Formalization of Uniprocessor and Multiprocessor Scheduling of Real-Time Systems Using Supervisory Control of Discrete-Event Systems

Vasudevan Janarthanan

A Thesis

In

The Department

Of

Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
at Concordia University
Montréal, Québec, Canada

©Vasudevan Janarthanan, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-30136-4
Our file Notre référence
ISBN: 978-0-494-30136-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Formalization of Uniprocessor and Multiprocessor Scheduling of Real-Time Systems Using Supervisory Control of Discrete-Event Systems

Vasudevan Janarthanan, Ph.D.
Concordia University, 2007

The theory of supervisory control of discrete-event systems has been applied to real-time systems. The contribution of our proposed work lies in the development of a formal constructive method for controlling the preemptive execution of real-time tasks on both uniprocessor and multiprocessor systems. The set of all possible timed traces of the system is specified by a discrete timed automaton, where each transition is associated with an event occurrence or the passage of one unit of time. This approach allows a unified view of scheduling theory based on the timing analysis of models of real-time applications, meaning that the problem of determining schedulability and finding out a suitable scheduling algorithm are assumed to be intermingled issues, with the solution of one in turn is a solution to the other too.

First, a framework for designing universal schedulers for real-time tasks on uniprocessors based on Supervisory Control Theory (SCT) is presented. For this purpose, priorities are introduced in SCT and applied to the setting of discrete timed automata in order to develop a formal and unified framework for task scheduling on

a single CPU. A universal scheduler nondeterministically selects a task for execution in such a way that all timing constraints are met in a minimally restrictive fashion, while it contains all feasible deterministic scheduling policies.

We then extend that framework by providing a formal constructive method for controlling the preemptive and migrative execution of hard real-time tasks while scheduling them on a set of uniform processors. The methodology relies on the idea that the model of the scheduled system can be obtained by successive and appropriate restrictions of controllable actions of a model representing the real-time application. In uniform multiprocessors, each processor is characterized by its own computing capacity, with the interpretation that a task that executes on a processor of computing capacity $\mathbf{s}$ for τ time units completes $\mathbf{s}\tau$ units of execution.

Since we represented explicitly discrete time in our scheduler design, model sizes were considerably large. The complexity in the synthesis of a scheduler using supervisory control [62] stems from the fact that, with the synchronous product, the number of states of a composite Timed Discrete-Event System (TDES) increases exponentially with the number of real-time tasks. We have attempted to alleviate some of the state explosion problems we had faced while designing schedulers for real-time systems using supervisory control of discrete-event systems framework [39, 41], by providing an informal procedure to design schedulers with reduced state space.

Acknowledgments

I was fortunate and privileged to have Dr. Peyman Gohari as my advisor. He shaped my path to research by guiding me through his extensive knowledge, and at the same time opened up doors so that I could gain from the expertise of others. He has shown extreme enthusiasm for my progress and supported me when I was struggling, and it was he who introduced me into the area of supervisory control, nurturing me along in the last three years. He has been extraordinarily patient and supportive, having been always available for discussion and responding speedily to research reports. I would like to take this opportunity to thank him for his continued encouragement and guidance throughout the course of my research.

I owe a lot of gratitude to my thesis committee for their advice and patient reading of my thesis. Dr.Anjali Agarwal, Dr.Rachida Dssouli and Dr.Olga Ormandjieva gave me insightful critical advice, both in overview and in detail, and it was always reassuring to hear from them that I was on the right track during my research progress in the last three years.

This work would not have been possible without the environment at Concordia's Electrical and Computer Engineering department, with stimulating top-notch research, and at the same time an atmosphere that I can hardly imagine friendlier. I am grateful to all my fellow colleagues in the research group for their valuable discussions during this research.

Finally, and most importantly, thanks to my parents and my wife, to whom I owe what I am, who encouraged and supported me all these years, and who accepted the hardships of being far away so that I could see this through.

Vasudevan Janarthanan

I dedicate this work to my loving Dad, Mom and Wife.....

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter an overview of real-time systems in terms of their various constraints and scheduling techniques are presented. Also a brief introduction to Discrete-Event Systems (DES), Timed Discrete-Event Systems (TDES) and Supervisory Control of TDES are provided. Then we introduce the design procedure for schedulers on uniprocessor and multiprocessor systems, followed by our methodology for solving the state space explosion problem, which we had encountered during the design procedure.

1.1 Real-Time Systems

Real-time systems [55] are a form of control systems having constraints on the execution time of their tasks. These constraints are expressed as real-time constraints. A real-time constraint can be defined as a condition on the timing of enabling, firing, initiation and termination of system events. A real-time constraint can be expressed as a boolean condition on the values of clock variables, whose values increase with time. A real-time constraint of a task could be either the specification of its deadline or its complete execution.

A real-time system can be designed as a set of tasks that can be differentiated based on their timing requirements as hard real-time, soft real-time and non real-time tasks. A hard real-time task is defined as one whose timely and logically correct execution is considered crucial for the normal operation of the entire system. The deadline of a hard real-time task is referred to as hard deadline, because of the criticality of meeting the deadline. Hence it is obvious that missing a hard deadline can potentially result in a catastrophic system failure. On the other hand, a soft real-time task is characterized by an execution deadline which when met is desirable, but not critical for the functioning of the system. The deadline of a soft real-time task is referred to as soft deadline. Non real-time tasks are those having no real-time requirements at all.

Real-time tasks are further classified as periodic, aperiodic and sporadic tasks. Periodic tasks are those that occur, and will have to be executed, at regular intervals of time. Examples of applications where such tasks are commonly used are nuclear reactors and aircraft control systems, which are characterized by hard deadlines. On the other hand, aperiodic tasks are those whose executions are determined by the occurrence of internal or external events. For example, a task responding to a request from an external operator can be modeled by an aperiodic task. These tasks are usually characterized by soft deadlines. Finally, sporadic tasks are aperiodic tasks that are characterized by hard deadlines. For example, tasks dealing with emergency requests from a shuttle operator can be modeled by sporadic tasks.

The time domain in a real-time system can be either discrete or dense. Among the two, discrete-time allows for simpler analysis and design procedures since the real-time tasks could then be simply taking turns. A smallest measurable time unit is specified a priori in the discrete-time model. The clock used in our work includes an explicit tick transition, making time a global state variable. Each tick increments

time by some predetermined time quantum. Also, in our model, events between the i^{th} and $(i+1)^{th}$ clock ticks are assumed to occur at some unspecified time between times i and $(i+1)$.

1.2 Continuous And Discrete Time

In order to formalize the notion of time, two different varieties of time have been studied in the literature. One of them is the dense (continuous) modeling of time, wherein the time domain is equated with the set of nonnegative real numbers $\mathbb{R}^+ \cup \{0\}$. In this model, an event or a transition occurs at an arbitrary time point on the real scale. Whereas, the discrete modeling of time allows transitions to occur only at discrete time quantums. Here, the modeling of time is done using the set of nonnegative integer numbers $\mathbb{N}$. In [12, 36], the authors have done a comparative study on the relative merits of the two approaches.

If we compare the two timing models in terms of their expressiveness and their efficiency, then the dense time model is more expressive than the discrete time model. Further, while modeling delays that are arbitrarily small, it is beneficial to employ dense (continuous) timing model [12, 36]. Also, by using the dense modeling of time, there is no need to check if the granularity of the clock is suitable for modeling the various behaviors of the system; and in case of composing two dense-time systems, there is no need to check if the granularity of the two clocks match or not.

But for some classes of timed systems, using discrete time model helps in preserving certain properties. In [35, 36], the authors discuss timed transition systems, while showing that all qualitative, time-independent and some common quantitative properties such as time-bounded invariance and time-bounded response are preserved by discrete-time model. Also, they argue that if a property expressed in a certain timed

logic holds in the continuous-time model, a weaker, derived property is guaranteed to hold in the discrete time model.

However, in [16], the author shows that certain qualitative properties are not preserved if a discrete-time model is used instead of the continuous time model. In that paper, the author analyzes combinational circuits, wherein the timing constraints are expressed as bounded delays which are imposed on the output of each gate. For acyclic circuits, a discretization quantum is found such that the qualitative behavior is preserved. But in addition, there also exist certain cyclic circuits whose continuous time qualitative behavior is not preserved by any discretization.

In terms of efficiency, both discrete and continuous time models have their pros and cons. But according to [25], practical results for discrete-time models have been found to be better than the dense-time models. Discrete-time techniques allow efficient representation techniques from the untimed domain to be used, such as binary decision diagrams [27]. However, discrete-time techniques tend to be more sensitive to the size of the constants appearing in the model description, and large constants can result in state space explosion.

1.3 Real-Time Scheduling

Real-time scheduling is defined as assigning the exact execution times for a set of real-time tasks such that all the temporal constraints are satisfied. In a real-time system, the purpose of the scheduling algorithm is to determine the sequence of execution of the real-time tasks, thereby ensuring their adherence to resource and timing constraints. While designing a real-time system, the choice of an appropriate scheduling algorithm or policy depends on factors like task synchronization methods, number of processors available in the system and the priorities of the tasks. In

addition, characteristics of tasks pertaining to a particular application may influence the choice of a scheduling algorithm. For example, real-time application tasks can be preemptable or non-preemptable. A preemptable task is one whose execution can be suspended by other tasks, and resumed later; whereas a non-preemptable task must run until it finishes its execution, without interruption. Thus, both preemptive and non-preemptive algorithms have been proposed in [48].

Scheduling may be time-driven or priority-driven. A time-driven scheduling algorithm determines the exact execution time of all tasks. A priority-driven scheduling algorithm assigns priorities to tasks and determines which task is to be executed at a particular moment. Depending on the type of priority assignments, scheduling algorithms can be classified as fixed priority, dynamic priority and mixed priority scheduling algorithms. When the priorities assigned to tasks are fixed and do not change between job executions, the algorithm is called fixed priority scheduling algorithm. When priorities change dynamically between job executions, the algorithm is called dynamic priority scheduling. When a subset of tasks is scheduled using fixed priority assignment and the rest using dynamic priority assignment, the algorithm is called mixed priority scheduling.

The scheduling of periodic tasks on a single processor was one of the first scheduling problems analyzed in real-time systems [48]. In that paper, two different approaches were proposed to solve this problem. The approaches were based on the assignment of either a fixed or a dynamic priority value to each real-time task. These two approaches in turn led to the emergence of a number of preemptive scheduling policies. Among them are Rate Monotonic (RM), Earliest Deadline First (EDF) and Least Slack Time First (LSTF) policies.

In the RM algorithm, a task is assigned a fixed priority value based on the condition that shorter the task period, higher the task priority. In [48], the authors show

that the RM policy is optimal among fixed priority policies, meaning that for a given set of tasks, the RM policy always produces a feasible schedule if any other algorithm which is based on fixed priorities can do so. In the EDF algorithm, a task is assigned a priority value dynamically based on the condition that earlier the deadline of a task, higher the priority assigned to that task. In the LSTF policy, a task gets its priority based on its slack time, which is defined as the difference between the amount of time (from the current time value) to the deadline of the task, and the amount of time that the task requires to complete its computation. In the LSTF policy, smaller the slack time of a task, higher the priority value assigned to that task.

For the purpose of scheduling aperiodic tasks, five different policies were proposed in [46]. According to the first policy, aperiodic tasks are allowed to do their computations and thereby get scheduled only when no periodic tasks are active. In the second policy, a fixed priority periodic process is formed in order to serve the aperiodic task requests. This method is sometimes called polling. While this policy is cyclic, aperiodic tasks are bursty in nature. This leads to a huge incompatibility problem.

According to [46], the third and fourth policies are Priority Exchange (PE) and Deferrable Server (DS) policies. In both policies, a high priority periodic server is used to handle aperiodic task requests. The server preserves the execution time allocated to it if no aperiodic task requests are pending. This in turn improves the responsiveness of aperiodic tasks. The only difference between the two policies is in the way they manage the high priority of their periodic servers. In the PE policy, the server exchanges its priority with that of the pending highest priority periodic task if no aperiodic task requests occur at the beginning of the server period. Whereas in the DS policy, the server maintains its priority for the duration of its entire period. Therefore, aperiodic task requests can be handled at the servers high priority, provided that the servers execution time for the current period has not been exhausted.

The fifth policy that has been proposed in [46] is the Sporadic Server (SS) policy, which has been designed to handle the scheduling of aperiodic soft real-time tasks. This policy is also based on the creation of a periodic server of aperiodic requests, but the difference is in the lower implementation complexity of this policy compared to that of PE and DS policies.

1.4 Discrete-Event Systems

A discrete-event system (DES) is a dynamic system that evolves with the occurrence of events, such as the arrival of a job or the completion of a task. Because of the complex dynamics resulting from the various interactions between such events over time, modeling, design, and optimization of DES can be challenging problems. At the same time, the study of such systems has become increasingly important in recent times because of modern technological advances and wide use of computers in control applications. Such systems normally have discrete quantities that must be controlled, for example, communication networks. In a logical DES, the system is characterized by a set of states and the transitions (triggered by events) among these states. The behavior of the system is thus described by sequences of events. The occurrence of an event, caused by some unmodelled mechanism internal to the system, moves the system to a new state.

Discrete-event systems have been studied by researchers from various fields for the last couple of decades. During that time, a number of models have been proposed and analyzed with respect to the modeling, analysis and control of DES. These models can be classified as untimed DES models and timed DES models. In the untimed DES model, only the logical behavior of the system is considered. This means that the sequence of states visited is of concern; for instance, whether or not the system

will enter a particular state, but we do not care when the system enters that state or how long the system remains there. In a timed model [31], both logical behavior and timing information are considered. That is, we are concerned not only with the problem of whether or not the system will enter a particular state, but also with when the system enters that state and how long the system will remain there.

1.5 Supervisory Control Of Discrete-Event Systems

A discrete-event system, which is considered as a plant, is controlled by a supervisor that observes the events that occur in the plant. Each time an event is observed, the supervisor presents the plant with a set of events to be *disabled*. The supervisor, while doing that, tries to keep the plant away from *forbidden* states and prevent undesirable event sequences from occurring. A common objective of DES designers is to compute supervisors that ensure that the controlled behavior is within a specified *legal* behavior.

Peter J. Ramadge and W. Murray Wonham [54] devised a control theoretic formalism that facilitated the study of DES systems. They modeled such systems as state machines that permit, from an initial state, a set of event sequences or strings to occur. The set of such strings forms a language that contains every possible event sequence that can occur in the DES. When a supervisor disables events in the plant, the resulting behavior is another language (i.e., set of event sequences) called the closed-loop language. In order to make sure that the controlled behavior of the plant is within a specified *legal* behavior, we specify the *legal* behavior as a *legal* language that contains all allowable event sequences. The objective of the supervisor is to ensure that every event sequence possible in the controlled system is in the *legal* language.

Formally, when the plant and specification are represented by languages L and E, respectively, we say the system or the plant satisfies the specification if $L \subseteq E$, that is, if every event sequence generated by the plant is acceptable by the specification. When this is not the case, the objective of supervisory control is to design a supervisor S that restricts the behavior of the plant in such a way that the supervised system, whose behavior we denote by K, satisfies the specification, i.e. $K \subseteq E$. An important issue in supervisory control is to identify those sublanguages K of a given language E that can be the behavior of the supervised plant under some supervisory control. A supervisor is called *optimal* if it restricts the behavior of the plant to the supremal controllable sublanguage of the specification language. The supremal controllable sublanguage is computed in time polynomial in the state sizes of plant and specification when they are modeled by finite automata.

1.6 Supervisory Control Of Timed Discrete-Event Systems

In a timed discrete-event system (TDES), the behavior of the system is influenced in addition by the temporal characteristics of event sequences. As a matter of fact, in a TDES, the state transition could be triggered either by some unmodelled mechanism acting on the system, or by the passage of time. Supervisory control of a TDES [47], [51] in essence means timely disablement or enforcement 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*. The work reported here

adopts the framework for supervisory control of TDES originally proposed in [26]. This section summarizes the key concepts essential to the subsequent developments.

A TDES can be expressed as a five-tuple $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$, where Q is a set of states, Σ is a finite set of events, including a special event denoting a tick of the global digital clock, the partial function $\delta : Q \times \Sigma \rightarrow Q$ 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 completion of certain tasks of the system. Let Σ^* denote the set which contains, in addition to an empty string, all possible finite sequences over Σ. A state $q \in Q$ is *reachable* if there is a string $s \in \Sigma^*$ such that $\delta(q_0, s) = q$. A state $q \in Q$ is *coreachable* if there is a string $s \in \Sigma^*$ such that $\delta(q_0, s) \in Q_m$. A TDES is *trim* if all its states are reachable and coreachable.

A TDES can be obtained from an Activity Transition Graph (ATG). An ATG is an automaton in which each event α is defined with two time bounds, i.e., $(\alpha, l_\alpha, u_\alpha)$, where $l_\alpha \in T$ and $u_\alpha \in T$ are the lower and upper time bounds, respectively. The interpretation for such a definition is that α may occur after l_α ticks, and must occur at the latest after u_α ticks. An event is called *prospective* if $0 \leq l_\alpha \leq u_\alpha < \infty$, and *remote* if $0 \leq l_\alpha < u_\alpha = \infty$. Each event is thus associated with a timer interval between 0 and the event's upper time bound, i.e., $T_\sigma = [0, u_\sigma]$ for event σ (if σ is prospective), or the event's lower time bound, i.e., $T_\sigma = [0, l_\sigma]$ for event σ (if σ is remote).

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 a set of controllable events Σ_c and a set of uncontrollable events Σ_u. An event is controllable if it can be prevented from occurring at a specific point in a logical event sequence or in time, and is uncontrollable otherwise.

In a hard real-time system, an event must occur at the latest at its 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 events, called *forcible* events, is defined to describe events that can preempt a *tick* of the global clock. In this work, our concern is on tasks having a finite hard deadline. Therefore, all execution events considered are forcible unless stated otherwise. A supervisor of a TDES G can be considered as an automaton V that monitors the states of G, and enables, disables or forces certain events in G when necessary so as to influence the behavior of G. The readers are referred to [26] and [62] for a complete exposition of the theory of supervisory control of TDES.

1.7 Uniprocessor Scheduler Design

Any real-time application consists of a set of tasks that interact with each other, and the tasks' executions are subject to various temporal constraints such as completion times, deadlines, periods, resource sharing and synchronization delays. The crucial aspect of real-time scheduling is to make sure that these tasks satisfy their temporal constraints and that the overall system performs correctly according to its specification. Given this fact, we propose a framework for designing such schedulers for hard real-time systems upon uniprocessors based on Supervisory Control Theory (SCT) for timed discrete-event systems.

In order to design a scheduler, we first model the execution of a set of tasks as a TDES and then compute the supremal controllable sublanguage of timing constraints with respect to task TDES to find the desired behaviour of the system. For ensuring tasks' schedulability, we simply check the supremal controllable sublanguage for emptiness. Nonemptiness implies that a *legal* and controllable execution sequence of

all tasks exists such that all scheduling requirements are met.

The approach followed in this work to design a scheduler is independent of *a priori* fixed scheduling policies. That is, the scheduler need not be constrained to follow a predefined scheduling policy but may adopt its decisions based on the behavior of the environment and the property to be satisfied, as is the case with the controller synthesis of [26]. The method for designing schedulers is based on successive restriction of the system to be scheduled by the use of constraints derived from scheduling requirements.

As in [6, 49, 56], this approach allows a unified view of scheduling theory based on the timing analysis of models of real-time applications. One of the main contributions of this work is the synthesis of a *universal* scheduler, which contains all feasible deterministic scheduling policies. A universal scheduler nondeterministically selects a task for execution in such a way that all timing constraints are met in a minimally restrictive fashion.

1.8 Multiprocessor Scheduler Design

The problem of scheduling hard real-time systems upon multiprocessor platforms has been extensively dealt with in [18], [19], [20], [21]. However, no formal treatment has been provided in these works. A formal semantics must be provided so that the behavior of the scheduler (read *supervisor*) and the meaning of the specifications are clearly defined. For the above-mentioned reasons, we have utilized the theory of supervisory control of TDES to formalize and realize the scheduler design procedure for real-time systems. We synthesize schedulers using the *proof by construction* approach, wherein we demonstrate the existence of a scheduler capable of scheduling tasks on multiple CPUs by providing a method for constructing such a scheduler.

We discuss the scheduling procedure of hard real-time tasks on uniform multiprocessor [18, 19] platforms based on the assumptions that task preemption and interprocessor migration are permitted, while intra-task parallelism is forbidden (i.e., at any instant in time each task may be executing on at most one processor). In uniform multiprocessors, each processor is characterized by its own computing capacity, with the interpretation that a task that executes on a processor of computing capacity $\mathbf{s}$ for τ time units completes $\mathbf{s}\tau$ units of execution. The scheduler design procedure for hard real-time tasks on uniform multiprocessors is similar to the one followed in uniprocessor platform.

1.9 Symbolic Scheduler Design

In [39, 41], we had shown that supervisory control theory (SCT) of timed discrete-event systems could be applied to the scheduling of hard real-time systems. In particular, we had presented a formal framework for the synthesis of real-time schedulers on uniprocessor systems using priority-based supervisory control of timed discrete-event systems. The execution of a set of tasks was modeled as a Discrete Timed Automaton (DTA). Then the supremal controllable sublanguage [26, 62] of timing constraints with respect to task DTA subject to a priority relation was computed to find all executions in which no deadline was missed. We had also provided a method for designing schedulers on uniform multiprocessor systems based on SCT [40, 42].

Since we represented discrete time explicitly, model size for each task is considerably large and proportional to its period. The complexity in the synthesis of a scheduler using supervisory control [62] stems from the fact that, with the synchronous product, the number of states of a composite TDES increases exponentially with the number of real-time tasks. As far as complexity in both time and space is concerned,

the procedure TDES3 = sync(TDES1, TDES2), which computes the synchronous product of two TDES, has complexity proportional to the product of the state sizes of the two machines. Theoretically, the number of states of the synchronous product of two TDES is less than or equal to the product of their number of states. But in reality it is often much less than their product for a nontrivial system. Therefore, we often need to allocate much more space than is actually required to store the result.

One approach to confine this "state explosion problem" relies on the *symbolic* representation [13], [28], [38], [50] of sets of states, and computes the set that satisfies a formula as a fixpoint of a functional on state predicates. In our model, the formula is a guard for the transition from one state to the next. We have presented an informal idea to address state space explosion in scheduler synthesis for real-time tasks on uniprocessor systems using symbolic methods [24]. The scheduler thus designed is a feasible one for a given set of real-time tasks, and also smaller than the ones designed with any other known mechanism.

1.10 Contribution Of The Thesis

1. In this thesis, we have utilized the concept of supervisory control theory (SCT) of discrete-event systems in order to formalize the process of scheduling hard real-time tasks on single processor platforms. For this purpose, we have provided a formal framework which helps in synthesizing real-time schedulers on single processor systems using priority-based supervisory control of timed discrete-event systems. We name such synthesized schedulers *universal* since they contain all feasible deterministic scheduling policies. In other words, a universal scheduler nondeterministically selects a task for execution in such a way that all timing constraints are met in a minimally restrictive fashion.

2. We have then extended our formalization technique to multiple processor systems by presenting a framework for designing schedulers for hard real-time systems upon *uniform* multiprocessors based on Supervisory Control Theory (SCT) for timed discrete-event systems. Our contribution in this respect has been the development of a formal constructive method for controlling the preemptive and migrative execution of real-time tasks on a set of uniform processors.

3. As we had considered discrete time models in our scheduler design, the state sizes were substantially large, and increased exponentially with the number of real-time tasks. In order to reduce the state space explosion problem in our models, we have utilized a modified form of symbolic modeling methodology [24], along with the pre-stable algorithm proposed in [24], for reducing state space while designing schedulers for real-time tasks on uniprocessor systems. The main contribution here has been the development of an informal procedure for uniprocessor scheduler design with reduced state space for both non-preemptive and preemptive real-time tasks.

1.11 Organization Of The Thesis

The rest of the thesis is organized as follows. Chapter 2 delves in detail into some of the previous work related to our thesis that has been found in the literature. In Chapter 3, we introduce the theory of priority-based supervisory control of discrete-event systems, and propose a method for constructing a universal scheduler. The chapter also provides an example illustrating in depth the procedure followed in designing a scheduler for hard real-time tasks under various temporal constraints on a single processor system. Chapter 4 illustrates various multiprocessor set ups, and looks into specific reasons for considering *uniform* multiprocessors in our models. Further, it

presents a formal framework for scheduling preemptive and migrative hard real-time tasks on uniform multiprocessors, followed by an example to illustrate the design procedure. In Chapter 5, we provide the framework for scheduler design with reduced state space using modified symbolic technique. The framework is supplemented by an exhaustive example that clearly illustrates the procedure followed in designing a scheduler for non-preemptive real-time tasks with reduced state space. This chapter also explains the scheduler design procedure under preemptive conditions. Finally in Chapter 6, we conclude the thesis with a brief note on some of the future prospectives of our work.

Chapter 2

Previous Work

Controller synthesis has been studied extensively, both for discrete-time and dense-time systems. One of the oldest discrete-event frameworks is the one of supervisory control of discrete-event systems [54]. Within the control community, Ramadge and Wonham [54] have built an extensive automata-theoretic framework for defining and solving control synthesis problems for discrete-event systems. In the paradigm of standard supervisory control theory, the authors have formulated the supervisory control problem by two languages that correspond to minimal acceptable behavior and legal behavior, respectively. In this formulation, both general and nonblocking solutions are well discussed.

Various formalisms have been developed to model timed discrete-event systems, including [8] and [26]. Work in [26] corresponds to discrete-time using a single system timer as opposed to [8] where the dense-time model includes several asynchronous clocks and is more expressive than other formalisms, allowing composition of timed processes and independent timing conditions for each system component. Here, we are concerned with timed automata as defined in [26].

The idea of applying synthesis to timed automata was first explored in [60]. Here

the methods of Ramadge and Wonham [54] are adapted to construct a supervisor for a dense real-time discrete-event system modeled by a timed automaton using untiming procedures. The control problem is tackled by completely discretizing the timed automaton into a finite state automaton and then the discrete synthesis problem is solved, which is somewhat similar to what we have proposed in our work.

An algorithm for safety controller synthesis for timed automata, based on operation on zones, was first reported in [49] and later in [13], where an example of a simple scheduler was given. In these and other works on treating scheduling problems as synthesis problems for timed automata, such as [4], the emphasis has been on existence properties, such as the existence of a feasible schedule in the presence of an uncontrolled adversary. The approach followed in our work for the construction of a scheduler is somewhat similar to [4, 5, 6, 14, 49, 56], but time in our work is discrete and it is explicitly represented, as opposed to dense-time which is often implicitly represented in the aforementioned references. In implicit representation of time, a set of inequalities over timer variables are used to define conditions on the firing of transitions (known as guards), as well as to specify timing requirements on the system behavior. The method for designing schedulers is based on successive restriction of the system to be scheduled by the use of constraints defined from the scheduling requirements.

Ostroff and Wonham were among the pioneers in the modeling and analysis of real-time discrete-event systems. In [52, 53], the authors had presented a framework/procedure for supervisory control of possibly infinite state real-time discrete-event systems using timed transition models (TTMs) and real-time temporal logic (RTTL). TTM has been used to represent the processes of plants and controllers, while RTTL has been used as the assertion language for specifying the legal plant behaviour, and it is shown that the controller indeed satisfies the required specification.

In [2], the synthesis algorithm computes iteratively, from a constraint K characterizing scheduling requirements, the maximal control invariant K': $K' \Rightarrow K$. The formula K' denotes the set of states from which K is guaranteed. The behavior of the scheduled system is obtained by restricting the controllable actions of the processes so as to respect the control invariant K'. As in [4, 5, 6, 14, 49, 56], our approach allows a unified view of scheduling theory based on the timing analysis of models of real-time applications.

The problem of scheduling the non-preemptive execution of a set of periodic tasks with hard deadlines on a single processor is considered in [29, 30]. For this purpose, the authors have used the theory of supervisory control of discrete-event systems [54]. It has also been shown there that the computation of the supremal controllable sublanguage with respect to a finite timed discrete-event system can be completed in polynomial time. The present work is also based on the unified approach, i.e., schedulability check and finding a scheduling algorithm are considered as one problem. But our proposed approach not only schedules non-preemptive tasks but also preemptive ones as well.

In [43, 44, 45], real-time behavior is represented using timed automaton which uses a dense model of time. The control is achieved by prioritized synchronous composition of a plant and a supervisor timed automaton. The notion of prioritized synchronization for the untimed systems has been extended to the real-time setting. The authors have also shown that the prioritized synchronous composition is associative and under certain mild conditions, which hold in the supervisory control setting, can be reduced to the strict synchronous composition by using a technique known as *augmentation.*

A methodology for treating the problem of scheduling partially-ordered tasks (task graph) on parallel machines has been considered in [1]. The authors have shown how

one can schedule tasks on a limited number of identical machines, while respecting some precedence constraints. In the framework, the scheduling problem admits a state-space representation, and an optimal schedule corresponds to a shortest path in the timed automaton. This work shows how formal state-based models can be used to express parallel scheduling problems and support efficient algorithms for solving such problems.

The problem of scheduling hard real-time systems upon multiprocessor platforms has been extensively dealt with in [18, 19, 20, 21, 32, 33, 34, 58]. It has been shown there that earliest deadline first remains a predictable and resource-efficient algorithm to use in multiprocessor systems. Also, through the results in their work, the authors conclude that scheduling upon multiprocessor platforms is not an obvious extension of one's knowledge concerning the uniprocessor case, but a lot more than that. But no formal treatment has been provided in these works, which is what we seek in our work.

Our approach to confine the *state explosion* problem, which we faced while designing schedulers for uniprocessor and multiprocessor systems, relies on the *symbolic* representation of sets of states, which in essence computes the set that satisfies a formula as a fixpoint of a functional on state predicates. In our model, the formula is a guard for the transition from one state to the next. In [38], the authors illustrate the working of a symbolic model checking algorithm that works on a quotient of the region graph that depends on the formula being checked. They have shown how a symbolic fixpoint approach can be used to test if a guarded-command real-time task is non-zeno and, if not, how it can be converted into an equivalent non-zeno task. But this algorithm fails to terminate if no such quotient exists. Also, this algorithm requires that the explicit representation of discrete structure of the automaton be constructed a priori.

In [17], the author has introduced an approximation scheme that further reduces the complexity related to timing. The work in that paper is based on the observation that not all the timing information in the description of a timed system is usually needed to guarantee the satisfaction of a given property. This approximation scheme has been used in the model-checker RT-Cospan [9]. In [9], the underlying untimed description of the system is composed with an automaton representing the time bounds, and only the bounds that are necessary to verify the given property are introduced in the composition.

In [37], the authors have employed *on-the-fly* and *space-efficient* model-checking methods to solve the state explosion problem. Using the first method on a real-time program, they explore only the regions needed for checking the satisfaction of a specification. The second method is used to store only necessary and minimal information in the memory. But the main drawback in [37] is that in the worst case situation, the on-the-fly method explores the entire region graph.

The authors in [24] have proposed an algorithm combining the symbolic and on-the-fly approaches. Their algorithm performs an on-the-fly exploration of the symbolic graph. The authors name the resulting graph as *simulation* graph, and they show in their paper that simulation graphs are much smaller than region graphs. They also perform model-checking for a temporal logic formula using simulation graph in [24]. We have incorporated the algorithm proposed in [24] in our work for the purpose of designing schedulers for real-time tasks on uniprocessor systems rather than for model-checking.

Chapter 3

Uniprocessor Scheduler Design

In this chapter we formalize real-time task scheduling by applying an extension of Supervisory Control Theory (SCT) of discrete-event systems to real-time models. The set of all possible timed traces of the system is specified by a discrete timed automaton, where each transition is associated with an event occurrence or the passage of one unit of time. We introduce priorities to SCT, and apply them to the setting of discrete timed automata in order to develop a formal and unified framework for task scheduling on a single CPU.

The chapter is organized as follows. Section 3.1 introduces the theory of priority-based supervisory control of DES, which is ultimately used to synthesize schedulers. Section 3.2 sets up a framework for modeling real-time systems and specifications based on a timed automata called Discrete Timed Automata (DTA), and proposes a method for constructing an *universal* scheduler. Section 3.3 provides a comprehensive example illustrating in detail the procedure followed in designing a scheduler under the various task requirements of Section 3.2. Finally, we conclude this chapter in Section 3.4.

3.1 Priority-Based Supervisory Control Of Discrete-Event Systems

In real-time scheduling theory, priorities are assigned to tasks in order to improve their schedulability. When several tasks are ready for execution, the one with the highest priority will be executed first. In addition, if allowed, a higher-priority task can *preempt* a lower priority task. In this section we formalize priorities in supervisory control of DES framework. The theory developed will be extended to timed models in the next section to synthesize supervisors for scheduling real-time tasks.

3.1.1 Priority Relations

In supervisory control of DES [54] the desired behavior of a system is often specified by a language[1] E over the alphabet of system events Σ. After observing a string s generated by the plant, a supervisor will guarantee that the specification is met by offering the plant with events from a set $\gamma(s) \subseteq \Sigma$ that if executed, the resulting behavior stays in E, i.e., $\{s\}\cdot\gamma(s) \subseteq E$. In classical supervisory control theory all events in the set $\gamma(s)$ have equal "priority", in the sense that one is nondeterministically selected for execution by the plant. However, in some applications it may be desirable to give an event priority of execution over another when they are competing for a shared resource. A convenient way to state this property is by introducing a *priority relation.*

Definition 1 A *priority relation* P is an antisymmetric relation over Σ_c with the interpretation that $(\alpha_1, \alpha_2) \in P$, denoted by $\alpha_1 \prec \alpha_2$, when α_2 has a higher priority than α_1. □

[1] We assume that all languages are prefix-closed.

Remark. Priorities are introduced to enhance a controller's scheduling ability. For example, a controller assigns higher priorities of execution to tasks with smaller deadlines to improve the probability of meeting all the deadlines. As such, it is a reasonable assumption to define a priority relation between controllable events only. □

The relation $\prec$ is assumed to be antisymmetric, that is, if α_2 has a higher priority than α_1, then α_1 cannot have a higher priority than α_2. In the next subsection we will see how priority relations can be taken into account in supervisory control design.

3.1.2 Theory Of Priority-Based Supervisory Control

Let Σ be an alphabet, $P \subseteq \Sigma_c \times \Sigma_c$ be a priority relation, and L and E be two prefix-closed languages over Σ representing plant and specification behaviors, respectively. For the sake of convenience assume for now that E is controllable with respect to L. Then in the absence of a priority relation E can be thought of as the language generated by a controller that when coupled with the plant, the closed-loop behavior $E \cap L$ is controllable with respect to the plant and satisfies the specification: $E \cap L \subseteq E$.

However, not all strings in $E \cap L$ respect the priority relation P. In particular, if $\alpha_1 \prec \alpha_2$ and at a string $s \in E \cap L$ both α_1 and α_2 are possible continuations of s in $E \cap L$, then for the priority relation to be respected a supervisor must disable the event with the lower priority (i.e. α_1). The following definition characterizes the class of languages that respect a priority relation.

Definition 2 Let $P \subseteq \Sigma_c \times \Sigma_c$ be a priority relation and $N, M \subseteq \Sigma^*$ be prefix-closed languages. We say N *conforms* to P with respect to M, denoted by $N \vdash_M P$ if:

$$\forall s \in \Sigma^* \forall \alpha_1, \alpha_2 \in \Sigma_c.\ \alpha_1 \prec \alpha_2 \wedge s\alpha_2 \in M \Rightarrow s\alpha_1 \notin N$$

□

In the sequel, we set M equal to a controllable sublanguage of $E \cap L$ and $N \subseteq M$. The definition states that in the presence of higher priority events in a string of M, to conform to a priority relation a controller must disable all lower priority events at that string in N.

Remark. From the above definition it can be inferred that N conforms to P when, for one reason or another, high priority events are disabled by a controller; in particular, in order to meet the specification of some legal behavior. Thus, there is no need to further restrict a behavior by disabling low-priority events when high-priority events are already disabled. As we will see shortly this makes it possible to properly define maximally permissive behaviors conforming to a given priority relation. □

When a prefixed-closed language $N \subseteq \Sigma^*$ does not conform to P with respect to M, additional control may be required to remove low priority events that compete with high priority ones. To this end we define the class of sublanguages of N in which all low priority events are disabled when they compete with high priority events in M:

$$\mathcal{K}_P(N; M) = \{K \subseteq N |\ K \vdash_M P\}$$

The following result states that $\mathcal{K}_P(N; M)$ contains its supremal element.

Proposition 1 *The supremum of $\mathcal{K}_P(N; M)$, denoted by $\mathcal{K}_P^{\uparrow}(N; M)$, exists and belongs to $\mathcal{K}_P(N; M)$.*

Proof. For simplicity we drop references to N and M. First observe that $\mathcal{K}_P$ is nonempty since $\emptyset \in \mathcal{K}_P$, and therefore $\mathcal{K}_P^{\uparrow} = \bigcup \mathcal{K}_P$, where $\bigcup \mathcal{K}_P$ denotes the union of all members of $\mathcal{K}_P$. To show that $\bigcup \mathcal{K}_P \in \mathcal{K}_P$, we first note that $\bigcup \mathcal{K}_P \subseteq N$, which is immediate since N is an upperbound of $\mathcal{K}_P$ and is therefore larger than its least

upperbound. Let $s \in \Sigma^*$ and $\alpha_1, \alpha_2 \in \Sigma_c$ be such that $\alpha_1 \prec \alpha_2$. We have:

$$\begin{aligned} s\alpha_1 \in \bigcup \mathcal{K}_P &\Longrightarrow s\alpha_1 \in K; \quad \text{for some } K \in \mathcal{K}_P \\ &\Longrightarrow s\alpha_2 \notin M; \quad \text{since } K \vdash_M P \end{aligned}$$

Therefore $\mathcal{K}_P^{\uparrow} \in \mathcal{K}_P$, as desired. ■

Note that if N conforms to P with respect to M then $N = \mathcal{K}_P^{\uparrow}(N; M)$. In this case, priorities of the competing events in N are not comparable.

Recall from classical supervisory control theory [54] that a supervisor that generates the supremal controllable sublanguage of a given specification language E, denoted by $\sup\mathcal{C}(E)$, restricts the system behavior in a minimally restrictive fashion. When a priority relation between controllable events is defined, the language $\mathcal{K}_P^{\uparrow}(\sup\mathcal{C}(E); \sup\mathcal{C}(E))$ is minimally restrictive in the sense that a controllable event is disabled at a string $s \in \mathcal{K}_P^{\uparrow}(\sup\mathcal{C}(E); \sup\mathcal{C}(E))$ only when it leads to the violation of the safety specification E, or else when it competes with higher priority events at s. Note that since P is defined between controllable events, $\mathcal{K}_P^{\uparrow}(\sup\mathcal{C}(E); \sup\mathcal{C}(E))$ remains controllable and can therefore be implemented by a supervisory control map $V : L \to \Gamma$ [62].

Corollary 2 *Let L and E be plant and specification languages, respectively, and P be a priority relation. Then a generator for $\mathcal{K}_P^{\uparrow}(\sup\mathcal{C}(E); \sup\mathcal{C}(E))$ controls the system in a minimally restrictive fashion, in the sense that $\mathcal{K}_P^{\uparrow}(\sup\mathcal{C}(E); \sup\mathcal{C}(E))$:*

1. *is controllable with respect to L,*

2. *conforms to P with respect to $\sup\mathcal{C}(E)$,*

3. is a subset of E, and:

4. For any supervisor $\mathbf{K}$ *generating a language* K*, if* K *satisfies (1), (2), and (3) then* $K \subseteq \mathcal{K}_P^{\uparrow}(\sup \mathcal{C}(E); \sup \mathcal{C}(E))$. ■

3.1.3 Computation of priority-based supervisory control

We assume that M and N are represented by two automata $\mathbf{M} = (X, x_0, \Sigma, \xi)$ and $\mathbf{N} = (Y, y_0, \Sigma, \eta)$, respectively. We would like to compute the language $\mathcal{K}_P^{\uparrow}(N; M)$ of the previous subsection.

Definition 3 The *prioritized system*, denoted by $\mathbf{M_P}$, is a 4-tuple (X, x_0, Σ, ξ_P), where $\xi_P : X \times \Sigma \rightarrow X$ is defined on a pair $(x, \sigma) \in X \times \Sigma$, denoted by $\xi_P(x, \sigma)!$, if and only if ξ is defined on that pair and

$$\forall \sigma' \in \Sigma.\ \sigma \prec \sigma' \Rightarrow \neg \xi(x, \sigma')!$$

in which case $\xi_P(x, \sigma) := \xi(x, \sigma)$. □

The language generated by $\mathbf{M_P}$ has the property that when intersected with any language N it will produce $\mathcal{K}_P^{\uparrow}(N; M)$ restricted to M. This is formally stated and proved in the following proposition.

Proposition 3 *Let* $M, N \subseteq \Sigma^*$ *be prefixed-closed languages and* $P \subseteq \Sigma_c \times \Sigma_c$ *be a priority relation. Then we have:*

$$N \cap M_P = \mathcal{K}_P^{\uparrow}(N; M) \cap M$$

where M_P *is the language of the prioritized system. Thus, when* $N \subseteq M$ *we have* $K_P^{\uparrow}(N; M) = N \cap M_P$.

Proof. Denote $\mathbf{M} := (X, x_0, \Sigma, \xi)$, $\mathbf{M_P} := (X, x_0, \Sigma, \xi_P)$ and $\mathbf{N} := (Y, y_0, \Sigma, \eta)$. We start by observing that

$$\mathcal{K}_P^{\uparrow}(N; M) \cap M = \mathcal{K}_P^{\uparrow}(N \cap M; M)$$

To show $N \cap M_P = \mathcal{K}_P^{\uparrow}(N \cap M; M)$, we must show that 1) $N \cap M_P \in \mathcal{K}_P(N \cap M; M)$, which implies $N \cap M_P \subseteq \mathcal{K}_P^{\uparrow}(N \cap M; M)$, and 2) $N \cap M_P$ is an upper bound for $\mathcal{K}_P(N \cap M; M)$, which implies $N \cap M_P \supseteq \mathcal{K}_P^{\uparrow}(N \cap M; M)$.

1. Since $M_P \subseteq M$ it follows that $N \cap M_P \subseteq N \cap M$. Let $s \in \Sigma^*$ and $\sigma, \sigma' \in \Sigma$ be such that $\sigma \prec \sigma'$. If $s\sigma \in N \cap M_P$ then $s\sigma \in M_P$. Denote $x := \xi_P(x_0, s) = \xi(x_0, s)$. Since $s\sigma \in M_P$ we have $\xi_P(x, \sigma)!$, and therefore it follows from the definition that $\neg\xi(x, \sigma')!$, i.e. $s\sigma' \notin M$. We conclude that $N \cap M_P \in \mathcal{K}_P(N \cap M; M)$.

2. Let $K \in \mathcal{K}_P(N \cap M; M)$. We must show $K \subseteq N \cap M_P$, i.e. for all $s \in \Sigma^*$,

$$s \in K \Rightarrow s \in N \cap M_P$$

We prove this by induction on the length of s.

- Base: If $\epsilon \in K$ it follows that $N \cap M \neq \emptyset$; therefore $N \cap M_P \neq \emptyset$ and $\epsilon \in N \cap M_P$.
- Inductive step: suppose for all s of length $|s| \leq n$ we have $s \in K \Rightarrow s \in N \cap M_P$. Let $s\sigma \in K$. We must show $s\sigma \in N \cap M_P$. Since $K \vdash P$ and $s\sigma \in K$ it follows that:

$$\forall \sigma' \in \Sigma.\ \sigma \prec \sigma' \Rightarrow s\sigma' \notin M \tag{3.1}$$

Note that $s\sigma \in K$ and $K \subseteq N \cap M$ implies $s\sigma \in N$ and $s\sigma \in M$. In addition, since K is prefix-closed it follows that $s \in K$, implying by the induction assumption that $s \in M_P$. Let $x := \xi_P(x_0, s) = \xi(x_0, s)$. Since $s\sigma \in M$, we have $\xi(x, \sigma)!$, implying by (3.1) that $\xi_P(x, \sigma)!$, i.e. $s\sigma \in M_P$. Thus we have shown that $s\sigma \in N \cap M_P$. ■

So far we have shown how priorities can be taken into account when computing a supervisor for an untimed system. In the next section, we apply our method to real-time systems by modeling real-time tasks and specifications using a simple extension of automata called Discrete Timed Automata (DTA). First the execution of a set of tasks is modeled as a DTA. Then the supremal controllable sublanguage of timing constraints with respect to task DTA subject to a priority relation is computed to find all executions in which no deadline is missed.

3.2 Real-Time Systems And Specifications

In this section we set up a general framework for modeling real-time systems and specifications. DTA is used to model real-time systems and to specify requirements on their behavior, including timing constraints. Time is assumed to be *discrete* (as opposed to dense [8]), and it is *explicitly* represented in our models (as opposed to implicit representation).

3.2.1 Discrete Timed Automaton

A Discrete Timed Automaton (DTA) is a four-tuple $\mathbf{L} := (X, x_0, \Sigma^t, \delta)$, where X is the state set, x_0 is the initial state, Σ^t is an alphabet, and $\delta : X \times \Sigma^t \rightarrow X$ is a partial transition function. We assume that $\Sigma^t := \Sigma \dot{\cup} \{t\}$, where Σ is the alphabet of system

events and t is a special symbol which denotes the passage of one unit of time, or one *tick* of the global *digital* clock. We denote by *tickcount* a function over the set of all strings that returns the number of *ticks* in a string:

$$tickcount : \Sigma^{t*} \rightarrow \mathbb{N} : s \mapsto \text{ the number of } t\text{'s in } s$$

Time is measured in discrete steps; if $x = \delta(x_0, s)$ and $tickcount(s) = n \in \mathbb{N}$ then at state x discrete-time is equal to n while the value of real time could be anywhere in the real interval $[n, n+1)$. As before, we assume that the system events are instantaneous.

For a DTA to model a real-time system, time always has to have a chance to advance. We say a DTA satisfies *time-progress* property if

$$\forall x \in X. \exists u \in \Sigma^*.\ \delta(x, ut)!$$

In our work, we restrict our attention to the class of DTA that satisfies the time-progress property.

Let L be a prefix-closed language over Σ^t. We call L a *timed language* if

$$\forall s \in L. \exists u \in \Sigma^*.\ sut \in L$$

It is immediate from the definitions that if $\mathbf{L}$ is a DTA satisfying time-progress property, then the language generated by $\mathbf{L}$, denoted by L, is a timed language.

An important property of timed languages is that they are closed under arbitrary union. Then, when the language returned by a control algorithm is not a timed language, it is guaranteed that a largest timed sublanguage of the behavior exists.

Formally, for $E \subseteq \Sigma^{t*}$ let

$$\mathcal{T}(E) := \{K \subseteq E | K = \overline{K} \wedge K \text{ is a timed language}\}$$

Proposition 4 *The supremum of $\mathcal{T}(E)$, denoted by $\mathcal{T}^{\uparrow}(E)$, belongs to $\mathcal{T}(E)$.*

Proof. First notice that since $\emptyset \in \mathcal{T}$ the set $\mathcal{T}$ is nonempty and therefore $\mathcal{T}^{\uparrow} = \bigcup \mathcal{T}$. It can be readily verified that $\bigcup \mathcal{T}$ is prefix-closed and a subset of E. Let $s \in \bigcup \mathcal{T}$. It follows that $s \in K$ for some $K \in \mathcal{T}$. Since K is a timed language it follows that

$$\exists u \in \Sigma^*. \quad sut \in K; \quad \text{since } K \text{ is a timed language}$$
$$\exists u \in \Sigma^*. \quad sut \in \bigcup \mathcal{T}; \quad \text{since } K \subseteq \bigcup \mathcal{T}$$

Therefore $\mathcal{T}^{\uparrow} = \bigcup \mathcal{T}$ is a timed language. ■

Let E' denote the largest timed sublanguage of E. Then it can be readily shown that the DTA $\mathbf{E}'$ is obtained from $\mathbf{E}$ by recursively removing all states from which time does not have a chance to advance.

In [62] it is shown that the class of controllable and prefix-closed sublanguages of a given language is also closed with respect to arbitrary union, resulting in the following corollary.

Corollary 5 *Let*

$$\mathcal{C}_t(E, L) := \{K \subseteq E \,\big|\, K = \overline{K} \wedge K \textit{ is a timed language} \wedge K \textit{ is controllable with respect to } L\}$$

Then $\mathcal{C}_t(E, L)$ has a largest element, which we denote by $\sup \mathcal{C}_t(E, L)$.

To calculate the supremal element of $\mathcal{C}_t(E, L)$, we first find the supremal controllable sublanguage of E [62]. If the result is a timed language, the procedure terminates.

Otherwise, we obtain its largest timed sublanguage by recursively removing from $\sup \mathcal{C}(E, L)$ all states in which time has no chance to advance, and repeat the above procedure if necessary.

3.2.2 General Task Model

In this subsection we study a system consisting of n tasks that are to be scheduled on a single CPU. For $i \in I := \{1, 2, \ldots, n\}$ let $\mathbf{T_i}$ denote a task with an execution time of $\nu_i \in \mathbb{N}$ time units. Each task is modeled by a DTA, so that the *composite* task model can be described as:

$$\mathbf{T} := \Big|\Big|_{i \in I} \mathbf{T_i}$$

where the operator || denotes synchronous product [62].

A task consists of several identical *instances*. Arrival of an instance of a task $\mathbf{T_i}$ is denoted by an event a_i called *arrival of the task* $\mathbf{T_i}$. We divide execution of an instance into ν_i *segments*, where each segment has a duration of one time unit. While execution of an instance can be interrupted, a segment—which can be regarded as the unit of execution—is uninterruptible. We denote the execution of segment j, $j \in [1, \nu_i)$, of any instance of task i by an event e_i. In our DTA model of task i we follow e_i by a *tick* to indicate that the execution of each segment consumes one unit of the processor time. We introduce a new symbol c_i to denote the execution of the last segment of $\mathbf{T_i}$. The general model of task $\mathbf{T_i}$ is shown in Figure 3.1. According to Figure 3.1, other tasks can always be executed except when the execution of a segment of the current task is in progress.

For future reference we define sets of events as shown in Table 3.1.

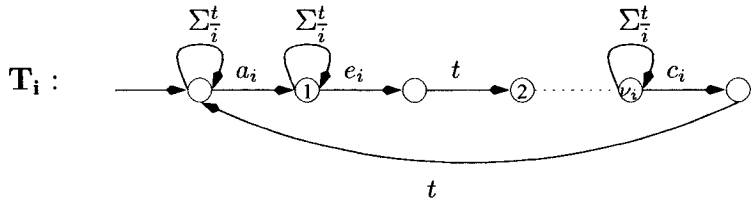

Figure 3.1: General model of task $\mathbf{T_i}$. State labels indicate which segment of task i is to be executed next.

Alphabet	Description
$\Sigma_i := \{a_i, c_i, e_i\}$	Set of all events of task i.
$\Sigma_{i,f} := \{c_i, e_i\}$	Set of forcible events of task i.
$\Sigma_{i,c} := \{c_i, e_i\}$	Set of controllable events of task i.
$\Sigma_{i,u} := \Sigma_i \backslash \Sigma_{i,c}$	Set of uncontrollable events of task i.
$\Sigma := \bigcup_{j=1}^{n} \Sigma_j$	Set of events of all tasks.
$\Sigma_{\bar{i}} := \Sigma \backslash \Sigma_i$	Set of events of all but task i.
$\Sigma_a := \bigcup_{j=1}^{n} \{a_j\}$	Set of arrival events of all tasks.

Table 3.1: Event sets used for uniprocessor scheduler design.

3.2.3 Task Requirements

Priority

When the execution of task j has a higher priority than the execution of task i, task j will be scheduled for execution first when they both are ready for execution, and if preemption is allowed, task j preempts task i as soon as the execution of the current *segment* of task i is complete. This can be modeled in our setting by introducing a priority relation $P \subseteq \Sigma_c \times \Sigma_c$ defined as:

$$P := \{(c_i, c_j), (c_i, e_j), (e_i, c_j), (e_i, e_j)\}$$

Events a_i and a_j are not considered in the definition of P since they are uncontrollable events.

Periodicity

A task is *periodic* if its instances arrive at equidistant moments in time. Assuming without loss of generality that the first instance always arrives at time zero, the DTA of Figure 3.2 specifies that arrivals of $\mathbf{T_i}$ are $\wp_i$ time units apart, where for simplicity $\wp_i = 4$.

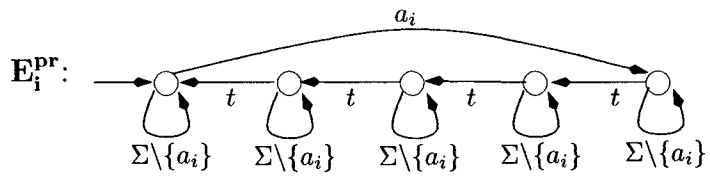

Figure 3.2: Specification for making $\mathbf{T_i}$ periodic with period $\wp_i$ ($\wp_i = 4$).

Note that $\mathbf{E_i^{pr}}$ is an *environmental constraint* in that it is the environment that *forces* a_i at integer multiples of $\wp_i$ and *disables* it elsewhere. However, a_i is neither forcible nor controllable, that is, the *scheduler* yet to be designed can neither force nor disable any of the arrival events.

Periodicity and other environmental requirements can be expressed by a specification automaton $\mathbf{E^{env}}$ defined as:

$$\mathbf{E^{env}} := \Big|\Big|_{i \in I} \mathbf{E_i^{pr}}$$

Non-Preemption

When preemption is *not* allowed, the model of Figure 3.1 needs to be restricted by requiring that after the execution of $\mathbf{T_i}$ has started (event e_i), other tasks may not be executed until the execution of $\mathbf{T_i}$ is complete (event c_i). This is shown in Figure 3.3.

Let $J \subseteq I$ be the index subset of nonpreemptible tasks. The nonpreemption requirement can then be expressed by an automaton $\mathbf{E^{sch}}$ (*scheduling constraint*)

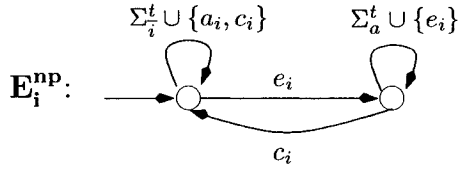

Figure 3.3: Specification for making $\mathbf{T_i}$ nonpreemptible.

defined as:

$$\mathbf{E^{sch}} := \Big|\Big|_{j \in J} \mathbf{E_j^{np}}$$

From a supervisory control perspective, a scheduler is just a supervisor that controls the system in such a way that all logical and timing constraints are met. According to Corollary 5, a supremal scheduler, denoted by $\mathbf{S}$ exists and is given by

$$\mathbf{S} := \mathcal{K}_{\mathbf{p}}^{\uparrow}\ (\sup \mathcal{C}_{\mathbf{t}}(\mathbf{E^{sch}}, \mathbf{L}), \sup \mathcal{C}_{\mathbf{t}}(\mathbf{E^{sch}}, \mathbf{L}))$$

where $\mathbf{L} := \mathbf{T}||\mathbf{E^{env}}$.

The entire design procedure is depicted in Figure 3.4. At first, various tasks are modeled by task automata as in Figure 3.1. The task automata are then combined to obtain a composite task model $\mathbf{T}$. Meanwhile, task periods modeled as in Figure 3.2 are combined together to yield the environmental constraint denoted by $\mathbf{E^{env}}$ in the figure. Next, the task model $\mathbf{T}$ and the environmental constraint $\mathbf{E^{env}}$ are combined to obtain the plant $\mathbf{L}$. Also, if task preemption is not allowed, the requirements are specified by the DTA of Figure 3.3, which are then combined to obtain the scheduling constraint $\mathbf{E^{sch}}$ of Figure 3.4. Finally we arrive at the maximally permissive supervisor $\mathbf{S}$ by computing the supremal controllable sublanguage of the specification $\mathbf{E^{sch}}$ with respect to plant $\mathbf{L}$ in which all competing low-priority evens are disabled. The supervisor thus synthesized is called a *scheduler* in real-time community, which is a

program that dispatches real-time tasks for execution in some specific order. The scheduler obtained is *minimally restrictive* in the sense that under the same set of constraints, any other feasible scheduler will impose additional restrictions on the system behavior.

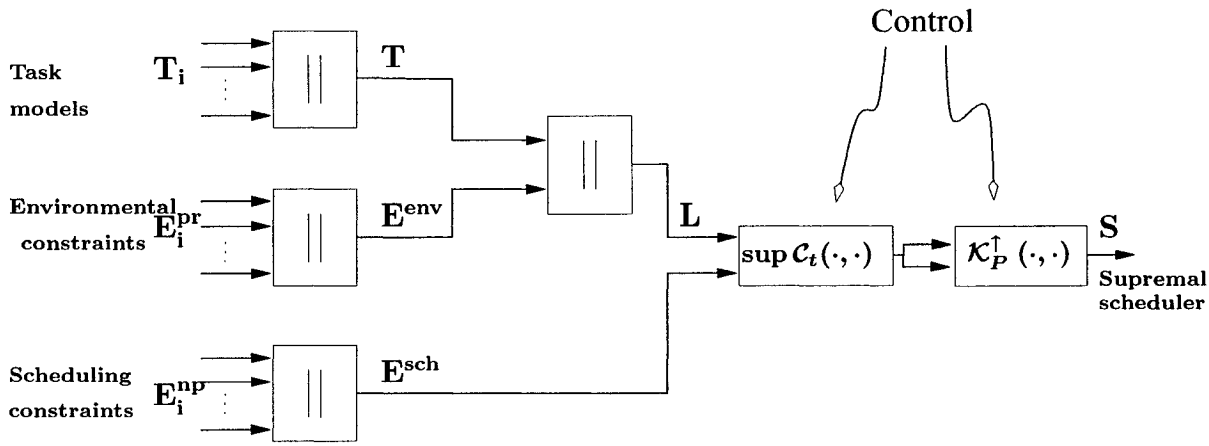

Figure 3.4: Procedure of scheduler design.

3.3 Example

Consider a water vessel as shown in Figure 3.5. Water is let into the vessel through a valve named *valve_in* and let out of the vessel by a valve named *valve_out*. The two sensors *water high* and *water low* help in indicating the level of water inside the vessel. There is also a *pump* inside the vessel useful in flushing water out of the vessel through *valve_out*. It is assumed that *valve_out* is open if and only if *pump* is on.

The normal operation of the water vessel system is as follows: Water is let into the vessel through *valve_in* when the water level is below the low level (ℓ). As soon as the water level exceeds the high level (h), the pump is used to drive the excess water out of the vessel through *valve_out*.

The various properties of the water vessel problem are:

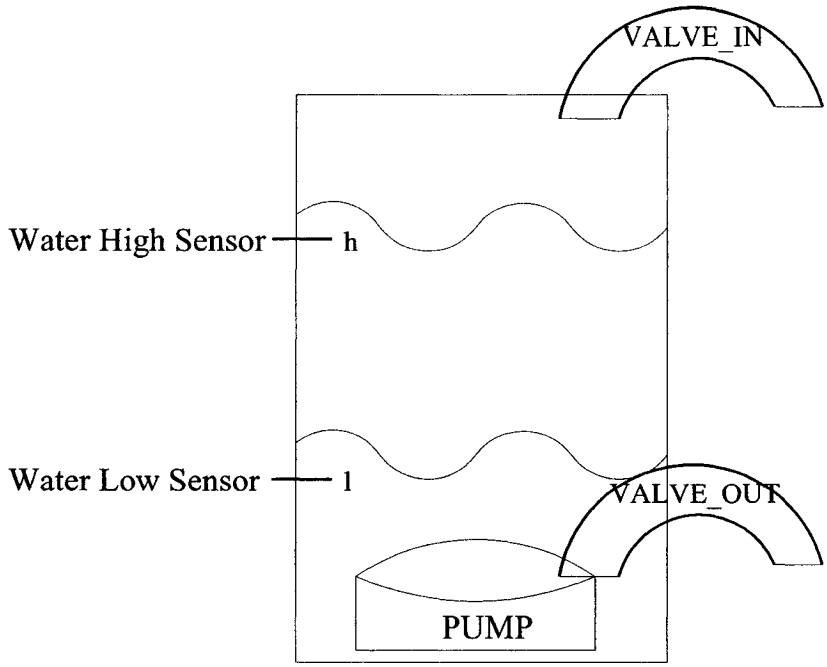

Figure 3.5: Water controller.

1. Water should not overflow.

2. Water should not underflow.

3. Positive but limited influx of water when *pump* is off and *valve_in* is open.

4. Positive but limited outflux of water when *pump* is on and *valve_in* is closed.

The above-mentioned problem could be modeled by two periodic tasks which are operated alternately by a controller or a universal scheduler. To this end let us define $\mathbf{T_1} : (4, 2)$ and $\mathbf{T_2} : (2, 1)$ to be two periodic tasks representing the conditional opening of *valve_in* and starting of *pump*, respectively, where in $\mathbf{T_i} : (\wp_i, \nu_i)$, $\wp_i$ and ν_i stand respectively for period and execution time of task i. So, when task $\mathbf{T_1}$ is being executed, *valve_in* can be opened and *pump* is off. When task $\mathbf{T_2}$ is being executed, *pump* can be turned on and *valve_in* is closed.

While periodically monitoring the water level through sensors, the periodic task $\mathbf{T_1}$ opens *valve_in* and turns *pump* off if water level is less than ℓ. The periodic task $\mathbf{T_2}$ turns the *pump* on and close *valve_in* when the water level is greater than h. According to our example, task $\mathbf{T_1}$ checks for the water level every 4 time units and if the vessel's water level is below ℓ, *valve_in* is opened for 2 time units. Similarly, task $\mathbf{T_2}$ checks for the water level every 2 time units and if the vessel's water level is above h, *pump* is switched on for 1 time unit. The scheduler to be designed is based on the timing constraints of the two tasks, namely, periodicity and execution time rather than the events triggering the two tasks. This means that the scheduler to be designed is time-driven (as opposed to event-driven). Define $\mathbf{T} := \mathbf{T_1}||\mathbf{T_2}$ and $\mathbf{E^{env}} := \mathbf{E_1^{pr}}||\mathbf{E_2^{pr}}$.

Let us now analyze the set of tasks for the following assumptions:

1. Preemptive scheduling with no priority relation.

2. Preemptive scheduling with priority relation.

3. Nonpreemptive scheduling with no priority relation.

We use the TTCT [61] software tool for analysis and verification of TDES.

3.3.1 Preemptive Scheduling

1) $P = \emptyset$

The automata for the two task models are given in Figure 3.6.

Since the alphabets are made identical, the operation `meet` in TTCT computes the synchronous product between the two task models.

$$\mathbf{T} := \texttt{meet}(\mathbf{T_1}, \mathbf{T_2})$$

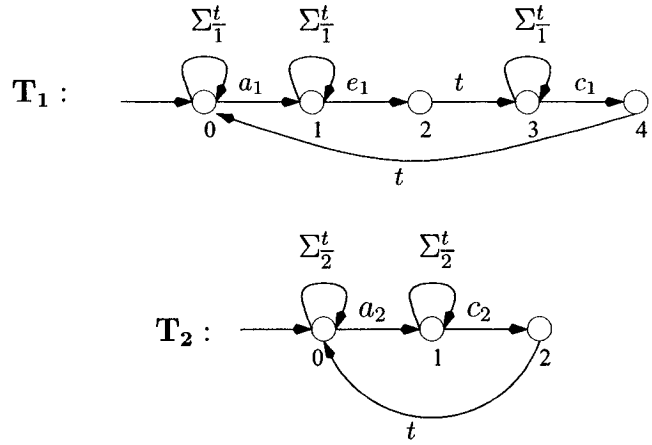

Figure 3.6: Task automata.

The periodicity constraint of the two tasks are shown in Figure 3.7.

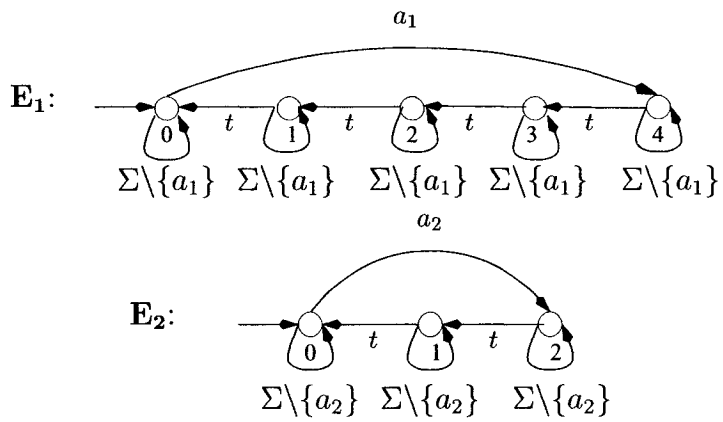

Figure 3.7: Automata for periodicity of tasks.

The environmental constraint is calculated by taking the synchronous product of the two periodicity requirements:

$$\mathbf{E^{env}} := \mathtt{meet}(\mathbf{E_1}, \mathbf{E_2})$$

With no priority relation defined we have $\mathbf{L} = \mathbf{T}||\mathbf{E^{env}}$, which is obtained by

taking the meet of the task model and the environmental constraint.

$$\mathbf{L} := \mathtt{meet}(\mathbf{T}, \mathbf{E^{env}})$$

Furthermore, there are no scheduling constraints, i.e. $E^{sch} = \Sigma^*$, and therefore

$$\mathbf{S} := \mathtt{supcon}\mathcal{C}_\mathbf{t}(\mathbf{L}, \mathbf{E^{sch}})$$

returns the largest controllable timed sublanguage of L subject to the environmental (and scheduling) constraints. The automaton **S** is shown in Figure 3.8.

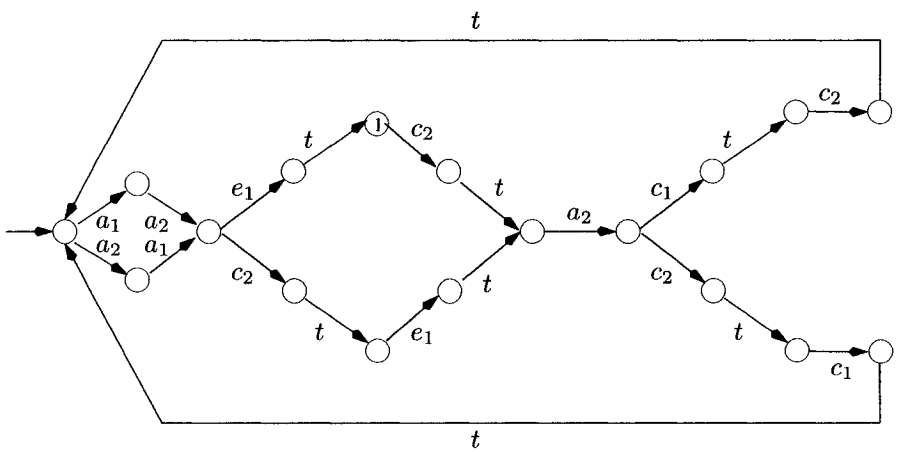

Figure 3.8: The universal scheduler for $\mathbf{T_1}$ and $\mathbf{T_2}$ ($P = \emptyset$, $E^{sch} = \Sigma^*$).

We call **S** a *universal* scheduler. The system controlled by a universal scheduler can generate all legal event sequences subject to the environmental constraints. In order to meet the timing constraint of $\mathbf{T_2}$, the execution of $\mathbf{T_1}$ is disabled (suspended) at state 1 while the execution of $\mathbf{T_2}$ is forced; in other words, at state 1 $\mathbf{T_2}$ preempts $\mathbf{T_1}$. This can be verified by the `condat` function of the TTCT.

2) $P \neq \emptyset$

Now suppose the behavior in Subsection 3.3.1 is further restricted by a priority relation. For example, let us assume that $\mathbf{T_2}$ has a higher priority of execution than $\mathbf{T_1}$. This can be represented by the priority relation P, where:

$$P := \{(c_1, c_2), (e_1, c_2)\} \tag{3.2}$$

The only change compared to the previous case is in the plant model, which is now equal to

$$\mathbf{L = T_P || E^{env}}$$

The prioritized model $\mathbf{T_P}$ is obtained by removing execution transitions of the lower priority task $\mathbf{T_1}$ whenever the execution of the higher priority task $\mathbf{T_2}$ is possible. This is shown in Figure 3.9.

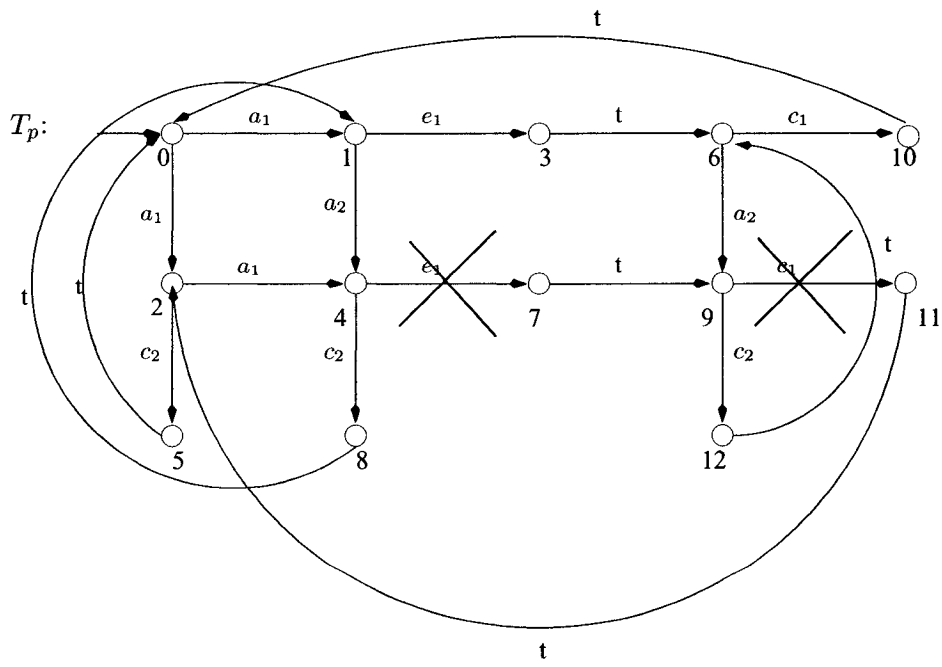

Figure 3.9: Prioritized task.

The scheduler $\mathbf{S}$ is shown in Figure 3.10. Observe that at states 1 and 2, the execution of $\mathbf{T_1}$ is disabled because the execution of $\mathbf{T_2}$ has a higher priority.

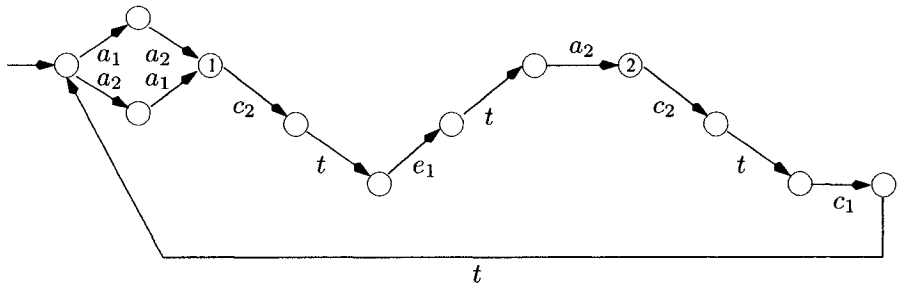

Figure 3.10: The (supremal) scheduler for $\mathbf{T_1}$ and $\mathbf{T_2}$, with $P = \{(e_1, c_2), (c_1, c_2)\}$ and when preemption is allowed.

3.3.2 Nonpreemptive Scheduling

When preemption is not allowed we have $\mathbf{E^{sch}} := \mathbf{E_1^{np}} || \mathbf{E_2^{np}}$, while as in Subsection 3.3.1 assuming that $P = \emptyset$ we have $\mathbf{L} = \mathbf{T} || \mathbf{E^{env}}$.

The automata of Figure 3.11 express the non-preemptiveness of the two tasks under investigation.

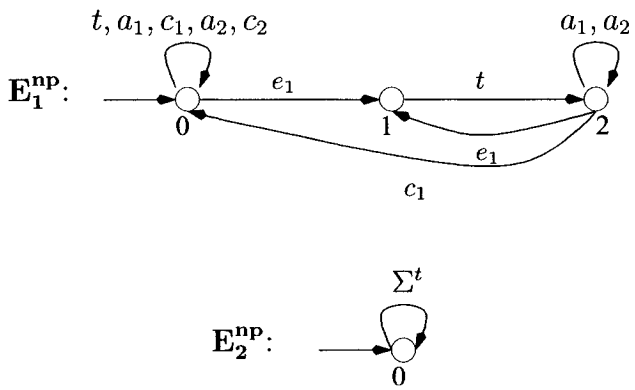

Figure 3.11: Automata expressing the non-preemptiveness of tasks.

The scheduling constraint is calculated by taking the synchronous product of the

non-preemptiveness requirement of each task.

$$\mathbf{E^{sch}} := \mathtt{meet}(\mathbf{E_1^{np}}, \mathbf{E_2^{np}})$$

The supremal scheduler

$$\mathbf{S} := \mathtt{supcon}\mathcal{C}_\mathbf{t}(\mathbf{L}, \mathbf{E^{sch}})$$

is shown in Figure 3.12.

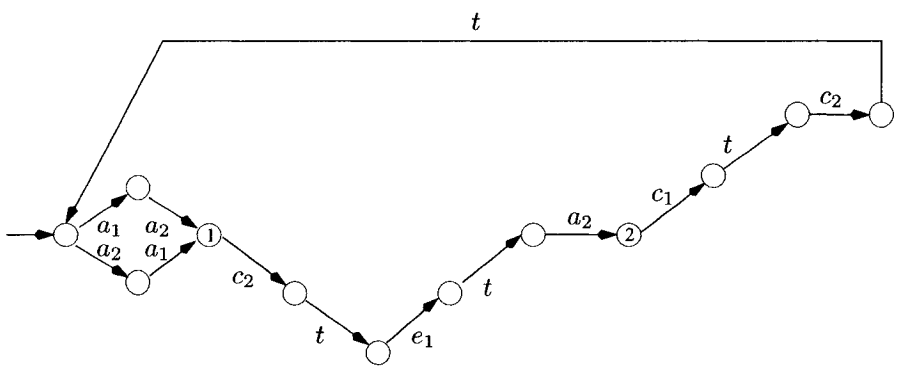

Figure 3.12: The (supremal) scheduler for $\mathbf{T_1}$ and $\mathbf{T_2}$ when preemption is not allowed ($P = \emptyset$).

Observe that when in state 1, task $\mathbf{T_2}$ would miss its deadline if task $\mathbf{T_1}$ were fully executed. Thus, e_1 is disabled in state 1. Also in state 2 the execution of $\mathbf{T_1}$ is underway, and since preemption is not allowed, the execution of task $\mathbf{T_2}$ must be disabled.

3.4 Conclusion

Through this chapter, we have introduced the concept of priority-based supervisory control of discrete-event systems. In particular, we have shown the synthesis of *uni-*

versal schedulers for hard real-time tasks on *uniprocessor* platforms by modeling and specifying tasks using discrete-timed automata. We have also provided an exhaustive example in this chapter in order to illustrate the synthesis procedure.

Chapter 4

Multiprocessor Scheduler Design

In this chapter, we present a framework for designing schedulers for hard real-time systems upon uniform multiprocessors based on Supervisory Control Theory (SCT) for timed discrete-event systems. The contribution of this work lies in the development of a formal constructive method for controlling the preemptive and migrative execution of real-time tasks on a set of uniform processors. We synthesize schedulers using the *proof by construction* approach, wherein we demonstrate the existence of a scheduler capable of scheduling tasks on multiple CPUs by providing a method for constructing such a scheduler.

The chapter is organized as follows. Section 4.1 illustrates various multiprocessor machines, and looks into specific reasons for considering *uniform* multiprocessors in our models. Section 4.2 describes a framework for modeling hard real-time tasks as TDES and for synthesizing a supervisory control for such tasks on uniform multiprocessors. Section 4.3 provides an example illustrating in detail the procedure followed in designing a scheduler under the various task requirements of Section 4.2. Finally, we conclude this chapter in Section 4.4.

4.1 Types Of Multiprocessors

As the name indicates, in multiprocessor platforms, one can execute tasks on one of the many processors available for utilization. In most of the early work in the literature on real-time scheduling on multiprocessors, it was assumed that all processors were identical in terms of their processing/computing capacities. However, recent work published by scheduling theorists identify two different kinds of multiprocessors based on their computing capacities:

1. Identical Multiprocessors: Identical multiprocessors are those in which all the processors have the same computing capacity.

2. Uniform Multiprocessors: In uniform multiprocessors, each processor is characterized by its own computing capacity, with the interpretation that a task that executes on a processor of computing capacity $\mathbf{s}$ for τ time units completes $\mathbf{s}\tau$ units of execution.

We have considered the uniform multiprocessors in our work since we believe that they could be used for designing many practical application systems in the real world. Furthermore, using this kind of multiprocessors during modeling provides the application system designer the freedom to use processors of different speeds, rather than constraining him/her to always use identical processors.

Further, as technology is fast improving, newer and faster processors are developed all the time. In order to improve the performance of a system, it is crucial to upgrade the system with faster processors. If we have uniform multiprocessors model, it would be easier to just replace the slower processors with the new faster ones, and keep the ones that are sufficiently fast intact. Whereas, if our model was identical multiprocessors, it becomes imperative to change all the processors simultaneously.

Assumptions

While designing schedulers for hard real-time systems upon uniform multiprocessor systems, we have made the following assumptions on task execution:

1. Task preemption is permitted, meaning that a task executing on a particular processor may be preempted prior to the completion of its execution, and its execution may be resumed later.

2. Task migration is permitted, meaning that a task that has been preempted on a particular processor may resume its execution on the same or a different processor.

3. Task parallelism is forbidden, meaning that each task must execute on at most one processor at any given instant in time.

In the next section, we revisit the timing model of our work as was explained in the previous chapter. We model real-time tasks and specifications using a simple extension of automata called discrete timed automata (DTA).

4.2 Timing Model

This section closely parallels Section 3.2; some parts are repeated for completeness' sake. Time in our work is measured in discrete steps. In order to account for real-time systems, time always has to have a chance to advance without ever being blocked. We call this the *time-progress* property of DTA, which excludes non-realistic executions where time does not have a chance to advance. A DTA is considered to satisfy the time-progress property if

$$\forall x \in X \exists u \in \Sigma^*.\ \delta(x, ut)!$$

Let L be a prefix-closed language over Σ^t. We call L a *Timed Language (TL)* if

$$\forall s \in L \exists u \in \Sigma^*.\ sut \in L$$

It is immediate from the definitions that if $\mathbf{L}$ is a DTA satisfying the time-progress property then the language generated by $\mathbf{L}$, denoted by L, is a timed language.

An important property of timed languages is that they are closed under arbitrary union. This property is essential when the language returned by a control algorithm is not a timed language. Then this property guarantees that a largest timed sublanguage of the behavior exists. Formally, for $E \subseteq \Sigma^{t*}$ let

$$\mathcal{T}(E) := \{K \subseteq E | K = \overline{K} \wedge K \text{ is a timed language}\}$$

Proposition 6 *The supremum of $\mathcal{T}(E)$, denoted by $\mathcal{T}^{\uparrow}(E)$, exists and belongs to $\mathcal{T}(E)$.*

Proof can be found in [41].

Let E' denote the largest timed sublanguage of E. Then it can be readily shown that the DTA $\mathbf{E}'$ obtained from $\mathbf{E}$ by recursively removing all states from which time does not have a chance to advance satisfies the time-progress property.

In [62] it is shown that the class of controllable and prefix-closed sublanguages of a given language is also closed with respect to set union, resulting in the following corollary.

Corollary 7 *Let*

$$\mathcal{C}_t(L, E) := \{K \subseteq E \Big| K = \overline{K} \wedge K \textit{ is cont. w.r.t. } L \wedge K \textit{ is TL}\}$$

Then $\mathcal{C}_t(L, E)$ has a largest element, which we denote by $\sup \mathcal{C}_t(L, E)$.

To calculate $\sup \mathcal{C}_t(L, E)$, we first find the supremal controllable sublanguage of E [62]. If the result is a timed language, the procedure terminates. Otherwise, we obtain its largest timed sublanguage by recursively removing all states of $\sup \mathcal{C}(L, E)$ from which time has no chance to advance, and repeat the above procedure if necessary. The fact that this iterative procedure computes the largest timed sublanguage of E that is controllable and prefix-closed is demonstrated below.

Let p and q be predicates on languages over Σ, that is $p, q : pwr(\Sigma^*) \to \{T, F\}$. For $E \subseteq \Sigma^*$ define

$$P(E) := \{K \subseteq E |\ p(K)\}$$
$$Q(E) := \{K \subseteq E |\ q(K)\}$$

Note that

$$P(E) \cap Q(E) = \{K \subseteq E |\ p(K) \wedge q(K)\}.$$

Assume that each predicate is closed under arbitrary union. It follows that

$$P(E)^\uparrow \in P(E),\ Q(E)^\uparrow \in Q(E) \text{ and } (P(E) \cap Q(E))^\uparrow \in P(E) \cap Q(E)$$

where for $\mathcal{K} \subseteq pwr(\Sigma^*)$ the supremum of $\mathcal{K}$, denoted by $\mathcal{K}^\uparrow$, is the union of all members of $\mathcal{K}$, that is

$$\mathcal{K}^\uparrow = \bigcup_{K \in \mathcal{K}} K.$$

We would like to show that if we have procedures Ψ_p and Ψ_q to compute $P(E)^\uparrow$ and $Q(E)^\uparrow$, respectively, then $(P(E) \cap Q(E))^\uparrow$ can be computed by iteratively applying $\Psi = \Psi_p \circ \Psi_q$ on E until a fixed-point is reached.

To this end let

$$\Psi_p : pwr(\Sigma^*) \to pwr(\Sigma^*) : E \mapsto P(E)^\uparrow$$

$$\Psi_q : pwr(\Sigma^*) \to pwr(\Sigma^*) : E \mapsto Q(E)^\uparrow$$

and $\Psi := \Psi_p \circ \Psi_q$.

Lemma 8 *The maps Ψ_p, Ψ_q and Ψ are contractive and order-preserving.*

Proof. Contractiveness of Ψ_p is immediate from the definition. To prove that it is order-preserving, let $E_1 \subseteq E_2$. We must show that $\Psi_p(E_1) \subseteq \Psi_p(E_2)$, or $P(E_1)^\uparrow \subseteq P(E_2)^\uparrow$, i.e. $P(E_2)^\uparrow$ is an upperbound for $P(E_1)$.

Let $K \in P(E_1)$, i.e. $K \subseteq E_1$ and $p(K)$ (is true). It follows that $K \subseteq E_2$ and $p(K)$, i.e. $K \in P(E_2)$, and thus $K \subseteq P(E_2)^\uparrow$, i.e. we have shown that $P(E_2)^\uparrow$ is an upperbound for $P(E_1)$, as desired.

To show that Ψ is order-preserving, let $E_1 \subseteq E_2$. Then

$$\begin{aligned} \Psi(E_1) &= \Psi_p(\Psi_q(E_1)) \\ &\subseteq \Psi_p(\Psi_q(E_2)) \quad (\Psi_p \text{ and } \Psi_q \text{ are order-preserving}) \\ &= \Psi(E_2) \end{aligned}$$

■

Lemma 9 *For $E \subseteq \Sigma^*$ let*

$$fix\Psi(E) = \{M \subseteq E |\ \Psi(M) = M\}$$

We have $fix\Psi(E) = P(E) \cap Q(E)$ and therefore $\nu\Psi(E) = (P(E) \cap Q(E))^\uparrow$, where

$\nu\Psi(E)$ denotes the greatest fixed-point of Ψ that is smaller than E.

Proof. ($\subseteq$) Let $M \subseteq E$ be a fixed-point of Ψ, i.e.

$$\Psi_p \circ \Psi_q(M) = M$$

We argue that M is also a fixed-point of Ψ_p and Ψ_q. Since Ψ_p and Ψ_q are contractive maps we have:

$$M = \Psi_p(\Psi_q(M)) \subseteq \Psi_q(M) \subseteq M$$

Thus $\Psi_q(M) = M$, which in turn implies $\Psi_p(M) = M$. It follows that

$$M = \Psi_p(M) = P(M)^{\uparrow} \in P(M) \subseteq P(E)$$

Similarly, $M \in Q(E)$ and therefore $M \in P(E) \cap Q(E)$.

($\supseteq$) Let $M \in P(E) \cap Q(E)$. It follows that $M \subseteq E$, and that $p(M)$ and $q(M)$ are true. Since M is an upperbound of $P(M)$ and $M \in P(M)$, it follows that $\Psi_p(M) = P(M)^{\uparrow} = M$, i.e. M is a fixed-point of Ψ_p. Similarly, we have that $\Psi_q(M) = M$. It follows that

$$\Psi(M) = \Psi_p(\Psi_q(M)) = \Psi_p(M) = M$$

i.e. $M \in \text{fix}\Psi(E)$. ■

Theorem 10 *Let the sequence $\{\Psi^i(E)\}_{i\in\mathbb{N}}$ terminates at $i = i^*$, i.e. $\Psi^{i^*+1}(E) = \Psi^{i^*}(E)$. Then Ψ has a greatest fixed-point that is smaller than E, and $\nu\Psi(E) = \Psi^{i^*}(E)$.*

Proof. We have:

$$\Psi\Psi^{i^*}(E) = \Psi^{i^*}(E)$$

i.e. $\Psi^{i^*}(E)$ is a fixed-point of Ψ. It remains to show that if $K \subseteq E$ is any fixed-point of Ψ then we have $K \subseteq \Psi^{i^*}(E)$. Since $K \subseteq E$, applying Ψ to both sides of the inequality i^* times will yield:

$$K = \Psi^{i^*}(K) \subseteq \Psi^{i^*}(E)$$

That completes the proof. ■

4.2.1 Basic Task Model

In this subsection we study a system consisting of n tasks to be scheduled on m uniform processors. Let $I = \{1, 2, \ldots, n\}$ be an index set, and let $\mathbf{T_i}$ denote a task with an execution time of $\nu_i \in \mathbb{N}$ time units. We use the TTCT [61] software tool for analysis and verification of timed discrete-event systems. Also, the function names `meet` and `sup` written in typeset font in this work are procedure calls in TTCT.

Each task is modeled by a DTA, so that the *composite task model* can be described as:

$$\mathbf{T} := \Big|\Big|_{i \in I} \mathbf{T_i}$$

where || denotes *meet* [62].

A task consists of several identical *instances*. Arrival of an instance of a task $\mathbf{T_i}$ is denoted by an event a_i called *arrival of the task* $\mathbf{T_i}$. We divide execution of an instance into ν_i *segments*, where each segment has a duration of one time unit. We denote the execution of segment j, $j \in [1, \nu_i)$, of any instance of task i by an event e_i. We introduce a new symbol c_i to denote the execution of the last segment of $\mathbf{T_i}$. The

complete model of a task $\mathbf{T_i}$ is shown in Fig. 4.1. Note that suffix j has been added to e_i and c_i to indicate the execution of task $\mathbf{T_i}$ on processor $j \in J$, and a special event n is used to "reset" the processors, i.e. allow a processor to return to its initial state when execution of the tasks are all complete.

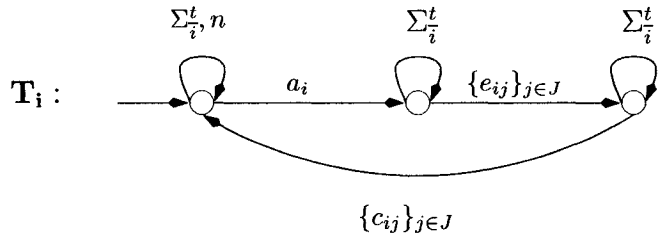

Figure 4.1: Complete model of task i (ν_i=2).

For future reference, relevant alphabets are defined in Table 4.1. As before, t superscripts an alphabet's name when it is included in the alphabet.

Sets Of Events	Description
$\Sigma_i := \{a_i, e_{ij}, c_{ij}\}_{j\in J}$	Set of all events of task i.
$\Sigma_{i,f} := \{e_{ij}, c_{ij}\}_{j\in J}$	Set of forcible events of task i.
$\Sigma_{i,c} := \{e_{ij}, c_{ij}\}_{j\in J}$	Set of controllable events of task i.
$\Sigma_{i,u} := \{a_i\}$	Set of uncontrollable events of task i.
$\Sigma := \bigcup_{j=1}^{n} \Sigma_j$	Set of events of all tasks.
$\Sigma_{\bar{i}} := \Sigma \backslash \Sigma_i$	Set of events of all but task i.

Table 4.1: Event sets used for multiprocessor scheduler design.

4.2.2 Task Requirements

Release Time

The tasks to be scheduled on the given set of uniform processors are released at different instances of time. For example, the DTA of Fig. 4.2 specifies that the only instance of $\mathbf{T_i}$ arrives after two time units.

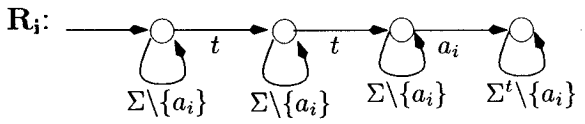

Figure 4.2: Specification of $\mathbf{T_i}$ with release time of two time units.

The *composite release time* model can be described as:

$$\mathbf{R} := \Big|\Big|_{i \in I} \mathbf{R_i}$$

Resource Requirements

Since we have considered uniform multiprocessors (CPUs), the DTA should depict the varying capacities (speeds) of the CPUs. The DTA shown in Fig. 4.3 specifies a CPU indexed j with speed $\mathbf{s} = 2$.

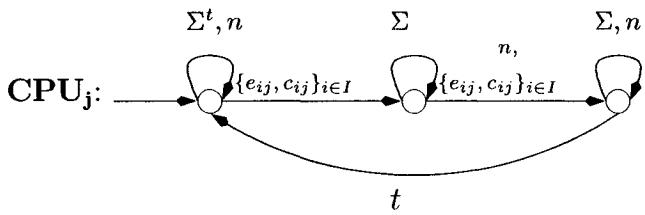

Figure 4.3: Specification for a CPU_j with $\mathbf{s} = 2$.

Resource requirements can be expressed by a specification automaton $\mathbf{E^{res}}$ defined as:

$$\mathbf{E^{res}} := \Big|\Big|_{j \in J} \mathbf{CPU_j}$$

where $J = \{1, 2, \ldots, m\}$ is the index set of all CPUs.

Deadline Requirements

Since hard real-time tasks are studied in our work, each of these tasks is characterized by a stringent deadline. The deadline requirement can be expressed by an automaton $\mathbf{E^{sch_i}}$ (*scheduling constraint*) as shown in Fig. 4.4.

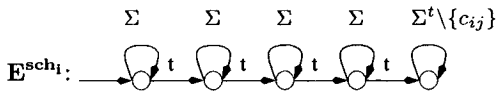

Figure 4.4: Specification for $\mathbf{T_i}$ with deadline of four time units.

Deadline requirements are expressed by a specification automaton $\mathbf{E^{sch}}$ defined as:

$$\mathbf{E^{sch}} := \Big|\Big|_{i \in I} \mathbf{D_i}$$

According to Corollary 7, a supremal scheduler, denoted by $\mathbf{S}$ exists and is given by

$$\mathbf{S} := \sup \mathcal{C}_\mathbf{t}(\mathbf{L}, \mathbf{E^{sch}})$$

where $\mathbf{L} := \mathbf{T}||\mathbf{R}||\mathbf{E^{res}}$.

The entire design procedure is depicted in Fig. 4.5.

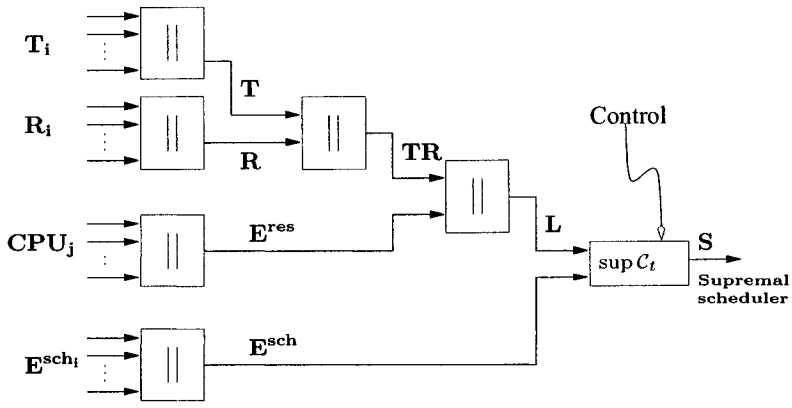

Figure 4.5: Procedure of scheduler design.

4.3 Example

Consider $\mathbf{T_1} : (2, 3, 4)$, $\mathbf{T_2} : (1, 5, 5)$ and $\mathbf{T_3} : (0, 8, 7)$ to be three hard real-time tasks to be scheduled on two CPUs named $\mathbf{CPU_1}$ and $\mathbf{CPU_2}$ of speeds 3 and 2, respectively. In $\mathbf{T_i} : (r_i, \nu_i, d_i)$, r_i, ν_i and d_i denote respectively the release time, execution time and deadline of task i. A CPU speed of 3 means that any task running on it can execute 3 segments in one unit of time. Define $\mathbf{T} := \mathbf{T_1}||\mathbf{T_2}||\mathbf{T_3}$, $\mathbf{R} := \mathbf{R_1}||\mathbf{R_2}||\mathbf{R_3}$ and $\mathbf{E^{res}} := \mathbf{CPU_1}||\mathbf{CPU_2}$.

Tasks are modeled as in Fig. 4.6, Fig. 4.7 and Fig. 4.8.

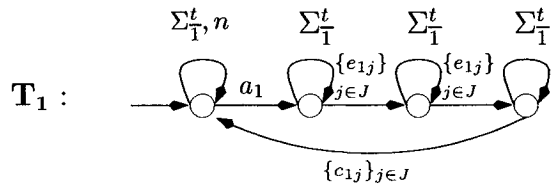

Figure 4.6: Task T_1's automaton.

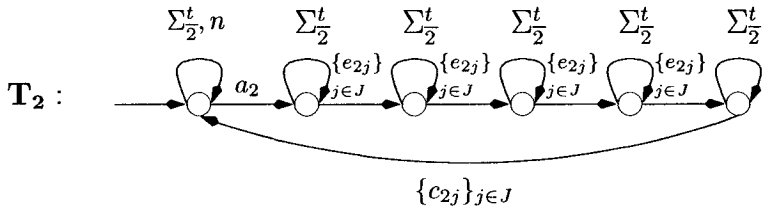

Figure 4.7: Task T_2's automaton.

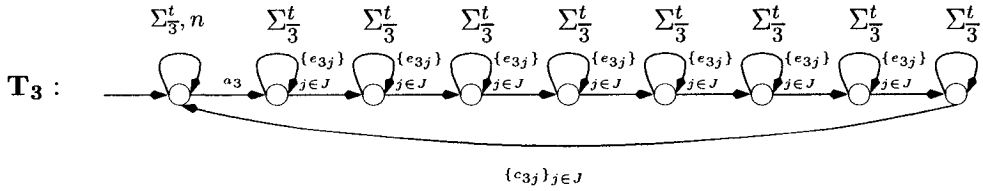

Figure 4.8: Task T_3's automaton.

Define:

$$\mathbf{T_{12}} := \texttt{meet}(\mathbf{T_1}, \mathbf{T_2}); \mathbf{T} := \texttt{meet}(\mathbf{T_{12}}, \mathbf{T_3}).$$

The release time specifications for $\mathbf{T_1}$ and $\mathbf{T_2}$ are shown in Fig. 4.9 and Fig. 4.10 respectively.

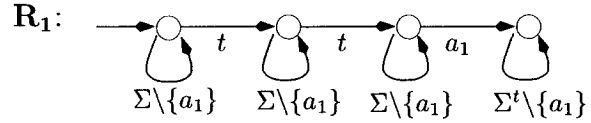

Figure 4.9: Specification for the release time (r_1) of $\mathbf{T_1}$.

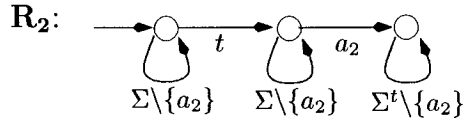

Figure 4.10: Specification for the release time (r_2) of $\mathbf{T_2}$.

Since the release time of $\mathbf{T_3}$ is zero, the release time constraint is computed by taking the *meet* of the release times of $\mathbf{T_1}$ and $\mathbf{T_2}$:

$$\mathbf{R} := \texttt{meet}(\mathbf{R_1}, \mathbf{R_2})$$

The task model with the release time constraints is:

$$\mathbf{TR} := \texttt{meet}(\mathbf{T}, \mathbf{R})$$

The resource constraints of the individual CPUs can be modeled according to the automaton of Fig. 4.3. $\mathbf{CPU_1}$ and $\mathbf{CPU_2}$ are modeled by the automata of Fig. 4.11 and Fig. 4.12, respectively.

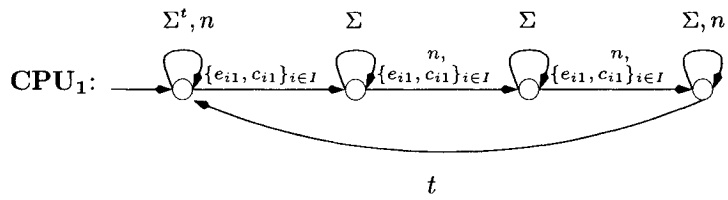

Figure 4.11: Specification for $\mathbf{CPU_1}$ with $s = 3$.

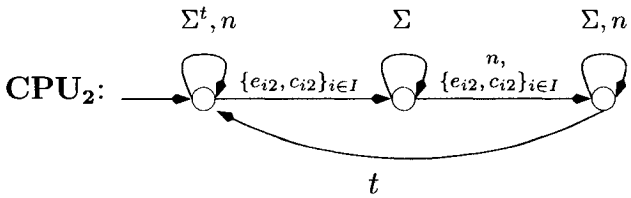

Figure 4.12: Specification for $\mathbf{CPU_2}$ with $s = 2$.

The resource constraint is computed by taking the *meet* of the CPU requirements:

$$\mathbf{E^{res}} := \mathtt{meet}(\mathbf{CPU_1}, \mathbf{CPU_2})$$

Now the plant is given by:

$$\mathbf{L} := \mathtt{meet}(\mathbf{TR}, \mathbf{E^{res}})$$

The scheduling constraint is calculated by taking the *meet* of the deadlines of the three tasks. Thus, following the model of Fig. 4.4, the specification for deadlines of the three tasks are expressed by the automata of Fig. 4.13, Fig. 4.14 and Fig. 4.15.

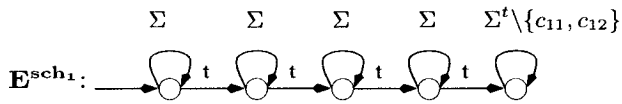

Figure 4.13: Specification for deadline of $\mathbf{T_1}$.

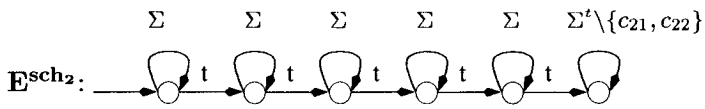

Figure 4.14: Specification for deadline of $\mathbf{T_2}$.

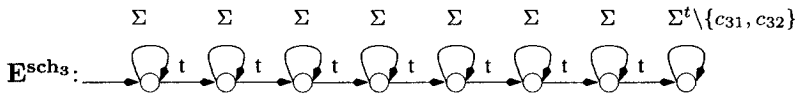

Figure 4.15: Specification for deadline of $\mathbf{T_3}$.

Define:

$$\mathbf{E}^{\mathbf{sch_{12}}} := \mathtt{meet}(\mathbf{E}^{\mathbf{sch_1}}, \mathbf{E}^{\mathbf{sch_2}}); \mathbf{E}^{\mathbf{sch}} := \mathtt{meet}(\mathbf{E}^{\mathbf{sch_{12}}}, \mathbf{E}^{\mathbf{sch_3}}).$$

After modeling the tasks and their scheduling requirements, we compute

$$\mathbf{S} := \mathtt{sup}\mathcal{C}_\mathtt{t}(\mathbf{L}, \mathbf{E}^{\mathbf{sch}})$$

which returns the largest controllable timed sublanguage of $E^{sch} \cap L$ subject to the scheduling and resource constraints.

One of many feasible schedules contained in $\mathbf{S}$ is shown in Fig. 4.16. Recall that the computing capacities of the two processors are different. $\mathbf{CPU_1}$ has a processing speed of 3 while that of $\mathbf{CPU_2}$ is 2. Also, the arrival of tasks, denoted by a_1, a_2 and a_3, do not take processor time. Only the actual execution of tasks cause the advancement of processor time. The upward and downward dotted arrows indicate task migration between the two processors.

The corresponding Gantt diagram is shown in Fig. 4.17.

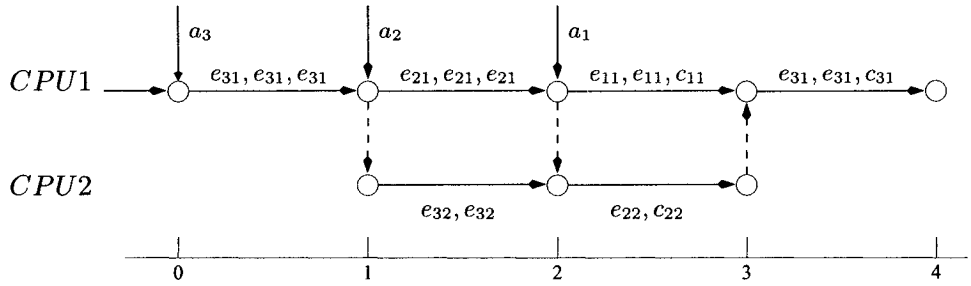

Figure 4.16: Multiprocessor scheduler.

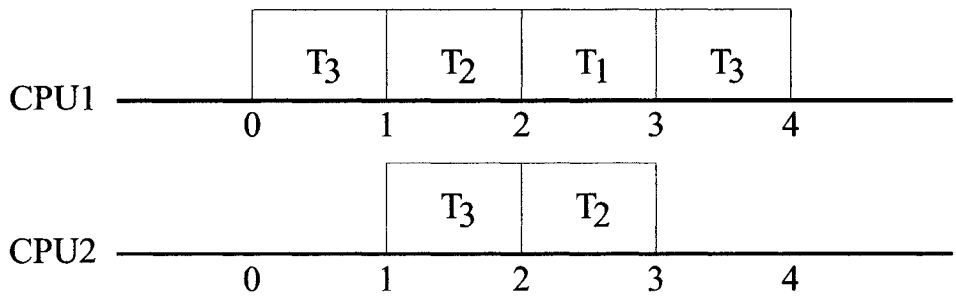

Figure 4.17: The (supremal) scheduler for $\mathbf{T_1}$, $\mathbf{T_2}$ and $\mathbf{T_3}$.

4.4 Conclusion

In this chapter, we have applied our framework (from the previous chapter) based on the theory of supervisory control of discrete-event systems on *uniform multiprocessor* systems. In that respect, we have presented the synthesis procedure for schedulers of real-time tasks on uniform multiprocessor platforms. A comprehensive example has also been provided in this chapter to illustrate the scheduler synthesis procedure. In the next chapter, we try to alleviate the state space explosion problem we had come across in our scheduler design procedure (due to discrete-time model of the real-time tasks) by applying a modified form of symbolic analysis.

Chapter 5

Modified Symbolic Scheduler Design

We consider a modified form of symbolic modeling methodology [24] to alleviate some of the state explosion problems we had faced while designing schedulers for real-time systems using supervisory control of discrete-event systems framework [39, 41]. In this chapter we attempt, rather informally, to address the problem of state space explosion in scheduler synthesis using symbolic methods. We first define some of the basic concepts about region and simulation graphs in Section 5.1, and then provide a framework for scheduler design with reduced state space in Section 5.2. In Section 5.3, we explain the modeling of real-time systems and specifications based on timed automata under non-preemptive assumption. A comprehensive example illustrating in detail the procedure followed in designing a scheduler for non-preemptive real-time tasks with reduced state space is presented in Section 5.4. Even though we utilize the *proof by construction* approach for scheduler synthesis, in the absence of a formal design procedure for reduced state space schedulers, we employ the concept of *zone automata* [7, 10, 11] to check the correctness of the synthesized scheduler. In

Section 5.5, we explain the scheduler design under preemptive conditions. Finally, we conclude this chapter in Section 5.6.

5.1 Basic Definitions

5.1.1 Symbolic Discrete Timed Automaton

In order to reduce the state space in our scheduler design, we have incorporated a slightly modified version of discrete timed automaton (DTA) to model a real-time task in this chapter. We refer to it as *symbolic* DTA and is expressed as a four-tuple $\mathbf{TA} := (C, S, \Sigma, E)$. In the tuple, C is a set of timer variables, which in our task model are variables denoting *periods (p)* and *execution times (c)* of the tasks. S in the tuple is a set of states and Σ is a finite set of events. Let T_C be the set of timing constraints over C (that is, completion of execution times of tasks within their periods). E in the tuple for **TA** is a finite set of transitions of the form $e = (s, g, c, s')$, where $s, s' \in S$ are the initial and final states, respectively, $g \in T_C$ is a guard on the transition between states and $c \subseteq C$ is a set of timer variables to be reset.

Consider a real-time task **T** with execution time 2 and period 3 to be modeled as a symbolic DTA. Fig. 5.1 illustrates the task automaton model.

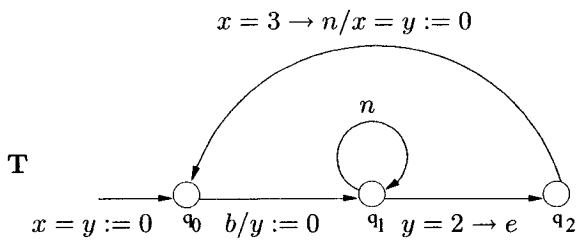

Figure 5.1: Symbolic DTA of task **T**.

In Fig. 5.1, $(x, y) \in C$ are the timer variables representing the execution time and

period of task **T**. $S = \{q_0, q_1, q_2\}$ is the set of states and $\Sigma = \{b, e, n\}$ is a finite set of events, where b and e denote respectively the beginning and the end of execution of a task, and n is an event that marks the beginning of a new cycle for the current or other tasks in the system. The guard g is assigned based on the timer variables of **T**. For example, in the figure, the guard on the transition from state q_2 to q_0 is $x = 3$, meaning that event n would occur only when the period of task **T** (denoted in the figure as x) is reached. A more detailed explanation of the various events of Fig. 5.1 is given in Section 5.3. The equivalent discrete timed automata for task **T** in which the passage of time is explicit is shown in Fig. 5.2.

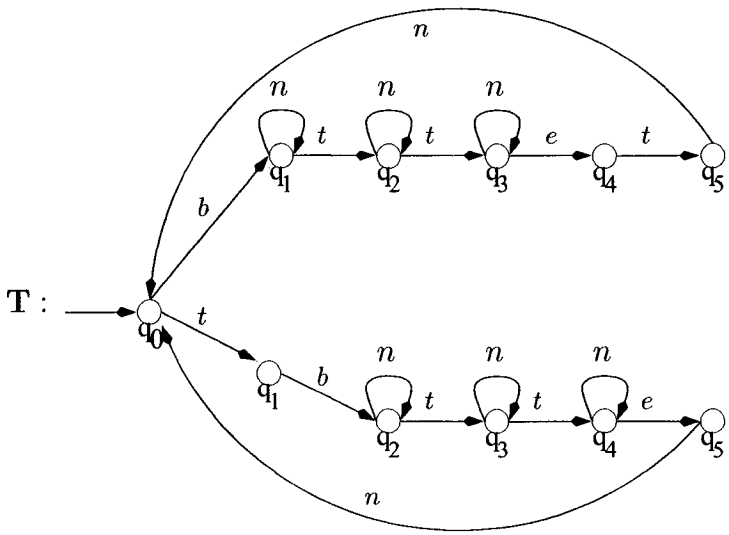

Figure 5.2: Equivalent DTA of task **T**.

5.1.2 Region Graph

The region graph [24] is defined as a quotient structure that is induced by an equivalence relation on the timed states of a timed automaton. Two timed states are considered equivalent if they have the same control location, with all their clock values (timer variables) matching on their integral parts and have the same ordering

of their fractional parts. Clocks that exceed a certain value, which can be taken as the maximal constant in the description of the automaton, are considered equivalent. In region graph, a *node* is referred to as a *region* having a set of equivalent states. Formally, a region graph can be defined as follows.

Definition 4 Let **TA** be a symbolic timed automaton, S be a finite set of states, C be a finite set of timer variables with $|C| =: N$, and T_C be the set of timing constraints over C. Let k_i be the (largest) constant to which $c_i \in C$ is compared, $i = 1, 2, \ldots, N$. The *region graph* of **TA**, denoted by $RG(\mathbf{TA})$, is a finite graph, in which nodes are regions given as $r = (s, \underline{v})$, where $s \in S$ and $\underline{v} \in \prod_{i=1}^{N}[0, k_i]$ is the vector of values of variables in C, and edges are transitions between the nodes which denote the passage of one unit of time (a *tick*of the global clock). □

Fig. 5.3 shows the region graph for the symbolic DTA of Fig. 5.1.

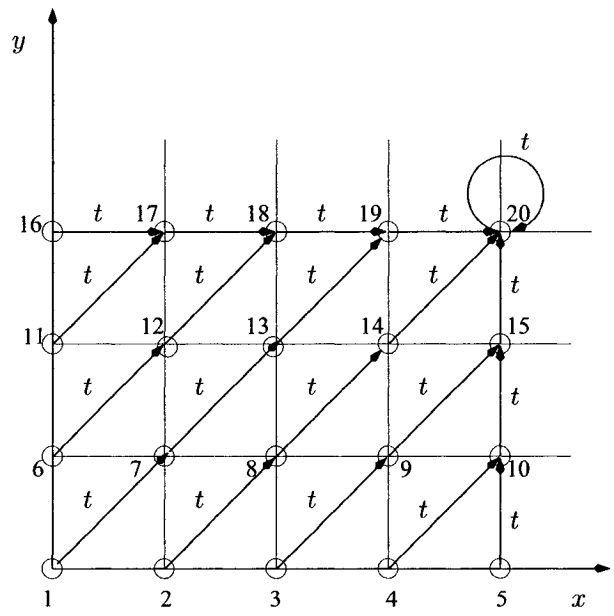

Figure 5.3: Region graph of task **T**.

The timer variables of task **T**, namely, execution time and period denoted by x and y, form the axes in the region graph. The different regions in the graph denote the values of timer variables in the symbolic DTA. For example, region 6 denotes $x = 0$ and $y = 1$, meaning that with one time unit has already gone by, **T** has been executed for one time unit. As the graph indicates, a *tick* of the global clock moves the graph from region 6 to region 12. As another example, in region 16, the timer variable y has reached its limit, and thus subsequent *tick*s do not increment the value of y.

5.1.3 Modified Symbolic Graph

The size of a region graph is exponential in the sizes of constants in a timed system. In order to reduce the number of regions from the region graph, we have come up with a graph that groups identical states (with different timing parameters) into a single region set. We refer to it as the *modified symbolic graph.* It is constructed by combining the different event transitions portrayed in the symbolic DTA model with the timing representations of the region graph. In other words, the modified symbolic graph is obtained by considering the different transitions that could be traversed based on the various events of Fig. 5.1 while satisfying the timing constraints.

Fig. 5.4 enumerates the modified symbolic graph of task **T**. It consists of the various states as in Fig. 5.1 along with the values of the timer variables for period and execution time from the region graph. For example, $(q_0, 1)$ means that at state q_0 the system is in state q_0 of the DTA and region 1 of the region graph, i.e. the timer variables of task **T**, namely, x and y, are equal to zero, i.e. they have just been reset. All nodes or regions (as in region graph) having the same state component are clubbed together into a region set. For example, in Fig. 5.4, different nodes having

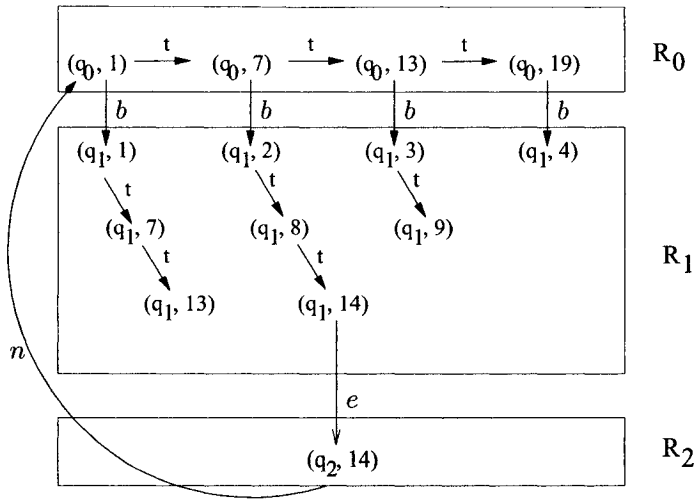

Figure 5.4: Modified symbolic graph of task **T**.

state q_0 are clubbed into a single region set R_0. This considerably reduces the number of regions in our scheduler design.

5.1.4 Simulation Graph

The modified symbolic graph helped us in reducing the number of regions from what we had in the region graph. But still, the number of nodes in the modified symbolic graph were large because of the explicit representation of "tick (t)". In [24], the authors have illustrated a methodology based on symbolic modeling of tasks, that reduces the size of the system state space caused by region graphs. The authors call this the *simulation graph*, where nodes are "region sets" and only discrete transitions are explicit, while time passes implicitly inside the nodes, meaning that "tick (t)" could be removed from the graph altogether. We incorporate the concept of simulation graph from [24] in our work in order to reduce the number of nodes created because of the explicit "tick (t)" transition. The simulation graph is formally defined as follows.

Definition 5 Let **TA** be a symbolic timed automaton and R_0 be an initial region set,

then the *simulation graph* $SG(\mathbf{TA}, R_0)$ is the graph reachable from R_0 by computing regions reached through a finite sequence of timed transitions, followed by discrete transitions. □

Fig. 5.5 illustrates the simulation graph of task **T**.

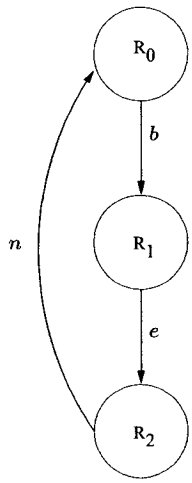

Figure 5.5: Simulation graph of task **T**.

5.1.5 Pre-Stable Condition

Further reduction of state space in our scheduler design is achieved by applying the *pre-stable condition* given in [24] on the modified symbolic graph (Fig. 5.4). Formally, the condition is defined below.

Definition 6 Given two region sets R and R', and a transition T, the region set R is said to be pre-stable with respect to R' for T iff for every node r_1 in region set R there exists a node r_2 in region set R' such that r_2 is a T-successor of r_1, i.e. it can be reached by a sequence of tick steps, followed by a T transition. □

We refer to the graph obtained by applying the pre-stable condition on modified symbolic graph as *pre-stable modified symbolic graph*, which is shown in Fig. 5.6. In

Fig. 5.6, region set R_0' is a subset of region set R_0 of Fig. 5.4.

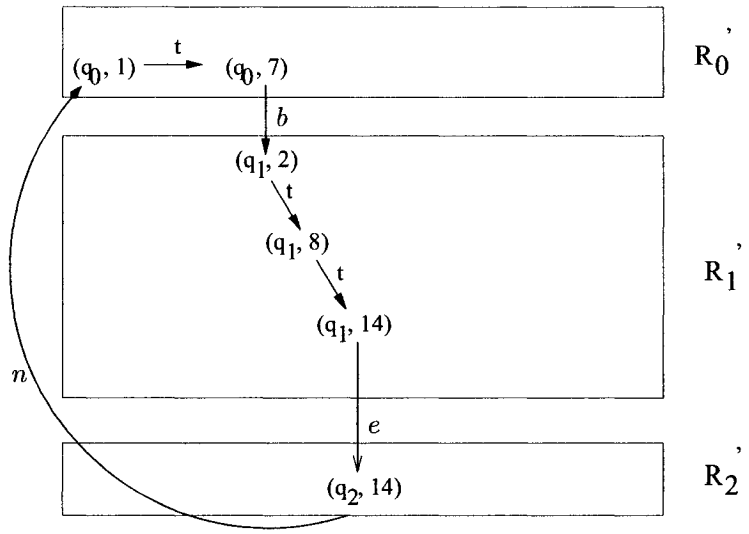

Figure 5.6: Pre-stable modified symbolic graph of task **T**.

The corresponding pre-stable simulation graph is presented in Fig. 5.7.

5.2 Framework for Scheduler Design with Reduced State Space

Our framework for scheduler design with reduced state space based on modified symbolic and simulation graphs is shown in Fig. 5.8. The real-time tasks (for simplicity of presentation we consider only two tasks $\mathbf{T_1}$ and $\mathbf{T_2}$) to be scheduled on a single processor are first modeled as timed automata and then synchronized to get a composed automaton **T**.

The operation sync in TTCT [61] computes the synchronous product between the task models.

$$\mathbf{T} := \mathtt{sync}(\mathbf{T_1}, \mathbf{T_2})$$

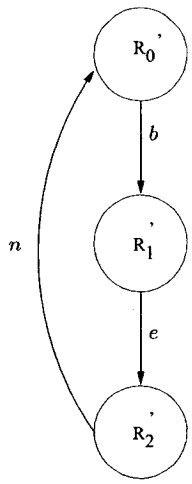

Figure 5.7: Pre-stable simulation graph of task **T**.

The composed automaton is then synchronized with the specification automaton to arrive at the automaton **TS**.

$$\mathbf{TS} := \texttt{meet}(\mathbf{T}, \mathbf{S})$$

The modified symbolic graph is obtained by considering the different transitions that could be traversed based on the various events of the composed task automaton. The modified symbolic graph of the composed model of $\mathbf{T_1}$ and $\mathbf{T_2}$ consists of the various states of **TS** along with the values of timer variables for period and execution time of the two tasks.

The simulation graph is the graph that computes the regions reached from a region set by a finite sequence of timed transitions followed by a task transition, and is finite since there is a finite number of regions. From the modified symbolic graph, we arrive at its simulation graph by removing explicit passages of time (or tick events) and explicitly representing task transitions only.

In order to reduce the state space in the composed tasks' modified symbolic graph,

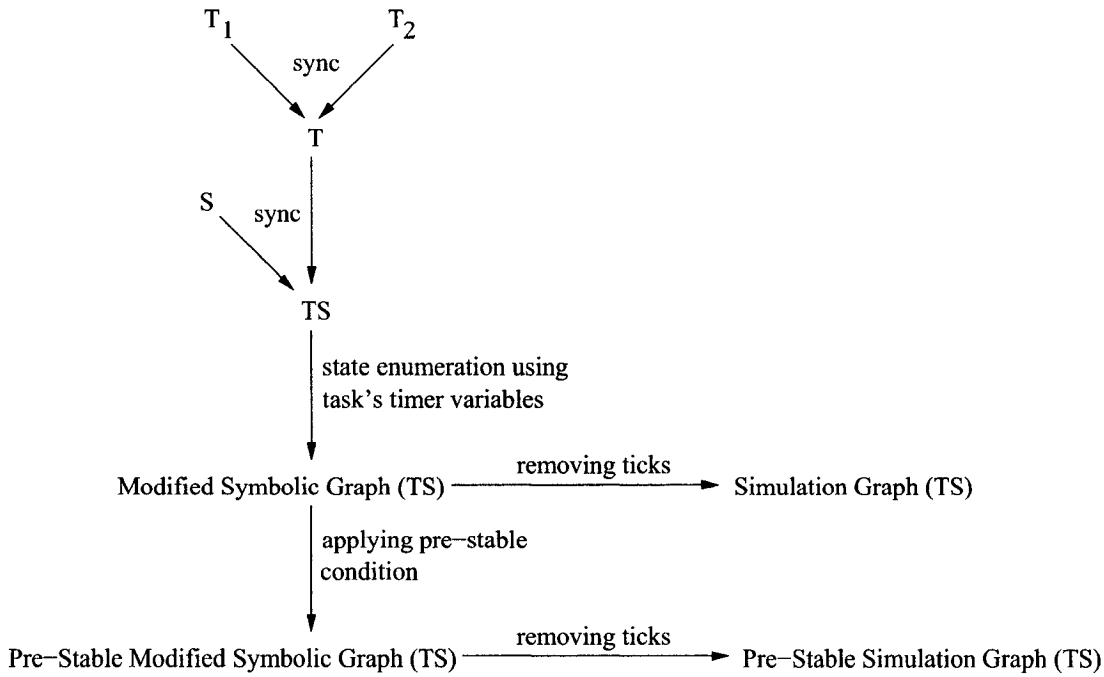

Figure 5.8: Framework for reducing state space.

we apply the pre-stable condition given in [24]. According to this condition, given two nodes R and R', and a transition T, R is pre-stable with respect to R' for T iff every region in R has a successor in R' based on the transition T. We call the graph obtained after applying the pre-stable condition pre-stable modified symbolic graph.

The pre-stable modified symbolic graph is a scheduler for the two tasks, and this can be compared with the scheduler design of [39, 41], but with a reduced state space. The pre-stable simulation graph is then obtained by removing explicit passages of time (or tick events) from the pre-stable modified symbolic graph and explicitly representing task transitions only.

5.3 Modeling of Real-Time Systems Under Non-Preemptive Assumption

In this section we study a system consisting of n tasks to be scheduled on a single processor. Let I be an index set, and let $\mathbf{T_i}$ denote a task with an execution time of $c_i \in \mathbb{N}$ time units. Each task is modeled by a symbolic timed automaton, so that the *composite task model* can be described as:

$$\mathbf{T} := \Big|\Big|_{i \in I} \mathbf{T_i}$$

where $||$ denotes synchronous product [62]. For simplicity we assume that $n = 2$, although the results can be generalized to arbitrarily large values of n.

A task is modeled using three main events: beginning of execution, completion of execution, and arrival of the next cycle. Beginning of execution of a task $\mathbf{T_i}$ is denoted by an event b_i, which resets its execution timer to zero. Event e_i is used when execution of $\mathbf{T_i}$ is complete, while event n is used when either $\mathbf{T_i}$ or another task in the system reaches its period. Also, x_i and y_i are the timer variables for period and execution time of $\mathbf{T_i}$, respectively. The complete model of the two tasks $\mathbf{T_1}$ and $\mathbf{T_2}$ are shown in Fig. 5.9 and Fig. 5.10.

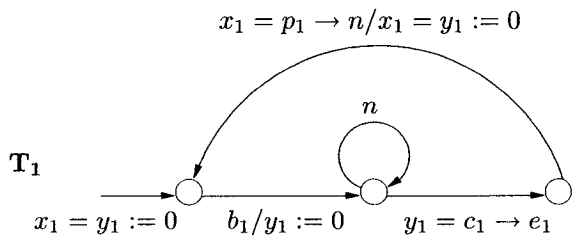

Figure 5.9: Task model of $\mathbf{T_1}$.

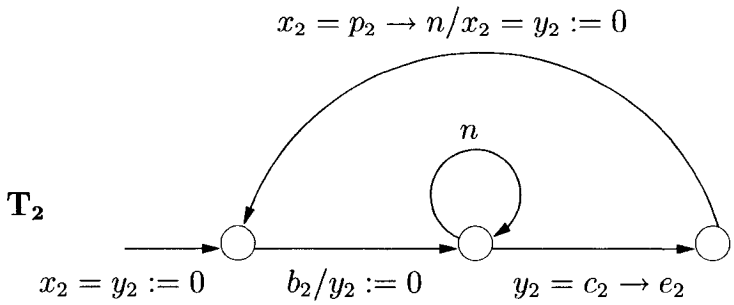

Figure 5.10: Task model of $\mathbf{T_2}$.

The composed automaton model is obtained as in Fig. 5.11.

$$\mathbf{T} := \mathsf{sync}(\mathbf{T_1}, \mathbf{T_2})$$

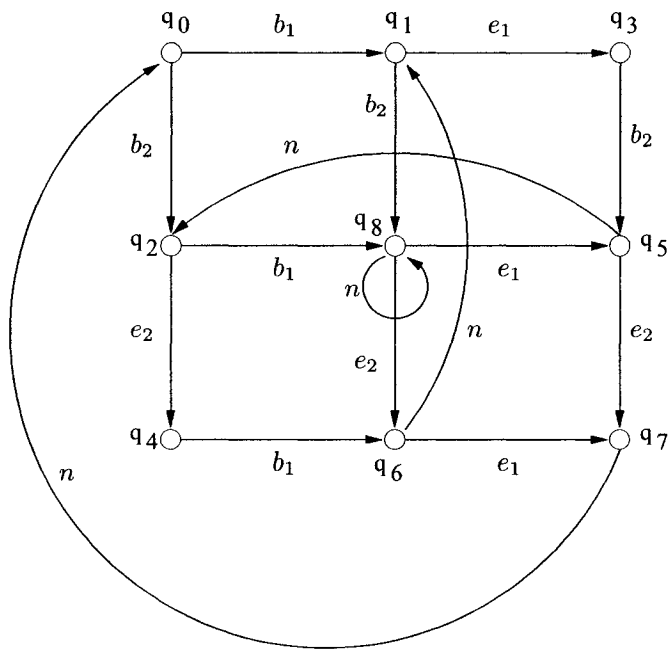

Figure 5.11: Composed automaton of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$.

The automaton portraying the specification for scheduler design for non-preemptive set of tasks is given in Fig. 5.12. It simply states that if, say, the execution of $\mathbf{T_1}$

has already started, the execution of $\mathbf{T_2}$ cannot begin until after the first execution is complete.

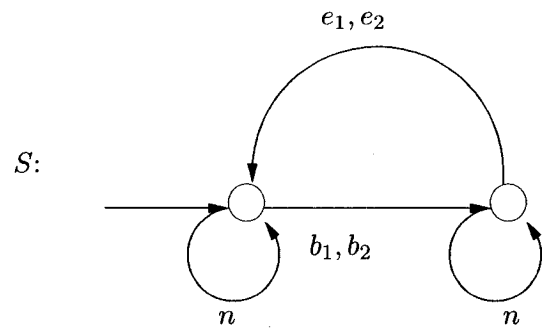

Figure 5.12: Specification for non-preemption of $\mathbf{T_1}$ and $\mathbf{T_2}$.

The scheduler automaton of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$ is obtained by taking the synchronous product [62] of specification with the composed task model, as depicted in Fig. 5.13.

$$\mathbf{TS} := \texttt{meet}(\mathbf{T}, \mathbf{S})$$

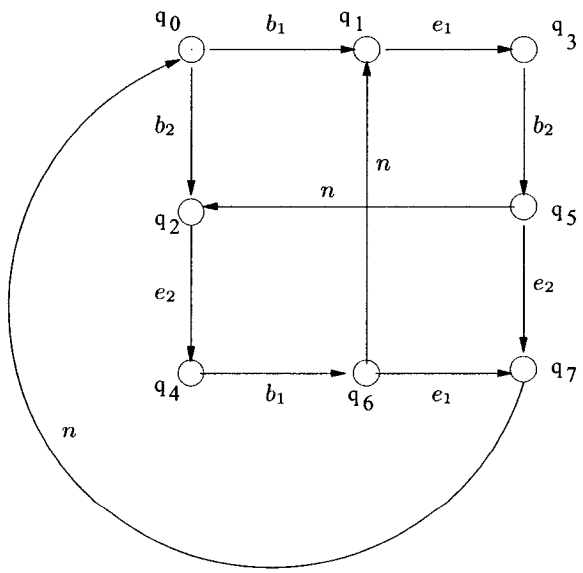

Figure 5.13: Automaton used to design schedulers with reduced state space.

The composed model of Fig. 5.13 could be used to design a scheduler based on state description and task events. But then this would lead to an exponential growth of states with an unmanageable number of transitions. In our modified approach of symbolic scheduler design, we first enumerate the state space and then apply the pre-stable condition to reduce the state space by removing the paths leading to states having no successor. This in turn removes the transitions along those paths. A state in the composed task model does not have a successor when one of the tasks misses its deadline at a particular state, which could be calculated based on the values of the timer variables of task periods and execution times.

5.4 Example

In this section, we will provide a comprehensive example to illustrate in detail the procedure followed in designing a scheduler. Consider $\mathbf{T_1} : (2, 1)$ and $\mathbf{T_2} : (4, 2)$ to be two hard real-time tasks to be scheduled on a single CPU. In $\mathbf{T_i} : (p_i, c_i)$, p_i and c_i stand respectively for period and execution time of task i.

The task automata for $\mathbf{T_1}$ and $\mathbf{T_2}$ are shown in Fig. 5.14 and Fig. 5.15.

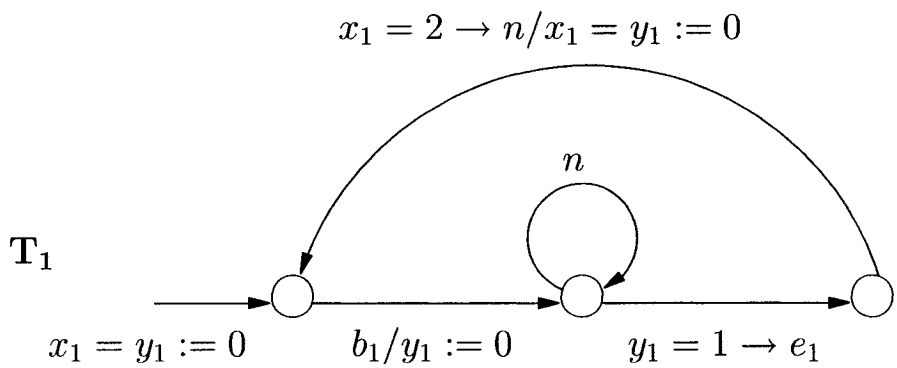

Figure 5.14: Task model of $\mathbf{T_1}$.

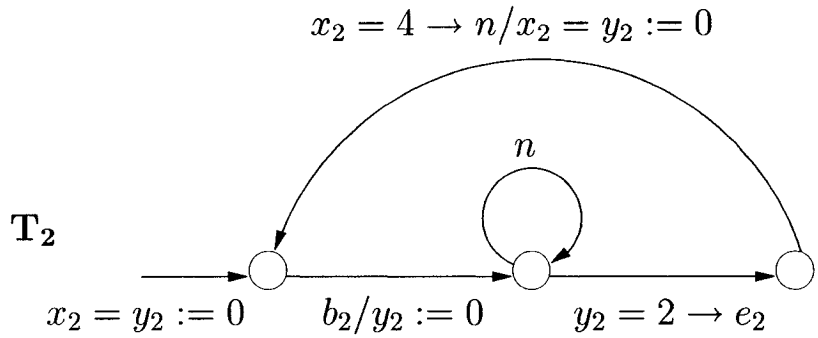

Figure 5.15: Task model of $\mathbf{T_2}$.

The composed automaton is obtained as in Fig. 5.11 and the specification automaton is as illustrated in Fig. 5.12. The scheduler automaton of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$ obtained by taking the synchronous product of specification with the composed task model is as given in Fig. 5.13. The composed model of Fig. 5.13 could be used to design a scheduler based on state description and task events.

Fig. 5.16 enumerates the modified symbolic graph of composed model of the two real-time tasks $\mathbf{T_1}$ and $\mathbf{T_2}$. It consists of the various states as in Fig. 5.13 along with the values of the timer variables for period and execution time for the two tasks. For example, $q_0 0000$ means that at state q_0, the timer variables of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$, namely, x_1, y_1, x_2 and y_2, respectively, are equal to zero, i.e. they have just been reset. The modified symbolic graph is then obtained by considering the different transitions that could be traversed based on the various events of Fig. 5.13 while satisfying the timing constraints.

According to Fig. 5.13, from state q_0, either event b_1 or event b_2 could be executed. In Fig 5.16, we have considered event b_1 first.

The region sets in Fig. 5.16 denoted by R_0, R_1, etc., have nodes with time (portrayed by the *tick* event) passing explicitly between them. One can observe that transitions from some of the nodes inside these region sets are not portrayed beyond

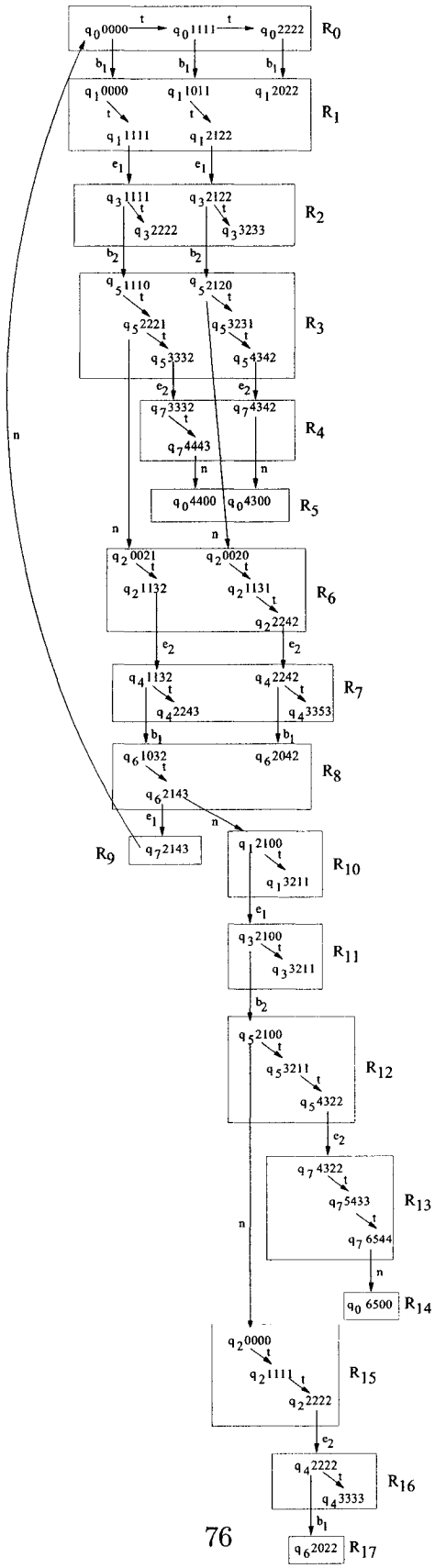

Figure 5.16: Modified symbolic graph of the composed task taking event b_1 first.

a certain point. For example, consider node $q_1 2022$ in region R_1 of the figure. In that node, the timer variables for task $\mathbf{T_1}$, namely, period (x_1) and execution time (y_1) have values *2* and *zero*, respectively (meaning that two time units have elapsed with task $\mathbf{T_1}$ never getting a chance to run), while the timer variables of task $\mathbf{T_2}$, namely, x_2 and y_2 have values *2* and *2*, respectively (meaning that task $\mathbf{T_2}$ had been executed for two units of time). From that node, no more transitions have been illustrated because task $\mathbf{T_1}$ had already missed its deadline (since it must have been executed for one time unit before 2 units of time had elapsed), meaning that the path traversed from that node would not provide us with a feasible scheduler.

Instead of taking b_1, if event b_2 is executed first by the scheduler, the modified symbolic graph in Fig. 5.17 illustrates that a feasible schedule does not exist. Region set R_3 in the figure has all its regions blocked, meaning that they could not propagate further because the deadlines of either or both tasks are missed. This provides us with insight that a feasible schedule must start with event b_1, and not with event b_2.

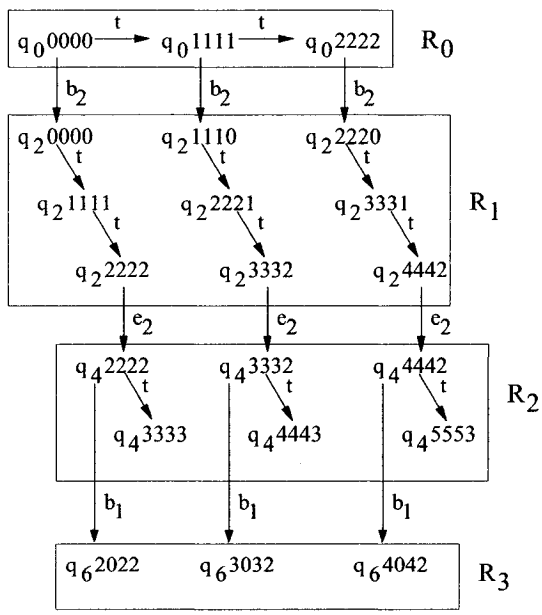

Figure 5.17: Modified symbolic graph of the composed task taking event b_2 first.

The simulation graph is a graph that computes the regions reached from a region set by a finite sequence of timed transitions followed by a task event, and is finite since there is a finite number of regions. This is obtained by enumerating nodes as region sets while making only the task transitions explicit. Time passes implicitly inside the nodes as shown in Fig. 5.18.

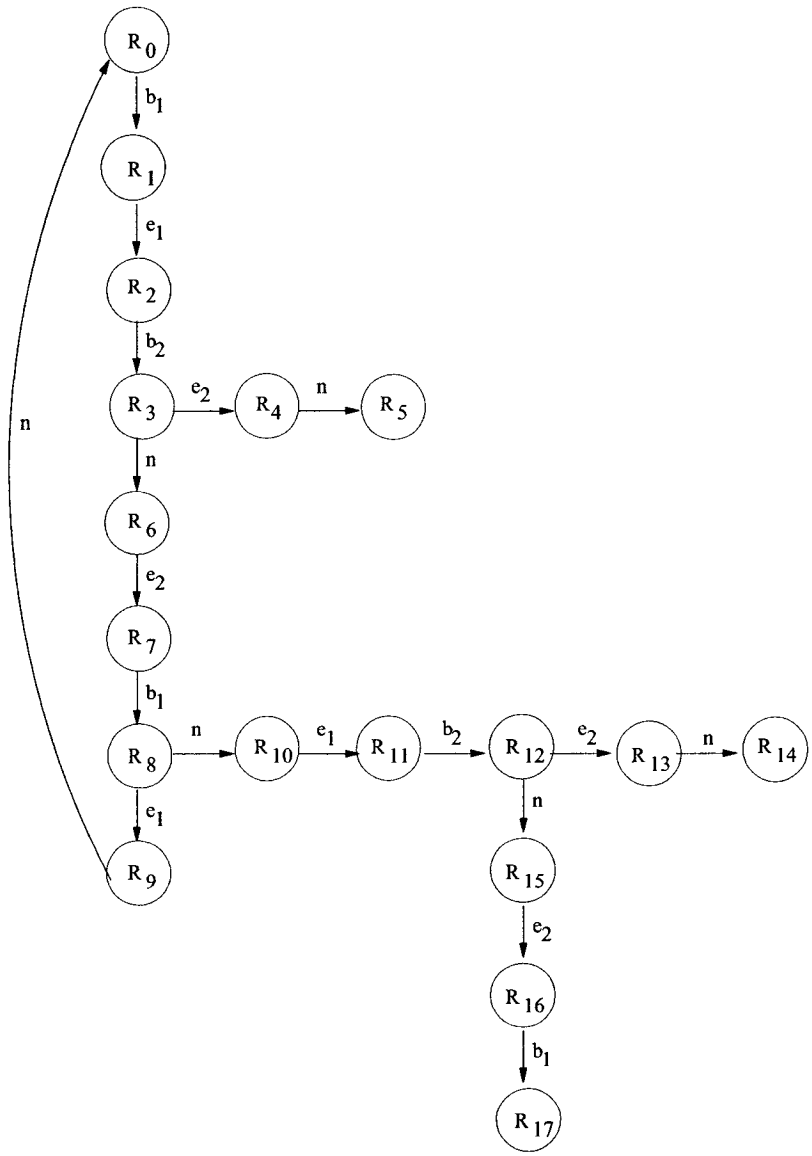

Figure 5.18: Simulation graph of the composed tasks.

In order to reduce the state space in the composed tasks' modified symbolic graph, we apply the pre-stable condition [24] to the modified symbolic graph of Fig. 5.16. We call the resulting graph pre-stable modified symbolic graph. The pre-stable modified symbolic graph, shown in Fig. 5.19, is a scheduler for the two tasks, and this can be compared with the scheduler of [39, 41], but with a reduced state space.

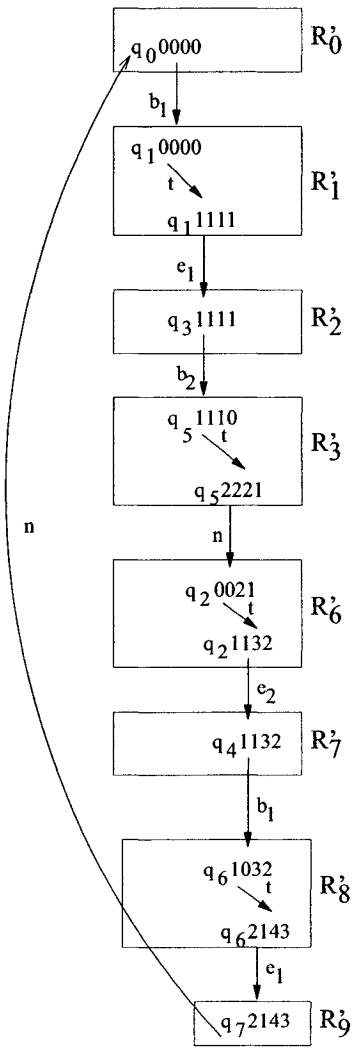

Figure 5.19: Pre-stable modified symbolic graph of the composed tasks (b_1 is taken at the initial state).

The region sets obtained in the pre-stable modified symbolic graph are subsets of

the corresponding region sets in the modified symbolic graph. For example, $R'_0 \subseteq R_0$ and $R'_1 \subseteq R_1$, and so on. We obtain region R'_0 by removing the states $q_0 1111$ and $q_0 2222$ from R_0 since event and tick transitions from these nodes do not lead to a valid scheduling sequence. On the other hand, the state $q_0 0000$ is part of the region set R'_0 since it is possible to reach the initial state by taking a sequence of event and tick transitions, thereby forming a closed path as per the pre-stable condition of [24].

The corresponding pre-stable simulation graph is obtained by removing explicit tick transitions from Fig. 5.19, and representing region sets by nodes, as shown in Fig. 5.20.

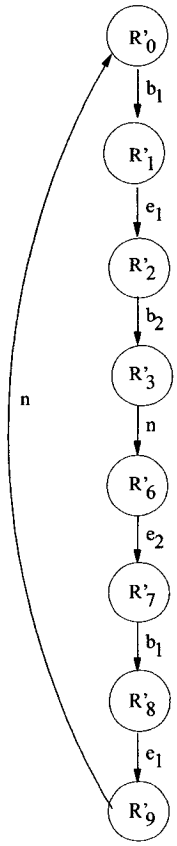

Figure 5.20: Pre-stable simulation graph of the composed tasks (b_1 is taken at the initial state).

To verify the feasibility of our designed scheduler, we employ a model checking approach based on zone automata [7]. A zone is defined as a set of possible values for variables in a given state. Assume that a transition labeled with σ is *eligible* at state $s \in S$. If σ is guarded by g, we denote by G a subset of $\mathbf{N^n}$ that makes the guard formula g true. If we denote by V the set of possible values for variables in C at state s, then σ is *enabled* at s if $V \cap G \neq \emptyset$.

Fig. 5.21 illustrates an automaton model of our scheduler, as obtained in Fig. 5.20. Instead of region representation, we outline state notations in Fig. 5.21 for simplicity in understanding.

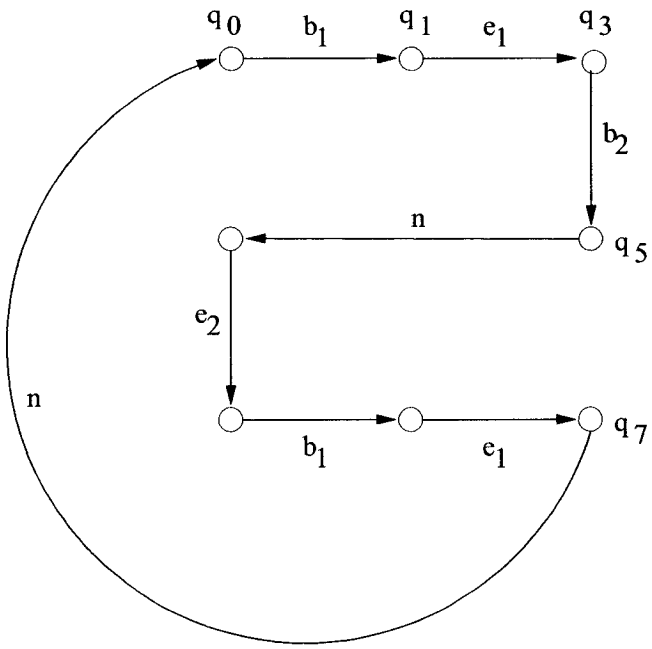

Figure 5.21: Automaton of the scheduler.

Zonal representation of our scheduler automaton based on guards on the transitions is shown in Fig. 5.22.

The figure depicts zonal changes in terms of timer variables for both tasks resulted from the occurrence of an event. In the figure, axis x_1 denotes the timer variable for

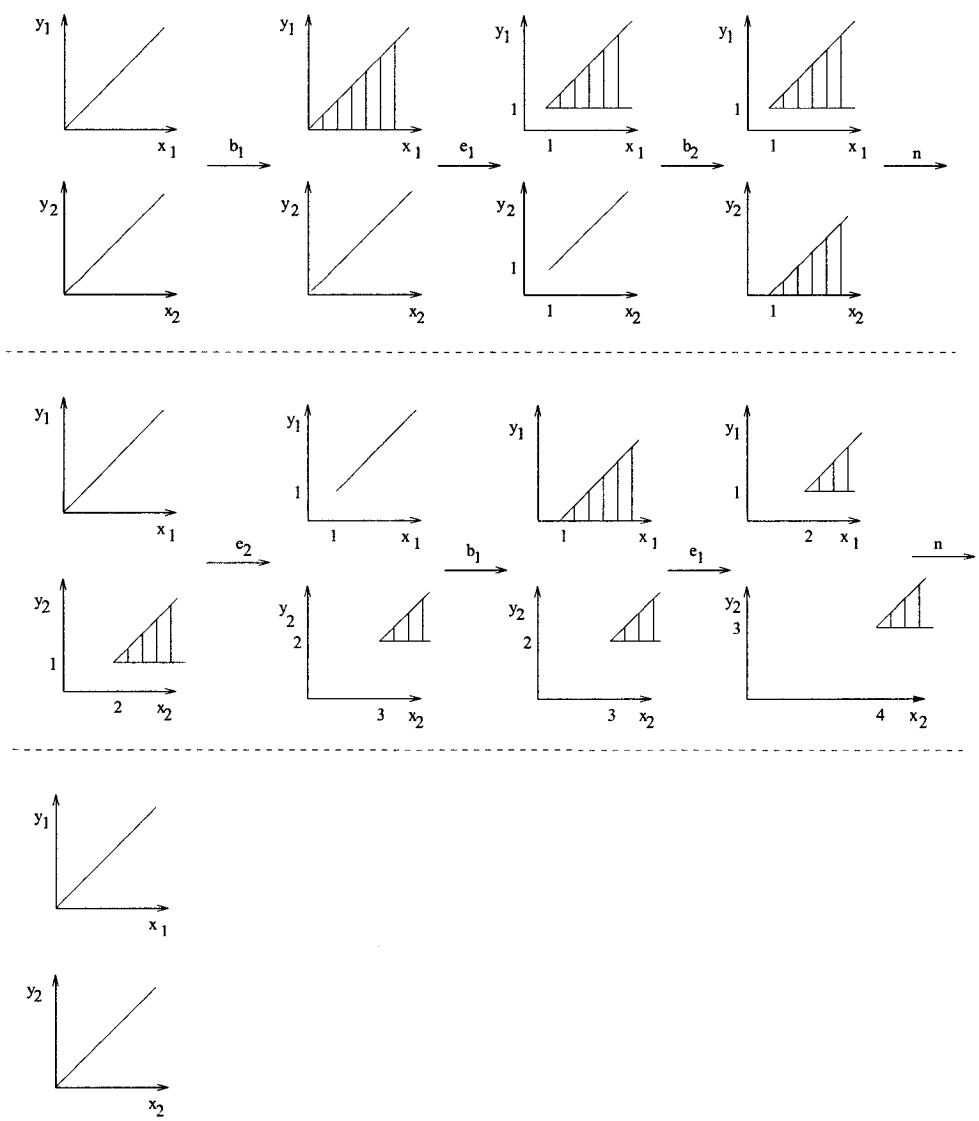

Figure 5.22: Zonal representation of the scheduler automaton.

task $\mathbf{T_1}$'s period and axis y_1 for its execution time, while axis x_2 denotes the timer variable for task $\mathbf{T_2}$'s period and axis y_2 for its execution time.

In Fig. 5.21, state q_0 is the initial state with timer variables of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$ reset. Hence in Fig. 5.22, the zones for both tasks consist of straight lines $y_1 = x_1$ and $y_2 = x_2$. When event b_1 is executed, the zone representing task $\mathbf{T_1}$ has y_1 reset to zero while x_1 continues to increment with time. After y_1 is reset, both x_1 and y_1 uniformly increment with time, thus the corresponding zone is the lower triangle described by $0 \leq y_1 \leq x_1$ (note that x_1 and y_1 are nonnegative integers). At the same time, the zone for $\mathbf{T_2}$ still exhibits a straight line since no event of this task had yet been executed.

Before executing the next event e_1 of task $\mathbf{T_1}$, the values of its timer variables, namely x_1 and y_1, are checked since the event is guarded by those values. In other words, the guard for event e_1 has to be satisfied before it can be enabled. In particular, for enabling event e_1, the value of y_1 has to be equal to one. In the zone representation of Fig. 5.22, for finding out whether event e_1 could be enabled at this state q_1 or not, we draw the guard formula $y_1 = 1$, which is a straight horizontal line, and check if this line intersects the lower triangle zone (formed due to previous event transition). From Fig. 5.22, it is clear that the straight line $y_1 = 1$ does intersect the triangle, meaning that the relationship $V \cap G \neq \emptyset$ is true. Hence event e_1 is enabled at that state and the corresponding change in the shape of the triangle after e_1 is taken is depicted in Fig. 5.22.

Then event b_2's execution doesn't change the zone for $\mathbf{T_1}$ while that of $\mathbf{T_2}$'s has y_2 reset to zero and x_2 could get incremented by any number of time units. The same reasoning could be applied for the rest of the event transitions in Fig. 5.22. While executing the final event transition of n based on the automaton in Fig. 5.21, the timer variables of both the tasks' zones get reset, since both of them satisfy the guard

on event n. This takes us back to the zone from where the transition sequence had initially started, thereby validating the feasibility of the scheduler in Fig. 5.21.

Fig. 5.23 illustrates an automaton model of another proposed scheduler, and through zonal methods we will now verify the non-feasibility of this scheduler.

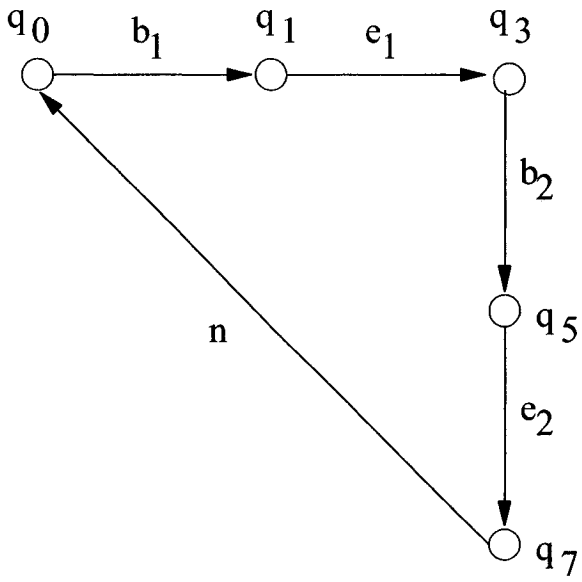

Figure 5.23: Automaton of a non-feasible scheduler.

Applying the reasoning used in Fig. 5.22 on Fig. 5.24, we reach a stage in the zonal graph sequence (based on the event transitions of the automaton in Fig. 5.23), where the transition gets blocked and does not take the automaton back to its initial state, thereby validating the non-feasibility of the scheduler in Fig. 5.23. The blocking in Fig. 5.24 happens while trying to execute the event n. The reason for the blocking is that event n cannot be taken as per Fig. 5.23, since the value of timer variable x_1 has exceeded its guard of two.

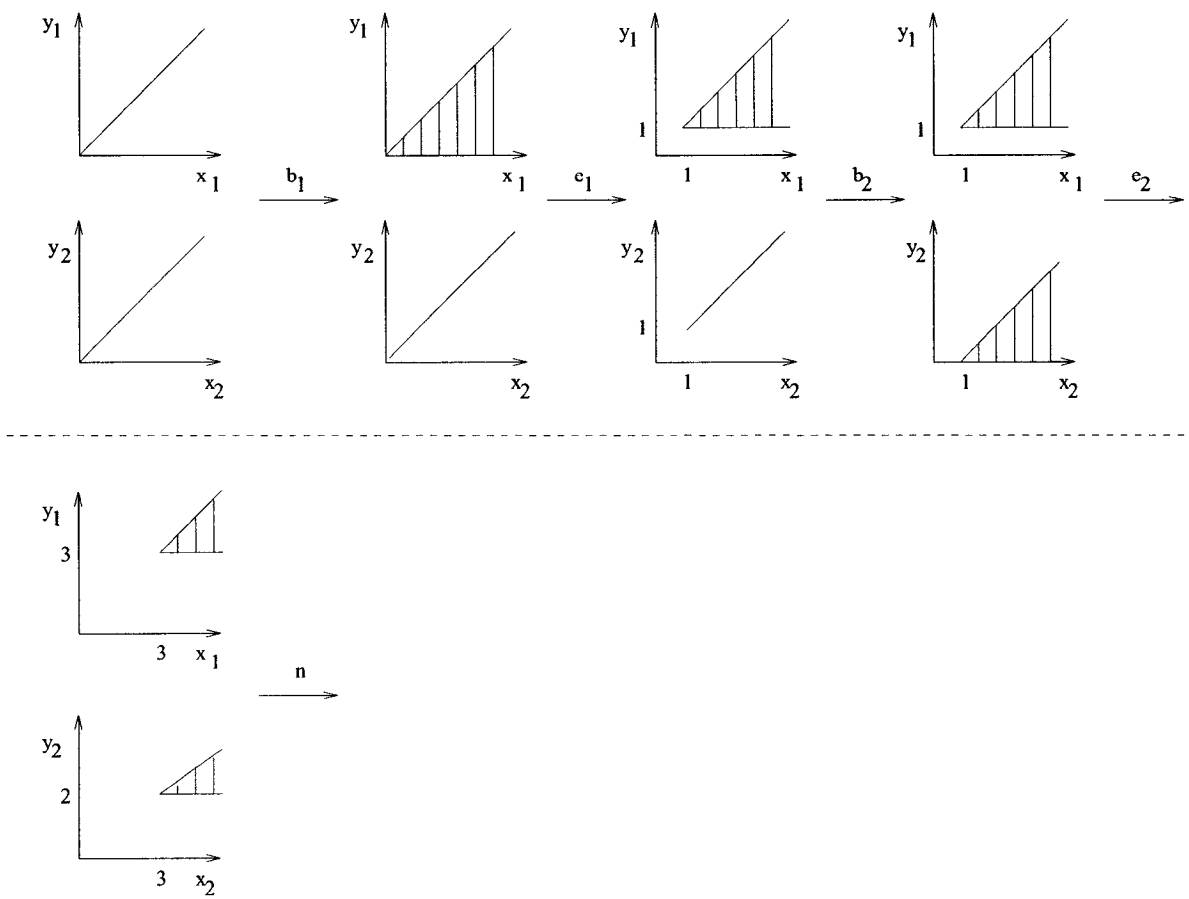

Figure 5.24: Zonal representation of a non-feasible scheduler automaton.

5.5 Modified Symbolic Scheduler Design for Preemptive Real-Time Tasks

This section presents the design of a scheduler for a preemptive set of real-time tasks using the modified symbolic method. At first, we provide the various automata required for modeling the tasks under preemptive assumption. Fig. 5.25 illustrates the automaton of task $\mathbf{T_1}$; except for extra events s_1 and r_1, and an extra variable z_1, the automaton is similar to the one under the non-preemptive assumption. Event s is used when a task gets preempted while its execution is not yet complete. Event s is called a *suspend* event. When event s occurs, the current execution value of the task is stored in a variable. Variable z is used to store the value of preempted task's execution time; in other words, it records how much of the task has been executed thus far. Event r is used when a preempted task resumes its execution and is referred to as a *resume* event. When event r occurs, the stored value of the task's execution is retrieved from z and the task continues its execution until its completion.

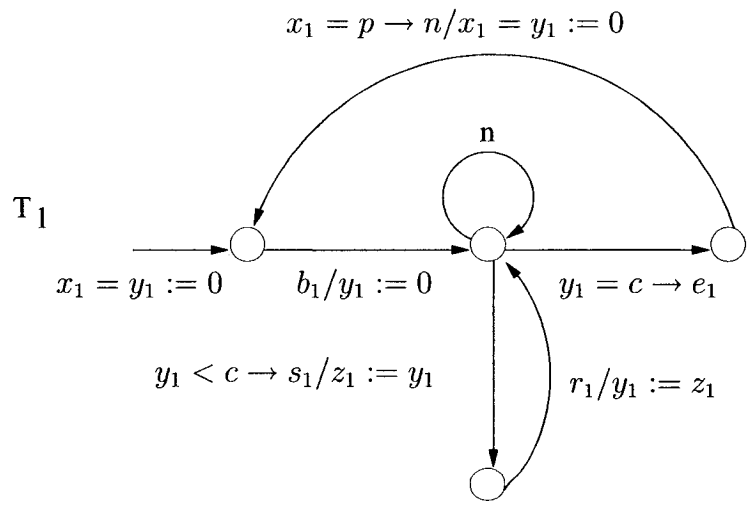

Figure 5.25: Automaton of Task $\mathbf{T_1}$ under preemptive assumption.

The automaton for task $\mathbf{T_2}$ is depicted in Fig. 5.26.

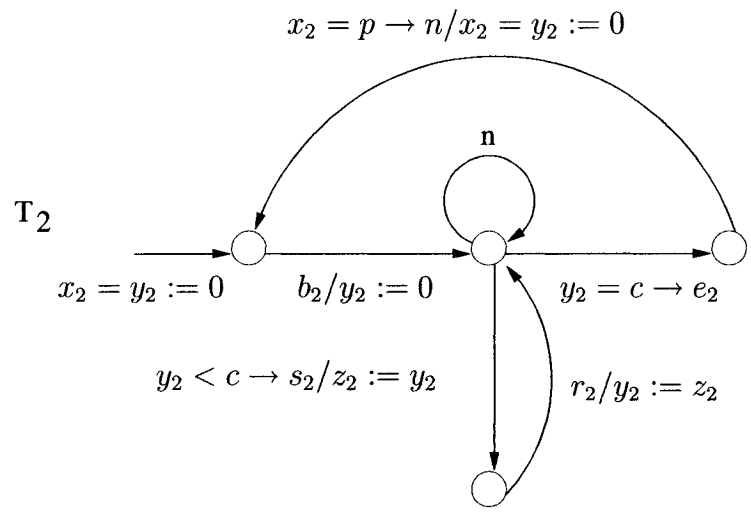

Figure 5.26: Automaton of Task $\mathbf{T_2}$ under preemptive assumption.

We provide the specification automaton for preemptive tasks in Fig. 5.27. It states that when the execution of a task is started (b_i) or has just been resumed (r_i), the other task cannot begin its execution until after the execution of the first task is complete (e_i) or is suspended (s_i).

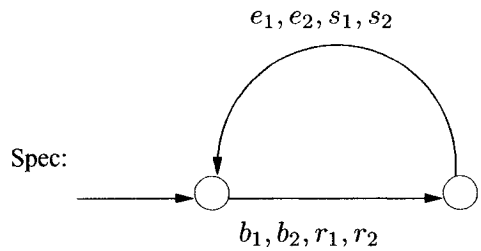

Figure 5.27: Specification for preemptive tasks.

The composed automaton model is shown in Fig. 5.28.

$$\mathbf{T} := \mathsf{sync}(\mathbf{T_1}, \mathbf{T_2})$$

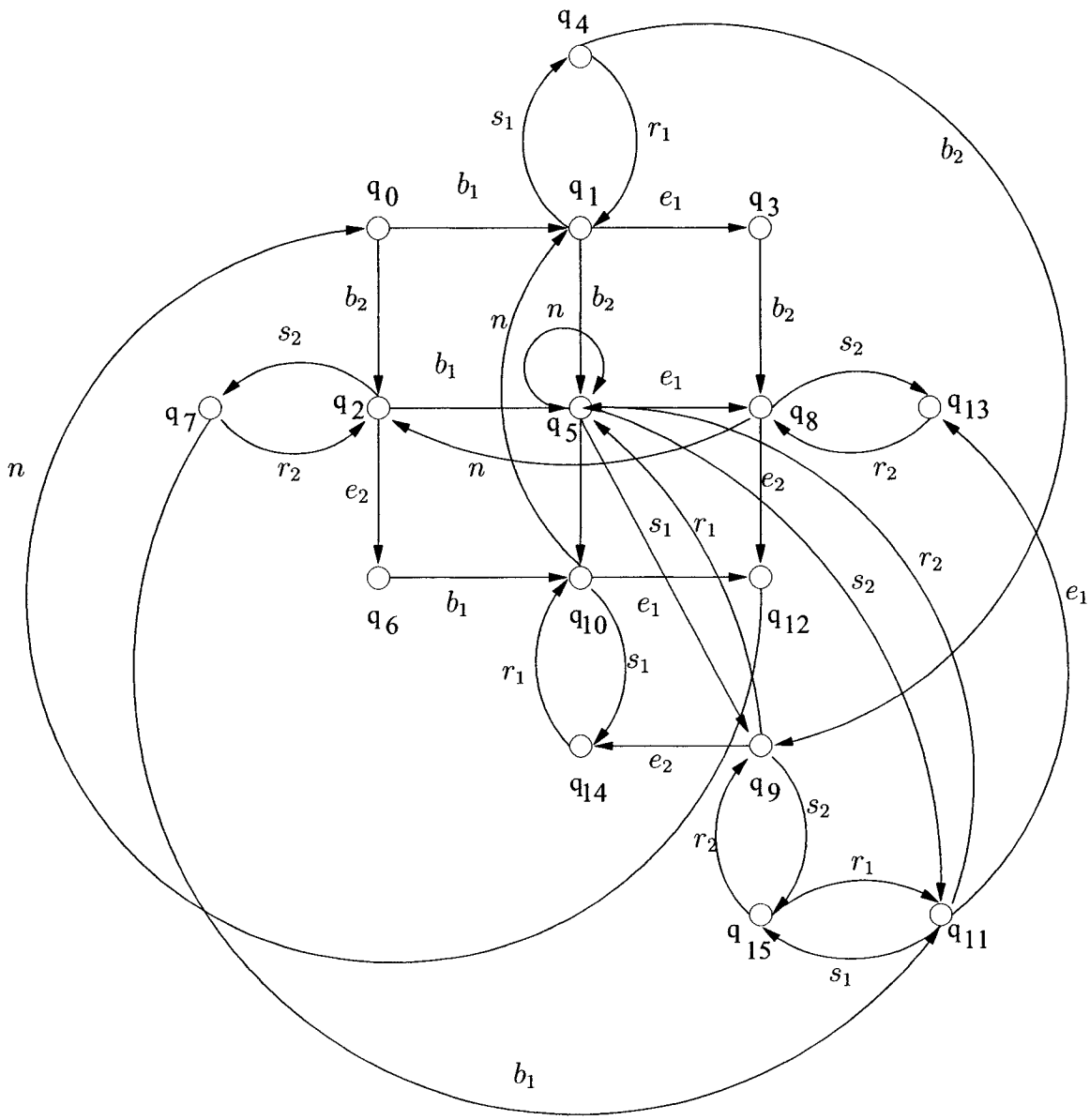

Figure 5.28: Composed automaton of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$.

The automaton that could be used to design schedulers for tasks $\mathbf{T_1}$ and $\mathbf{T_2}$ with reduced state space is obtained by taking the synchronous product of specification with the composed task model, which is depicted in Fig. 5.29.

$$\mathbf{TS} := \texttt{meet}(\mathbf{T}, \mathbf{S})$$

where $\mathbf{T}$ is composed automaton model and $\mathbf{S}$ is specification for task's preemption.

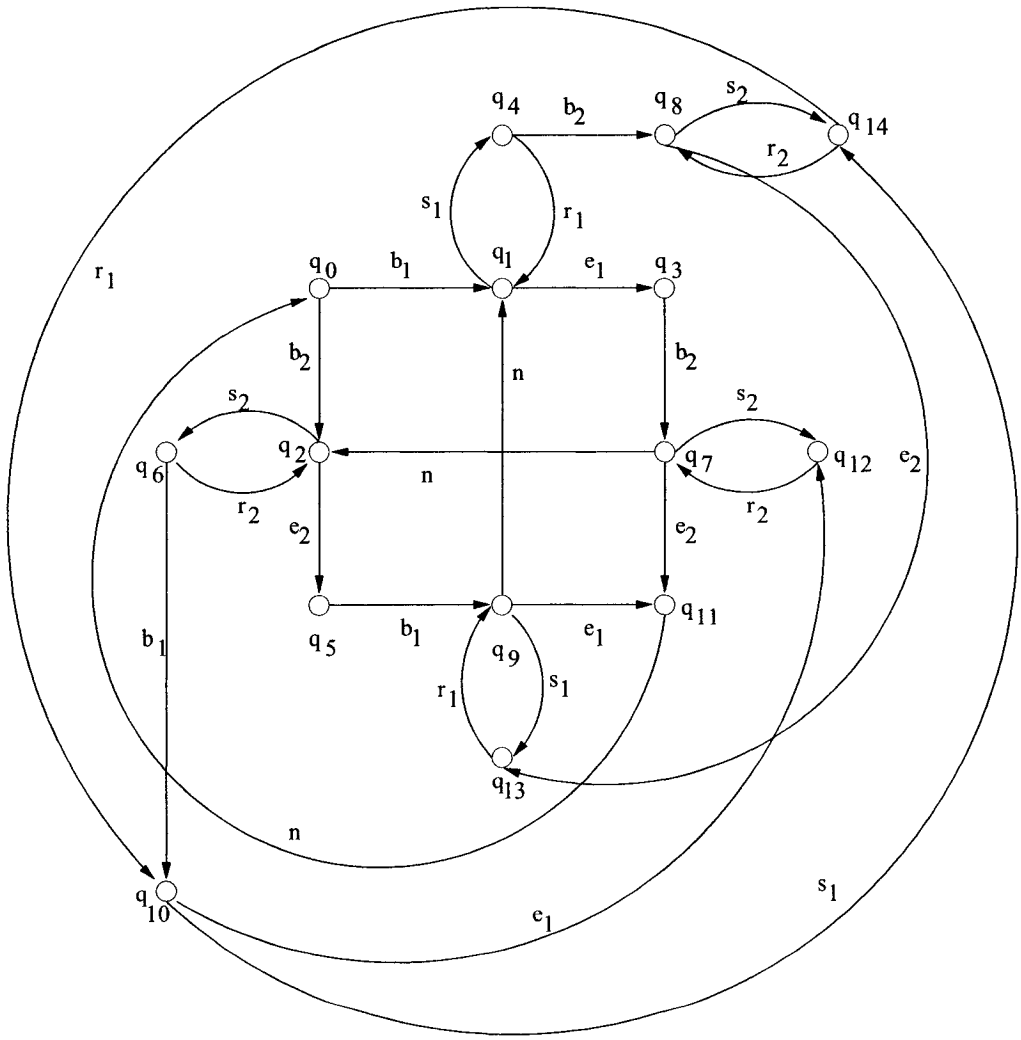

Figure 5.29: Automaton used to design schedulers with reduced state space.

Fig. 5.30 enumerates the modified symbolic graph of composed model of the two tasks under preemptive assumption. It consists of the various states as in Fig. 5.29 along with the values of the timer variables for period and execution time for the two tasks. For example, $q_0 0000$ means that at state q_0, the timer variables of tasks $\mathbf{T_1}$ and $\mathbf{T_2}$, namely, x_1, y_1, x_2 and y_2 respectively, are equal to zero i.e. they have just been reset. The modified symbolic graph is then obtained by considering the different transitions that could be traversed based on the various events of Fig. 5.29 while satisfying the timing constraints.

The region sets in Fig. 5.30 denoted by R_0, R_1, etc., have nodes with time (portrayed by the *tick* event) passing explicitly between them. One can observe that transitions from some of the nodes inside these region sets are not portrayed beyond a certain point. For example, consider node $q_9 2022$ in region R_3 of the figure. In that node, the timer variables for task $\mathbf{T_1}$, namely, period (x_1) and execution time (y_1) have values *2* and *zero*, respectively (meaning that two time units have elapsed with task $\mathbf{T_1}$ never getting a chance to run), while the timer variables of task $\mathbf{T_2}$, namely, x_2 and y_2 have values *2* and *2*, respectively (meaning that task $\mathbf{T_2}$ had been executed for two units of time). From that node, no more transitions have been illustrated because task $\mathbf{T_1}$ had already missed its deadline (since it must have been executed for one time unit before 2 units of time had elapsed), meaning that the path traversed from that node would not provide us with a feasible scheduler.

Also, in Fig. 5.30, from region R_5, event s_1 could also be executed but has not been depicted in the figure. This is because, suspending $\mathbf{T_1}$ would obviously lead to the execution of task $\mathbf{T_2}$, thereby resulting in the missing of its deadline. For example, from node $q_{10} 0000$ in region set R_5, executing event s_1 will result in the suspension (preemption) of $\mathbf{T_1}$ without the completion of its execution, and thereby leading to the missing of its deadline.

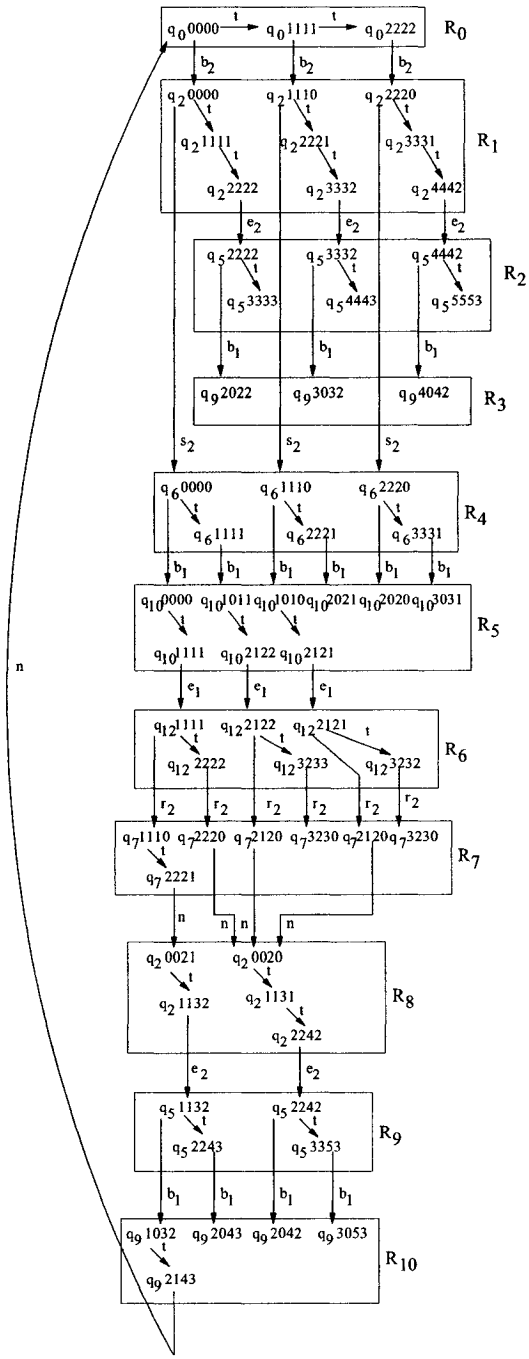

Figure 5.30: Modified Symbolic graph of the composed tasks.

Next, the simulation graph is obtained by enumerating nodes as region sets where only task transitions are explicit, while time passes implicitly inside the nodes as shown in Fig. 5.31.

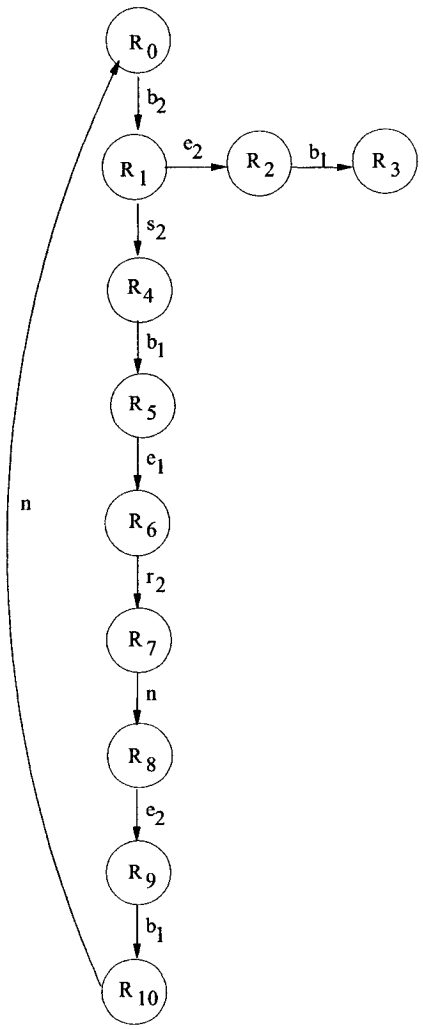

Figure 5.31: Simulation graph of the composed tasks.

The pre-stable modified symbolic graph of Fig. 5.32 is a scheduler for the two tasks, and this can be compared with the scheduler of [39, 41], but with a reduced state space. The region sets obtained in the pre-stable modified symbolic graph are subsets of the corresponding region sets in the modified symbolic graph.

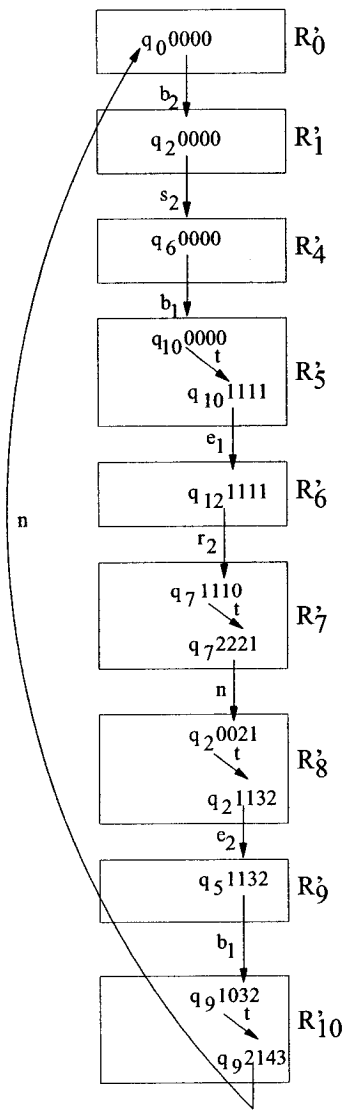

Figure 5.32: Pre-stable modified symbolic graph of the composed tasks.

The corresponding pre-stable simulation graph is obtained by removing explicit time transitions from Fig. 5.32, and also representing region sets by nodes. This is shown in Fig. 5.33.

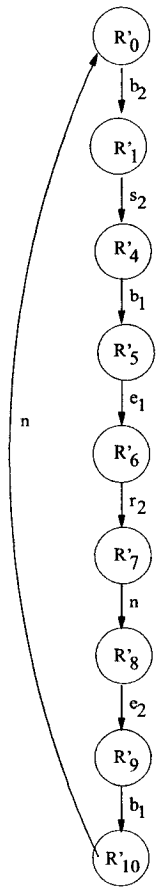

Figure 5.33: Pre-stable simulation graph of the composed tasks.

5.6 Conclusion

This chapter provided a formal framework for the synthesis of schedulers for hard real-time tasks on single processor platforms with a reduced state space. For the purpose of state space reduction, we had utilized a modified form of symbolic graph called *simulation graph* [24] and applied the pre-stable condition [24] on it. We then

employed the concept of zone automata for verifying the designed scheduler. Also, in the chapter, we have extended the scheduler design procedure under the preemptive assumption.

Chapter 6

Conclusion and Future Work

We have shown that supervisory control theory (SCT) of discrete-event systems can be applied to the scheduling of hard real-time systems. In that sense, we have presented a formal framework for the synthesis of real-time schedulers on uniprocessor systems using priority-based supervisory control of timed discrete-event systems. Then we applied SCT to design schedulers for uniform multiprocessor systems. Such a formal theory helps in a systematic approach to problem solving and presents a rigorous tool for analysis and synthesis of real-time systems.

The method for designing schedulers is based on successive restriction of the system to be scheduled by constraints defined from scheduling and environmental requirements. This approach allows a unified view of scheduling theory based on the timing analysis of models of real-time applications. Using this method, the problem of finding out whether a set of real-time tasks are schedulable or not, and also the problem of finding out a suitable scheduling algorithm are treated as dual in nature: if we find out a solution to the schedulability problem, it necessarily implies that we have a solution for the type of algorithm to be applied, too.

There are known limitations associated with our choice to explicitly represent

discrete time. Namely, model sizes are considerably large when time is explicitly represented, and since the granularity of time has to be fixed *a priori*, discrete-timed models are less flexible in expressing timing constraints than dense-time models. Yet we believe our choices make scheduler synthesis algorithms easier to conceptualize and lead to more efficient design procedures. We believe that the method is tractable for non-trivial systems of medium size. As the synthesized schedulers are maximal, i.e., they contain all the schedules satisfying the given property, simpler deterministic schedulers could be obtained by eliminating nondeterminism.

Also in our work, we have presented a framework based on modified symbolic methodology for designing schedulers with reduced state space for hard real-time tasks on single processor systems. The procedure followed in our scheduler design approach with reduced state space is rather informal, but we hope that the presented framework would be an interesting idea for future work on formalization of real-time schedulers with reduced state space. We have utilized the algorithm presented in [24] in order to manipulate sets of states, rather than individual states. Due to lack of formalism in our synthesis approach for reduced state space schedulers, we verify their correctness using the concept of zone automaton. Another contribution of our work has been the consideration of both non-preemptive and preemptive set of real-time tasks. In comparison with the universal scheduler obtained in [39, 41], we found the pre-stable symbolic scheduler obtained in this work to be far smaller in size, thereby handling the state space explosion problem that we faced in [39, 41].

6.1 Future Work

1. As part of our future work, we intend to relax some of the assumptions we had made while designing schedulers for multiprocessor systems. In particular,

we would like to investigate the scheduler design by including factors such as real-time task parameter size and intra-task parallelism, which could pose much bigger challenges in the design procedure.

2. We would also like to explore the possibility of applying our scheduler design procedure on identical multiprocessors, wherein all processors have identical computing capacities and one thus can hope for developing simpler design algorithms.

3. We would also be trying to explore the possibility of using Zone Automata for synthesizing schedulers rather than for verifying them. For example, in our current work, we had used the zonal representations for verifying the feasibility of schedulers, which were already synthesized and modeled as automata using the modified symbolic method. Now it would be interesting to see if it was possible to get a scheduler with reduced state space directly from the composed task model using Zone Automata.

4. We would also like to develop a software to automate the symbolic approach, using more realistic models of real-time systems (taking into account dependency through resource sharing and precedence graphs, which can be readily expressed as specifications of environmental constraints). Of course, the development of software is only possible after formalizing the design procedure for reduced state space schedulers.

Bibliography

[1] Y. Abdeddam, A. Kerbaa and O. Maler, "Task Graph Scheduling using Timed Automata," *FMPPTA*, pp.237–238, 2003.

[2] Y. Abdeddam, E. Asarin and O. Maler, "On Optimal Scheduling under Uncertainty," *TACAS*, LNCS 2619, pp.240–255, 2003.

[3] Y. Abdeddam, E. Asarin and O. Maler, "Scheduling with Timed Automata," *Theoretical Computer Science*, pp.272–300, 2005.

[4] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, "A Framework for Scheduler Synthesis," *Proceedings of RTSS*, pp.154–163, 1999.

[5] K. Altisen, G. Gossler and J. Sifakis, "A Methodology for the Construction of Scheduled Systems," *FTRTFT*, LNCS 1926, pp.106–120, 2000.

[6] K. Altisen, G. Goessler and J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," *Journal of Real-Time Systems, special issue on Control Approaches to Real Time Systems*, vol. 23, pp.55–84, 2002.

[7] R. Alur and D. L.Dill, "A theory of timed automata," *Theoretical Computer Science*, pp.183–235, 1994.

[8] R. Alur and D. Dill, "Automata for modeling real-time systems," *ICALP: Automata, Languages and Programming*, LNCS, pp.322–335, 1990.

[9] R. Alur and R.P. Kurshan. "Timing analysis in COSPAN," *in R. Alur, T. A. Henzinger and E. D. Sontag, editors, Hybrid Systems III. Verification and Control*, LNCS, vol. 1066, pp.220–231, 1995.

[10] R. Alur, C. Courcoubetis and D. Dill, "Modelchecking for real-time systems," *LICS*, pp.414–425, 1990.

[11] R. Alur and D. Dill, "Automata for modeling real-time systems," *In M. S. Paterson, editor, Automata, Languages, and Programming. 17th International Colloquium Proceedings*, LNCS, vol. 443, pp.322–335, 1990.

[12] R. Alur and T.A. Henzinger, "Logics and models of real time: A survey," *In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, Real-Time: Theory in Practice. REX Workshop Proceedings*, LNCS, vol. 600, pp.74–106, 1991.

[13] E. Asarin, O. Maler and A. Pnueli, "Symbolic Controller Synthesis for Discrete and Timed Systems," *Hybrid Systems II*, LNCS, vol. 999, pp.1–20, 1995.

[14] E. Asarin, O. Maler, A. Pnueli and J. Sifakis, "Controller Synthesis for Timed Automata," *Proceedings of IFAC Symposium on System Structure and Control*, Elsevier, pp.469–474, 1998.

[15] E. Asarin and O. Maler, "As Soon as Possible: Time Optimal Control for Timed Automata," *In F. Vaandrager and J. van Schuppen (Eds.), Hybrid Systems: Computation and Control*, LNCS, vol. 1569, pp.19–30, 1999.

[16] E. Asarin, O. Maler and A. Pnueli, "On discretization of delays in timed automata and digital circuits," *In Sangiorgi and de Simone*, pp.470–484, 1998.

[17] F. Balarin, "Approximate reachability analysis of timed automata," *RTSS*, pp.52–61, 1996.

[18] S. Baruah, "The multiprocessor scheduling of precedence-constrained task systems in the presence of interprocessor communication delays," *Operations Research*, vol. 46, no. 1, pp.65–72, 1998.

[19] S. Baruah, "Scheduling Periodic Tasks on Uniform Multiprocessors," *Information Processing Letters*, vol. 80, no. 2, pp.97–104, 2001.

[20] S. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," *IEEE Transactions on Computers*, vol. 52, no. 7, pp.966–970, 2003.

[21] S. Baruah and J. Goossens, "The static-priority scheduling of periodic task systems upon identical multiprocessor platforms," *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pp.427–432, 2003.

[22] W.A. Belluomini, "Algorithms for Synthesis and Verification of Timed Circuits and Systems," PhD thesis, University of Utah, 1999.

[23] W.A. Belluomini and C.J. Myers, "Timed event/level structures," *ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1997.

[24] A. Bouajjani, S. Tripakis and S. Yovine, "On-the-fly symbolic model checking for real-time systems," *18th IEEE Real-Time Systems Symposium*, pp.25–34, 1997.

[25] M. Bozga, O. Maler and S. Tripakis, "Efficient verification of timed automata using dense and discrete time semantics," LNCS, vol. 1703, pp.125–141, 1999.

[26] B. Brandin and W.M. Wonham, "Supervisory Control of Timed Discrete-Event Systems," *IEEE Transactions on Automatic Control*, vol. 39, no. 2, pp.329–342, 1994.

[27] R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp.677–691, 1986.

[28] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and L.J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp.142–170, 1992.

[29] P.C.Y. Chen and W.M. Wonham, "Real-Time Supervisory Control of a Processor for Non-Preemptive Execution of Periodic Tasks," *Real-Time Systems*, vol. 23, no. 3, pp.183–208, 2002.

[30] P.C.Y. Chen and W.M. Wonham, "Non-Preemptive Scheduling of Periodic Tasks: A Discrete-Event Control Approach," *Proceedings of Fifth International Conference on Control, Automation, Robotics and Vision*, pp.1674–1678, 1998.

[31] C.H. Golaszewski and P.J. Ramadge, "On the control of real-time discrete event systems," *Proceedings of 23rd Conference on Information Sciences and Systems*, pp.98–102, 1989.

[32] J. Goossens, S. Funk and S. Baruah, "EDF scheduling on multiprocessor platforms: some counterintuitive observations," *Proceedings of the International Conference on Real-Time Computing Systems and Applications*, pp.321–330, 2002.

[33] J. Goossens, S. Baruah and S. Funk, "Real-time scheduling on multiprocessors," *Proceedings of the 10th International Conference on Real-Time Systems*, pp.189–204, 2002.

[34] J. Goossens, S. Funk and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-time Systems*, vol. 25, no. 2, pp.187–205, 2003.

[35] T. Henzinger, Z. Manna and A. Pnueli, "Timed Transition Systems," *In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, Proceedings of the REX Workshop Real-Time: Theory in Practice*, LNCS, vol. 600, pp.226–251, 1992.

[36] T.A. Henzinger, Z. Manna and A. Pnueli, "What good are digital clocks?," *In W. Kuich, editor, Automata, Languages, and Programming. 19th International Colloquium Proceedings*, LNCS, vol. 623, pp.545–558, 1992.

[37] T.A. Henzinger, O. Kupferman and M. Vardi, "A Space- Efficient On-the-Fly Algorithm for Real-Time Model- Checking," *CONCUR*, LNCS, vol. 1119, pp.514–529, 1996.

[38] T.A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, pp.193–244, 1994.

[39] V. Janarthanan and P. Gohari, "Universal Scheduler Design on Uniprocessors in Supervisory Control of Discrete-Event Systems Framework," *IEEE Conference on Control Applications*, pp.916–921, 2005.

[40] V. Janarthanan and P. Gohari, "Supervisory Control-Based Design of Real-Time Schedulers for Uniform Multiprocessor Systems," *IASTED International Conference on Intelligent Systems and Control*, pp.488–493, 2005.

[41] V. Janarthanan, P. Gohari and A. Saffar, "Formalizing Real-time Scheduling Using Priority-Based Supervisory Control of Discrete- Event Systems," *IEEE Transactions on Automatic Control*, vol. 51, no. 6, pp.1053–1058, 2006.

[42] V. Janarthanan and P. Gohari, "Multiprocessor Scheduling in Supervisory Control of Discrete-Event Systems Framework," *To appear in IASTED Journal for Control and Intelligent Systems, ACTA Press*, 2006.

[43] R. Kumar and M.A. Shayman, "Supervisory Control of Real-Time Systems Using Prioritized Synchronization," *Proceedings of DIMACS Workshop on Verification and Control of Hybrid Systems*, pp.350–361, 1995.

[44] R. Kumar and M.A. Shayman, "Non-blocking Supervisory Control of Nondeterministic Systems via Prioritized Synchronization," *IEEE Transactions on Automatic Control*, vol. 41, no. 8, pp.1160–1175, 1996.

[45] R. Kumar and S. Jiang, "Supervisory Control of Nondeterministic Discrete Event Systems with Driven Events via Masked Prioritized Synchronization," *IEEE Transactions on Automatic Control*, vol. 47, no. 9, pp.1438–1449, 2002.

[46] J. Lehoczky, B. Sprunt and L. Sha, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-time Systems*, vol. 1, no. 1, pp.27–60, 1989.

[47] F. Lin and W.M. Wonham, "Supervisory Control of Timed Discrete Event Systems Under Partial Observation," *IEEE Transactions on Automatic Control*, vol. 40, no. 3, pp.558–562, 1995.

[48] L.C. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp.46–61, 1973.

[49] O. Maler, A. Pnueli and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," *STACS*, LNCS, vol. 900, pp.229–242, 1995.

[50] K.L. McMillan, "Symbolic Model Checking," *Kluwer Academic Publishers*, 1993.

[51] R. Minhas and W.M. Wonham, "Modelling of Timed Discrete-Event Systems," *Proceedings of Thirty-Seventh Annual Allerton Conference on Communication, Control and Computing*, pp.75–84, 1999.

[52] J.S. Ostroff, "Synthesis of controllers for real-time discrete event systems," *Proceedings of 23rd Conference on Information Sciences and Systems*, pp.98–102, 1989.

[53] J.S. Ostroff and W.M. Wonham, "A framework for real-time discrete event control," *IEEE Transactions on Automatic Control*, vol. 35, no. 4, pp.386–397, 1990.

[54] P.J. Ramadge and W.M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp.206–230, 1987.

[55] J. Sifakis, "Modeling real-time systems – challenges and work directions," *EMSOFT*, pp.373–389, 2001.

[56] J. Sifakis, "Scheduler modeling based on the controller synthesis paradigm," LNCS, vol. 2469, pp.107–110, 2002.

[57] O. Sokolsky and S. Smolka, "Local Model Checking for Real-Time Systems," *CAV*, LNCS, vol. 939, pp.211–224, 1995.

[58] A. Srinivasan and S. Baruah, "Deadline-based Scheduling of Periodic Task Systems on Multiprocessors," *Information Processing Letters*, vol. 84, no. 2, pp.93–98, 2002.

[59] H. Wong-Toi, "Symbolic Approximations for Verifying Real- Time Systems," PhD thesis, Stanford University, 1994.

[60] H. Wong-Toi and G. Hoffmann, "The Control of Dense Real-Time Discrete Event Systems," *Technical report*, STAN-CS-92-1411, Stanford University, 1992.

[61] W.M. Wonham, "Design Software XPTTCT," Updated 2004.07.01 and found at www.control.utoronto.ca/DES

[62] W.M. Wonham, "Supervisory control of discrete-event systems," *Department of Electrical and Computer Engineering, University of Toronto*, 2004.