REAL-TIME SUPERVISORY CONTROL OF
A PROCESSOR FOR NON-PREEMPTIVE
EXECUTION OF PERIODIC TASKS

by

Peter C. Y. Chen\textsuperscript{1} and W. M. Wonham

Systems Control Group Report No. 9802
98.08.17

Systems Control Group
Department of Electrical & Computer Engineering
University of Toronto
Toronto, Ontario, Canada
M5S 3G4

\textsuperscript{1}Currently with Gintic Institute of Manufacturing Technology, 71 Nanyang Drive, Singapore 638075
Email: cychen@gintic.gov.sg
Abstract

In this article, a method for scheduling a processor for non-preemptive execution of periodic tasks is presented. This method is based on the formal framework of supervisory control of timed discrete-event systems. It is shown that, with this method, the problem of determining schedulability and the problem of finding a scheduling algorithm are dual since a solution to the former necessarily implies a solution to the latter and vice versa. Furthermore, the solution to the latter thus obtained is complete in the sense that it contains all “safe” sequences of task execution with the guarantee that no deadline is missed. Examples are described to illustrate this method. Implication of the results and computational complexity associated with this method are discussed.
1 Introduction

Real-time scheduling has been a fundamental issue in the development of hard real-time systems [4, 15]. Most research in this area has been conducted mainly in the context of preemptive computer operating systems. However, the theory of real-time scheduling (especially non-preemptive scheduling where execution of a task cannot be interrupted) has potentially far wider applications, in other areas such as manufacturing.

As manufacturing processes are increasingly being adapted towards just-in-time operation, proper management of manufacturing resources could significantly improve the efficiency of manufacturing processes by guaranteeing on-time performance. For example, consider a Computer-Numerical-Control (CNC) machine which can be programmed to process 10 types of parts. Suppose that the time for processing each part (without interruption) is known, and each part has a (possibly unique) periodic delivery deadline. For just-in-time manufacturing operation, it may be desirable to schedule the CNC machine such that various mixes of the 10 types of parts can be produced periodically while meeting certain hard deadlines. Such time-driven utilization of a CNC machine falls within the problem domain of non-preemptive scheduling of periodic tasks.

In this article, we consider the problem of scheduling the non-preemptive execution of a set of periodic tasks with hard deadlines on a single processor. A periodic task is a task that “arrives” (or in other words, is invoked) repetitively at fixed time instants. Often the deadline of a task coincides with the end of the task period; that is, a task must be completed before its next invocation. The time required to execute a task is usually known (or at least a worst-case estimate can be determined). A set of tasks is schedulable if it is possible to schedule their execution such that all deadlines are met.

Non-preemptive execution of a task means that, once started, the execution of a task cannot be interrupted. The processor must perform the task to its completion, and only then may initiate execution of another task. Compared to preemptive task execution, non-preemptive task execution incurs less overhead and is easier to implement. But more importantly, for certain types of application, such as the scheduling of a CNC machine discussed above, non-preemptive task execution might be the only economically viable
solution.

In the literature on real-time systems theory, the problem of non-preemptive scheduling of a processor to meet the deadlines of a set of tasks is usually “decomposed” into two sub-problems: (i) determining the condition(s) under which a set of tasks is schedulable, and (ii) finding an algorithm to generate a sequence by which the tasks are to be executed. Our method for non-preemptive scheduling of periodic tasks on a processor, as presented in this article, does not follow this “problem-decomposition” approach. Instead, we treat the overall scheduling problem as an integral problem, and show that, with our proposed method, solving the schedulability problem implies solving the scheduling-algorithm problem and vice versa.

Our method is based on the theory of supervisory control of discrete-event systems [14]. A discrete-event system (DES) is a dynamical process whose evolution is characterized by abrupt occurrence of events. In a logical DES, the system is characterized by a set of states and the transitions (also called events) among these states. The behavior of the system is thus described by a sequence of events. The occurrence of an event (caused by some unmodelled mechanism internal or external to the system) instantaneously moves the system to a new state.

In a timed discrete-event system (TDES), the behavior of the system is also influenced by the temporal characteristics of the event sequence. Specifically, in a TDES, the occurrence of an event could be triggered either by some unmodelled mechanism acting on (or in) the system, or by the passage of time. Supervisory control of a TDES in essence means timely disablement of certain events in the transition structure of the TDES such that its behavior meets certain specifications. A supervisor which prevents events from occurring only when absolutely necessary is described as minimally restrictive. The minimally restrictive behavior of a supervisor is expressed in a so-called supremal controllable sublanguage.

To investigate the problem of scheduling a processor for non-preemptive task execution, we first model the execution of a set of periodic tasks as a TDES. We then compute the supremal controllable sublanguage with respect to the TDES and the desired behavior of the system. To determine whether a set of tasks can be scheduled such that all
deadlines of the tasks are met, we simply check whether the supremal controllable sub-language corresponding to this TDES is nonempty. If it is nonempty, then it contains all “safe” sequences of task execution which guarantee that the deadlines are met. (A precise definition of a “safe sequence” in the context of supervisory control of discrete-event systems requires an elaborate technical exposition. For the purpose of this article, it suffices to interpret it to mean a sequence that, when under the action of some events whose occurrence cannot be suppressed, does not cause the behavior of the system to deviate from its desired behavior.)

Our approach differs from the “conventional” real-time scheduling approaches in the following ways. First, in our approach, we do not treat checking for schedulability and finding a scheduling algorithm as two separate problems. The reason is that in the process of establishing the existence of a nonempty supremal controllable sublanguage for our TDES model, we also obtain (if the supremal controllable sublanguage is indeed nonempty) all safe sequences of task execution that guarantee timely completion of the tasks. This set of sequences is in fact a proper supervisor for the TDES.

Second, “conventional” scheduling approaches usually result in a scheduling algorithm which produces one sequence of task execution for a given set of tasks. In our approach, the nonempty supremal controllable sublanguage (if exists) is complete in the sense that it contains all safe task execution sequences which guarantee that all deadlines are met. This issue of completeness does not appear to have been addressed in the general literature on real-time scheduling.

The contribution of this work lies in the development of a formal constructive method for controlling the non-preemptive execution of periodic tasks on a processor. With this method, the problems of determining schedulability and finding a scheduling algorithm are treated as dual problems: a solution to the former necessarily implies a solution to the latter and vice versa. Furthermore, the solution to the latter thus obtained is complete.

The remainder of this article is organized as follows. Section 2 presents the background on scheduling of periodic tasks for non-preemptive execution on a single processor. Section 3 introduces the theory of supervisory control of timed discrete-event systems. Sections 4 and 5 describe the proposed method for modeling periodic tasks as timed discrete-
event systems and for synthesizing a supervisory control for non-preemptive execution of such tasks on a single processor. Section 6 discusses some implications of this work and computational complexity associated with the proposed method. Section 7 concludes this article.

2 Background

This section introduces the general problem of scheduling a set of periodic tasks on a processor for non-preemptive execution, and reviews previous results dealing with certain aspects of this problem.

2.1 Periodic Tasks with Hard Deadlines

In the following development, time is assumed to be discrete, and is incremented by a clock tick which is indexed by the natural numbers. Let $T$ be the set of discrete time instants, i.e., $T = \{0, 1, 2, 3, \ldots\}$.

A task is said to have been invoked when its existence is made known to the processor. The first instant at which a task is invoked is referred to as the release time of the task, and is denoted by $\varphi \in T$. Once released, a task is invoked repetitively forever. A task is said to be completed when it has been successfully executed by the processor.

For a given task, let $c \in T$ be the task execution time, and $p \in T$ be the task period. Let $t^o(k) \in T$ be the time of the $k^{th}$ invocation of a task, and $\bar{t} \in T$ be the time at which execution of the task is started. Execution of a task may be initiated only after the task has been invoked.

Sometimes a certain preparation is necessary for the processor to execute a task. Such preparation may involve “configuring” the processor in a certain way in order to execute a particular task [5]. We refer to the time required for “reconfiguring” the processor as the task setup time (denoted by $r \in T$). Note that reconfiguration (if necessary) is to take place after $\bar{t}$, i.e., the time instant when execution of the task is initiated.

A task is periodic with period $p \in T$ if $t^o(k+1) = t^o(k) + p$ and $c + r \leq p$. The deadline
of a task, denoted by \( d \in T \), is the time (measured from \( t^* \), the time of invocation) by which execution of the task must be completed. A task is said to have missed a deadline if \( t + r + c > t^* + d \).

Let \( A \) be a set of configurations the processor can assume, i.e., \( A = \{a_1, a_2, \ldots, a_m\} \). Let \( \theta \) denote a periodic task, i.e., \( \theta = (c, p, d, a, \varphi) \), where \( c \) is the task execution time, \( p \) is the task period, \( d \) is the deadline, \( a \in A \) is the task configuration, and \( \varphi \in T \) is the task release time. For a given task \( \theta \), it is assumed that the task execution time \( c \) and the task period \( p \) are known. The task setup time \( r \), however, is configuration-dependent. That is, for a given task \( \theta \), the task setup time \( r \) depends on the current configuration of the processor and the task configuration of \( \theta \). Let \( a_o \in A \) be the current configuration of the processor, and let \( a \in A \) be the task configuration of \( \theta \). Then the task setup time \( r \) for \( \theta \) is determined by \( r = \Omega(a_o, a) \), where \( \Omega(\cdot) : A \times A \rightarrow T \) is referred to as the reconfiguration cost function. Informally, \( \Omega(a_1, a_2) \) can be considered as a “look-up” table, which can be constructed by determining the time required for the transition between any two configurations \( a_1 \) and \( a_2 \). Obviously, \( \Omega(a, a) = 0 \).

Let \( \Gamma \) denote a set of \( n \) periodic tasks to be scheduled, i.e., \( \Gamma = \{\theta_1, \theta_2, \ldots, \theta_n\} \). A set of tasks \( \Gamma \) equipped with a transition cost function \( \Omega \) is referred to as a task pattern, and is denoted by \( \Lambda = (\Gamma, \Omega) \). The problem of scheduling a processor for non-preemptive execution of periodic tasks can be stated as follows: (i) determine the condition(s) under which a task pattern \( \Lambda \) is schedulable, and (ii) find an algorithm to schedule this task pattern.

In the subsequent development, we consider a set of \( n \) periodic tasks with the following characteristics:

1. The values for \( c_i \) and \( p_i \) are known and remain constant.

2. The deadlines of the tasks are prior or at the end of the respective task periods.

3. The task configurations and the release time of the tasks in \( \Lambda \) are the same, i.e., \( a_i = a_j \) and \( \varphi_i = \varphi_j = 0 \), for all \( i, j \in [1, n] \).
2.2 Previous Results

In the area of real-time scheduling, significant results have been established concerning preemptive execution of periodic tasks. Conditions for schedulability and universality of scheduling algorithms have been developed [11, 1]. Specifically, two fundamental results have been established. First, given a set of \( n \) tasks – each task with period \( p_i \) and execution time \( c_i \), this set of tasks is schedulable on a single processor if the utilization factor \( \sum_{i=1}^{n} \left( \frac{c_i}{p_i} \right) \) does not exceed unity. Second, the Earliest-Deadline-First (EDF) algorithm (which selects the task with the earliest deadline as the task to be executed next) is universal in the sense that if a set of tasks is schedulable, then the EDF algorithm will produce one schedule that meets all deadlines.

Compared to preemptive scheduling, reported results on non-preemptive scheduling appear to be less extensive. In [7, 8], the problem of non-preemptive scheduling is investigated in the context that tasks (with precedence order) are invoked only once and each task requires only a single unit of processing time. In [19], 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 [9], 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 the Earliest-Deadline-First (EDF) algorithm is examined.

3 Supervisory Control of Timed Discrete-Event Systems

The work reported in this article adopts the framework for supervisory control of timed discrete-event system originally proposed in [3]. This section summarizes the key concepts essential to the subsequent development.
3.1 Timed Discrete-Event Systems (TDES)

A discrete-event system (DES) is a dynamical process whose evolution is characterized by abrupt occurrence of events. In a logical DES, the system is characterized by a set of states and the transitions (or events) among these states. The behavior of the system is thus described by a sequence of events. The occurrence of an event, caused by some unmodelled mechanism internal or external to the system, instantaneously moves the system to a new state.

In a timed discrete-event system (TDES), the behavior of the system is also influenced by the temporal characteristics of the event sequence. Specifically, in a TDES, the occurrence of an event could be triggered either by some unmodelled mechanism acting on (or in) the system, or by the passage of time.

A TDES can be formally expressed as an automaton of the form

\[ G = (Q, \Sigma, \delta, q_0, Q_m), \]

where \( Q \) is a set of states, \( \Sigma \) is a (finite) set of events (including the tick event whose occurrence advances a global clock by one unit of time), the partial function \( \delta : Q \times \Sigma \to Q \) is a transition function which determines the new state of the system after the occurrence of an event, \( q_0 \) is the initial state of the system, and \( Q_m \) is a set of marker states which can be interpreted as the “tasks” to be completed by the system.

An event \( \alpha \) in a TDES is defined with two time bounds, i.e., \( (\alpha, l, u) \), where \( l \in T \) and \( u \in T \) are the lower and upper time bounds respectively. The interpretation for such a definition is that \( \alpha \) may occur (with reference to some point in time) after \( l \) ticks, and must occur after at most \( u \) ticks. An event is called prospective if \( 0 \leq l \leq u < \infty \), and remote if \( 0 \leq l < u = \infty \). In this article, we are concerned only with tasks having a finite hard deadline. Therefore, all timed events considered in the sequel are prospective unless stated otherwise.

Each event is thus associated with a timer interval \( T_\sigma = [0, u] \). A state \( q \) of a TDES consists of an activity \( b \in B \) (where \( B \) is a set of logical states without any reference to time and is referred to as an activity set) and a timer \( t_\sigma \in T_\sigma \), i.e., \( q = (b, \{t_\sigma : \sigma \in \Sigma_B\}) \),
where $\Sigma_B = \Sigma - \{\text{tick}\}$. The default value for $t_\sigma$ is $u$.

The activity set $B$ is equipped with an activity transition (partial) function $\delta_B : B \times \Sigma_B \to B$. The TDES transition function $\delta : Q \times \Sigma \to Q$ for $G$ is defined in terms of $\delta_B$ and the time bounds, according to detailed rules given in [3]. An event $\sigma \in \Sigma_B$ is enabled at $q = (b, \cdot) \in Q$ if $\delta_B(b, \sigma)$ is defined (written as $\delta_B(b, \sigma)!$), and is disabled otherwise. An enabled event $\sigma$ (either tick or in $B$) is eligible if $\delta(q, \sigma)!$. Only eligible events can actually occur.

The timing mechanism of a TDES [3] can be summarized informally as follows. Once an event $\alpha$ is enabled, the timer $t_\alpha$ of $\alpha$ is decremented by one time unit at every subsequent tick of a global clock, until (i) $t_\alpha$ reaches zero (at which point $\alpha$ is forced to occur), or (ii) $\alpha$ occurs, or (iii) $\alpha$ is disabled due to the occurrence of some other transition. In all of these cases, $t_\alpha$ is reset to its default value $u$.

### 3.2 Supervisory Control

Supervisory control of a TDES in essence means timely disablement of certain events in the transition structure of the TDES, such that its behavior meets certain specifications. The theories of automata and formal language form the basis for synthesis of supervisory control of TDES.

The language (i.e., the sequences of event labels, or strings) generated by $G$, denoted by $L(G)$ is called the closed behavior of $G$, i.e., $L(G) = \{ s | s \in \Sigma^*, \delta(q_0, s)! \}$, where $\Sigma^*$ contains, in addition to the empty string $\epsilon$, all possible finite sequences over $\Sigma$. In other words, $L(G)$ describes all possible transition sequences within the TDES modelled by $G$. Of all possible sequences, some may be considered as the steps required for completing certain tasks. Such sequences are called the marked behavior of $G$, denoted by $L_m(G)$, i.e., $L_m(G) = \{ s | s \in \Sigma^*, \delta(q_0, s) \in Q_m \}$. By definition, $L_m(G) \subseteq L(G)$.

A string $t \in \Sigma^*$ is called a prefix of another string $s \in \Sigma^*$, expressed as $t \preceq s$, if $s = tu$ for some $u \in \Sigma^*$. The prefix closure of a language $L \subseteq \Sigma^*$, denoted by $\bar{L}$, is defined as $\bar{L} = \{ t \in \Sigma^* | t \preceq s, s \in L \}$. A language is prefix-closed if $L = \bar{L}$.

The notion of supervisory control of a TDES is based on the concept of controllability.
Controllability is defined in the context that the set of events in a TDES is partitioned into controllable events and uncontrollable events, i.e., $\Sigma_B = \Sigma_c \cup \Sigma_u$, where $\Sigma_c$ is the set of controllable events and $\Sigma_u$ is the set of uncontrollable events. An event is controllable if it can be prevented from occurring, and is uncontrollable otherwise. More technical definitions of controllable and uncontrollable events in the context of TDES can be found in [3]. For the purpose of this article and economy of presentation, the above intuitive definitions are sufficient.

In a hard real-time system, an event must occur at the latest at the deadline. Supervision of a TDES then must include a mechanism for forcing certain events to occur before a specific time instant. For this reason, an additional type of event, called forcible event, is defined to describe events that can preempt a tick of the global clock.

The transition structure of a TDES can be visualized as a graph, with the states of the TDES being the nodes, and the events being the edges. At any particular node, there is at least one eligible event, namely tick, if no enabled prospective event has timed out. We define a set of (extended) eligible events $\mathcal{H}(s)$ with respect to $G$ as $\mathcal{H}_G(s) = \{ \sigma \in \Sigma \mid s\sigma \in L(G) \}$. Given $G$ over a certain $\Sigma$, and a language $K \subseteq L(G)$ with the eligible events $\mathcal{H}_K(s) = \{ \sigma \in \Sigma \mid s\sigma \in K, s \in \Sigma^* \}$, we say that $K$ is controllable with respect to $G$ if, for all $s \in K$,

$$\mathcal{H}_K(s) \supseteq \begin{cases} H_G(s) \cap \Sigma_u & \text{if } H_K(s) \cap \Sigma_f = \emptyset, \\ H_G(s) \cap \Sigma_u & \text{if } H_K(s) \cap \Sigma_f \neq \emptyset, \end{cases}$$

where $\Sigma_u = \Sigma_u \cup \{\text{tick}\}$, and $\Sigma_f$ are the forcible events. This formal definition of controllability can be interpreted intuitively as follows. We can consider a TDES as a generator of a set of "legal" sequences, i.e., sequences that are admitted by the transition structure of the TDES. Thus a TDES is controllable with respect to $G$ if it always admits the occurrence of uncontrollable events eligible in $G$, and, if tick is eligible in $G$, admits tick as well, unless an eligible forcible event is available to preempt it.

A supervisor of a TDES $G$ can be considered as an automaton $V$ that monitors the state of $G$, and enables or disables certain events in $G$ when necessary so as to influence
the behavior of $G$. Formally, a supervisor is a map $V : L(G) \to 2^\Sigma$ such that, for all $s \in L(G)$,

$$V(s) \supseteq \begin{cases} 
\Sigma_u \cup (\{\text{tick}\} \cap H_G(s)) & \text{if } V(s) \cap H_G(s) \cap \Sigma_f = \emptyset, \\
\Sigma_u & \text{if } V(s) \cap H_G(s) \cap \Sigma_f \neq \emptyset.
\end{cases}$$

The closed behavior of $G$ under the supervision of $V$, denoted by $L(V/G)$, is defined according to (i) $\epsilon \in L(V/G)$, (ii) if $s \in L(V/G)$, $\sigma \in V(s)$, and $s\sigma \in L(G)$, then $s\sigma \in L(V/G)$, and (iii) no other strings belong to $L(V/G)$. The marked behavior of $G$ under $V$ is $L_m(V/G) = L(V/G) \cap L_m(G)$. A supervisor $V$ is nonblocking for $G$ if $L_m(V/G) = L(V/G)$.

The problem of supervisory control synthesis can be summarized as follows. Given a TDES $G$ over a certain $\Sigma$, and some non-empty languages $W_1$ and $W_2$ with $W_1 \subseteq W_2 \subseteq L(G)$, find a nonblocking supervisor $V$ such that $W_1 \subseteq L(V/G) \subseteq W_2$. (Conditions for and proof of existence of supervisory control for a given TDES are discussed in detail in [3]. An algorithm for computing $V$ has also been developed in [16].) Solution to this problem depends on the controllability of $W_2$. An important point to emphasize here is that, if $W_2$ is not controllable, then a largest (i.e., supremal) controllable sublanguage of $W_2$, denoted by $\sup \mathcal{C}(W_2, G)$, can always be found (even though it may be empty). The supervisor $V$ is said to be “minimally restrictive” in the sense that its only action is to disable certain events when necessary so as to preserve the desired behavior of the system. As a consequence, the solution generates the largest possible subset of legal sequences.

A software package called $TTCT$ has been developed by the Systems Control Group at the University of Toronto. Functionalities of $TTCT$ include creating new TDES, composition of two or more TDES, and computing the supremal controllable sublanguage of a given language.
4 Modeling of Non-preemptive Task Execution

In this section, it is shown that a periodic task can be modelled as a TDES, and that a set of individual tasks can be combined into a “composite” TDES. Supervisory control synthesis is then discussed, in conjunction with the development of the specification to account for non-preemptive execution of the tasks. Two examples are presented to illustrate the overall method.

4.1 TDES Model of a Task Execution

Let $\alpha_i$ denote the event “execution of task $\theta_i$ starts”, $\beta_i$ denote the event “execution of task $\theta_i$ finishes”, and $\gamma_i$ denote the event “task $\theta_i$ is invoked”. Let $E_i$ denote the state of the processor executing task $\theta_i$, $I_i$ denote the idle state, and $W_i$ denote a “delay” state. Note that $W_i$ differs from $I_i$ in that $W_i$ is a pure delay while at $I_i$ the event $\alpha_i$ (which has material effect on the system) is allowed to occur. The event $\alpha_i$ is controllable, while the events $\beta_i$ and $\gamma_i$ are uncontrollable; all events are forcible.

Invocation of a task $\theta_i$ corresponds to the enabling of the event $\alpha_i$. Once enabled, an event may occur in the period between the lower time bound and the upper time bound, and can be forced to occur when the upper time bound is reached.

We introduce the TDES model of a processor executing a task $\theta_i$ as an automaton

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

where $Q = \{I_i, E_i, W_i\}$, $\Sigma = \{\alpha_i, \beta_i, \gamma_i, tick\}$, $q_0 = I_i$, $Q_m = \{I_i\}$, and $\delta$ is defined by the following transitions: $E_i = \delta(I_i, \alpha_i)$, $W_i = \delta(E_i, \beta_i)$, $I_i = \delta(W_i, \gamma_i)$, and $(\cdot) = \delta((\cdot), tick)$.

The activity transition graph (ATG) of this TDES is shown in Figure 1. An ATG describes only the logical behavior of a TDES. The marked state is indicated by the shaded circle.
Recall that a task $\theta_i$ is expressed in the following notation: $\theta_i = (c_i, p_i, d_i, a_i, \varphi_i)$, where $c_i$ is the task execution time, $p_i$ is the task period, $d_i$ is the deadline, $a_i$ is the configuration under which the task can be executed, and $\varphi_i$ is the task release time. The temporal behavior of a task is characterized by the following timing constraints:

1. Once enabled, the event $\alpha_i$ must occur in the period between 0 and $(d_i - c_i)$ ticks, i.e., $(\alpha, 0, d_i - c_i)$. This is to ensure that there is sufficient time for the execution of the task so that the deadline can be met,

2. The event $\beta_i$ must occur exactly $c_i$ ticks after the event $\alpha_i$ has occurred, i.e., $(\beta_i, c_i, c_i)$. This is to enforce the requirement that the task $\theta_i$ takes $c_i$ ticks to execute, and

3. The event $\gamma_i$ must occur exactly every $p_i$ ticks, i.e., $(\gamma_i, p_i, p_i)$, with reference to a global clock. This is to reflect the periodic invocation of a task.

To represent the temporal behavior, we introduce a convention for drawing the timed transition graph (TTG), as illustrated in Figure 2.

![Timed Transition Diagram](image)

**Figure 2. Timed Transition.**

In Figure 2, the self-loop with the label $t(l, u)$ intersects the edge with the event label $\alpha$. This is to indicate that the event $\alpha$ may occur after $l$ ticks, and must occur after at most $u$ ticks.
The temporal behavior of a task execution can now be expressed in the TTG shown in Figure 3, where $t_\alpha$ is the time instant when $\alpha_i$ occurs.

![TTG of a Periodic Task](image)

$* = t(0, 0)$ if the task is to be invoked for the very first time; otherwise $* = t(p_i - t_\alpha - c_i, p_i - t_\alpha - c_i)$

Figure 3. TTG of a Periodic Task.

**Remark.** We refer to the graph shown in Figure 3 (and the subsequent detailed version of such a graph) as a timed transition graph (TTG) in the sense that it describes the temporal characteristics of a task execution. In a formal framework, one may need to impose the condition that a TTG must be an entity that can be derived from its corresponding ATG [12]. The TTG's discussed in this article, however, may not satisfy this condition, because compliance to this condition would result in a TTG that contains incorrect information about the release time of the task represented by the TTG. □

The process depicted in Figure 3 can be described as follows. When the system enters the initial Wait state $W_i$, the task $\theta_i$ is immediately invoked by the event $\gamma_i$ (i.e., $\alpha_i$ is enabled), thus bringing the system to the Idle state $I_i$. Execution of the task may start between the next 0 to $(d_i - c_i)$ ticks, and must occur (if not already) after the $(d_i - c_i)^{th}$ tick. Occurrence of $\alpha_i$ (at time $t_\alpha$) moves the system to the Execution state $E_i$. Exactly $c_i$ ticks thereafter, the event $\beta_i$ occurs, which brings the system to the Wait state $W_i$. The system remains in the Wait state for exactly $(p_i - t_\alpha - c_i)$ ticks (at which point the task period ends), and then makes the transition $\gamma_i$ to the Idle state $I_i$ (where $\alpha_i$ is enabled again) or exits to other states.

The TTG in Figure 3 is a compact representation of the task execution. It captures the essence of the execution of a periodic task in the framework of timed discrete event systems. To synthesize a supervisor for controlling the execution process, a more detailed
representation is required. The construction of such a detailed representation involves identifying all possible timed sequences by which a task can be executed to meet the deadline. To do so, each timed self loop in the TTG as shown in Figure 3 must be “unfolded” to reveal the detailed temporal behavior of the task execution.

Figure 4 shows the detailed TTG of a general task $\theta_i$.

Figure 4. Detailed TTG of a General Periodic Task.

In Figure 4, each trace starting from the state where the task is invoked (i.e., the marked state) and ending at the same state is one execution sequence that meets the deadline of the task. For instance, one such sequence is

$$
\gamma_i \quad \alpha_i \quad t \ldots t \quad \beta_i \quad t \ldots t
$$

As a specific example, Figure 5 shows the TTG of a task with an execution time of 1 tick, a period of 6 ticks, and a deadline equals to the period.
4.2 Task Composition

The TTG as shown in Figure 5 represents a TDES model of the execution of a task. For a set of tasks to be executed on a processor, the TDES representing the execution of this set of tasks is required.

![TTG of a Task](image)

Figure 5. TTG of a Task with $c = 1$, $p = 6$, and $d = 6$.

One way to construct such a “composite” TDES for a set of tasks is to apply the synchronous product [18] over the individual TDES. Given two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ with $\Sigma = \Sigma_1 \cup \Sigma_2$ and the natural projection $P_i : \Sigma^* \rightarrow \Sigma_i^*$ defined by: (i) $P_i(\varepsilon) = \varepsilon$, (ii) $P_i(\sigma) = \varepsilon$ if $\sigma \notin \Sigma_i$, (iii) $P_i(\sigma) = \sigma$ if $\sigma \in \Sigma_i$, and (iv) $P_i(s\sigma) = P_i(s)P_i(\sigma)$, $s \in \Sigma^*$ and $\sigma \in \sigma^*$, the synchronous product of $L_1$ and $L_2$, denoted by $L_1 || L_2$, is defined as $L_1 || L_2 = P_1^{-1}L_1 \cap P_2^{-1}L_2$. In the TTCT software, the synchronous product is computed using the procedure `sync`.

In essence, the synchronous product of two TDES (namely TDES1 and TDES2) is a new TDES (namely TDES3), which is constructed by synchronizing the events possessed in common by TDES1 and TDES2. Since execution of a task on a single processor is
characterized by a distinct triplet \( \{\alpha_i, \beta_i, \gamma_i\} \), the only shared event between any two tasks is the tick event. Thus two TDES, namely \textbf{TASK1} and \textbf{TASK2}, representing executions of two tasks can be combined into a single TDES, namely \textbf{TASKSET}, by application of the procedure \texttt{sync} as follows

\[
\textbf{TASKSET} = \text{sync}(\textbf{TASK1}, \textbf{TASK2}).
\]

For a set of \( n \) tasks, the composite model of task execution can be constructed by repeated applications of the synchronous product on the individual TDES models of task execution, i.e.,

\[
\begin{align*}
T_1 &= \text{sync}(\textbf{TASK1}, \textbf{TASK2}), \\
T_2 &= \text{sync}(T_1, \textbf{TASK3}), \\
&\vdots \\
\textbf{TASKSET} &= \text{sync}(T_{n-1}, \textbf{TASKn}).
\end{align*}
\]

The TDES \textbf{TASKSET} resulted from the \texttt{sync} operation contains all execution sequences that meet the deadlines of the tasks, but with the implicit assumption that resources are available to execute the tasks concurrently. This result is illustrated in the following example.

Figures 6 and 7 depict the TTG of two individual tasks. For the first task, \textbf{TASK.A}, \( c_a = p_a = d_a = 2 \). For the second task, \textbf{TASK.B}, \( c_b = p_b = d_b = 1 \).

![Figure 6. TTG of \textbf{TASK.A.}](image)
The composite task execution model is

\[ \text{TASKSET.A} = \text{sync}(\text{TASK.A}, \text{TASK.B}), \]

which has 20 states and 29 transitions, and is illustrated in Figure 8.

To see why concurrent task execution is implicitly assumed in this TTG, we isolate one cyclic sequence:

\[ \gamma_1 \alpha_1 \gamma_2 \alpha_2 \beta_2 \gamma_2 \alpha_2 \beta_2 \beta_1. \]

In this sequence, TASK.A is first invoked \((\gamma_1)\), and execution is initiated \((\alpha_1)\). TASK.B is then invoked \((\gamma_2)\), and execution is initiated \((\alpha_2)\) immediately. After one tick, execution of TASK.B terminates \((\beta_2)\), and TASK.B is invoked and executed again \((\gamma_2 \alpha_2)\). After the second tick, execution of both TASK.B and TASK.A are completed \((\beta_2 \beta_1)\). Note that for the duration of two ticks, a total of one TASK.A and two TASK.B are executed.
Since TASK_A alone requires two ticks to execute, the sequence above implicitly assumes that resources are available to execute the two tasks concurrently.

5 Supervisory Control Synthesis

To account for the fact that only a single processor with fixed computational resources is available, and that task execution is non-preemptive, we shall impose specifications on the behavior of the composite TDES. We then synthesize a supervisory control that forces the behavior of the TDES to conform to the specifications. In the framework of supervisory control of discrete-event systems, such a synthesis is achieved by finding the supremal controllable sublanguage of the TDES. A detailed discussion on this topic can be found in [16]. For the following analysis, we apply the software implementation of the solution algorithm discussed in [16], namely the procedure supcon of TTCT.

In general terms, given a TDES TDES1 and a specification expressed in another TDES TDES2, the procedure supcon returns a new TDES TDES3. TDES3 is called a proper supervisor of TDES1 in the sense that under the action of TDES3, the behavior of TDES1 conforms to the constraints imposed by TDES2. If the TDES3 returned by the procedure supcon is empty, then it can be concluded that no supervisory control exists for TDES1 to conform to TDES2.

The specification for enforcing non-preemptive task execution is that, once started, execution of a task must be completed before another execution is initiated. Note that enforcing non-preemptive execution necessarily implies that the processing power of a processor is fixed.

The specification for non-preemptive execution of one task θ_i is illustrated in Figure 9. It imposes the constraint that only γ and tick events are allowed in any string starting with α_i and ending with β_i. This ensures that if the processor is currently executing a task, it cannot start executing another task. However, invocation of tasks during task execution is permitted, as is indicated by the self-loop on the left.
Figure 9. SPECi: Specification for Non-preemptive Execution of a Task.

Analogous to composition of task execution models using the sync procedure, the “composite” specification of a set of tasks can be built using the meet procedure [18] of TTCT, i.e.,

\[ S_1 = \text{meet}(\text{SPEC}_1, \text{SPEC}_2), \]
\[ S_2 = \text{meet}(S_1, \text{SPEC}_3), \]
\[ \ldots \]
\[ \text{SPECSET} = \text{meet}(S_{n-1}, \text{SPEC}_n). \]

In fact, the meet operation is the special case of sync corresponding to the case where \( \Sigma_1 = \Sigma_2 \), that is, all events are considered shared and synchronization is total.

The supremal controllable sublanguage corresponding to a set of tasks TASKSET with the specification SPECSET can now be computed with

\[ \text{SUPERVISOR} = \text{supcon}(\text{TASKSET}, \text{SPECSET}). \]

To determine whether a set of tasks can be scheduled such that all deadlines of the tasks are met, we check whether the supremal controllable sublanguage (namely, the SUPERVISOR) corresponding to TASKSET with the specification SPECSET is nonempty. If SUPERVISOR is nonempty, then it contains all safe execution sequences which guarantee that the deadlines for the tasks are met. If, on the other hand, SUPERVISOR is empty, then it can be concluded that no sequence exists that guarantees the on-time execution of the tasks. In other words, the tasks are not schedulable.
5.1 Examples

5.1.1 Example 1: Schedulable Tasks

Consider the following two tasks:

\[
\begin{align*}
\text{TASK1} & : \theta_1 = (1, 6, 6, 1, 0), \\
\text{TASK2} & : \theta_2 = (2, 3, 3, 1, 0).
\end{align*}
\]

Note that the tasks are released at the same instant, and both can be executed in the same configuration. Since \(c_1 = 1, c_2 = 2, p_1 = 6, p_2 = 3, d_1 = 6,\) and \(d_2 = 3,\) the corresponding timed events are \((\alpha_1, 0, 5), (\beta_1, 1, 1), (\gamma_1, 6, 6), (\alpha_2, 0, 1), (\beta_2, 2, 2),\) and \((\gamma_2, 3, 3).\) Note that the first occurrence of \(\gamma_1\) and \(\gamma_2\) takes place immediately after the tasks are released. The time bounds above for these two events only apply to the subsequent occurrences.

The TTG for executing \(\theta_1\) is shown in Figure 10. It contains 34 states and 39 transitions.

![TTG of TASK1](image)

**Figure 10.** TTG of TASK1.
The TTG for executing \( \theta_2 \) is shown in Figure 11. It contains 10 states and 11 transitions.

![Figure 11. TTG of TASK2.](image)

To model the execution of these two tasks by a single processor, we combine the two TDES into a single TDES by taking the synchronous product, i.e.,

\[
\text{TASKSET1} - \text{sync} (\text{TASK1}, \text{TASK2}).
\]

This composite TDES has 112 states and 150 transitions.

As noted earlier, the specification for enforcing non preemptive execution is that, once started, execution of a task must be completed before another execution is initiated. The specifications for non-preemptive execution of \text{TASK1} and \text{TASK2} are illustrated in Figures 12 and 13 respectively.

![Figure 12. SPEC1: Specification for Non-preemptive Execution of TASK1.](image)

![Figure 13. SPEC2: Specification for Non-preemptive Execution of TASK2.](image)
The composite specification, $\text{SPECSET1}$, is obtained by

$$\text{SPECSET1} = \text{meet}(\text{SPEC1, SPEC2})$$

which has 3 states and 13 transitions, and is illustrated in Figure 14.

![Figure 14. SPECSET1: Specification for Non-preemptive Execution of TASKSET1.](image)

The supremal controllable sublanguage of $\text{TASKSET1}$ with $\text{SPECSET1}$ is computed with

$$\text{SUPERVISOR1} = \text{supcon}(\text{TASKSET1, SPECSET1}).$$

The resulting $\text{SUPERVISOR1}$ is nonempty; it has 58 states and 71 transitions. It contains all safe sequences of task execution with the guarantee that no deadline is missed. The TTG of $\text{SUPERVISOR1}$ is shown in Figure 15.

![Figure 15. Task Execution Sequences.](image)
By tracing through the graph, it can be verified that (i) one $\gamma_1$ occurs exactly every 6 ticks, (ii) one $\gamma_2$ occurs exactly every 3 ticks, (iii) one pair of $\alpha_1$ and $\beta_1$ occurs every 6 ticks, and (iv) one pair of $\alpha_2$ and $\beta_2$ occurs every 3 ticks. These are in fact the timing requirements of the tasks as specified earlier.

To express the above results in a form more appealing to one’s intuition, we project out all events except $\alpha_1$ and $\alpha_2$ in the TTG as shown in Figure 15 to obtain the simplified ATG as shown in Figure 16. We will refer to the cyclic sequences for executing the two tasks in this simplified ATG as logical sequences, i.e., sequences that indicate the order of task execution without reference to time.

![Figure 16. Simplified ATG of Task Execution Sequences for TASKSET1.](image)

The ATG depicted in Figure 16 contains the following three distinct cyclic logical sequences: $s_1 = \alpha_1\alpha_2\alpha_2$, $s_2 = \alpha_2\alpha_1\alpha_2$, and $s_3 = \alpha_2\alpha_2\alpha_1$. Note that since 5 ticks are required for executing the two tasks in any one 6-tick interval, all timed execution sequences can be recovered from the three logical sequences above by inserting a tick into these sequences.

For practical implementation, one can simply choose a sequence such as $s_1$ and execute it once every 6 ticks. Ignoring the inserted tick as noted above, a logical execution sequence, denoted by $S_E$, might be $S_E = s_1s_1\ldots s_1\ldots$. It is possible for a logical execution sequence to contain $s_1$, $s_2$ and $s_3$, i.e., $S_E = \Sigma^+$, where $\Sigma^+$ contains all possible sequences over $\Sigma_s = \{s_1, s_2, s_3\}$.

Now suppose that the deadline of TASK2 is changed to $d = 2$, i.e., execution of TASK2 must be completed no later than at the second tick after invocation. We will refer to this modified task as TASK2a. The TTG of TASK2a is obtained by removing from the TTG of TASK2 (as shown in Figure 11) the sequence $\gamma_2\alpha_2\alpha_2\beta_2$ (which does not satisfy this new timing requirement). The TTG of TASK2a is depicted in Figure 17; it has 6 states and 6 transitions.
Figure 17. TTG of **TASK2a**.

The composite model of task execution is now

\[
\text{TASKSET1a} = \text{sync(TASK1, TASK2a)}.
\]

This composite TDES has 67 states and 86 transitions. The specification for non-preemptive task execution is the same as that shown in Figure 14 (with the exception that all events of **TASK2** are replaced by that of **TASK2a**, i.e., \((\cdot)_2\) becomes \((\cdot)_{2a}\)). We will refer to this specification as **SPECSET1a**.

The corresponding supremal controllable sublanguage is computed with

\[
\text{SUPERVISOR1a} = \text{supcon(TASKSET1a, SPECSET1a)}.
\]

The resulting **SUPERVISOR1a** is nonempty; it has 27 states and 32 transitions. To obtain a simplified ATG of **SUPERVISOR1a**, we project out all but the \(\alpha\) events. The result is shown in Figure 18.

Figure 18. Simplified ATG of Task Execution Sequences for **TASKSET1a**.

The ATG depicted in Figure 18 contains the following two cyclic logical sequences: \(\alpha_{2a}\alpha_1\alpha_{2a}\) and \(\alpha_{2a}\alpha_{2a}\alpha_1\). It can be seen, by comparing Figure 16 with Figure 18, that moving the deadline of **TASK2** ahead by 1 tick invalidates the cyclic logical sequence \(\alpha_1\alpha_2\alpha_2\), since **TASK2a** must be executed immediately upon its initial release and subsequent invocation.
5.1.2 Example 2: Non-Schedulable Tasks

In this example, we consider a task set consisting of the following two tasks:

\[
\text{TASK3} : \quad \theta_3 = (3, 6, 6, 1, 0), \\
\text{TASK4} : \quad \theta_4 = (1, 2, 2, 1, 0).
\]

For this simple case, it can be verified by inspection that this task set is not schedulable, because in a given duration of 6 ticks, 3 ticks must be reserved to execute \( \theta_3 \), while \( \theta_4 \) is invoked every 2 ticks and requires 1 tick to execute. The TTG for executing \( \theta_3 \) is shown in Figure 19. It contains 27 states and 30 transitions.

![Figure 19. TTG of TASK3.](image)

The TTG for executing \( \theta_4 \) is shown in Figure 20. It contains 8 states and 9 transitions.

![Figure 20. TTG of TASK4.](image)
The composite model of executing these two tasks is constructed by synchronizing the tick events in TASK3 and TASK4, i.e.,

\[ \text{TASKSET2} = \text{sync(TASK3, TASK4)}. \]

\text{TASKSET2} consists of 108 states and 145 transitions.

The specification for enforcing non-preemptive execution of these two tasks is the same (except the event labels) as that for the previous example, i.e.,

\[ t, \gamma_3, \gamma_4 \quad \alpha_3 \quad t, \gamma_3, \gamma_4 \quad \alpha_4 \quad t, \gamma_3, \gamma_4 \quad \beta_3 \quad \beta_4 \]

Figure 21. SPECSET2: Specification for Non-preemptive Execution of TASKSET2.

The supremal controllable sublanguage of TASKSET2 with SPECSET2 is computed with

\[ \text{SUPERVISOR2} = \text{supcon(TASKSET2, SPECSET2)}. \]

The SUPERVISOR2 produced by the supcon procedure is empty, indicating that there exists no supervisory control that guarantees on-time execution of the tasks. This result is consistent with the fact that these two tasks are not schedulable for non-preemptive task execution, as discussed earlier.

Note that, if preemption were allowed, these two tasks would be schedulable, since the utilization factor, \[ \sum_{i=1}^{2} \left( \frac{\alpha_i}{P_i} \right) = \left( \frac{3}{6} + \frac{1}{2} \right) = 1, \] does not exceed unity.

6 Discussion

In this section, we discuss some aspects of the proposed scheduling method to provide a perspective on its results and practicality. Specifically, we compare the proposed method
with an existing algorithm, namely the Earliest-Deadline-First (EDF) algorithm. We then examine the possibility provided by our proposed method for further optimization, and discuss the computational complexity of the proposed method.

6.1 Comparison with the EDF Algorithm

The “classical” EDF algorithm was originally developed for non-preemptive scheduling of non-periodic tasks, mainly in the context of job sequencing in production planning. It has become a popular algorithm for scheduling of non-preemptive tasks, due to its simplicity and low implementation overhead.

For a given set of tasks $\Gamma = \{\theta_1, \theta_2, \ldots, \theta_n\}$ and a set $\hat{\Gamma} \subseteq \Gamma$ containing the invoked outstanding tasks, the EDF algorithm selects from $\hat{\Gamma}$ the task with the earliest deadline as the task to be executed next. A schedule can thus be generated “off-line” and then checked to ensure that it guarantees that the deadline of each task is met.

For the example as presented in Section 4.4.1 with the tasks

\[
\text{TASK1 : } \theta_1 = (1, 6, 6, 1, 0), \\
\text{TASK2 : } \theta_2 = (2, 3, 3, 1, 0),
\]

the EDF algorithm will produce a schedule as follows. Initially both tasks are released simultaneously, i.e., $\theta_1$ and $\theta_2$ are invoked at the same instant. Since $\theta_2$ has an earlier deadline, $\theta_2$ is selected as the task to be executed.

After two ticks, the execution of $\theta_2$ terminates. Since the next invocation of $\theta_2$ is still one tick away and $\theta_1$ is now the only invoked outstanding task, $\theta_1$ is selected to be executed. The execution of $\theta_1$ terminates after one tick, at which time $\theta_2$ is invoked. The newly invoked $\theta_2$ is now the only outstanding task, and its execution is initiated immediately. This execution terminates after two ticks. At this point, there is no invoked task outstanding so the processor remains idle for another tick, after which $\theta_2$ is again invoked. The above event sequence, i.e., $\alpha_2tt\beta_2\alpha_1t\beta_1\alpha_2tt\beta_2t$, then repeats.

Figure 22 illustrates two complete cycles of this sequence, where $d_i$ indicates the dead-
line of task $\theta_i$.

![Figure 22. Scheduling using the EDF Algorithm.](image)

For a set of $n$ tasks, let $w_i$ denotes the time (measured from the time of task invocation) at which a task $\theta_i$ is completed, then the “lateness” of a task (denoted by $l_i$) can be defined as

$$l_i = w_i - d_i,$$

where $d_i$ is the deadline of the task. A schedule is guaranteed to meet the individual deadlines of the task set $\Gamma$ if

$$L \equiv \max[l_i] \leq 0, \quad \forall i \in [1, n].$$

For the cyclic sequence produced by the EDF algorithm in the above example, i.e., $\alpha_2t\beta_2\alpha_1t\beta_1\alpha_2t\beta_2t$, we have $L = -1 < 0$ for one complete cycle of 6 ticks.

This sequence is in fact one of the safe sequences shown in Figure 15, which contains all safe sequences of execution for the task set $\{\theta_1, \theta_2\}$. As was discussed in Section 4.1.1, the logical sequences associated with Figure 15 were: $s_1 = \alpha_1\alpha_2\alpha_2$, $s_2 = \alpha_2\alpha_1\alpha_2$, and $s_3 = \alpha_2\alpha_2\alpha_1$. When the tick event is taken into account, the logical sequence $s_2$ in fact generates four distinct timed execution sequences, i.e., $\alpha_2tt\beta_2\alpha_1t\beta_1\alpha_2tt\beta_2t$, $\alpha_2tt\beta_2t\alpha_1t\beta_1t\alpha_2tt\beta_2$, $\alpha_2tt\beta_2\alpha_1t\beta_1\alpha_2tt\beta_2$, and $t\alpha_2tt\beta_2\alpha_1t\beta_1\alpha_2tt\beta_2$. Of these four, the first is identical to the sequence produced by the EDF algorithm.

It can be seen that while our proposed method results in a minimally restrictive supervisor (i.e., it contains all safe sequences of task execution), the results produced by the EDF algorithm does not have such a characteristic. One immediate benefit of having
a minimally restrictive supervisor is that it offers the possibility for optimization among this set of safe sequences.

6.2 Optimization

The minimally restrictive behavior of the supervisor is represented by the complete set of safe sequences, as discussed in Section 4. Any one of these sequences meets the deadlines of the task set.

One advantage of having such a complete solution is that further optimization is possible. In other words, we can select one schedule, among all safe sequences, based on some additional criterion. One such criterion, for example, may be that a task should be completed as close to its deadline as possible. This “deadline-matching” property of a schedule ensures that tasks are completed “just-in-time”. Scheduling with deadline-matching optimization would have important application in the management of manufacturing operations, where a production unit (e.g., a workcell) can be controlled such that parts are produced just-in-time in order to minimized inventory (or temporary storage) of finished parts.

The deadline-matching optimization problem can be stated as follows. For a set of \( n \) tasks, the collective deviation of the task completion time from the deadlines is to be minimized, i.e.,

\[
\min \left( \sum_{i=1}^{n} (d_i - w_i) \right).
\]

According to this criterion, the optimal cyclic sequence (with the \( \gamma \) events omitted) for the example discussed in Section 4.1.1 is a timed version of \( s_3 \), i.e., \( t\alpha_2t\beta_2\alpha_2tt\beta_2\alpha_1t\beta_1 \) or \( \alpha_2tt\beta_2t\alpha_2tt\beta_2\alpha_1t\beta_1 \). The collective deviation of the completion time from the deadlines for either of these two sequences is 1 tick.

6.3 Computational Complexity

The computational complexity of the method presented in this article is characterized by (i) the modelling of clock increment as a distinct event in the TDES framework as
proposed in [3], and (ii) the exponential growth in the number of states when combining
individual tasks into a single TDES using the synchronous product.

The TDES framework adopted in this work models time by explicitly adding the tick
event to represent the increment of the clock by one time unit. If a task has a large period
as compared to the time resolution of the clock, then a significant number of transitions
(and states) will be generated simply due to the passage of time. One possible approach
to avoid this difficulty is to investigate the potential application of the theory of max-
algebra for modelling of discrete-event systems [2], so that passage of time is not modelled
explicitly [6].

It has been shown that the computation of the supremal controllable sublanguage with
respect to a finite TDES can be completed in polynomial time [14]. The complexity in the
synthesis of a supervisory control stems from the fact that, with the synchronous product,
the number of states of a composite TDES increases exponentially with the number of
components. This could represent a fundamental barrier to “scaling up” of the proposed
method for real-time scheduling of large task sets. Modular synthesis approaches (e.g.,
[17]), therefore, should be explored to circumvent this difficulty.

7 Concluding Remarks

In this article, we have presented a formal constructive method for scheduling the non-
preemptive execution of a set of periodic tasks on a processor. With this method, the
problems of determining schedulability and finding a scheduling algorithm are treated as
dual problems: a solution to the former necessarily implies a solution for the latter and
vice versa. Furthermore, the solution to the latter thus obtained is complete in the sense
that it contains all safe sequences of task execution with the guarantee that no deadline
is missed.

We have shown that the theory of supervisory control of discrete-event systems can be
applied to the scheduling of hard real-time systems. Such a formal theory offers advantages
in terms of a systematic approach to problem solving and completeness of its solution; it
thus represents a rigorous tool for analysis and synthesis of real-time systems.
In our formulation of the scheduling problem, both the processor and tasks are taken to be generic entities. For instance, a processor could be a CNC machine, a robot, or an assembly line worker, while a task could be a manual assembly process or a robotic pick-and-place operation. This leads to the possibility that the method presented in this article can be readily applied in more practical context, such as in “hard real-time manufacturing” [5], where the objective is to manage a manufacturing process so as to ensure periodic on-time delivery of various products.

References


