SPEC_1).$$

Unfortunately, there is no path in $NPRE.PRI_1$ in which 'continuity in time' is respected, and therefore $NPRE.PRI.SUP_1 = \varnothing$. We conclude that no safe sequence exists that guarantees on-time execution of both tasks according to the desired specifications SPEC.TA and SPEC.TB.[4] In other words, no feasible non-preemptive/priority-based schedule exists for the tasks NPRE.TA and NPRE.TB.

Changing the priority relation to $P_2=\{(A.TA, A.TB)\}$ results in a nonempty supremal supervisory control which can be obtained in 'TTCT' software as follows:

$$PRE.PRI.SPEC_2 = \underline{sync}\ (NPRE.TAB_{p_2}, SPEC.TAB)$$
$$NPRE.PRI_2 = \underline{supcon}\ (NPRE.TAB, NPRE.PRI.SPEC_2).$$

Once again, the language generated by $NPRE.PRI_2$ is not a timed language. We obtain its largest timed sublanguage by removing all paths in which 'continuity in time' is not respected. The result, denoted by $NPRE.PRI.SUP_2$, is shown in Figure 4.16.

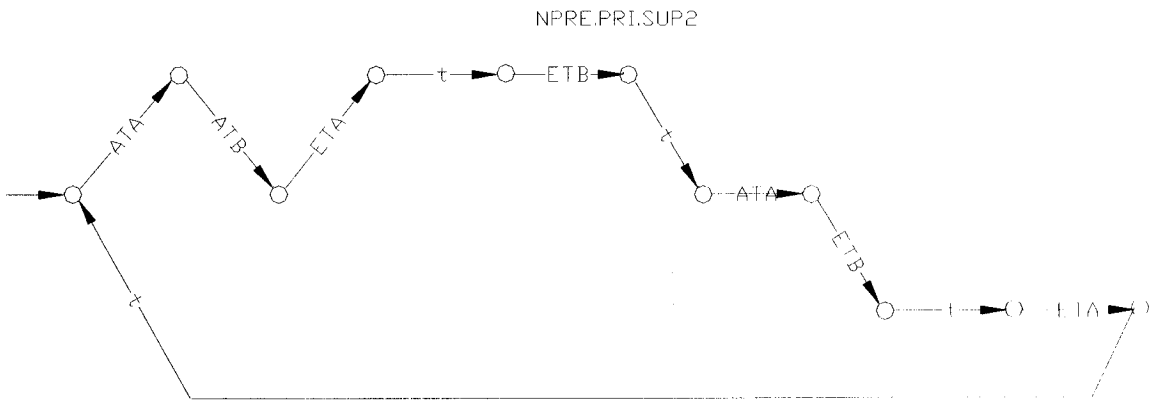


Figure 4.16: The non-preemptive/priority-based supremal supervisor for controlling the operations of valves A and B

[4] An empty supervisor returned by supcon is discussed in Appendix C.

Figure 4.17 illustrates the only possible non-preemptive/priority-based schedule corresponding to the priority relation P_2 that can be selected by the supervisor for controlling both valves A and B. This schedule guarantees that the operation of valves A and B is in the right order as specified by SPEC.TA and SPEC.TB. This is in fact identical to the sequence generated by the earliest deadline first scheduling algorithm when no preemption is allowed.

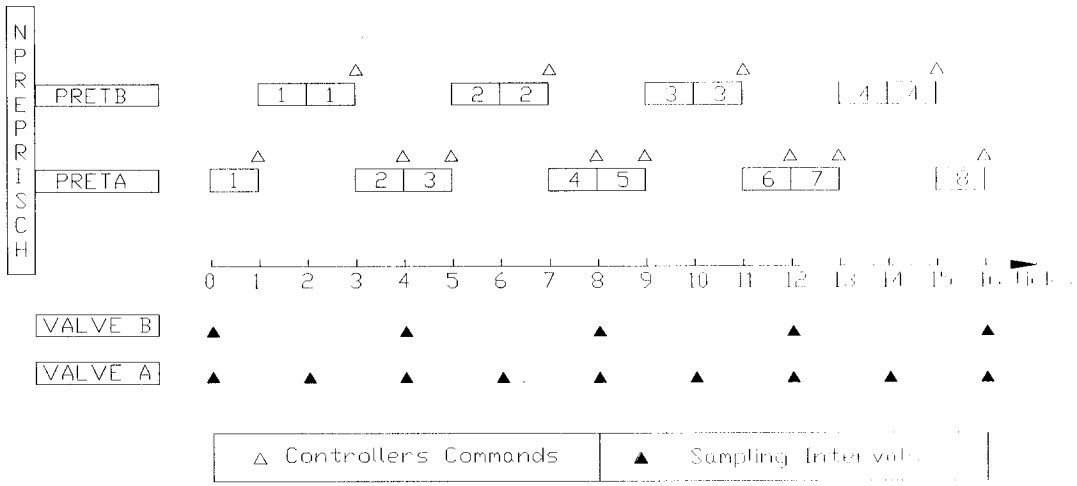


Figure 4.17: The non-preemptive/priority-based schedule for controlling the operations of valves A and B.

4.5 Task Scheduling in Supervisory Control: An application

This example shows how modeling a process using DTA enable us to meet several specifications with a varying production layout. Let us consider an embedded system containing three alarm sensors connected to a processor via a slow-speed serial link. Individual monitoring processes of alarm points #1, #2 and #3 take 2, 3 and 4 *msec,* respectively. It is assumed that alarm points #1, #2 and #3 are scanned once every 10, 9 and 9 *msec,* respectively.

Monitoring process times and the scanning periods required by the serial link are summarized in Table 4.4.

Required monitoring process times			Required scanning periods		
Alarm point	Alarm point	Alarm point	Alarm point	Alarm point	Alarm point
2 *msec*	3 *msec*	4 *msec*	10 *msec*	9 *msec*	9 *msec*

Table 4.4: Monitoring process times and scanning periods required by the serial link.

Let us denote the serial link monitoring processes by PRE.T1, PRE.T2, and PRE.T3, respectively, as shown in Figure 4.18.

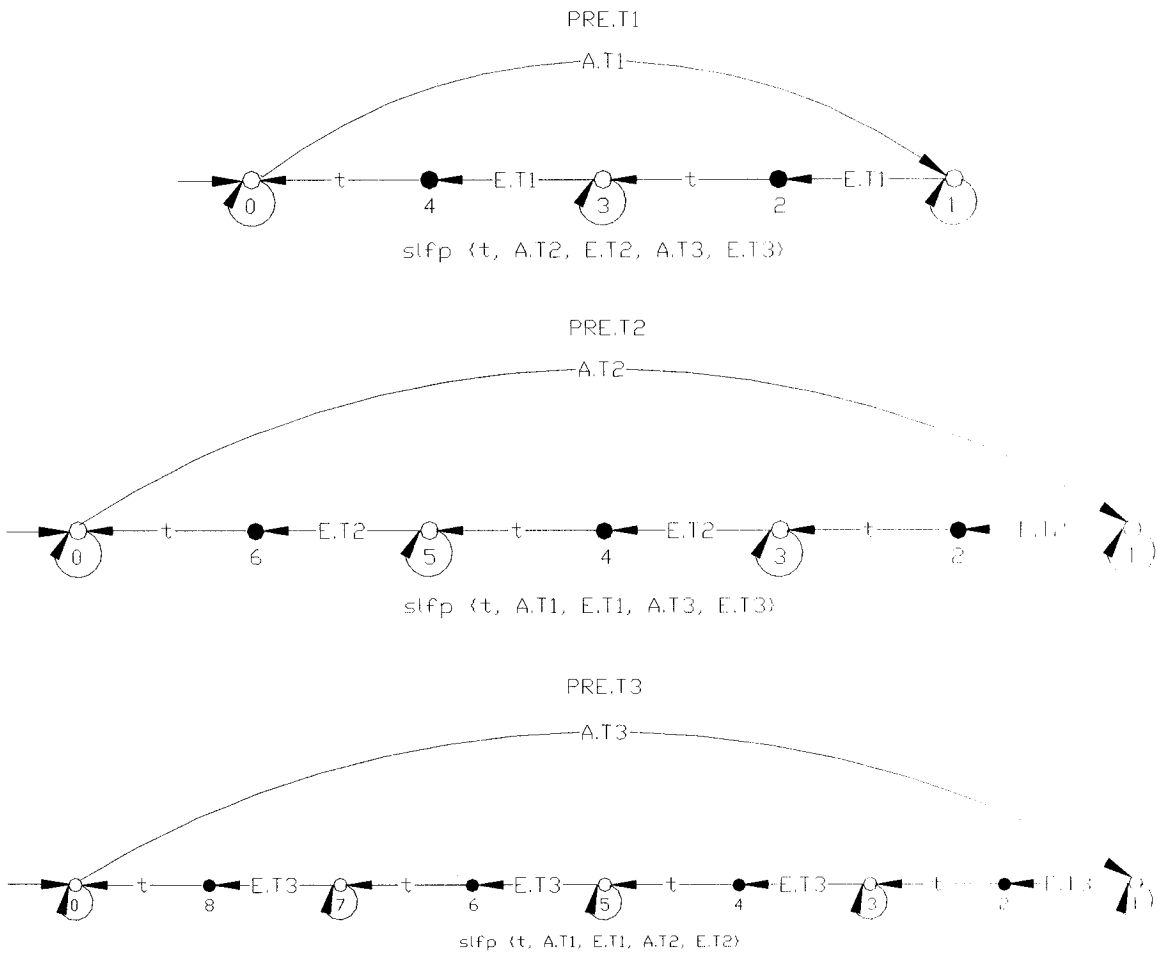


Figure 4.18: The DTA models of the serial link monitoring processes.

The events for each process are summarized in Table 4.5.

Event	Descriptions	Event	Descriptions
A.T1	Arrival of the task PRE.T1	E.T1	Commencing execution of one segment of the task PRE.T1
A.T2	Arrival of the task PRE.T2	E.T2	Commencing execution of one segment of the task PRE.T2
A.T3	Arrival of the task PRE.T3	E.T3	Commencing execution of one segment of the task PRE.T3
t	One *msec*		

Table 4.5: Events descriptions for the serial link.

We introduce three specifications named SPEC.T1 and SPEC.T2 and SPEC.T3 to guarantee that events A.T1, A.T2 and A.T3 occur once every 10, 9 and 9 *msec,* respectively. This will enforce the scanning periods of 10, 9 and 9 *msec* for the serial link to scan alarm points #1, #2 and #3, respectively. The task specifications, SPEC.T1, SPEC.T2 and SPEC.T3 are illustrated in Figure 4.19.

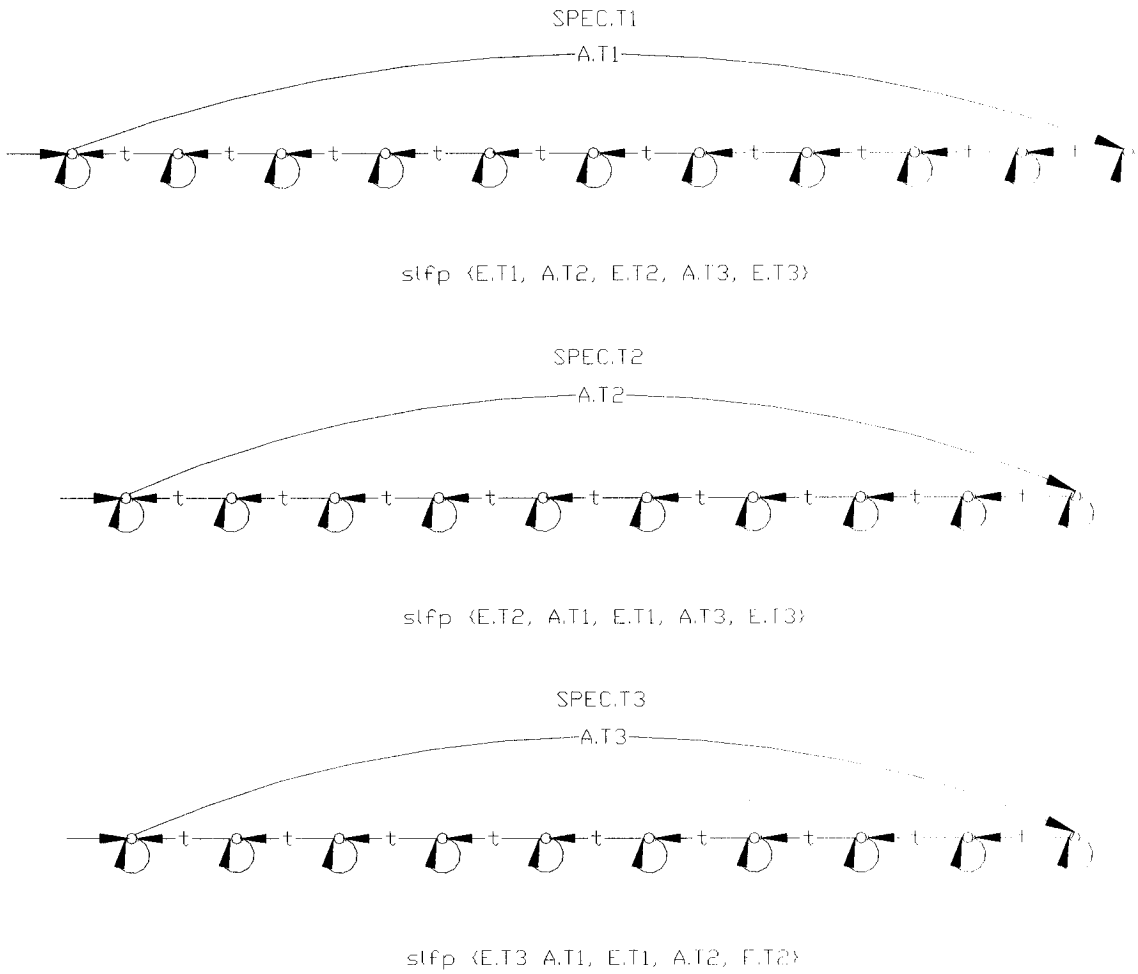


Figure 4.19: The DTA models of the serial link monitoring processes specifications.

In order to model the combined execution of serial link processes, we take the synchronous product of PRE.T1, PRE.T2 and PRE.T3 to obtain a single DTA named PRE.T123.

PRE.T12 = sync (PRE.T1, PRE.T2)

PRE.T123 = sync (PRE.T12, PRE.T3) .

The synchronous product of the individual specifications can capture the periodic execution of tasks PRE.T1, PRE.T2 and PRE.T3, that is:

SPEC.T12 = sync (SPEC.T1, SPEC.T2)
SPEC.T123 = sync (SPEC.T12, SPEC.T3).

The supremal supervisory control, which contains all allowable sequences of task executions, can then be obtained by taking the supcon of plant and specification.

PRE.NPRI.SUP123 = supcon (PRE.T123, SPEC.T123).

It is worthwhile to note that for the sake of convenience all events are assumed to be controllable. Since the transition graph of PRE.NPRI.SUP123 contains 3508 states and 5218 transitions, we present below only three different schedules generated by the supcon procedure.

PRE.NPRI.SUP123 schedule #1:

A.T2;A.T3;A.T1;E.T1;t;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T3;t;E.T3;t ;E.T3;t;E.T3;t;
S_1.

PRE.NPRI.SUP123 schedule #2:

A.T2;A.T3;A.T1;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T1;t;E.T3;t;E.T3;t;E.T1;t;
S_1.

PRE.NPRI.SUP123 schedule #3:

A.T2;A.T3;A.T1;E.T3;t;E.T3;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T2;t;E.T1;t;E.T1;t;
S_2.

Where S_1 and S_2 are presented in Appendix D. Careful examination of the above schedules reveals that:

- A.T1 occurs once every 10 *msec.*
- A.T2 occurs once every 9 *msec.*
- A.T3 occurs once every 9 *msec.*
- PRE.T1 is completely executed once every 10 *msec.*
- PRE.T2 is completely executed once every 9 *msec.*
- PRE.T3 is completely executed every 9 *msec.*

It can be seen that in each schedule the timing requirements specified earlier for the tasks are all met. Thus, the supervisor PRE.NPRI.SUP123 contains all safe sequences of task executions with the guarantee that all desired timing requirements are met.

To obtain a preemptive/priority-based scheduling scheme, let us assume that alarm point #2 covers the most sensitive zone of the plant, which is to be controlled by the embedded system of our example. To this end we define a priority relation to ensure that task PRE.T2 is dispatched with a higher priority than either of the tasks PRE.T1 or PRE.T3.

The priority relation P is defined as:

$$P = \{(\text{E.T2, E.T1}), (\text{E.T2, E.T3})\}.$$

To obtain a schedule that conforms with the priority relation P we first obtain PRE.T123_P by applying priorities to the open loop system, i.e. PRE.T123, as described in the constructive procedure of Definition 2.3.1.

The specification of all requirements on the plant behavior is denoted by PRE.PRI.SPEC.T123 and is obtained by taking the synchronous product of PRE.T123_P and SPEC.T123. A centralized supervisor PRE.PRI.SUP123 is then obtained in 'TTCT' by taking the supcon of plant PRE.T123 and the final specification PRE.PRI.SPEC.T123.

PRE.PRI.SPEC.T123 = sync (PRE.T123$_P$, SPEC.T123)

PRE.PRI.SUP123 = supcon (PRE.TAB, PRE.PRI.SPEC.T123).

The supremal supervisory control PRE.PRI.SUP123 contains 708 states and 901 transitions. By examining the schedules generated by PRE.PRI.SUP123 it can be verified that the event E.T2 is the only event that can occur at the state reached by the sequence 'A.T2;A.T3;A.T1', compared with PRE.NPRI.SUP123 schedules where all three events E.T1, E.T2 and E.T3 have a chance to occur at the same state. In fact, this sequence can be continued until all task executions are complete according to the required specifications. For instance one of the major cyclic sequences generated by PRE.PRI.SUP123 is:

PRE.PRI.SUP123 schedule #1:

A.T2;A.T3;A.T1;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T1;t;E.T3;t;E.T3;t;E.T1;t;
S_1.

It is worthwhile to mention that PRE.NPRI.SUP123 is optimal and contains all possible execution sequences. In comparison, PRE.PRI.SUP123 disables E.T1 and E.T3 and grants E.T2 a chance to occur, which results in eliminating schedules #1 and #3 of PRE.NPRI.SUP123 while keeping its schedule #2. Thus, our proposed approach keeps only those schedules that conform with the desired priority relation.

To see how a non-preemptive/non-priority-based schedule can be generated for our example using DTA supervisory control, let us assume that tasks PRE.T1, PRE.T2 and PRE.T3 are non-preemptive, i.e. the execution of a task must be completed before another execution can be initiated. To accommodate this requirement in our setting, it is more convenient to modify the DTA of PRE.T1, PRE.T2 and PRE.T3 by removing the *tick* event from the self-loop of states in between the first and the last segments. Let us denote the modified models by NPRE.T1, NPRE.T2 and NPRE.T3.

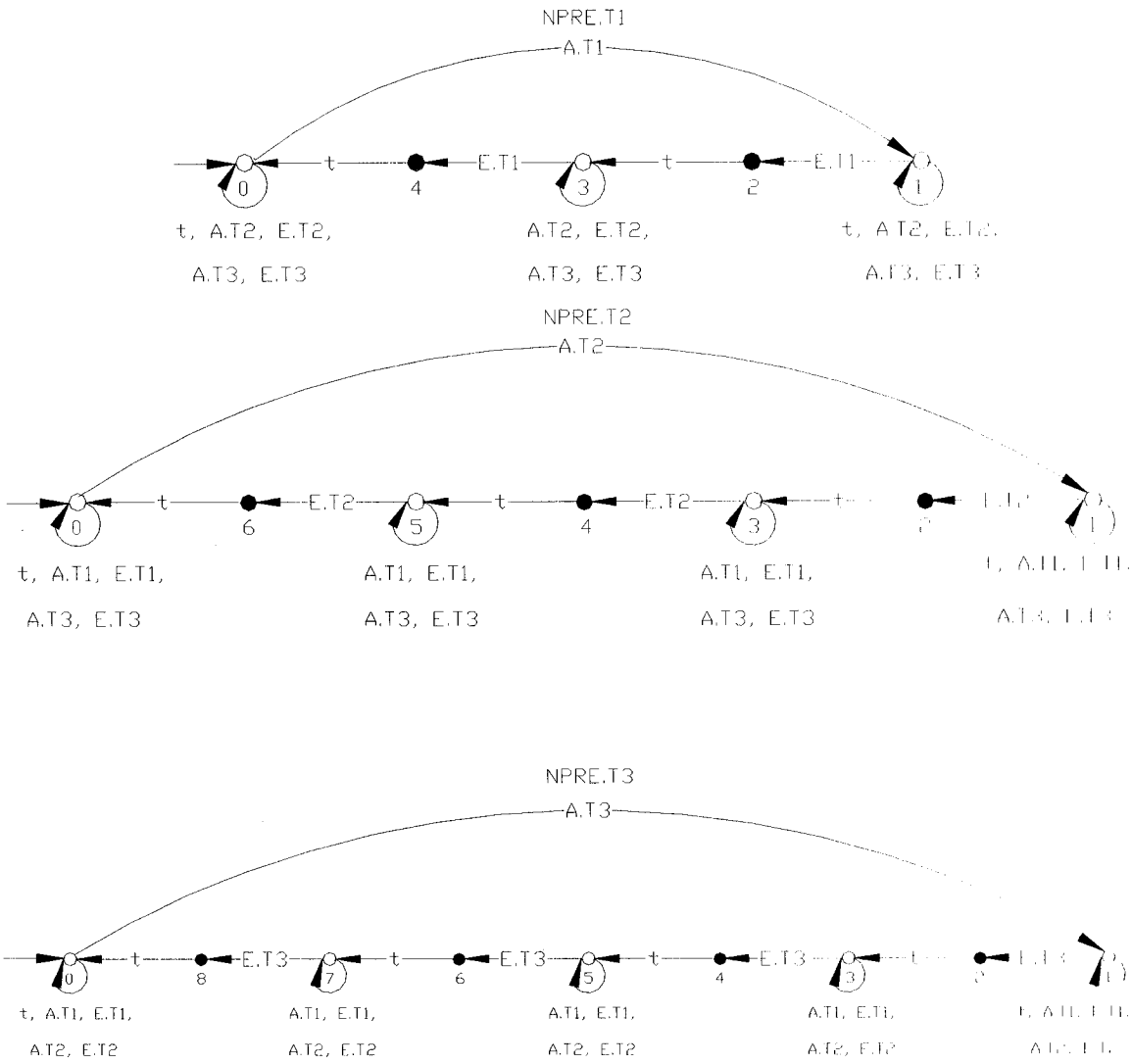


Figure 4.20: The DTA models of the serial link monitoring process preemptive

A model for the entire plant is then obtained by combing the modified DTAs into one using the synchronous product operation. To implement periodicity of all tasks we use the same DTA model introduced in the preemptive/non-priority case, i.e., SPEC.T123. A centralized supervisor NPRE.NPRI.SUP123 is obtained in 'TTCT' software as described below:

NPRE.T12 = sync (NPRE.T1, NPRE.T2)

NPRE.T123 = sync (NPRE.T12, NPRE.T3)

NPRE.NPRI.SUP123 = supcon (NPRE.T123, SPEC.T123).

Examining the transition table of NPRE.NPRI.SUP123 reveals that in some paths continuity in time is not observed. In other words, the language generated by NPRE.NPRI.SUP123 is not a timed language. In order to obtain its largest timed sublanguage, we remove all paths in which 'continuity in time' is not respected.

Since the supremal supervisory control NPRE.NPRI.SUP123 contains 5628 states and 5969 transitions, we present below only three different schedules generated by the supcon procedure.

NPRE.NPRI.SUP123 schedule #1:

A.T3;A.T1;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t; S_3.

NPRE.NPRI.SUP123 schedule #2:

A.T2;A.T1;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t; S_4.

NPRE.NPRI.SUP123 schedule #3:

A.T2;A.T1;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t; S_5.

Where S_3, S_4 and S_5 are presented in Appendix D.

After careful examination of the above schedules we find that execution of any of the tasks NPRE.T1, NPRE.T2 or NPRE.T3 must be continued to completion, and only then another task may be started.

It is worthwhile to note that despite the fact that an arrival event can occur while a task is being executed, the execution is not really interrupted because arrival events are instantaneous.

At last we study a non-preemptive/priority-based schedule. After defining a priority relation, we determine whether a feasible non-preemptive schedule for satisfying the timing requirements exists.

We begin by defining the same priority relation as discussed earlier, i.e.:

P={(E.T2, E.T1), (E.T2, E.T3)}.

A centralized supervisor NPRE.PRI.SUP123 is obtained using 'TTCT' software as follows.

NPRE.PRI.SPEC123 = sync (NPRE.T123p, SPEC.T123)

NPRE.PRI.SUP123 = supcon (NPRE.T123, NPRE.PRI.SPEC123).

Note that the language generated by NPRE.PRI.SUP123 is not a timed language. We obtain its largest timed sublanguage by removing all paths in which 'continuity in time' is not respected. Since the supremal supervisory control NPRE.PRI.SUP123 contains 2020 states and 2167 transitions, below we present only one of the schedules generated by the supcon procedure:

NPRE.PRI.SUP123 schedule #1:

A.T2;A.T1;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;
S_4.

Which is identical to schedule #2 of NPRE.NPRI.SUP123.

One can verify that NPRE.NPRI.SUP123 will reach a state where all events E.T1, E.T2 and E.T3 are eligible to occur. However, NPRE.PRI.SUP123 disables E.T1 and

E.T3, and grants the chance of occurrence to E.T2, which results in eliminating schedules #1 and #3 of NPRE.NPRI.SUP123, while keeping its schedule #2.

In other words, NPRE.PRI.SUP123 results in those schedules that conform with the desired priority relation P while eliminating those schedules that do not conform with P.

Chapter 5

Conclusions and Related Works

5.1 Conclusion

We have proposed a method for designing a priority-based supremal supervisory control such that the system under supervision conforms with a given priority relation. We have shown that the theory of supervisory control of discrete-event systems can be applied to task scheduling. We have developed a formal framework based on discrete timed automata to address task scheduling. Our models encompass non-preemptive/preemptive as well as periodic/non-periodic tasks.

We then employed DTA supervisory control to obtain non-preemptive/preemptive and priority/non-priority-based scheduling. In priority-based scheduling the theory developed in Chapter 1 is used to design supervisors that conform with a given priority relation.

In our formulation of the scheduling problem, a task is assumed to be any physical activity that consumes a certain amount of time; opening of a valve and loading of a truck are examples. As such, our proposed method can be applied to industrial processes to ensure on-time execution of various tasks.

5.2 Related Works

In the area of real-time scheduling several results have been established concerning preemptive execution of periodic tasks. Conditions for schedulability and

universality of scheduling algorithms have been developed (Liu and Layland, 1973; Audsley et al., 1995).

Compared to preemptive scheduling, reported results on non-preemptive scheduling appear to be less extensive. In (Frederickson, 1983; Garey et al., 1981), the problem of non-preemptive scheduling is investigated where given a precedence order, tasks are arrived only once and each task requires only a single unit of processing time. In (Yuan et al., 1994), an approach is proposed for non–preemptive scheduling of non-periodic tasks on a processor. Results on more general characterizations of periodic tasks have been reported in (Jeffay et al., 1991), where non-preemptive scheduling without inserted idle time on a single processor is studied; sufficient conditions for schedulability of a set of periodic tasks are derived, and the universality of earliest deadline first algorithm is examined.

In [6] a method for scheduling non-preemptive execution of periodic tasks with hard deadlines on a single processor is investigated. This approach utilizes techniques from formal language and automata theories to generate a complete set of solutions (if one exists) to a given scheduling problem.

Our proposed method formally covers the problem of preemptive as well as non-preemptive tasks scheduling, compared with semiformal works dominant in real-time literatures. It generates a complete set of solutions (if one exists) to a given scheduling problem. Such solutions are correct by construction, and thus do not require subsequent verification.

5.3 Future works

Although the theory of real-time scheduling can be developed using rigorous mathematics, results are often presented less formally. The assumptions made are often imprecise and based on intuitive explanations rather than rigorous arguments. It is difficult to correctly analyze complex scheduling policies using only informal models and proofs. Formal method should become an essential tool for validating the most critical aspects of real-time systems.

Real-time scheduling problems are complex, and must be certified to the highest degree of assurance for supporting critical applications, and thus require very precise and detailed verification, or better yet, synthesis approaches. Although we have taken initial steps in this direction by formalizing real-time scheduling theory, this is by no means a trivial exercise, and remains an important area for further research.

Another interesting subject is to develop a computer tool for automating the procedures in our framework. For instance if a priority relation is defined we need to obtain the prioritized system manually. In order to obtain the prioritized system from the original system, we should remove transitions labeled with lower priority events from states in which transitions labeled with higher priority events have a chance to occur. This objective is currently achieved by stepping through the transition graph. Developing a software that automatically generates the prioritized system or even check the 'continuity in time' condition is therefore highly desired.

Another interesting issue is modeling precedence relations where some tasks must be executed before others can be initiated. Introducing an appropriate DTA specification could be a way to represent preference relations in our framework.

Appendix A

Conformance with a priority relation

Recall that conformance of a sublanguage with a priority relation P with respect to a language L was defined as:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in K \Rightarrow s\sigma' \notin L. \qquad (i)$$

One may be tempted to define conformance as:

$$\forall s \in \Sigma^*, \forall \sigma, \sigma' \in \Sigma. (\sigma', \sigma) \in P \wedge s\sigma \in K \Rightarrow s\sigma' \notin K. \qquad (ii)$$

However, the above definition suffers from a setback: namely, when K does not conform with P, a largest sublanguage of K which conforms with the priority relation P does not in general exist.

To illustrate this let us consider an example. Two languages, L and $K \subseteq L$, are shown in Figure A.1.

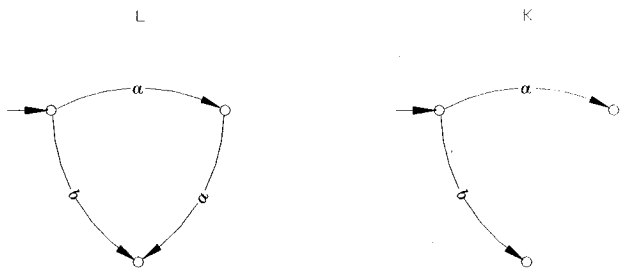


Figure A.1: The DES models of L and K

Let us define the priority relation as $P = \{(a, b)\}$. Note that K does not conform with the priority relation. Two sublanguages of K that conform with the priority relation according to (ii) are illustrated in Figure A.2.

However, the union of K_1 and K_2, which happens to be equal to K, does not conform with the priority relation P. It follows that the definition provided in (ii) is not preserved under union, and as a result a largest sublanguage of K that conforms with the priority relation P does not exist.

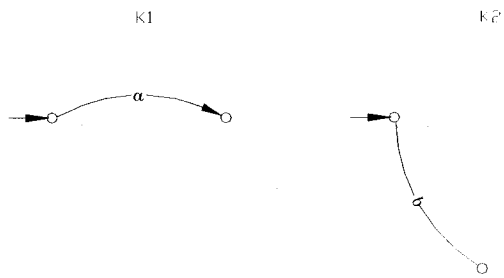


Figure A.2: The DES models of K1 and K2

The supremal sublanguage of K that conforms with the priority relation P with respect to L can be obtained by applying the definition provided in (i). The supremal sublanguage of K, denoted by K_s is illustrated in Figure A.3.

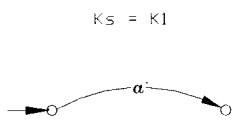


Figure A.3: The DES models of Ks

Appendix B

Tasks schedulability using utilization techniques

A fundamental question in scheduling theory is whether a given set of tasks is schedulable. In other words, whether the system meets its requirements within the specified deadlines [14].

When a system is made up of n tasks, the overall system utilization, denoted by U, is defined to be $U = \sum_{j=1}^{n} (C_j / T_j)$, where C_j and T_j are the execution time and sampling period of task J , respectively.

Given any schedule, to guarantee meeting all tasks deadlines it is necessary for percentage utilization to be smaller than 1. Sufficient conditions for RM and EDF are provided in the following theorems.

Theorem 1 (RM Scheduling)

A system of n independent, preemptible periodic tasks with relative deadlines equal to their respective periods can be feasibly scheduled according to RM priority assignment if its total utilization U is less than or equal to $n(2^{1/n}-1)$ [15].

Theorem 2 (EDF Scheduling)

A system of n independent, preemptible tasks can be feasibly scheduled according to EDF priority assignment if its utilization is equal to or less than one [15].

Appendix C

Supcon and task schedulability

The result of the supcon operation is the set of all safe execution sequences. If the resulting supervisor is nonempty, then it contains all execution sequences that guarantee all task deadlines are met. Otherwise, it can be concluded that no sequence exists that can guarantee the on-time execution of all tasks; in other words, the tasks are not schedulable.

We might conjecture that the resulting supervisor is nonempty if and only if $U \leq 1$.

To show this, we must show that:

1. $SUPCON \neq \varnothing \Rightarrow U \leq 1$.
2. $U \leq 1 \Rightarrow SUPCON \neq \varnothing$.

1. Show $SUPCON \neq \varnothing \Rightarrow U \leq 1$.

If $SUPCON \neq \varnothing$, it follows that the tasks are schedulable. In addition, as discussed in Appendix B, a set of tasks is schedulable only if the system utilization is smaller than one. We conclude that:

$$SUPCON \neq \varnothing \Rightarrow U \leq 1. \quad (1)$$

2. Show $U \leq 1 \Rightarrow SUPCON \neq \varnothing$.

According to Theorem 2 presented in Appendix B an earliest-deadline-first schedule meets all task deadlines if and only if $U \leq 1$. Thus, if $U \leq 1$, there exists an EDF schedule for the given set of tasks. In addition, the existence of a schedule for a set of tasks implies that the resulting supervisor is nonempty. We conclude that:

$$U \leq 1 \Rightarrow SUPCON \neq \varnothing. \quad (2)$$

From (1) and (2) it follows that:

$$SUPCON \neq \varnothing \Leftrightarrow U \leq 1. \quad \therefore$$

Appendix D

Event sequences of Section 4.5

The event sequences of strings S_1, S_2, S_3, S_4 and S_5 are as follows:

S_1

A.T2;A.T3;E.T2;t;A.T1;E.T2;t;E.T2;t;E.T3;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;A. T2;
A.T3;E.T2;t;E.T2;t;A.T1;E.T2;t;E.T3;t;E.T3;t;E.2T3;t;E.T1;t;E.T3;t;E.T1;t;A.T3;A.T2;
E.T2;t;E.T2;t;E.T2;t;A.T1;E.T1;t;E.T3;t;E.T1;t;E.T3;t;E.T3;t;A.T2;A.T3;E.T2;t;
E.T2;t;E.T2;t;E.T3;t;A.T1;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;
E.T2;t;E.T3;t;E.T3;t;A.T1;E.T1;t;E.T1;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;
E.T3;t;E.T3;t;E.T3;t;A.T1;E.T1;t;E.T3;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;
E.T3;t;E.T3;t;E.T3;t;A.T1;E.T1;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;
E.T3;t;E.T3;t;t;A.T1;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T1;t;E.T3;t;E.T3;
t;E.T3;t;t.

S_2

A.T3;A.T2;E.T2;t;A.T1;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T3;t;E.T2;t;E.T2;t;A.T3;A.
T2;E.T2;t;E.T3;t;A.T1;E.T3;t;E.T2;t;E.T3;t;E.T1;t;E.T1;t;E.T3;t;E.T2;t;A.T3;A.T2;E.T3;
t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T1;t;E.T2;t;E.T1;t;E.T2;t;E.T2;t;A.T3;A.T2;E.T2;t;E.T2;t;
E.T3;t;E.T3;t;A.T1;E.T1;t;E.T3;t;E.T1;t;E.T3;t;E.T2;t;A.T2;A.T3;E.T3;t;E.T3;t;E.T2;t;E.
T3;t;E.T2;t;A.T1;E.T1;t;E.T1;t;E.T2;t;E.T3;t;A.T3;A.T2;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T
2;t;E.T2;t;A.T1;E.T1;t;E.T1;t;E.T2;t;A.T3;A.T2;E.T3;t;E.T2;t;E.T3;t;E.T3;t;E.T2;t;E.T2;
t;E.T3;t;A.T1;E.T1;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;t;E.T2;t;E.T2;t;E.T1;t
;A.T1;E.T1;t;A.T2;A.T3;E.T3;t;E.T3;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T1;t;t;E.T3;t.

S_3

A.T3;A.T2;E.T2;t;A.T1;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.
T2;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T1;t;E.T1;t;E.T2;t;E.T2;t;E.T2;t;A.T3;A.T2;E.T2;

t;E.T2;t;E.T2;t;A.T1;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.T2;t;t;E.T2;t;E.T2;t;A.T1;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T1;t;A.T1;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2t;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T1;t;A.T3;E.T1;t;E.T1;t;A.T2;A.T3;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T2;t;E.T2;t;E.T2;t;A.T1;E.T1;t;E.T1;t;A.T2;A.T3;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T1;t;A.T2;A.T3;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;t;E.T2;t;E.T2;t;E.T2;t.

S_4

A.T2;A.T3;E.T2;t;A.T1;E.T2;t;E.T2;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;A.T1;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;A.T1;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T1;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T1;t;E.T1;t;A.T3;A.T2;t;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t.

S_5

A.T2;A.T3;E.T2;t;A.T1;E.T2;t;E.T2;t;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;A.T1;E.T2;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;A.T1;E.T1;t;E.T1;t;E.T3;t;E.T3;t;E.T3;t;E.T3;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T3;t;E.T1;t;E.T1;t;A.T3;A.T2;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;A.T1;E.T3;t;E.T1;t;E.T1;t;A.T2;A.T3;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T1;t;E.T1;t;A.T3;A.T2;t;t;E.T2;t;E.T2;t;E.T2;t;E.T3;t;E.T3;t;E.T3;t;A.T1;E.T3;t;A.T3;A.T2;E.T3;t;E.T3;t;E.T3;t;E.T3;t;E.T2;t;E.T2;t;E.T2;t;E.T1;t;E.T1;t.

References

[1] Peter J. G. Ramadge and W. M. Wonham. "Control of Discrete Event Systems". The institute of Electrical and Electronics Engineers, 77(1), pp. 81-98, 1989.

[2] W.M. Wonham. "Notes on Control of Discrete Event Systems". Systems Control Group, Department of Electrical and Computer Engineering, University of Toronto, Canada, 2002.

[3] C.G. Cassandras and S. Lafourtune. "Introduction to Discrete Event Systems". Kluwer Academic Publishers, 1999.

[4] Software "TTCT", available at http://www.control.utoronto.ca/DES.

[5] S.Blouin, M. Guay and K. Rudie. "An Application of Discrete-Event Theory to Truck Dispatching", Technical Report #2000-440, Queens University, 2000.

[6] Peter C. Y. Chen and W. M. Wonham. "Real–times Supervisory Control of a Processor for a Non-preemptive Execution of Periodic Tasks". The international Journal of Time-Critical Computing Systems, pp. 1-28, 2002.

[7] Bruno Dutertre and Victoria Stavridou. Formal Analysis for Real-time Scheduling". The institute of Electrical and Electronics Engineers., p.1.D.4-1,2000.

[8] C. L. Liu and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". J. Assoc. Comput. Mach. Vol. 24, pp. 46-61, 1973.

[9] P. Gohari. "Analysis and Design of Real Time Computer Control Systems". ELEC 6061, Lecture 10. Department of Electrical and Computer Engineering, Concordia University, Canada, 2003.

[10] J. P. Lehoczky, L. Sha, and Y. Ding. "Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment". Proc. Real-Time Systems Symp, pp. 261-270, 1987.

[11] G. Fohler. "Changing Operational Modes in the Context of Pre-run-time Scheduling". Real-time Systems Group, Vienna University of Technology, Austria, 1993.

[12] H. M. Hanisch and S. Kowalewski. "Algebraic Synthesis and Verification of Discrete Supervisor Controllers for Forbidden Path Specifications". 4th International Conf. on Computer Integrated Manufacturing and Automation Technology. IEEE Computer Society Press, pp. 157-162. 1999.

[13] H. M. Deitel, "An Introduction to Operating Systems". IEICE Trans. Inf. & Syst.Vol. E76-D, pp. 1333-1340, Nov. 1993.

[14] J. Cooling."Software Engineering for Real-Time Systems". Addison Wesley, 2003.

[15] Jane W.S.Liu. "Real-Time Systems". Prentice Hall, New Jersey, 2000.