Real-time Scheduler Design for Safety-critical Systems: A Supervisory Control Approach

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy
in
Computer Science and Engineering

Submitted by
Rajesh D

Under the guidance of
Dr. Arnab Sarkar and Dr. Santosh Biswas

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

December, 2018

Abstract

In safety-critical real-time systems, failure to meet task deadlines may lead to consequences that are deemed to be unacceptable. Reliability of such systems are typically guaranteed by enforcing all timing and safety-related requirements to be honored, even under the assumption of the worst-case request arrival behavior and service requirements. Hence, static off-line schedulers are often preferred in safety-critical systems in order to achieve better predictability. In addition, off-line computation also allow exhaustive solution space enumeration to pre-compute optimal schedules at design time, thus ensuring lower design costs through higher resource utilization. To ensure correctness and completeness of the enumeration process, formal mechanisms such as automata/model-based approaches are often preferred. In recent years, researchers have shown that off-line formal approaches such as *Supervisory Control of Timed Discrete Event Systems* (SCTDES) can be used to synthesize optimal schedulers for real-time systems. *In this dissertation, we present a few novel real-time scheduler designs for safety-critical systems consisting of various types of task and execution platform scenarios, using SCTDES as the underlying formalism.*

The entire thesis work is composed of multiple distinct contributions which are categorized into five phases. In the first phase, both non-preemptive as well as preemptive scheduling strategies for uniprocessor systems have been considered. The second phase extends the uniprocessor scheduling mechanisms designed in the first to provide fault-tolerance in homogeneous multiprocessor / multi-core systems, against permanent processor faults. Apart from guaranteeing timing and resource constraints, safety-critical systems implemented on multi-core platforms need to satisfy stringent power dissipation constraints such as *Thermal Design Power* thresholds. Hence, in the third phase, we have developed a scheduler synthesis framework which guarantees adherence to a system level peak power constraint. While the first three phases dealt with the design of scheduling approaches for independent tasks, in the fourth phase, we have endeavored towards the development of an optimal real-time scheduler synthesis scheme for precedence-constrained task graphs executing on homogeneous multi-cores. Further, this scheme has been extended to provide robustness against multiple transient processor faults. In the final phase, we have developed models that are able to accurately capture the execution of tasks on a heterogeneous platform. Experimental results have demonstrated the versatility and efficacy of the proposed scheduler synthesis approaches.

Declaration

I certify that:

a. The work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Rajesh D

Certificate

This is to certify that this thesis entitled, **"Real-time Scheduler Design for Safety-critical Systems: A Supervisory Control Approach"**, being submitted by **Rajesh D**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by him under our supervision and guidance. The thesis, in our opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of our knowledge, it has not been submitted elsewhere for the award of the degree.

....

Dr. Arnab Sarkar
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

....

Dr. Santosh Biswas
Associate Professor
Department of Computer Science and Engineering
IIT Guwahati

Dedicated to

Almighty GOD and all my respected teachers

Whose knowledge, blessing, love and inspiration paved my path of success

Acknowledgments

I wish to express my deepest gratitude to my supervisors, Prof. Arnab Sarkar and Prof. Santosh Biswas for their valuable guidance, inspiration, and advice. I feel very privileged to have had the opportunity to learn from, and work with them. Their constant guidance and support not only paved the way for my development as a research scientist but also changed my personality, ability, and nature in many ways. I have been fortunate to have such advisors who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. Besides my advisors, I would like to thank the rest of my thesis committee members: Prof. J. K. Deka, Prof. P. Bhaduri, and Prof. H. K. Kapoor, for their insightful comments and encouragement. Their comments and suggestions helped me to widen my research from various perspectives.

I would like to express my heartful gratitude to the director, the deans and other managements of IIT Guwahati whose collective efforts has made this institute a place for world-class studies and eduaction. I am thankful to all faculty and staff of Dept. of Computer Science and Engineering for extending their co-operation in terms of technical and official support for the successful completion of my research work. I thank TCS (TATA Consultancy Services) for providing Ph.D. research fellowship for the period July 2014 to June 2018.

I am thankful to my friends Piyoosh, Sanjit, Satish, Sangeet, Sanjay, Mritunjay, Rahul, Sibaji, Anirban, Rana, Madhu and Rohit Verma (Wg Cmdr), Arijit, Kankana and Sayani for supporting and motivating to overcome any problems either in work and otherwise. The countless discussions, sharing ideas has improved our research. I am also grateful to all my seniors, friends and juniors especially Bala, Satisha, Brijesh, Mirza, Rakesh Tripathi, Mohit, Sandeep, Biswajit, Shounak and Sukarn for their unconditional help and support. You made my life at IIT Guwahati a memorable experience.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms

ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Integrated Product*
ATG	*Activity Transition Graph*
BDD	*Binary Decision Diagram*
CE	*Cyclic Executive*
CPU	*Central Processing Unit*
DTM	*Dynamic Thermal Management*
DAG	*Directed Acyclic Graph*
DM	*Deadline Monotonic*
DSP	*Digital Signal Processor*
EDF	*Earliest Deadline First*
FPGA	*Field Programmable Gate Array*
GCD	*Greatest Common Divisor*
GPP	*General Purpose Processor*
GPU	*Graphical Processing Unit*

ILP *Integer Linear Programming*

ICS *Instrument Control System*

LCM *Least Common Multiple*

LDF *Least Density First*

LLF *Least Laxity First*

PID *Proportional-Integral-Derivative*

PAS *PEAK POWER-AWARE SYNTHESIS*

PASS *PEAK POWER-AWARE SYMBOLIC SYNTHESIS*

PTG *Precedence-constrained Task Graph*

RM *Rate Monotonic*

RT *Real-Time*

SCTDES *Supervisory Control of Timed Discrete Event Systems*

SIMD *Single Instruction-stream Multiple Data-stream*

TDP *Thermal Design Power*

TDES *Timed Discrete Event Systems*

WCET *Worst Case Execution Time*

List of Symbols

τ_i	i^{th} task
I	$I = \{\tau_1, \tau_2, \ldots, \tau_n\}$; Set of n tasks
A_i	Arrival time of task τ_i
E_i	Execution time of task τ_i
D_i	Relative deadline of task τ_i (with respect to its arrival)
P_i	Fixed (Minimum) inter-arrival time for periodic (sporadic) task τ_i
V_j	j^{th} processor / processing core
V	$V = \{V_1, V_2, \ldots, V_m\}$; Set of m processors / processing cores
$E_{i,j}$	Execution time of task τ_i on core V_j
Σ_{spe}	Set of *prospective* events
Σ_{rem}	Set of *remote* events
Σ_{uc}	Set of *uncontrollable* events
Σ_c	Set of *controllable* events
Σ_{for}	Set of *forcible* events
Σ_i	Set of events associated with task τ_i

a_i	Arrival (event) of task τ_i
s_i	Start of execution (event) of task τ_i
c_i	Completion of execution (event) of task τ_i
p_i	Acceptance (event) of a task τ_i
r_i	Rejection (event) of a task τ_i
e_i	Execution (event) of a segment of τ_i (in preemptive execution)
c_i	Execution (event) of the last segment of τ_i (in preemptive execution)
f_i	Fault (event) of a processor V_i
$tick$	Passage of one unit time of the global clock
t	Shorthand notation for *tick*
fa_i	Arrival (event) of task τ_i's first instance
$r_{i,j}$	Assignment (event) of task τ_i on the ready queue associated with V_j
$s_{i,j}$	Start of execution (event) of task τ_i on the processing core V_j
$c_{i,j}$	Completion of execution (event) of task τ_i on the processing core V_j
$W_{i,j}$	j^{th} execution window associated with task τ_i
$\mathcal{B}$	Chip-level power cap
$\mathcal{B}_i$	Worst-case instantaneous power consumption of task τ_i
ϕ	A temporal logic formula
ϕ_i	Phase of a periodic task τ_i

Chapter 1

Introduction

Safety-criticality is fast becoming a premier constraint in the design of modern embedded systems. This is more so as embedded computing capabilities are being deployed in complex systems including avionics and automotive, spacecrafts, nuclear reactors etc. where failure to meet stipulated performance specifications may lead to catastrophic consequences [5, 50, 58, 61, 66]. For example, failure of a critical embedded controller on a fly-by-wire avionic system may lead to an aircraft crash. The development of these embedded systems are subjected to precisely stipulated design methodologies so that all performance and reliability related objectives may be honored even under worst-case operating conditions and service requirements. The design may need to satisfy various stringent performance constraints including those related to timeliness, resource utilization, verifiability, fault-tolerance, power dissipation, cost etc.

Many computation activities in embedded systems are dedicated towards the control of dynamically evolving physical processes. The control actions/updates produced by these activities (often called *tasks*) may need to be generated either periodically (in a time-triggered fashion), or in response to abrupt changes in system dynamics (even-triggered). The characteristics of the activities may also vary depending on issues such as, (i) severity of the consequences of missing deadlines associated with the control actions, (ii) interrupts thay may happen in their operating environment, (iii) nature of the underlying hardware processing platform (i.e., single core, homogeneous/heterogeneous multi-core), (iv) behavior of co-executing computation activities etc. Design methodologies dealing with the modeling and synthesis of supervision / control frameworks in embedded computing systems must be appropriately tailored to take care of the specific functional and platform requirements associated with the computation activities of a given system. Formal model based safe design approaches are often employed in the construction of safety-critical embedded systems since such schemes aid in the construction of sound and complete *controllers / supervisors.* Given a set of real-time applications that co-execute on a shared computing platform, design of a supervisor that successfully satisfies all timing, resource and performance related constraints, is ultimately a *scheduler synthesis* problem.

A system is classified as *real-time* if it is characterized by a dual notion of correctness: *logical* as well as *temporal* [73, 82]. Applications such as pacemakers in health-care, fly-by-wire in aircrafts, reactors in nuclear plants, anti-lock braking systems in automobiles etc. are examples of real-time systems [97]. The applications in real-time systems often consist of a set of *recurrent* tasks. Each such task may represent a piece of code (i.e., program) which is triggered by external events that may happen in their operating environment. Each execution instance of the task is referred to as a job. A recurrent task may be *periodic, aperiodic or sporadic*, based on its arrival pattern. *Aperiodic* tasks have no restriction on the arrival pattern of their jobs. *Sporadic* tasks are a special case of aperiodic tasks where consecutive jobs must be separated by a specified minimum inter-arrival time. *Periodic* tasks in turn may be considered as a special case of sporadic tasks whose jobs must be activated at fixed regular intervals [6, 22, 23, 32, 89].

Real-time systems have traditionally been implemented on platforms consisting of a single processor as they allow enhanced predictability and controllability over on-line co-execution behavior of tasks. However, over the years, the industry is witnessing a significant shift in the nature of processing platforms that are used in real-time embedded systems. Homogeneous single cores have slowly given way to multi-core platforms in order to cater to higher computation demands while adhering to restrictions on temperature and/or energy dissipation. In addition, the need to satisfy stringent performance requirements, often along with additional constraints on size, weight, power etc., have ushered in the era of heterogeneous processing platforms in today's complex embedded control systems. For example, a modern System-on-Chip platform can contain multi-core *Central Processing Unit* (CPU) with specialized graphics processing cores, digital signal processing cores, floating-point units, customizable *Field Programmable Gate Array* (FPGA), *Application Specific Integrated Product* (ASIP), *Application Specific Integrated Circuit* (ASIC) etc. Platforms with such varying types of computing elements are called *heterogeneous* (or unrelated) processing platforms. *On a heterogeneous platform, the same piece of code may require different amounts of time to execute on different processing cores* [10]. For example, a task responsible for rendering images may take far less time to execute on a graphics processor compared to a general-purpose CPU, while number-crunching routines would execute more efficiently on CPUs.

Given a set of recurrent tasks and a processing platform, successfully satisfying all task execution and deadline requirements is a *scheduling* problem [36]. Real-time tasks may allow themselves to be scheduled either preemptively or non-preemptively. In *preemptive* scheduling, execution of a job can be interrupted before completion. On the contrary, a job's execution cannot be interrupted until its completion in case of *non-preemptive* scheduling. Depending on whether the schedule is generated statically at design-time or dynamically at run-time, scheduling algorithms are differentiated as *off-line* or *on-line* schedulers.

In safety-critical real-time systems, failure to meet task deadlines may lead to consequences that are determined to be unacceptable [95]. In addition, it is often advisable that all timing requirements should be guaranteed off-line, before putting the safety-critical system in operation. Hence, there is a necessity for developing off-line exhaustive enumeration techniques to pre-compute optimal schedules at design time and use them during on-line execution. It may also be noted that exhaustive off-line schedule generators also bring in the ability to generate all possible feasible solutions. This empowers the designer to select one or more scheduling solutions which best fits the requirements of a given system scenario under consideration. For example, the designer may select a schedule which minimizes the number of preemptions, guarantees work-conservation etc.

In order to ensure the correctness and completeness of an exhaustive off-line schedule enumeration process, formal approaches are often preferred.

In recent years, many researchers have shown that off-line formal approaches such as *Supervisory Control of Timed Discrete Event Systems* (SCTDES) [18] can be used for the design of hard real-time schedulers [29,53,85,87,105–107]. SCTDES based schedulers essentially determine only and all the feasible execution sequences corresponding to the specifications of a given task set and platform, by carefully discarding the prefixes of all scheduling sequences that may lead to unsafe situations in future. In addition, this approach allows the leverage of narrowing down further on the subset of the most desired schedules with respect to a chosen set of softer constraints related power, weight, cost etc. Chen and Wonham presented an optimal uniprocessor scheduling strategy for *non-preemptive periodic tasks* in [29]. Janarthanan et al. extended the SCTDES framework by introducing the notion of priorities in order to develop a formal and unified procedure for the scheduling of *periodic tasks, both preemptively as well as non-preemptively*, on uniprocessors [53]. Park and Cho [85] attempted to synthesize a *preemptive scheduler for dynamically arriving sporadic tasks* on uniprocessor systems. Recently, Wang et al. [105] extended the models presented in [29, 53] to handle *non-preemptive periodic tasks with multiple periods.* The notion of multiple periods allows tasks to dynamically reconfigure their periods at run-time. They have also proposed a framework for *priority-free, conditionally-preemptive scheduling of real-time tasks* in [106, 107].

The research presented in this dissertation focuses towards the theoretical and practical aspects of scheduling mechanisms for safety-critical systems and developing SCTDES-based optimal scheduler synthesis strategies subject to various constraints (such as fault-tolerance, precedence-constraints, power minimization etc.).

1.1 Challenges

A scheduling mechanism which efficiently caters to diverse applications and serves the variety of processing platforms in today's safety-critical systems, must meet several challenges. We now enumerate a few such important challenges and discuss them [23].

1. **Timing requirements:**
   Real-time systems are characterized by computational activities with stringent timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a task is the deadline, which represents the time before which a process should complete its execution without causing any damage to the system. Another timing characteristic that can be specified on a real-time task concerns the regularity of its activation. In particular, tasks can be defined as *aperiodic*, or *sporadic*, or *periodic*. Scheduling schemes for safety-critial real-time systems must be able to guarantee the timing requirements (i.e., deadlines) associated with various types of tasks that co-exist in the system.

2. **Predictability:**
   Scheduler designs for safety-critical systems must deliver guaranteed co-execution behavior and performance during on-line operation of the system, under pre-specified load and failure conditions. To achieve a desired level of performance, the system must be analyzable to predict the consequences of any scheduling decision. An important mechanism for enhancing predictability is to conduct static analysis followed by off-line schedule generation so that performance of the system is guaranteed a priori even under worst-case task arrival behavior and service requirements.

3. **Resource constraints:**
   Safety-critical systems are implemented on platforms consisting of limited number of processing elements (i.e., resources). For example, providing a lot of redundant hardware is not always possible in cost-sensitive safety-critical systems like cars, where a cost differential of even a hundred dollars can make a commercial difference [31, 61, 62, 93]. In addition, the nature of processing elements are also changing over the years. Specifically, homogeneous single cores have slowly given way to multi-core platforms in order to cater to higher computation demands while adhering to restrictions on power/energy dissipation. Scheduling schemes designed for real-time systems must be able to effectively utilize the processing capacity of

underlying platform consisting of limited number of processing resources, to satisfy the constraints associated with a given real-time task set.

4. **Fault-tolerance:**
   Apart from guaranteeing the timely execution of tasks in a resource-constrained environment, ensuring proper functioning of the system even in the presence of faults (i.e., fault tolerance) has currently become a design constraint of paramount importance. Specifically, the processors on which the tasks are executed, are subject to a variety of faults. Such faults are broadly classified to be either permanent or transient [62]. Permanent processor faults are irrecoverable and do not go away with time. On the other hand, transient faults are short-lived (momentary) and their effect goes away after some time. Thus, robustness against permanent/transient faults is emerging as a very important criterion in the design of safety critical real-time systems.

5. **Power constraints:**
   Safety-critical systems implemented on multi-core platforms need to satisfy stringent power dissipation constraints such as Thermal Design Power (TDP) thresholds used by chip manufacturers [1]. Power dissipation beyond TDP may trigger Dynamic Thermal Management (DTM) in order to ensure thermal stability of the system. However, application of DTM makes the system susceptible to higher unpredictability and performance degradations for real-time tasks [67, 78, 99]. This necessitates the development schedulers that can guarantee adherence to a system level peak power constraint.

6. **Energy Minimization:**
   If all timing and safety related constraints are satisfied, then the system designer can focus on further optimization, such as minimizing the overall energy consumption to prolong the battery lifetime of systems or to cut the power bills in servers.

1.2 Objectives

The principle aim of this dissertation has been to investigate the theoretical and practical aspects of scheduling strategies keeping in view the challenges/hurdles discussed in the previous section. In particular, the objectives of this work may be summarized as follows:

1. Development of generic execution models for real-time tasks (such as periodic, sporadic, aperiodic) executing in preemptive/non-preemptive fashion on uniprocessor systems.

2. Extending the models developed under uniprocessor resource-constraint to multiprocessor platforms consisting of homogeneous as well as heterogeneous processing cores.

3. Incorporating fault-tolerance mechanism in the developed models to handle transient as well as permanent processor faults that may affect the execution of real-time tasks.

4. Design and implementation of peak power-aware scheduler synthesis framework which guarantees adherence to a system level peak power constraint while allowing optimal resource utilization in multi-cores.

5. Empowering the scheduler synthesis scheme with a *Binary Decision Diagram* (BDD) based symbolic computation mechanism to control the exponential state-space complexity of the exhaustive enumeration-oriented synthesis methodology.

1.3 Proposed Framework

This research work conducts optimal off-line scheduler synthesis for safety-critical systems. The pictorial representation of the proposed framework is presented in Figure 1.1. The proposed framework receives the information regarding *task set*, *processing platform*, *constraints* and *modeling style*, as input parameters. Here, task set may consist of a set of independent / precedence-constrained tasks with aperiodic / periodic / sporadic

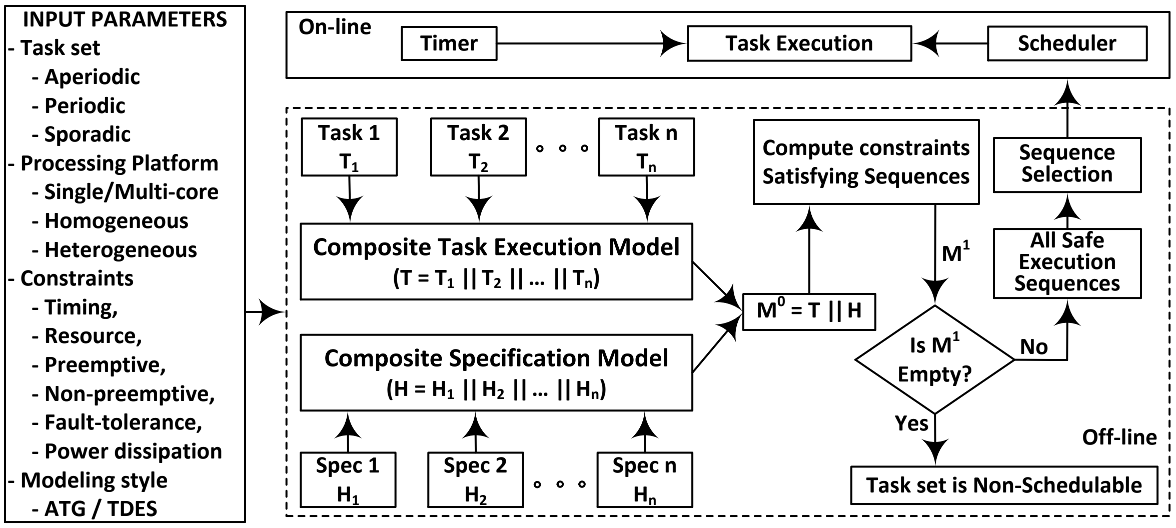


Figure 1.1: *Pictorial representation of the proposed scheduler synthesis framework*

arrival pattern. Each task is characterized by its arrival time, execution and deadline requirements. The processing platform may consist of a single processor or multiple homogeneous / heterogeneous processing cores. The constraints for scheduler synthesis can be timing, resource, fault-tolerance, power dissipation etc.

Given the task set, processing platform and a set of constraints, our framework first constructs the (finite state automata based) models for each task, processor and constraints. Such a construction process may either start from an (untimed) *Activity Transition Graph* (ATG) or *Timed Discrete Event Systems* (TDES). Suppose ATG is used to represent the models. Then, their corresponding TDES representation will be derived from ATG, by following the construction rules prescribed in Brandin-Wonham framework [18]. Given n individual TDES models T_1, T_2, ..., T_n, corresponding to tasks τ_1, τ_2, ... τ_n in the system, a *synchronous product* [18] (denoted by $||$) composition $T = ||_{i=1}^{n} T_i$ on the models gives us the composite model representing the *concurrent execution of all tasks*. Similarly, the composite specification model H can be obtained from individual models that captures the constraints such as timing, resource etc.

In order to find all sequences in the composite system model T that satisfy the constraints modelled by the composite specification model H, their composite model is obtained. That is, $M^0 = T||H$. The model M^0 may consist of deadlock states in it (which will block the execution of system) and hence, to obtain a non-blocking sub-part of M^0, we apply supervisor synthesis algorithm [77]. The resulting model M^1 contains all feasible scheduling sequences that satisfy the given constraints. It may happen that the state set of M^1 to be empty, which implies that the given task set is non-schedulable under the given set of constraints. Since, M^1 contains all feasible scheduling sequences, it allows the leverage of narrowing down further on the subset of the most desired schedules with respect to a chosen set of softer constraints related power, weight, cost etc. Finally, scheduling sequences obtained using our framework are then employed to supervise on-line task execution.

It may be noted that the number of states in the composite models increases *exponentially* as the number of tasks, processors, execution times increases. So, the proposed scheme may be highly time consuming and unacceptably memory intensive even for moderately large systems, thus severely restricting scalability, especially for industrial applications with many tasks. Over the years, *Binary Decision Diagram* (BDD) [21]) based symbolic synthesis mechanisms have proved to be a key technique towards the efficient computation of large finite state machine models including SCTDES based supervisors [77]. This observation motivated us to derive a symbolic computation based adaptation of the proposed scheduler synthesis framework. Hence, we have also transformed our proposed framework to compute the supervisor symbolically.

1.4 Contributions

As a part of the research work, multiple scheduler design schemes for safety-critical systems have been developed based on SCTDES framework.

1. **Scheduling of Real-time Tasks on a Uniprocessor Platform**
   We have developed models that can be used to synthesize schedulers for aperiodic / sporadic tasks executing (non-preemptively / preemptively) on uniprocessors, and they are listed as follows:

   (a) **Real-time Scheduling of Non-preemptive Sporadic Tasks**
   This work proposes an optimal scheduler synthesis framework for non-preemptive

sporadic real-time tasks executing on uniprocessors. Although, in recent years, there has been a few significant works dealing with real-time scheduling using SCTDES, this is possibly the first work which addresses the scheduler synthesis problem for sporadic tasks.

With respect to our proposed framework (ref. Figure 1.1), this work considers the following input parameters: aperiodic as well as sporadic task set, uniprocessor platform, timing and resource constraints, and directly represents models in TDES form. Here, task execution model captures the execution time, co-execution of tasks and uniprocessor resource constraint. Deadline constraint associated with a real-time task is captured by a specification model.

(b) **Work-conserving Scheduler Synthesis of Preemptive Tasks**
This work attempts to synthesize a work-conserving preemptive scheduler for a set of real-time sporadic tasks executing on uniprocessors. Work-conserving approach allows the synthesis of schedules which avoid processor idling in the presence of ready to execute tasks.

2. **Fault-tolerant Scheduling of Preemptive tasks on Multiprocessors**
A methodology for synthesizing an optimal preemptive multiprocessor aperiodic task scheduler using a formal supervisory control framework, is presented. The scheduler can tolerate single/multiple permanent processor faults. With respect to scheduler synthesis schemes developed for uniprocessor systems, the state space of the final supervisor model increases drastically as the number of processors in the system increases. Hence, the synthesis framework has been further empowered with a novel BDD-based symbolic computation mechanism to control the exponential state-space complexity of the optimal exhaustive enumeration-oriented synthesis methodology [21, 76, 77].

3. **Power-aware Real-time Scheduling**
This work presents a scheduler synthesis framework which guarantees adherence to a system level peak power constraint while allowing optimal resource utilization

in multi-cores. All steps starting from individual models to construction of the scheduler have been implemented through BDD-based symbolic computation [101]. The synthesis framework has been extended to handle tasks with phased execution behavior [67].

With respect to our framework summarized in Figure 1.1, this work considers the following input parameters: periodic task set, homogeneous multi-core platform, timing, resource and power constraints, and represents models in ATG form. Here, task execution model captures the execution time and deadline of a task. Uniprocessor resource constraint is captured by specification model. From the resulting set of timing and resource constraint satisfying sequences, we conduct a search technique to filter our the sequences that violate power constraint and we also find execution sequences that dissipate minimal power.

4. **Scheduling of Non-preemptive Tasks on Heterogeneous Multi-cores**
   Real-time systems are increasingly being implemented on heterogeneous multi-core platforms in which the same piece of software may require different amounts of time to execute on different processing cores. Computation of optimal schedules for such systems is non-trivial and prohibitively expensive to be conducted on-line. This work presents a systematic way of designing an optimal off-line scheduler for systems consisting of a set of independent non-preemptive periodic tasks executing on heterogeneous multi-cores. Existing SCTDES based scheduler synthesis mechanisms do not provide the flexibility to model heterogeneity of the underlying platform. The models developed in this work are able to accurately capture the execution of tasks on a heterogeneous platform. The models can then be employed to synthesize a scheduler to supervise on-line task execution.

5. **Static Scheduling of Parallel Real-time Tasks**
   All our earlier works assume the tasks in a given real-time system to be *independent.* However, many real-time applications such as radar tracking, autonomous driving, and video surveillance, are highly parallelizable [44]. One of the most

generic mechanisms for modeling parallel real-time applications is *Precedence-constrained Task Graph* (PTG)/*Directed Acyclic Graph* (DAG) [34]. This work deals with the synthesis of an optimal real-time scheduler for PTGs executing on homogeneous multi-cores. We extend our scheme to provide robustness against multiple transient processor faults. Further, we show that the proposed framework can be adapted to derive the best set of schedules with respect to one or multiple performance objectives and demonstrate this idea by devising search strategies to obtain schedules that, (i) minimize makespan and (ii) maximize fault-tolerance. Conducted experiments reveal the practical efficacy of our scheme.

1.5 Organization of the Thesis

The thesis is organized into eight chapters. A summary of the contents in each chapter is as follows:

- **Chapter 2**: *Background on Real-time Systems and Supervisory Control*
  This chapter presents background on real-time systems and supervisory control of timed discrete event systems. In particular, we try to present the vocabulary needed to understand the following chapters.

- **Chapter 3**: *Scheduling of Sporadic Tasks on Uniprocessors*
  In the third chapter, we present scheduler synthesis mechanisms for real-time sporadic tasks executing on uniprocessor systems. First, the scheduling of non-preemptive version of sporadic tasks is presented. Then, a work-conserving scheduler synthesis framework for preemptive sporadic tasks is presented.

- **Chapter 4**: *Fault-tolerant Scheduling of Aperiodic Tasks*
  Research conducted in the fourth chapter deals with the fault-tolerant scheduler synthesis mechanism. This work considers a set of dynamically arriving aperiodic tasks executing on a homogeneous multiprocessor platform. By assuming that at most one processor in the system may undergo a permanent fault at any instant of time, a single permanent processor fault tolerant scheduler synthesis scheme has

been presented. Then, this scheme has been extend to tolerate multiple permanent processor faults.

- **Chapter 5**: *Power-aware Scheduling on Homogeneous Multi-cores*
  This chapter proposes a formal scheduler synthesis framework which guarantees adherence to a system level peak power constraint while allowing optimal resource utilization in multi-cores. Further, the method to synthesis supervisor for tasks with phased execution behavior, is also presented. Practical efficacy of our proposed scheme has been illustrated using simulation based experiments conducted using real-world benchmark programs.

- **Chapter 6**: *Scheduling on Heterogeneous Multi-cores*
  This chapter proposes a systematic way of designing a scheduler for systems consisting of a set of independent non-preemptive periodic tasks executing on heterogeneous multi-cores. We have also discussed the optimality and working of the scheduler synthesized using our proposed models.

- **Chapter 7**: *Scheduling of Parallel Real-time Tasks*
  Chapter 7 primarily delves towards the scheduler synthesis for parallel real-time tasks modeled by precedence-constrained task graphs, executing on homogeneous multi-cores. We extend our scheme to provide robustness against multiple transient processor faults. Further, we show that the proposed framework can be adapted to derive the best set of schedules with respect to one or multiple performance objectives and demonstrate this idea by devising search strategies to obtain schedules that, (i) minimize makespan and (ii) maximize fault-tolerance.

- **Chapter 8**: *Conclusion and Future Work*
  The thesis concludes with this chapter. We discuss the possible extensions and future works that can be done in this area.

Chapter 2

Background: Real-time Systems, Supervisory Control

This dissertation is oriented towards the synthesis of schedulers for safety-critical real-time systems consisting of a set of tasks with stringent timing constraints, executing on a shared computing platform with limited number of processing elements. The previous chapter provided a view of the challenges imposed by the diversity in the types of computation activities, nature of computing platform, performance requirements etc. towards the synthesis of schedulers for safety-critical real-time systems.

In this background chapter, we present a brief introduction to the definitions related to real-time systems and supervisory control. In particular, we try to present the vocabulary needed to understand the following chapters and explain how these systems can be modeled so as to enable researchers to design efficient real-time schedulers using supervisory control. We first provide an overview on the structure of real-time systems. Then, the evloution of scheduling algorithms for real-time systems implemented on uniprocessors to homogeneous and heterogeneous multiprocessors are discussed. Subsequently, we introduce the formal definitions and synthesis algorithms related to supervisory control. Also, we illustrate the scheduler synthesis process using the open-source design software TTCT [2] (developed by Systems Control Group at the University of Toranto). It may be noted that the synthesis algorithms introduced in this chapter will be appropriately referred later in this dissertation, during the synthesis of supervisors.

2.1 Real-time Systems

Typically, real-time systems are composed of three distinct layers [80]:

- **An application layer** which consists of set of all applications that should be executed.
- **A real-time scheduler** which takes the scheduling decisions and provides services to the application layer.
- **A hardware platform** which includes the processors (among other things such as memories, communication networks, etc.).

We will now present each of these layers in detail and introduce the theoretical models enabling researchers to analyze these systems and design efficient schedulers for real-time systems to schedule the application tasks on the hardware platform.

2.1.1 The Application Layer

The application layer is composed of all the applications that the system need to execute. The applications in real-time systems often consist of a set of *recurrent* tasks. Each such task may represent a piece of code (i.e., program) which is triggered by external events that may happen in their operating environment. Each execution instance of the task is referred to as a *job*. We now discuss the set of definitions related to a real-time task.

2.1.1.1 A Real-time Task Model

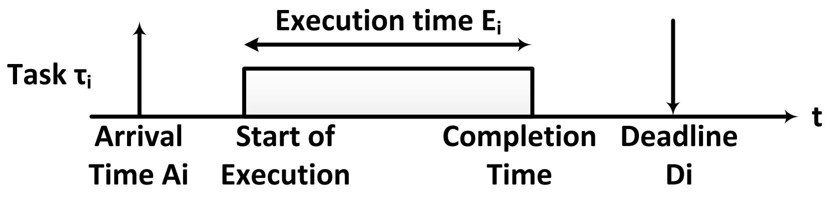


Figure 2.1: *Temporal Characteristics of real-time task* τ_i

Formally, a real-time task (denoted by τ_i; shown in Figure 2.1) can be characterized by the following parameters:

1. **Arrival time** (A_i) is the time at which a task becomes ready for execution. It is also referred as *release time* or *request time.*

2. **Start time** is the time at which a task starts its execution.

3. **Computation time** or *Execution time* (E_i) is the time necessary to the processor for executing the task without interruption.

4. **Finishing time** or *Completion time* is the time at which a task finishes its execution.

5. **Deadline** is the time before which a task should complete its execution requirement. If it is measured with respect to system start time (at 0), it will be called as *absolute deadline*(d_i). If it is measured with respect to its arrival time, it will be called as *relative deadline*(D_i).

6. **Response time** is the difference between the finishing time and the arrival time.

7. **Worst-case execution time** is the largest computation time of a task among all its possible execution.

8. **Laxity** or *Slack time* is the maximum time a task can be delayed on its activation to complete within its deadline: $D_i - E_i$.

9. **Priority** is the importance given to a task in context of the schedule at hand.

A real-time task τ_i can be classified as periodic, aperiodic and sporadic based on regularity of its activation [23].

1. **Periodic** tasks consist of an infinite sequence of identical activities, called instances or *jobs*, that are regularly activated at a constant rate. The activation time of the first periodic instance is called *phase* (ϕ_i). The activation time of the k^{th} instance is given by $\phi_i + (k-1)P_i$, where P_i is the activation period (*fixed inter-arrival time*) of the task.

2. **Aperiodic** tasks also consist of an infinite sequence of identical jobs. However, their activations are not regularly interleaved.
3. **Sporadic** tasks consist of an infinite sequence of identical jobs with consecutive jobs seperated by a *minimum inter-arrival time.*

There are three levels of constraint on task deadline:

1. *Implicit Deadline:* all task deadlines are equal to their periods ($D_i = P_i$).
2. *Constrained Deadline:* all task deadlines are less than or equal to their periods ($D_i \leq P_i$).
3. *Arbitrary Deadline:* all task deadlines may be less than, equal to, or greater than their periods.

We now provide few other defintions related to tasks and taskset.

Utilization: The utilization of a (implicit deadline) task τ_i is given by $U_i = E_i/P_i$. In case of constrained deadline, $U_i = E_i/D_i$.

Utilization Factor U: Given a set of (implicit-deadline) tasks, $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, U is the fraction of the processor time spent in the execution of the task set. $U = \sum_{i=1}^{n} E_i/P_i$

Hyperperiod: It is the minimum interval of time after which the schedule repeats itself. For a set of periodic tasks (with periods $P_1, P_2, \ldots, P_n$) activated simultaneously at $t = 0$, the hyperperiod is given by the least common multiple of the periods.

Static and Dynamic Task System: In a static task system, the set of tasks that is executed on the platform is completely defined before start running the application. In a dynamic task system, some tasks may experience a modification of their properties while other tasks leave or join the executed task set at run-time. *In this thesis, we deal only with the static task system.*

2.1.2 A Real-time Scheduler

A real-time scheduler acts as an interface between applications and hardware platform. It configures and manages the hardware platform (e.g., manage hardware interrupts, hardware timers, etc.). More importantly, it schedules the tasks using a real-time scheduling algorithm. The set of rules that, at any time, determines the order in which tasks are exeucted is called a *scheduling algorithm.*

Given a set of tasks, $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion. A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints. A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule. Scheduling algorithms can be classified based on preemption.

1. **Preemptive**: tasks can be interrupted at any time (so that the processor may be assigned to another task) and resumed later.

2. **Non-preemptive**: once activated, a task must be continuously executed until its completion.

In addition to the above classification, depending on whether the schedule is generated statically at design-time or dynamically at run-time, scheduling algorithms are differentiated as *off-line* or *online.* The scheduling decisions are based on the tasks priorities which are either assigned statically or dynamically. *Static priority* driven algorithms assigned fixed priorities to the tasks before the start of the system. *Dynamic priority* driven algorithms assign the priorities to tasks during run-time. A scheduling algorithm is said to be *optimal* if it is able to find a feasible schedule, if one exits. An algorithm is said to be heuristic if it is guided by a heuristic function in taking its scheduling decisions. A *heuristic* algorithm tends toward the optimal schdule, but does not guarantee finding it. In *work-conserving* scheduling algorithm, processor is never kept idle while there exist a task waiting for a execution on the processor.

2.1.3 Processing Platform

The term *processor* refers to a hardware element in the platform which is able to process the execution of a task.

1. **Uniprocessor** system can only execute one task at a time and must switch between tasks.

2. **Multiprocessor** system will range from several seperate uniprocessors tightly coupled using high speed network to multi-core. It can be classified as follows:
    (a) **Homogeneous:** The processors are *identical*, i.e., they all have the same functional units, instruction set architecture, cache sizes and hardware services. The rate of execution of all tasks is the same on all processors. Hence, the worst-case execution time of a task is not impacted by the particular processor on which it is executed.
    (b) **Uniform:** The processors are *identical* - but they are running at different frequencies. Hence, all processors can execute all tasks but the speed at which they are executed and their worst-case execution time vary in function of the processor on which they are executing.
    (c) **Heterogeneous:** The processors are different, i.e., processors may have different configurations, frequencies, cache sizes or instruction sets. Some tasks may therefore not be able to execute on some processors in the platform, while their execution speeds (and their worst-case execution times) may differ on the other processors.

In this thesis, we consider the scheduler synthesis for real-time systems implemented on uniprocessor to homogeneous / heterogeneous processing platforms.

2.1.4 A brief survey of scheduling algorithms

Initially, we review the theory of well-known uniprocessor scheduling algorithms. Later, we proceed towards multiprocessor scheduling algorithms.

2.1.4.1 Uniprocessor Preemptive scheduling algorithms

Cyclic Executive **(CE)**

1. Assumptions: Set of independent periodic tasks with common arrival time and implicit-deadline.
2. Clock driven (or time driven) *off-line* algorithm.

3. Temporal axis is divieded into slots of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements [23].

4. Duration of the time slot is called as *Minor cycle*, whereas the minimum interval of time after which the schedule repeats itself (the hyperperiod) is also called *Major cycle*. Minor cycle = $GCD\{P_1, P_2, ..., P_n\}$. Major cycle = $LCM\{P_1, P_2, ..., P_n\}$. (*Greatest Common Divisor* (GCD): Greatest Common Divisor, *Least Common Multiple* (LCM): Least Common Multiple).

5. Most used approach to handle periodic tasks in defense systems due to its *predictability*.

6. *Sensitive to application changes.* If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch.

Rate Monotonic **(RM) [73]**

1. Assumptions: Set of independent periodic tasks with common arrival time and implicit-deadline.

2. Tasks with higher request rates (that is, with shorter periods) will have higher priorities. Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ be the set of periodic tasks ordered by increasing periods, with τ_n being the longest period. According to the RM, τ_n will be the task with lowest priority.

3. RM is a *fixed-priority* assignment: a priority P_i is assigned to the task before execution and does not change over time. RM is not a work-conserving algorithm.

4. Liu and Layland [73] showed that RM is *optimal* among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM.

5. Liu and Layland also derived the least upper bound of the processor utilization factor for a generic set of n periodic tasks: $U_{lub} = n(2^{1/n} - 1)$, which is also known as *LL-Bound*. For $n \to \infty$, $U_{lub} = ln2 = 0.693$.

6. If the given task set satisifes the LL-Bound test, then it is *schedulable*. However, LL-bound test is *only a sufficiency test, not necessary*.

7. Derivation of U_{lub} is based on *critical instant* of a task. It is the arrival time that produces the largest task response time. *Worst-case response time of a task occurs when it is released simultaneously with all higher-priority tasks.*

Earliest Deadline First **(EDF) [73]**

1. Assumptions: Set of independent tasks. Tasks can be aperiodic, periodic, sporadic.

2. Tasks with earlier (absolute) deadlines will be executed at higher priorities.

3. EDF is a *dynamic* priority assignment. It does not make any specific assumption on the periodicity of tasks. Unlike RM, EDF is a work-conserving algorithm.

4. A set of periodic tasks is *schedulable* with EDF if and only if $\sum_{i=1}^{n} C_i/P_i \leq 1$.

5. EDF doesn't make any assumption about the periodicity of the tasks. Hence it can be used for scheduling of periodic as well as aperiodic tasks. In addition to this, EDF is *optimal* among dynamic priority scheduling algorithms.

Least Laxity First **(LLF) [37]**

1. The laxity of a real-time task τ_i at time t, $X_i(t)$ is defined as follows: $L_i(t) = D_i(t) - E_i(t)$, where $D_i(t)$ is the (relative) deadline by which the task must be completed and $E_i(t)$ is the amount of computation remaining to complete. In other words, the laxity is the time left by its deadline after the task is completed assuming that the task could be executed immediately without preemption.

2. A task with zero laxity must be scheduled right away and executed with no preemption to meet the deadline.

3. A task with negative laxity indicates that the task will miss its deadline. Therefore, the laxity of a task is a measure of its urgency in deadline-driven scheduling.

4. *The preemptive LLF scheduling algorithm always executes a task whose laxity is least.* This algorithm has beeen proved to be *optimal.*

Deadline Monotonic **(DM) [68]**

1. DM is a constrained deadline version of RM scheduling.

2. Each task is assigned a *static* priority inversely proportional to its deadline D_i.

3. Schedulability of a given task set can be verified by $\sum_{i=1}^{n} E_i/D_i \leq n(2^{1/n} - 1)$. However, as we mentioned in RM, it is a *pessimistic* bound.

2.1.4.2 Multiprocessor scheduling algorithms

In this subsection, we briefly summarize the scheduling algorithms that can be used to schedule tasks on homogeneous as well as heterogeneous multiprocessor systems. A comprehensive survey of multiprocessor scheduling algorithms can be found in [36]. Traditionally, schedulers for real-time tasks on multi-cores/multi processing cores make use of either a *partitioned* or *global* approach. In a *fully partitioned* approach, all jobs of a task are assigned to a single processor. This approach has the advantage of transforming the multiprocessor scheduling problem to a set of uniprocessor ones. Hence, well known optimal uniprocessor scheduling approaches such as Earliest Deadline First (EDF), Rate Monotonic (RM) [73] etc. may be used. As each task runs only on a single processor, there is no penalty due to inter-processor task migrations. However, one of the drawbacks of *partitioning* is that the general problem is provably *NP-hard* both for homogeneous and heterogeneous multi-cores [26]. Recently, Baruah et al. presented an *Integer Linear Program* (*Integer Linear Programming* (ILP)) based optimal solution to the partitioning problem for heterogeneous platforms [13]. However, solving ILPs for systems with a large number of tasks and processing cores may incur prohibitively expensive computational overheads in many real-time systems.

Unlike partitioning, a *global scheduling* methodology allows inter-processor migration of tasks from one processor to another during execution. The proportional fair algorithm *Pfair* [14] and it's work conserving version *ERFair* [7] are the first known optimal real-time *homogeneous* (i.e., identical) multiprocessor schedulers. Given a set of n implicit-deadline periodic tasks to be executed on m *homogeneous* processors, ERFair scheduler ensures that the minimum rate of progress for each task is proportional to a parameter called the task's weight wt_i, which is defined as the ratio of it's execution requirement E_i and period P_i. That is, in order to satisfy ERFairness, each task τ_i should complete at least $(\sum_{i=1}^{n}(E_i/P_i) \times t)$ part of it's total execution requirement E_i, at any time instant t, subsequent to the start of the task at time s_i. Following the above criterion, ERFair is able to deliver optimal/full resource utilisation; hence, any task set is guaranteed to be schedulable provided: $\sum_{i=1}^{n}(E_i/P_i) \leq m$. However, such a strict fairness criterion imposes heavy overheads including unrestricted preemptions/migrations.

Global Scheduling on Heterogeneous Multiprocessors: Based on a global approach, the problem of optimally scheduling independent preemptive jobs on heterogeneous multiprocessor systems is solved by Lawler et al. in [65]. The paper proposes an approach that can schedule a set of n aperiodic jobs on m heterogeneous processing cores requiring no more than $O(m^2)$ preemptions. According to [10], this result can easily be extended to optimally schedule periodic tasks. However, the resulting schedule is likely to contain a huge number of preemptions and interprocessor migrations, which makes it impractical. Raravi et al. [93] presented a task-to-core assignment algorithm for a heterogeneous platform consisting of two different types of processing cores (for example, ARM's big.LITTLE [10]). Subsequent to the partitioning of tasks among processing cores, the heterogeneous scheduling problem is transformed to homogeneous multi-core scheduling. Then, they have applied well known optimal homogeneous multi-core scheduling techniques such as PFfair [14], ERfair [7], DPfair [45] etc. Recently, Chwa et al. extended the DPfair [45] scheduling strategy and presented a global optimal algorithm called *Hetero-Fair* for the scheduling of periodic tasks on heterogeneous multi-cores with two-types of processing cores [33].

2.2 Supervisory Control of Timed Discrete Event Systems

Discrete Event Systems (DES) is a discrete state space, event driven system that evolves with the occurrence of events [25]. In timed model of DES, both logical behavior and timing information are considered. Supervisory control of a TDES is timely disablement or enforcement of certain events in the transition structure of the TDES such that it's behavior meets certain specifications.

2.2.1 Activity Transition Graph (ATG)

According to supervisory control theory, construction of an ATG is the first step towards synthesis of the supervisor. ATG can be used to represent the task execution behavior as well as specifications related to deadline, resource, faul-tolernace, precedence-constraints etc. Given a set of ATGs, an automated design procedure can be used to synthesize supervisor. In a real-time system consisting of a set of tasks, this design procedure allows the determination of all the feasible execution sequences corresponding to the specifications of a given task set and platform, by carefully discarding the prefixes of all scheduling sequences that may lead to unsafe situations in future.

According to the Brandin-Wonham framework [18], a TDES may be represented by an (untimed) *Activity Transition Graph* (ATG) described as:

$$G_{act} = (A, \Sigma_{act}, \delta_{act}, a_0, A_m),$$

where, A is a finite set of *activities*, Σ_{act} is a finite set of *events*, $\delta_{act} : A \times \Sigma_{act} \to A$ is an *activity transition function*, $a_0 \in A$ is the *initial activity*, and $A_m \subseteq A$ is the subset of *marker* activities. Let $\mathbb{N} = \{0, 1, 2, ...\}$. Each $\sigma \in \Sigma_{act}$ is associated with a *lower* ($l_\sigma \in \mathbb{N}$) and an *upper time bound* ($u_\sigma \in \mathbb{N} \cup \{\infty\}$). An event $\sigma \in \Sigma_{act}$ may either be *prospective* (Σ_{spe}) or *remote* (Σ_{rem}), with finite ($0 \leq l_\sigma \leq u_\sigma < \infty$) and infinite ($0 \leq l_\sigma \leq u_\sigma = \infty$) upper time bounds, respectively. The triple $(\sigma, l_\sigma, u_\sigma)$ is called a timed event. The *timer interval* for $\sigma \in \Sigma_{spe}$ is represented by T_σ and is defined to be $[0, u_\sigma]$ if $\sigma \in \Sigma_{spe}$ (respectively, $[0, l_\sigma]$ if $\sigma \in \Sigma_{rem}$).

To use TDES models for supervisory control, Σ_{act} is also categorized as Σ_{con} (*controllable*), Σ_{unc} (*uncontrollable*), and Σ_{for} (*forcible*). 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. SCTDES includes a mechanism for forcing certain events to occur before a specific time instant. Specifically, forcible events can preempt a tick of the global clock. In this thesis, we are concerned with tasks having a finite deadline and hence, we assume all timed controllable events to be forcible.

2.2.2 Timed Discrete Event Systems (TDES)

A TDES G is derived from its corresponding ATG G_{act} and represented by,

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

The *state set* $Q = A \times \prod\{T_\sigma | \sigma \in \Sigma_{act}\}$, where a state $q \in Q$ is of the form $q = \{a, \{t_\sigma | \sigma \in \Sigma_{act}\}\}$ in which $a \in A$ and $t_\sigma \in T_\sigma$ is the *timer* of event σ. The *initial state* $q_0 \in Q$ is defined as $q_0 = \{a_0, \{t_{\sigma 0} | \sigma \in \Sigma_{act}\}\}$, where $t_{\sigma 0}$ is u_σ if $\sigma \in \Sigma_{spe}$ (respectively, l_σ if $\sigma \in \Sigma_{rem}$). The *marker state set* $Q_m \subseteq Q$ is given by $Q_m \subseteq A_m \times \prod\{T_\sigma | \sigma \in \Sigma_{act}\}$. $\Sigma = \Sigma_{act} \dot\cup \{tick\}$, where *tick* denotes the passage of one unit time of the global clock. $\delta : Q \times \Sigma \rightarrow Q$ is the *state transition function* which is defined in terms of δ_{act} and the time bounds [18].

In case of $\sigma \in \Sigma_{spe}$, $\delta(q, \sigma)$ is defined, provided $q = (a, \_)$, $\delta_{act}(a, \sigma)$ is defined, and $0 \leq t_\sigma \leq \mu_\sigma - l_\sigma$. An event $\sigma \in \Sigma_{spe}$ is *enabled* at $q = (a, \_) \in Q$ if $\delta_{act}(a, \sigma)$ is defined, and is *disabled* otherwise. An enabled event σ (either *tick* or in A) is *eligible* if $\delta(q, \sigma)$ is defined. Only eligible events can actually occur. The *timer mechanism* can be summarized as: Once an event σ is enabled, the timer t_σ of σ is decremented by one time unit at every subsequent *tick* of the clock, until either, (i) t_σ reaches zero (at which point σ is forced to occur), or (ii) σ occurs, or (iii) σ is *disabled* due to the occurrence of some other transition to a new activity. After all these cases, t_σ is reset to its default value (i.e., $t_\sigma = u_\sigma$ if $\sigma \in \Sigma_{spe}$). We use $q \xrightarrow{\sigma} q'$ to denote $\delta(q, \sigma) = q'$.

2.2.3 Example

Let us consider a single instance of a task τ_i (i.e., job) with arrival time A_i, execution time E_i and deadline D_i. The ATG model and its corresponding TDES model, for τ_i are shown in Figures 2.2 and 2.3, respectively. Here, the set of activities $A = \{$IDLE, READY, EXECUTING, COMPLETION$\}$, the event set $\Sigma_{act} = \{a_i, s_i, c_i\}$, where, the event a_i represents the arrival of τ_i, s_i represents the start of execution of τ_i and c_i represents the completion of execution of τ_i. All events are categorized as *prospective* since they are assigned with finite time bounds. We have used the shorthand notation t to represent the *tick* event.

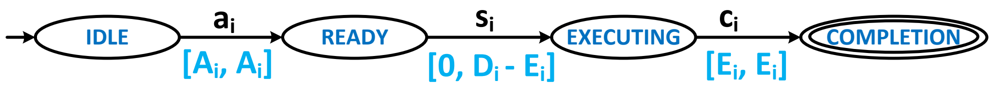


Figure 2.2: *ATG for a single instance of τ_i with parameters A_i, E_i and D_i*

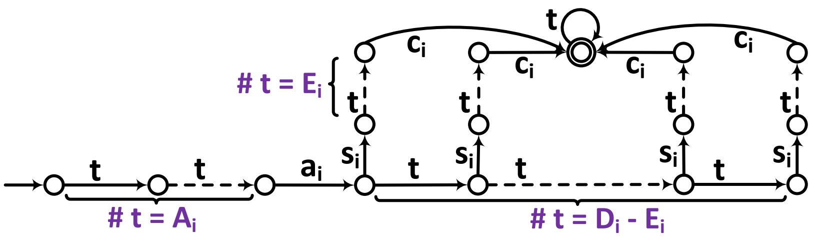


Figure 2.3: *TDES for a single instance of τ_i with parameters A_i, E_i and D_i*

2.2.4 Behavior of TDES

Let us denote by Σ^+ the set of all finite sequences of events (or strings) of Σ, of the form $\sigma_1\sigma_2...\sigma_k$ where $k \geq 1$, $k \in \mathbb{N}$ and $\sigma_k \in \Sigma$. Let $\epsilon \notin \Sigma$ be the empty event and define $\Sigma^* = \{\epsilon\} \cup \Sigma^+$. Function δ can be extended to $\delta : Q \times \Sigma^* \rightarrow Q$. The *closed* behavior of TDES G is the language $L(G) = \{s \in \Sigma^* | \delta(q_0, s)$ is defined$\}$. The *marked* behavior of TDES G is the language $L_m(G) = \{s \in \Sigma^* | \delta(q_0, s) \in Q_m\}$. The *prefix-closure* of a language $L \subseteq \Sigma^*$ is denoted by $\overline{L}$ and consists of all the prefixes of all strings in L. $\overline{L} = \{s \in \Sigma^* : (\exists x \in \Sigma^*)\ [sx \in L]\}$. L is said to be prefix-closed if $L = \overline{L}$. G is *non-blocking* if $L_m(G)$ satisfies $\overline{L_m(G)} = L(G)$. Likewise, G is *blocking* if $L(G) \neq \overline{L_m(G)}$.

2.2.5 Reachability, Co-reachability, Composition

A state $q \in Q$ is *reachable*, if $\delta(q_0, s) = q$, for some $s \in \Sigma^*$. TDES G is said to be reachable, if q is reachable for all $q \in Q$. We use the notation $Ac(G)$ to denote the operation of deleting all the states of G that are not reachable from q_0. A state $p \in Q$ is *co-reachable*, if $\delta(p, s) \in Q_m$, for some $s \in \Sigma^*$. A TDES G is said to be co-reachable, if p is co-reachable for every $p \in Q$. G is said to be *non-blocking*, if $L(G) = \overline{L_m(G)}$. Otherwise, G is said to be *blocking*.

Suppose we have n languages corresponding to n TDES, $L_i \subseteq \Sigma_i^*$ with $\Sigma = \cup_{i=1}^{n} \Sigma_i$. The *natural projection* $P_i : \Sigma^* \rightarrow \Sigma_i^*$ is defined as: (i) $P_i(\epsilon) = \epsilon$, (ii) $P_i(\sigma) = \epsilon$ if $\sigma \notin \Sigma_i$, (iii) $P_i(\sigma) = \sigma$ if $\sigma \in \Sigma_i$ and (iv) $P_i(s\sigma) = P_i(s)P_i(\sigma), s \in \Sigma^*, \sigma \in \Sigma$. The *inverse projection* of P_i is, $P_i^{-1} : 2^{\Sigma_i^*} \rightarrow 2^{\Sigma^*}$. The *synchronous product* of L_1, L_2, ..., L_n, denoted by $L_1||L_2||\ldots||L_n$, is defined as $P_1^{-1}L_1 \cap P_2^{-1}L_2 \cap \cdots \cap P_n^{-1}L_n$.

2.2.6 Supervisory Control

Formally, a *supervisory control* for G is a map $S : L(G) \rightarrow 2^{\Sigma}$. Given G and S, the resulting closed-loop system, denoted by S/G, is defined inductively according to (i) an empty event $\epsilon \in L(S/G)$ (ii) $[(s \in L(S/G))$ and $(s\sigma \in L(G))$ and $(\sigma \in S(s))] \Leftrightarrow [s\sigma \in L(S/G)]$ (iii) No other strings belong to $L(S/G)$. The marked behavior of S/G is: $L_m(S/G) = L(S/G) \cap L_m(G)$. To summarize, a supervisor of G can be considered as an automaton S that monitors each state of G, and disable certain events in G when necessary, in order to control its behavior [29].

Given the system G and a desired specification $K \subseteq L_m(G)$, $K \neq \emptyset$, the synthesis process first builds a supervisor candidate using $G||K$ and this must be *non-blocking* as well as *controllable* [104]. A TDES is controllable with respect to G if it always admits the occurrence of (i) uncontrollable events eligible in G, (ii) *tick* events, if eligible, and not preempted by an eligible forcible event. If the controllability condition is satisfied, then the resulting controlled system ensures $L_m(S/G) = K$. Otherwise, a largest (i.e., supremal) controllable sublanguage of K can always be found (even though it may be empty). Let $C(K)$ denote the family of controllable sub-languages of K. $C(K)$ is always non-empty, since $\emptyset$ is controllable. $C(K)$ is closed under arbitrary set unions and has a unique largest controllable sub-language sup$C(K)$ such that sup$C(K) \subseteq K$. Therefore, it is possible to design a supervisor which restricts the system behavior to sup$C(K)$. If this supervisor is adjoined with G, then $L_m(S/G) = \text{sup}C(K)$. The supervisor S is said to be *minimally restrictive* in the sense that its only action is to disable certain events when necessary so as to preserve the desired behavior of the system. As a consequence, the solution generates the largest possible subset of legal sequences.

2.2.7 Supervisor Computation

In order to transform $G||K$ (denoted by M) such that it becomes both *controllable* and *non-blocking*, we apply the standard *safe state synthesis* mechanism presented in Algorithm 1 (SAFE STATE SYNTHESIS) [77,104]. The synthesis algorithm iteratively removes the blocking states, together with all predecessor states that are reachable by uncontrollable transitions, and afterwards computes the set of nonblocking states. The algorithm stops either if all remaining states are non-blocking or if the initial state becomes blocking and has been eliminated, in which case a supervisor does not exist. This procedure returns a unique maximal least restrictive supervisor, i.e., sup$C(K)$ [74, 108]. Next, we present the detailed discussion of each step involved in Algorithm 1.

Algorithm 1 first initializes the set of safe (un-safe) states Q_s (Q_u) of M to empty. Then, it invokes Algorithm 2 (CO-REACHABLE) to compute the set of co-reachable states in M, denoted by Q'. Algorithm 2 first initializes SQ_i to the set of marked states Q_m and do not include any state in Q_u, i.e., $Q_m \setminus Q_u$. Then, SQ_i is iteratively extended by adding all states that can reach the co-reachable states using `PreImage`[1].

Next, Algorithm 1 invokes Algorithm 3 (NON-CO-REACHABLE) which first computes the initial set of non-co-reachable states in M through $Q \setminus Q'$, denoted by SQ_i. Then, SQ_i is iteratively extended by adding: (i) Q^{uc}: The set of states in M that can reach the non-co-reachable states in SQ_i through *only uncontrollable events* using `PreImage_UC`[2]. (ii) Q^t: The set of states in M that contain outgoing transition on t to a

[1]The operator `PreImage(`V`, `δ`)` computes the set of states that, by one transition, can reach a state in V ($\subseteq Q$), formally defined as: `PreImage(`V`, `δ`)` $= \{q \in Q | \exists q' \in V : \delta(q, \sigma) = q', \sigma \in \Sigma\}$.

[2]`PreImage_UC`$(V, \delta) = \{q \in Q | \exists q' \in V : \delta(q, \sigma) = q', \sigma \in \Sigma_{uc}\}$.

state in SQ_i (using `PreImage_t`[1]). In addition, these states should not contain outgoing transition on any forcible event (obtained using `Undef_for`[2]). The extended set of non-co-reachable states is denoted by Q''. In Algorithm 1, Q'' is added to the set of un-safe states Q_u^i and is iteratively extended until fix-point is reached.

ALGORITHM 1: SAFE STATE SYNTHESIS

```
Input: $M(= Q, \Sigma, \delta, q_0, Q_m)$
Output: The set of reachable safe states $Q_s$ of $M$
Set of safe states, $Q_s \leftarrow \emptyset$;
Set of un-safe states, $Q_u \leftarrow \emptyset$;
$i \leftarrow 0$; $Q_u^i \leftarrow Q_u$;
repeat
    $i \leftarrow i + 1$;
    $Q' \leftarrow$ CO-REACHABLE $(M, Q_u^{i-1})$;
    $Q'' \leftarrow$ NON-CO-REACHABLE $(M, Q')$;
    $Q_u^i \leftarrow Q_u^{i-1} \cup Q''$;
until $Q_u^i = Q_u^{i-1}$;
$Q_u \leftarrow Q_u^i$;
$i \leftarrow 0$; $SQ_i \leftarrow q_0$; {FORWARD-REACHABILITY}
repeat
    $i \leftarrow i + 1$;
    $SQ_i \leftarrow (SQ_{i-1} \cup$ Image$(SQ_{i-1}, \delta)) \setminus Q_u$;
until $SQ_i = SQ_{i-1}$;
return $Q_s = SQ_i$ ;
```

ALGORITHM 2: CO-REACHABLE

```
Input: $M$, $Q_u$
Output: The set of co-reachable states of $M$
$i \leftarrow 0$; $SQ_i \leftarrow Q_m \setminus Q_u$;
repeat
    $i \leftarrow i + 1$;
    $SQ_i \leftarrow (SQ_{i-1} \cup$ PreImage$(SQ_{i-1}, \delta)) \setminus Q_u$;
until $SQ_i = SQ_{i-1}$;
return $SQ_i$;
```

Finally, Algorithm 1 performs the reachability computation to obtain the set of safe states (denoted by Q_s) that only consists of reachable states and does not contain any of the un-safe states (Q_u). It first initializes SQ_i to the initial state q_0 of M and then, SQ_i

[1]`PreImage_t`$(V, \delta) = \{q \in Q | \exists q' \in V : \delta(q, t) = q'\}$.
[2]`Undef_for`$(V, \delta) = \{q \in V | \forall \sigma \in \Sigma_{for} : \delta(q, \sigma)$is undefined$\}$.

ALGORITHM 3: NON-CO-REACHABLE

```
   Input: $M$, The set of co-reachable states $Q'$
   Output: The set of non-co-reachable states
1  $i \leftarrow 0$; $SQ_i \leftarrow Q \setminus Q'$;
2  repeat
3      $i \leftarrow i + 1$;
4      $Q^{uc} \leftarrow \texttt{PreImage\_UC}(SQ_{i-1}, \delta)$;
5      $Q^{t} \leftarrow \texttt{Undef\_for}(\texttt{PreImage\_t}(SQ_{i-1}, \delta), \delta)$;
6      $SQ_i \leftarrow (SQ_{i-1} \cup Q^{uc} \cup Q^{t})$;
7  until $SQ_i = SQ_{i-1}$;
8  return $SQ_i$;
```

is iteratively extended by adding all states that can be reached using `Image`[1]. During such expansion, it may reach un-safe states also. However, such states (Q_u) are discarded at each iteration. Finally, Algorithm 1 results in Q_s containing all possible safe which guarantee the timely execution of tasks.

The set of safe states Q_s resulting from Algorithm 1 contains all possible safe execution sequences which guarantee the timely execution of tasks. It may be noted that the above synthesis procedure generates the exhaustive set of all feasible scheduling sequences. The supervisor (i.e., scheduler) can make use of anyone of these sequences depending on the specific system requirements in a given scenario and use it for the scheduling of tasks on-line.

Complexity Analysis: It can be observed that the number of fix-point iterations corresponding to Algorithms 2 (CO-REACHABLE) and 3 (NON-CO-REACHABLE) is bounded by the number of states in the composite model M, and each such iteration has a complexity that is linear in the size of M. It follows that the complexity of Algorithm 1 (SAFE STATE SYNTHESIS) is polynomial in the size of M.

2.2.8 Tools

A software package called TTCT [2] supports the creation of a new ATG (using the procedure `ACreate`), construction of TDES from ATG (using `Timed_Graph`), direct construction of a new TDES (using `Create`), combining two or more TDES (using `Sync`) and

[1]The operator `Image(`V`, `δ`)` computes the set of states that can be reached by executing one transition: `Image(`V`, `δ`)` $= \{q' \in Q | \exists q \in V : \delta(q, \sigma) = q', \sigma \in \Sigma\}$.

computing the supremal controllable sublanguage of a given language (using `Supcon`). For a given TDES representing G and K, `supcon` internally first computes `Sync`(G, K) and then, applies fixpoint iteration algorithm developed in [108] to compute the supremal controllable sublanguage of $G||K$. In the TTCT tool, this can be computed using `Supcon` procedure. We now explain the steps that can be used to obtain a supervisor / scheduler for a given system using the TTCT tool.

An Illustrative Example: Let us consider a uniprocessor system consisting of two aperiodic tasks (with single instance) τ_1 and τ_2. The parameters associated with task τ_1 are as follows: τ_1 arrival time $A_1 = 0$, execution time $E_1 = 2$, deadline $D_1 = 3$. Similarly, for task τ_2: $A_2 = 1$, $E_2 = 1$, $D_2 = 2$. We follow the ATG model presented in the example of Section 2.2.3 to construct ATGs EX_AT1 and EX_AT2 for task τ_1 and τ_2, respectively. As mentioned earlier, we use the procedure `ACreate` to construct the ATG model EX_AT1, as follows:

EX_AT1 = ACreate(EX_AT1,[mark 3],[time bounds [1,0,1],[2,0,0],[4,2,2]],[forcible 1],[tran [0,2,1],[1,1,2],[2,4,3]]) (4,3)

In the above command, EX_AT1 represents the name of the ATG model. [mark 3] represents that the activity 3 is marked. [time bounds [1,0,1],[2,0,0],[4,2,2]] represents the time bounds associated with the events representing the arrival (a_1 is mapped to 2), start of execution (s_1 is mapped to 1), and completion (c_1 is mapped to 4). For example, the time bounds associated with the start of execution event s_1 (mapped to 1) is represented by [1,0,1]. [forcible 1] represents the list of events that are considered to be forcible. [tran [0,2,1],[1,1,2],[2,4,3]] represents the transition structure of ATG. Finally, the tuple (4,3) represents the total number of activities and transitions present in the ATG, which is generated automatically by TTCT. The ATG model EX_AT1 constructed using `ACreate` is shown in Figure 2.4.

For a given ATG model, we can obtain its corresponding TDES version using the procedure `Timed_Graph`. Now, we apply EX_AT1 to construct its corresponding TDES version as follows:

EX_AT1_TIMED = Timed_Graph(EX_AT1) (7,8)

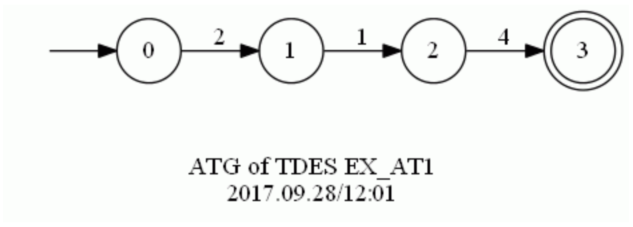


Figure 2.4: *ATG model EX_AT1 for task τ_1*

In the above command, the tuple (7,8) represents the total number of states and the transitions present in the TDES model obtained from its ATG representation. This tuple is generated automatically by the tool TTCT. The TDES version of EX_AT1, i.e., EX_AT1_TIMED, is shown in Figure 2.5.

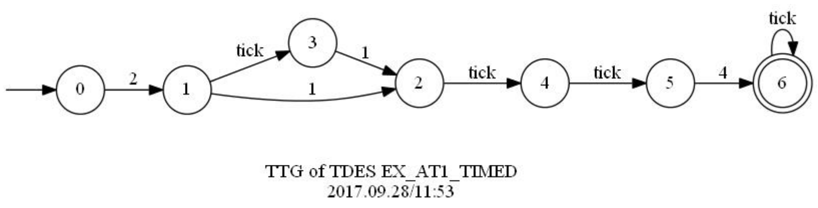


Figure 2.5: *TDES model EX_AT1_TIMED for task τ_1*

Similar to the construction of models related to task τ_1, we can construct the ATG and TDES models corresponding to the task τ_2. The ATG model EX_AT2 is shown in Figure 2.6 and its corresponding TDES model EX_AT2_TIMED is shown in Figure 2.7.

EX_AT2 = ACreate(EX_AT2,[mark 3],[time bounds [3,0,1],[6,1,1],[8,1,1]],[forcible 3],[tran [0,6,1],[1,3,2],[2,8,3]]) (4,3)

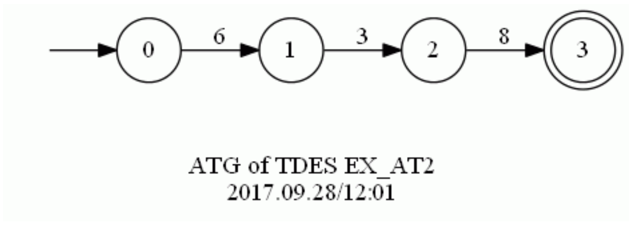


Figure 2.6: *ATG model EX_AT2 for task τ_2*

EX_AT2_TIMED = Timed_Graph(EX_AT2) (7,8)

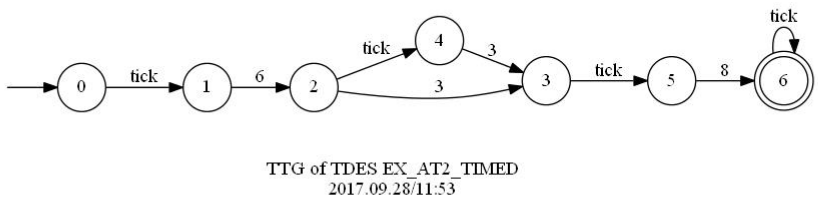


Figure 2.7: *TDES model EX_AT2_TIMED for task τ_2*

Given two individual TDES models EX_AT1_TIMED and EX_AT2_TIMED, we can obtain their *synchronous product* using the procedure `Sync`, as follows:

EX_AT12_TIMED = Sync(EX_AT1_TIMED,EX_AT2_TIMED) (23,30)

In the above command, EX_AT12_TIMED represents the name of the resulting composite model. The tuple (23,30) represents the total number of states and transitions present in the composite model. The transition structure of the composite model EX_AT12_TIMED is shown in Figure 2.8.

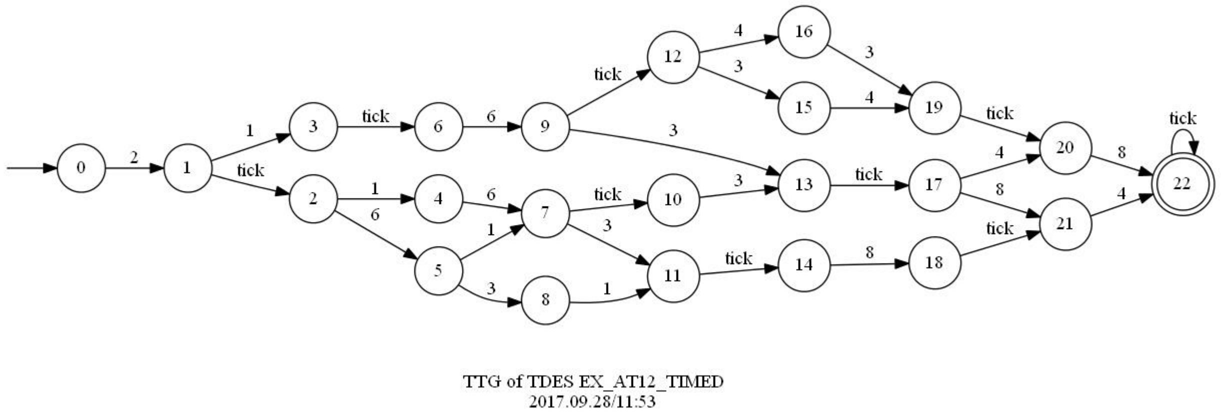


Figure 2.8: *TDES model EX_AT12_TIMED representing the synchronous product of EX_AT1_TIMED and EX_AT2_TIMED*

Next, we create a specificaiton model to enfore the resource constraint (i.e., only one task is allowed to execute on the processor at any instant of time). The TDES model EX_RC is constructed using the command given below and its transition structure is shown in Figure 2.9.

EX_RC = Create(EX_RC,[mark 0],[tran [0,0,0],[0,1,1],[0,2,0],[0,3,2],[0,6,0], [1,0,1],[1,4,0],[1,6,1],[2,0,2],[2,2,2],[2,8,0]],[forcible 1,3]) (3,11)

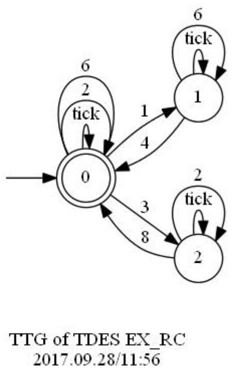


Figure 2.9: *TDES model EX_RC representing the resource constraint*

Given a system model EX_AT12_TIMED and a specification EX_RC, we can obtain their corresponding supremal controllable sublanguage using the following command:

EX_SUPCON = Supcon(EX_AT12_TIMED,EX_RC) (10,10)

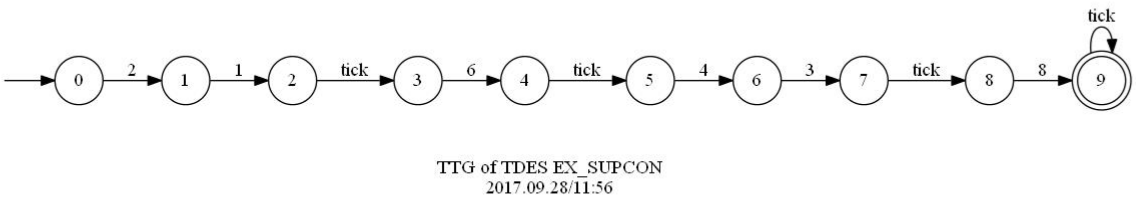


Figure 2.10: *A supervisor / scheduler for a given uniprocessor system*

Figure 2.10 shows the supremal controllable sublanguage, which can be used as a supervisor / scheduler to control the execution of tasks in a given uniprocessor system with two aperiodic tasks τ_1 and τ_2.

2.3 Summary

This chapter started with a brief overview of the evolution of real-time systems followed by scheduling algorithms related uniprocessors to homogeneous/heterogeneous multiprocessors. Then, we have presented the fundamental definitions related to supervisory control of timed discrete event systems. These concepts and definitions will be either referred or reproduced appropriately later in this thesis, to enhance the readability. In the next chapter, we present the two scheduler synthesis schemes for a set of independent real-time tasks executing (non-preemptively/preemptively) on a uniprocessor platform.

Chapter 3

Scheduling of Sporadic Tasks on Uniprocessors

In the last chapter, we discussed various scheduling algorithms related to real-time systems implemented on uniprocessor and multiprocessor systems. Also, the scheduler synthesis using *Supervisory Control of Timed Discrete Event Systems* (SCTDES) was presented. As mentioned earlier, this dissertation is oriented towards the design of SCTDES based optimal scheduling strategies for safety-critical real-time systems implemented on uni/multi-processor platforms. With this objective, we present two scheduler synthesis frameworks for the scheduling of sporadic tasks executing (non-preemptively/preemptively) on uniprocessor platforms, and they are listed as follows:

- **Non-preemptive Scheduler Synthesis Framework**: First, we develop models which can be used to synthesize a *non-preemptive* scheduler for a set of aperiodic tasks with known arrival times, executing on a uniprocessor platform. Then, we relax the restrction on task arrival times, and extend our initial models appropriately to synthesize schedulers for dynamically arriving sporadic tasks. We illustrate the practical applicability of our framework using a motor network example. Although, in recent years, there has been a few significant works dealing with real-time scheduling using SCTDES, this is possibly the first work which addresses the scheduler synthesis problem for *sporadic* tasks.

- **Preemptive Scheduler Synthesis Framework**: In this work, we develop a *preemptive* scheduler synthesis framework for the optimal scheduling of dynamically arriving, sporadic tasks. In case of preemptive execution, the scheduler has the ability to interrupt the execution of a task at each time instant. Due to this, there is a possibility of delaying the execution of tasks by keeping the processor idle in the presence of ready to execute tasks, as compared to its non-preemptive counter-part. Hence, we synthesize a work-conserving scheduler which avoid processor idling in the presence of ready to execute tasks. The synthesized scheduler is able to support concurrent execution of multiple accepted tasks and correctly model the inter-arrival time requirement of a sporadic task. Finally, applicability of the proposed scheme on realistic scenarios has been exhibited by presenting a case study on an *Instrument Control System*.

3.1 Related Works

In recent years, researchers have shown that off-line formal approaches such as SCTDES [18] can be used to synthesize the optimal, *correct-by-construction*, schedulers for real-time systems [29, 53, 85, 87, 105, 106]. Chen and Wonham presented the scheduler design for *non-preemptive, periodic tasks* executing on uniprocessors [29]. Later, the SCTDES framework has been extended by Janarthanan et al. for the scheduling of both *non-preemtive and preemptive periodic tasks* [53]. Recently, Wang et al. enhanced the models presented in [29, 53] to schedule *non-preemptive, periodic tasks with multiple periods* [105]. The notion of multiple periods allows the tasks to dynamically reconfigure their periods at run-time. They have also proposed an approach for *conditionally-preemptive, real-time scheduling of periodic tasks* [106]. Park and Cho developed the *preemptive scheduler for dynamically arriving sporadic tasks* on uniprocessor systems in [85]. Later, Park and Yang presented the preemptive scheduling scheme to allow the *co-execution of periodic and sporadic tasks* on uniprocessors with mutually exclusive accessibility of shared resources [87]. However, the scheduler synthesized in [85,87] does not model *minimum inter-arrival times*, which is necessary for the precise characterization of sporadic tasks.

Table 3.1 provides the summary of qualitative comparison between the state-of-the-art SCTDES based uniprocessor scheduling schemes. Although the scheduler synthesis approaches presented in [85, 87] attempt to handle sporadic tasks, they fail to correctly model minimum inter-arrival time constraint. In this chapter, we present scheduler synthesis framework which has the ability to correctly model sporadic tasks.

Table 3.1: *Comparison between SCTDES based uniprocessor scheduling schemes*

Method	**Tasks**	**Preemptive / Non-preemptive**	**Remarks**
[29]	Periodic	Non-preemptive	It considers non-preemptive periodic tasks.
[53]	Periodic	Both	It considers both preemptive & non-preemptive tasks.
[105]	Periodic	Preemptive	It considers tasks with multiple periods.
[106]	Periodic	Conditionally preemptive	It considers conditionally preemptive tasks.
[85]	Sporadic	Preemptive	It does not capture minimum inter-arrival time constraint of sporadic tasks.
[87]	Periodic, Sporadic	Preemptive	It does not correctly model periodic as well as sporadic tasks.
Section 3.2	**Aperiodic, Sporadic**	Non-preemptive	It correctly captures minimum inter-arrival time constraint of sporadic tasks.
Section 3.3	**Sporadic**	Preemptive	It correctly captures minimum inter-arrival time constraint of sporadic tasks.

3.2 Non-preemptive Scheduler Synthesis Framework

In this section, we present the synthesis framework for the design of an optimal non-preemptive scheduler for a set of: (i) aperiodic tasks with known arrival times, (ii) dynamically arriving aperiodic tasks, and (iii) dynamically arriving sporadic tasks.

3.2.1 Aperiodic Tasks with Known Arrival Times

System Model: We consider a real-time system consisting of a set I (= $\{\tau_1$, τ_2, ..., $\tau_n\}$) of n (≥ 1) non-preemptive, aperiodic tasks to be scheduled on a uniprocessor [31, 47, 54, 54]. Formally, the task τ_i to be scheduled is represented by a 3-tuple $\langle A_i, E_i, D_i \rangle$, where A_i is the *arrival time*, E_i is the *execution time* and D_i is the *relative deadline* of τ_i.

Assumptions: (1) Tasks are independent with no precedence constraints [30]. (2) Execution time for each task is constant and equal to its worst-case execution time WCET [73]. (3) $E_i \leq D_i$ for all tasks in I. (4) All tasks are reported to the scheduler which then controls their execution on the uniprocessor according to the pre-computed off-line scheduling policy.

3.2.2 Task execution model

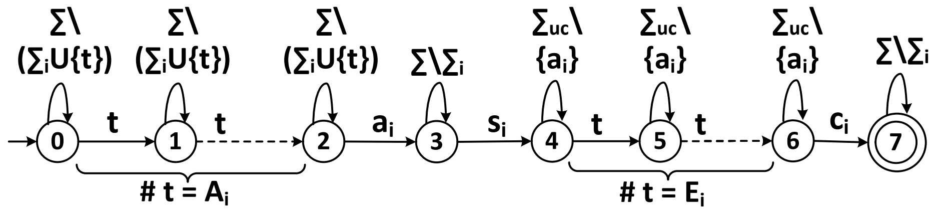


Figure 3.1: *Task execution model T_i for task τ_i $\langle A_i, E_i, D_i\rangle$.*

The TDES model T_i for executing a non-preemptive, aperiodic task τ_i $\langle A_i, E_i, D_i\rangle$ is (shown in Figure 3.1) defined as,

$$T_i = (Q_i,\ \Sigma,\ \delta_i,\ \ q_0,\ Q_m),$$

where, $Q_i = \{0, 1, ..., 7\}$[1], $q_0 = 0$, $Q_m = \{7\}$, $\Sigma = \cup_{i\in\{1,2,...,n\}}\Sigma_i \cup \{t\}$, where $\Sigma_i = \{a_i, s_i, c_i\}$. The events are described in Table 3.2 and they are categorized as follows: (i) $\Sigma_{uc} = \cup_{i\in\{1,2,...,n\}}\{a_i\}$, (ii) $\Sigma_c = \Sigma\setminus(\Sigma_{uc}\cup\{t\})$, (iii) $\Sigma_{for} = \Sigma_c$. Since the arrival events of tasks are induced by the environment, they are modeled as *uncontrollable* events. Apart from Σ_{uc} and *tick* event (t), remaining events in the system are modeled as *controllable* events. These controllable events are also modeled as *forcible* events which can preempt the *tick* event (t). All events in Σ are considered to be *observable*.

Table 3.2: *Description of events (for non-preemptive execution)*

Event	Description
a_i	Arrival of a task τ_i
s_i	Start of execution of τ_i
c_i	Completion of execution of τ_i

State 0 is the initial state of T_i and represents the state in which task τ_i resides prior to the start of the system. Events in the self-loop $\Sigma \setminus (\Sigma_i \cup \{t\})$ may be characterized

[1]The states are numbered sequentially starting from 0, for illustration purpose. However, the total number of states in T_i for a given task τ_i will vary depending on its execution time.

as follows: Since τ_i has not yet arrived, the events that are associated with τ_i (Σ_i) are excluded from Σ. To elaborate, the self-loop contains the events such as arrival, start of execution, and completion with respect to any other task ($\tau_j \in I, j \neq i$) in the system. Thus, $\Sigma \setminus (\Sigma_i \cup \{t\})$ does not impose any restriction on the execution of other tasks. *State 0* is replicated A_i times in order to measure the occurrence of A_i ticks subsequent to system start.

After the occurrence of a_i at ${A_i}^{th}$ tick, T_i reaches *State 3* from *State 2*. Here, the scheduler takes a decision whether to immediately allocate the task τ_i for execution on a processor or make it wait on the ready queue. The latter is indicated by the self-loop transition $\Sigma \setminus \Sigma_i$ at *State 3*. If the scheduler decides to start the execution of τ_i, then it will enable the event s_i to assign τ_i for execution on the processor. According to Figure 3.1, if event s_i occurs at *State 3*, then T_i reaches *State 4*. At *State 4*, the self-loop transition $\Sigma_{uc} \setminus \{a_i\}$ ensures that only arrivals (barring that of τ_i) but not the execution of tasks other than τ_i are allowed. This is because τ_i has already started its non-preemptive execution on the processor. After the elapse of E_i ticks from *State 4*, T_i reaches *State 6*. Finally, the event c_i which is used to mark the completion of τ_i's execution, takes T_i to the marked state *State 7*. Now, T_i continues to stay at *State 7* without imposing any constraint on other tasks currently executing on the processor.

It may be noted that τ_i consumes exactly E_i ticks from the start of its execution at *State 4*. However, τ_i may miss its deadline based on the amount of time spent in the ready-queue at *State 3*. Suppose τ_i stays at *State 3* for x ticks before s_i, then (i) τ_i is *deadline-meeting* if $(x + E_i) \leq D_i$, (ii) τ_i is *deadline-missing* if $(x + E_i) > D_i$. Hence, *$L_m(T_i)$ contains both deadline-meeting as well as deadline-missing execution sequences for task τ_i*. Now, let us formally introduce the notion of a deadline-meeting sequence as it is relevant for our discussion.

Definition: *Deadline-meeting sequence (with respect to non-preemptive execution):* A sequence $s = s_1 a_i s_2 c_i s_3 \in L_m(T_i)$, where $s_1, s_3 \in \Sigma^*$, $s_2 \in (\Sigma \setminus \{c_i\})^*$ is *deadline-meeting*, if *tickcount*$(s_2) \leq D_i$. Otherwise, s is a deadline-missing sequence. □

Example: Now, we illustrate the generalized task execution model discussed above using an example. Let us consider an example (adopted from [23]) uniprocessor system consisting of two tasks $\tau_1\langle 0,4,7\rangle$, $\tau_2\langle 1,2,4\rangle$. The task execution models T_1 for τ_1, T_2 for τ_2 are shown in Figures 3.2(a) and 3.2(b), respectively. It can be observed that the arrival of τ_1 (i.e., a_1) takes place at system start. Once τ_1 has started execution (i.e., s_1), it consumes exactly 4 ticks to complete its execution (i.e., c_1). During τ_1's execution, only the arrival of τ_2 is allowed until its completion which is modeled by the self-loops a_2 at states 2 through 6. Similarly, it can be observed from Figure 3.2(b) that the arrival of τ_2 takes place at the first tick subsequent to system start and it takes 2 ticks to complete its execution after the start of its execution. □

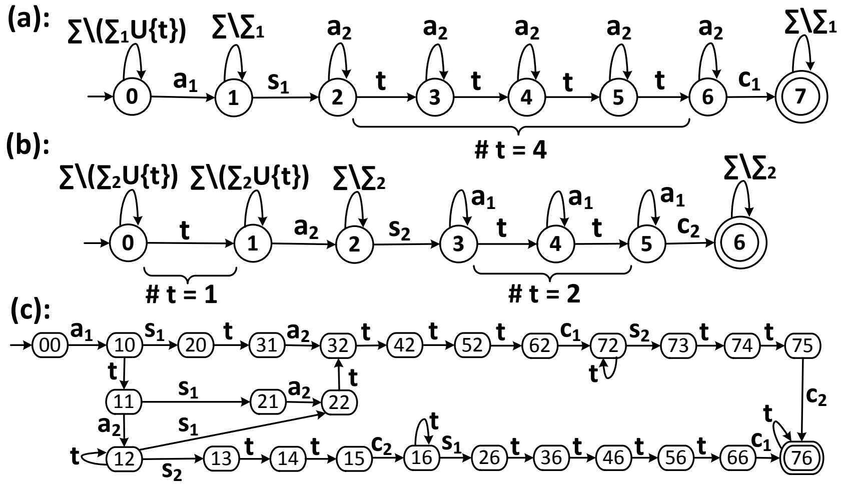


Figure 3.2: *TDES Models: (a) T_1 for τ_1, (b) T_2 for τ_2, (c) $T = T_1||T_2$*

Based on the above approach, we construct the TDES models for all the n aperiodic tasks in the system. Given these n individual TDES task models T_1, T_2, ..., T_n, a product composition $T = T_1||T_2||...||T_n$ on the models gives us the composite model for all the tasks executing concurrently.

Remark 3.2.1. (i) The marked behavior $L_m(T)$ *represented by the composite task execution model includes both deadline-meeting as well as deadline-missing execution sequences for all tasks in I.* (ii) It may be observed that T_i allows only one task for execution on the processor at any instant (which satisfies resource constraint). Specifically, when τ_i has started its execution (i.e., s_i) at *State 3*, only the arrival of other tasks

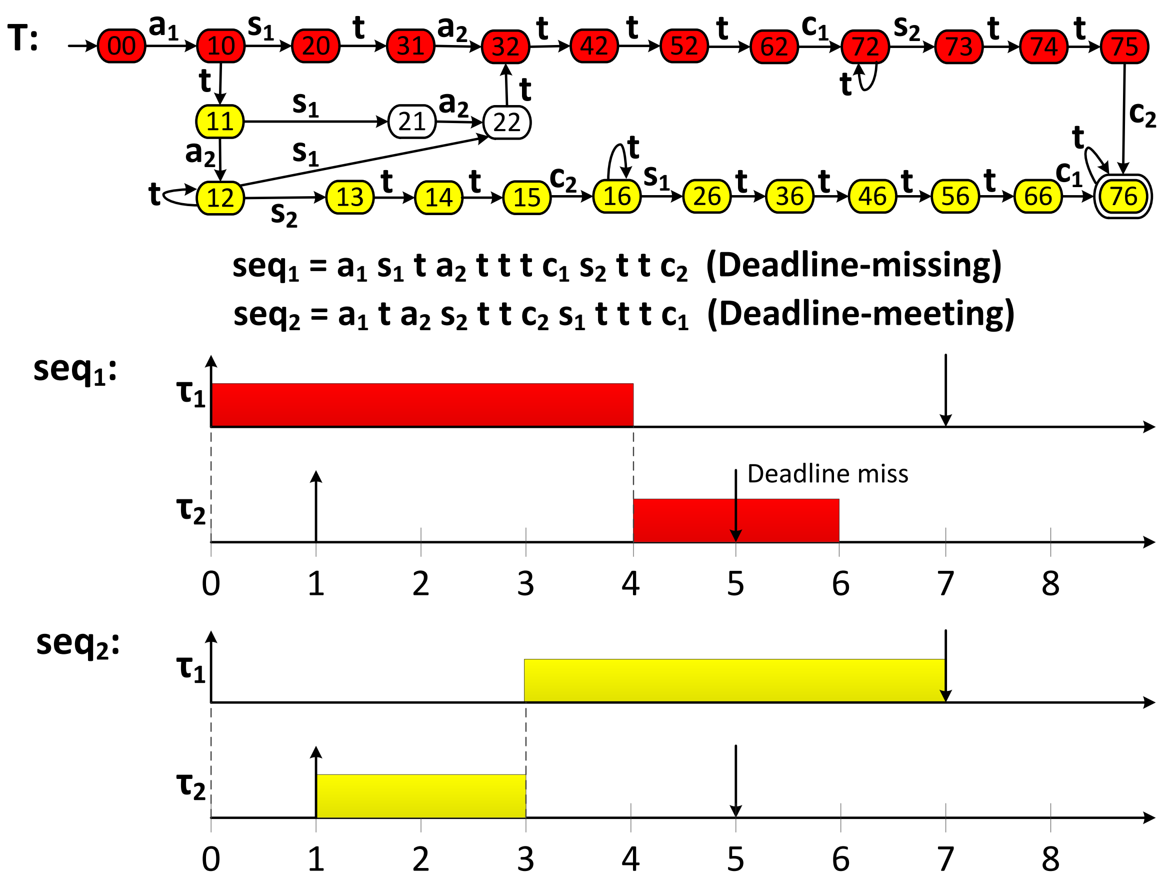

T:
seq₁ = a₁ s₁ t a₂ t t t c₁ s₂ t t c₂ (Deadline-missing)
seq₂ = a₁ t a₂ s₂ t t c₂ s₁ t t t c₁ (Deadline-meeting)
seq₁:
seq₂:
τ₁
τ₂
Deadline miss
t
0 1 2 3 4 5 6 7 8

Figure 3.3: *Gantt chart representation of* seq_1 *and* seq_2 *in* T

$(\Sigma_{uc} \setminus \{a_i\})$ in the system (and not their execution) are allowed until the completion of τ_i. Since $L_m(T) = \cap_{i=1}^{n} L_m(T_i)$, the composite model T also does not contain any sequence that violates resource constraint. □

Example (continued): Figure 3.2(c) shows the composite model T $(= T_1||T_2)$. Here, State XY ($X = 0$ to 7, $Y = 0$ to 6) represents state X of T_1 and state Y of T_2. Let us consider the sequence $seq_1 = a_1 s_1 t a_2 t t t c_1 s_2 t t c_2$ from the initial state of T. In this sequence, τ_1 arrives and starts execution before τ_2. Once τ_1 completes its execution, then τ_2 starts and completes its execution. Here, τ_1 meets its deadline, i.e., $tickcount(a_1 s_1 t a_2 t t t c_1) = 4 \leq 7$. However, τ_2 misses its deadline since $tickcount(a_2 t t t c_1 s_2 t t c_2) = 5$ which is greater than its deadline 4. Hence, seq_1 is a *deadline-missing* sequence. Now, let us consider another sequence $seq_2 = a_1 t a_2 s_2 t t c_2 s_1 t t t t c_1$. Here, both τ_1 $(tickcount(a_1 t a_2 s_2 t t c_2 s_1 t t t t c_1) =$

$7 \leq 7$) and τ_2 ($tickcount(a_2 s_2 t t c_2) = 2 \leq 4$) meet their deadlines. Hence, seq_2 is a *deadline-meeting* sequence. The gantt chart representation of the schedule is shown in Figure 3.3.

3.2.3 Deadline Specification for an Aperiodic Task

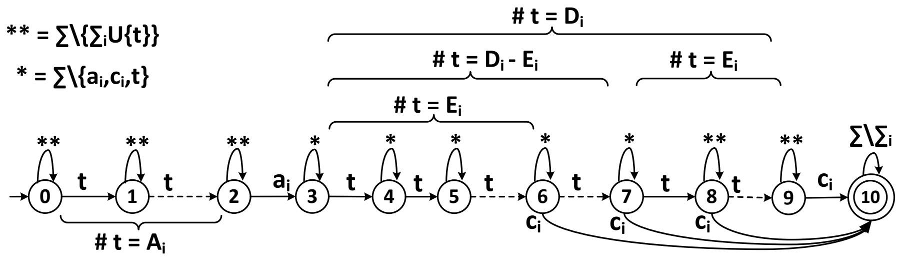


Figure 3.4: *Specification model H_i for an aperiodic task τ_i*

The TDES model H_i shown in Figure 3.4 captures the deadline specification of an aperiodic task τ_i. At the ${A_i}^{th}$ tick subsequent to system start (*State 0*), the arrival of τ_i is allowed at State 2 (i.e., a_i). After the occurrence of a_i, H_i reaches *State 3.* The self-loop transition $\Sigma \setminus \{a_i, c_i, t\}$ in *State 3* is used to model the fact that task τ_i is allowed to only execute event s_i associated with it, however, without imposing any restriction with respect to other tasks in the system. State 3 is replicated $D_i - E_i$ times (from states 3 to 7). Since, τ_i may complete its execution at any time after E_i ticks from its arrival, outgoing transitions on c_i are defined from states 6, 7, 8 to *State 10.* It may be noted that if τ_i does not start its execution at least $D_i - E_i$ ticks before its deadline, then τ_i is guaranteed to miss its deadline. Hence, s_i is disallowed in the self-loops $\Sigma \setminus (\Sigma_i \cup \{t\})$ at states 8 to 9. After the elapse of D_i ticks from a_i, H_i moves to *State 10.* It may be noted that $L_m(H_i)$ *contains only and all deadline-meeting sequences for task* τ_i. Based on this approach, H_i is constructed for all other tasks in the system. Then we perform product composition to obtain the composite deadline specification model $H = H_1||H_2||...||H_n$.

Remark 3.2.2. (i) The marked behavior $L_m(H)$ *represented by the composite deadline specification model includes all deadline-meeting execution sequences for all tasks in* I. (ii) It may be observed that H_i does not restrict the simultaneous execution of multiple tasks on the processor. Specifically, the self-loops at any state in H_i do not restrict the execution of multiple tasks and they can be executed multiple times at each state to allow simultaneous execution. So, $L_m(H)$ contains the sequences that may violate resource-constraint. □

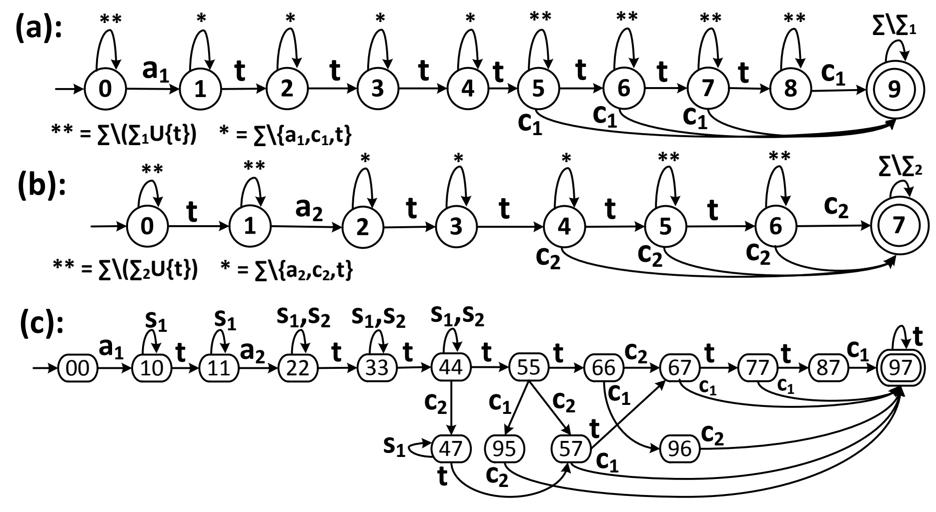

(a):
** = ∑\(∑₁U{t}) * = ∑\{a₁,c₁,t}
(b):
** = ∑\(∑₂U{t}) * = ∑\{a₂,c₂,t}
(c):

Figure 3.5: *TDES Models: (a) H_1 for τ_1, (b) H_2 for τ_2, (c) $H = H_1||H_2$*

Example (continued): The specification models H_1 for τ_1 and H_2 for τ_2 are shown in Figures 3.5(a) and 3.5(b), respectively. Since, τ_1 can start its execution any time within up to 4 ticks subsequent to its arrival, the event s_1 is allowed in the self-loops at states 1 through 4. However, s_1 is disallowed from State 5 since τ_1's execution time ($E_1 = 4$) is greater than the remaining time available to complete its execution at these states. Similar discussion holds for H_2 of τ_2. Figure 3.5(c) shows the composite model H ($= H_1||H_2$). As mentioned earlier, this model represents all possible deadline-meeting sequences for τ_1 and τ_2. In order to illustrate this fact, let us try to find the deadline-missing sequence seq_1 in the composite model H shown Figure 3.5(c). After proceeding through the states $00 \xrightarrow{a_1} 10 \xrightarrow{s_1} 10 \xrightarrow{t} 11 \xrightarrow{a_2} 22 \xrightarrow{t} 33 \xrightarrow{t} 44 \xrightarrow{t} 55 \xrightarrow{c_1} 95 \xrightarrow{s_2} 95$, the composite model H *gets blocked* due to the absence of a transition on s_2 at State 95. More specifically, after processing the sub-string $a_1s_1ta_2tttc_1$ of seq_1, the next event in seq_1 is s_2. However, s_2 is not present at State 95. Thus, $L_m(H)$ does not contain the deadline-missing sequence seq_1. However, the deadline-meeting sequence seq_2 can be retrieved by tracing the states $00 \xrightarrow{a_1} 10 \xrightarrow{t} 11 \xrightarrow{a_2} 22 \xrightarrow{s_2} 22 \xrightarrow{t} 33 \xrightarrow{t} 44 \xrightarrow{c_2} 47 \xrightarrow{s_1} 47 \xrightarrow{t}$

$57 \xrightarrow{t} 67 \xrightarrow{t} 77 \xrightarrow{t} 87 \xrightarrow{c_1} 97$.

3.2.4 Supervisor Synthesis

In order to find all deadline-meeting sequences from $L_m(T)$, we construct the finite state automaton $M = T||H$.

Theorem 3.2.1. $L_m(M)$ contains only and all the deadline-meeting sequences of $L_m(T)$.

Proof. We know that $L_m(T)$ contains both deadline-meeting as well as deadline-missing sequences of all tasks in I (Remark 3.2.1(i)). On the other hand, $L_m(H)$ contains all possible deadline-meeting sequences (Remark 3.2.2(i)). Since $L_m(M) = L_m(T) \cap L_m(H)$, $L_m(M)$ contains *only* and *all* the deadline-meeting sequences of $L_m(T)$.

Apart from the removal of deadline-missing sequences in $L_m(T)$, the synchronous product composition also eliminates the sequences in $L_m(H)$ that violate resource constraint. This is because, $L_m(T)$ does not contain any sequence that violates resource constraint (Remark 3.2.1(ii)) and $L_m(M) = L_m(T) \cap L_m(H)$. Hence, we can conclude that $L_m(M)$ contains *only* and *all* the deadline-meeting sequences of $L_m(T)$ that satisfies resource constraint. □

Although $L_m(M)$ is deadline-meeting, the deadline-missing sequences in T lead to *deadlock* states in M, i.e., M is *blocking*. Hence, we apply *safe state synthesis* mechanism presented in Algorithm 1 to remove the minimal number of states from M until it becomes both *controllable* and *non-blocking*.

Remark 3.2.3. It may happen that Q_s is an *empty* set implying that the given task set is *non-schedulable*. In such cases, no other scheduling scheme can find a schedulable execution sequence for the given task set since the scheduler synthesized using our scheme is *optimal*. In order to make the task set *schedulable*, modifications must be carried-out on individual task parameters (E_i or D_i). □

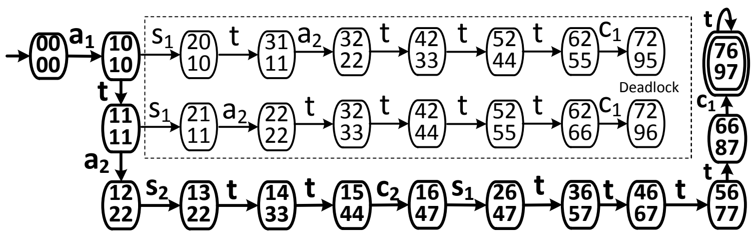


Figure 3.6: $M = T||H$, *Supervisor S (in thick lines)*

Example (continued): Figure 3.6 shows the initial supervisor candidate M. Here, *State WXYZ* represents *State W* of T_1, *State X* of T_2, *State Y* of H_1 and *State Z* of

H_2. For example, State 7697 represents *State 7* of T_1, *State 6* of T_2, *State 9* of H_1 and *State 7* of H_2. Now, let us apply Algorithm 1 to obtain Q_s which is used for constructing the supervisor.

Algorithm 1 invokes Algorithm 2 which initializes $SQ_0 = Q_m = \{7697\}$. Now, `PreImage (`SQ_0`, `δ`)` returns 6687 which makes $SQ_1 = \{7697, 6687\}$. In a similar manner, Algorithm 2 continues to iterate until $i = 14$, where a fix-point is reached. This returns the set of co-reachable states, $Q' = \{7697, 6687, 5677, 4667, 3657, 2647, 1647, 1544, 1433, 1322, 1222, 1111, 1010, 0000\}$.

Next, Algorithm 1 invokes Algorithm 3 which first computes the initial set of non-co-reachable states, $SQ_0 = Q \setminus Q' = \{7295, 7296, 6255, 6266, 5244, 5255, 4233, 4244, 3222, 3233, 3111, 2222, 2010, 2111\}$. It can be seen that there exists a transition on event s_1 from states $\{1010, 1111\} \in Q'$ to $\{2010, 2111\} \in SQ_0$, respectively. However, s_1 is a controllable event whose occurrence can be disabled by the supervisor. Thus, no state in Q' can reach a state in SQ_0 through an uncontrollable event. Hence, Algorithm 3 reaches fix-point at $i = 1$ which implies $SQ_1 = SQ_0$ and $Q'' = SQ_1$.

Finally, Algorithm 1 invokes forward reachability operation which starts with the initial state $SQ_0 = 0000$ of M and adds all reachable states excluding Q''. Now, `Image (`SQ_0`, `δ`)` returns 1010 which makes $SQ_1 = \{0000, 1010\}$. Similarly, Algorithm 1 (Line 11 to 15) continues to iterate until $i = 14$, where fix-point is reached. Finally, it returns the set of safe states $Q_s = \{0000, 1010, 1111, 1222, 1322, 1433, 1544, 1647, 2647, 3657, 4667, 5677, 6687, 7697\}$. Since Q_s is non-empty, the given task set is schedulable. Next, the scheduler S is constructed using Q_s which ensures that both τ_1 and τ_2 meet their deadlines. The scheduler S is shown using thick lines in Figure 3.6.

3.2.5 Handling Arbitrary Aperiodic Task Arrivals

The approach discussed above considers aperiodic tasks with known arrival times. Here, we assume task arrivals to be arbitrary, i.e., dynamic. In order to design a scheduler under this changed scenario, the task execution model T_i presented in Figure 3.1 has been modified. In particular, the set of states ({0, 1, 2}) that are used to measure the arrival time of an aperiodic task τ_i (with known arrival time A_i) have been replaced by

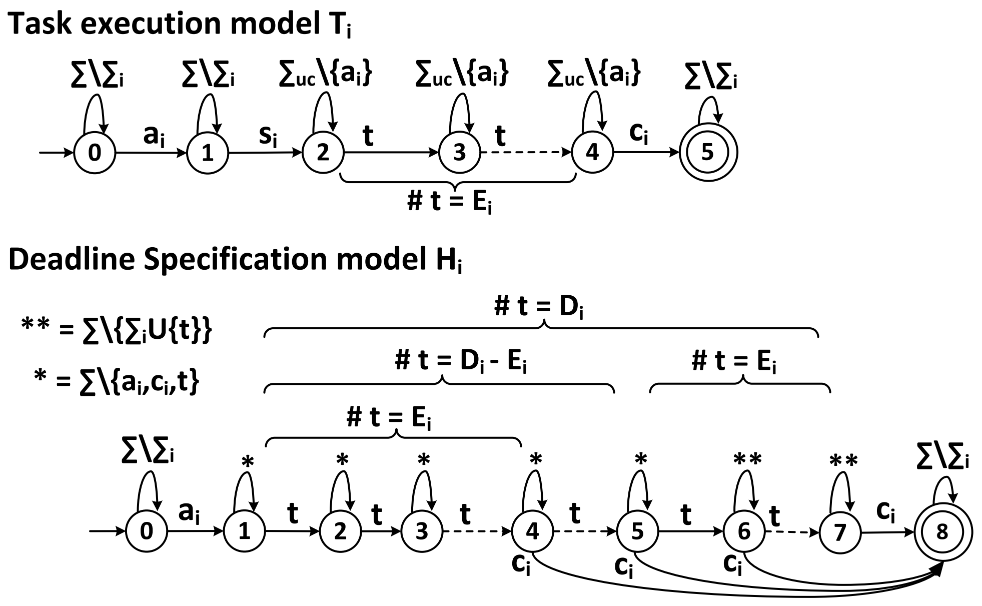


Figure 3.7: *(a) Task execution model, (b) Deadline specification model, for dynamic arrival*

a single state (*State 0*) in T_i shown in Figure 3.7(a). Here, T_i stays at *State 0* until the arrival of τ_i (i.e., a_i) by executing the events in the self-loop $\Sigma \setminus \Sigma_i$. Specifically, $\Sigma \setminus \Sigma_i$ contains t in it which is used to model the arbitrary arrival of τ_i, i.e., a_i (arrival at system start), ta_i (arrival after a tick), tta_i (arrival after two ticks), and so on. The remaining part of T_i (i.e., from arrival a_i to completion c_i) is same as Figure 3.1. Similar changes have been made to the deadline specification model H_i in Figure 3.4. The modified H_i for dynamically arriving aperiodic tasks is shown in Figure 3.7(b).

3.2.6 Handling Dynamically Arriving Sporadic Tasks

Let us consider a dynamically arriving sporadic task τ_i $\langle E_i, D_i, P_i \rangle$, where E_i is the execution time, D_i is the deadline and P_i is the minimum inter-arrival time. Here, we assume $D_i = P_i$, i.e., deadlines are *implicit*. *The sporadic task τ_i consists of an infinite sequence of identical instances that are separated by a minimum inter-arrival time P_i* [23,51]. Before presenting the execution and specification models for the sporadic task τ_i, let us introduce the formal definition of a sporadic sequence.

Definition: *Sporadic Sequence:* "Suppose T_i is a TDES corresponding to a sporadic

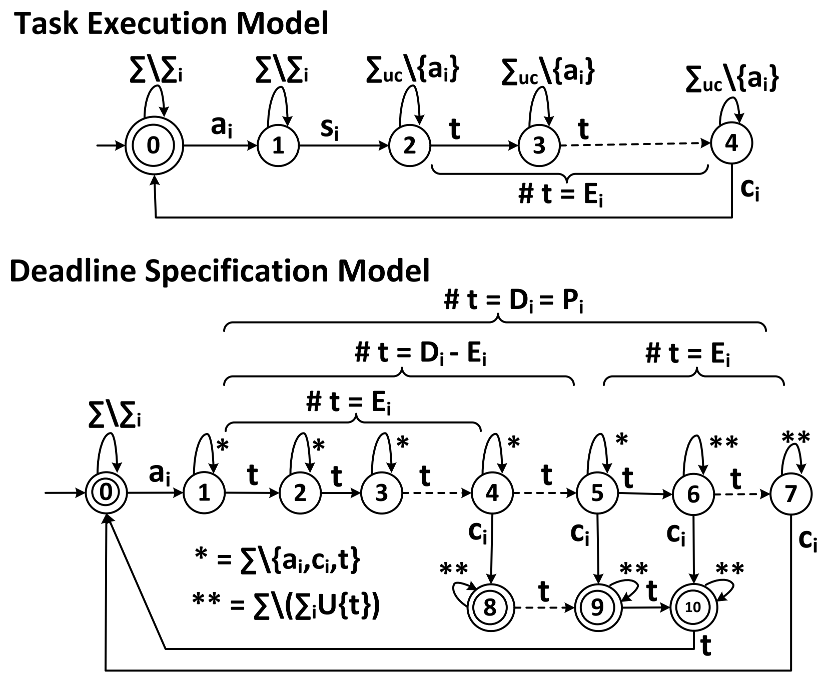


Figure 3.8: *(a) Task execution model, (b) Deadline specification model, for sporadic task*

task τ_i. A sequence $s \in L_m(T_i)$ is a prefix of a *sporadic* sequence if for all $s_1, s_2, ..., s_k \in ((\Sigma_i \cup \{t\}) \setminus \{a_i\})^*$, and $s = s_1 a_i s_2 a_i ... s_k a_i ...$ implies that $\forall k (k > 1)$ $tickcount(s_k) \geq P_i$. Otherwise, the sequence s is *non-sporadic*." Since T_i represents the sporadic task τ_i, this definition must be satisfied $\forall s \in L_m(T_i)$.

Task execution model T_i: In case of aperiodic tasks, we have considered the scheduling of a single instance. However, sporadic tasks consist of an infinite sequence of identical instances. Presently, the model T_i shown in Figure 3.7(a) moves to *State 5* from *State 4* after the completion of a single instance of τ_i and stays their indefinitely. This must be modified to support the execution of sporadic tasks. Figure 3.8(a) shows the modified task execution model T_i corresponding to the dynamically arriving sporadic task τ_i. Here, we have introduced a loop-back to the initial state (*State 0*) from *State 4* on completion event c_i such that the execution of infinite sequences can be supported. In addition to this, we set *State 0* as initial as well as marked state which implies that T_i stays at *State 0* prior to the arrival of next instance of τ_i as well as after the completion of its current instance.

Remark 3.2.4. The marked behavior $L_m(T_i)$ *represented by the task execution model includes (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic and (iv) non-sporadic execution sequences for all tasks in* I. The sporadicity of τ_i is decided by the amount of time spent at the initial state of T_i between any two of its consecutive instances. Suppose τ_i completes the execution of its current instance x ticks subsequent to its arrival and stays at *State 0* for y ticks before its next arrival. Then, τ_i satisfies the minimum inter-arrival time constraint, if $(x+y) \geq P_i$. Otherwise, τ_i is non-sporadic. It can be observed that a deadline-missing sequence may also be sporadic if all instances of each task in the sequence satisfy their minimum inter-arrival time constraints. All these sequences satisfy resource constraint (similar to Remark-3.2.1(ii)). □

The different types of sequences in $L_m(T_i)$ (as specified in Remark 3.2.4) are also carry forwarded to $L_m(T)$. In order to remove the sequences that violate the deadlines and / or sporadicity from $L_m(T)$, we construct the timing specification model which captures the following aspects of a sporadic task, (i) infinite sequence of identical instances, (ii) minimum inter-arrival time and (iii) deadline satisfaction.

Timing specification model H_i: Figure 3.8(b) shows the timing specification of a dynamically arriving, implicit deadline meeting sporadic task τ_i. This model includes the set of states $\{8, 9, ..., 10\}$ in addition to the states that are already present in H_i (shown in Figure 3.7(b)) to capture the minimum inter-arrival time constraint. In order to model the infinite sequence of identical instances, a loop-back on completion event c_i (from states $\{4, 5, ..., 6\}$) to the initial state (*State 0*) of H_i have been added via states $\{8, 9, ..., 10\}$. Now, we explain how these newly added states take care of minimum inter-arrival time. Suppose, τ_i executes c_i, the completion event, $P_i - j$ ticks ($j = \{1, 2, ..., (E_i - 1)\}$) subsequent to its arrival. Then, it will reach anyone (say, *State x*) among the states 8, 9, ..., 10, such that all substrings between *State x* and *State 0* contain j ticks. The self-loop $\Sigma \setminus (\Sigma_i \cup \{t\})$ at these states ($\{8, 9, ..., 10\}$) disallows the arrival of sporadic task τ_i's next instance before the elapse of P_i ticks from the arrival of its current instance. Apart from this, the remaining states ($\{0, 1, ..., 7\}$) are same as H_i (shown in Figure 3.7(b)), and hence $L_m(H_i)$ contains the deadline-meeting sequences.

So $L_m(H_i)$ satisfies the three aspects of a sporadic task mentioned above.

Remark 3.2.5. The marked behavior $L_m(H_i)$ *represented by the timing specification model includes all and only the deadline-meeting as well as sporadic execution sequences for all tasks in* I. However, these sequences may violate resource constraint. □

According to the scheduler synthesis framework, next we compute $M = T||H$. Since $L_m(M) = L_m(T) \cap L_m(H)$ and $L_m(T)$ satisfies resource constraint, $L_m(M)$ *contains only and all the deadline-meeting, sporadic sequences satisfying resource constraint.* Using M as an input to Algorithm 1, we can compute the set of safe states Q_s. From Q_s, the supervisor S will be constructed for the scheduling of real-time tasks at on-line. It may be noted that S contains all feasible sequences considering arbitrary task arrivals on-line.

3.2.7 Application Example

In this section, we illustrate our proposed scheduler synthesis framework using an example. In case of aperiodic tasks, we have already presented a running example in Section 3.2.1. Here, we consider a simple motor network example (adopted from [29]) to illustrate the scheduler synthesis mechanism for a set of dynamically arriving sporadic tasks. Consider two electric motors whose speed stabilization is controlled by a real-time controller implemented on a uniprocessor as illustrated in Figure 3.9(a). The controller consists of two independent sporadic tasks τ_1 and τ_2 to control *Motor 1* and *Motor 2*, respectively. While τ_1 requires 1 ms to stabilize *Motor 1*, τ_2 requires 2 ms for the same corresponding to *Motor 2*. Once stabilized, *Motor 1* remains within its maximum destabilization threshold for at least 6 ms, while *Motor 2* may destabilize after 3 ms subsequent to control action by τ_2. τ_1 and τ_2 are invoked whenever motors 1 and 2 destabilize beyond their threshold limits. Thus τ_1 and τ_2 may be represented as τ_1 $\langle 1, 6, 6 \rangle$ and τ_2 $\langle 2, 3, 3 \rangle$.

The task execution model T_1 for τ_1 and T_2 for τ_2 are shown in Figures 3.9(b) and 3.9(c), respectively. Apart from deadline-meeting and deadline-missing sequences, these task execution models contain both sporadic as well as non-sporadic sequences in it. For example, let us consider the sequences $seq_1 = a_1 s_1 t c_1 ttttt a_1$ and $seq_2 = a_1 s_1 t c_1 a_1$. Two consecutive arrivals of τ_1 are separated by 6 ticks in seq_1 (satisfies sporadicity) and 1

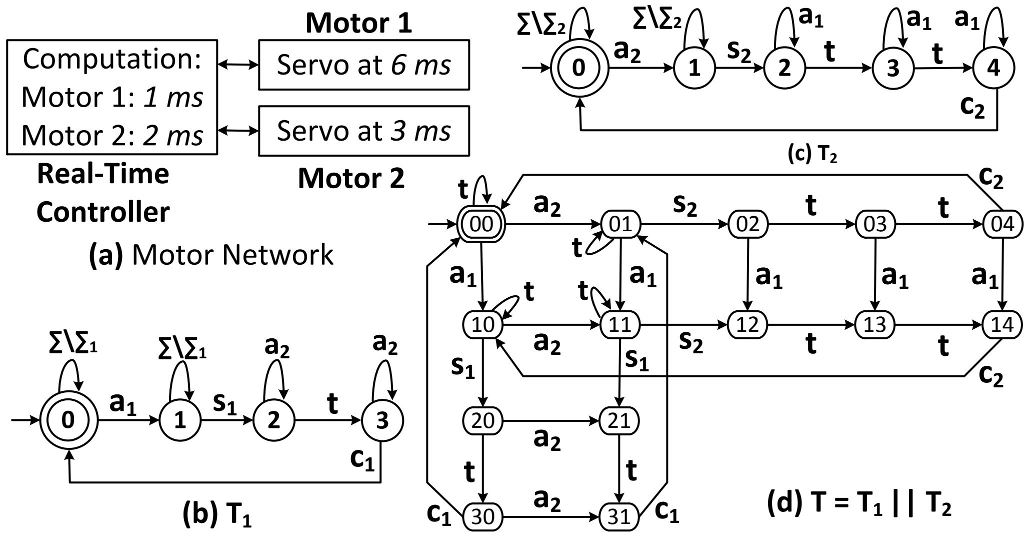


Figure 3.9: *A simple motor network adopted from [29].*

tick in seq_2 (violates sporadicity). The non-sporadic sequences are also carry forwarded to the composite task execution model T $(= T_1||T_2)$ shown in Figure 3.9(d). It may be observed that T allows the dynamic arrivals of τ_1 and τ_2.

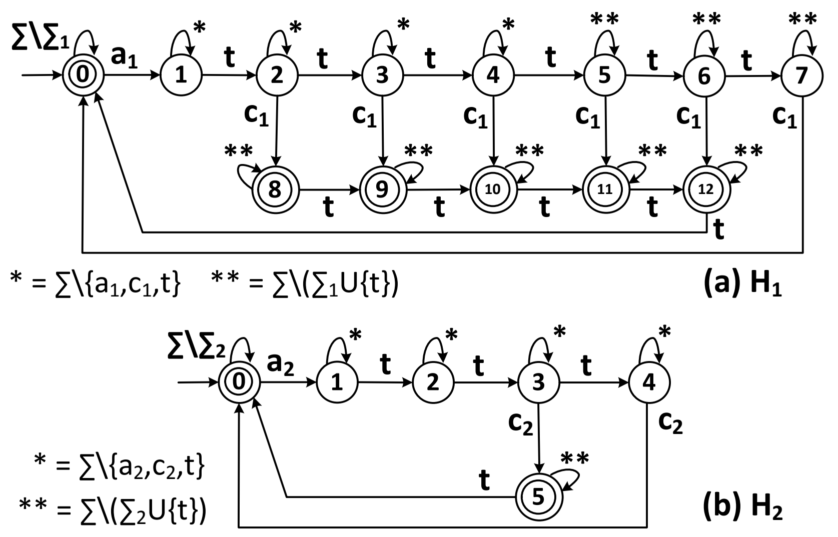


Figure 3.10: *Deadline specification models H_1 for τ_1, H_2 for τ_2.*

The timing specification model H_1 for τ_1 and H_2 for τ_2 are shown in Figures 3.10(a) and 3.10(b), respectively. These models enforce sporadicity as well as deadline specifications. For example, the deadline-meeting sequence seq_2 which violates sporadicity

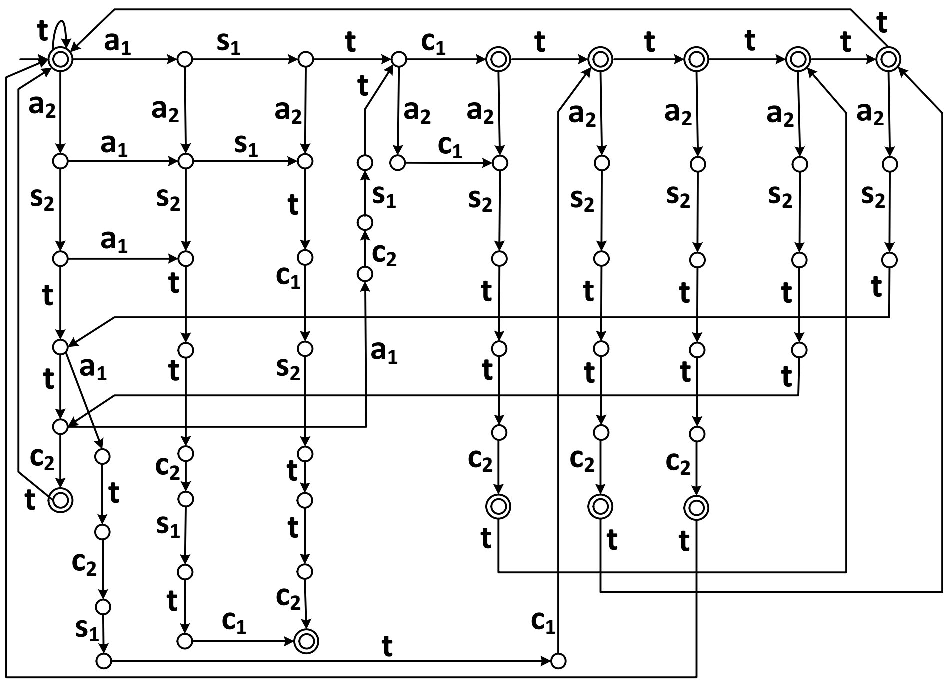


Figure 3.11: *The supervisor S for motor network.*

of τ_1 is not part of H_1. If we attempt to trace seq_2, H_1 gets blocked at *State 8* after processing the sub-string $a_1s_1tc_1$ from the initial state due to the absence of a_1. On the other hand, seq_1 is part of H_1 since it satisfies the minimum inter-arrival time constraint of τ_1.

After obtaining T and H, they are product composed to obtain M whose marked behavior contains the set of sporadic as well as deadline-meeting sequences satisfying resource constraint. Then, we apply Algorithm 1 to obtain the final non-blocking supervisor S (shown in Figure 3.11) which contains all feasible schedules of the dynamically arriving sporadic tasks τ_1 and τ_2.

Now, let us consider a scenario in which the computation and minimum inter-arrival time constraint of τ_1 are 3 and 6, while that for τ_2 are 1 and 2, respectively. It may be observed that in this scenario, M is blocking and the set of safe states Q_s returned by Algorithm 1 is *empty*. Thus, the given task set is *non-schedulable*. Since, the scheduler synthesized using our scheme is *optimal*, there cannot exist a mechanism that can feasibly schedule τ_1 and τ_2.

3.3 Preemptive Scheduler Synthesis Framework

In the previous section, we have considered the non-preemptive execution of sporadic tasks. In this section, we proceed towards the design of an optimal scheduler synthesis mechanism for preemptive sporadic tasks. It may be noted that our non-preemptive framework started with the scheduler synthesis for aperiodic tasks and was extended towards sporadic tasks. Here, we directly proceed from sporadic tasks. However, the procedure presented in this section can also be easily adopted for aperiodic tasks. In addition, we ensure that the synthesized scheduler is *work-conserving*, to avoid the possibility of keeping the processor idle in the presence of ready to execute tasks.

System Model: We consider a real-time system consisting of a set I $(= \{\tau_1, \tau_2, ..., \tau_n\})$ of n (≥ 1) preemptive real-time sporadic tasks with arbitrary arrival times, to be scheduled on a uniprocessor system. Formally, a sporadic task τ_i is represented by a 3-tuple $\langle E_i, D_i, P_i \rangle$, where E_i is the *execution time*, D_i is the *relative deadline* and P_i is the *minimum inter-arrival time* [15, 23, 47, 89].

Problem Statement: Given a set of n preemptive real-time sporadic tasks with arbitrary arrival times and a single processor, design an optimal work-conserving scheduler which guarantees that all tasks meet their timing constraints (such as execution time, deadline and minimum inter-arrival time) and resource constraints (i.e., single processor).

3.3.1 Task execution model

The TDES model T_i for executing a preemptive, sporadic task τ_i $\langle E_i, D_i, P_i \rangle$ is defined as, $T_i = (Q_i,\ \Sigma,\ \delta_i,\ q_0,\ Q_m)$, where, $Q_i = \{\#1, \#2, ..., \#8\}$, $q_0 = \#1$, $Q_m = \{\#1\}$, $\Sigma = \cup_{i \in \{1,2,...,n\}} \Sigma_i \cup \{t\}$, where $\Sigma_i = \{a_i, p_i, r_i, e_i, c_i\}$.

The events are described in Table 3.3 and they are categorized as follows: (i) $\Sigma_{uc} = \cup_{i \in \{1,2,...,n\}} \{a_i\}$, (ii) $\Sigma_c = \Sigma \setminus (\Sigma_{uc} \cup \{t\})$, (iii) $\Sigma_{for} = \Sigma_c$. Since the arrival events of tasks are induced by the environment, they are modeled as *uncontrollable* events. Apart from Σ_{uc} and *tick* event (t), remaining events in the system are modeled as *controllable* events. These controllable events are also modeled as *forcible* events which can preempt the *tick*

Table 3.3: *Description of events (for preemptive execution)*

Event	Description
a_i	Arrival of a task τ_i
p_i	Acceptance of a task τ_i
r_i	Rejection of a task τ_i
e_i	Execution of a segment of τ_i on the processor
c_i	Execution of the last segment of τ_i on the processor

event (t). All events in Σ are considered to be *observable*. Suppose E_i is equal to 1, i.e., each instance of τ_i requires only a single segment of execution; then $\Sigma_i = \{a_i, p_i, r_i, c_i\}$.

In this work, the notion of acceptance captures the following fact: *Once a real-time task is accepted by the scheduler, then it will not be rejected in the middle of its execution.* Similarly, rejection captures the scenario in which *a real-time task is not accepted by the scheduler in order to guarantee timely execution of the already accepted tasks in the system.* We have modeled the notion of acceptance and rejection using (mutually exclusive) transitions on events p_i and r_i, respectively. The transition structure of T_i (shown in Figure 3.12) is explained as follows:

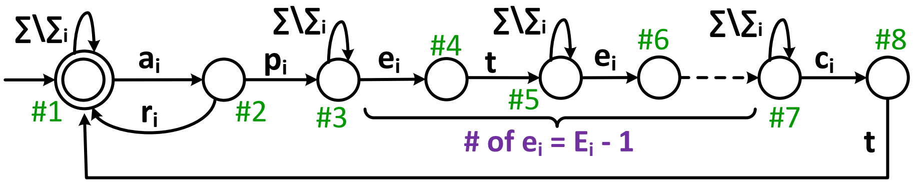


Figure 3.12: *Task execution model T_i for task τ_i $\langle E_i, D_i, P_i \rangle$.*

- $\#1 \xrightarrow{\Sigma \setminus \Sigma_i} \#1$: T_i stays at *State* #1 until the occurrence of arrival event a_i by executing the events in $\Sigma \setminus \Sigma_i$. Here, $\Sigma \setminus \Sigma_i$ contains the events such as arrival, rejection, acceptance, execution and completion of a task, that may occur with respect to other tasks in the system. In addition, $\Sigma \setminus \Sigma_i$ contains t which is used to *model the arbitrary arrival of* τ_i, i.e., a_i (no delay), ta_i (after a tick from the system start), tta_i (after two ticks) etc.

- $\#1 \xrightarrow{a_i} \#2$: On the occurrence of a_i, T_i moves to *State* #2. Here, the scheduler takes the decision whether to accept (p_i) or reject (r_i).

- #2 $\xrightarrow{r_i}$ #1: Suppose, the scheduler rejects τ_i, then T_i transits back to *State* #1.
- #2 $\xrightarrow{p_i}$ #3: Suppose, the scheduler accepts τ_i, then T_i transits to *State* #3. The self-loop transition $\Sigma \setminus \Sigma_i$ is similar to the one present in *State* #1.
- #3 $\xrightarrow{e_i}$ #4: T_i moves from *State* #3 to *State* #4 through τ_i's execution event e_i.
- #4 $\xrightarrow{t}$ #5: After the occurrence of *tick* (t), T reaches *State* #5 which is similar to *State* #3. It may be noted that e_it represents execution of τ_i for one time unit.
- #5 $\xrightarrow{e_i}$ #6 ... #7 $\xrightarrow{c_i}$ #8: By continuing in this way, T_i reaches *State* #7 and moves to *State* #8 through the completion event c_i. Note that the number of e_i events executed between the acceptance (p_i) event and the completion (c_i) event is $E_i - 1$.
- #8 $\xrightarrow{t}$ #1: After the occurrence of a *tick* event, T_i moves back to the initial (as well as marked) state #1 to represent task τ_i's completion.

Remark 3.3.1. *Generated and Marked Behavior of* T_i: The *generated* behavior $L(T_i)$ is the set of all strings that T_i (shown in Figure 3.12) can *generate*. The marked behavior $L_m(T_i)$ is the subset of all strings in $L(T_i)$ for which the terminal state belongs to $Q_m(= \{\#1\})$. The marked language may be described as: $L_m(T_i) = (s_{i0}(a_i(r_i + p_is_{i1}e_its_{i2}e_it...s_{iE_i}c_it))^*)^*$, where $s_{ij} \in (\Sigma \setminus \Sigma_i)^*$ and $j = \{0, 1, \ldots, E_i\}$. $\square$

Definition: *Deadline-meeting sequence* (with respect to preemptive execution): A sequence $x = x_1a_ix_2c_ix_3 \in L_m(T_i)$, where $x_1, x_3 \in \Sigma^*$, $x_2 \in (\Sigma \setminus \{c_i\})^*$ is *deadline-meeting*, if *tickcount*$(x_2) \leq D_i - 1$. Otherwise, x is a deadline-missing sequence. $\square$

Theorem 3.3.1. $L_m(T_i)$ contains both deadline-meeting and deadline-missing sequences of task τ_i.

Proof. Let us consider the execution of task τ_i on the processor. According to Remark 3.3.1, such execution may be represented by a sequence, $x = a_ip_ix_{ij}c_i$, where $x_{ij} = x_{i1}e_itx_{i2}x_it...x_{iE_i}$ and $x_{ij} \in (\Sigma \setminus \Sigma_i)^*$ and $j = \{1, 2, \ldots, E_i\}$. It may be noted that x_{ij} represents the self-loop $\Sigma \setminus \Sigma_i$ in T_i and contains t in it. This models the waiting time of task τ_i in the ready queue before it is being assigned onto the processor for execution. Suppose, the task τ_i is not preempted at anytime after its acceptance to completion, then *tickcount*$(x_{ij}) = E_i - 1$ ($\leq D_i$, Assumption 3). However, τ_i may be preempted by the scheduler during its execution. Let y be the total amount of time that task τ_i is kept in the ready queue between its acceptance to completion. If this preemption time (captured by y) is greater than $(D_i - E_i)$ ticks from the arrival of τ_i, then τ_i is guaranteed to miss its deadline. Hence, the sequence x is *deadline-meeting* if $(y + E_i - 1) \leq D_i$. Otherwise, x is *deadline-missing*. Thus, $L_m(T_i)$ contains both deadline-meeting and deadline-missing sequences of τ_i. □

As discussed earlier, deadline-misses cannot be tolerated in *hard real-time systems*. In later section, we develop a model that can be used to eliminate deadline-missing sequences from $L_m(T_i)$.

From the perspective of real-time task execution, a sporadic sequence may be considered as the most generic one and subsumes within it the definitions periodic and aperiodic sequences. Therefore, models specifying the behavior of periodic and aperiodic sequences may be easily derived from the model representing a sporadic sequence. The synthesis approach presented here attempts to derive scheduling sequences that satisfy sporadic task execution behavior.

Definition: *Work-conserving execution sequence [23]:* An execution sequence is *work-conserving* if and only if it never idles the processor when there exists at least one accepted task awaiting execution in the system. Otherwise, it is *non-work-conserving*.

Work-conserving execution has certain inherent advantages. Specifically, task response times under work-conserving execution may be lower than non-work-conserving execution since no processor time is wasted. Additionally, the processor time saved due to such early response may be utilized to execute other useful applications. Hence, our finally synthesized scheduler is intended to be *work-conserving*.

By extending the arguments presented in *Theorem 3.3.1*, it may be proved that *$L_m(T_i)$ represented by the task execution model T_i includes (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic, (iv) non-sporadic (v) work-conserving and (vi) non-work-conserving execution sequences for task τ_i.*

Example: Let us consider a simple two task system $\tau_1\langle 3, 6, 6\rangle$ and $\tau_2\langle 2, 4, 4\rangle$. The task execution models T_1 for τ_1 and T_2 for τ_2 are shown in Figures 3.13(a) and 3.13(b), respectively. Let us consider T_1 corresponding to task τ_1 shown in Figure 3.13(a). Using T_1, we illustrate the different types of execution sequences present in $L_m(T_1)$.

- The sequence $seq_1 = a_1 p_1\, e_1 t e_1 t c_1 t \in L_m(T_1)$ is *deadline-meeting* since $tickcount(a_1$

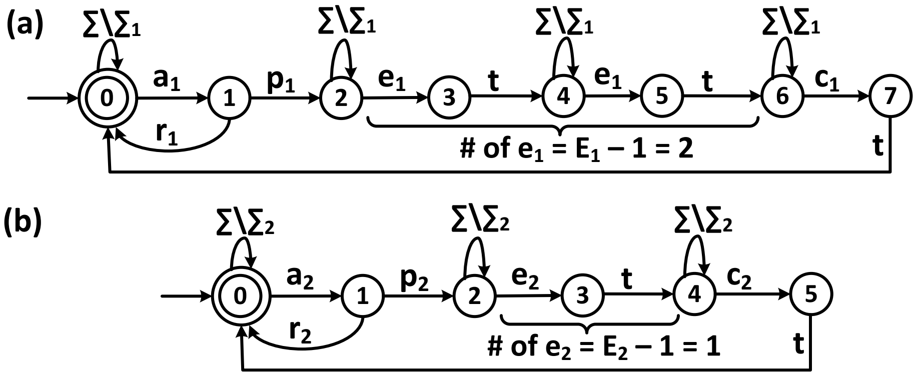


Figure 3.13: *TDES Models: (a) T_1 for $\tau_1\langle 3,6,6\rangle$, (b) T_2 for $\tau_2\langle 2,4,4\rangle$.*

$p_1e_1te_1tc_1) = 2 \leq D_1 - 1 (= 5)$. In addition, this sequence is also *work-conserving* since the processor is never kept idle when the task τ_1 is waiting for execution.

- The sequence $seq_2 = a_1p_1tttte_1te_1tc_1t \in L_m(T_1)$ is *deadline-missing* since $tickcount(a_1 p_1tttte_1te_1tc_1) = 6 \nleq D_1 - 1 (= 5)$. In addition, this sequence is also *non-work-conserving* since the processor is kept idle even when the task τ_1 is waiting for execution.

- The sequence $seq_3 = a_1p_1e_1te_1tc_1tttta_1p_1e_1te_1tc_1t \in L_m(T_1)$ is *sporadic* since $tickcount$ $(a_1p_1e_1te_1tc_1tttta_1) = 6 \geq P_1 (= 6)$.

- The sequence $seq_4 = a_1p_1e_1te_1tc_1ta_1p_1e_1te_1tc_1t \in L_m(T_1)$ is *non-sporadic* since $tickcount$ $(a_1p_1e_1te_1tc_1ta_1) = 3 \ngeq P_1 (= 6)$.

A similar discussion holds for T_2. □

3.3.2 Composite Task Execution Model

Given n individual TDES task models T_1, T_2, ..., T_n corresponding to tasks τ_1, τ_2, ... τ_n, a synchronous product composition $T = T_1||T_2||\ldots||T_n$ on the models gives us the composite model for all the tasks executing concurrently.

Theorem 3.3.2. The marked behavior $L_m(T)$ contains (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic, (iv) non-sporadic (v) work-conserving and (vi) non-work-conserving sequences for all tasks in I.

Proof. We know that $L_m(T_i)$ contains (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic, (iv) non-sporadic, (v) work-conserving and (vi) non-work-conserving sequences for task τ_i. Since $L_m(T) = \cap_{i=1}^{n} L_m(T_i)$, $L_m(T)$ *also includes all the six types of sequences mentioned above.* In addition, $L_m(T)$ does not contain any sequence that violate resource constraint since $L_m(T_i)$ satisfies them. □

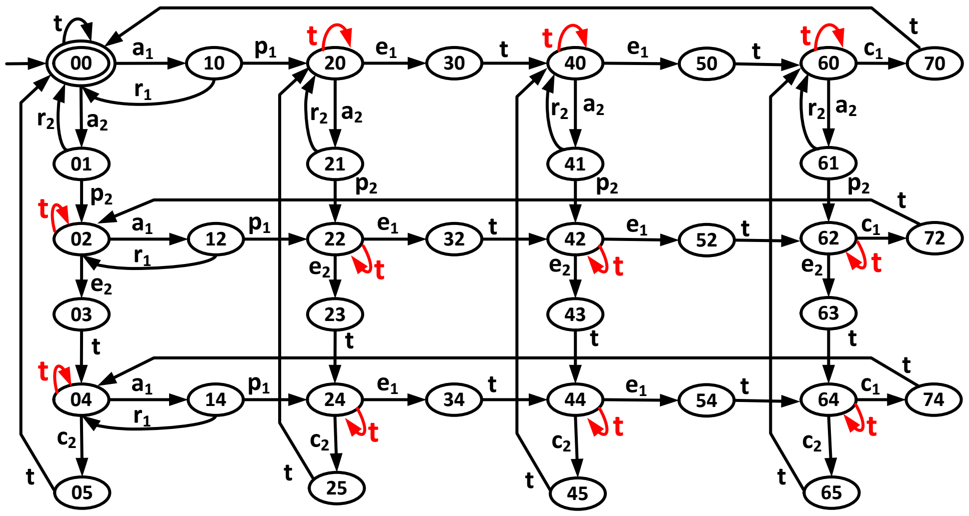


Figure 3.14: *The composite task execution model* $T = T_1||T_2$.

Example (continued): Figure 3.14 shows the composite task execution model T $(= T_1||T_2)$. Here, *State* 12 represents *State* 1 of T_1 and *State* 2 of T_2. It may be observed that $L_m(T)$ contains the different types of execution sequences in it. However, presently we limit our discussion only with *work-conserving* and *non-work-conserving* sequences.

- The sequence $seq_5 = a_1p_1a_2p_2tte_1te_1tc_1te_2tc_2t \in L_m(T)$ is *non-work-conserving* since the processor is kept idle even both tasks are waiting for execution.
- The sequence $seq_6 = a_1p_1a_2p_2e_2tc_2te_1te_1tc_1t \in L_m(T)$ is *work-conserving* since the processor is never kept idle when a task is waiting for execution.

3.3.3 Removal of Non-work-conserving Sequences

An important design objective of this work is to synthesize a *work-conserving* scheduler. However, during the construction of T using a synchronous product of T_1, T_2, $\ldots$, T_n, the presence of the self-loop $\Sigma \setminus \Sigma_i$ in the individual models results in a self-loop $\{t\}$ in the composite model T. That is, $(\Sigma \setminus \Sigma_1) \cap (\Sigma \setminus \Sigma_2) \cap \ldots \cap (\Sigma \setminus \Sigma_n) = t$. The presence of self-loops $\{t\}$ (except at the initial state) in T introduces *non-work-conserving* sequences in $L_m(T)$ by possibly allowing the progression of time without actual task execution on the processor. We now present a methodology to construct a *work-conserving* composite task execution model T' from T in Algorithm 4.

ALGORITHM 4: T' CONSTRUCTION

```
  Input: $T = (Q, \Sigma, \delta, q_0, Q_m)$
  Output: $T'$
1 begin
2     foreach state $q \in T$ and $q \neq q_0$ do
3         if ($\delta(q, t) == q$) then
4             // Delete the self-loop transition $t$ from state $q$;
5             $\delta(q, t) = \emptyset$;;
6     Let the resultant automata obtained after carrying out the above transformations
      on $T$ be denoted by $T'$;
```

The above construction procedure removes the self-loop transitions on t from all states in T except from the initial state. The reason for excluding the initial state is to support the arbitrary arrival of tasks during concurrent execution.

Theorem 3.3.3. $L_m(T')$ contains all the work-conserving sequences of $L_m(T)$.

Proof. As the construction procedure of T' from T removes only the self-loop transitions on t, T' eliminates the sequences in T that may possibly lead to non-work-conserving execution. Therefore, $L_m(T')$ contains *all* the work-conserving sequences of $L_m(T)$. □

Remark 3.3.2. It may be noted that $L_m(T')$ *includes (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic and (iv) non-sporadic execution sequences for task* τ_i. However, all these sequences are *work-conserving.*

Example (continued): Figure 3.15 shows the transition structure of T' obtained from T using Algorithm 4. Let us verify whether $L_m(T')$ contains the *non-work-conserving* sequence $seq_5 = a_1p_1a_2p_2tte_1te_1tc_1te_2tc_2t \in L_m(T)$ in it. After proceeding through the states $\langle 00\rangle \xrightarrow{a_1} \langle 10\rangle \xrightarrow{p_1} \langle 20\rangle \xrightarrow{a_2} \langle 21\rangle \xrightarrow{p_2} \langle 22\rangle$, T' *gets blocked* due to the absence of a transition on t at State $\langle 22\rangle$. Hence, $seq_5 \notin L_m(T')$.

Let us consider the *work-conserving* sequence $seq_6 = a_1p_1a_2p_2e_2tc_2te_1te_1tc_1t \in L_m(T)$. It can be retrieved in T' by tracing the states $\langle 00\rangle \xrightarrow{a_1} \langle 10\rangle \xrightarrow{p_1} \langle 20\rangle \xrightarrow{a_2} \langle 21\rangle \xrightarrow{p_2} \langle 22\rangle \xrightarrow{e_2}$

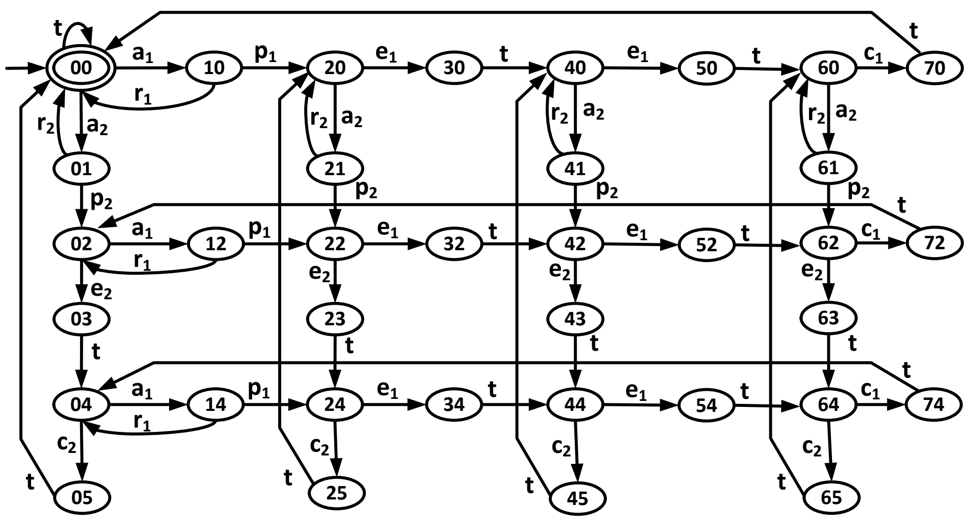


Figure 3.15: *Work-conserving composite task execution model T'.*

$\langle 23\rangle \xrightarrow{t} \langle 24\rangle \xrightarrow{c_2} \langle 25\rangle \xrightarrow{t} \langle 20\rangle \xrightarrow{e_1} \langle 30\rangle \xrightarrow{t} \langle 40\rangle \xrightarrow{e_1} \langle 50\rangle \xrightarrow{t} \langle 60\rangle \xrightarrow{c_1} \langle 70\rangle \xrightarrow{t} \langle 00\rangle$. Hence, $seq_6 \in L_m(T)$ is present in $L_m(T')$. That is, $L_m(T')$ *contains the work-conserving sequences of* $L_m(T)$. We discuss the presence of sequences apart from work-conserving in $L_m(T')$.

- $seq_7 = a_1p_1a_2p_2e_1te_1tc_1te_2tc_2t \in L_m(T')$ is *deadline-missing* since $tickcount(a_2p_2e_1t\ e_1tc_1te_2tc_2) = 4 \nleq D_2 - 1(= 3)$.

- $seq_8 = a_1p_1a_2p_2e_2tc_2te_1ta_2p_2e_1tc_1te_2tc_2t \in L_m(T')$ is *deadline-meeting* execution sequence. This is because, $tickcount(a_1p_1a_2p_2e_2tc_2te_1ta_2p_2e_1tc_1) = 4 \leq D_1-1(= 5)$ (task τ_1 meets its deadline) and $tickcount(a_2p_2e_2tc_2) = 1 \leq D_2 - 1(= 3)$ (the first instance of task τ_2 meets its deadline) and $tickcount(a_2p_2e_1tc_1te_2tc_2) = 3 \leq D_2 - 1(= 3)$ (the second instance of task τ_2 meets its deadline). In addition, seq_8 is a *non-sporadic* sequence, i.e., $tickcount(a_2p_2e_2tc_2te_1ta_2) = 3 \ngeq P_2(= 4)$ (task τ_2 violates its sporadicity). Hence, the sequence seq_8 is a *deadline-meeting, non-sporadic* sequence.

- $seq_9 = a_1p_1a_2p_2e_2tc_2te_1te_1t\ a_2p_2c_1te_2tc_2tt \in L_m(T')$ is *deadline-meeting, sporadic sequence.* This is because, (i) $tickcount(a_1p_1a_2p_2e_2tc_2te_1te_1ta_2p_2c_1) = 4 \leq D_1 - 1(= 5)$ (task τ_1 meets its deadline), (ii) $tickcount(a_2p_2e_2tc_2) = 1 \leq D_2 - 1(= 3)$

(the first instance of task τ_2 meets its deadline), (iii) $tickcount(a_2p_2c_1te_2tc_2) = 2 \leq D_2 - 1(= 3)$ (the second instance of task τ_2 meets its deadline) and (iv) $tickcount(a_2p_2e_2tc_2te_1te_1ta_2) = 4 \geq P_2(= 4)$ (task τ_2 meets its sporadicity).

To summarize, the marked behavior $L_m(T')$ contains deadline-meeting and deadline-missing execution sequences with arbitrary inter-arrival times. In order to restrict the marked behavior to correctly capture only those sequences which satisfy all task deadlines and inter-arrival time constraints, the timing specification models corresponding to each sporadic task τ_i are developed.

3.3.4 Timing Specification Model

The timing specification model SH_i (shown in Figure 3.16) for sporadic task $\tau_i\langle E_i, D_i, P_i\rangle$ is defined as follows:

$$SH_i = (SQ_i, \Sigma, S\delta_i, Sq_0, SQ_m),$$

where, $SQ_i = \{0, 1, ..., 13\}$, $Sq_0 = 0$, $SQ_m = \{0, 13\}$, $\Sigma = \cup_{i=\{1,2,...,n\}}\Sigma_i \cup \{t\}$, where $\Sigma_i = \{a_i, p_i, r_i, e_i, c_i\}$.

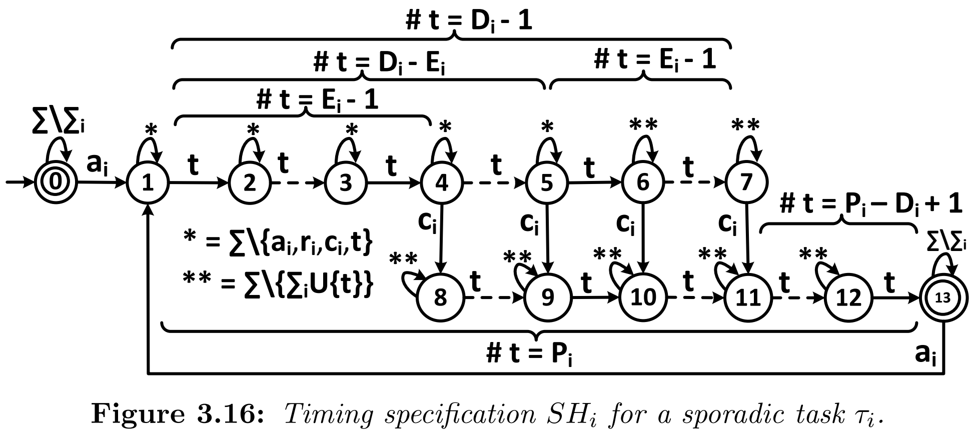


Figure 3.16: *Timing specification SH_i for a sporadic task τ_i.*

The self-loop $\Sigma \setminus \Sigma_i$ at *State* 0 excludes the events (Σ_i) that are associated with τ_i since it has not yet arrived. However, it does not restrict the events that may happen with respect to other tasks in the system. Since $\Sigma \setminus \Sigma_i$ includes the *tick* event, SH_i can correctly model the arbitrary arrival time of a task τ_i. After the occurrence of arrival event a_i, SH_i reaches *State* 1. The self-loop transition * $(= \Sigma \setminus \{a_i, r_i, c_i, t\})$ in *State* 1 is used to model the fact that task τ_i is allowed to take only the events from the set $\{p_i, e_i\}$ associated with it and without imposing any restriction with respect to other tasks in the system. After the elapse of one tick event, SH_i reaches *State* 2 in which the self-loop is similar to that in *State* 1. Since τ_i is not allowed to execute event c_i before the elapse of $E_i - 1$ tick events (because at least $E_i - 1$ time ticks must be incurred before signaling the completion of τ_i), states that are similar to *State* 1 and *State* 2 are instantiated $E_i - 1$ times starting from *State* 1. Following that, at *State* 4, task τ_i is allowed to execute c_i in addition to $\{p_i, e_i\}$ because τ_i is allowed to complete after executing for $E_i - 1$ ticks subsequent to arrival.

It may be noted that if τ_i does not start its execution at least E_i ticks before its deadline, then τ_i is guaranteed to miss its deadline. Hence, $\{p_i, e_i\}$ is allowed only up to $D_i - E_i$ ticks from the arrival of τ_i which is captured in the self-loop transitions present at states 1 to 5. When the time remaining before deadline is less than $E_i - 1$ ticks, $\{p_i, e_i\}$ is disallowed which is captured by the self-loop transition $\Sigma \setminus \{\Sigma_i \cup \{t\}\}$ present at states 6 to 7.

Suppose, the sporadic task τ_i executes c_i, the completion event, $E_i + j$ ticks ($j = \{0, 1, 2, ..., (D_i - E_i)\}$) subsequent to its arrival. Then, SH_i will reach anyone (say, *State* x) among the states 8, 9, ..., 11, such that all substrings between *State* 1 and *State* x contain $E_i + j$ ticks. Specifically, SH_i reaches *State 11* after $D_i - 1$ ticks from the arrival of τ_i's current instance. From *State 11* onwards, SH_i does not contain any transition on c_i to disallow the possibility of deadline violation. After the elapse of $P_i - D_i$ ticks, SH_i reaches *State 12* from *State 11.* The self-loop $\Sigma \setminus \{\Sigma_i \cup \{t\}\}$ at these states ($\{8, 9, ..., 12\}$) does not allow any events related to τ_i to guarantee the minimum inter-arrival time. Finally, SH_i reaches *State 13* after the elapse of P_i ticks from the arrival of its current instance. At *State 13*, the self-loop $\Sigma \setminus \Sigma_i$ allows the possibility of arbitrary arrival of task τ_i's next instance. On the occurrence of a_i (i.e., τ_i's next instance), SH_i transits back to *State 1.* Hence, it may be concluded that SH_i models both deadline and minimum inter-arrival time constraints of the sporadic task τ_i. Therefore, $L_m(SH_i)$ *contains all the deadline-meeting, sporadic execution sequences for* τ_i.

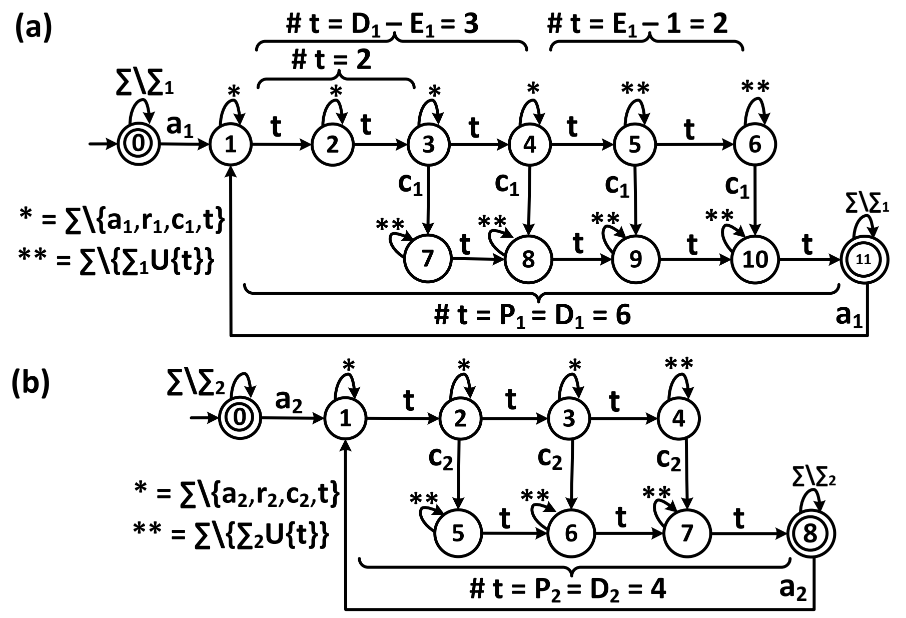


Figure 3.17: *(a) SH_1 for τ_1, (b) SH_2 for τ_2.*

Example (continued): The timing specification SH_1 for $\tau_1\langle 3,6,6\rangle$ and SH_2 for $\tau_2\langle 2,4,4\rangle$ are shown in Figures 3.17(a) and 3.17(b), respectively. □

3.3.5 Composite Timing Specification Model

Given n individual TDES timing specification models, a synchronous product composition on the models gives us the composite model for all the tasks executing concurrently. That is, the composite timing specification model of sporadic tasks (denoted by SH) can be obtained as: $SH = SH_1||SH_2||\ldots||SH_n$.

Theorem 3.3.4. $L_m(SH)$ contains all and only the sequences that satisfy the timing specification (i.e., deadline and sporadicity) of all sporadic tasks in I.

Proof. We know that $L_m(SH_i)$ contains all the sequences that satisfy the timing specification of a sporadic task τ_i. Since $L_m(SH) = \cap_{i=1}^{n} L_m(SH_i)$, the marked behavior $L_m(SH)$ *represented by the composite timing specification model includes all and only the sequences that satisfy the timing specification of all sporadic tasks in I.* □

It may be observed that the timing specification model SH_i does not restrict the simultaneous execution of multiple tasks on the processor. Specifically, the self-loops at any state in SH_i do not restrict the execution of multiple tasks and they can be executed multiple times at each state to allow simultaneous execution. Hence, *the deadline-meeting sequences in $L_m(SH_i)$ may violate resource constraints.* Consequently, *the deadline-meeting sequences in $L_m(SH)$ may violate resource constraints.*

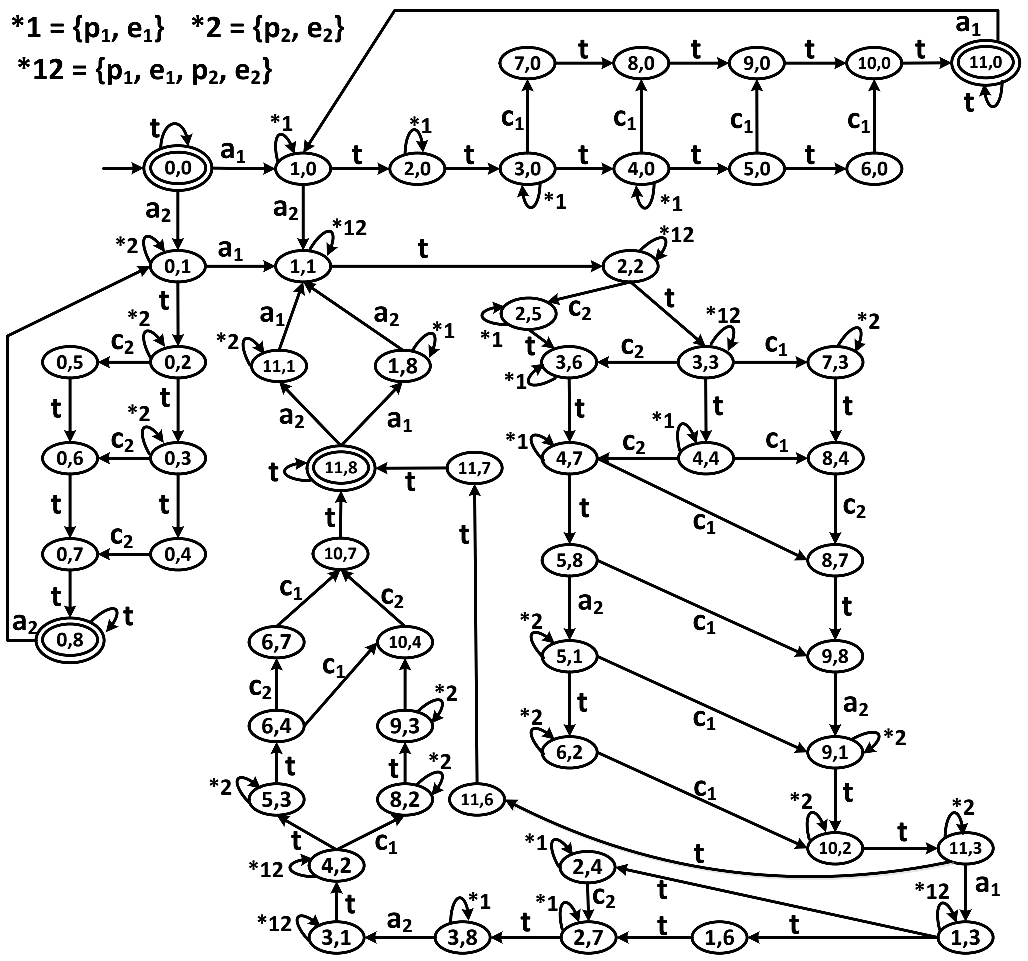


Figure 3.18: *$SH = SH_1||SH_2$ (partial diagram).*

Example (continued): Figure 3.18 shows the composite model SH ($= SH_1||SH_2$). Here, *State* 1, 2 represents *State* 1 of SH_1 and *State* 2 of SH_2. As mentioned earlier, this model represents all possible deadline-meeting, sporadic execution sequences for τ_1 and τ_2. In order to illustrate this fact, let us try to find the deadline-missing sequence $seq_7 = a_1p_1a_2p_2e_1te_1tc_1te_2tc_2t \in L_m(T')$ in the composite model SH shown Figure 3.18. After proceeding through the states, $\langle 0,0\rangle \xrightarrow{a_1} \langle 1,0\rangle \xrightarrow{p_1} \langle 1,0\rangle \xrightarrow{a_2} \langle 1,1\rangle \xrightarrow{p_2} \langle 1,1\rangle \xrightarrow{e_1} \langle 1,1\rangle \xrightarrow{t} \langle 2,2\rangle \xrightarrow{e_1} \langle 2,2\rangle \xrightarrow{t} \langle 3,3\rangle \xrightarrow{c_1} \langle 7,3\rangle \xrightarrow{t} \langle 8,4\rangle$, the composite model SH *gets* *blocked* due to the absence of a transition on e_2 at *State* $\langle 8,4\rangle$. More specifically, after processing the sub-string $a_1p_1a_2p_2e_1te_1tc_1t$ of seq_7, the next event in seq_7 is e_2. However, e_2 is not present at *State* $\langle 8,4\rangle$. Thus, $L_m(SH)$ *does not contain the deadline-missing sequence* seq_7.

Let us consider the *deadline-meeting, non-sporadic* sequence $seq_8 = a_1p_1a_2p_2e_2t\ c_2te_1ta_2p_2e_1tc_1te_2tc_2t \in L_m(T')$. After proceeding through the states $\langle 0,0\rangle \xrightarrow{a_1} \langle 1,0\rangle \xrightarrow{p_1} \langle 0,1\rangle \xrightarrow{a_2} \langle 1,1\rangle \xrightarrow{p_2} \langle 1,1\rangle \xrightarrow{e_2} \langle 1,1\rangle \xrightarrow{t} \langle 2,2\rangle \xrightarrow{c_2} \langle 2,5\rangle \xrightarrow{t} \langle 3,6\rangle \xrightarrow{e_1} \langle 3,6\rangle \xrightarrow{t} \langle 4,7\rangle \xrightarrow{c_1} \langle 8,7\rangle \xrightarrow{t} \langle 9,8\rangle$, the composite model SH *gets blocked* due to the absence of a transition on e_2 at *State* $\langle 9,8\rangle$. More specifically, after processing the sub-string $a_1p_1a_2p_2e_2tc_2te_1ta_2p_2e_1tc_1t$ of seq_8, the next event in seq_8 is e_2. However, e_2 is not present at *State* $\langle 9,8\rangle$. Thus, $L_m(SH)$ *does not contain the deadline-meeting, non-sporadic sequence* seq_8.

Now, consider the deadline-meeting, sporadic sequence $seq_9 = a_1p_1a_2p_2e_2tc_2te_1\ te_1ta_2p_2\ c_1te_2tc_2tt \in L_m(T')$. It can be retrieved by tracing the states $\langle 0,0\rangle \xrightarrow{a_1} \langle 1,0\rangle \xrightarrow{p_1} \langle 0,1\rangle \xrightarrow{a_2} \langle 1,1\rangle \xrightarrow{p_2} \langle 1,1\rangle \xrightarrow{e_2} \langle 1,1\rangle \xrightarrow{t} \langle 2,2\rangle \xrightarrow{c_2} \langle 2,5\rangle \xrightarrow{t} \langle 3,6\rangle \xrightarrow{e_1} \langle 3,6\rangle \xrightarrow{t} \langle 4,7\rangle \xrightarrow{e_1} \langle 4,7\rangle \xrightarrow{t} \langle 5,8\rangle \xrightarrow{a_2} \langle 5,1\rangle \xrightarrow{p_2} \langle 5,1\rangle \xrightarrow{c_1} \langle 9,1\rangle \xrightarrow{t} \langle 10,2\rangle \xrightarrow{e_2} \langle 10,2\rangle \xrightarrow{t} \langle 11,3\rangle \xrightarrow{c_2} \langle 11,6\rangle \xrightarrow{t} \langle 11,7\rangle \xrightarrow{t} \langle 11,8\rangle$. Hence, $seq_9 \in L_m(SH)$. □

3.3.6 Scheduler Synthesis

In order to find all deadline-meeting sequences from $L_m(T')$, we construct the finite state automaton $M = T'||H$.

Theorem 3.3.5. *$L_m(M)$ contains only and all the (i) work-conserving, (ii) deadline-meeting and (iii) sporadic execution sequences of $L_m(T')$.*

Proof. We know that $L_m(T')$ *includes (i) deadline-meeting, (ii) deadline-missing, (iii) sporadic and (iv) non-sporadic execution sequences* (Remark 3.3.2). On the other hand, $L_m(SH)$ contains all possible deadline-meeting, sporadic sequences (Theorem 3.3.4). Since $L_m(M) = L_m(T') \cap L_m(SH)$, $L_m(M)$ contains *only* and *all* the (i) work-conserving, (ii) deadline-meeting and (iii) sporadic sequences of $L_m(T')$. □

Remark 3.3.3. Apart from the removal of deadline-missing sequences in $L_m(T')$, the synchronous product also eliminates the sequences in $L_m(SH)$ that violate resource constraint. This is because, $L_m(T')$ does not contain any sequence that violates resource constraint. □

Although $L_m(M)$ is deadline-meeting, the deadline-missing sequences in T' and resource-constraint violating sequences in SH, lead to *deadlock* states in M, i.e., M is *blocking*. To ensure that all the accepted tasks meet their individual deadlines, a scheduler should be designed to achieve $L_m(M)$, i.e., the scheduler must be able to avoid reaching any non-co-reachable state of M which may lead to deadlock states. This requires that M must be controllable with respect to T'. Specifically, M must be controllable by disabling certain controllable events (r_i, p_i, e_i and c_i) present at corresponding states, such that none of the deadlock states in M are reached. This guarantees that the closed-loop system behavior $L(M/T')$ stays always within the desired behavior $L_m(M)$. However, in general, $L_m(M)$ may not always be controllable with respect to $L_m(T')$. That is, all instances of all sporadic tasks in the given real-time cannot be scheduled to meet the timing and resource constraints. Under such a scenario, rather than accepting and scheduling all task instances, we may reject a newly arrived task to guarantee the feasible scheduling of already accepted tasks. In order to handle this scenario, we present a methodology to construct M' from M in Algorithm 5.

ALGORITHM 5: M' CONSTRUCTION

```
Input: $M = (Q, \Sigma, \delta, q_0, Q_m)$, $n$ (number of tasks)
Output: $M'$
1  begin
2      foreach co-reachable state $q$ of $M$ do
3          for $i = 1$ to $n$ do
4              if $(\delta(q, a_i) \neq \emptyset)$ then
5                  Determine the sequence $s \in \overline{L_m(M)}$ such that $\delta(q_0, s) = q$;
6                  if $(sa_i \notin \overline{L_m(M)})$ then
7                      Find $q' = \delta(q, a_i)$; // $q'$ is a non-co-reachable state of $M$;
8                      $\delta(q', r_i) = q$; // Add $r_i$ from $q'$ to $q$;
9                      $\delta_M(q', p_i) = \emptyset$; // Delete $p_i$ from $M$;
10                     $M = trim(M)$; // Non-blocking $M$ ;
11     Let the resultant automata obtained be denoted by $M'$;
```

The above construction procedure removes the non-co-reachable (deadlock) states from M created by the acceptance of a task. Specifically, M' rejects (r_i) any newly arrived task that may lead the system to a deadlock state.

Theorem 3.3.6. $L_m(M')$ contains only and all the deadline-meeting sequences of $L_m(M)$.

Proof. Let us consider a sequence $uv \in L_m(M)$ such that $u \in \overline{L_m(M)}$, $\delta_M(q_0, u) = q$, $\delta_M(q, a_i)$ is defined, $i \in \{1, 2, ..., n\}$ and $v \in \Sigma^*$. If $\delta_M(q, a_i) = q'$ and q' is a non-co-accessible state, then $ua_i \notin \overline{L_m(M)}$. Algorithm 5 transforms M to M' by adding the rejection transition from q' to q, i.e., $\delta_M(q', r_i) = q$ which makes q' co-accessible. Therefore $ua_ir_i \in \overline{L_m(M')}$ and $ua_i \in \overline{L_m(M')}$. Since, a_ir_i do not contain any *tick* event in it, the incorporation of sequence ua_ir_iv do not introduce any deadline-missing sequence in $L_m(M')$. In general, the sequences of the form $us_mv \in L_m(M') \setminus L_m(M)$, where $s_m \in \{a_1r_1, a_2r_2, ..., a_nr_n\}$ makes q' to be co-accessible; hence, $ua_ir_i \in \overline{L_m(M')}$. Therefore, $L_m(M')$ contains only and all the deadline-meeting sequences of $L_m(M)$. □

Apart from deadline-meeting sequences, it may also be proved that $L_m(M')$ contains work-conserving and sporadic sequences of $L_m(M)$. Hence, it may be inferred that $L_m(M')$ is the largest schedulable language that contains all feasible scheduling sequences for the given real-time system. Therefore, an optimal supervisor (or scheduler) S which follows $\overline{L_m(M')}$ can be designed, as follows: for any $s \in \overline{L_m(M')}$, $S(s) := \{\sigma \in \Sigma_c | s\sigma \in \overline{L_m(M')}\} \cup \{\sigma \in \Sigma_{uc} | \delta(q_0, s\sigma) \text{ is defined}\}$, which denotes the set of events that are enabled after observation of the string s (without restricting the possibility of any eligible uncontrollable event at q). As a result of supervision, we obtain $L(S/T') = \overline{L_m(M')}$, i.e., the task executions controlled by the optimal scheduler remains within the schedulable language. The sequence of steps involved in the scheduler synthesis framework has been summarized in Figure 3.19.

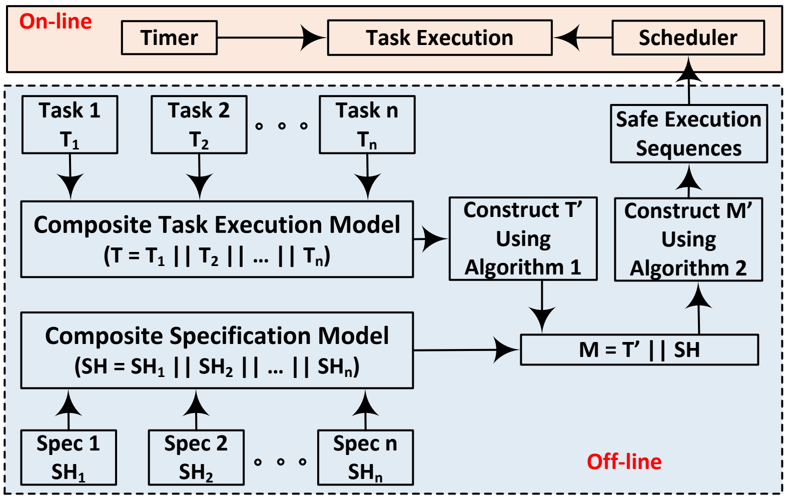

On-line
Timer
Task Execution
Scheduler
Task 1 T₁
Task 2 T₂
Task n Tₙ
Safe Execution Sequences
Composite Task Execution Model (T = T₁ || T₂ || … || Tₙ)
Construct T' Using Algorithm 1
Construct M' Using Algorithm 2
Composite Specification Model (SH = SH₁ || SH₂ || … || SHₙ)
M = T' || SH
Spec 1 SH₁
Spec 2 SH₂
Spec n SHₙ
Off-line

Figure 3.19: *Proposed Scheduler Synthesis Framework.*

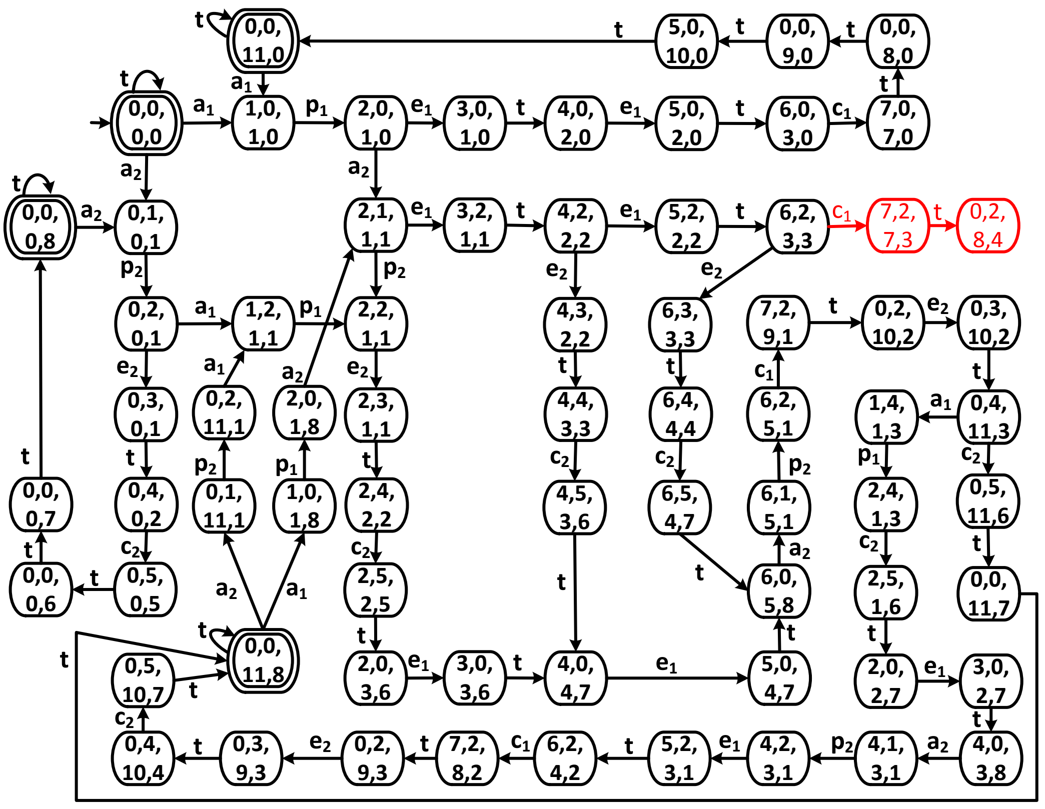


Figure 3.20: *Partial diagram of $M = T'||SH$ and M' (except states $\langle 7, 2, 7, 3\rangle$ and $\langle 0, 2, 8, 4\rangle$)*

Example (continued): Figure 3.20 shows the (partial) transition structure of M ($= T'||SH$). Here, *State* $\langle 1, 2, 3, 4\rangle$ represents *State* 1 of T_1, *State* 2 of T_2, *State* 3 of SH_1 and *State* 4 of SH_2. As mentioned earlier, deadline-missing sequences in $L_m(T')$ lead to deadlock states in M. For example, let us consider the deadline-missing sequence $a_1p_1a_2p_2e_1te_1tc_1te_2tc_2t \in L_m(T')$. After processing the sub-string $a_1p_1a_2p_2e_1te_1tc_1t$, M reaches the deadlock state $\langle 0, 2, 8, 4\rangle$. We apply Algorithm 5 to construct M' from M.

As discussed earlier, M' represents all possible deadline-meeting, sporadic execution sequences for τ_1 and τ_2. In order to illustrate this fact, let us try to find the sequence $seq_9 = a_1p_1a_2p_2e_2tc_2te_1te_1ta_2p_2c_1te_2tc_2tt \in L_m(T')$. It can be retrieved by tracing the states $\langle 0, 0, 0, 0\rangle \xrightarrow{a_1} \langle 1, 0, 1, 0\rangle \xrightarrow{p_1} \langle 2, 0, 1, 0\rangle \xrightarrow{a_2} \langle 2, 1, 1, 1\rangle \xrightarrow{p_2} \langle 2, 2, 1, 1\rangle \xrightarrow{e_2} \langle 2, 3, 1, 1\rangle \xrightarrow{t} \langle 2, 4, 2, 2\rangle \xrightarrow{c_2} \langle 2, 5, 2, 5\rangle \xrightarrow{t} \langle 2, 0, 3, 6\rangle \xrightarrow{e_1} \langle 3, 0, 3, 6\rangle \xrightarrow{t} \langle 4, 0, 4, 7\rangle \xrightarrow{e_1} \langle 5, 0, 4, 7\rangle \xrightarrow{t} \langle 6, 0, 5, 8\rangle \xrightarrow{a_2} \langle 6, 1, 5, 1\rangle \xrightarrow{p_2} \langle 6, 2, 5, 1\rangle \xrightarrow{c_1} \langle 7, 2, 9, 1\rangle \xrightarrow{t} \langle 0, 2, 10, 2\rangle \xrightarrow{e_2} \langle 0, 3, 10, 2\rangle \xrightarrow{t} \langle 0, 4, 11, 3\rangle \xrightarrow{c_2} \langle 0, 5, 11, 6\rangle \xrightarrow{t} \langle 0, 0, 11, 7\rangle \xrightarrow{t} \langle 0, 0, 11, 8\rangle$. Hence, $seq_9 \in L_m(M')$. We may observe that $seq_9 \in L_m(M')$ represents the concurrent execution of τ_1 and τ_2.

Using $L_m(M')$, the final scheduler S is constructed. Let us discuss how S controls the execution of tasks on the processor by considering the sequence seq_9 (=$a_1p_1a_2p_2e_2tc_2te_1t$ $e_1ta_2p_2c_1te_2tc_2tt$). Firstly, the task τ_1 arrives and accepted by the scheduler for execution (i.e., a_1p_1). Similarly, the task τ_2 arrives and is accepted (i.e., a_2p_2). Among these two accepted tasks, the scheduler S allows the execution of τ_2 until its completion (e_2tc_2t). Then, S allows the task τ_1 to execute for two ticks (e_1te_1t). Meanwhile, the next instance of τ_2 arrives and is accepted for execution by the scheduler (a_2p_2). After accepting τ_2, the scheduler S still allows τ_1 to execute for one more tick so that it completes its execution (c_1t). Then, S allows τ_2 to execute until its completion (e_2tc_2t). Since, there are no tasks to execute further, S keeps the processor idle by allowing the occurrence of the tick event. Hence, S correctly schedules τ_1 and τ_2 such that both of them meet their timing constraints.

Discussion: EDF (Earliest Deadline First) is an optimal, work-conserving algorithm for the scheduling of real-time preemptive tasks on uniprocessors [23, 73, 89]. EDF produces a single schedule for a given taskset. On the other hand, the SCTDES based scheduler synthesis procedure finds out all possible feasible schedules corresponding to a given taskset [29]. This empowers the designer to select one or more scheduling solutions which best fits the requirements of a given system scenario under consideration. For example, the designer may select one among a subset of schedules which results in the minimum number of preemptions while simultaneously being work-conserving.

In case of non-preemptive real-time tasksets on uniprocessors, EDF is not guaranteed to find a feasible schedule. On the contrary, the SCTDES based scheduler synthesis approach ensures determination of a feasible schedule, if one actually exists. This can also be observed from an example discussed in Section 3.2.2. Specifically, the deadline-missing sequence seq_1 in Figure 3.3 corresponds to an EDF schedule.

3.3.7 Case Study: Instrument Control System

In this section, we illustrate the work-conserving nature of the scheduler synthesized using our proposed scheme through a real-world case study. For this purpose, let us

Table 3.4: *Task parameters over different task sets*

Task set	Task τ_1		Task τ_2		Task τ_3		Task τ_4		Task τ_5		#Idle slots
	E_1	D_1	E_2	D_2	E_3	D_3	E_4	D_4	E_5	D_5	
I_1	2	10	3	20	1	20	1	25	2	25	171
I_2	3	25	5	50	2	50	2	60	3	60	197
I_3	6	60	8	120	5	120	5	140	8	140	210
I_4	10	125	12	250	10	250	12	300	15	300	227

consider an *Instrument Control System* (*Instrument Control System* (ICS)) which is used for collecting and analyzing diagnostic information of the electrical and electronic components in a modern automotive system [88]. This system is modeled as a collection of five real-time tasks, τ_1: Mode management, τ_2: Mission data management, τ_3: Instrument monitoring, τ_4: Instrument configuration, and τ_5: Instrument processing. We consider four different task sets (I_1, I_2, I_3 and I_4), all of which contain the five tasks τ_1 to τ_5. However, for each task, the execution times and deadlines are assumed to monotonically increase from I_1 to I_4. The deadlines (D_i) and minimum inter-arrival times (P_i) have been considered to be same for all tasks. The task parameters under each task set are listed in Table 3.4.

By following our proposed scheduler synthesis framework, we can synthesize an optimal work-conserving scheduler for each task set I_i ($i = 1, 2, 3, 4$). The maximum amount of idle time slots (denoted by IPT) saved through our work-conserving scheduling scheme, over the time duration of 300 time slots from the system start, is listed in the last column of Table 3.4. It may be observed that IPT obtained over the task set increases monotonically (from I_1 to I_4). This is because the amount of increase in the execution times (E_i) is less than the increase in deadlines (D_i), over task sets I_1 to I_4. In general, the total amount of idle processor time slots obtained using the our scheduler, will vary depending on the individual task set parameters. The obtained processor times may be useful in executing lower criticality best-effort tasks in the system.

3.4 Summary

In this chapter, we have presented scheduler synthesis frameworks for real-time sporadic tasks executing (non-preemptively / preemptively) on uniprocessors. The first framework considered the scheduling of a set of known / dynamically arriving aperiodic tasks. Then, proposed models have been extended towards sporadic tasks. We have illustrated the scheduler synthesis process using a motor network example. The second framework proposed the synthesis of preemptive scheduler for sporadic tasks. The synthesized scheduler is guaranteed to be *work-conserving*, i.e., it will never keep the processor idle in the presence of ready to execute tasks. The scheduler is also able to support concurrent execution of multiple accepted tasks and correctly model the inter-arrival time requirement of a sporadic task. The practical applicability of our proposed framework has been illustrated using a Industrial control system example. Although, in recent years, there has been a few significant works dealing with real-time scheduling using SCTDES, this is possibly the first work which addresses the scheduler synthesis problem for sporadic tasks. In the next chapter, we consider the fault-tolerant scheduler synthesis for a set of real-time tasks executing on a homogeneous multiprocessor platform.

Chapter 4

Fault-tolerant Scheduling of Aperiodic Tasks

In the previous chapter, we have assumed the underlying hardware processing platform to be *fault-free*. However, the computing platforms are subjected to permanent and transient faults [62]. *Permanent faults* are irreparable and affect the functionality of the processor for its entire life-time. On the other hand, *transient faults* are momentary and the processor resumes its normal operation after sometime. Thus, tolerating faults plays a crucial role in order to ensure guaranteed performance of a real-time system. In this chapter, we develop task execution and specification models which can be used to synthesize a multiple permanent processor fault-tolerant scheduler for a multiprocessor system executing a set of dynamically arriving aperiodic tasks. For this purpose, we extend the models developed in the previous chapter from uniprocessor to multiprocessor platform, and empower them with the ability to tolerate single permanent processor fault. As the number of processors in the system increases, the state space of the final supervisor also increases exponentially as compared to its uniprocessor counter-part. Hence, the synthesis framework has been empowered with a novel BDD based symbolic computation mechanism to control the exponential state-space complexity. We illustrate the scheduler synthesis process using an illustrative example starting from the inidividual models to the final supervior, under both TDES and BDD-based symbolic computation schemes. Later, we extend our proposed models to handle multiple permanent processor faults. Finally, we present a comparative study of our proposed scheme along with the SCTDES based state-of-the-art fault-tolerant scheduling scheme [86].

4.1 Related Works

Many researchers have proposed techniques for on-line fault-tolerant scheduling of aperiodic tasks. Ghosh et al. [48] proposed a Primary-Backup based multiprocessor algorithm for the on-line scheduling of dynamically arriving aperiodic, non-preemptive tasks. The algorithm attempts to achieve high resource utilization by maintaining on a processor the overlapped backups of those tasks whose primaries are mapped on mutually distinct processors. Then, it deallocates the backup corresponding to the primary which has executed without any faults to reclaim the resource allocated to the backup. However, their approach can tolerate more than one processor failure only if successive permanent faults are separated by large time intervals. In [3], authors proposed a slack distribution technique to incorporate both fault-tolerance as well as energy minimization in heterogeneous distributed *Real-Time* (RT) dynamic task systems consisting of periodic, sporadic and aperiodic tasks. The model consists of a checkpoint based fault-tolerance mechanism combined with Dynamic Voltage Scaling during busy periods and Dynamic Power Management during idle intervals for energy management. Aperiodic tasks are assumed to be non-critical and accepted only in the absence of periodic and sporadic tasks.

Apart from on-line approaches, many researchers have proposed off-line techniques for the fault-tolerant scheduling of aperiodic tasks. Given a set of preemptive aperiodic tasks scheduled using Earliest Deadline First (EDF) on a uniprocessor system, Liberato et al. [72] developed a necessary and sufficient feasibility checking algorithm for fault-tolerant scheduling with the ability to tolerate multiple transient faults. Their feasibility-check uses an off-line dynamic programming technique to explore all possible fault patterns in the system. This approach is only applicable when all task release times are known *a priori*. In [71], the authors have addressed the problem of multiprocessor global scheduling for preemptive RT tasks so that the timeliness of the system can be guaranteed even in the presence of faults. They presented a frame-based model to schedule aperiodic task sets such that at most one transient fault may be tolerated in each frame. They provided an optimal fault-tolerant on-line scheduling strategy for aperiodic task sets with identical recovery times. However, for optimal fault tolerant scheduling with generic task sets, they recommended off-line exhaustive generation of all feasible schedules with dynamic switching between the schedules as needed. Further, they presented a heuristic approach for the general case. Ripoll et al. [94] presented an EDF based slack stealing algorithm which provides a solution to the problem of co-scheduling periodic and aperiodic tasks in dynamic-priority preemptive (uniprocessor) systems. During its off-line phase, the algorithm computes a slack table and uses it on-line to allow the execution of accepted aperiodic tasks. The algorithm has been proved to be optimal in terms of providing minimal response times to the accepted aperiodic tasks. However, this procedure does not provide any mechanism for tolerating faults. Most of the above discussed scheduling approaches often lack the required predictability in guaranteeing the reliability as demanded by many safety-critical applications. So, formal off-line scheduling is often the preferred option for ensuring both predictability of the worst-case behavior as well as high resource utilization. We have qualitatively compared our scheme with other methods that can provide fault-tolerance to aperiodic tasks and the summary is presented in Table 4.1.

Table 4.1: *Comparison with other scheduling approaches*

Method	**Pre emptive**	**Dynamic task arrivals**	**Multi processor**	**Multiple Faults**	**Optimal / Heuristic**	**Scalability**
[86]	No	Yes	Yes	No	Optimal	Exponential
[48]	No	Yes	Yes	Yes	Heuristic	Polynomial
[72]	Yes	No	No	Yes	Optimal	Polynomial
[71]	Yes	No	Yes	No	Heuristic	Polynomial
Proposed Scheme	Yes	Yes	Yes	Yes	Optimal	**Exponential. Handled by BDDs**

4.2 Proposed Scheme

This section presents the proposed *Supervisory Control of Timed Discrete Event Systems* (SCTDES) based scheduler synthesis scheme. The principal steps are as follows: (i) Development of the task execution as well as deadline specification models for each task in the system. (ii) Computation of the composite task execution model using product composition of individual task execution models. (iii) Computation of the composite deadline specification model using product composition of individual deadline specification models. (iv) Computation of the supervisor using product composition of the composite task execution and deadline specification models. (v) Checking the controllability of the resulting supervisor with respect to system behavior and uncontrollable events.

4.2.1 Assumptions on System Model

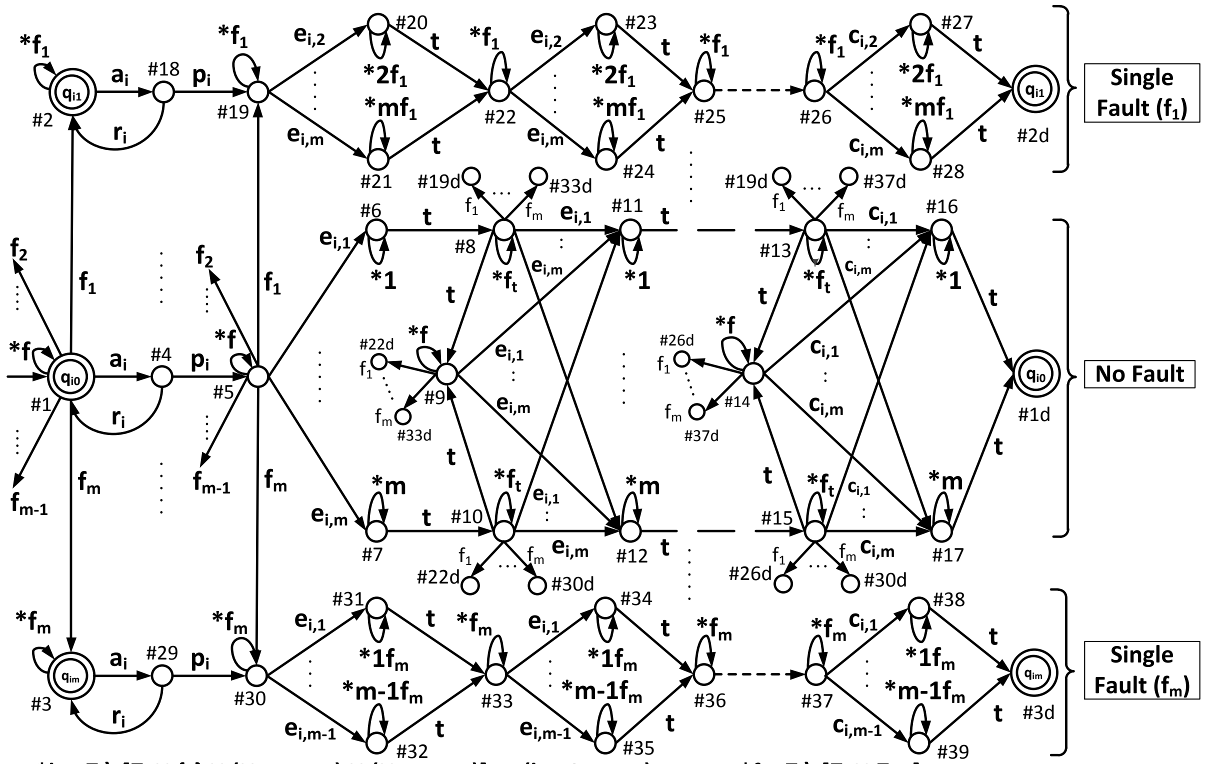
We consider a RT multiprocessor system consisting of a set I $(= \{\tau_1, \tau_2, ..., \tau_n\})$ of n aperiodic tasks to be scheduled on a set V $(= \{V_1, V_2, ..., V_m\})$ of m identical processors. In addition, there is also a separate processor to execute the scheduler/supervisor. Each task τ_i is characterized by a 3-tuple $\langle A_i, E_i, D_i \rangle$, where A_i is the *arrival time*, E_i is the *execution time* and D_i is the *relative deadline.*

Assumptions about the system: (1) All tasks are independent with no precedence constraints. (2) All instances of an aperiodic task have the same execution time (E_i) and deadline (D_i) requirements (such that $E_i \leq D_i$) which are known a priori. (3) All dynamically arrived tasks are reported to the scheduler which then controls the admission and execution of the tasks on the available processors according to the pre-computed off-line scheduling policy. (4) At most one processor may fail and the scheduling processor monitors (observes) processor faults. Task which was executing on a processor just before its fault will be restarted from the beginning in another non-faulty processor. (5) The scheduling processor never fails. (6) The scheduler takes negligible time to take the mutually exclusive decision of acceptance / rejection of a newly arrived task. These acceptance and rejection events are exclusively schedule enabled. (7) Tasks are allowed to migrate from one processor to another and the migration overhead is also negligible. (8) A task assigned to a particular processor must execute for at least one tick event on it before being preempted.

4.2.2 A TDES model of task execution under single fault

The TDES model for executing an aperiodic task τ_i is as follows:

$$T_i = (Q_i,\ \Sigma,\ \delta_i,\ q_{i0},\ \{q_{i0}, q_{i1}, ..., q_{im}\})$$



$*k = \Sigma \setminus [\Sigma_i \cup \{t\} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k})]$ $(k = 1, \ldots, m)$ $*f = \Sigma \setminus [\Sigma_i \cup \Sigma_{fau}]$

$*f_k = \Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k})]$ $(k = 1, \ldots, m)$ $*f_t = \Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup \{t\}]$

$*xf_k = \Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup \{t\} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k}) \cup (\cup_{j=1\ldots n} e_{j,x}) \cup (\cup_{j=1\ldots n} c_{j,x})]$ $(k, x = 1, \ldots, m)$

Figure 4.1: *The execution model T_i of a task τ_i under single fault.*

Here, $\Sigma = \cup_{i \in \{1,2,...,n\}} \Sigma_i \cup \Sigma_{fau} \cup \{t\}$, where $\Sigma_{fau} = \{f_1, ..., f_m\}$, and $\Sigma_i = \{a_i, p_i, r_i, e_{i,1}, ..., e_{i,m}, c_{i,1}, ..., c_{i,m}\}$. The events are described in Table 4.2 and they are categorized as follows: (i) $\Sigma_{uc} = \cup_{i \in \{1,2,...,n\}} \{a_i\} \cup \Sigma_{fau}$ (since the arrival events of tasks and the fault of a processor are induced by the environment, they are modeled as *uncontrollable* events), (ii) $\Sigma_c = \Sigma \setminus (\Sigma_{uc} \cup \{t\})$ (apart from Σ_{uc} and *tick* event (t), remaining events in the system are modeled as *controllable* events), (iii) $\Sigma_{for} = \Sigma_c$ (controllable events are also modeled as *forcible* events which can preempt the *tick* event (t)). Suppose E_i is equal to 1, that is each instance of τ_i requires only a single segment of execution, then $\Sigma_i = \{a_i, p_i, r_i, c_{i,1}, ..., c_{i,m}\}$. All events in Σ are considered to be *observable*.

Table 4.2: *Description of events (for fault-tolerant preemptive execution)*

Event	Description
a_i	Arrival of a task τ_i
p_i	Acceptance of a task τ_i
r_i	Rejection of a task τ_i
$e_{i,j}$	Execution of a segment of τ_i on a processor V_j
$c_{i,j}$	Execution of the last segment of τ_i on a processor V_j
f_j	Fault of a processor V_j

The execution model T_i for the task τ_i is shown in Figure 4.1. Labels have not been specified for all the states shown in the figure in order to reduce its size. However, states are numbered sequentially starting from #1 and these numbers will be used as references while explaining the execution model. In addition to this, a few states have been duplicated to avoid cluttering. A duplicated state, say #A, is denoted by #Ad in the figure. For example, transition f_m from State #8 actually goes to State #33, but it is directed to its duplicated State #33d to avoid overlapping with other transitions in the figure. q_{i0} (State #1) is the initial state of T_i and represents the state in which task τ_i resides prior to its arrival under a non-faulty system scenario. q_{i0} is also a marked state that represents the completion of the execution of task τ_i. The set of events related to self-loop transition $^*f$ ($= \Sigma \setminus [\Sigma_i \cup \Sigma_{fau}]$) at q_{i0} are described as follows:

1. Since task τ_i has not yet arrived, the events associated with τ_i (Σ_i) are excluded from $^*f$.

2. Since q_{i0} represents the non-faulty scenario, the processor fault events (Σ_{fau}) are also excluded from $^*f$.

3. After excluding Σ_i and Σ_{fau} from Σ, $^*f$ contains the events such as arrival, rejection, acceptance, execution, and completion of any other task ($\tau_j \in I, j \neq i$) in the system. Thus, $^*f$ does not impose any restriction on the execution of other tasks.

4. In addition, $^*f$ also contains the *tick* event in it which is used to model the arbitrary arrival time of the task τ_i, e.g., a_i (τ_i arrives at system start time), ta_i (after one tick event), tta_i (after two tick events), ... $\in L(T_i)$.

After the occurrence of an arrival event a_i, T_i reaches State #4 from the initial state q_{i0} (State #1). Now, the scheduler takes the mutually exclusive decision of whether to accept (p_i) or reject (r_i) the task τ_i. If the scheduler enables the rejection event r_i (and disables p_i), then T_i goes back to the initial state from State #4 and continues to stay in that state until either the occurrence of the next release (or arrival) event (a_i) or fault of anyone of the processors. If the scheduler enables the acceptance event p_i (and disables r_i) at State #4, then T_i reaches State #5. Here, the scheduler takes a decision whether to immediately allocate the task τ_i for execution on a processor or make it wait on the ready queue. The latter is indicated by the self-loop transition $^*f$ at State #5. If the scheduler decides to execute the task τ_i, then it will enable the event $e_{i,k}$ to assign τ_i to anyone of the available processors V_k ($V_k \in V, k \in \{1, 2, ..., m\}$) for execution. According to Figure 4.1, if event $e_{i,1}$ occurs at State #5, then T_i reaches State #6, i.e., task τ_i is assigned to processor V_1. At State #6, self-loop transition ^{*1} ($= \Sigma \setminus [\Sigma_i \cup \{t\} \cup (\cup_{j=1,\ldots,n} e_{j,1}) \cup (\cup_{j=1,\ldots,n} c_{j,1})]$) models the following three scenarios:

1. After assigning τ_i on V_1 ($e_{i,1}$), τ_i will not be allowed to execute any event associated with it. This is modeled by excluding the events Σ_i from ^{*1}.

2. τ_i is allowed to stay at State #6 by executing events in ^{*1} until the occurrence of the next *tick* event. This is modeled by excluding the *tick* event from ^{*1}.

3. After assigning τ_i on V_1, no other task ($\tau_j \in I, j \neq i$) will be allowed to execute on processor V_1. This is modeled by excluding the events $[(\cup_{j=1,\ldots,n} e_{j,1}) \cup (\cup_{j=1,\ldots,n} c_{j,1})]$ from ^{*1}. It ensures that *task* τ_i *cannot be preempted by another task* τ_j ($\tau_i \neq \tau_j$) for at least one *tick* event, thus enforcing Assumption 8. As $e_{i,j} \in \Sigma_i$, the task τ_i is also restricted from re-executing the event $e_{i,1}$ until the next *tick* event.

After τ_i completes its execution ($e_{i,1}$) on processor V_1 for one time unit t, i.e., T_i reaches State #8 from State #6, the scheduler takes one of the **three decisions**:

1. τ_i is allowed to continue execution on the same processor V_1 for one more time unit, indicated by the path $a_i\ p_i\ e_{i,1}\ t\ e_{i,1}\ t$ from q_{i0}. This takes T_i to State #11.

2. τ_i is migrated from processor V_1 to another processor V_m ($1 \neq m$) for execution. This is indicated by the path $a_i\ p_i\ e_{i,1}\ t\ e_{i,m}\ t$ from the initial state which takes T_i to State #12. This obviously necessitates a preemption of τ_i on V_1.

3. τ_i is moved to the ready queue being preempted by another task τ_j ($\tau_j \neq \tau_i$). This implies that τ_j is allocated to processor V_1 (captured by the execution model T_j corresponding to the task τ_j). In this case, T_i remains in State #8 by executing the event $e_{j,1}$ in self-loop $^*f_t (= \Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup \{t\}])$ that captures all events that are part of $^*f$ except the *tick* event.

In case of Decision 3, a *tick* event takes T_i from State #8 to State #9. Now, τ_i is again eligible for execution on any of the available processors in future time ticks (using Decision 2). This is depicted by the outgoing transitions $\{e_{i,1}, ..., e_{i,m}\}$ at State #9.
Note: For a non-preemptive scheduling system [86], the scheduler always takes Decision 1, unless there is a processor fault. In this paper, Decision 2 and Decision 3 are included to allow task migrations (Assumption 7) and preemptions (Assumption 8), thus providing a more flexible scheduling system with higher resource utilization compared to non-preemptive systems.

Now, we move our discussion from the *No Fault* part to the *Single Fault* part of the TDES model T_i. Since there are m processors in V, we have m different *Single Fault* transition structures corresponding to the fault of each processor as shown in Figure 4.1. T_i stays within the *No Fault* structure as long as no processor in the system is affected by a fault. On the occurrence of a processor fault event ($\{f_1, ..., f_m\}$), T_i moves and continues to stay in the *Single Fault* structure corresponding to the faulty processor. For example, if T_i is at State #1, then a fault of the processor V_1 will take T_i to State #2 through the transition f_1. However, there are no fault events defined at State #4 of T_i. Since the scheduler takes negligible time (according to Assumption 6) to decide on the acceptance / rejection of τ_i, no time has elapsed between State #1 and State #5. Because fault events for the current time tick have already been handled at State #1, they have not been defined at State #4. However, the active event set of State #5 includes the *tick* event (in $^*f$). Therefore, fault events are defined at State #5 and those fault events are similar to State #1. The argument for the absence of fault transitions at State #4 holds at State #6 also.

To explain the *Single Fault* structure, we consider the fault of processor V_1 by assuming that T_i is at State #2 (i.e., τ_i has not yet arrived and V_1 is affected by a fault). At State #2, the self-loop labeled $^*f_1$ has the following semantics: $\Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup (\cup_{j=1,...,n} e_{j,1}) \cup (\cup_{j=1,...,n} c_{j,1})]$. In addition to capturing the scenarios mentioned for $^*f$, $^*f_1$ imposes the additional restriction that the no task will be allowed to execute on the faulty processor V_1 (by excluding the events $[(\cup_{j=1,...,n} e_{j,1}) \cup (\cup_{j=1,...,n} c_{j,1})]$). After the arrival of τ_i, T_i will move from State #2 to State #18 through the transition a_i. If τ_i is accepted by the scheduler, then T_i will move to State #19 through the transition p_i. Since, V_1 is already affected by the fault, State #19 considers only $m-1$ processors from V_2 to V_m which can be observed from the outgoing transitions on $\{e_{i,2}, ..., e_{i,m}\}$. If τ_i is assigned on to processor V_2, then T_i will move to State #20 through the transition $e_{i,2}$. At State #20, the self-loop labeled $^*2f_1$ has the following semantics: $\Sigma \setminus [\Sigma_i \cup \Sigma_{fau} \cup (\cup_{j=1,...,n} e_{j,1}) \cup (\cup_{j=1,...,n} c_{j,1}) \cup (\cup_{j=1,...,n} e_{j,2}) \cup (\cup_{j=1,...,n} c_{j,2})]$. In addition to capturing the scenarios mentioned for $^*f_1$ above, $^*2f_1$ imposes the additional restriction that the task τ_i allocated to V_2 cannot be preempted by any other task as well as τ_i is restricted to re-execute the event $e_{i,2}$ until the next *tick* event by excluding the events $[(\cup_{j=1,...,n} e_{j,2}) \cup (\cup_{j=1,...,n} c_{j,2})]$ which is similar to the self-loop ^{*1} at State #8. After the elapse of one tick event, T_i reaches State #22 where the scheduler will take anyone of the three possible decisions mentioned above and continue with the execution until it completes by reaching the marked State #2d. Similarly, we can analyze the *Single Fault* structure for other processor faults.

Now, we consider three different situations in which T_i will transit from the *No Fault* to *Single Fault* structure:

1. Let us assume that T_i is at State #8. If processor V_1 fails during the execution of task τ_i, then τ_i must be *restarted* from the beginning on any of the other non-faulty processors $\{V_2, V_3, ..., V_m\}$. Hence, T_i moves to State #19d (actually to State #19) from State #8 through the transition f_1.

2. It may happen that when τ_i is executing on V_1, some other processor $V_k (k \neq 1)$ may be affected by the fault $(f_k, k \neq 1)$. For example, when T_i is in State #8 after τ_i is being executed on V_1 during the last tick event, if processor V_m $(m \neq 1)$ is affected by the fault (event f_m occurs), then T_i must transit to State #33d.

3. In addition to the tasks which are executed on non-faulty processors, tasks which are in the ready queue must also transit to an appropriate state depending on which processor was affected by the fault at the last *tick* event. For example, when T_i is in State #9, if processor V_1 fails (event f_1 occurs), then T_i must transit to State #22d to capture this information. Otherwise, τ_i may be wrongly assigned to the faulty processor V_1 since at State #9 all processors are assumed to be non-faulty.

[Note:] T_i contains five different types of self-loops($^*f$, $^*f_t$, $^*k$ $(k = 1, ..., m)$, $^*f_k$ $(k = 1, ..., m)$, $^*xf_k$ $(k, x = 1, ..., m)$). As the number of task arrivals, task-to-processor assignments and the number of uncontrollable events between any two time ticks are always finite, a *tick* transition cannot ever be indefinitely preempted by the repeated execution of non-tick events in any of these self-loops. So, T_i is activity-loop-free.

Based on the above approach, we construct the TDES models T_1, T_2, ..., T_n for all the n tasks in the system. A product composition $T = T_1||T_2||...||T_n$ generates the composite model for all the tasks executing concurrently. As discussed earlier, T includes both deadline-meeting as well as deadline-missing execution sequences.

4.2.3 Problem formulation for scheduler design

Before presenting the scheduler design problem, we define three related notions, viz. *Instantaneous load*, *Deadline-meeting sequences* and *Fault-tolerant specification.*

Definition: *Instantaneous Load* ($\rho(t)$) [23]: It estimates the maximum load within a pre-specified time interval in a RT task system where dynamic task arrivals are allowed. Let the system contain n active tasks $I = \{\tau_1, \tau_2, ..., \tau_n\}$ with each task having currently remaining execution requirement $RE_i(t)$ and relative deadline D_i at time t. Without loss of generality, let $D_1 \leq D_2 \leq ... \leq D_n$. Now, the partial load $\rho_i(t)$ within any interval $[t, D_i]$ is given by: $\rho_i(t) = \frac{\sum_{k=1}^{i} RE_k(t)}{(D_i - t)}$. Given the partial load values $\rho_i(t)$ for all the intervals $[t, D_1]$, $[t, D_2], ...,$ $[t, D_n]$, the instantaneous load $\rho(t)$ is defined as the maximum over all the partial loads $\rho_i(t)$: $\rho(t) = \max_i\{\rho_i(t)\}$. □

Definition: *Deadline-meeting sequence for a task* τ_i: Let $s = s_1 a_i p_i s_2 c_{i,j} s_3$, such that $s_1, s_3 \in \Sigma^*$ and $s_2 \in (\Sigma \setminus \{c_{i,j}\})^*$. Then, a task τ_i is said to be deadline-meeting with respect to the sequence $s \in L_m(T)$ if the number of *tick* events in s_2 is less than or equal to $D_i - 1$ for any $a_i, p_i, c_{i,j}$ in s. Otherwise, s is deadline-missing for task τ_i. □

Given a sequence $s \in L_m(T)$ containing a set of tasks I, if the sequence is deadline-meeting for all the tasks, s is called as *deadline-meeting sequence for the task set* I. It is obvious that for a given deadline-meeting sequence, the condition $\rho(t) \leq m$ (for normal mode of operation) / $\rho(t) \leq (m-1)$ (subsequent to a processor fault) must hold at all instants in the schedule corresponding to the sequence. Hence, the task set I is *schedulable* if there exists a deadline-meeting sequence for I.

Definition: *Fault-tolerant Specification:* A language $K \subseteq L_m(T)$ is fault-tolerant and schedulable if K includes only deadline-meeting sequences for a task set I and it is controllable with respect to $L(T)$ and Σ_{uc}. □

The scheduler design problem may therefore be formulated as follows:

Find the largest fault-tolerant and schedulable language that contains all possible deadline-meeting sequences in T. If we find such a language K, then we can design a non-blocking *scheduler (or supervisor) S such that $L(S/T) = \overline{K}$.*

4.2.4 A TDES model of deadline specification

Once activated, a task τ_i has to finish its execution within its relative deadline. Such a specification can be modeled using TDES by allowing τ_i to execute the events from the following sets: (1) $\{e_{i,1}, ..., e_{i,m}\}$ for $E_i - 1$ ticks after its acceptance (subsequent to its arrival) (2) $\{e_{i,1}, ..., e_{i,m}, c_{i,1}, ..., c_{i,m}\}$ from the E_i^{th} tick to the $(D_i - 1)^{th}$ tick. The TDES model H_i shown in Figure 4.2 models the deadline specification of τ_i.

The self-loop transition $\Sigma \backslash \Sigma_i$ at State #1 excludes the events (Σ_i) that are associated with τ_i since it has not yet arrived. However, it does not restrict the events that are happening with respect to other tasks in the system. Since $\Sigma \setminus \Sigma_i$ also contains the *tick* event, it models the arbitrary arrival time of a task τ_i. After the occurrence of an arrival event a_i, H_i reaches State #2. If the execution time E_i of a task τ_i is greater than 1, only then will Σ_i contain $\{e_{i,1}, ..., e_{i,m}\}$. Suppose $E_i > 1$, self-loop transition * $(= \Sigma \setminus \{t, a_i, r_i, c_{i,1}, c_{i,2}, ..., c_{i,m}\})$ in State #2 is used to model the fact that the task τ_i is allowed to take only the events from the set $\{p_i, e_{i,1}, e_{i,2}, ..., e_{i,m}\}$ associated with it and without imposing any restriction with respect to other tasks in the system. After the elapse of one tick event, H_i reaches State #3 in which the self-loop is similar to that in State #2. Since the task τ_i is not allowed to execute events from the set $^*c$ $(= \{c_{i,1}, c_{i,2}, ..., c_{i,m}\})$ before the elapse of $E_i - 1$ tick events (because at least $E_i - 1$ time ticks must be incurred before executing the last segment of τ_i even if it executes uninterruptedly after its acceptance), states that are similar to State #2 and State #3 are instantiated $E_i - 1$ times starting from State #2. Following that, at State #5, task τ_i is allowed to execute events from the set $\{c_{i,1}, c_{i,2}, ..., c_{i,m}\}$ in addition to $\{p_i, e_{i,1}, e_{i,2}, ..., e_{i,m}\}$ because τ_i is allowed to complete its execution after executing for $E_i - 1$ ticks from its acceptance. Since τ_i may postpone the execution of its last segment at most up to $D_i - 1$ tick events, states that are similar to State #5 are instantiated from E_i to $D_i - 1$ tick events measured from State #2. After the execution of an event from the set $\{c_{i,1}, c_{i,2}, ..., c_{i,m}\}$ at anyone of the states from State #5 to State #8, H_i reaches State #9 in which there are no events associated with τ_i and this is modeled by the self-loop transition $\Sigma \setminus (\Sigma_i \cup \{t\})$. After the elapse of a tick event, H_i reaches State #1. From Figure 4.2, it can be observed that, $L(H_i)$ contains all possible execution sequences of τ_i that meets the deadline D_i. Based on this approach, H_i is constructed

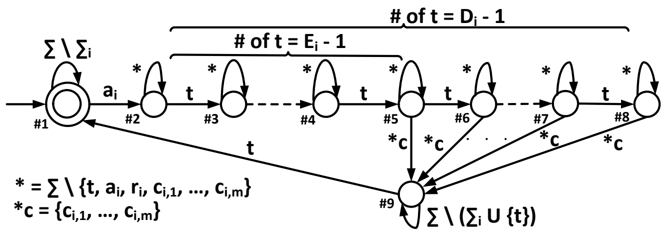


Figure 4.2: *A deadline specification model H_i for a task τ_i [86].*

for all other tasks in the system. Then we perform the product composition to obtain the composite model $H = H_1||H_2||...||H_n$. This composite specification model H includes all sequences that meet the deadlines of concurrently executing aperiodic tasks. However, H neither restricts multiple tasks from being assigned to a particular processor at the same instant, nor does it disallow the execution of tasks on a faulty processor.

In order to find all sequences that meet the deadlines of accepted tasks from the sequences of $L_m(T)$, we construct the following finite state automaton $M = T||H$.

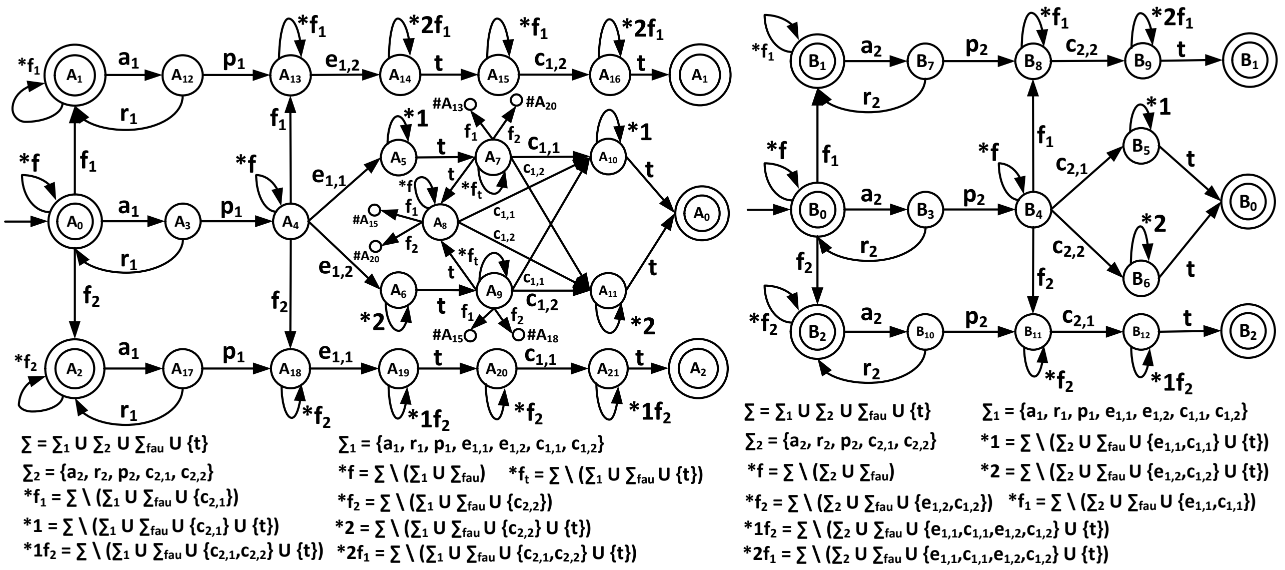
Proposition 4.2.1. *$L_m(M)$ contains only and all the deadline-meeting sequences of $L_m(T)$.*

Proof. We know that $L_m(T)$ contains both deadline-meeting and deadline-missing sequences of all tasks accepted by the scheduler. On the other hand, $L_m(H)$ contains all possible deadline-meeting sequences. Since $L_m(M) = L_m(T) \cap L_m(H)$, we can conclude that $L_m(M)$ contains *only* and *all* the deadline-meeting sequences of $L_m(T)$. $\square$

Here we illustrate the concepts discussed till now using an example. Let us consider a RT multiprocessor system composed of two identical processors $V = \{V_1, V_2\}$ executing two aperiodic tasks $I = \{\tau_1, \tau_2\}$ with arbitrary arrival times having execution requirements $E_1 = 2$, $E_2 = 1$ and deadlines $D_1 = 3$, $D_2 = 2$. Task execution models T_1 for τ_1 and T_2 for τ_2 are shown in Figure 4.3a[1] and Figure 4.3b[2], respectively. Since the execution time of task τ_2 is 1, Σ_2 does not contain the events $\{e_{2,1}, e_{2,2}\}$ in it. Deadline

[1]States in T_1 are assigned with the labels of the form A_j instead of q_{1j} to reduce the figure size, i.e., q_{1j} is represented by A_j.

[2]q_{2j} is represented by B_j to reduce the figure size.


Σ = Σ1 U Σ2 U Σfau U {t}
Σ2 = {a2, r2, p2, c2,1, c2,2}
*f1 = Σ \ (Σ1 U Σfau U {c2,1})
*1 = Σ \ (Σ1 U Σfau U {c2,1} U {t})
*1f2 = Σ \ (Σ1 U Σfau U {c2,1,c2,2} U {t})
Σ1 = {a1, r1, p1, e1,1, e1,2, c1,1, c1,2}
*f = Σ \ (Σ1 U Σfau)
*ft = Σ \ (Σ1 U Σfau U {t})
*f2 = Σ \ (Σ1 U Σfau U {c2,2})
*2 = Σ \ (Σ1 U Σfau U {c2,2} U {t})
*2f1 = Σ \ (Σ1 U Σfau U {c2,1,c2,2} U {t})
Σ = Σ1 U Σ2 U Σfau U {t}
Σ2 = {a2, r2, p2, c2,1, c2,2}
*f = Σ \ (Σ2 U Σfau)
*f2 = Σ \ (Σ2 U Σfau U {e1,2,c1,2})
*1f2 = Σ \ (Σ2 U Σfau U {e1,1,c1,1,e1,2,c1,2} U {t})
*2f1 = Σ \ (Σ2 U Σfau U {e1,1,c1,1,e1,2,c1,2} U {t})
Σ1 = {a1, r1, p1, e1,1, e1,2, c1,1, c1,2}
*1 = Σ \ (Σ2 U Σfau U {e1,1,c1,1} U {t})
*2 = Σ \ (Σ2 U Σfau U {e1,2,c1,2} U {t})
*f1 = Σ \ (Σ2 U Σfau U {e1,1,c1,1})

(a) *TDES of* T_1 **(b)** *TDES of* T_2

Figure 4.3: *TDES models* T_1 *and* T_2

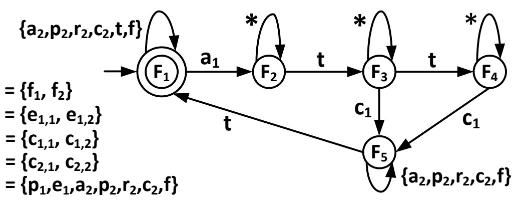


Figure 4.4: *TDES of* H_1

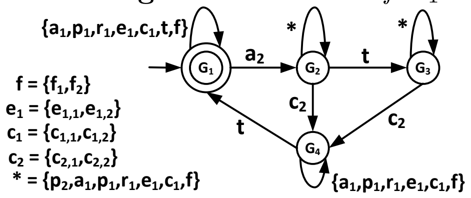


Figure 4.5: *TDES of* H_2

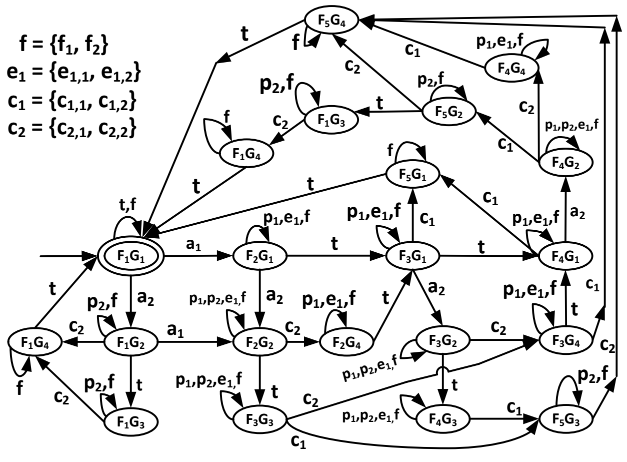


Figure 4.6: $H = H_1||H_2$

specification models H_1 for τ_1 and H_2 for τ_2 have been presented in Figure 4.4[1] and Figure 4.5, respectively. Then, we perform the product composition of T_1 and T_2 to obtain the composed model T $(= T_1||T_2)$. Due to space limitation, only a partial diagram of this composed model has been shown in Figure 4.7. However, this figure contains both deadline-meeting as well as deadline-missing sequences for I.

In Figure 4.7, consider the following sequence from the initial state (A_0B_0) of T: $s_1 = a_1p_1f_1a_2p_2e_{1,2}tc_{1,2}tc_{2,2}t$. After the arrival (a_1) and acceptance (p_1) of the task τ_1, processor V_1 fails (f_1). At the same time, task τ_2 arrives (a_2) and is accepted (p_2) by the scheduler. Then, the scheduler decides to execute task τ_1 on V_2 $(e_{1,2})$ and continues to execute the task τ_1 to completion $(c_{1,2})$ non-preemptively before allowing task τ_2 to execute. As a consequence, task τ_2 misses its deadline. This is because, the number of *tick* events in the substring $a_2p_2e_{1,2}tc_{1,2}tc_{2,2}$ of s_1 is 2 which is greater than $D_2 - 1$ $(= 2 - 1 = 1)$. Hence, s_1 is a deadline-missing sequence of I. On the other hand, consider another sequence from the initial state (A_0B_0): $s_2 = a_1p_1f_1a_2p_2e_{1,2}tc_{2,2}tc_{1,2}t$. Here, the scheduler has taken the decision to preempt the task τ_1 after executing it for one unit of time and schedules task τ_2 on V_2 $(c_{2,2})$ before completing the task τ_1. So, both the tasks will meet their deadlines. With reference to Definition 2, the number of *tick* events in the substring $a_1p_1f_1a_2p_2e_{1,2}tc_{2,2}tc_{1,2}$ of s_2 is 2 which is equal to $D_1 - 1$ $(= 3 - 1 = 2)$. Similarly, the number of *tick* events in the substring $a_2p_2e_{1,2}tc_{2,2}$ of s_2 is 1 which is equal to $D_2 - 1$ $(= 2 - 1 = 1)$. Hence, s_2 becomes a deadline-meeting sequence of I. However, $L_m(T)$ contains both the sequences (s_1 and s_2) in it.

Figure 4.6 shows the product composed model (H) of the deadline specifications H_1 and H_2. As mentioned earlier, this model represents all possible deadline meeting sequences of the task set I. From Figure 4.6, it may be observed that sequence s_2 may be retrieved by tracing the states F_1G_1, $(F_2G_1)^3$, $(F_2G_2)^3$, F_3G_3, F_3G_4, F_4G_1, F_5G_1, F_1G_1. Following a similar approach for s_1, we see that after proceeding through the states F_1G_1, $(F_2G_1)^3$, $(F_2G_2)^3$, F_3G_3, F_5G_3, the composite model H gets blocked due to the absence of a transition on a tick event (t) at State F_5G_3. More specifically, after

[1] Short notations f, e_1, c_1, c_2 have been used in Figure 4.4, Figure 4.5, and Figure 4.6 to reduce the figure size.

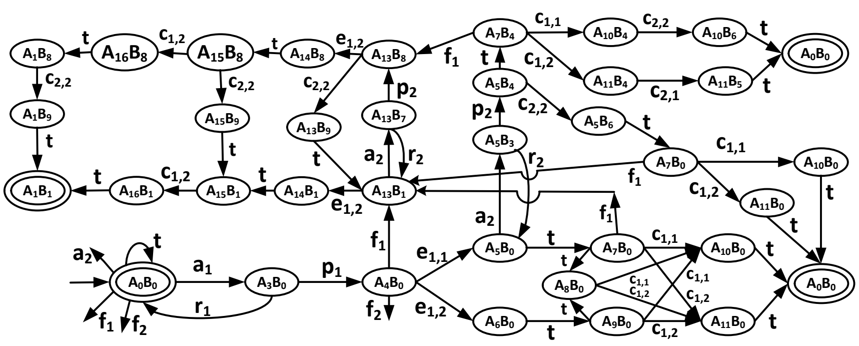


Figure 4.7: $T = T_1||T_2$ *(partial diagram)*

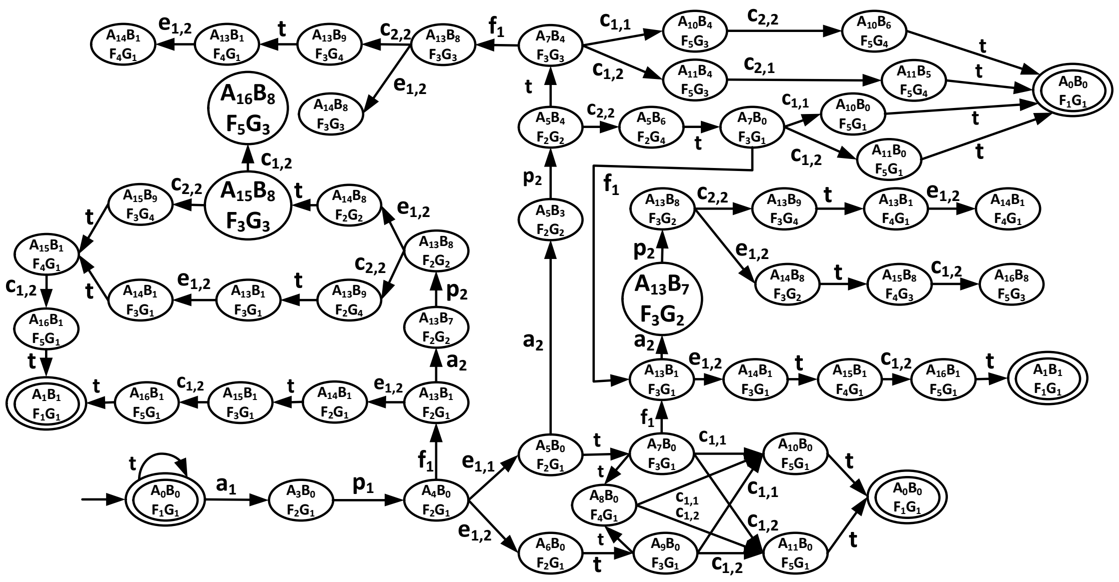


Figure 4.8: $M = T||H$ *(partial diagram)*

processing the sub-string $a_2p_2e_{1,2}tc_{1,2}$ of s_1, the next event in s_1 is t. However, t is not present at State F_5G_3. Hence, $L_m(H)$ contains the deadline-meeting sequence s_2 and does not contain the deadline-missing sequence s_1.

Figure 4.8 depicts the model M resulting from the product composition of T and H. Here, M does not contain rejection events (r_i) that are present in T. This is due to the fact that H is devoid of r_i, and hence, r_i gets eliminated from M during the production composition of T and H. Therefore, M always accepts all tasks irrespective of the instantaneous load condition. From Figure 4.8, it can be observed that the deadline-missing sequence s_1 is not part of M ($s_1 \notin L_m(M)$) as the suffix $tc_{2,2}t$ in s_1 gets eliminated from M during the composition $T||H$, i.e., there are no outgoing transitions from State $A_{16}B_8F_5G_3$ which represents a *deadlock*. Considering sequences s_1 and s_2 on M (Figure 4.8), it may be seen that the automaton reaches State $A_{15}B_8F_3G_3$ after executing the common prefix $a_1p_1f_1a_2p_2e_{1,2}t$ of s_1 and s_2. At this state, the scheduler will disable the controllable event $c_{1,2}$ and enable $c_{2,2}$ from the active event set using its control action which in turn ensures that the system T reaches State $A_{15}B_9$ (in Figure 4.7) and not into State $A_{16}B_8$. Subsequently, by following the set of events enabled by the scheduler, the system T will reach State A_1B_1 which represents the successful completion (meeting the deadline) of both the accepted tasks. Hence, $L_m(M)$ does not contain the deadline-missing sequence s_1.

4.2.5 Controllability of $L_m(M)$ w.r.t. $L(T)$ and Σ_{uc}

To ensure that all the accepted tasks meet their individual deadlines, a scheduler should be designed to achieve $L_m(M)$, i.e., the scheduler must be able to avoid reaching any non-co-accessible state of M which may lead to deadlock states (deadline-miss). For this, it is necessary that $L_m(M)$ must be controllable with respect to $L(T)$ and Σ_{uc}. That is, after executing any arbitrary prefix s in $\overline{L_m(M)}$, the next event σ must always be allowed ($s\sigma \in \overline{L_m(M)}$) if (i) $\sigma \in \Sigma_{uc}$ or, (ii) $\sigma = t$ and no forcible events are eligible. According to Proposition 4.2.1, $L_m(M)$ contains only and all the deadline-meeting sequences of $L_m(T)$. However, if $s \in \overline{L_m(M)}$ is extended by $\sigma \in \Sigma_{uc}$, there are possibilities for $s\sigma$ to become deadline-missing. This has been proved in the following proposition by discussing one scenario for each type of uncontrollable event. Later, we have presented a methodology to modify M such that the modified model will be able to control the execution of accepted tasks to meet their deadlines even in the presence of uncontrollable events.

Proposition 4.2.2. $L_m(M)$ is not controllable w.r.t. $L(T)$ and Σ_{uc}.

Proof. Let us consider any arbitrary sequence $s = s_1a_ip_is_2c_{i,j}s_3 \in \overline{L_m(M)}$, where $s_1, s_3 \in \Sigma^*$ and $s_2 \in (\Sigma \setminus \{c_{i,j}\})^*$. Since, $s \in \overline{L_m(M)}$, the number of *ticks* in s_2 must be less than D_i. Given the two types of uncontrollable events, there are two different cases: (1) $\sigma \in \{a_1, ..., a_n\}$, (2) $\sigma \in \{f_1, ..., f_m\}$.

***Case 1*:** Let us assume that x be the instant at which task τ_i is accepted (p_i) by the scheduler (in sequence s) and the instantaneous load in the interval $[x, x + D_i]$ is exactly equal to the available processing capacity, i.e., the system operates in full load condition after the acceptance of task τ_i. The sequence $s \in \overline{L_m(M)}$ still remains deadline-meeting as the system does not transit into overload. Now, let us consider another sequence s_y such that it is a prefix of s, i.e., $s_y = s_1 a_i p_i$. $s \in \overline{L_m(M)}$ implies $s_y \in \overline{L_m(M)}$. Let s_y be extended by the arrival ($a_j \in \Sigma_{uc}$) and acceptance (p_j) of another task τ_j ($\tau_i \neq \tau_j$). Let y be the instant at which task τ_j is accepted (p_j) by the scheduler. If $D_i \geq D_j$, then the instantaneous load in the interval $[y, y + D_j]$ becomes greater than the available processing capacity. This will lead to a situation where at least one of the accepted tasks is guaranteed to miss its deadline. Consequently, $s_y a_j p_j \notin \overline{L_m(M)}$ (in accordance with Proposition 4.2.1). From Assumption 6 and the transition structure of M which does not contain any rejection event, it can be inferred that a_j will always be extended by p_j. This implies that $s_y a_j$ is the only possible prefix for $s_y a_j p_j$. So, $s_y a_j \notin \overline{L_m(M)}$. However, $s_y a_j \in L(T)$. Hence, we conclude that $L_m(M)$ is *not controllable* with respect to uncontrollable arrival events, i.e., $s_y \in \overline{L_m(M)}$ and $a_j \in \Sigma_{uc}$ and $s_y a_j \in L(T)$ does not imply $s_y a_j \in \overline{L_m(M)}$.

***Case 2*:** Let us assume that $s = s_1 a_i p_i s_2 c_{i,j} s_3$ does not contain any processor fault event in it. Now, consider another sequence s_y such that $s_y = s_1 a_i p_i s_2$, a prefix of s. Here, $s \in \overline{L_m(M)}$ implies $s_y \in \overline{L_m(M)}$. If s_y is extended by a fault event ($f_j \in \Sigma_{uc}$) of a processor V_j, then the extended sequence becomes $s_y f_j$. If the instantaneous load at the instant $le(s_2)$ (i.e., last event of s_2) is greater than $(m - 1)$, then the processor fault event f_j immediately following it will lead to an overload situation and a deadline miss of at least one of the already accepted tasks cannot be avoided. Consequently, from Proposition 4.2.1, $s_y f_j \notin \overline{L_m(M)}$. However, $s_y f_j \in L(T)$. Hence, we conclude that $L_m(M)$ is *not controllable* with respect to uncontrollable processor fault event, i.e., $s_y \in \overline{L_m(M)}$ and $f_j \in \Sigma_{uc}$ and $s_y f_j \in L(T)$ does not imply $s_y f_j \in \overline{L_m(M)}$. □

Therefore, mechanisms must be designed in order to modify M such that the transformed model does not reach any state which may lead to a violation of controllability in the presence of uncontrollable events. Towards this objective, first we proceed with Case 1 ($\sigma \in \{a_1, ..., a_n\}$; refer proof of Proposition 4.2.2).

With reference to the TDES of M shown in Figure 4.8, it may be observed that M is not controllable because after following the sequence $s = a_1 p_1 e_{1,1} t f_1 a_2$ ($le(s) = a_2 \in \Sigma_{uc}$) the system reaches $A_{13}B_7F_3G_2$, a state where the instantaneous load $\rho(t)$ becomes greater than 1 ($\rho(t) = (2/2) + (1/2) = (3/2) > 1$) leading to a system overload. Therefore, a feasible schedule is impossible to obtain by continuing further from s. It is clear that the only way to circumvent this problem is to reject a task whenever its acceptance may lead to a subsequent overload. Hence, a new automaton M' must be designed that modifies the active event set and transition function of M such that appropriate rejections may be incorporated to handle overloads caused by arbitrary task arrivals. We now present a methodology to construct M' from M in Algorithm 6. The

ALGORITHM 6: M' CONSTRUCTION

```
Input: $M$, $n$ (number of tasks)
Output: $M'$
1  begin
2      foreach co-accessible state $q$ of $M$ do
3          for $i = 1$ to $n$ do
4              if $(\delta_M(q, a_i) \neq \emptyset)$ then
5                  Determine the sequence $s \in \overline{L_m(M)}$ such that $\delta_M(q_0, s) = q$;
6                  if $(sa_i \notin \overline{L_m(M)})$ then
7                      Find $q' = \delta_M(q, a_i)$; // $q'$ is a non-co-accessible state of $M$;
8                      $\delta_M(q', r_i) = q$; // Add $r_i$ from $q'$ to $q$;
9                      $\delta_M(q', p_i) = \emptyset$; // Delete $p_i$ from $M$;
10                     $M = Ac(M)$; // Take the accessible part of $M$ ;
11     Let the resultant automata obtained be denoted by $M'$;
```

above construction procedure removes the non-co-accessible (deadlock) states from M created by the acceptance of a task which increased the instantaneous load beyond the available system capacity. Therefore, M' will remain same as M, if the instantaneous load never crosses the available system capacity.

Figure 4.9 shows the modified automaton M' constructed from M (depicted in Figure 4.8) using Algorithm 6. It can be observed that the acceptance transition (p_2) coming out from State $A_{13}B_7F_3G_2$ (Figure 4.8) has been removed and a rejection transition (r_2) has been added from State $A_{13}B_7F_3G_2$ to State $A_{13}B_1F_3G_1$ in Figure 4.9. Also, the corresponding non-co-accessible part of M has been removed. If the system reaches State $A_{13}B_7F_3G_2$, then the task τ_2 will be rejected and the system goes back to State $A_{13}B_1F_3G_1$ which is co-accessible. Now, we need to check (i) Whether M' preserves all the deadline-meeting sequences of M, i.e., if $s \in L_m(M)$ implies $s \in L_m(M')$ (Proposition 4.2.3), (ii) Whether the new sequences in M' (not part of M), i.e., $s \in L_m(M') \setminus L_m(M)$, are deadline-meeting (Proposition 4.2.4).

Proposition 4.2.3. $L_m(M')$ contains all the deadline-meeting sequences of $L_m(M)$.

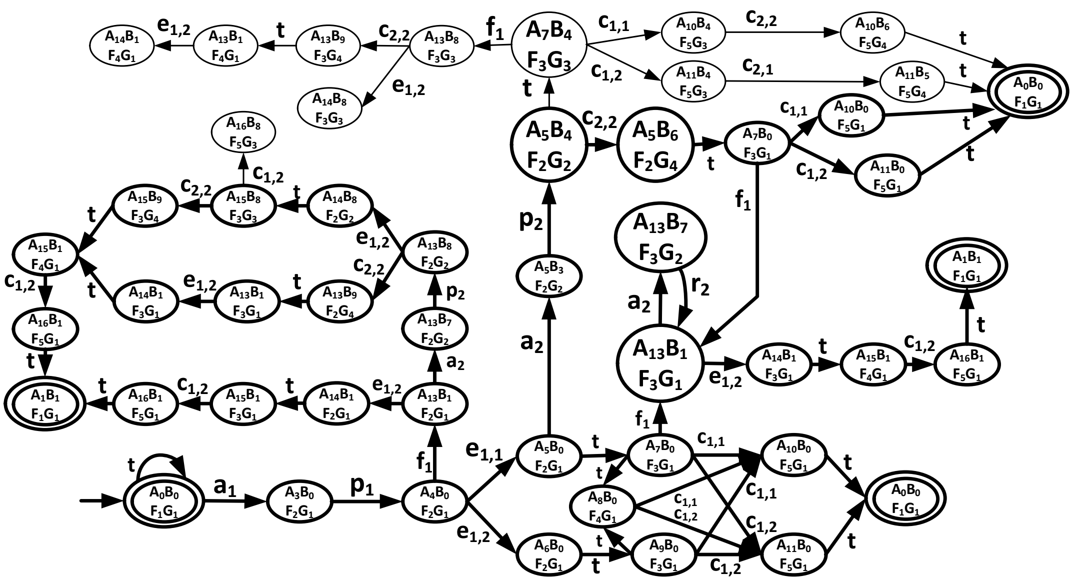


Figure 4.9: *TDES of M' (partial diagram)*

Proof. As the construction procedure of M' from M removes only the non-co-accessible part of M induced by uncontrollable arrival events, M' eliminates the sequences in M that may possibly terminate in a deadlock. Therefore, $L_m(M')$ contains *all* the deadline-meeting sequences of $L_m(M)$. □

Proposition 4.2.4. $L_m(M')$ contains only and all the deadline-meeting sequences of $L_m(M)$.

Proof. Let us consider a sequence $uv \in L_m(M)$ such that $u \in \overline{L_m(M)}$, $\delta_M(q_0, u) = q$, $\delta(q, a_i)$ is defined, $i \in \{1, 2, ..., n\}$ and $v \in \Sigma^*$. If $\delta_M(q, a_i) = q'$ and q' is a non-co-accessible state, then $ua_i \notin \overline{L_m(M)}$. Algorithm 6 transforms M to M' by adding the rejection transition from q' to q, i.e., $\delta_M(q', r_i) = q$ which makes q' co-accessible. Therefore $ua_ir_i \in \overline{L_m(M')}$ and $ua_i \in \overline{L_m(M')}$. Since, a_ir_i do not contain any *tick* event in it, the incorporation of sequence ua_ir_iv do not introduce any deadline-missing sequence in $L_m(M')$. In general, the sequences of the form $us_mv \in L_m(M') \setminus \overline{L_m(M)}$, where $s_m \in \{a_1r_1, a_2r_2, ..., a_nr_n\}$ makes q' to be co-accessible; hence, $ua_ir_i \in \overline{L_m(M')}$. Therefore, $L_m(M')$ contains only and all the deadline-meeting sequences of $L_m(M)$. □

From Proposition 4.2.1 and Proposition 4.2.3, it may be concluded that $L_m(M')$ contains *all* deadline-meeting sequences of $L_m(T)$. Consequently, from Proposition 4.2.4, it is implied that $L_m(M')$ contains *only* and *all* deadline-meeting sequences of $L_m(T)$.

Remark 4.2. *Condition for no task rejection:* The schedulability condition imposed through the construction of M' on all sequences is that the instantaneous load ($\rho(t)$) at each time instant is at most m (for normal mode of operation) / $(m-1)$ (subsequent to a processor fault). This implicitly guarantees that a sequence containing a task arrival whose acceptance may lead to a possible system overload at a subsequent time instant will not be part of $L_m(M')$. $\square$

Although arrival events that may lead to deadlock states in M has been removed from M', it may still contain deadlock states due to the presence of processor fault events. Now, we discuss about the controllability of $L_m(M')$ with respect to the processor fault events (Case 2: $\sigma \in \{f_1, ..., f_m\}$; refer proof of Proposition 4.2.2). Referring M' (Figure 4.9), it may be observed that if the system reaches State $A_7B_4F_3G_3$, then there is a possibility of missing the deadline of at least one of the accepted tasks due to the occurrence of an uncontrollable processor fault event (f_1) which leads to the situation $\rho(t) = (2/2) + (1/2) = (3/2) > 1$. So, in order to avoid the system from reaching a deadlock state due to the presence of a processor fault event, we need to compute a *maximally permissive* supervisor that restricts the system behavior within the *supremal controllable sub-language* of $L_m(M')$, denoted by $supC(L_m(M'))$. From M' (Figure 4.9), let us consider the sequence $s = a_1p_1e_{1,1}a_2p_2tf_1e_{1,2}$, where $s \in L(T)$ and $s \notin L_m(M')$, which implies that $s \in (L(T) - L_m(M'))$. The uncontrollable tail in s is $tf_1e_{1,2}$ which leads to a deadlock in M'. The prefix of the uncontrollable tail in s is given by $D_{uc}(s) = a_1p_1e_{1,1}a_2p_2t$. Hence, $supC(L_m(M'))$ does not include the sequences that have the prefix $a_1p_1e_{1,1}a_2p_2t$ in it. In a similar way, if we remove all the prefixes leading to an uncontrollable tail from $L_m(M')$, we can obtain $supC(L_m(M'))$. The portion of M' (Figure 4.9) shown in thick lines represents the $supC(L_m(M'))$. Now, if the system reaches the State $A_5B_4F_2G_2$, then the scheduler will enable the event set $\Sigma_{uc} \cup \{c_{2,2}\}$. In this situation, the forcible event $c_{2,2}$ will preempt the tick event to reach the State $A_5B_6F_2G_4$, which is co-accessible.

Theorem 4.2.1. *The language $supC(L_m(M'))$ is the largest fault-tolerant and schedulable language.*

Proof. We know that $L_m(M')$ contains only and all the deadline-meeting (schedulable) sequences of $L_m(T)$. However, $L_m(M')$ *is not always controllable in the presence of uncontrollable processor fault events.* Therefore, if $C(L_m(M'))$ denotes the set of the controllable sub-languages of $L_m(M')$, the language $supC(L_m(M'))$ ($\in C(L_m(M'))$) denoting the unique largest controllable sub-language in $C(L_m(M'))$, may be computed from it. $\square$

Proposition 4.2.5. $L_m(M')$ and $supC(L_m(M'))$ are always non-empty.

Proof. As the systems we consider in this work are composed of at least two processors ($m \geq 2$) and the computation demand of a single task cannot exceed the capacity of a single processor, at least one task will always be accepted by the scheduler even in the presence of a single permanent processor fault. Hence, $L_m(M')$ as well as $supC(L_m(M'))$ are always non-empty. $\square$

An Optimal (admissible) scheduler S can be designed as follows: For any $s \in \overline{supC(L_m(M'))}$, $S(s)$ denotes the set of events that are legitimate after observation of the sequence s (without restricting the occurrence of uncontrollable events). As a result of the supervision, we obtain $L(S/T) = \overline{supC(L_m(M'))}$. If S accepts a newly arrived task, a safe execution sequence will always be present (in $supC(L_m(M'))$) to meet its deadline even in the presence of a possible processor fault event.

4.3 Complexity Analysis & Symbolic Computation using BDD

This section first discusses the complexity analysis of the proposed scheme. The state-space complexity of T_i (Figure 4.1) is computed as follows: The initial state of T_i has $(m+1)$ branches emanating from it based on the events $\{f_1, f_2, ..., f_m\}$ representing the fault of anyone of the m processors, along with the fault-free branch (*No Fault*). Among them, the *No Fault* branch further contains m sub-branches on events $\{e_{i,1}, ..., e_{i,m}\}$ (emanating from State #5) depicting the possible execution of the first segment of task τ_i on anyone of the m available processors. With execution time E_i, each of these sub-branches (in the *No Fault* branch) will contain E_i states due to transitions on execution events and another E_i states due to transitions on *tick* events. Hence, the *No Fault* branch contains $\mathcal{O}(mE_i)$ states. On the other hand, a *Single Fault* branch contains $\mathcal{O}((m-1)E_i)$ states. This is because one of the m processors is affected by a fault and only $(m-1)$ processors are available for execution. Therefore, with m *Single Fault* branches, the overall state-space complexity of T_i becomes $\mathcal{O}(m^2E_i)$. The state-space complexity of H_i is $\mathcal{O}(D_i)$ because distinct states are used to count the occurrence of each tick starting from the arrival to the deadline of task τ_i. Given n TDESs T_1, T_2, ..., T_n, an upper bound for the number of states in the composite task execution model T is $\prod_{i=1}^{n} |Q^{T_i}|$, where $|Q^{T_i}|$ is the total number of states in T_i. Similarly, the total number of states in the composite specification model H is $\prod_{i=1}^{n} |Q^{H_i}|$. The state-space complexity of M, M' and $supC(L_m(M'))$ are $\mathcal{O}(|T| \times |H|)$. The time-complexity for computing T, H and M are exponential. However, the computation of M' and $supC(L_m(M'))$ are polynomial time as they involve simple graph traversal [19].

It may be observed that the number of states in the composite models T and H grows *exponentially* as the number of tasks increases. To address this issue, we present the steps for *Binary Decision Diagram (BDD)* [21] based symbolic computation of the supervisor in the following sub-section.

4.3.1 Symbolic representation of TDESs using BDDs

Before presenting the mechansim for transforming a TDESs into its symbolic *BDD* representation, we first introduce the notion of a *characteristic function* which is required to represent TDESs using BDDs.

Definition: *Characteristic Function:* Given a finite set U, each $u \in U$ is represented by an unique Boolean vector $\langle s_n, s_{n-1} ..., s_1 \rangle$, $s_i \in \{0,1\}, 1 \le i \le n$. A subset W $(\subseteq U)$ is represented by the Boolean function χ_W which maps u onto 1(0) if $u \in W (u \notin W)$. χ_W is the *characteristic function* (interchangeably used as *BDD*) of W.

The various components of TDES $G = (Q, \Sigma, \delta, q_0, Q_m)$ are represented symbolically in terms of *BDDs* as follows.

- *BDDs* for Q, q_0, Q_m, Σ are $\chi_Q, \chi_{q_i}, \chi_{Q_m}, \chi_\Sigma$, respectively:

  Each $q_i \in Q$ is represented as a binary m-tuple in $\mathbb{B}^m$ ($m = \lceil log_2 n \rceil$ and $|Q| = n$). The tuple is $\langle s_{im}, s_{i(m-1)}, ..., s_{i1} \rangle$ which is mapped to the Boolean function $\langle s_m.s_{m-1}...s_1 \rangle$ where $s_{ij} (\in \{0,1\})$ is s_j $(\overline{s_j})$ if s_{ij} is 1 (if s_{ij} is 0); this is denoted by χ_{q_i} and finally, $\chi_Q = \vee_{i=1}^{n} \chi_{q_i}$. In a similar manner, *BDDs* for q_0, Q_m, Σ can be represented.

- $\delta : Q \times \Sigma \mapsto Q$. Three different sets of Binary tuples represent the source states, events and target states of δ. So, $q_i{}' = \delta(q_i, \sigma)$ can be expressed using *BDD* $\chi_{i\mapsto} \equiv (q_i = \langle s_m s_{m-1} ... s_1 \rangle) \wedge (\sigma = \langle e_k e_{k-1} ... e_1 \rangle) \wedge (q_i{}' = \langle s'_m s'_{m-1} ... s'_1 \rangle)$. Finally, $\chi_{\mapsto} = \vee_{i=1}^{N} \chi_{i\mapsto}$, where $|\delta| = N$.

Steps for the Symbolic Computation of the Supervisor:

1. The task execution model and deadline specification model for each task τ_i is represented by its corresponding *BDD* $(\chi_{T_{i\mapsto}})$ and $(\chi_{H_{i\mapsto}})$, respectively.

2. Symbolic representation for the product composition $T(= T_1||T_2||...||T_n)$ and $H(= H_1||H_2||...||H_n)$ are obtained by applying the `AND` operation on the *BDDs* of individual task and specification models. Such a direct application of the `AND` operation suffices in our case because the event sets corresponding to the finite automata representing T_i's and H_i's are same [77]. So, BDD for T is denoted by: $\chi_{T\mapsto} = \wedge_{i=1}^{n} \chi_{T_{i\mapsto}}$. Similarly, $\chi_{H\mapsto}$ can be obtained.

3. BDD for $M(= T||H)$ is then computed as $\chi_{M\mapsto} = \chi_{T\mapsto} \wedge \chi_{H\mapsto}$. Similarly, χ_Q, χ_{Q_m}, $\chi_{q_0{}^M}$, and χ_Σ of M can be determined.

4. M' from M:

   (a) Computation of co-reachable states in M (represented by BDD $\chi_{Q'}$) using `PreImage(`$\chi_{Q_m}, \chi_{M\mapsto}$`)` [1].

   (b) Computation of blocking states in M (BDD as χ_{block}) $:= \chi_Q \wedge \urcorner \chi_{Q'}$. $\urcorner \chi_{Q'}$ is BDD for all non-co-reachable states, χ_Q represents all states in M and their conjunction gives only the blocking states in M.

   (c) Computation of states which are blocking and have transition(s) with accepting event p_i emanating from them (BDD as $\chi_{Q_{p_i}}$): First we obtain, $\chi_{p_{i\mapsto}} := (\chi_{M_\mapsto} \wedge \chi_{block}) \wedge \chi_{p_i}$. Here, $(\chi_{M_\mapsto} \wedge \chi_{block})$ is the set of transitions emanating from the blocking states and conjunction with χ_{p_i} filters transition(s) having

[1] `PreImage(`$\chi_{Q_m}, \chi_{M\mapsto}$`)` uses the standard BDD operation `pre`$_\exists$ to compute the set of states in M that in one transition can reach a (marked) state in χ_{Q_m}. It is repeated until fix-point.

p_i. $\chi_{Q_{p_i}}$ (i.e., source states of χ_{p_i}) can be obtained from $\chi_{p_i\mapsto}$ by extracting out only source states (from the transition relations of the form $\langle s_m s_{m-1} ... s_1$ $e_k e_{k-1} ... e_1$ $s'_m s'_{(m-1)} ... s'_1 \rangle$) using `exists` [1], i.e., $\exists e_k e_{k-1} ... e_1$ $s'_m s'_{m-1} ... s'_1$ $\chi_{p_i\mapsto}$.

(d) Computation of states with outgoing transition(s) on arrival event a_i and leading to states in $\chi_{Q_{p_i}}$ (BDD as $\chi_{Q_{a_i}}$): First we compute $\chi_{a_i\mapsto} := (\chi_{M\mapsto} \wedge \chi_{Q_{p_i}}[X^{Q_{p_i}} \rightarrow X^{Q'_{p_i}}]) \wedge \chi_{a_i}$. Here, $\chi_{Q_{p_i}}[X^{Q_{p_i}} \rightarrow X^{Q'_{p_i}}]$ represents re-naming of the source state label $X^{Q_{p_i}}$ with the target state label $X^{Q'_{p_i}}$. Its conjunction with $\chi_{M\mapsto}$ generates transitions leading to states in $\chi_{Q_{p_i}}$. Among those, transitions having arrival event a_i can be filtered through the conjunction with χ_{a_i}. Finally, (source states of χ_{a_i}) $\chi_{Q_{a_i}}$ can be obtained from $\chi_{a_i\mapsto}$ using `exists`.

(e) Addition of r_i from a state in $\chi_{Q_{p_i}}$ (from which p_i emanates) to the state in $\chi_{Q_{a_i}}$ (from which the corresponding a_i emanates). This can be obtained as $\chi_{M'\mapsto} := \chi_{M\mapsto} \vee \langle \chi_{Q_{p_i}} \vee \chi_{r_i} \vee \chi_{Q_{a_i}} \rangle$.

The steps 4c to 4e are repeated for all n tasks in the system. The BDD $\chi_{M'\mapsto}$ represents the transition structure of M'. If any r_i is added to M, then re-compute $\chi_{Q'}$ and χ_{block}.

5. Computation of states in $\chi_{M'\mapsto}$ (represented by BDD $\chi_{Q''}$) that are co-reachable to deadlock states in χ_{block} through uncontrollable event using `PreImage-uc` ($\chi_{block}, \chi_{M'\mapsto}$) [2]. Here, $\chi_{Q''}$ represents the *uncontrollable tails* in M' which result in a deadlock, i.e., $D_{uc}(L(T) - L_m(M'))\Sigma^*$.

   The steps 4 and 5 are repeated until the fix-point is reached.

6. Computation of safe states in M' (represented by BDD χ_{Q_s}) using a forward reachability search `Image_Restricted` ($\chi_{Q''}, \chi_{q_0 M'}, \chi_{M'\mapsto}$) [3] to remove the safe states in

[1] `exists` is a standard BDD operation applied on the boolean variable x and boolean function f, $\exists x.f = f|_{x=0} \vee f|_{x=1}$; it makes f independent of x.

[2] `PreImage-uc`($\chi_{block}, \chi_{M'\mapsto}$) is similar to `PreImage` which computes the set of states in M' that in one transition can reach a state in χ_{block} through uncontrollable event. It is repeated until fix-point.

[3] `Image_Restricted`($\chi_{Q''}, \chi_{q_0 M'}, \chi_{M'\mapsto}$) computes the set of states in M' that can be reached in one transition from $\chi_{q_0 M'}$ except the states in $\chi_{Q''}$. It is repeated until fix-point.

Table 4.3: *Comparison of number of states in $supC(L_m(M'))$: TDES vs BDD nodes*

	#Tasks	[Range for E_i: 10 to 25, Range for D_i: 75 to 100]						
	2		**10**		**15**		**20**	
#Processors	**TDES**	**BDD**	**TDES**	**BDD**	**TDES**	**BDD**	**TDES**	**BDD**
2	709	443	7,276	783	11,845	947	16,049	1,123
5	2,895	505	24,253	1,983	39,843	2,789	53,496	3,209
10	5,478	612	46,852	2,848	76,247	4,837	95,720	5,543

M' that are not reached from the initial state q_0. The resulting BDD χ_{Q_s} is same as $supC(L_m(M')) = L_m(M') - D_{uc}(L(T) - L_m(M'))\Sigma^*$ followed by *trim* operation.

Table 4.3 presents the total number of TDES and BDD (implemented using CUDD [101]) states in the final supervisor. It can be observed that in general, the number of BDD nodes are much lower compared to the number of TDES states. Further, their differences widen with increase in tasks and/or processors.

4.3.2 Example for Symbolic Computation of Supervisor

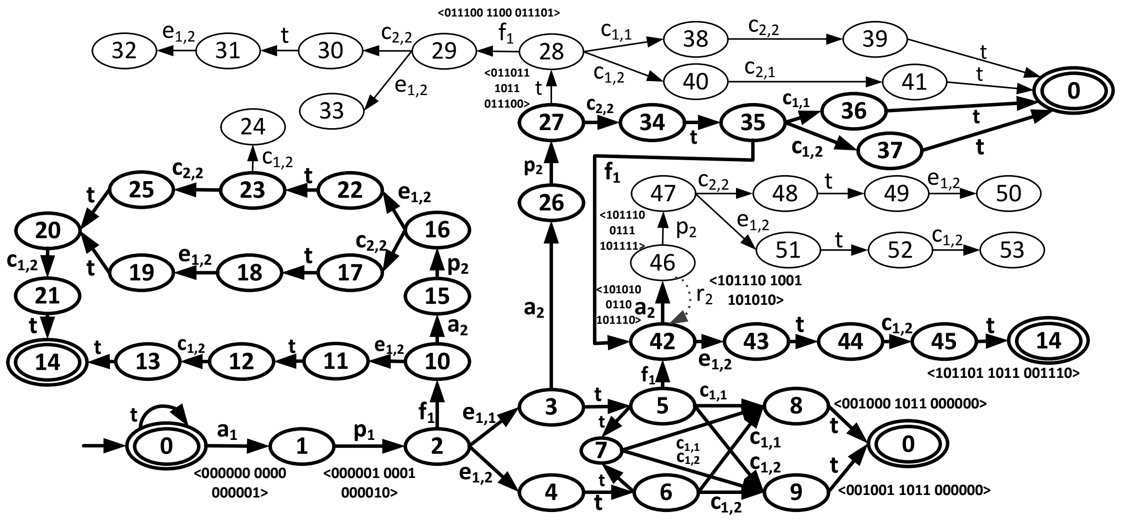


Figure 4.10: *Symbolic representation of M.*

In this section, the symbolic computation of the supervisor has been explained using the same example presented for TDES based representation. The first three steps of the symbolic computation involves: 1. Representation of TDESs corresponding to individual task execution (T_i) and specification (H_i) models using BDDs, 2. Computation of the

BDD for the composite task execution model T using AND $(\wedge)$ over individual task execution models and similarly, the computation of the BDD for the specification model H, 3. Computation of the BDD for $M = T||H$ using AND of the BDDs corresponding to T and H.

As these steps are straight forward [Miremadi et al. 2012], we start the demonstration from Step 4. The BDD representation of M is shown in Figure 4.10. Here, each state is assigned with a unique 6-bit boolean vector and the corresponding decimal representation is shown within each state. For example, State 0 is encoded as $\langle 000000 \rangle$. Similarly, each event is assigned with a unique 4-bit boolean vector. For example, events a_1, p_1, t are assigned with boolean vectors $\langle 0000 \rangle$, $\langle 0001 \rangle$ and $\langle 1011 \rangle$, respectively. Each transition is represented by $\langle s_6s_5s_4s_3s_2s_1\ e_4e_3e_2e_1\ s'_6s'_5s'_4s'_3s'_2s'_1 \rangle$ where, $\langle s_6s_5s_4s_3s_2s_1 \rangle$ represents the source state, $\langle s'_6s'_5s'_4s'_3s'_2s'_1 \rangle$ represents the target state and $\langle e_4e_3e_2e_1 \rangle$ represents the event. For example, the transition from State 0 to State 1 on event a_1 is represented by $\langle 000000\ 0000\ 000001 \rangle$. To limit the figure size, only a few transitions have been shown explicitly using their boolean representations. Let the BDDs χ_{Q_m} and $\chi_{M_{\mapsto}}$ represent the set of marked states $Q_m (= \{0, 14\})$ and the transition relation of M, respectively. Let us continue from Step 4 of the symbolic supervisor computation.

4a. *Computation of BDD $\chi_{Q'}$ representing all the states in $\chi_{M_{\mapsto}}$ that are co-reachable to χ_{Q_m}*: It uses PreImage($\chi_{Q_m}, \chi_{M_{\mapsto}}$) to compute the set of states in M that in one transition can reach a state in χ_{Q_m} and it is repeated until the fix-point is reached. The Steps (i) to (v) given below explains the process of computing PreImage($\chi_{Q_m}, \chi_{M_{\mapsto}}$).

i. *Initialization*: $\chi_{Q'} := \chi_{Q_m}$, where χ_{Q_m} contains the set of marked states $\{0, 14\}$.

ii. *Swapping source and target state of a transition in $\chi_{Q'}$*: PreImage takes the set of marked states $\{0, 14\}$ and starts with State 0 which is represented as $\langle 000000\ ....\ ...... \rangle$ (a . (dot) represents *don't care*). The operation PreImage first assigns the value 000000 to the boolean variables $\langle s'_6s'_5s'_4s'_3s'_2s'_1 \rangle$ that are used for encoding the target states, while the values of $\langle s_6 s_5 s_4 s_3 s_2 s_1\rangle$ that are used for encoding the source states become *don't cares.* So, the encoding of the marked state 0 becomes $\langle$...... .... 000000$\rangle$ representing that the target state is 0.

iii. *Conjunction of $\chi_{Q'}$ and $\chi_{M_{\mapsto}}$*: The source and target states swapped version of BDD $\chi_{Q'}$, is conjuncted with the BDD $\chi_{M_{\mapsto}}$. This operation returns the set of transitions which contain the marked state 0 as a target state, i.e., $\{\langle$001000 1011 000000$\rangle$, $\langle$001001 1011 000000$\rangle$, $\langle$100100 1011 000000$\rangle$, $\langle$100101 1011 000000$\rangle$, $\langle$100110 1011 000000$\rangle$, $\langle$100111 1011 000000$\rangle$, $\langle$101001 1011 000000$\rangle\}$.

iv. *Extracting source states from the result of $\chi_{Q'} \wedge \chi_{M_{\mapsto}}$*: Let us consider the transition $\langle$001000 1011 000000$\rangle$ representing $\langle 8, t, 0\rangle$. From this transition, the source state is extracted using `exists`, i.e., $\exists \hat{E}.\ \hat{S}'.\langle$001000 1011 000000$\rangle$, where $\hat{E} = \langle e_4, e_3, e_2, e_1\rangle$ and $\hat{S}' = \langle s'_6, s'_5, s'_4,\ s'_3, s'_2, s'_1\rangle$ returns $\langle$001000 ....$\rangle$ which corresponds to State 8. In a similar manner, all the remaining source states will be obtained from the transition relation using $\exists \hat{E}.\hat{S}'.(\chi_{Q'} \wedge \chi_{M_{\mapsto}})$. This operation returns the following set of states: $\{\langle$001000$\rangle$, $\langle$001001$\rangle$, $\langle$100100$\rangle$, $\langle$100101$\rangle$, $\langle$100110$\rangle$, $\langle$100111$\rangle$, $\langle$101001$\rangle\}$; the states are: $\{8, 9, 36, 37, 38, 39, 41\}$.

Similarly, when the Steps (ii) to (iv) are applied for the marked state 14, the set of states $\{13, 21, 45\}$ which can reach the marked state 14 through a single transition is obtained.

v. *Continue until fix-point is reached*: `PreImage` continues to iterate over the co-reachable states, until no more co-reachable states are found.

Finally, `PreImage(`$\chi_{Q_m}, \chi_{M_{\mapsto}}$`)` returns the BDD $\chi_{Q'}$ representing the following set of states: $\{0, 1, 2, 3,\ ...,\ 22, 23, 25,\ 26, 27,\ 28, 34, 35,\ ..., 45\}$.

4b. *Computation of BDD χ_{block} representing blocking states in M*: This can be done by computing the complement of $\chi_{Q'}$ against the universal set consisting of states encoded from 0 to 63 (because $2^6 = 64$). However, the numbers 0 to 53 only represent the valid states in M. Hence, the result of the complement will include invalid states from 54 to 63. Therefore, $\neg\chi_{Q'}$ is conjuncted with the BDD χ_Q representing the states in M to discard the invalid states in $\neg\chi_{Q'}$. The resulting BDD χ_{block} contains $\{29, 30, 31, 32, 33, 46, 47, ..., 53\}$ that represents the states leading to deadlock.

4c. *Computation of BDD $\chi_{Q_{p_i}}$ representing the subset of states in χ_{block} having transition with p_i:*

   i. Compute the set of all transitions leading from states in χ_{block} using $(\chi_{M_{\mapsto}} \wedge \chi_{block})$. This will return the following set of transitions: $\{\langle 29, e_{1,2}, 33\rangle$, $\langle 29, c_{2,2}, 30\rangle$, $\langle 30, t, 31\rangle$, $\langle 31, e_{1,2}, 32\rangle$, $\langle 46, p_2, 47\rangle$, $\langle 47, c_{2,2}, 48\rangle$, $\langle 47, e_{1,2}, 51\rangle$, $\langle 51, t, 52\rangle$, $\langle 52, c_{1,2}, 53\rangle$, $\langle 48, t, 49\rangle$, $\langle 49, e_{1,2}, 50\rangle\}$.

   ii. Then, compute the BDD $\chi_{p_{i\mapsto}} := (\chi_{M_{\mapsto}} \wedge \chi_{block}) \wedge \chi_{p_i}$ to filter out only the transitions on acceptance event p_i. It may be observed that there is only one transition emanating from the blocking states that contain acceptance event p_2, i.e., $\langle 46, p_2, 47\rangle$. The resulting BDD $\chi_{p_{i\mapsto}}$ represents the transition from State 46 to 47 on event p_2, i.e., $\langle 101110\ 0111\ 101111\rangle$.

   iii. From this transition, the BDD $\chi_{Q_{p_i}}$ representing the source state is extracted using `exists`, i.e., $\exists \hat{E}.\hat{S}'.\chi_{p_{i\mapsto}}$, where $\hat{E} = \langle e_4, e_3, e_2, e_1\rangle$ and $\hat{S}' = \langle s'_6, s'_5, s'_4, s'_3, s'_2, s'_1\rangle$ returns $\langle 101110\rangle$ which corresponds to the state $\{46\}$.

4d. *Computation of BDD $\chi_{Q_{a_i}}$ representing the subset of states in $\chi_{M_{\mapsto}}$ having transition with a_i leading to $\chi_{Q_{p_i}}$:*

   i. Swap the source state in $\chi_{Q_{p_i}}$ to target state: In $\chi_{Q_{p_2}}$, the source state 46 has been changed to target state, i.e., $\langle ......\ ....\ 101110\rangle$.

   ii. Then, it is conjuncted with $\chi_{M_{\mapsto}}$ and it results in the transition $\langle 101010\ 0110\ 101110\rangle$ which represents $\langle 42, a_2, 46\rangle$.

iii. Again, the resulting transition is conjuncted with χ_{a_2}. Since, χ_{a_2} represents ⟨...... 0110 ......⟩, the conjunction results in ⟨101010 0110 101110⟩ which is denoted by the BDD $\chi_{a_i \mapsto}$. Here, ⟨101010 0110 101110⟩ represents the transition from State 42 to 46 on event a_2.

iv. From this transition, the BDD $\chi_{Q_{a_2}}$ representing the source state is extracted using `exists`, i.e., $\exists \hat{E}.\hat{S}'.\chi_{a_2 \mapsto}$, i.e., ⟨101010⟩ which corresponds to the state $\{42\}$.

4e. The rejection transition r_2 is added from state 46 to 42 (i.e., ⟨101110 1001 101010⟩) and finally, it is added to the transition structure of M by disjuncting it with the BDD $\chi_{M \mapsto}$. The resulting BDD is named as $\chi_{M' \mapsto}$.

Re-computation of BDDs $\chi_{Q'}$, χ_{block}:

i. Re-compute the set of co-reachable states by invoking `PreImage` with χ_{Q_m} and $\chi_{M'_{\mapsto}}$. It returns the BDD $\chi_{Q'}$ representing the following set of states: $\{0, 1, 2, 3, ..., 22, 23, 25, 26, 27, 28, 34, 35, ..., 45, 46\}$. It may be noted that state 46 has been included into the set of co-reachable states $\chi_{Q'}$. This is because the addition of the rejection transition from State 46 to 42 makes state 46 to become co-reachable.

ii. Using this updated $\chi_{Q'}$, the BDD χ_{block} is recomputed $\{29, 30, 31, 32, 33, 47, ..., 53\}$. Since, state 46 has become co-reachable, it is not added to χ_{block}.

5 *Computation of BDD $\chi_{Q''}$ representing uncontrollable tails in $\chi_{M'_{\mapsto}}$*: It uses `PreImage-uc` `(`χ_{block}`,` $\chi_{M'_{\mapsto}}$`)` to compute the states in M' which can potentially lead to a deadlock state via uncontrollable events and it is repeated until fix-point is reached.

i. *Initialization*: $\chi_{Q''} := \chi_{block}$, where χ_{block} contains $\{29, 30, 31, 32, 33, 47, ..., 53\}$.

ii. *Swapping source and target state in $\chi_{Q''}$*: `PreImage-uc` takes the set of blocking states $\{29, 30, 31, 32, 33, 47, ..., 53\}$ and starts with State 29 where it is represented as ⟨011101 ....⟩ (a . (dot) represents *don't care*). The operation `PreImage-uc` first assigns the value 011101 to the target state, while the source state becomes *don't care*. So the encoding of state 29 becomes $\langle$...... .... 011101$\rangle$, thus transforming it into a target state.

ii. *Conjunction of $\chi_{Q''}$ and $\chi_{M'_{\mapsto}}$*: The source and target state swapped version of BDD $\chi_{Q''}$, is conjuncted with the BDD $\chi_{M'_{\mapsto}}$. This operation returns the set of transitions which contain the state 29 as a target state, i.e., $\{\langle$011100 1100 011101$\rangle\}$ which represents the transition $\langle 28, f_1, 29\rangle$.

iii. *Conjunction of $(\chi_{Q''} \wedge \chi_{M'_{\mapsto}})$ and $\chi_{\Sigma_{uc}}$*: The result of $(\chi_{Q''} \wedge \chi_{M'_{\mapsto}})$ is conjuncted with the BDD $\chi_{\Sigma_{uc}}$ to filter out the set of transitions containing the uncontrollable events. This operation retains the transition $\{\langle$011100 1100 011101$\rangle\}$. This is because it contains the uncontrollable event f_1 (encoded as $\langle$1100$\rangle$).

iv. *Extracting source states from the result of $(\chi_{Q''} \wedge \chi_{M'_{\mapsto}}) \wedge \chi_{\Sigma_{uc}}$*: From transition $\{\langle$011100 1100 011101$\rangle\}$, the source state is extracted using `exists`, i.e., $\exists \hat{E}.\hat{S}'.\chi_{p_2\mapsto}$, where $\hat{E} = \langle e_4, e_3, e_2, e_1\rangle$ and $\hat{S}' = \langle s'_6, s'_5, s'_4, s'_3, s'_2, s'_1\rangle$ returns $\langle$011100 ....$\rangle$ which corresponds to State 28. Similarly, the set of states which can reach the remaining states in $\chi_{Q''}$ in one uncontrollable transition will be obtained.

v. *Continue until fix-point is reached*: `PreImage-uc` continues to iterate over the co-reachable states, until no more co-reachable states are found through uncontrollable transition.

Finally, `PreImage-uc(`$\chi_{block}, \chi_{M'_{\mapsto}}$`)` returns the BDD $\chi_{Q''}$ which contains $\{28, 29, 30, 31, 32, 33, 47, ..., 53\}$. It can be observed that $\chi_{Q''}$ contains State 28 in addition to the states that are present in $\chi_{Q'}$.

6 *Compute the set of safe states Q_s in M'*: This can be obtained by removing the states that are present in $\chi_{Q''}$ from M'. However, this will result in a scenario where some of the states in M' are co-reachable to Q_m, but not reachable from the initial state q_0. Therefore, we use `Image_Restricted (`$\chi_{Q''}, \chi_{q_0 M'}, \chi_{M'_{\mapsto}}$`)` to perform the restricted forward reachability search starting from the initial state q_0

of M' which is represented by BDD $\chi_{q_0 M'}$. Here, the term *restricted* emphasizes that no state in $\chi_{Q''}$ will be considered during the forward reachability search.

i. *Initialization*: Let χ_{Q_s} be the BDD representation of set of safe states Q_s. Initialize $\chi_{Q_s} := \chi_{q_0 M'}$, where $\chi_{q_0 M'}$ contains $\{0\}$.

ii. *Conjunction of* χ_{Q_s} *and* $\chi_{M'_{\mapsto}}$: χ_{Q_s} contains $\langle 000000 \text{ ....}\rangle$ representing the initial state 0 of M' and it is conjuncted with the BDD $\chi_{M'_{\mapsto}}$. This operation returns the set of transitions which contain the state 0 as a source state, i.e., $\{\langle 000000\ 0000\ 000001\rangle\}$ which represents the transition $\langle 0, a_1, 1\rangle$.

iii. *Extracting target states from the result of* $(\chi_{Q_s} \wedge \chi_{M'_{\mapsto}})$: From transition $\langle 000000\ 0000\ 000001\rangle$, the target state is extracted using `exists`, i.e., $\exists\hat{E}.\ \hat{S}.(\chi_{Q_s} \wedge \chi_{M'_{\mapsto}})$, where $\hat{E} = \langle e_4, e_3, e_2, e_1\rangle$ and $\hat{S} = \langle s_6, s_5, s_4, s_3, s_2, s_1\rangle$ returns $\langle \text{...... ....}\ 000001\rangle$ which corresponds to State 1.

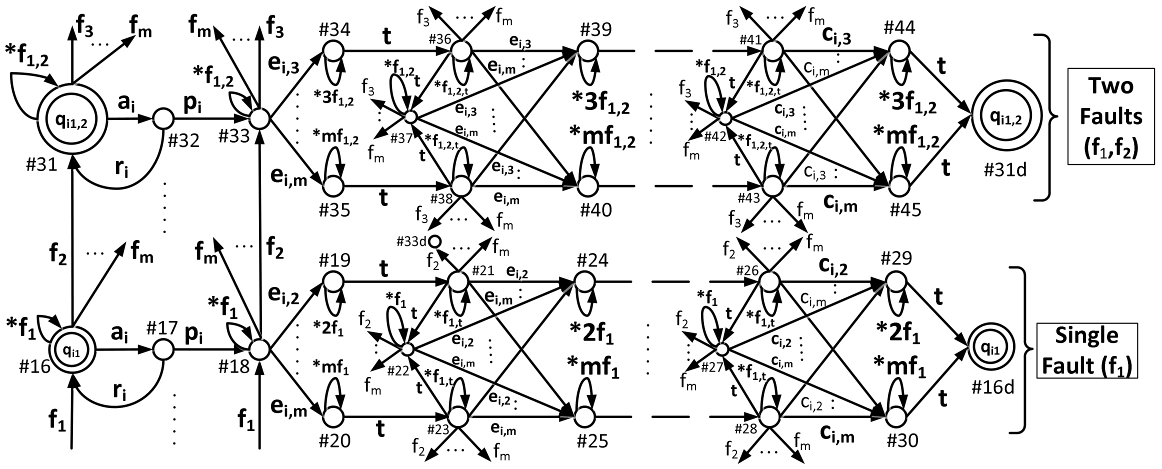
iv. *Conjunction of* $(\chi_{Q_s} \wedge \chi_{M'_{\mapsto}})$ *and* $\neg\chi_{Q''}$: This conjunction with the complement of $\chi_{Q''}$ is performed to remove any state that is part of $\chi_{Q''}$. Since, State 1 is not part of $\chi_{Q''}$, it will be retained and added to the set of safe states $\chi_{Q_s} = \{0, 1\}$.

v. *Continue until fix-point is reached*: `Image_Restricted` continues to iterate over the reachable states in χ_{Q_s}, until no more reachable states are found.

Finally, `Image_Restricted` $(\chi_{Q''}, \chi_{q_0 M'}, \chi_{M'_{\mapsto}})$ returns the BDD χ_{Q_s} representing the safe states in M' that are reachable from the initial state, i.e., $Q_s = \{0, 1, ..., 27, 34, ..., 37, 42, ..., 46\}$. This matches with the $supC(L_m(M'))$ shown using thick lines in Figure 4.10.

4.4 Handling multiple faults

In this section, we discuss a methodology for extending the TDES based task execution model T_i (presented in Section 4.2.2) to handle multiple faults. Two fundamental restrictions in the task execution model T_i (presented earlier in Figure 4.1) which makes it



$*f_{V'} = \sum \setminus [\sum_i \cup \sum_{fau} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k})]$ $(k \in V')$, where V' is the set of faulty processors

$*f_{V',t} = \sum \setminus [\sum_i \cup \sum_{fau} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k}) \cup \{t\}]$ $(k \in V')$

$*xf_{V'} = \sum \setminus [\sum_i \cup \sum_{fau} \cup \{t\} \cup (\cup_{j=1\ldots n} e_{j,k}) \cup (\cup_{j=1\ldots n} c_{j,k}) \cup (\cup_{j=1\ldots n} e_{j,x}) \cup (\cup_{j=1\ldots n} c_{j,x})]$ $(x = 1, \ldots, m)$, $(k \in V')$

Figure 4.11: *The execution model T_i for a task τ_i (multiple faults).*

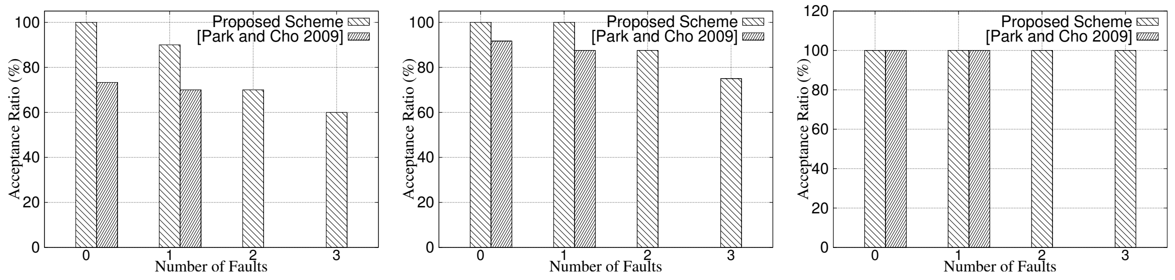
ineligible to be applied in a multi fault scenario are as follows: (1) There are no outgoing transitions defined based on processor fault events from the *Single Fault* structure. (2) When T_i is within the *No Fault* structure, it preserves the information about the processor on which task τ_i was executing before the occurrence of a *tick*. However, once T_i moves to the *Single Fault* structure from the *No Fault* structure, it does not preserve this information. This makes it difficult to decide whether the task needs to be restarted from the beginning or it can continue with the normal execution after a fault. For example, let us assume that T_i is at State #20. Here, T_i forgets the information that τ_i was executing on processor V_2 after it reaches State #22 through the *tick* event. Hence, we present a revised task execution model for T_i in Figure 4.11 so that it can handle multiple faults. The *No Fault* sub-structure is not shown in Figure 4.11 since it is same as Figure 4.1. The part associated with the handling of single faults in Figure 4.1 have been modified and appropriately replicated in Figure 4.11 to address the additional requirements mentioned above for tolerating up to a maximum of w faults ($w < m$). The *Single Fault* structure shown in Figure 4.11 now contains outgoing transitions on fault events to allow another fault. For example, the states #16, #18, ..., #26 contain transition on fault events $\{f_2, ..., f_m\}$. Also, the *Single Fault* structure captures the processor on which τ_i was executing during the last *tick* event. For example, if it is at State #21, then it can be easily traced that τ_i was executed on V_2 during the last *tick* event.

We now present an overview of the working principle of this model by first considering a two fault scenario under two distinct cases: (1) two *interleaved* faults (2) a *burst* of two consecutive faults. Let us assume that T_i is initially at State #5 of the *No Fault* structure (shown in Figure 4.1) for both cases. *Case 1:* If processor V_1 fails, then T_i will move to State #18 through the transition f_1. Now, the scheduler may assign τ_i on anyone of the available processors $(V_2, ..., V_m)$ for execution. Suppose, τ_i is assigned and executed on V_2 for one *tick* unit, thus moving T_i to State #21. Here, the second fault is encountered on processor V_2. This drives T_i to move from State #21 to State #33d through transition f_2 and causes τ_i to restart its execution. In this case, the two faults are separated by a single *tick*. The case where the faults are separated by multiple *ticks* is also handled in a similar fashion. *Case 2:* If both processors V_1 and V_2 fail simultaneously at State #3 (say, f_1 followed by f_2), then T_i will first move to State #18 through transition f_1 and then to State #33 through transition f_2. Hence, in both cases, T_i is able to dynamically reconfigure itself to the appropriate state irrespective of the fault scenario.

Though the formal steps to synthesize the scheduler for the multiple fault scenario is not discussed here, it will be similar to the synthesis procedure employed for the single fault scenario (Section III). However, the condition under which no task will be rejected in M' gets slightly modified as follows: *the instantaneous load* $(\rho(t))$ *at each time instant is at most* m (*for normal mode of operation*), $m-1$ (*subsequent to single processor fault*), ..., $m-w$ (*subsequent to* w *processor faults*).

4.5 Comparative Study

In this sub-section, we present a comparative study on the *acceptance ratio* (i.e., ratio of the total number of accepted tasks to the total number of tasks arrived) of our proposed scheme with that of [86]. We consider a task set I of size 30 executing on a multiprocessor platform consisting of 11 processors. I is: $\{(2,8)$, $(1,4)$, $(4,16)$, $(3,12)$, $(1,4)$, $(1,4)$,



(a) *All tasks arrive together* **(b)** *24 tasks arrive together* **(c)** *10 tasks arrive together*

Figure 4.12: *Acceptance ratio for proposed scheme and [86]*

$(5, 20)$, $(8, 32)$, $(4, 16)$, $(1, 4)$, $(2, 3)$, $(3, 5)$, $(10, 30)$, $(1, 4)$, $(1, 3)$, $(1, 4)$, $(5, 15)$, $(6, 10)$, $(6, 10)$, $(3, 5)$, $(3, 5)$, $(2, 3)$, $(2, 3)$, $(2, 8)$, $(1, 4)$, $(8, 15)$, $(1, 4)$, $(4, 15)$, $(2, 8)$, $(3, 15)\}$. We have measured the *acceptance ratio* under different arrival patterns of tasks in I and fault patterns on processors. Since, [86] deals with single faults, we restrict the comparison of our scheme with them under no fault and single fault scenarios only. However, we have shown the results for our scheme under multiple faults (up to 3). A brief summary on the comparative study is as follows:

- **Figure 4.12c**: When only first 10 tasks in I arrive simultaneously, both the proposed scheme and [86] accept all the arrived tasks under no fault and single fault cases, since the system is under-loaded. Even for the cases of faults in 2 and 3 processors, the proposed scheme still maintains 100% acceptance ratio. This is because even with 3 processor faults, there are 8 remaining processors and the instantaneous load of the first 10 tasks when they arrive simultaneously is $\rho(t) = 2.5$ (< 8); the system remains under-loaded even in the presence of 3 faults.

- **Figure 4.12b**: When only the first 24 tasks in I arrive simultaneously, our scheme is still able to archive 100% acceptance ratio as compared to [86] which achieves 91.6% and 87.5%, under no fault and single fault cases, respectively. It may be noted that our proposed scheme can accept more tasks due to its preemptive nature. However, as the number of faults increases, the decrease in acceptance ratio becomes inevitable for our scheme as well due to system overload (i.e., $\rho(t) = 9.25 > 8$). For example under 2 faults, the acceptance ratio is 87.5%.

- **Figure 4.12a**: When all tasks in I (with $\rho(t) = 11$) arrive simultaneously, decrease in acceptance ratio (with respect to number of faults) for both the schemes are higher compared to the cases for 10 and 24 tasks; this is obvious as higher number of tasks implies higher system load. However, it may be noted that the decrease in acceptance ratio is higher in [86] compared to the proposed scheme in all cases. For example, comparing the drops in acceptance ratios under single fault, as the number of arrived tasks is raised from 10 to 24 (24 to 30), it may be observed that the proposed scheme suffers a drop of 0% (10%). However, for [86] the drop is 12.5% (17.5%), for the same scenarios.

To summarize, the study shows that as system load increases (due to rise in number of tasks, rise in number of processor faults etc.), the difference in acceptance ratio of the proposed scheme with that of [86] widens. The results obtained above may vary depending upon the task set under consideration, the arrival pattern of the tasks, number of processors, time of occurrence and number faults etc.

4.6 Summary

In this chapter, we have presented a systematic way of synthesizing an *optimal fault-tolerant scheduler* for RT multiprocessor systems which process a set of dynamically arriving aperiodic tasks. The fundamental strategy which has the ability to handle at-most one permanent processor fault has been extended to incorporate multiple fault tolerance. A mechanism to obtain a non-blocking supervisor using BDD based symbolic computation has been proposed to control the exponential state space complexity of the exhaustive enumeration oriented synthesis methodology.

It may be noted that the proposed scheduler synthesis procedure considers aperiodic tasks only. However, the procedure can be extended for sporadic tasks by replacing the deadline specification model of aperiodic tasks presented in Section 4.2.4 with the timing specification model for sporadic tasks presented in Section 3.3.4. The next chapter presents the power-aware scheduler synthesis framework for a set of real-time tasks executing on a multi-core platform.

Chapter 5

Power-aware Scheduling on Homogeneous Multi-cores

In the previous chapter, we have considered the fault-tolerant scheduling of dynamically arriving real-time tasks executing on a homogeneous multiprocessor platform. Apart from providing tolerance against processor faults, safety-critical systems implemented on modern multi-core chips with high gate densities, must adhere to a strict power budget called *Thermal Design Power* (TDP) constraint, in order to control functional unreliability due to temperature hot-spots [78]. Performance throttling mechanisms such as *Dynamic Thermal Management* (DTM) are triggered whenever power dissipation in the system crosses the stipulated *TDP* constraint [1]. Activation of *DTM* involves steps such as powering-down cores, clock gating, dynamic supply voltage and frequency, etc. These steps introduce unpredictability in timing behavior of the systems. An important objective of safety-critical scheduler designs implemented on these platforms is therefore, to ensure *DTM-free operation* over the entire schedule length. This can be achieved by always ensuring the cumulative peak power consumption of the cores to be within a specified *TDP* limit [78]. Hence in this chapter, we propose *an optimal off-line non-preemptive scheduler synthesis mechanism that guarantees chip-level peak power consumption to be capped within a stipulated TDP*. In addition, to accurately model system level power dissipation at any instant of time during the co-execution of multiple tasks, we also extend our scheduler synthesis scheme to accurately capture phased power dissipation behavior of a task.

5.1 Related Works

Recent research works [57, 79, 92] applied *Proportional-Integral-Derivative* (PID) control based techniques to minimize peak power consumption in multi-core systems. A hierarchical control framework to obtain optimal power versus performance trade-offs in asymmetric multi-core systems under a specified TDP has been presented in [79]. Amir et al. [92] developed a reliability-aware run time power management scheme. In [57], Anil et al. designed a power management scheme that can switch between accurate and approximate modes of execution, subject to system throughput requirements. However, these works [57,79,92] do not provide any guarantee on the timely execution of real-time tasks.

An adaptive micro-architecture based approach to meet power constraints is presented in [59]. This approach dynamically reconfigures micro-block sizes of a processor at run time. Similarly, Meng et al. explored power management through the adaptive reconfiguration of processor speeds and/or cache sizes at run time [75]. As dynamic reconfigurability introduces unpredictability in task execution times, these approaches become inapplicable in hard real-time systems.

In [67], Lee et al. developed a static scheduling method that attempts to minimize peak power consumption while satisfying all timing constraints of real-time tasks in a multi-core system. Later, Munawar et al. presented a scheme to minimize peak power usage for frame-based and periodic real-time tasks [78]. However, both these works [67, 78] are *heuristic* in nature and hence, they do not provide any guarantee towards finding a solution, if and whenever there exists a solution.

On the other hand, *SCTDES* based scheduling strategies result in optimal scheduler with respect to a given set of constraints. Recently, many researchers have developed a variety of real-time and fault-tolerant schedulers based on the SCTDES approach [41,105,106]. However, there do not currently exist any real-time power-aware *SCTDES* based scheduler synthesis mechanism. In this work, we utilize *SCTDES* to compute a correct-by-construction optimal *power-aware real-time scheduler* for multi-core systems. Table 5.1 synopsizes a qualitative comparison among the power-aware and SCTDES

Table 5.1: *A qualitative comparison among related works*

Method	**Real-time guarantee**	**Peak power aware**	**Approach**	**Optimal / Heuristic**	**Computational Complexity**
[57, 79, 92]	No	Yes	PID control	Optimal	Exponential
[59, 75]	No	Yes	Dynamic re-configuration	Heuristic	Polynomial
[41, 105, 106]	Yes	No	SCTDES	Optimal	Exponential
[67]	Yes	Yes	Static priority	Heuristic	Polynomial
[78]	Yes	Yes	Bin Packing	Heuristic	Polynomial
Proposed scheme	Yes	Yes	SCTDES	Optimal	Exponential. Handled by BDDs

based scheduling approaches discussed above.

5.2 Proposed Framework

System Model: We consider a real-time multi-core system consisting of a set I ($= \{\tau_1, \tau_2, ..., \tau_n\}$) of n independent non-preemptive periodic tasks to be scheduled on m homogeneous cores ($= \{V_1, V_2, ..., V_m\}$). Formally, we represent a periodic task τ_i as a 5-tuple $\langle A_i, E_i, D_i, P_i, \mathcal{B}_i \rangle$, where, $A_i (\in \mathbb{N})$ is the *arrival time* of the first instance of τ_i (relative to system start), $E_i (\in \mathbb{N})$ represents its *execution time*, $D_i (\in \mathbb{N};\ E_i \leq D_i)$ is the *relative deadline*, $P_i (\in \mathbb{N})$ denotes the fixed inter-arrival time between consecutive instances of τ_i and $\mathcal{B}_i$ $(\in \mathbb{N})$ denotes its *worst-case instantaneous power consumption* (*peak power*; measured in *watts*).

Problem Statement: *Given a set of n tasks and m processing cores, design an optimal supervisor which contains scheduling sequences that guarantee (i) no deadline miss for any task instance and (ii) chip-level peak power consumption to be upper bounded by a stipulated power-cap $\mathcal{B}$.*

5.2.1 Task Execution Model

The ATG model $T_{i,act}$ for executing a *non-preemptive* periodic task τ_i on a homogeneous multi-core system is shown in Figure 5.1. The ATG model $T_{i,act}$ is formally defined as follows:

$$T_{i,act} = (A_i^T, \Sigma_{i,act}, \delta_{i,act}^T, a_{i,0}^T, A_{i,m}^T)$$

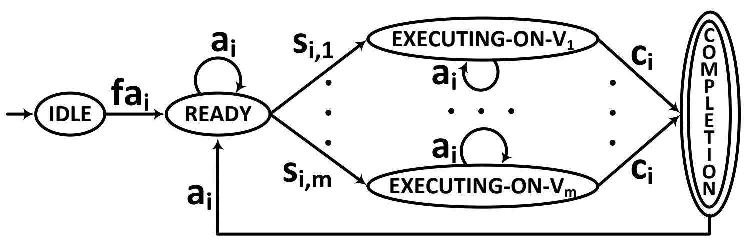


Figure 5.1: *Task execution model $T_{i,act}$ for periodic task τ_i*

where, A_i^T = {IDLE, READY, EXECUTING-ON-V_1, EXECUTING-ON-V_2, ..., EXECUTING-ON-V_m, COMPLETION} denotes a finite set of activities, $\Sigma_{i,act} = \{fa_i, a_i, s_{i,1}, s_{i,2}, ..., s_{i,m}, c_i\}$ is a finite set of events (the events are described in Table 5.2), $a_{i,0}^T$ = IDLE denotes the initial activity, and $A_{i,m}^T$ = {COMPLETION} is the marked activity. Given n individual ATG models $T_{1,act}$, $T_{2,act}$, ..., $T_{n,act}$, corresponding to the tasks τ_1, τ_2, ... τ_n, the finite set of all events becomes $\Sigma_{act} = \cup_{i=1}^{n}\Sigma_{i,act}$. The events in Σ_{act} is categorized as: (i) the set of *prospective* events $\Sigma_{spe} = \Sigma_{act}$ (since all events have finite time bounds), (ii) the set of *remote* events $\Sigma_{rem} = \emptyset$, (iii) the set of *uncontrollable* events $\Sigma_{uc} = \cup_{i=1}^{n}\{fa_i, a_i\} \cup \cup_{i=1}^{n}\{c_i\}$, (iv) the set of *controllable* events $\Sigma_c = \cup_{i=1}^{n} \cup_{j=1}^{m} \{s_{i,j}\}$, and (v) the set of *forcible* events $\Sigma_{for} = \Sigma_c$. The event set associated with the TDES representation becomes $\Sigma = \Sigma_{act} \cup \{t\}$, where, t denotes the passage of one unit time of the clock.

Table 5.2: *Description of events*

Event	**Description**
fa_i	Arrival of task τ_i's first instance
a_i	Arrival of task τ_i's next instance
$s_{i,j}$	Start of execution of τ_i on core V_j
c_i	Completion of τ_i's execution
t	Passage of one unit time of the global clock

Initially, $T_{i,act}$ stays at activity *IDLE* until the arrival of τ_i's first instance. On the occurrence of event fa_i, $T_{i,act}$ transits to activity *READY*. At this activity, there are m outgoing transitions on events $s_{i,j}$ ($j = 1, 2, ..., m$) to model the possibility of starting τ_i's execution on the processing core V_j. On $s_{i,j}$, $T_{i,act}$ moves to activity *EXECUTING-ON-V_j* to capture the execution of τ_i on processing core V_j. After the completion of execution,

$T_{i,act}$ transits to the marked activity *COMPLETION* on c_i. Next, $T_{i,act}$ moves back to activity *READY* on the arrival of τ_i's next instance (a_i) and τ_i continues its execution in a similar manner. The self-loops on event a_i at all activities of $T_{i,act}$ except *IDLE* and *COMPLETION*, are used to model the fixed inter-arrival time of periodic task τ_i.

In order to obtain a TDES model T_i from the ATG model $T_{i,act}$, we assign suitable lower and upper time bounds for each event in $\Sigma_{i,act}$, as follows:

- (fa_i, A_i, A_i): fa_i must occur exactly at the A_i^{th} tick from system start, to model the arrival of τ_i's first instance.
- (a_i, P_i, P_i): a_i must occur exactly at the P_i^{th} tick from the arrival of τ_i's previous instance, to model the periodic arrival pattern of τ_i.
- $(s_{i,j}, 0, D_i - E_i)$: $s_{i,j}$ must occur between 0 and $D_i - E_i$ ticks from the arrival of τ_i's current instance to ensure that there is sufficient time for the execution of task τ_i so that its deadline can be met.
- (c_i, E_i, E_i): c_i must occur exactly E_i ticks from the occurrence of $s_{i,j}$. This is to enforce the execution requirement E_i of task τ_i.

The detailed diagram of the TDES model T_i obtained using $T_{i,act}$ and the above time bounds, is shown in Figure 5.2.

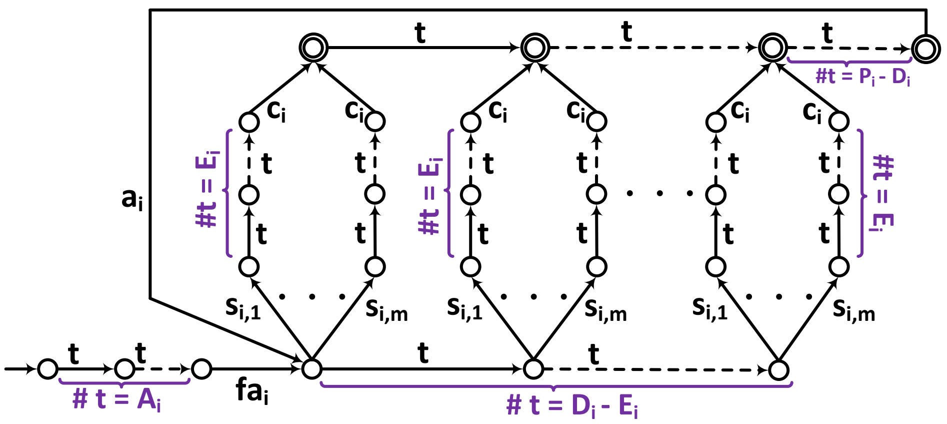


Figure 5.2: *TDES model T_i for periodic task $\tau_i\langle A_i, E_i, D_i, P_i, \mathcal{B}_i\rangle$*

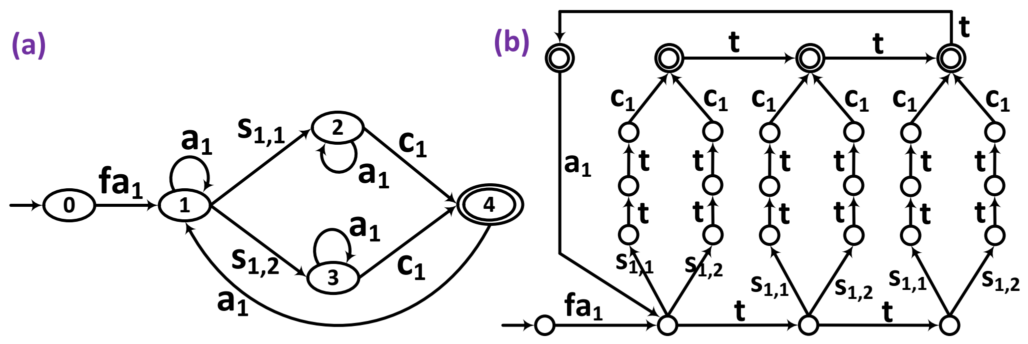


Figure 5.3: *(a) ATG $T_{1,act}$ for τ_1, (b) TDES model T_1 for τ_1.*

Example: Let us consider a simple two core system ($\{V_1, V_2\}$) consisting of two periodic tasks $\tau_1\langle 0,2,4,5,2\rangle$ and $\tau_2\langle 0,2,4,5,3\rangle$ with the power cap $\mathcal{B} = 4W$. Here, both τ_1 and τ_2 arrive together at system start (i.e., at 0). When executing, τ_1 (τ_2) dissipates $2W$ ($3W$) of power per unit time. The ATG model $T_{1,act}$ and TDES model T_1 for the execution of task τ_1 are shown in Figures 5.3(a) and (b), respectively. Similarly, the models $T_{2,act}$ and T_2 for task τ_2 can be constructed. $\square$

Composite Task Execution Model: It may be observed that the marked behavior $L_m(T_i)$ satisfies the following constraints corresponding to task τ_i: (i) arrival of the first instance at the exactly stipulated time instant, (ii) correct allocation of execution time, (iii) deadline satisfaction and (iv) ensuring fixed inter-arrival time between task instances. Given n individual TDES models T_1, T_2, ..., T_n, corresponding to tasks τ_1, τ_2, ... τ_n, a *synchronous product* composition $T = ||_{i=1}^{n} T_i$ on the models gives us the composite model representing the *concurrent execution of all tasks*. Here, individual models do not share any common event (i.e., $\cap_{i=1}^{n}\Sigma_{i,act} = \emptyset$) except *tick*. Thus, all models synchronize only on the *tick* event. Since, the individual models satisfy deadline and fixed-inter arrival time constraints, $L_m(T)$ also satisfies them. However, sequences in $L_m(T)$ *may violate resource constraint.*

Example (continued): Figure 5.4 shows the composite task execution model T (= $T_1||T_2$). Here, the composition procedure identifies states 37, 44, 40, 47, 43, 50 and 51 as *marked*. These states represent the situation when both tasks τ_1 and τ_2 have completed their execution. Let us consider the sequence $seq_1 = fa_1 fa_2 s_{1,2} s_{2,2} ttc_1 c_2 \in \overline{L_m(T)}$, which

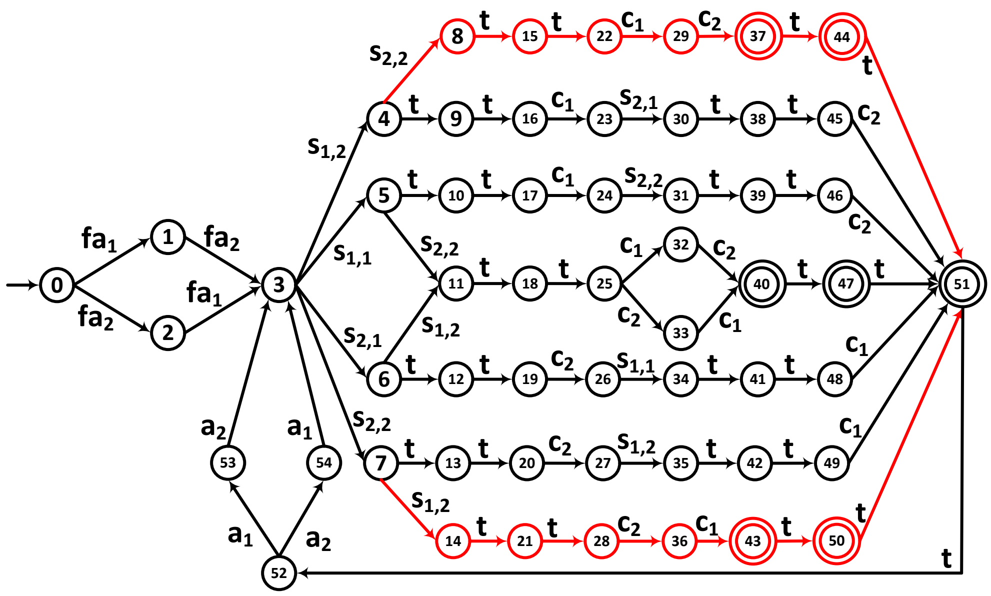
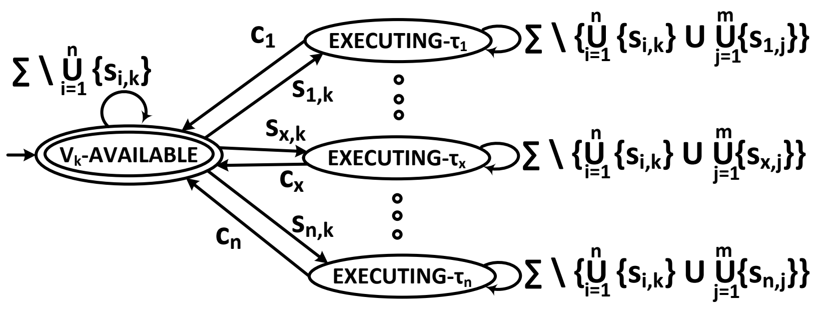


Figure 5.4: $T = T_1||T_2$ *(partial diagram).*

can be traced in T as follows: $0 \xrightarrow{fa_1} 1 \xrightarrow{fa_2} 3 \xrightarrow{s_{1,2}} 4 \xrightarrow{s_{2,2}} 8 \xrightarrow{t} 15 \xrightarrow{t} 22 \xrightarrow{c_1} 29 \xrightarrow{c_2} 37$. The sequence seq_1 depicts a co-execution scenario consisting of the two tasks which arrive simultaneously. Subsequent to their arrival, both tasks are assigned on core V_2 for execution (i.e., $s_{1,2}s_{2,2}$). Such an assignment violates resource constraint due to the simultaneous execution of both tasks τ_1 and τ_2 on core V_2 at the same time. Now, let us consider the sequence $seq_2 = fa_1fa_2s_{1,1}s_{2,2}ttc_1c_2 \in \overline{L_m(T)}$ which can be traced in T as: $0 \xrightarrow{fa_1} 1 \xrightarrow{fa_2} 3 \xrightarrow{s_{1,1}} 5 \xrightarrow{s_{2,2}} 11 \xrightarrow{t} 18 \xrightarrow{t} 25 \xrightarrow{c_1} 32 \xrightarrow{c_2} 40$. Here, τ_1 and τ_2 are assigned on cores V_1 and V_2, respectively. This assignment satisfies the *resource-constraint.* $\square$

It may be observed that $L_m(T)$ does not restrict the erroneous possibility of allowing multiple tasks to execute simultaneously on the same processing core. Hence, we develop the resource-constraint model to capture the following specification: *Once a task τ_i is allocated onto a processing core V_k, it remains allocated on V_k until its completion. Meanwhile, no other task τ_k ($\neq \tau_i$) is allowed to execute on V_k.* Such a specification is captured by the TDES model RC_k, as we discuss next in subsection.



$\Sigma \setminus \bigcup_{i=1}^{n}\{s_{i,k}\}$
V_k-AVAILABLE
EXECUTING-τ_1
EXECUTING-τ_x
EXECUTING-τ_n
c_1
$s_{1,k}$
$s_{x,k}$
c_x
$s_{n,k}$
c_n
$\Sigma \setminus \{\bigcup_{i=1}^{n}\{s_{i,k}\} \cup \bigcup_{j=1}^{m}\{s_{1,j}\}\}$
$\Sigma \setminus \{\bigcup_{i=1}^{n}\{s_{i,k}\} \cup \bigcup_{j=1}^{m}\{s_{x,j}\}\}$
$\Sigma \setminus \{\bigcup_{i=1}^{n}\{s_{i,k}\} \cup \bigcup_{j=1}^{m}\{s_{n,j}\}\}$

Figure 5.5: *TDES model RC_k for processing core V_k*

5.2.2 Resource-constraint Model

The TDES model RC_k for a processing core V_k in a homogeneous multi-core system is shown in Figure 5.5. RC_k is formally defined as, $(Q, \Sigma, \delta, q_0, Q_m)$, where $Q = \{V_k$-AVAILABLE, EXECUTING-τ_1, EXECUTING-τ_2, ..., EXECUTING-$\tau_n\}$, $\Sigma = \Sigma_{act} \cup \{t\}$, $q_0 = Q_m = V_k$-AVAILABLE. This model contains $n + 1$ states to capture scenarios in which anyone of the n tasks is executing on V_k or the processing core V_k is idle.

The self-loop $\Sigma \setminus \cup_{i=1}^{n}\{s_{i,k}\}$ at the initial state allows the possibility executing all events in Σ except $\cup_{i=1}^{n}\{s_{i,k}\}$, i.e., it disallows the start of any task on core V_k. The start of execution of any task, say τ_x, on V_k is modeled by an outgoing transition on the event $s_{x,k}$ to the state EXECUTING-τ_x from V_k-AVAILABLE. At this state, the self-loop $\Sigma \setminus \{\cup_{i=1}^{n}\{s_{i,k}\} \cup \cup_{j=1}^{m}\{s_{x,j}\}\}$ allows all but the events related to, (i) starts of any task on V_k ($\cup_{i=1}^{n}\{s_{i,k}\}$) and (ii) start of task τ_x on any processor ($\cup_{j=1}^{m}\{s_{x,j}\}$). On completion of execution, RC_k transits back to the initial state on event c_x from the state EXECUTING-τ_x. Hence, it may be observed that $L_m(RC_k)$ *ensures the exclusive execution of a single task on core V_k at any time instant.* Given m TDES models RC_1, RC_2, ..., RC_m corresponding to m cores, we can compute the composite resource-constraint satisfying model, $RC = ||_{i=1}^{m} RC_i$.

Example (continued): The models RC_1 for core V_1 and RC_2 for V_2 are shown in Figures 5.6(a) and 5.6(b), respectively.

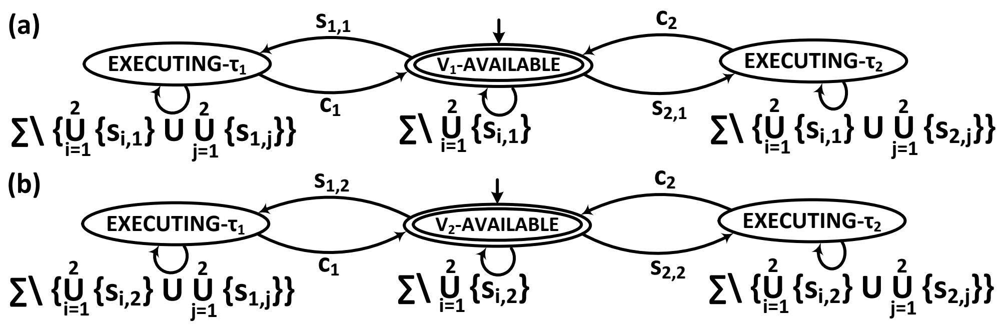


Figure 5.6: *(a) RC_1 for core V_1, (b) RC_2 for core V_2.*

5.2.3 Finding Resource-constraint Satisfying Sequences

Given the composite models T and RC, we can compute the initial supervisor candidate M^0 as: $T||RC$. In order to transform M^0 such that it becomes both *controllable* and *non-blocking*, we apply the *safe state synthesis* mechanism presented in [104]. This synthesis algorithm takes model M^0 and a set of forbidden states (which is an empty set, in our case) as inputs and computes M^1 which contains the set of safe states (both controllable and non-blocking) that are reachable from the initial state of M^0.

Theorem 5.2.1. $L_m(M^1)$ contains only and all sequences in $L_m(T)$ that satisfy the resource-constraint.

Proof. $L_m(T)$ contains both resource-constraint satisfying and violating sequences for all tasks in I. On the other hand, $L_m(RC)$ contains the sequences that satisfy the resource-constraint of all tasks in I. Since $L_m(M^0) = L_m(T) \cap L_m(RC)$, the marked behavior $L_m(M^1)$ contains *only* and *all* sequences in $L_m(T)$ that satisfy resource-constraint. □

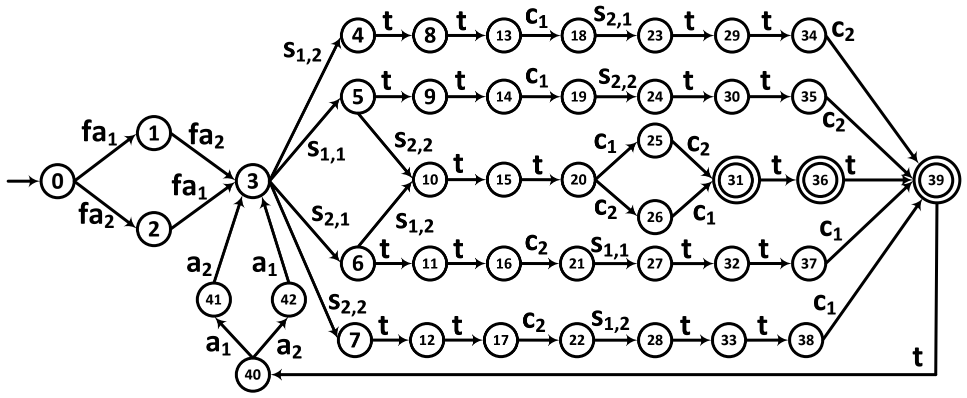


Figure 5.7: *Partial diagram of M^1.*

Example (continued): Figure 5.7 shows the partial digram of M^1. It may be observed that $L_m(M^1)$ does not contain the *resource-constraint* violating sequence seq_1 ($= fa_1fa_2s_{1,2}s_{2,2}ttc_1c_2tt \in \overline{L_m(T)}$). After proceeding through the states $0 \xrightarrow{fa_1} 1 \xrightarrow{fa_2} 3 \xrightarrow{s_{1,2}} 4$, M^1 *gets blocked* due to the absence of a transition on $s_{2,2}$ at *State* 4 (where $s_{2,2}$ is present at *State* 4 of T in Figure 5.4). More specifically, after processing the sub-string $fa_1fa_2s_{1,2}$ of seq_1, the next event in seq_1 is $s_{2,2}$. However, $s_{2,2}$ is not present at State 4 since it has been removed during the composition of $T||RC$. Thus, $seq_1 \notin L_m(M^1)$. Now, let us consider the *resource-constraint* satisfying sequence seq_2 ($= fa_1fa_2s_{1,1}s_{2,2}ttc_1c_2 \in \overline{L_m(T)}$) which can be traced in M^1: $0 \xrightarrow{fa_1} 1 \xrightarrow{fa_2} 3 \xrightarrow{s_{1,1}} 5 \xrightarrow{s_{2,2}} 10 \xrightarrow{t} 15 \xrightarrow{t} 20 \xrightarrow{c_1} 25 \xrightarrow{c_2} 31$. Hence, $seq_2 \in L_m(M^1)$.

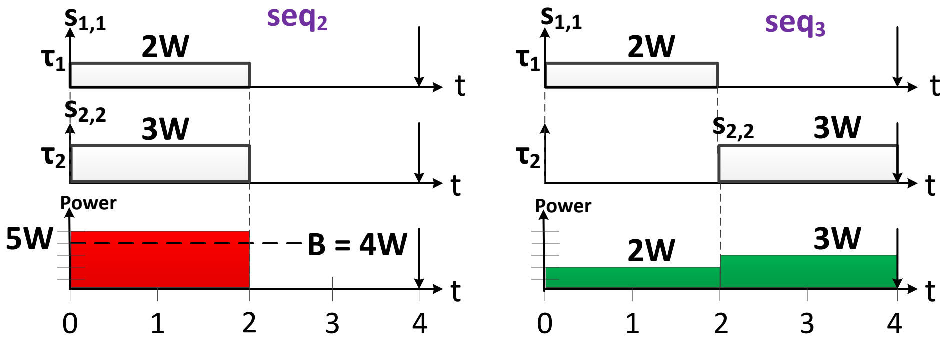


Figure 5.8: *Power dissipation: seq_2 vs seq_3.*

Although the sequence seq_2 ($= fa_1fa_2s_{1,1}s_{2,2}ttc_1c_2 \in L_m(M^1)$) is *resource-constraint* satisfying, it violates the *power-constraint* $\mathcal{B}$. Specifically, the power dissipated during the concurrent execution of τ_1 and τ_2 is $2W + 3W = 5W \nleq \mathcal{B}$ ($= 4W$) (shown in Figure 5.8). Now, consider another sequence $seq_3 = fa_1fa_2s_{1,1}ttc_1s_{2,2}ttc_2 \in L_m(M^1)$. Here, tasks τ_1 and τ_2 are executing in an interleaved fashion and dissipates a maximum power of $3W$ which is less than $\mathcal{B}$ (Figure 5.8). Hence, seq_3 is a *resource* and *power-constraint* satisfying. $\square$

5.2.4 Finding Power-Constraint Satisfying Sequences

In order to remove the sequences that violate power cap $\mathcal{B}$ from $L_m(M^1)$, we develop a state-space search and refinement procedure called *PEAK POWER-AWARE SYN-*

THESIS (PAS). This procedure takes M^1 and $\mathcal{B}$ as inputs and produces the model M^2 that contains all and only the deadline-meeting as well as resource and power-constraint satisfying sequences. This algorithm is essentially based on the idea of Breadth-First Search (BFS) and proceeds as described in Algorithm 7.

ALGORITHM 7: PEAK POWER-AWARE SYNTHESIS

```
Input: $M^1 = (Q, \Sigma, \delta, q_0, Q_m)$, Power cap $\mathcal{B}$
Output: $M^2$
1 Initialize each state $q$ ($\in Q$) of $M^1$ with dissipated power (denoted by $q.DP$) to
  be 0. Start the search operation from the initial state $q_0$ of $M^1$.
2 Find the set of states that can be reached in one transition from the states that
  have been visited.
3 For each newly reached state $q_x$ from $q_y$ (i.e., $\delta(q_y, \sigma) = q_x$), compute dissipated
  power at $q_x$ based on whether $q_x$ is reached on $s_{i,j}$ (the start of execution of $\tau_i$), $c_i$
  (the completion of $\tau_i$), or any other events.
```

$$q_x.DP = \begin{cases} q_y.DP + \mathcal{B}_i & \text{if } \sigma \text{ is start of } \tau_i \\ q_y.DP - \mathcal{B}_i & \text{if } \sigma \text{ is completion of } \tau_i \\ q_y.DP & \text{Otherwise} \end{cases} \tag{5.1}$$

```
4 If $q_x.DP > \mathcal{B}$, then remove power cap violating transition $\delta(q_y, \sigma)$ leading to $q_x$ in
  $M^1$, i.e., $\delta(q_y, \sigma) = \emptyset$. Restore the previous value of $q_x.DP$.
5 Repeat steps (2) to (4) until all states in $M^1$ are visited.
6 Perform reachability operation starting from the initial state $q_0$ of $M^1$
  (transformed) to obtain the set of states that are reachable from $q_0$.
```

Finally, we denote the resulting model consisting of the set of safe reachable states in M^1 as M^2. If the reachability operation does not lead to anyone of the marked states in M^1, then $L_m(M^2)$ becomes empty. This implies that the given task set is not schedulable under the given power-cap $\mathcal{B}$. It may be noted that $L_m(M^2)$ *contains the sequences that satisfy deadlines, resource and power constraints.*

Example (continued): First, let us compute the power-dissipation value at each state of M^1 shown in Figure 5.7 using PAS algorithm. Initialize each state by setting DP to 0 and then progressively update the DP values starting from state 0. Application of BFS from state 0 results in states 1 and 2. The power dissipations at both these states remain as 0, since no task has started execution. Similarly, DP of state 3 remains 0. Now, states 4, 5, 6 and 7 are reached through BFS. For state 4, DP is updated to 2.

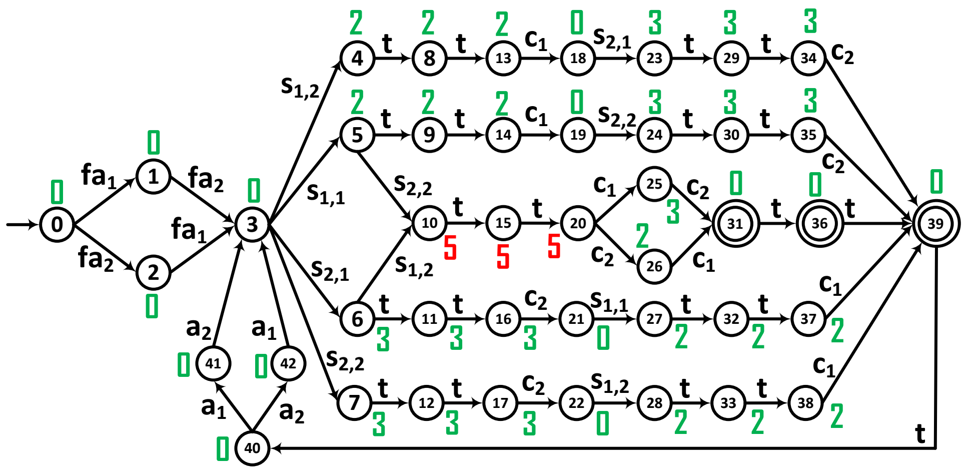


Figure 5.9: *Partial diagram of M^1. Each state is associated with its power dissipation value (shown in green & red colours).*

This is because, the transition $s_{1,2}$ from state 3 to 4 (i.e., $3 \xrightarrow{s_{1,2}} 4$) represents the start of execution of τ_1 which dissipates at most $2W$ of power. Similarly, states 5, 6 and 7, which are reached through transitions $s_{1,1}$, $s_{2,1}$ and $s_{2,2}$, assume DP values of 2, 3 and 3, respectively. During the next iteration, BFS adds states 8, 9, 10, 11 and 12. Here, state 10 is reached either from state 5 or 6 and represents the concurrent execution of τ_1 and τ_2. Hence, step 3 of the PAS algorithm updates the DP value of state 10 to 5. Now, both the transitions $5 \xrightarrow{s_{2,2}} 10$ and $6 \xrightarrow{s_{1,2}} 10$, are deleted by step 4 of PAS, as state 10 violates the power cap of $4W$. However, this deletion has not been actually carried-out to clearly illustrate the cumulative power dissipation values at each state of M^1. We continue to find the DP values corresponding to the remaining states and obtain the final M^1 as shown in Figure 5.9.

Figure 5.10 shows M^2 obtained from M^1 (Figure 5.7) using PAS algorithm. According to this, the transitions leading to *State* 10 will be deleted since they lead to the violation of the power cap. Finally, during the reachability operation, *State* 10 becomes unreachable and discarded from M^2. Hence, the power-constraint violating sequence, $seq_2 = fa_1 fa_2 s_{1,1} s_{2,2} t t c_1 c_2$ is eliminated from $L_m(M^2)$. On the contrary, the *power-constraint satisfying* sequence $seq_3 = fa_1 fa_2 s_{1,1} t t c_1 s_{2,2} t t c_2$ is retained in $L_m(M^2)$.

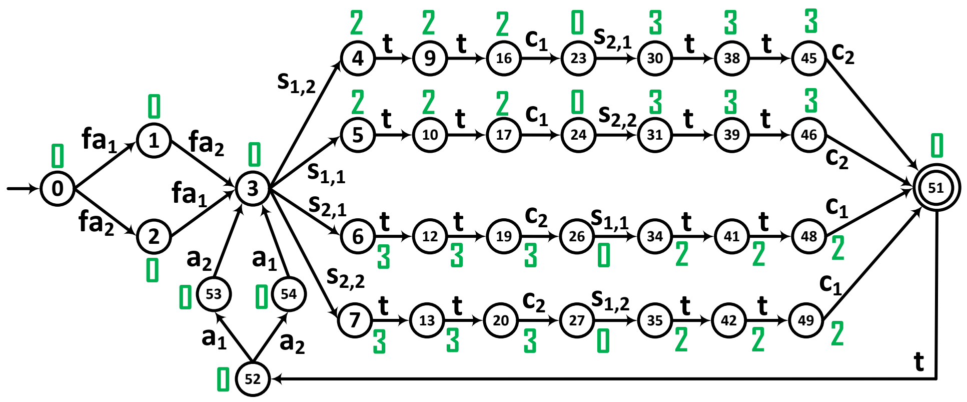


Figure 5.10: M^2 *(partial diagram).*

5.2.5 Peak Power Minimization

Given the set of all feasible sequences (i.e., $L_m(M^2)$), we endeavor to achieve the next obvious design goal: *To determine the subset of scheduling sequences for which peak power dissipation is minimal.* The minimization procedure presented in Algorithm 8 starts by initializing the upper ($\mathcal{B}^{max}$) and lower ($\mathcal{B}^{min}$) bounds on the range of feasible values for the minimal peak power dissipating sequences. While $\mathcal{B}^{max}$ is initialized to the power cap $\mathcal{B}$ itself, the lower bound is set to $\mathcal{B}^{min} = \max_i \mathcal{B}_i$.

ALGORITHM 8: MINIMIZE PEAK-POWER DISSIPATION

```
   Input: $M^2$, $\mathcal{B}^{max}$, $\mathcal{B}^{min}$
   Output: $M^3$
1  while $\mathcal{B}^{min} \leq \mathcal{B}^{max}$ do
2      $\mathcal{B}^{opt} = \lfloor(\mathcal{B}^{min} + \mathcal{B}^{max})/2\rfloor$ ;
3      $M^3$ = PEAK POWER-AWARE SYNTHESIS ($M^2, \mathcal{B}^{opt}$);
4      if $Q$ of $M^3$ is non-empty then
5          $\mathcal{B}^{max} = \mathcal{B}^{opt} - 1$;
6      else
7          $\mathcal{B}^{min} = \mathcal{B}^{opt} + 1$;
```

In order to find the final set of sequences $L_m(M^3)$, that dissipate minimal peak power (denoted by $\mathcal{B}^{opt}$; $\mathcal{B}^{min} \leq \mathcal{B}^{opt} \leq \mathcal{B}^{max}$), we apply the interval bisection based iterative sequence filtering technique as depicted in Algorithm 8. Anyone of the sequences in $L_m(M^3)$ can be used to schedule tasks on-line. Further, the set of feasible scheduling sequences in $L_m(M^3)$ may be filtered to obtain the best schedule with respect to one or a combination of chosen performance parameters such as schedule length, degree of fault-tolerance, resource consumption etc.

Example (continued): For M^2 (in Figure 5.10), $\mathcal{B}^{opt}$ is $3W$ and hence, M^3 remains same as M^2. $\square$

Remark 5.2.1. It may be noted that $L_m(M^3)$ *is optimal containing only and all sequences in* $L_m(T)$ *that satisfy the resource-constraint and dissipate minimal peak power.* Thus, if $L_m(M^3)$ is empty, implying that the given task set is *non-schedulable* for the specified constraints, no other scheme can find a schedulable sequence for the task set.

Remark 5.2.2. Let us consider a task set with a stipulated priority relationship (among all tasks), known a priori. By following our framework, compute M^3 for the given task set. Suppose the state set of the resulting supervisor M^3 is *empty.* Then, remove the subset of the lowest priority tasks from the give task set and compute M^3 again. Repeat this process until M^3 becomes *non-empty.* The resulting non-empty supervisor M^3 contains all feasible scheduling sequences for the maximal number of the highest priority tasks in the given task set.

5.2.6 Complexity Analysis

1. The state-space complexity of the task execution model T_i (in Figure 5.2) may be analyzed as: There are A_i states to measure the arrival time of τ_i. From each state starting with the occurrence of the arrival event a_i and continuing up to the passage of $D_i - E_i$ ticks, m branches emanate based on the events $s_{i,1}, s_{i,2}, ..., s_{i,m}$. Each of these m branches contain E_i states due to the transitions on *tick* events. Hence, the state-space complexity of T_i is $\mathcal{O}(A_i + ((D_i - E_i) \times m \times E_i))$.

2. The state-space complexity of the resource-constraint model RC_i (in Figure 5.5) is $\mathcal{O}(n)$ because it contains distinct states connected to the initial state to represent the execution of each task on the processing core V_i.

3. Given n TDESs T_1, T_2, ..., T_n, an upper bound on the number of states in the composite task execution model T is given by: $\prod_{i=1}^{n} |Q^{T_i}|$, where $|Q^{T_i}|$ is the total

number of states in T_i. Similarly, the total number of states in the composite resource constraint model RC is: $\prod_{i=1}^{n} |Q^{RC_i}|$, where $|Q^{RC_i}|$ is the total number of states in RC_i. Thus, the total number of states in M^0 becomes: $\prod_{i=1}^{n} |Q^{T_i}| \times \prod_{i=1}^{n} |Q^{RC_i}|$.

4. The time-complexity of the safe state synthesis algorithm [104] (for M^1) is *polynomial* in the size of M^0.

5. The computation of M^2 makes use of the *Breadth First Search* approach and hence, time complexity of Algorithm 7 (PAS) is *linear* in the size of M^1.

6. The computation of M^3 makes use of the bisection approach (binary search) to compute the minimum peak power $\mathcal{B}^{opt}$ and hence, time complexity of Algorithm 8 is $log_2(\mathcal{B} - \mathcal{B}^{min})$ times the size of M^2.

It may be noted that the number of states in the composite models T and RC increases *exponentially* as the number of tasks and cores increases. Over the years, *BDD* (*Binary Decision Diagram*) based symbolic synthesis mechanisms have proved to be a key technique towards the efficient computation of large finite state machine models including SCTDES based supervisors. This observation motivated us to derive a symbolic computation based adaptation of the proposed framework.

5.3 Symbolic Computation using BDD

First, we introduce the notion of a *characteristic function.* Given a finite set $\mathcal{U}$, each $u \in \mathcal{U}$ is represented by an unique Boolean vector $\langle z_k, z_{k-1} ..., z_2, z_1 \rangle$, $z_i \in \{0, 1\}, 1 \leq i \leq k$. A subset $\mathcal{W}$ ($\subseteq \mathcal{U}$) is represented by Boolean function $\chi_{\mathcal{W}}$ which maps u onto 1 (respectively, 0) if $u \in \mathcal{W}$ (respectively, $u \notin \mathcal{W}$). Here, $\chi_{\mathcal{W}}$ is the *characteristic function* (interchangeably referred to as *BDD*) of $\mathcal{W}$.

Symbolic representation of TDESs using BDDs: The various components of TDES $G = (Q, \Sigma, \delta, q_0, Q_m)$ are represented symbolically in terms of *BDDs* as follows.

- *BDDs* for Q, q_0, Q_m, Σ of G are $\chi_{G_Q}, \chi_{G_{q_0}}, \chi_{G_{Q_m}}, \chi_{G_\Sigma}$, respectively: Each state $q_i \in Q$ is represented as a unique minterm in a k-variable characteristic function, where $k = \lceil log_2|Q| \rceil$. This is denoted by $\chi_{G_{q_i}}$ and finally, $\chi_{G_Q} = \vee_{i=1}^{|Q|} \chi_{G_{q_i}}$. In a similar manner, *BDDs* for q_0, Q_m, Σ of G can be represented.

- $\delta : Q \times \Sigma \mapsto Q$. Three different characteristic functions are used to represent the source states, events and target states of δ. The BDD representing the transition relation $q_i{}' = \delta(q_i, \sigma)$ is then expressed as $\chi_{G_{\delta_i}} \equiv (q_i = \langle z_k z_{k-1} ... z_1 \rangle) \wedge (\sigma = \langle e_l e_{l-1} ... e_1 \rangle) \wedge (q_i{}' = \langle z'_k z'_{k-1} ... z'_1 \rangle)$ where $k = \lceil log_2|Q| \rceil$ and $l = \lceil log_2|\Sigma| \rceil$. Finally, $\chi_{G_\delta} = \vee_{i=1}^{|\delta|} \chi_{G_{\delta_i}}$.

Steps for the Symbolic Computation of the Supervisor: **Step-1:** TDES models T_i for task execution and RC_i for resource constraint are represented by their corresponding *BDDs* χ_{T_i} and χ_{RC_i}, respectively.
Step-2: Next, BDD for the composite task execution model is computed using *synchronous product* over individual models. In terms of the languages represented by individual models, this synchronous product is defined as: $\cap_{i=1}^{n} P_i^{-1} L(T_i)$ The TDES (say, T'_i) that generates $P_i^{-1} L(T_i)$ can be obtained by adding self-loops for all the events in $\Sigma \setminus \Sigma_i$ at all states of T_i. BDD corresponding to the synchronous product representing the composite task execution model T can be computed as: $\chi_T = \wedge_{i=1}^{n} \chi_{T'_i}$, where $\chi_{T'_i}$ denotes the BDD of TDES model T'_i. A similar transformation is applicable to RC_i and then, χ_{RC} can be computed as: $\chi_{RC} = \wedge_{i=1}^{n} \chi_{RC'_i}$.
Step-3: Now, we symbolically compute $M^0 = T || RC$. In this case, as the event sets of T and RC are same, direct application of the `AND` operation is sufficient to obtain the BDD for M^0, i.e., $\chi_{M^0} = \chi_T \wedge \chi_{RC}$. The next step is to compute the controllable and non-blocking sub-part of χ_{M^0}.
Step-4: BDD χ_{M^1} consisting of the *resource and deadline constraint satisfying* sub-part of χ_{M^0} is computed by applying the symbolic safe state synthesis algorithm, presented in [43].
Step-5: Next, we compute BDD χ_{M^2} consisting of that sub-part of χ_{M^1} in which power dissipation is upper bounded by the power cap $\mathcal{B}$ in all states. For this purpose we use

Algorithm 9, *PEAK POWER-AWARE SYMBOLIC SYNTHESIS* (PASS), the symbolic version of PAS algorithm (Algorithm 7). A step-by-step description of this algorithm is as follows:

Lines 1 to 4 (Step 1 of PAS): Create a look-up table to track dissipated power DP at each state in $\chi_{M_Q^1}$ and initialize the table entries to 0. BDDs $\chi_{Visited}$ (the set of already visited states in $\chi_{M_Q^1}$) and χ_{New} (the set of newly visited states in $\chi_{M_Q^1}$) are initialized to χ_{q_0} and $\emptyset$. BDDs χ_{q_0}, $\chi_{s_{i,j}}$ and χ_{c_i} represent the initial state of $\chi_{M_\delta^1}$, the set of start events $\cup_{i=1}^{n} \cup_{j=1}^{m} s_{i,j}$ and the set of completion events $\cup_{i=1}^{n} c_i$, respectively. Then, Algorithm 9 conducts BFS over $\chi_{M_\delta^1}$ (Lines 5 to 21).

Lines 6 to 9 (Step 2 of PAS): The `Image`$(\chi_{Source}, \chi_{M_\delta^1})$ function returns the set of states in $\chi_{M_\delta^1}$ that can be reached from a state in χ_{Source} through a single transition, by applying two BDD operations: `AND` followed by `exists` ($\exists$). *Line 7*: χ_{New} stores only the set of states that have been visited during current iteration and it is obtained through the conjunction over χ_{Dest} and $\neg\chi_{Visited}$. *Line 8*: χ_{New} is copied to χ_{Source} so that the newly visited states can act as source states in the next iteration. *Line 9*: It appends χ_{New} to $\chi_{Visited}$.

Lines 10 to 21 (Steps 3 and 4 of PAS): These lines compute the power dissipated at each state in χ_{New} and compares it against the power cap $\mathcal{B}$. In case of a power cap violation, the transition due to which this violation occurred, is deleted. The corresponding symbolic computation steps are as follows: For each state in χ_{New} (denoted by χ_{q_x}), the set of transitions leading to State x (denoted by χ_{δ_x}) is computed by $\chi_{q_x}[Z \to Z'] \wedge \chi_{M_\delta^1}$. Here, $\chi_{q_x}[Z \to Z']$ represents change in source and target state variables. Each of these transitions (denoted by χ_{δ_x}) are conjuncted with BDD $\chi_{s_{i,j}}$ (Line 12). If the transition BDD χ_{δ_x} contains any event of type $s_{i,j}$, then the conjunction results in a non-empty output. Consequently, the look-up table entry for $\chi_{q_x}.DP$ is updated as: $\chi_{q_y}.DP + \mathcal{B}_i$ (Line 13). If $\chi_{q_x}.DP > \mathcal{B}$ (Line 14), then the transition χ_{δ_x} is removed from $\chi_{M_\delta^1}$ (Line 15) and $\chi_{q_x}.DP$ is restored to its previous value (Line 16). Supposing that the transition from q_y to q_x is due to the event c_i, $\chi_{q_x}.DP$ becomes $\chi_{q_y}.DP - \mathcal{B}_i$ (Line 18). In case of other events (i.e., a_i, t), $\chi_{q_x}.DP$ remains same (Line 20). The above state search

ALGORITHM 9: PEAK POWER-AWARE SYMBOLIC SYNTHESIS

```
Input: BDD $\chi_{M^1}(= \chi_{M^1_Q}, \chi_{M^1_\Sigma}, \chi_{M^1_\delta}, \chi_{M^1_{q_0}}, \chi_{M^1_{Q_m}}), \mathcal{B}$
Output: BDD $\chi_{M^2}$
1 Create a look-up table to track $DP$ value of each state $\chi_q \in \chi_{M^1_Q}$ and initialize $DP$ to 0 ;
2 Initialize $\chi_{Visited} = \chi_{q_0}$; $\chi_{New} = \emptyset$;
3 Let $\chi_{s_{i,j}}$ be the BDD corresponding to $\cup_{i=1}^{n} \cup_{j=1}^{m} s_{i,j}$;
4 Let $\chi_{c_i}$ be the BDD corresponding to $\cup_{i=1}^{n} c_i$;
5 repeat
6     $\chi_{Dest} \leftarrow$ Image($\chi_{Source}, \chi_{M^1_\delta}$);
7     $\chi_{New} \leftarrow \chi_{Dest} \wedge \neg\chi_{Visited}$;
8     $\chi_{Source} \leftarrow \chi_{New}$;
9     $\chi_{Visited} \leftarrow \chi_{Visited} \vee \chi_{New}$;
10    foreach $\chi_{q_x} \in \chi_{New}$ do
11        foreach $\chi_{\delta_x} \in (\chi_{q_x}[Z \rightarrow Z'] \wedge \chi_{M^1_\delta})$ do
12            if $\chi_{\delta_x} \wedge \chi_{s_{i,j}} \neq \emptyset$ then
13                $\chi_{q_x}.DP = \chi_{q_y}.DP + \mathcal{B}_i$;
14                if $\chi_{q_x}.DP > \mathcal{B}$ then
15                    $\chi_{M^1_\delta} \leftarrow \chi_{M^1_\delta} \wedge \neg\chi_{\delta_x}$;
16                    Restore previous value of $\chi_{q_x}.DP$;
17            else if $\chi_{\delta_x} \wedge \chi_{c_i} \neq \emptyset$ then
18                $\chi_{q_x}.DP = \chi_{q_y}.DP - \mathcal{B}_i$;
19            else
20                $\chi_{q_x}.DP = \chi_{q_y}.DP$;
21 until $\chi_{New} \neq \emptyset$;
22 $i \leftarrow 0$; $\chi_{MQ_i} \leftarrow \chi_{M^1_{q_0}}$;
23 repeat
24     $i \leftarrow i + 1$;
25     $\chi_{MQ_i} \leftarrow \chi_{MQ_{i-1}} \vee$ Image($\chi_{MQ_{i-1}}, \chi_{M^1_\delta}$);
26 until $\chi_{MQ_i} = \chi_{MQ_{i-1}}$;
27 Let resultant BDD consisting of $\chi_{MQ_i}$ be denoted by $\chi_{M^2}$;
28 if $\chi_{MQ_i} \wedge \chi_{M^1_{Q_m}} \neq \emptyset$ then
29     Task set is scheduleable under the power cap $\mathcal{B}$;
30 else
31     Task set is not scheduleable under the power cap $\mathcal{B}$;
32 return $\chi_{M^2}$;
```

operation continues up to the iteration in which χ_{New} becomes empty. This situation implies that all states in $\chi_{M^1_\delta}$ have been visited.

Lines 22 to 31 (Step 5 of PAS): Forward reachability operation over $\chi_{M^1_\delta}$ starting from its initial state is performed (Lines 22 to 26). The resultant BDD $\chi_{M_{Q_i}}$ consists of the set of reachable states in $\chi_{M^1_\delta}$ and is used for the construction of χ_{M^2} (Line 27). It may be noted that BDD $\chi_{M_{Q_i}}$ can be potentially devoid of any marked state in it. To validate this, we check the non-emptiness of the conjunction operation of $\chi_{M_{Q_i}}$ and $\chi_{M^1_{Q_m}}$ (Line 28). Here, non-emptiness ensures that the presence of a marked state in $\chi_{M_{Q_i}}$. Thus, implying the existence of at least single feasible schedule. Otherwise, the task set is non-schedulable under power cap $\mathcal{B}$.

Step-6: Finally using the symbolic representation of Algorithm 8, we obtain BDD χ_{M^3} by extracting that sub-part of χ_{M^2} which contains scheduling sequences dissipating minimal power. This symbolic representation of this procedure remains almost identical to its non-symbolic version (Algorithm 8) with the exception that it now takes χ_{M^2} and produces χ_{M^3} (the BDD versions of M^2 and M^3, respectively).

5.4 Modeling Window Based Power Consumption

The scheduler synthesis framework discussed in the previous sections assumes a single instantaneous peak power bound $\mathcal{B}_i$, for the entire execution span of each task τ_i. This assumption however is not sometimes accurate, in cases where tasks exhibit *phased execution behavior*, so that power consumption characteristics within a phase remains approximately same and distinct from its preceding and succeeding phases [8,67]. For example, let us consider the sampled power dissipation of the *qsort* application from the MiBench benchmark, shown in Figure 5.11. A glance at the power profile in this figure reveals the existence of at least three prominent execution phases with distinct power dissipation characteristics, i.e., phases, $(0, 3150ms]$, $(3150ms, 4050ms]$ and $(4050ms, 5000ms]$ with power caps, $27W$, $34W$ and $23W$, respectively.

Further, the worst-case instantaneous power cap of $34W$ in phase $(3150ms, 4050ms]$ is significantly higher than the power caps, $27W$ and $23W$, for the other two phases.

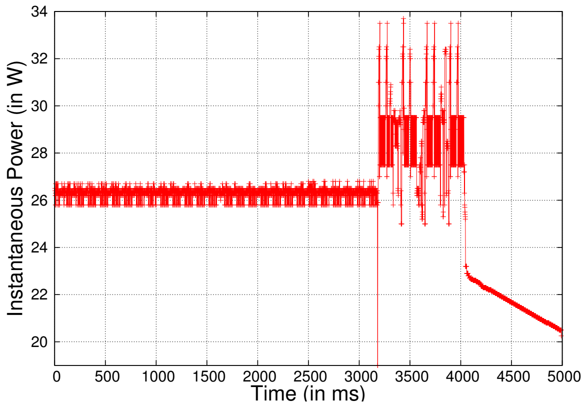


Figure 5.11: *Instantaneous power dissipation of* qsort *[67]*

However, the scheduler synthesis framework presented in the previous sections would assume a single peak power bound of $\mathcal{B}_i = 34W$, for the entire execution span of *qsort*. This assumption would lead to pessimistic system-level power dissipation estimates (at all time instants when *qsort* executes in phases $(0, 3150ms]$ or $(4050ms, 5000ms]$) during the computation of power-constraint satisfying execution sequences from M^2. Due to such pessimistic estimation, execution sequences which are actually power-constraint satisfying may be deemed to be infeasible and consequently, removed from $L_m(M^2)$. In order to address the problem discussed above, the scheduler synthesis framework discussed in Sections 5.2 and 5.3 has been extended to capture phased execution behavior of tasks in: (i) the individual task execution (Figure 5.1) and resource constraint (Figure 5.5) models, (ii) the state-space refinement algorithms PAS and MINIMIZE-PEAK-POWER-DISSIPATION, and (iii) the symbolic computation mechanism.

5.4.1 Window Based Task Execution Model

The execution time E_i of a task τ_i having phased execution behavior, is represented as an ordered sequence of L_i disjoint execution windows $\langle W_{i,1}, W_{i,2}, \ldots, W_{i,L_i} \rangle$. Each window $W_{i,j}$ of τ_i is characterized by an execution duration $E_{i,j}$ and *worst-case instantaneous peak power dissipation* $B_{i,j}$, such that, $\Sigma_{j=1}^{L_i} E_{i,j} = E_i$.

A modified version of the task execution model $T_{i,act}$ that appropriately captures the window based execution of task τ_i, is shown in Figure 5.12. According to this model,

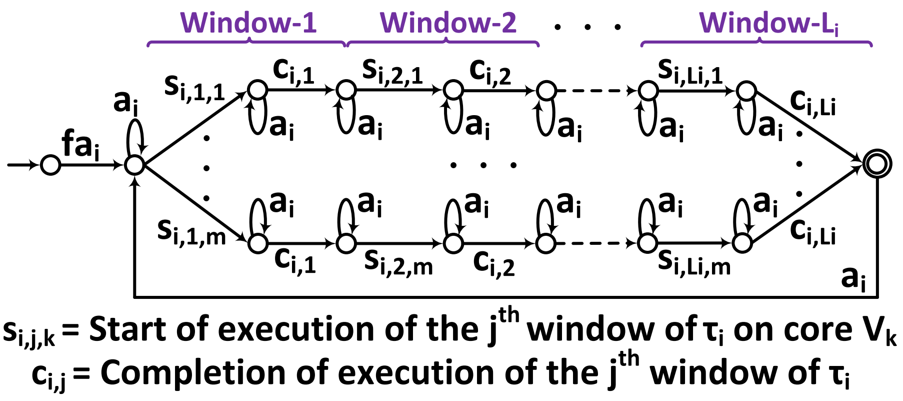

Window-1
Window-2
Window-L$_i$
fa$_i$
a$_i$
s$_{i,1,1}$
c$_{i,1}$
s$_{i,2,1}$
c$_{i,2}$
s$_{i,Li,1}$
c$_{i,Li}$
s$_{i,1,m}$
s$_{i,2,m}$
s$_{i,Li,m}$
s$_{i,j,k}$ = Start of execution of the jth window of τ$_i$ on core V$_k$
c$_{i,j}$ = Completion of execution of the jth window of τ$_i$

Figure 5.12: *Task execution model $T_{i,act}$ for τ_i with L_i windows*

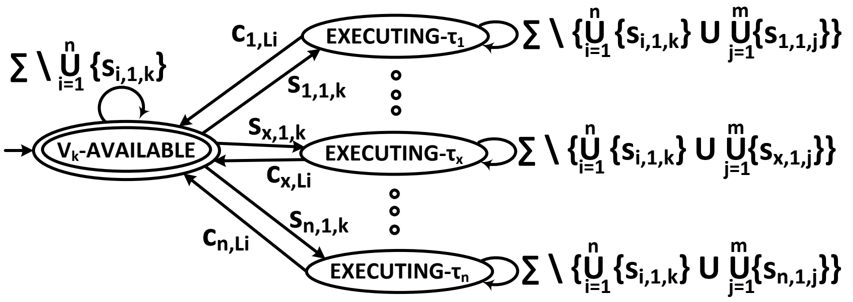
τ_i starts the execution of its 1st window on anyone of the processing cores, subsequent to its arrival. Suppose core V_1 is chosen for execution. This is captured by the event $s_{i,1,1}$, where, the three subscripts denote the task id, window id and processing core id, respectively. The completion of execution of τ_i's first window is denoted by the event $c_{i,1}$. Start of execution of the second window (denoted by $s_{i,2,1}$) happens immediately after the occurrence of $c_{i,1}$ on the same core V_1 to capture the non-preemptive execution of τ_i. After the completion of execution of all L_i windows associated with τ_i, model $T_{i,act}$ reaches the marked activity (denoted by a double circle) and waits for the arrival of the next instance of τ_i. A discussion on the lower (l_σ) and upper (u_σ) time bounds for the newly added events (σ) represented as $(\sigma, l_\sigma, u_\sigma)$, is as follows:

- $(s_{i,1,k}, 0, D_i - E_i)$: $s_{i,1,k}$ must occur between 0 and $D_i - E_i$ ticks from the arrival of τ_i's current instance, to ensure sufficient time that is necessary to complete τ_i's execution before deadline.

- $(c_{i,j}, E_{i,j}, E_{i,j})$: $c_{i,j}$ must occur exactly $E_{i,j}$ ticks from the occurrence of $s_{i,j,k}$. This enforces non-preemptive execution and completion of the execution requirement $E_{i,j}$ of τ_i's j^{th} window.

- $(s_{i,j,k}, 0, 0)$ where, $2 \leq j \leq L_i$: $s_{i,j,k}$ must occur immediately after the completion of τ_i's previous window $W_{i,j-1}$, to model the non-preemptive execution of τ_i.

5.4.2 Window Based Resource-constraint Model



$\Sigma \setminus \bigcup_{i=1}^{n} \{s_{i,1,k}\}$
V_k-AVAILABLE
$c_{1,Li}$
$s_{1,1,k}$
$s_{x,1,k}$
$c_{x,Li}$
$s_{n,1,k}$
$c_{n,Li}$
EXECUTING-τ_1
EXECUTING-τ_x
EXECUTING-τ_n
$\Sigma \setminus \{\bigcup_{i=1}^{n} \{s_{i,1,k}\} \cup \bigcup_{j=1}^{m} \{s_{1,1,j}\}\}$
$\Sigma \setminus \{\bigcup_{i=1}^{n} \{s_{i,1,k}\} \cup \bigcup_{j=1}^{m} \{s_{x,1,j}\}\}$
$\Sigma \setminus \{\bigcup_{i=1}^{n} \{s_{i,1,k}\} \cup \bigcup_{j=1}^{m} \{s_{n,1,j}\}\}$

Figure 5.13: *TDES model RC_k for processing core V_k*

The modified TDES model RC_k for processing core V_k is shown in Figure 5.13. It may be noted that the execution of any task (say, τ_x) is *non-preemptive* and hence, all its windows execute continuously in sequence without any interruption on the same processor until completion. Therefore, RC_k captures only the start of execution of the 1st window ($s_{x,1,k}$) and the completion of execution of the last window (c_{x,L_x}) associated with τ_x. The self-loops at each state (except V_k-AVAILABLE) ensures the following: (1) $\cup_{i=1}^{n}\{s_{i,1,k}\}$: No task is allowed to start the execution of its 1st window on core V_k since τ_x is currently executing on V_k, and (2) $\cup_{j=1}^{m}\{s_{x,1,j}\}$: task τ_x is not allowed to start the execution of its 1st window on any processing core as it has already started its execution on V_k.

5.4.3 Window Based Scheduler Synthesis

Given the individual TDES models for task execution and resource constraint, we can follow the scheduler synthesis framework presented in Section 5.2 to obtain all resource constraint satisfying scheduling sequences, i.e., $L_m(M^1)$. The next step is to filter-out all power-constraint violating sequences from M^1 to obtain M^2 using PAS (Algorithm 7). However, to conduct the search and refinement over the modified version of M^1 using PAS, it must now be appropriately adapted to recognize the phased execution behavior of tasks. Specifically, Eqn. 5.1 in Step 3 of the PAS algorithm should be modified as follows:

$$q_x.DP = \begin{cases} q_y.DP + \mathcal{B}_{i,j} & \text{if } \sigma \text{ is start of } W_{i,j} \text{ of } \tau_i \\ q_y.DP - \mathcal{B}_{i,j} & \text{if } \sigma \text{ is completion of } W_{i,j} \text{ of } \tau_i \\ q_y.DP & \text{Otherwise} \end{cases} \tag{5.2}$$

That is, the chip level power dissipated at a given state q_x, now depends on the power caps $\mathcal{B}_{i,j}$ corresponding to the instantaneously active execution windows $W_{i,j}$ associated with each running task τ_i, at state q_x. It may be noted that during this updation process, the phase oblivious version of the PAS algorithm only considered the worst-case power dissipation bound $\mathcal{B}_i$, instead of $\mathcal{B}_{i,j}$.

The algorithm MINIMIZE PEAK-POWER DISSIPATION (Algorithm 8) remains almost identical for window based scheduler synthesis with one slight modification. Specifically, the initial lower bound on peak power is now given by the maximum power cap over all windows of all tasks, i.e., $\mathcal{B}^{min} = \max_i \max_j \mathcal{B}_{i,j}$, instead of, $\max_i \mathcal{B}_i$, as used earlier. With these modifications, the final resource and power-constraint aware supervisor M^3 for tasks exhibiting phased execution behavior can be obtained.

5.4.4 Complexity Analysis

The state-space complexity of the task execution model T_i (constructed from $T_{i,act}$ in Figure 5.12) is: $\mathcal{O}(A_i + ((D_i - E_i) \times m \times E_i \times L_i))$. With respect to T_i in subsection 5.2.1, the size of the window based task execution model (refer subsection 7.5) has increased by $\mathcal{O}(L_i)$, since the start and completion of each window associated with τ_i has been captured using dedicated states. Given n TDESs T_1, T_2, ..., T_n, an upper bound on the number of states in the composite task execution model T is given by: $\prod_{i=1}^{n} |Q^{T_i}|$, where $|Q^{T_i}|$ is the total number of states in T_i. However, it may be noted that the *state-space of the composite model T for the window based approach will be exponentially larger than its non-window based counter-part, presented in Section 5.2.* Specifically, if each task has L windows, then the size of T may be up to $\mathcal{O}(L^n)$ times larger than the non-window based approach. Similarly, the computational complexities associated with the other models (RC, M^0, M^1 and M^2) also increase with respect to the size of the models presented in Section 5.2.

In order to handle the exponential state-space complexity associated with the synthesis process, we may use the symbolic synthesis scheme presented in Section 5.3. However, it requires the following modifications: (i) (Line No. 3) Compute BDD $\chi_{s_{i,j,k}}$ corresponding to $\cup_{i=1}^{n} \cup_{j=1}^{L_i} \cup_{k=1}^{m} s_{i,j,k}$, (ii) (Line No. 4) Compute BDD $\chi_{c_{i,j}}$ corresponding to $\cup_{i=1}^{n} \cup_{j=1}^{L_i} c_{i,j}$, (iii) (Line No. 12) Replace BDD $\chi_{s_{i,j}}$ by $\chi_{s_{i,j,k}}$, (iv) (Line No. 17) Replace BDD χ_{c_i} by $\chi_{c_{i,j}}$ and (v) (Line Nos. 13 and 18) Replace $\mathcal{B}_i$ by $\mathcal{B}_{i,j}$. These changes ensure that window based execution of a task is correctly captured during the computation of peak power dissipation in the system.

5.5 Experimental Results

The proposed scheduler synthesis schemes presented in Sections 5.2, 5.3 and 5.4 have been evaluated through simulation based experiments using standard real-world benchmark programs (listed in Table 5.3) from *MiBench* [49]. First, we have measured memory and timing overheads by varying (1) number of tasks and processing cores (Experiment 1), and (2) power cap $\mathcal{B}$ (Experiment 2). Next, the performance of our proposed framework has been compared against two state-of-the-art peak power-aware scheduling schemes [67, 78], in terms of (1) peak power dissipation (Experiment 3), and (2) acceptance ratio (Experiment 4). Finally, a comparative evaluation of the scalability of our non window (Section 5.3) and window based (Section 5.4) symbolic scheduler synthesis schemes, has been performed (Experiment 5). Before discussing the results in detail, we first present a description of our task set gneration and simulation setups.

Table 5.3: $\mathcal{B}_i$ *(in watts) for programs in MiBench [67]*

Application	$\mathcal{B}_i$	**Application**	$\mathcal{B}_i$	**Application**	$\mathcal{B}_i$
bitcnts	24	*qsort*	34	*adpcm*	21
basicmath	23	*rijndael*	29	*patricia*	24
dijkstra	23	*sha*	31	*jpeg*	30
fft	24	*susan*	33	*gsm*	28

Task set generation: A detailed discussion on the procedure for measuring the peak power ($\mathcal{B}_i$) of each program is presented in [67] and the results have been listed

Table 5.4: *#Transitions in M^3: TDES vs BDD nodes (varying #processors)*

Model M^3	**#Tasks**											
	2		**4**		**6**		**8**		**10**		**12**	
#Processors	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD
2	360	332	842	591	1824	1292	4146	2461	0	0	0	0
4	796	544	2161	1381	3612	3075	8674	3815	0	0	0	0
6	1645	1109	4803	2693	8593	3239	18421	6139	0	0	0	0
8	3889	2313	10804	4371	17103	5311	34914	13915	0	0	0	0

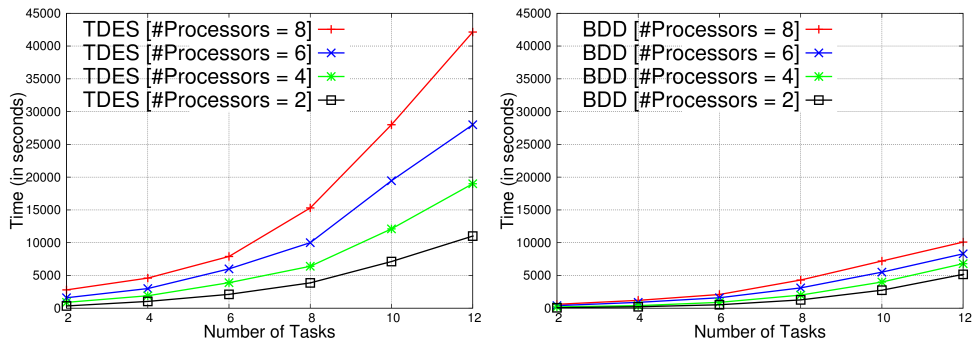
Table 5.5: *#Transitions in M^3: TDES vs BDD nodes (varying power cap)*

Model M^3	**#Tasks**											
	2		**4**		**6**		**8**		**10**		**12**	
Power cap $\mathcal{B}$	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD	TDES	BDD
$85W$	3889	2313	10804	4371	17103	5311	34914	13915	0	0	0	0
$135W$	3889	2313	10804	4371	17103	5311	34914	13915	50351	19105	65673	23163
$165W$	3889	2313	10804	4371	17103	5311	34914	13915	50351	19105	65673	23163

in Table 5.3. These values have been obtained with the help of a combination of the *Wattch* simulator [20] and *McPAT* power simulator [70] for an Intel Xeon processor, 65 nm CMOS technology, operating at 3.4 GHz. With reference to Intel Xeon processor data sheet [1], we have considered the following values for chip level power cap $\mathcal{B}$: $85W$, $135W$ and $165W$.

The program/task arrival times A_i are obtained from an uniform distribution varying within the range 0 to 50. Task execution times (E_i) are randomly generated from normal distributions with mean (μ) and standard deviations (λ) of 25 and 10, respectively. Using the obtained E_i values, we computed deadlines D_i of tasks by varying the ratio E_i/D_i uniformly between 0.1 and 0.5. We assume deadlines to be implicit, i.e., $D_i = P_i$. Different workloads/task set utilizations (i.e., $U = \frac{\Sigma_{i=1}^{n}(E_i/P_i)}{m}$) have been considered.

Simulation setup: The TDES and BDD based scheduler synthesis steps have been implemented using TTCT [2] and CUDD [101], respectively. The computations were performed using a 24 Core Intel(R) Xeon(R) CPU E5-2420 v2 @ 2.2 GHz with 64 GB RAM running linux kernel 2.6.32-042stab113.11. The maximum heap memory used by CUDD was set to 8 GB. Measurement of peak power dissipation for the different approaches has been performed on a simulated system platform based on Intel x86 architecture built using gem5 [17] system level cycle-accurate simulator. The final power profiles have been generated by feeding the execution traces produced by gem5 to McPAT [70] power simulator.



(a) *TDES approach* **(b)** *BDD based symbolic computation*

Figure 5.14: *Time taken for the computation of M^3.*

Experiment 1: ***Varying number of tasks and processing cores***: This experiment *measures the number of transitions present in both the TDES and BDD* based representations of the model M^3 (obtained using the phase oblivious synthesis schemes presented in Sections 5.2 and 5.3), with the objective of evaluating and comparing their relative memory overheads. The power cap $\mathcal{B}$ and task set utilization U have been set to $85W$ and 40%, respectively. We have varied both the number of tasks (from 2 to 12) and cores (from 2 to 8). The results are reported in Table 5.4. The observations are as follows:

- As is obvious, sizes of the models increase as the number of tasks and/or the number of processors grow.
- Comparing sizes of the TDES and BDD based representations, we observe that the BDD models are far more efficient than their corresponding TDES counter-parts.
- The synthesis of M^3 results in an *empty set* for systems with 10 or 12 tasks. This implies that the *given task set is non-schedulable under the given power constraint.*

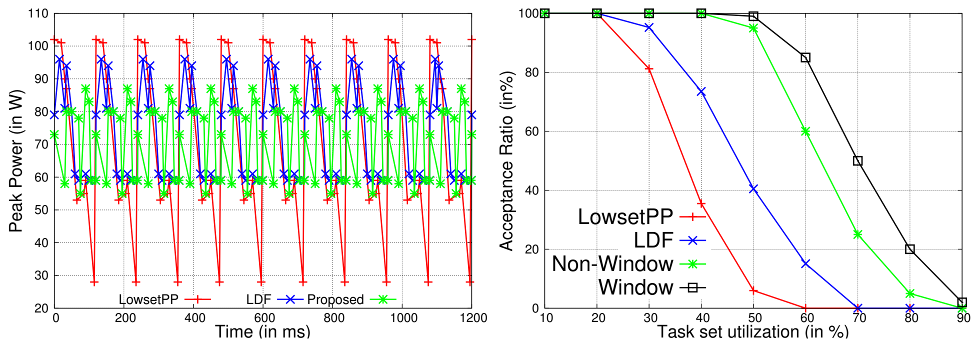
Time taken to synthesize the final model M^3 under the TDES and BDD oriented schemes are reported in Figures 5.14a and 5.14b, respectively. It may be seen that BDD based symbolic computation (Section 5.3) is much faster than its TDES counter-part. For example, considering a system composed of 10 tasks to be scheduled on a 8 core system, our BDD based symbolic synthesis scheme takes $\sim$2 hours to produce a solution. In case of TDES based synthesis, the corresponding value for run time is $\sim$8 hours.

Experiment 2: ***Varying power cap*** $\mathcal{B}$: The number of processing cores and task set utilization U are set to 8 and 40%, respectively. Then, we varied the power cap $\mathcal{B}$ from $85W$ to $165W$ and the number of tasks from 2 to 12. The sizes of the resulting model M^3 (obtained using the phase oblivious synthesis schemes presented in Sections 5.2 and 5.3), is captured in Table 5.5 and the observations are as follows:

- When $\mathcal{B}$ is relaxed to $135W$ or $165W$, system configurations with 10 and 12 tasks which were deemed to be non-schedulable under $\mathcal{B} = 85W$, become schedulable.
- Although, the constraint on peak power dissipation is relaxed, the size of M^3 remains same for schedulable scenarios. This is because M^3 contains only the sequences that satisfy the minimum peak power in it.

Comparison with state-of-the-art schemes: The framework presented in this work has been compared with two other state-of-the-art schemes [67, 78]. In [67], Lee et al. developed a scheduling algorithm called *LowestPP* that attempts to minimize peak power dissipation while satisfying all timing constraints of a given set of real-time tasks. This scheme logically divides the cores in a chip into groups of two cores. Then for each group, it attempts to schedule tasks using a fixed priority scheduling scheme such that peak power dissipation is minimized. Munawar et al. [78] presented a scheme called *Least Density First* (LDF) to minimize peak power usage for frame-based and periodic real-time tasks. Here, all tasks in a frame have the same deadline. This scheme starts by scheduling the highest power consuming task at the time slot in which power dissipation is minimal, and this process is repeated for all tasks. Now, we compare the performance of our scheme with *LowestPP* [67] and *LDF* [78].

Experiment 3: ***Comparing peak power dissipation***: In this experiment, the number of processing cores, tasks, task set utilization and power cap, have been fixed at 4, 12, 40% and $165W$, respectively. For fair comparison with other schemes, we have considered a frame-based periodic task set with the frame duration being $120ms$. Also,



(a) *Peak power dissipation* **(b)** *Acceptance ratio*

Figure 5.15: *Comparing with state-of-the-art schemes.*

we consider only the phase oblivious synthesis scheme presented in Section 5.3, since none of the two approaches in [67, 78] consider phased execution behavior of tasks. We have measured the peak power dissipation under each scheduling scheme and reported it in Figure 5.15a. The maximum peak power dissipated under *LowestPP* [67], *LDF* [78] and the proposed scheme are $102W$, $96W$ and $87W$, respectively. Being *optimal* in nature, our scheme is more efficient towards minimizing peak power compared to the other two state-of-the-art approaches. Specifically, our scheme is able to carefully distribute the execution of tasks over time such that chip-level peak power is minimized.

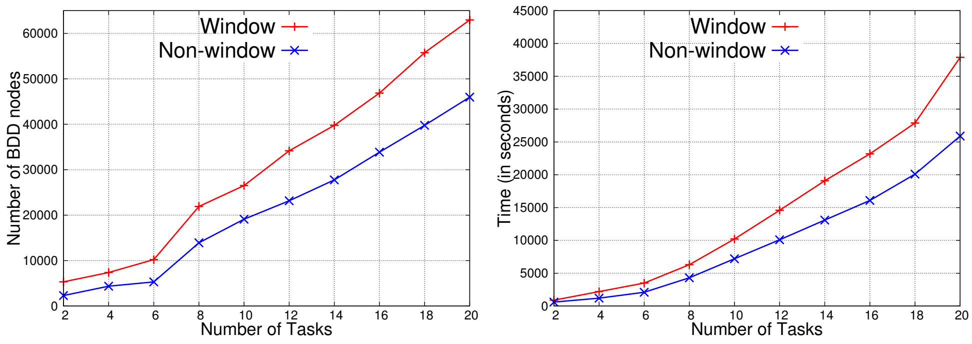
Table 5.6: *Phased execution of programs in MiBench*

Application	$\lvert W_i\rvert$, $\langle \mathcal{B}_{i,1}, \ldots, \mathcal{B}_{i,\lvert W_i\rvert} \rangle$	**Application**	$\lvert W_i\rvert$, $\langle \mathcal{B}_{i,1}, \ldots, \mathcal{B}_{i,\lvert W_i\rvert} \rangle$
bitcnts	2, $\langle 17, 24 \rangle$	*qsort*	3, $\langle 27, 34, 23 \rangle$
basicmath	2, $\langle 15, 23 \rangle$	*rijndael*	4, $\langle 29, 20, 24, 27 \rangle$
dijkstra	3, $\langle 20, 15, 23 \rangle$	*sha*	3, $\langle 31, 28, 20 \rangle$
fft	4, $\langle 15, 21, 24, 12 \rangle$	*susan*	4, $\langle 25, 28, 33, 20 \rangle$
adpcm	2, $\langle 21, 12 \rangle$	*patricia*	3, $\langle 21, 18, 24 \rangle$
jpeg	3, $\langle 25, 30, 22 \rangle$	*gsm*	2, $\langle 22, 28 \rangle$

Experiment 4: ***Measuring Acceptance Ratio***: The number of cores, tasks and power cap have been fixed at 8, 25 and $165W$, respectively. Then, we have varied the task set utilization from 10% to 90%. For each system configuration, we have measured the *acceptance ratio*, i.e., the ratio of the number of task sets that have been deemed to be schedulable against the number of task sets that have been presented to

the system, under pre-specified resource and power constraints. It may be observed from Figure 5.15b that the proposed schemes (both phase oblivious (non-window based) and phase-aware (window based) task execution) are able to achieve significantly better acceptance ratios compared to the two other existing schemes. This is because, our scheme is able to explore all possible task-to-core assignment alternatives while satisfying resource, timing and power constraints.

In order to evaluate the performance of the *window based synthesis scheme* (Section 5.4), we have divided the execution durations of the benchmark programs listed in Table 5.3 into multiple windows (according to their power profiles) such that each window $W_{i,j}$ is associated with the distinct power cap $\mathcal{B}_{i,j}$. The number of windows ($|W_i|$) along with their corresponding power values ($\mathcal{B}_{i,1}, \mathcal{B}_{i,2}, \ldots, \mathcal{B}_{i,|W_{i,j}|}$) associated with each program, is listed in Table 5.6. From Figure 5.15b, it can be seen that the acceptance ratios of the tasks under window based approach is much better than their non-window counter-parts (Section 5.3). Specifically, the non-window based approach assumes a single conservative peak power estimate for the entire execution duration of tasks which consequently leads to lower acceptance ratios. On the contrary, the window based approach with its ability to model phased power dissipation behavior of tasks, is far more accurate in estimating system level power consumption at any time and hence, deliver better acceptance ratios.



(a) *Number of BDD nodes* **(b)** *Computation time*

Figure 5.16: *Comparing non-window and window based synthesis.*

Experiment 5: ***Scalability***: In this experiment, we evaluate the scalability of the proposed symbolic computation approach by increasing the number of tasks in the system. For this purpose, we have included 8 additional benchmark programs namely, *blowfish*, *pgp*, *lame*, *mad*, *ispell*, *stringsearch*, *ghostscript* and *rsynth*, also from MiBench. Using this updated taskset and a multi-core platform consisting of 8 cores with the power cap $\mathcal{B}$ of 165W, we have computed the total number of states and the time taken to compute M^3 using non-window based (Section 5.3) and window (Section 5.4) based approaches. The experimental results are presented in Figures 5.16a and 5.16b. It may be observed that the proposed symbolic computation schemes are able to compute the final model M^3 in reasonable amounts of time with moderate state space overheads. For example, considering a system composed of 10-tasks with mean execution times around 20 time units (where the exact duration of a time unit is a tunable design parameter), to be scheduled on a 8 core system, our non-window based scheme takes around ∼2 hours to produce a solution. The final supervisor consisting of all feasible solutions consume upto ∼19000 BDD nodes (assuming each node to be about 3 bytes in size, the total space required becomes ∼56KB). In case of window based synthesis, the corresponding values for run time is around ∼3 hours while the space required by the final supervisor is about ∼77KB. Thus, it may be seen that the window based approach consumes significantly higher overheads compared to the non-window based scheme.

5.6 Summary

In this chapter, we have presented a systematic way of synthesizing an *optimal scheduler* for non-preemptive real-time tasks on multi-core systems with pre-specified chip level peak power constraints. The synthesis process started with the development of the execution models for tasks and resource-constraint models for processing cores. Composition over these models ultimately provided the deadline and resource constraint satisfying supervisor. Further, the power-constraint violating sequences are filtered-out from the initial supervisor through a search and refinement mechanism. Subsequently, the targeted supervisor containing only scheduling sequences that dissipate minimal power is retained. To control state-space complexity involved in the synthesis process, a BDD based computation flow has been designed corresponding to the TDES oriented construction steps. Finally, we presented the experimental evaluation of our proposed framework using real-world benchmark programs. This framework can be extended to consider the execution of sporadic tasks. Specifically, the time bound associated with the event a_i can be set as (a_i, P_i, ∞) to model the minimum inter-arrival time constraint P_i associated with the sporadic task τ_i. In the next chapter, we consider the scheduler synthesis for a set of real-time task executing on a heterogeneous multi-core platform.

Chapter 6

Scheduling on Heterogeneous Multi-cores

In the earlier chapters, we have assumed the processing cores in a multi-core platform to be *identical* (i.e., *homogeneous*). However, the nature of processors in embedded systems is changing over the years. To handle specific tasks, specialized processing cores are now integrated on to a single hardware platform. For example, today we have *heterogeneous* multi-core platforms with specialized *Graphical Processing Unit* (GPU), *Single Instruction-stream Multiple Data-stream* (SIMD) accelerators, *Digital Signal Processor* (DSP), specialized floating-point units, integrated on a single die. *On a heterogeneous platform, the same piece of code (i.e., task) may require different amounts of time to execute on different processors* [10]. For example, a task responsible for rendering images may take far less time to execute on a GPU compared to a *General Purpose Processor* (GPP), while a number-crunching routine would execute more efficiently on a GPP.

In this chapter, we consider the optimal scheduling of a set of real-time non-preemptive tasks executing on a heterogeneous multi-core. The general problem of the optimal scheduling of non-preemptive tasks is intractable and requires exhaustive enumeration of the entire solution space [12, 12, 23, 27, 27, 28, 28, 54, 93, 110]. Such exhaustive enumeration methodologies incur exponential overheads, and therefore are prohibitively expensive to be applied on-line. Therefore on-line approaches, which often do not allow the liberty of exhaustive enumeration of the solution space, tend to be sub-optimal and heuristic in nature. So, off-line techniques are often preferred over on-line techniques for solving such problems. Hence, there is a necessity for developing *off-line* exhaustive enumeration techniques to pre-compute schedules at design time and use them during on-line execution. As mentioned earlier in this dissertation, we apply supervisory control approach for scheduler synthesis. Although in recent years, there has been a few significant works dealing with real-time scheduling using supervisory control, this is possibly the first work which addresses the scheduler synthesis problem for non-preemptive periodic tasks.

6.1 Related Works

In recent years, researchers have shown that off-line formal approaches such as Supervisory Control of Timed Discrete Event Systems (SCTDES) [18] can be used to synthesize optimal schedulers for real-time systems [29] [53] [87] [106] [105] . Chen and Wonham presented a scheduler design scheme for *non-preemptive, periodic tasks* on uniprocessors [29]. Later, Janarthanan et al. extended the supervisory control framework for the priority-based scheduling of *preemptive periodic tasks* [53]. Recently, Wang et al. enhanced the models presented in [29] [53] to schedule *non-preemptive, periodic tasks with multiple periods* [105]. Further, Wang et al. proposed an approach for *priority-free, conditionally-preemptive, real-time scheduling of periodic tasks* [106]. It may be noted that none of these SCTDES based scheduler synthesis schemes can model the variation in execution times of a particular task on different processors, in a heterogeneous platform. To address this, we develop in this work task execution and resource-constraint models which can be used to synthesize optimal off-line schedulers applicable to heterogeneous multi-core platforms. Further, we present a working example to illustrate the scheduler synthesis mechanism designed here. Table 6.1 synopsizes a qualitative comparison among the SCTDES based scheduling approaches.

6.2 Proposed Scheduler Synthesis Scheme

System Model: Consider a real-time system consisting of a set ($\{\tau_1, \tau_2, ..., \tau_n\}$) of n non-preemptive periodic tasks to be scheduled on m heterogeneous multi-cores ($\{V_1, V_2, \ldots, V_m\}$).

Table 6.1: *Comparison between SCTDES based scheduling schemes*

Method	Tasks	Preemptive / Non-preemptive	Uniprocessor / Multi-core	Remarks
[29]	Periodic	Non-preemptive	Uniprocessor	It does not consider task priorities
[53]	Periodic	Both	Uniprocessor	It considers task priorities
[87]	Periodic & Sporadic	Preemptive	Uniprocessor	It does not consider inter-arrival time constraint
[39]	Aperiodic & Sporadic	Non-preemptive	Uniprocessor	It correctly captures inter-arrival time constraint
[105]	Periodic	Preemptive	Uniprocessor	It supports tasks with multiple periods to allow reconfiguration
[106]	Periodic	Conditionally preemptive	Uniprocessor	It can be extended to homogeneous multi-cores
[107]	Periodic & Sporadic	Conditionally preemptive	Uniprocessor	It can be extended to homogeneous multi-cores
Present work	Periodic	Non-preemptive	Heterogeneous multi-core	Correctly models execution of tasks on heterogeneous multi-cores

A periodic task τ_i consists of an infinite sequence of instances and is represented by a 4-tuple $\langle A_i,\ \langle E_{i,1}, E_{i,2}, \ldots, E_{i,m}\rangle,\ D_i,\ P_i\rangle$, where $A_i (\in \mathbb{N})$ is the *arrival time* of the first instance of τ_i (relative to system start), $E_{i,j} (\in \mathbb{N}$ and $j = 1, 2, ..., m)$ is the *execution time of τ_i on the V_j^{th} processing core*, $D_i (\in \mathbb{N})$ is the *relative deadline* and $P_i (\in \mathbb{N})$ denotes the fixed inter-arrival time between consecutive instances of τ_i [10].

Assumptions: (1) Tasks are independent. (2) Execution time for each task is constant and equal to its *Worst Case Execution Time* (WCET) [73]. (3) The tasks are periodic and have fixed interval arrival times. (4) $E_{i,j}$, D_i and P_i are *discrete* and *finite*.

6.2.1 Task Execution Model (ATG)

The ATG model $PT_{i,act}$ for executing a *non-preemptive* periodic task τ_i on a heterogeneous multi-core system is shown in Figure 6.1. $PT_{i,act} = (A_i^{PT}, \Sigma_{i,act}, \delta_{i,act}^{PT}, a_{i,0}^{PT}, A_{i,m}^{PT})$, where, $A_i^{PT} = \{$IDLE, READY, READYQ1, ..., READYQm, EXECUTING-ON-V_1, $\ldots$, EXECUTING-ON-V_m, COMPLETION$\}$, $\Sigma_{i,act} = \{fa_i,\ a_i,\ r_{i,1},\ \ldots,\ r_{i,m},\ s_{i,1},\ \ldots,\ s_{i,m},\ c_{i,1},\ \ldots,\ c_{i,m}\}$ (the description of events is presented in Table 6.2), $a_{i,0}^{PT}$ = IDLE and $A_{i,m}^{PT} = \{$COMPLETION$\}$. Here, $\Sigma_{act} = \cup_{i=1}^{n} \Sigma_{i,act}$, which is categorized as: $\Sigma_{spe} = \Sigma_{act}$ (since all events have finite time bounds), $\Sigma_{rem} = \emptyset$, $\Sigma_{unc} = \{\cup_{i=1}^{n}\{fa_i, a_i\}\} \cup \{\cup_{i=1}^{n} \cup_{j=1}^{m} \{c_{i,j}\}\}$ (task arrival and completion events cannot be prevented by supervisor), $\Sigma_{con} = \cup_{i=1}^{n} \cup_{j=1}^{m} \{r_{i,j}, s_{i,j}\}$ (assignment on the ready queue and start of execution can be controlled by supervisor), $\Sigma_{for} = \Sigma_{con}$ (all controllable events can preempt the occurrence of $tick$), $\Sigma = \Sigma_{act} \cup \{tick\}$.

Table 6.2: *Description of events*

Event	Description
fa_i	Arrival of task τ_i's first instance
a_i	Arrival of task τ_i's next instance
$r_{i,j}$	Assignment of task τ_i on the ready queue associated with V_j
$s_{i,j}$	Start of the execution of task τ_i on V_j
$c_{i,j}$	Completion of the execution of task τ_i on V_j

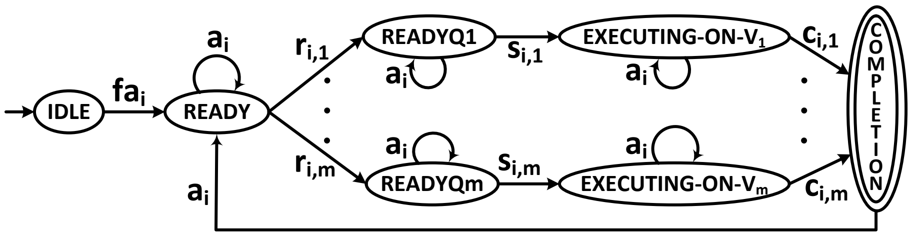


Figure 6.1: *Task execution model $PT_{i,act}$ for periodic task τ_i*

Initially, $PT_{i,act}$ stays at activity *IDLE* until the arrival of τ_i's first instance. On the occurrence of fa_i, $PT_{i,act}$ transits to activity *READY*. At this activity, there are m outgoing transitions on events $r_{i,j}$ ($j = 1, 2, ..., m$) to model the possibility of assigning of τ_i on the ready queue associated with core V_j. Once event $r_{i,j}$ is executed, $PT_{i,act}$ moves to activity *READYQj* and stays at the ready queue of V_j until the start of τ_i's execution on V_j. On $s_{i,j}$, $PT_{i,act}$ moves to activity *EXECUTING-ON-V_j* to capture the execution of τ_i on V_j. After the completion of execution, $PT_{i,act}$ transits to the marker activity *COMPLETION* on $c_{i,j}$. Next, $PT_{i,act}$ moves back to activity *READY* on the arrival of τ_i's next instance (a_i) and τ_i continues its execution in a similar manner. The self-loops on a_i are used to model the periodicity of τ_i. An elaboration on the modeling of τ_i's periodicity is presented separately later in this section.

To obtain a TDES model, we assign time bounds for events in $\Sigma_{i,act}$, as follows: 1. (fa_i, A_i, A_i): fa_i must occur exactly at the A_i^{th} tick from the system start, to correctly model the arrival of τ_i's first instance. 2. (a_i, P_i, P_i): a_i must occur exactly at the P_i^{th} tick from the arrival of τ_i's previous instance, to model the periodic arrival of τ_i. 3. $(r_{i,j}, 0, 0)$: $r_{i,j}$ must occur immediately after the arrival of τ_i's current instance. 4.

$(s_{i,j}, 0, D_i - E_{i,j})$: $s_{i,j}$ must occur between 0 and $D_i - E_{i,j}$ ticks from the arrival of τ_i's current instance to ensure that there is sufficient time for the execution of τ_i on V_j so that its deadline can be met. 5. $(c_{i,j}, E_{i,j}, E_{i,j})$: $c_{i,j}$ must occur exactly $E_{i,j}$ ticks from the occurrence of $s_{i,j}$. This is to enforce the execution requirement $E_{i,j}$ of τ_i on V_j.

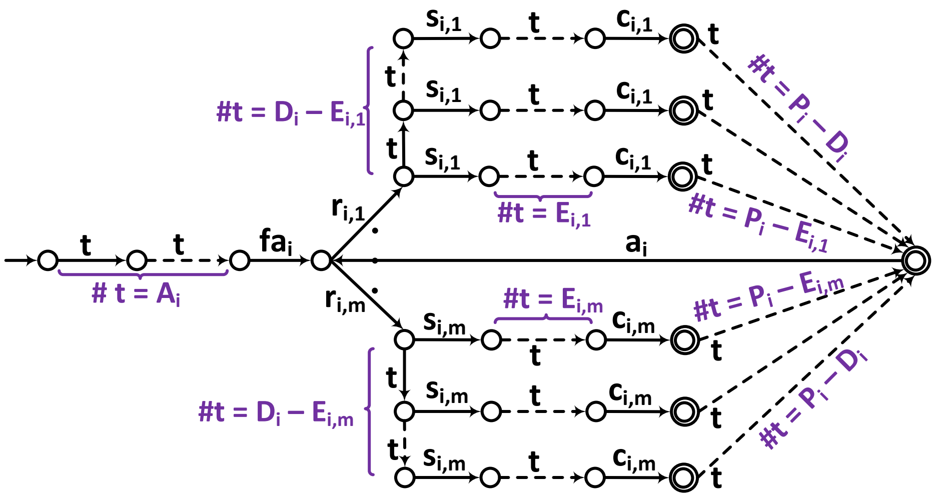


Figure 6.2: *TDES model PG_i for periodic task $\tau_i\langle A_i, E_i, D_i, P_i\rangle$*

6.2.2 Task Execution Model (TDES)

The TDES model PG_i obtained using $PT_{i,act}$ and the above time bounds, is shown in Figure 6.2. Here, t represents the *tick* event. In the TTCT software [2], construction of TDES from ATG is implemented by the procedure `timed_graph`. It may be observed (from Figure 6.2) that the arrival of τ_i's first instance (i.e., fa_i) occurs at A_i^{th} tick from system start. Then, τ_i is immediately assigned to the ready queue associated with any one of the processing cores. Let us assume that τ_i is assigned to the ready queue of V_1 through $r_{i,1}$. At this instant, τ_i may either be allowed to immediately start its execution through $s_{i,1}$ or τ_i's execution may be delayed by at most $D_i - E_{i,1}$ ticks. Subsequent to start of execution, τ_i consumes exactly $E_{i,1}$ ticks to complete on V_1 and this is marked by the event $c_{i,1}$. Suppose, τ_i started its execution without any delay. In this case, PG_i contains exactly $P_i - E_{i,1}$ ticks between the completion event $c_{i,1}$ and the arrival of τ_i's next instance. In case of a delay of $D_i - E_{i,1}$ ticks, this interval reduces to exactly $P_i - D_i$

ticks.

Modeling Periodicity of τ_i: Periodicity of τ_i requires the arrival events (a_i's) of any two consecutive instances of τ_i to be separated by exactly P_i ticks. This is achieved by setting P_i as both the lower and upper time bounds of a_i. Additionally, the TDES model PG_i must accurately account for the time elapsed from the arrival of the current instance of τ_i, at all subsequent states of the model, so that the next arrival can be enforced to occur exactly after P_i ticks. However, for the TDES model to correctly implement this behavior, the arrival event a_i must remain enabled (and hence, should be defined) at all activities subsequent to the occurrence of τ_i's first instance, in the ATG $PT_{i,act}$ from which PG_i is constructed. For this purpose, self-loops on a_i has been added at all activities in $PT_{i,act}$ except IDLE (where the first instance of τ_i has not yet arrived) and COMPLETION (where a_i is already defined). On the occurrence of fa_i, $PT_{i,act}$ transits from *IDLE* to the activity *READY* at which events a_i, $r_{i,1}$, $\ldots$, $r_{i,m}$ become enabled with associated timer values $t_{a_i} = P_i$, and $t_{r_{i,j}} = 0$ ($j = 1, \ldots, m$). On all activities reachable from *READY* to *COMPLETION*, timer t_{a_i} is continuously decremented at each clock tick (and not reset to P_i, as a_i is enabled at these activities). After $PT_{i,act}$ reaches *COMPLETION*, say x ticks ($t_{a_i} = P_i - x$) from the arrival of τ_i's current instance, t_{a_i} is continued to be decremented until it reduces to 0. Here, x captures the delay in the start of execution (i.e., $(s_{i,j}, 0, D_i - E_{i,j})$) along with the time taken for the execution of τ_i (i.e., $(c_{i,j}, E_{i,j}, E_{i,j})$). Hence, the value of x is lower and upper bounded by $E_{i,j}$ and D_i, respectively. When t_{a_i} reaches zero, occurrence of a_i becomes eligible and consequently, $PT_{i,act}$ transits from *COMPLETION* to *READY* on a_i with t_{a_i} being reset to P_i. Thus, the periodicity of τ_i is correctly captured by $PT_{i,act}$.

6.2.3 Composite Task Execution Model

It may be inferred that $L_m(PG_i)$ satisfies (i) distinct execution times of τ_i on different processing cores, (ii) deadline requirement and (iii) fixed inter-arrival time constraints of τ_i. Given n individual TDES models PG_1, ..., PG_n corresponding to τ_1, $\ldots$ τ_n, a *synchronous product* $PG = PG_1 || ... || PG_n$ on the models gives us the composite model for all the tasks executing *concurrently*. In the *TTCT* software [2], *synchronous* *product* is computed using the procedure `sync`. As individual models (PG_i's) do not share any common event (i.e., $\cap_{i=1}^{n}\Sigma_{i,act} = \emptyset$) except *tick*, all models synchronize only on the *tick* event. Since, the individual models satisfy deadline and fixed-inter arrival time constraints, $L_m(PG)$ also satisfies them. However, the sequences in $L_m(PG)$ *may violate resource constraints*. That is, PG does not restrict the concurrent execution of multiple tasks on the same processing core. Hence, we develop a resource-constraint model to capture the following specification: *Once a task τ_i is allocated onto a processing core V_j, it remains allocated on V_j until its completion. Meanwhile, no other task τ_j ($\neq \tau_i$) is allowed to execute on V_j*. This is captured by the TDES model RC_k, as we discuss next.

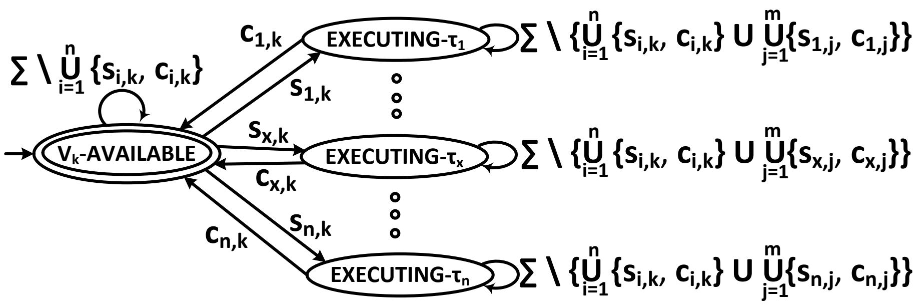

$\Sigma \setminus \cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\}$
V_k-AVAILABLE
$c_{1,k}$
$s_{1,k}$
$s_{x,k}$
$c_{x,k}$
$s_{n,k}$
$c_{n,k}$
EXECUTING-τ_1
EXECUTING-τ_x
EXECUTING-τ_n
$\Sigma \setminus \{\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\} \cup \cup_{j=1}^{m}\{s_{1,j}, c_{1,j}\}\}$
$\Sigma \setminus \{\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\} \cup \cup_{j=1}^{m}\{s_{x,j}, c_{x,j}\}\}$
$\Sigma \setminus \{\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\} \cup \cup_{j=1}^{m}\{s_{n,j}, c_{n,j}\}\}$

Figure 6.3: *TDES model RC_k for processing core V_k*

6.2.4 Resource-constraint Model

The TDES model RC_k for a core V_k is shown in Figure 6.3. $RC_k = (Q, \Sigma, \delta, q_0, Q_m)$, where, $Q = \{V_k\text{-AVAILABLE, EXECUTING-}\tau_1, \ldots, \text{EXECUTING-}\tau_n\}$, $\Sigma = \Sigma_{act} \cup \{t\}$, $q_0 = Q_m = V_k$-AVAILABLE. RC_k contains $n+1$ states to capture the idle state of V_k as well as execution of anyone of the n tasks on V_k. The self-loop $\Sigma \setminus \cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\}$ at the initial state, allows the possibility of executing all events in Σ except $\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\}$, i.e., it disallows the start and completion of any task on V_k. The start of execution of any task, say τ_x, on V_k is modeled by an outgoing transition on $s_{x,k}$ to the state EXECUTING-τ_x from V_k-AVAILABLE. At this state, the self-loop $\Sigma \setminus \{\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\}$ $\cup \cup_{j=1}^{m}\{s_{x,j}, c_{x,j}\}\}$ allows all but the events related to, (i) starts or completions of any task on V_k ($\cup_{i=1}^{n}\{s_{i,k}, c_{i,k}\}$) and (ii) start or completion of task τ_x on any core ($\cup_{j=1}^{m}\{s_{x,j}, c_{x,j}\}$). On completion, RC_k transits back to the initial state on $c_{x,k}$ from EXECUTING-τ_x.

Hence, $L_m(RC_k)$ *ensures the exclusive execution of a single task on core* V_k *at any time instant.* To summarize, $L_m(RC_k)$ contains all sequences over Σ^* excluding those which allow the concurrent execution of multiple tasks on V_k. Thus, $L_m(RC_k)$ is the minimally restrictive language which satisfies the resource-constraint with respect to V_k.

Composite Resource-constraint Model: Given m TDES models $RC_1, \ldots, RC_m$ corresponding to the m cores, we can compute the composite resource-constraint model RC as: $RC_1 || \ldots || RC_m$. Hence, $L_m(RC)$ represents the minimally restrictive language that disallows the concurrent execution of multiple tasks on the same processing core. Although, all sequences in $L_m(RC)$ satisfy the resource-constraints, however, sequences in $L_m(RC)$ may not correctly capture timing properties such as execution times, deadlines and periods associated with the tasks (which is captured by PG).

6.2.5 Supervisor Synthesis

In order to find only and all sequences in $L_m(PG)$ that satisfy resource-constraints, the initial supervisor $S_0 = PG||RC$, is computed. We first show that $L_m(S_0)$ contains resource-constraint satisfying sequences of $L_m(PG)$ and then, we show that $L_m(S_0)$ contains timing-constraint satisfying sequences of $L_m(PG)$.

Theorem 6.2.1. *$L_m(S_0)$ contains only and all resource-constraint satisfying sequences of $L_m(PG)$.*

Proof. ($\Rightarrow$): $L_m(S_0) = L_m(PG) \cap L_m(RC)$, since $S_0 = PG||RC$ and Σ is same for both PG and RC. So, the following inference may be drawn: a string $s \in L_m(S_0)$ implies that $s \in L_m(PG)$ and $s \in L_m(RC)$. Since $L_m(RC)$ satisfies resource-constraints, $L_m(S_0)$ $(\subseteq L_m(RC))$ *contains only resource-constraint satisfying sequences of* $L_m(PG)$.

($\Leftarrow$): By contradiction: Let $s \in L_m(PG)$ is resource-constraint satisfying, but $s \notin L_m(S_0)$. As s is resource-constraint satisfying, it must be part of $L_m(RC)$, the minimally restrictive language that disallows the concurrent execution of multiple tasks on the same processing core. Hence, $s \in L_m(S_0)$ since $S_0 = PG||RC$. This contradicts the assumption. Thus, $L_m(S_0)$ *contains all resource-constraint satisfying sequences of* $L_m(PG)$. □

Corollary 6.2.1. *$L_m(S_0)$ contains only and all resource as well as timing constraints satisfying sequences of $L_m(PG)$.*

Proof. We know that all sequences in $L_m(PG)$ satisfy timing constraints and $L_m(S_0) = L_m(PG) \cap L_m(RC)$. By following of the proof structure of Theorem 1, we can show that $L_m(S_0)$ contains only and all resource as well as timing constraints satisfying sequences of $L_m(PG)$. □

It may be noted that sequences in $L(S_0)$ which violate resource and/or timing constraints are not part of $L_m(S_0)$ and hence, such sequences lead to *deadlock* states in S_0. This implies $L(S_0) \neq L_m(S_0)$, i.e., S_0 is *blocking*. However, to ensure that all instances of all periodic tasks meet their resource and timing constraints, S_0 must be made controllable with respect to PG. This is achieved by determining sup$C(L_m(S_0))$, the controllable and non-blocking sub-part of S_0. Here, sup$C(L_m(S_0))$ is essentially obtained by disabling certain controllable events ($r_{i,j}$, $s_{i,j}$) at appropriate states in S_0, such that none of the deadlock states are reached. In TTCT, this can be computed using the `supcon` procedure. This guarantees that the closed-loop system behavior always stays within the desired behavior sup$C(L_m(S_0))$.

Theorem 6.2.2. The language sup$C(L_m(S_0))$ is the largest schedulable language.

Proof. Let us consider the set of all sublanguages of $L_m(S_0)$ that are controllable with respect to $L_m(PG)$, i.e., $C(L_m(S_0)) := \{L_m(S_0') \subseteq L_m(S_0) \mid L_m(S_0')$ is controllable with respect to $L_m(PG)\}$. Among all sublanguages in $C(L_m(S_0))$, consider the largest controllable sublanguage. The existence of such a unique supremal element sup$C(L_m(S_0))$ in $C(L_m(S_0))$ is already shown in [18]. The language sup$C(L_m(S_0))$ contains all feasible scheduling sequences (i.e., the largest schedulable language) for the given task set. $\square$

From Theorem 6.2.2, it may be inferred that sup$C(L_m(S_0))$ *contains the exhaustive set of only and all the feasible scheduling sequences* that satisfy the set of specifications related to (i) execution times, (ii) deadlines, and (iii) resource-constraints of the given real-time system. It may be noted that sup$C(L_m(S_0))$ can be empty which implies that there cannot exist any feasible scheduling sequence corresponding to the given specifications, irrespective of the employed scheduling strategy. Hence, the scheduler synthesis mechanism described in this work is optimal. Therefore, using sup$C(L_m(S_0))$, we can design an optimal scheduler S. As a result of supervision, the task executions controlled by S remain within the largest schedulable language sup$C(L_m(S_0))$.

Summary: Given n individual TDES models PG_1, PG_2, ..., PG_n corresponding to periodic tasks τ_1, τ_2, ... τ_n and m TDES models RC_1, RC_2, ..., RC_m corresponding to m processing cores, we can compute the supervisor as follows: (1) $PG =$ `sync`(PG_1, PG_2, $\ldots$, PG_n), (2) $RC =$ `sync`(RC_1, RC_2, $\ldots$, RC_m) and (3) $S =$ `supcon`(PG, RC).

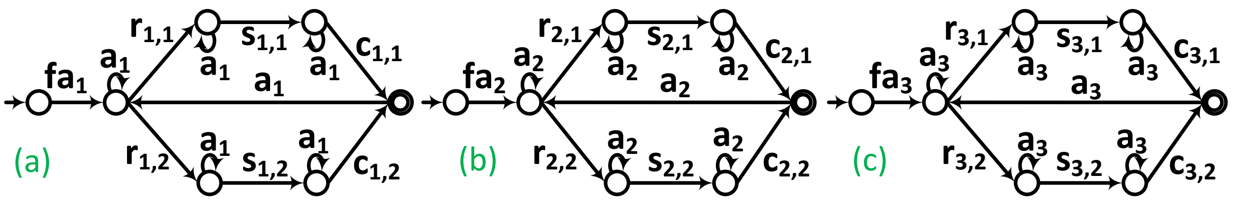
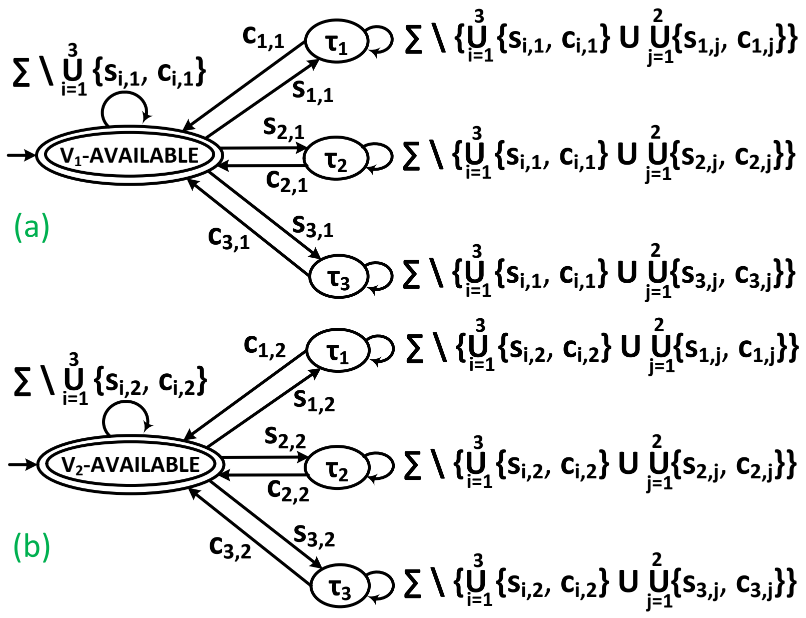


Figure 6.4: *(a) PT_1, (b) PT_2, and (c) PT_3.*



(a)
Σ \ ∪_{i=1}^{3} {s_{i,1}, c_{i,1}}
V1-AVAILABLE
c1,1
s1,1
s2,1
c2,1
s3,1
c3,1
τ1
τ2
τ3
Σ \ {∪_{i=1}^{3} {s_{i,1}, c_{i,1}} ∪ ∪_{j=1}^{2} {s_{1,j}, c_{1,j}}}
Σ \ {∪_{i=1}^{3} {s_{i,1}, c_{i,1}} ∪ ∪_{j=1}^{2} {s_{2,j}, c_{2,j}}}
Σ \ {∪_{i=1}^{3} {s_{i,1}, c_{i,1}} ∪ ∪_{j=1}^{2} {s_{3,j}, c_{3,j}}}
(b)
Σ \ ∪_{i=1}^{3} {s_{i,2}, c_{i,2}}
V2-AVAILABLE
c1,2
s1,2
s2,2
c2,2
s3,2
c3,2
τ1
τ2
τ3
Σ \ {∪_{i=1}^{3} {s_{i,2}, c_{i,2}} ∪ ∪_{j=1}^{2} {s_{1,j}, c_{1,j}}}
Σ \ {∪_{i=1}^{3} {s_{i,2}, c_{i,2}} ∪ ∪_{j=1}^{2} {s_{2,j}, c_{2,j}}}
Σ \ {∪_{i=1}^{3} {s_{i,2}, c_{i,2}} ∪ ∪_{j=1}^{2} {s_{3,j}, c_{3,j}}}

Figure 6.5: *(a) RC_1 and (b) RC_2.*

6.2.6 Example

Consider a system consisting of two unrelated processing cores ($\{V_1, V_2\}$) and three tasks ($\tau_1\langle 0,\ \langle 2, 1\rangle, 3, 4\rangle$, $\tau_2\langle 0,\ \langle 2, 2\rangle, 3, 4\rangle$ and $\tau_3\langle 0,\ \langle 1, 2\rangle, 3, 4\rangle$). The ATG models PT_1 for τ_1, PT_2 for τ_2 and PT_3 for τ_3 are shown in Figures 6.4(a), (b) and (c), respectively. Using these ATG models, we can obtain their corresponding TDES models PG_1, PG_2 and PG_3 (not shown in figure). The TDES models RC_1 for V_1 and RC_2 for V_2 are shown in Figures 6.5(a) and (b), respectively. The composite task and resource-constraint models PG and RC are shown in Figure 6.6(a) and Figure 6.7, respectively. The initial supervisor candidate S_0 and sup$C(L_m(S_0))$ are shown in Figure 6.8.

To illustrate that $L_m(PG)$ contains both resource-constraint satisfying and violating

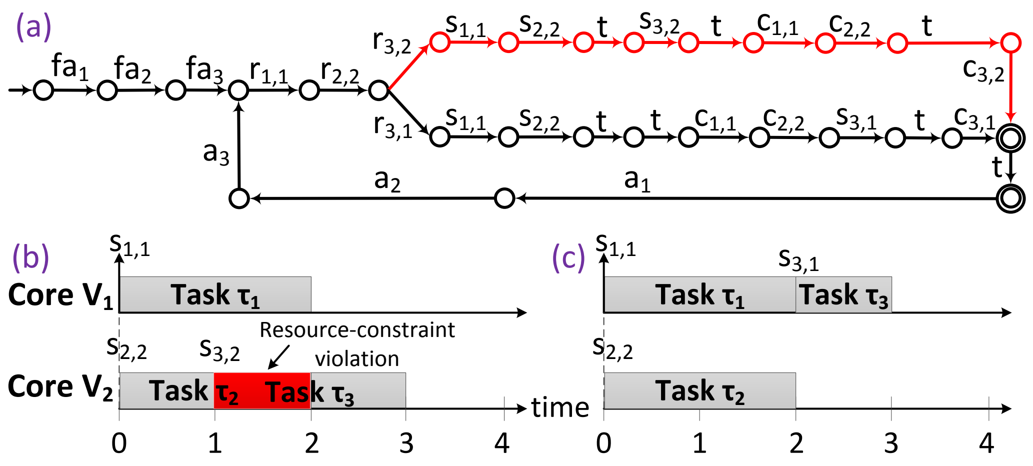


Figure 6.6: *$PG = PG_1||PG_2$ (partial diagram).*

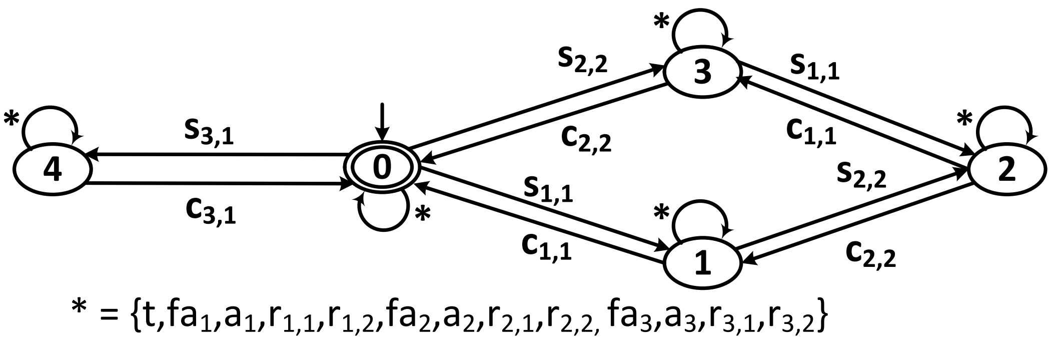


Figure 6.7: *$RC = RC_1||RC_2$ (partial diagram).*

sequences, let us consider the sequences: seq_1 $(= fa_1fa_2fa_3r_{1,1}r_{2,2}r_{3,2}s_{1,1}\ s_{2,2}ts_{3,2}tc_{1,1}c_{2,2}$ $tc_{3,2})$ and seq_2 $(= fa_1fa_2fa_3r_{1,1}r_{2,2}r_{3,1}\ s_{1,1}s_{2,2}ttc_{1,1}c_{2,2}s_{3,1}tc_{3,1})$. The gantt chart representation of seq_1 and seq_2 are shown in Figure 6.6(b) and (c), respectively. seq_1 is resource-constraint violating due to the simultaneous execution of both τ_2 and τ_3 on V_2. On the other hand, seq_2 is resource-constraint satisfying. Let us consider RC and try to find seq_1. After processing the substring $fa_1fa_2fa_3r_{1,1}r_{2,2}r_{3,2}s_{1,1}s_{2,2}t$ of seq_1, RC reaches state 2 where there is no outgoing transition on $s_{3,2}$ and hence, $seq_1 \notin L_m(RC)$. Subsequently, seq_1 leads to deadlock in S_0 implying $seq_1 \notin \sup C(L_m(S_0))$. Meanwhile, we can find seq_2 in $L_m(RC)$ as: $\delta(0, fa_1fa_2fa_3r_{1,1}r_{2,2}r_{3,1}s_{1,1}) = 1$, $\delta(1, s_{2,2}) = 2$, $\delta(2, ttr_{1,1}c_{1,1}) =$ 3, $\delta(3, c_{2,2}) = 0$, $\delta(0, s_{3,1}) = 4$, $\delta(4, tc_{3,1}) = 0$. Further from Figure 6.8 (within the dotted box), we can find seq_2, implying $seq_2 \in L_m(S_0)$ and $seq_2 \in \sup C(L_m(S_0))$. Similarly, we

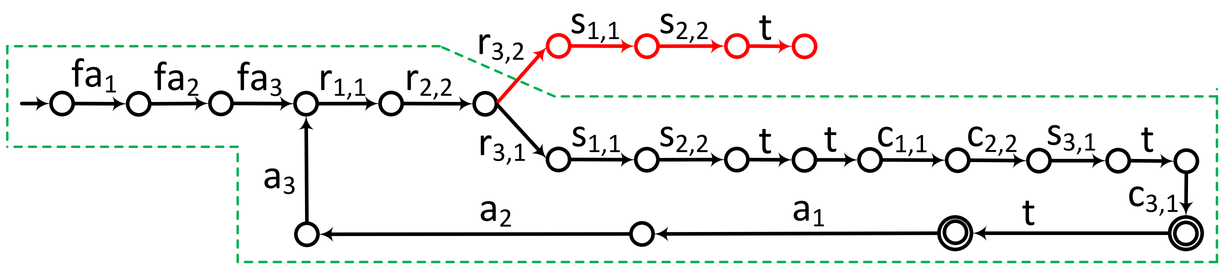


Figure 6.8: *S_0 (partial diagram); supC($L_m(S_0)$) (in dotted box).*

can find that all resource-constraint satisfying sequences in $L_m(PG)$ are also present in sup$C(L_m(S_0))$. Hence, we can design an optimal scheduler S using sup$C(L_m(S_0))$.

Working of scheduler S: When all three tasks arrive simultaneously, S allocates τ_1 on V_1 ($r_{1,1}$), τ_2 on V_2 ($r_{2,2}$) and τ_3 on V_1 ($r_{3,1}$). The system behavior $L(PG)$ allows the possibility of assigning τ_3 on either V_1 or V_2. However, assigning τ_3 on V_2 leads to a deadlock state. Hence, S assigns τ_3 on V_2 by disabling $r_{3,2}$ (and enabling $r_{3,1}$) and ensures that no deadlock state is reached.

6.3 Summary

In this chapter, we have presented *task execution* and *resource-constraint* models that can be used to synthesize scheduler for a set of independent, non-preemptive periodic tasks executing on *heterogeneous* multi-cores. The synthesis process begins by developing the task execution models for each periodic task which effectively captures a task's, (i) *distinct execution requirements* on different processing cores, (ii) *deadline requirement* and (iii) *fixed inter-arrival time* constraint between any two consecutive instances. Then, we developed for each processor the specification models in order to enforce *resource-constraints*. Through a *synchronous product* on the individual models, we obtained the composite task execution model and specification model. Finally, the scheduler which contains the set of valid execution sequences, that can be used during on-line execution, is synthesized. We have also discussed the optimality and working of the scheduler synthesized using our proposed models.

The presented models can be adapted to consider the execution of sporadic tasks. Specifically, the time bound associated with a_i can be set as (a_i, P_i, ∞) to model the minimum inter-arrival time constraint P_i associated with the sporadic task τ_i. Further, a BDD based symbolic computation presented in Section 5.3 can be utilized to control state-space complexity involved in the synthesis process. In the next chapter, we develop the models which can be used to synthesize a scheduler for a multiprocessor system executing a real-time application modelled as precedence-constrained task graphs.

Chapter 7
Scheduling of Parallel Real-Time Tasks

In the earlier chapters of this dissertation, we have considered the scheduling of independent real-time tasks on a multiprocessor / multi-core platform. To effectively utilize multi-cores, real-time software APIs are expected to take the advantage of parallel processing/programming [90]. For example, many real-time applications such as radar tracking, autonomous driving, and video surveillance, are highly parallelizable [44]. One of the most generic mechanisms for modeling parallel real-time applications is *Precedence-constrained Task Graphs*(PTG)/*Directed Acyclic Graphs*(DAG) [30, 34, 89]. In the PTG model of an application, each node corresponds to a task and edges denote inter-task dependencies. In this chapter, we present a scheduler synthesis framework for a parallel real-time application represented by a PTG, executing on a multi-core platform.

Apart from guaranteeing the timely execution of tasks in a resource-constrained environment, ensuring proper functioning of the system even in the presence of faults (i.e., *fault tolerance*) has currently become a design constraint of paramount importance. Specifically, the processors on which the tasks are executed, are subject to a variety of faults. Such faults are broadly classified to be either *permanent* or *transient* [62]. Permanent processor faults are irrecoverable and do not go away with time. On the other hand, transient faults are short-lived (momentary) and their effect goes away after some time. According to studies [60, 100], the ratio of transient-to-permanent faults can be 100:1 or even higher. Thus, robustness against transient faults is emerging as a very important criterion in the design of safety critical real-time systems [56].

Hence, we extend our scheduler synthesis scheme to *consider the incorporation of efficient mechanisms for handling transient processor faults that may occur during the execution of PTGs.*

7.1 Related Works

The classical multiprocessor scheduling theory is focused on the sequential programming model, where the problem is to schedule many independent real-time tasks on multiple processing cores [10, 24]. Parallel programming models introduce a new dimension to this problem, where some sub-tasks (threads) in a program (task) can run in parallel to produce partial results individually. Certain threads should synchronize to integrate the partial results. Several task models have been proposed in the literature to analyze the timing behavior of parallel real-time applications, a few important ones being the *fork-join* model, the *synchronous-parallel* model and the *PTG* model [46, 64, 102].

Fork-Join Task Model: It consists of an alternate sequence of interleaved sequential and parallel regions, called *segments*, and all the threads within each segment should synchronize in order to proceed to the next segment. Lakshmanan et al. [63] developed a *stretching algorithm* which attempts to judiciously stretch the durations of the parallel segments such that the overall deadline is not violated. The objective of this stretching process is to minimize the number of processors required during the execution of parallel segments. The processors which become partially free in this process may be allocated to other applications so that overall resource utilization of the system may be enhanced. Later, Fauberteau et al. [42] improved the *stretching algorithm* with the objective of minimizing thread migrations and preemptions within parallel segments.

Synchronous-Parallel Task Model: Relaxing the restriction that sequential and parallel segments alternate (that was considered in the Fork-Join model), this model allows each segment to have any arbitrary number of threads. Safifullah et al. [96] devised a *decomposition algorithm* which divides a parallel task into a set of independent sequential segments, each with its own release and deadline. Later, Nellison et al. [81] presented an improved technique which requires only m' processors to schedule a parallel task as compared to $m(\geq m')$ processors required by [96].

PTG Task Model: It generalizes the *synchronous-parallel* model by relaxing the restriction of segment-level synchronization by all threads. Here, each task node may spawn multiple parallel threads and any arbitrary number of threads may synchronize at designated task nodes, as required. Baruah et al. [9] presented a polynomial time algorithm which can provide suitable abstractions to conditional constructs (such as *-if-then-else-*) in a program so that modeling using PTGs become possible. Most of the existing scheduling schemes for PTG models have applied *Global Earliest Deadline First* (GEDF) and its related schedulability analysis [11,34,69]. However, schedulability of parallel real-time tasks can be improved significantly if thread-level parallelism is directly reflected into scheduling algorithms as well. Hence, there is a need to develop new real-time scheduling techniques for PTGs.

SCTDES based scheduling works: Table 7.1 synopsizes a qualitative comparison among the SCTDES based scheduling approaches. It may be observed that SCTDES based existing scheduler synthesis works deal with a variety of real-time and fault-tolerant scheduling schemes. However, these works deal with independent tasks only and do not consider dependencies among tasks. Hence, they are not applicable to the scheduling of parallel dependent real-time tasks. In this work, *we utilize SCTDES to compute a correct-by-construction optimal scheduler for such parallel real-time tasks modeled as PTGs, executing on homogeneous multi-core systems.*

Table 7.1: *Comparison between SCTDES based scheduling schemes*

Method	**Tasks**	**Preemptive / Non-preemptive**	**Uniprocessor / Multi-core**	**Remarks**
[29]	Periodic	Non-preemptive	Uniprocessor	It does not consider task priorities
[53]	Periodic	Both	Uniprocessor	It considers task priorities
[87]	Periodic & Sporadic	Preemptive	Uniprocessor	It does not consider inter-arrival time constraint
[39]	Aperiodic & Sporadic	Non-preemptive	Uniprocessor	It correctly captures inter-arrival time constraint
[105]	Periodic	Preemptive	Uniprocessor	It supports tasks with multiple periods to allow reconfiguration
[106]	Periodic	Conditionally preemptive	Uniprocessor	It can be extended to homogeneous multi-cores
[107]	Periodic & Sporadic	Conditionally preemptive	Uniprocessor	It can be extended to homogeneous multi-cores
Present work	Sporadic	Task nodes are non-preemptive	Homogeneous multi-core	Correctly models execution of PTGs on multi-cores

7.2 Scheduler Synthesis for PTGs

With a basic understanding of the fundamental notions on SCTDES based system modeling presented in the above section, we proceed towards the design of an optimal scheduler synthesis mechanism for PTGs. First, we describe the system model, assumptions and problem statement.

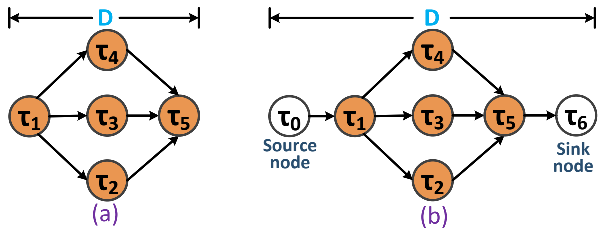
System Model: We consider a real-time system consisting of an application composed of multiple dependent tasks modeled as a *Precedence-constrained Task Graph* (PTG) to be scheduled on a multi-core system with m homogeneous processing cores ($V = \{V_1, V_2, \ldots, V_m\}$). A PTG G is represented by a five tuple $G = \langle I, L, E, D, P \rangle$, where,

- $I = \{\tau_1, \tau_2, \ldots, \tau_n\}$ represents a set of *n-task* nodes.
- $L \subseteq I \times I$ is a set of edges that describe the *precedence relationships* among nodes in I.
- $E = \{E_1, E_2, \ldots, E_n\}$ is the set of execution times (E_i denotes the execution time of *task* τ_i).
- D is the *end-to-end deadline* by which all task nodes in G must complete their execution.
- $P(\geq D)$ is the *minimum inter arrival time* between two consecutive instances of PTG G.

Assumptions:

1. All task nodes execute *non-preemptively.*
2. *Communication cost* between task nodes is negligible.
3. Without loss of generality, we add a dummy *source* node τ_0 and *sink* node τ_{n+1} to G such that (i) τ_0 has outgoing edges to all nodes in G that do not have any incoming edges, (ii) τ_{n+1} has incoming edges from all nodes in G that do not have

any outgoing edges. We set $E_0 = E_{n+1} = 0$ and use G' to refer to this transformed version of PTG G. All nodes in G' except the source and sink nodes in G' are referred to as *intermediate* nodes. Example PTG G and its transformed version G' are shown in Figures 7.1(a) and (b), respectively.



Event	Description
a	Arrival of an instance of PTG **G'**
$s_{i,j}$	Start of execution of τ_i on core V_j
c_i	Completion of execution of τ_i
t	Passage of one unit time of the clock

(c)

Figure 7.1: *(a) PTG G, (b) PTG G′, (c) Description of Events in Σ.*

Problem Statement-1: *Given a PTG G' and m homogeneous processing cores, design an optimal supervisor which contains scheduling sequences that guarantee the execution and end-to-end deadline requirement of G'.*

7.2.1 Scheduler Synthesis Scheme

We start the scheduler synthesis process by defining the event set associated with G'. Individual execution models for (i) source (T_0 for τ_0), (ii) intermediate (T_1 to T_n for τ_1 to τ_n, respectively), and (iii) sink task nodes (T_{n+1} for τ_n) in G' are developed next. Then, we integrate all the individual models (to obtain T) constructed for the different types of task nodes in G' using synchronous product composition ($T = ||_{i=0}^{n+1} T_i$). The marked behavior of T may contain execution sequences that violate timing requirements such as deadline and minimum inter-arrival time. Hence, a timing specification model H for G' is developed and composed with T. The resulting composite model $T||H$ may contain deadlock states in it. However, we prove that $T||H$ is controllable (in Lemma 7.2.2). The final supervisor which contains the exhaustive set of all feasible scheduling sequences, is obtained through *trim* operation over $T||H$.

Event set: $\Sigma_0 = \{a, c_0\}$, $\Sigma_i = \{\{s_{i,j} | 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{c_i | 1 \leq i \leq n\}\}$, $\Sigma_{n+1} = \{c_{n+1}\}$ and $\Sigma = \cup_{i=0}^{n+1} \Sigma_i \cup \{t\}$. The events are described in Table 7.1(c). The events are categorized as follows:

- $\Sigma_c = \cup_{i=1}^{n} \cup_{j=1}^{m} \{s_{i,j}\}$: The start of execution of a task node can be controlled by the supervisor (i.e., scheduler), and hence, it is modeled as *controllable*.
- $\Sigma_{uc} = \cup_{i=1}^{n}\{c_i\} \cup \{a\}$: Since, the arrival and completion events cannot be controlled by the scheduler, they are modeled as *uncontrollable* events.
- $\Sigma_{for} = \Sigma_c$: The controllable events are also modeled as *forcible* events which can preempt t.

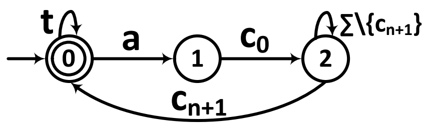
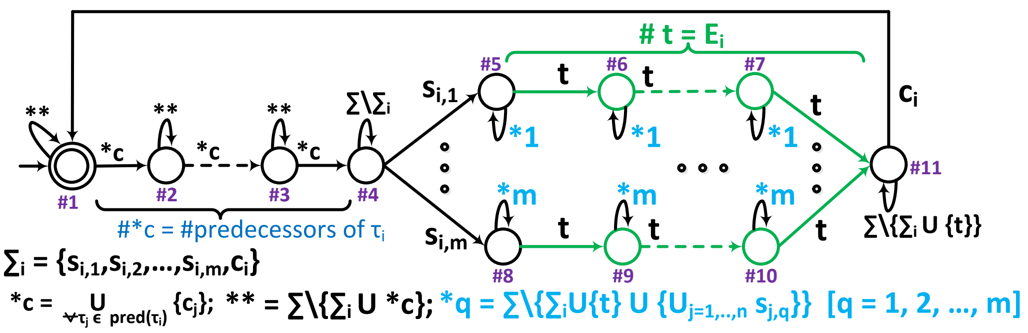


Figure 7.2: *Task execution model T_0 for source node τ_0*

Source node τ_0: The TDES model T_0 for task τ_0 is shown in Figure 7.2 and is formally defined as: $T_0 = (Q,\ \Sigma,\ q_0,\ Q_m,\ \delta_0)$, where, $Q = \{State\ 0,\ State\ 1,\ State\ 2\}$, Σ is the event set (defined at the start of Section 7.2.1), $q_0 = Q_m = \{State\ 0\}$. A description of transition function δ_0 is as follows:

- $\delta_0(State\ 0,\ t) = State\ 0$: T_0 stays at *State* 0 (self-loop on t) until the arrival (event a) of PTG G'. It implicitly models the arbitrary inter-arrival time between consecutive instances of PTG G'.
- $\delta_0(State\ 0,\ a) = State\ 1$: On the occurrence of arrival event a, T_0 transits to *State* 1.
- $\delta_0(State\ 1,\ c_0) = State\ 2$: It marks the completion of τ_0's execution. It may be noted that T_0 does not model the assignment/execution of τ_0 on any processor as it is a dummy task node.
- $\delta_0(State\ 2,\ \Sigma \setminus \{c_{n+1}\}) = State\ 2$: This self-loop represents that T_0 continues to stay at *State* 2 until the completion of sink task node τ_{n+1}.
- $\delta_0(State\ 2,\ c_{n+1}) = State\ 0$: On the occurrence of event c_{n+1}, T_0 transits to *State* 0 to mark the completion of the current instance of PTG G'.



# t = E_i
#5
#6
#7
t
t
t
c_i
**
**
**
$\Sigma\backslash\Sigma_i$
$s_{i,1}$
*c
*c
*c
*1
*1
*1
#11
#1
#2
#3
#4
*m
*m
*m
t
t
t
$\Sigma\backslash\{\Sigma_i \cup \{t\}\}$
#*c = #predecessors of τ_i
$s_{i,m}$
$\Sigma_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,m}, c_i\}$
#8
#9
#10
$*c = \bigcup_{\forall \tau_j \in pred(\tau_i)} \{c_j\}$; $** = \Sigma\backslash\{\Sigma_i \cup *c\}$; $*q = \Sigma\backslash\{\Sigma_i \cup \{t\} \cup \{\cup_{j=1,\ldots,n} s_{j,q}\}\}$ [q = 1, 2, …, m]

Figure 7.3: *Task execution model T_i for intermediate node τ_i*

Intermediate node τ_i: It may be noted that G' contains a single source and sink node. However, there may be multiple intermediate nodes. Let us consider a node τ_i from the set of intermediate nodes in G'. The TDES model T_i for node τ_i is shown in Figure 7.3. Labels have not been specified for all the states shown in the figure in order to reduce its size. However, states are numbered sequentially starting from #1 and these numbers will be used as references while explaining the model. T_i is formally defined as: $T_i = (Q, \Sigma, q_0, Q_m, \delta_i)$, where, $Q = \{State \ \#1, State \ \#2, \ldots, State \ \#11\}$, Σ is the event set (defined at the start of Section 7.2.1), $q_0 = Q_m = \{State \ \#1\}$. A description of transition function δ_i is as follows:

- $\delta_i(State \ \#1, \Sigma \setminus \{\Sigma_i \cup {}^*c\}) = State \ \#1$: This self-loop models the location of T_i at *State* #1 until the completion of anyone of its predecessor task nodes. The self-loop $\Sigma \setminus \{\Sigma_i \cup {}^*c\}$ may be described as:
  - Since τ_i has not yet started its execution, the events associated with it are excluded from Σ.
  - ${}^*c = \bigcup_{\forall \tau_j \in pred(\tau_i)} \{c_j\}$: This represents the set of completion events corresponding to all immediate predecessors of τ_i. The events in ${}^*c$ are excluded from Σ. In fact ${}^*c$ forms a label of a separate outgoing transition at *State* #1 in order to allow τ_i to proceed one step closer towards its start of execution.
- *State* #2 to *State* #4: States that are similar to *State* #1 are replicated $|{}^*c|$ times (from *State* #1 to *State* #3) to model the completion of all the immediate prede-

cessor nodes of τ_i. Subsequent to the completion of all immediate predecessors, T_i reaches *State* #4.

- $\delta_i(\textit{State}~\#4, \Sigma \setminus \Sigma_i) = \textit{State}~\#4$: The scheduler takes a decision whether to immediately allocate task τ_i for execution on a processing core or make it wait on the ready queue. The latter is indicated by the self-loop $\Sigma \setminus \Sigma_i$ at State #4.

- $\delta_i(\textit{State}~\#4, s_{i,1}) = \textit{State}~\#5$: Task node τ_i is assigned on processing core V_1 for its execution.

- $\delta_i(\textit{State}~\#5, {}^*1) = \textit{State}~\#5$: The self-loop transition ^{*1} ($= \Sigma \setminus \{\Sigma_i \cup \{t\} \cup \{\cup_{j=1,\ldots,n} s_{j,1}\}\}$) models the following three scenarios:

  1. After assigning τ_i on V_1 ($s_{i,1}$), τ_i will not be allowed to execute any event associated with it. This is modeled by excluding the events Σ_i from ^{*1}.
  2. τ_i is allowed to stay at State #5 by executing events in ^{*1} until the occurrence of the next *tick* event. This is modeled by excluding the *tick* event from ^{*1}.
  3. After assigning τ_i on V_1, no other task ($\tau_j \in I, j \neq i$) will be allowed to execute on core V_1. This is modeled by excluding the events $\{\cup_{j=1,\ldots,n} s_{j,1}\}$ from ^{*1}. It ensures that *task* τ_i *cannot be preempted by another task* τ_j ($\tau_i \neq \tau_j$) for at least one *tick* event. It also restricts task τ_i from re-executing event $s_{i,1}$ until the next *tick* event.

- $\delta_i(\textit{State}~\#5, t) = \textit{State}~\#6$: Task node τ_i completes one unit time execution on core V_1. The states that are similar to *State* #5 are replicated E_i times to model the non-preemptive execution of τ_i until its completion (i.e., from *State* #5 to *State* #11).

- $\delta_i(\textit{State}~\#11, \Sigma \setminus \{\Sigma_i \cup \{t\}\}) = \textit{State}~\#11$: This self-loop is used to allow the execution of untimed events that may happen with respect to other tasks in the system.

- $\delta_i(State\ \#11,\ c_i) = State\ \#1$: The completion of τ_i's execution is marked by transition on c_i to $State\ \#1$.

Here, we have assumed that τ_i started its execution on core V_1. However, it may be noted that task node τ_i is allowed to start its execution on anyone of the available cores $\{V_1, V_2, \ldots, V_m\}$.

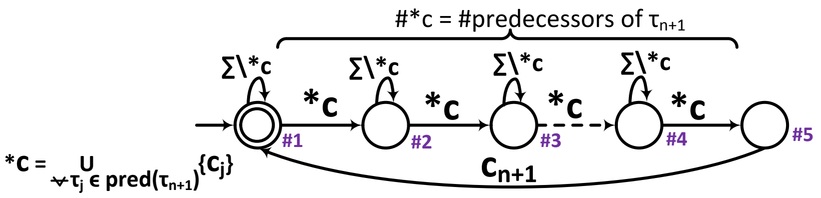


Figure 7.4: *Task execution model $PT_{i,act}$ for sink node τ_{n+1}*

Sink node τ_{n+1}: The TDES model T_{n+1} for task node τ_{n+1} is shown in Figure 7.4 and is formally defined as follows: $T_{n+1} = (Q,\ \Sigma,\ q_0,\ Q_m,\ \delta_{n+1})$, where, $Q = \{State\ \#1, State\ \#2, \ldots, State\ \#5\}$, Σ is the event set (defined at the start of Section 7.2), $q_0 = Q_m = \{State\ \#1\}$. Description of transition function δ_{n+1} is as follows:

- $\delta_{n+1}(State\ \#1,\ \Sigma\setminus{}^*c) = State\ \#1$: This self-loop models the location of T_{n+1} at $State\ \#1$ until the completion of anyone of its immediate predecessor task nodes. States that are similar to $State\ \#1$ are replicated $|{}^*c|$ times (from $State\ \#1$ to $State\ \#4$) to model the completion of all the predecessor nodes of τ_{n+1}. Subsequent to the completion of all immediate predecessors of τ_{n+1}, T_{n+1} reaches $State\ \#5$.

- $\delta_{n+1}(State\ \#5,\ c_{n+1}) = State\ \#1$: Completion of τ_{n+1}'s execution is marked by the transition on c_{n+1} to $State\ \#1$. Being a dummy node, the execution of τ_{n+1} does not incur any dealy.

The completion of τ_{n+1} implicitly marks the completion of the current instance of PTG G'.

Example: Now, we illustrate the task execution models discussed above using an example. Let us consider the PTG G shown in Figure 7.5(a). PTG G which contains five task nodes $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ with corresponding execution times $\{1, 1, 2, 1, 1\}$, is executed on a two core system. The deadline as well as minimum inter-arrival time of G is 5 time units. According to our proposed scheme, we first transform G into G' (shown in Figure 7.5(b)). The event set for each task node in G' is shown in Figure 7.5(c). The execution model of task nodes τ_0 to τ_6 are shown in Figures 7.5(d) to 7.5(j). $\square$

7.2.2 Composite Task Execution Model

Given individual TDES task models T_0, T_1, ...,T_{n+1}, a *synchronous product* $T = ||_{i=0}^{n+1} T_i$ on the individual models gives us the *composite task execution model* for PTG G'. The model T represents the overall execution of PTG G' commencing from its source node, through all the intermediate nodes and culminating with the completion of its sink node. T is also able to correctly capture all *precedence-constraints* among task nodes because it allows a task to start execution only after the completion of all the task's predecessor nodes. Apart from this, T also contains sequences that represent the concurrent execution of task nodes that are mutually independent (i.e., they do not have any precedence relations among them). Before proceeding further, we introduce three definitions related to *schedule length*, *deadline* and *inter-arrival time*.

Definition: *Makespan:* The overall completion time of PTG G' and is determined by the number of ticks between the arrival event a and the completion event c_{n+1} within any scheduling sequence corresponding to G'. That is, for any sequence $x = x_1 a x_2 c_{n+1} x_3 \in L_m(T)$, where $x_1, x_3 \in \Sigma^*$ and $x_2 \in (\Sigma \setminus \{c_{n+1}\})^*$, *makespan(x)* = *tickcount*(x_2). $\square$

Definition: *Deadline-meeting sequence:* A sequence $x = x_1 a x_2 c_{n+1} x_3 \in L_m(T)$, where $x_1, x_3 \in \Sigma^*$, $x_2 \in (\Sigma \setminus \{c_{n+1}\})^*$ is *deadline-meeting*, if *tickcount*$(x_2) \leq D$. Otherwise, x is deadline-missing. $\square$

Definition: *Sporadic Sequence:* Suppose T is a TDES corresponding to a PTG G with minimum inter-arrival time P. A sequence $s \in L_m(T)$ is a prefix of a *sporadic* sequence if for all $s_1, s_2, \ldots, s_k \in (\Sigma \setminus \{a\})^*$, and $s = s_1 a s_2 a \ldots s_k a ...$ implies that $\forall k (k > 1)$ *tickcount*$(s_k) \geq P$. Since T represents the sporadic PTG G, this definition must be satisfied $\forall s \in L_m(T)$. $\square$

As execution models for individual task nodes enforce that at most one task may execute on any processor at a given time, their composition in T also preserves the same,

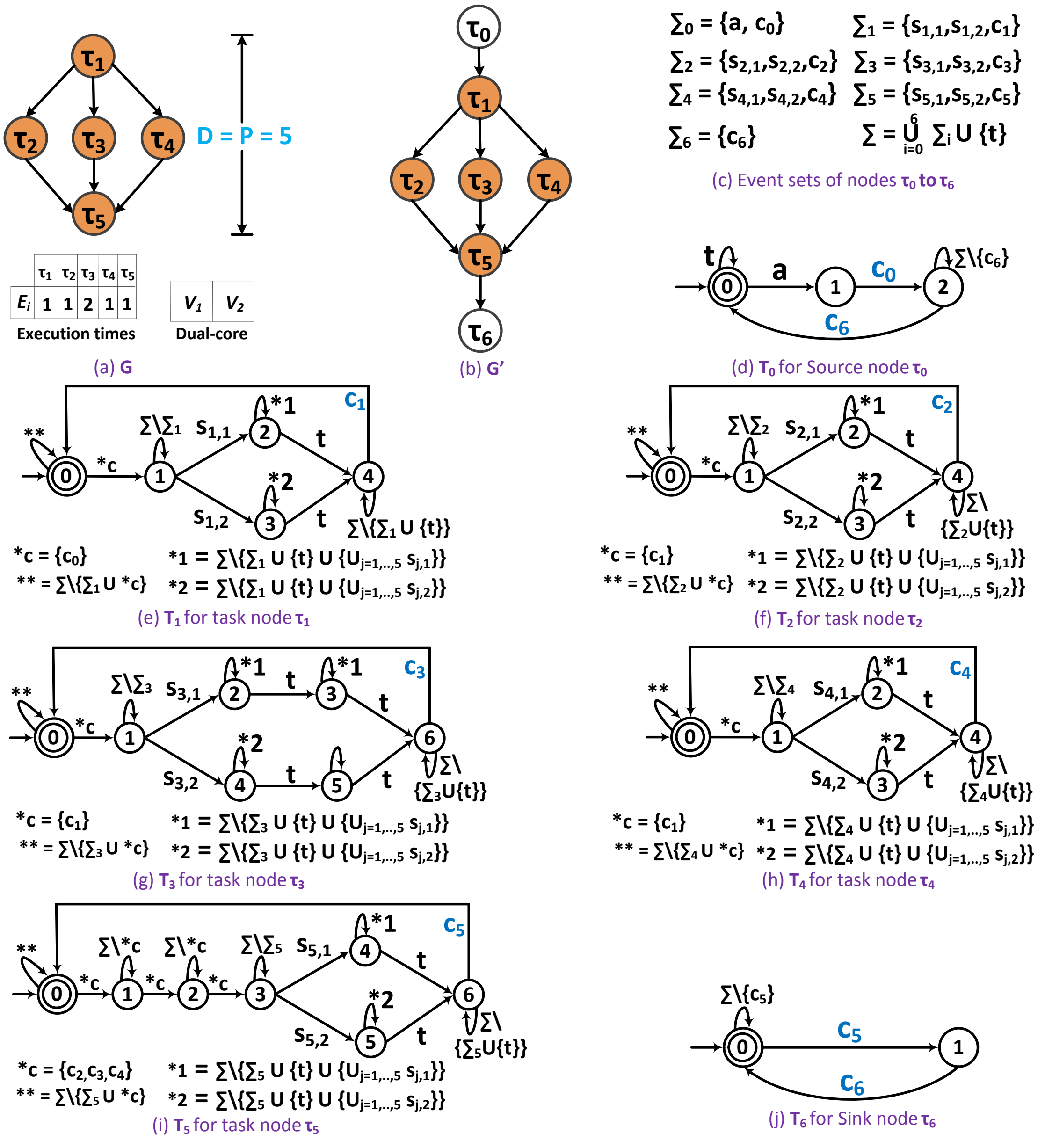

D = P = 5
Execution times
Dual-core
(a) G
(b) G'
Σ0 = {a, c0} Σ1 = {s1,1, s1,2, c1}
Σ2 = {s2,1, s2,2, c2} Σ3 = {s3,1, s3,2, c3}
Σ4 = {s4,1, s4,2, c4} Σ5 = {s5,1, s5,2, c5}
Σ6 = {c6} Σ = ⋃ i=0..6 Σi ∪ {t}
(c) Event sets of nodes τ0 to τ6
(d) T0 for Source node τ0
*c = {c0} ** = Σ\{Σ1 ∪ *c} *1 = Σ\{Σ1 ∪ {t} ∪ {∪j=1,..,5 sj,1}} *2 = Σ\{Σ1 ∪ {t} ∪ {∪j=1,..,5 sj,2}}
(e) T1 for task node τ1
*c = {c1} ** = Σ\{Σ2 ∪ *c} *1 = Σ\{Σ2 ∪ {t} ∪ {∪j=1,..,5 sj,1}} *2 = Σ\{Σ2 ∪ {t} ∪ {∪j=1,..,5 sj,2}}
(f) T2 for task node τ2
*c = {c1} ** = Σ\{Σ3 ∪ *c} *1 = Σ\{Σ3 ∪ {t} ∪ {∪j=1,..,5 sj,1}} *2 = Σ\{Σ3 ∪ {t} ∪ {∪j=1,..,5 sj,2}}
(g) T3 for task node τ3
*c = {c1} ** = Σ\{Σ4 ∪ *c} *1 = Σ\{Σ4 ∪ {t} ∪ {∪j=1,..,5 sj,1}} *2 = Σ\{Σ4 ∪ {t} ∪ {∪j=1,..,5 sj,2}}
(h) T4 for task node τ4
*c = {c2, c3, c4} ** = Σ\{Σ5 ∪ *c} *1 = Σ\{Σ5 ∪ {t} ∪ {∪j=1,..,5 sj,1}} *2 = Σ\{Σ5 ∪ {t} ∪ {∪j=1,..,5 sj,2}}
(i) T5 for task node τ5
(j) T6 for Sink node τ6

Figure 7.5: *Example PTG G and TDES models for its task nodes*

thus satisfying resource constraints. However, due to the existence of self-loops on *tick* events in T, *makespans* of sequences in $L_m(T)$ may violate the stipulated deadline D of PTG G'. For example, the processing cores might be kept idle even in the presence of ready to execute task nodes. Similarly, T does not provide any bounds on the number of ticks between consecutive instances of G', thus being unable to satisfy any pre-specified constraint on sporadicity. Hence, $L_m(T)$ *contains both deadline-meeting as well as deadline-missing sequences with arbitrary inter-arrival times for PTG G'.*

Example (cntd.): Figure 7.6 shows the partial diagram of the composite task execution model T $(= ||_{i=0}^{6} T_i)$ corresponding to the individual task execution models in Figure 7.5. Now, we consider three related sequences from $L_m(T)$ for discussion whose gantt chart representations are shown at the bottom of Figure 7.6.

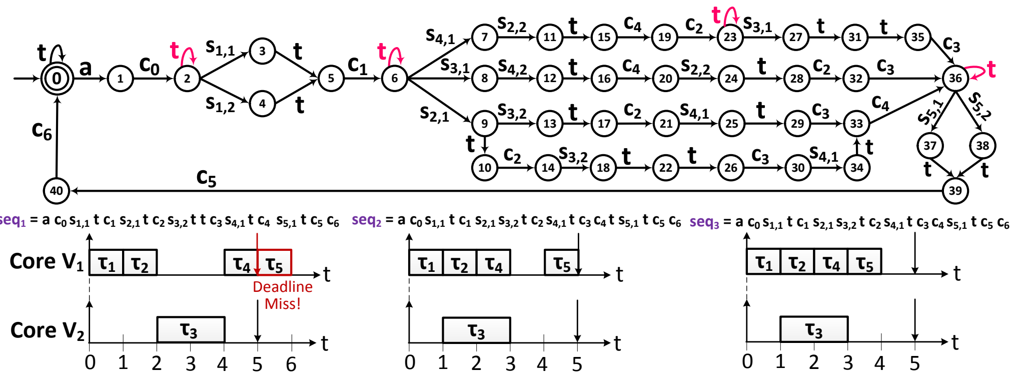

seq₁ = a c₀ s₁,₁ t c₁ s₂,₁ t c₂ s₃,₂ t t c₃ s₄,₁ t c₄ s₅,₁ t c₅ c₆
seq₂ = a c₀ s₁,₁ t c₁ s₂,₁ s₃,₂ t c₂ s₄,₁ t c₃ c₄ t s₅,₁ t c₅ c₆
seq₃ = a c₀ s₁,₁ t c₁ s₂,₁ s₃,₂ t c₂ s₄,₁ t c₃ c₄ s₅,₁ t c₅ c₆
Core V₁
Core V₂
Deadline Miss!
τ₁ τ₂ τ₃ τ₄ τ₅

Figure 7.6: *Composite task execution model $T = ||_{i=0}^{6} T_i$ (partial diagram).*

- *seq_1* $= ac_0s_{1,1}tc_1s_{2,1}tc_2s_{3,2}ttc_3s_{4,1}tc_4s_{5,1}tc_5c_6 \in L_m(T)$: In this sequence, *PTG G'* arrives at system start and the dummy start node τ_0 completes its execution without incurring any delay (i.e., ac_0). Subsequently, the task node τ_1 starts its execution on core V_1 and completes its execution after one tick (i.e., $s_{1,1}tc_1$). Similarly, other tasks τ_2, τ_3, τ_4 and τ_5 are executed in an interleaved fashion. Finally, the completion event associated with the dummy sink node τ_6 is executed marking the completion of the current instance of G'. The *makespan* of seq_1 is 6 time units which is greater than the deadline D (= 5) of G'. That is, $tickcount(ac_0s_{1,1}tc_1s_{2,1}tc_2s_{3,2}ttc_3s_{4,1}tc_4s_{5,1}tc_5c_6) = 6 \nleq 5$ and hence, seq_1 is a *deadline-missing.*

- $seq_2 = ac_0 s_{1,1} t c_1 s_{2,1} s_{3,2} t c_2 s_{4,1} t c_3 c_4 t s_{5,1} t c_5 c_6 \in L_m(T)$: In this sequence, subsequent to the completion of τ_1, task nodes τ_2 and τ_3 are executed concurrently on cores V_1 and V_2 (i.e., $c_1 s_{2,1} s_{3,2}$). After the elapse of one tick, task τ_2 completes its execution. Next, τ_4 starts its execution on V_1 and subsequently, both τ_3 and τ_4 complete their execution. Then, both the cores remain idle for one tick after which τ_5 starts and completes its execution on V_1. The makespan of seq_2 is 5 time units and hence, it is *deadline-meeting.*

- $seq_3 = ac_0 s_{1,1} t c_1 s_{2,1} s_{3,2} t c_2 s_{4,1} t c_3 c_4 s_{5,1} t c_5 c_6 \in L_m(T)$: This sequence is also similar to seq_2 except that the processing cores are never kept idle until the completion of all nodes in G'. The makespan of seq_3 is 4 and it is also *deadline-meeting.*

To summarize, the marked behavior $L_m(T)$ contains deadline-meeting and deadline-missing execution sequences with arbitrary inter-arrival times. In order to restrict the marked behavior to correctly capture only those sequences which satisfy the end-to-end deadline and inter-arrival time constraints, a timing specification model corresponding to PTG G' is developed.

7.2.3 Timing Specification Model

Before presenting the model, let us introduce the notion of *critical path* in a PTG.

Definition: *Critical Path (CP):* It is the longest path from the source node to the sink node in terms of the total execution time consumed by the task nodes in it. The length of the critical path is denoted by E_{cri}, i.e., $E_{cri} = \sum_{\tau_i \in CP} E_i$.

The timing specification model H (shown in Figure 7.7) for PTG G' with end-to-end deadline D and minimum inter-arrival time of P time units, is defined as: $H = (Q, \Sigma, q_0, Q_m, \delta_H)$, where, $Q = \{State\ \#1, State\ \#2, \ldots, State\ \#12\}$, Σ is the event set, $q_0 = State\ \#1$, $Q_m = \{State\ \#8, State\ \#9, \ldots, State\ \#12\}$. Description of transition function δ_H is as follows:

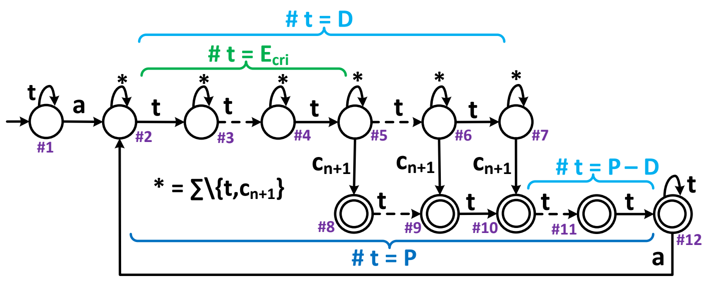


Figure 7.7: *Timing Specification model H for PTG G'*

- $\delta_H(State\ \#1,\ t) = State\ \#1$: This self-loop models the location of H at *State* #1 until the arrival of PTG G'. It implicitly models the arbitrary arrival of G'.

- $\delta_H(State\ \#1,\ a) = State\ \#2$: On the occurrence of event a, H transits to *State* #2.

- $\delta_H(State\ \#2,\ \Sigma \setminus \{t, c_{n+1}\}) = State\ \#2$: This self-loop is used to model the fact that G' is allowed to execute any event from Σ except t and c_{n+1}. The reason behind restricting c_{n+1} is that G' requires at least E_{cri} ticks to complete the execution of all nodes in it, irrespective of the amount of available resources.

- $\delta_H(State\ \#2,\ t) = State\ \#3$: On the occurrence of t, H moves to the next state. Since, H measures the inter-arrival time (P) between two consecutive instances of G', it contains $\mathcal{O}(P)$ distinct states that are separated by the tick event.

- *State* #3 to *State* #5: The states that are similar to *State* #3 are instantiated E_{cri} times. After the elapse of E_{cri} ticks from the arrival of the current instance, H reaches *State* #5.

- $\delta_H(State\ \#5,\ c_{n+1}) = State\ \#8$: After E_{cri} ticks are elapsed, G' may possibly complete its execution. On completion, c_{n+1} takes H from *State* #5 to *State* #8.

- *State* #5 to *State* #7: Since G' may complete its execution anytime in between E_{cri} and D, outgoing transition on c_{n+1} is defined at these states. After completion, H may reach anyone of the states from *State* #8 to *State* #10. All these states have been marked to represent completion of the current instance of G'.

- *State* #8 to *State* #12: At these states, execution of any event other than tick is disallowed. These tick events enforce the elapse of at least P ticks from the arrival of the current instance subsequent to which, H reaches *State* #12.

- δ_H(*State* #12, t) = *State* #12: The elapse of P ticks enables the arrival of the next instance of G'. However, the next instance of may not arrive immediately. This self-loop captures the situation allowing an arbitrary number of ticks to elapse.

- δ_H(*State* #12, a) = *State* #2: On the occurrence of a, H transits back to *State* #2 initiating the start of the next instance of G'.

From the transition structure of H, it may be inferred that H models both *deadline* and *minimum inter-arrival time* constraints of G'. Therefore, $L_m(H)$ *contains all the deadline-meeting, sporadic execution sequences for* G'. However, H does not model resource usage and consequently, allows the erroneous behavior of the concurrent execution of multiple tasks on the same processing core. In addition, H also does not model inter task dependencies in G'. Thus, sequences in $L_m(H)$ *may violate resource as well as precedence constraints.*

Example (cntd.): The timing specification model for the example system configuration under consideration is shown in Figure 7.8. From Figure 7.5(b), it may be seen that the critical path of G' is: $\tau_0 \rightarrow \tau_1 \rightarrow \tau_3 \rightarrow \tau_5 \rightarrow \tau_6$. The length of this path is (E_{cri} =) 4 time units. Hence, the completion event c_6 is allowed only after 4 ticks from the occurrence of the arrival event a. In addition, the deadline and minimum inter-arrival time of G' is 5 and hence, H allows c_6 to occur either after 4 or 5 ticks from the arrival of the current instance.

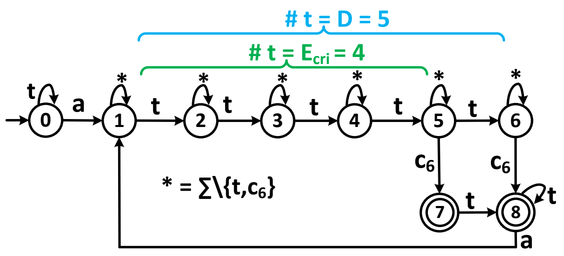


Figure 7.8: *Timing Specification Model H for PTG G′*

As mentioned earlier, model H represents all possible deadline-meeting sequences for G'. In order to illustrate this fact, let us try to find the deadline-missing sequence seq_1 = $ac_0s_{1,1}tc_1s_{2,1}tc_2s_{3,2}ttc_3s_{4,1}$ $tc_4s_{5,1}tc_5c_6$ (refer Figure 7.6) in model H shown in Figure 7.8. After proceeding through the states *State* 0 $\xrightarrow{a}$ *State* 1 $\xrightarrow{c_0}$ *State* 1 $\xrightarrow{s_{1,1}}$ *State* 1 $\xrightarrow{t}$ *State* 2 $\xrightarrow{c_1}$ *State* 2 $\xrightarrow{s_{2,1}}$ *State* 2 $\xrightarrow{t}$ *State* 3 $\xrightarrow{c_2}$ *State* 3 $\xrightarrow{s_{3,2}}$ *State* 3 $\xrightarrow{t}$ *State* 4 $\xrightarrow{t}$ *State* 5 $\xrightarrow{c_3}$ *State* 5 $\xrightarrow{s_{4,1}}$ *State* 5 $\xrightarrow{t}$ *State* 6 $\xrightarrow{c_4}$ *State* 6 $\xrightarrow{s_{5,1}}$ *State* 6, model H *gets blocked* due to the absence of a transition on t at *State* 6. More specifically, after processing the sub-string $ac_0s_{1,1}tc_1s_{2,1}tc_2s_{3,2}ttc_3s_{4,1}tc_4s_{5,1}$ of seq_1, the next event in seq_1 is t. However, t is not present at *State* 6. Thus, $L_m(H)$ does not contain the deadline-missing sequence seq_1.

The deadline-meeting sequence seq_2 ($= ac_0s_{1,1}tc_1s_{2,1}s_{3,2}tc_2s_{4,1}tc_3c_4t$ $s_{5,1}tc_5c_6$) can be retrieved by tracing the states *State* 0 $\xrightarrow{a}$ *State* 1 $\xrightarrow{c_0}$ *State* 1 $\xrightarrow{s_{1,1}}$ *State* 1 $\xrightarrow{t}$ *State* 2 $\xrightarrow{c_1}$ *State* 2 $\xrightarrow{s_{2,1}}$ *State* 2 $\xrightarrow{s_{3,2}}$ *State* 2 $\xrightarrow{t}$ *State* 3 $\xrightarrow{c_2}$ *State* 3 $\xrightarrow{s_{4,1}}$ *State* 3 $\xrightarrow{t}$ *State* 4 $\xrightarrow{c_3}$ *State* 4 $\xrightarrow{c_4}$ *State* 4 $\xrightarrow{t}$ *State* 5 $\xrightarrow{s_{5,1}}$ *State* 5 $\xrightarrow{t}$ *State* 6 $\xrightarrow{c_5}$ *State* 6 $\xrightarrow{c_6}$ *State* 8. Hence, $seq_2 \in L_m(H)$. Similarly, seq_3 can be traced in H.

7.2.4 Scheduler Synthesis

In order to determine only and all the sequences that satisfy deadline, minimum inter-arrival time, as well as resource and precedence constraints, we compose T and H to obtain the initial supervisor candidate M^0 $(= T||H)$.

Lemma 7.2.1. *$L_m(M^0)$ contains all the deadline-meeting and sporadic sequences in $L_m(T)$*

Proof. We know that $L_m(T)$ *includes deadline-meeting as well as deadline-missing sequences with arbitrary inter arrival times* for PTG G'. On the other hand, $L_m(H)$ contains all possible deadline-meeting, sporadic sequences. Since $L_m(M^0) = L_m(T) \cap L_m(H)$, $L_m(M^0)$ contains *all* the deadline-meeting and sporadic execution sequences in $L_m(T)$. □

Lemma 7.2.2. *$L_m(M^0)$ eliminates all sequences in $L_m(H)$ that violate resource and precedence constraints.*

Proof. Dual of Lemma 7.2.1. □

The sequences that violate deadline, resource, precedence and sporadicity constraints lead to *deadlock* states in M^0. That is, M^0 is *blocking*. Hence, we apply *trim* operation to remove the minimal number of states from M^0 until it becomes *non-blocking* and producing the resulting model M^1. Thus, $M^1 = trim(M^0)$. The steps towards the computation of the supervisor for correct schedule generation has been summarized in Algorithm 10.

ALGORITHM 10: COMPUTE SUPERVISOR

```
Input: PTG $G = (I, L, E, D, P)$, Number of processing cores $m$.
Output: $M^1$
1  Transform $G$ into $G'$;
2  Define the event set $\Sigma$;
3  for each task node $\tau_i \in I$ of $G'$ do
4      Build TDES model $T_i$ for $\tau_i$ (refer subsection 7.2.1);
5  Compute the composite task execution model, $T = ||_{i=0}^{n+1} T_i$;
6  Compute the length of the critical path $E_{cri}$;
7  Build TDES model $H$ representing timing specification (refer subsection 7.2.3);
8  Compute the initial supervisor candiate, $M^0 = T||H$;
9  $M^1 = trim(M^0)$;
10 return $M^1$;
```

Theorem 7.2.3. *$L_m(M^1)$ is: (1) controllable with respect to $L_m(M^0)$ and (2) optimal, producing the largest schedulable language corresponding to G'.*

Proof. All sequences which satisfy deadline, sporadicity, precedence and resource constraints, cannot lead to blocked states in M^0 (refer Lemmas 7.2.1 and 7.2.2). Hence, subsequent to the removal of blocked states resulting in the non-blocking model M^1, all these sequences remain preserved. Hence, $L_m(M^1)$ is controllable with respect to in $L_m(M^0)$. Also, $L_m(M^1)$ is optimal since it contains the exhaustive set of all valid scheduling sequences for PTG G'. □

$L_m(M^1)$ is empty if no valid scheduling sequence can be derived corresponding to G'.

Example (cntd.): We compute the initial supervisor candidate M^0 using the synchronous product of T (Figure 7.6) and H (Figure 7.8). Model M^0 is shown in Figure 7.9. It may be observed that the deadline-missing sequence $seq_1 \in L_m(T)$ leads to deadlock in M^0. Due to the application of the $trim$ operation, the portion of M^0 shown in red color will be removed from it. This makes the resulting model M^1 to be *non-blocking.* It may be noted that the marked behavior $L_m(M^1)$ retains the valid scheduling sequences seq_2 and seq_3 (refer Figure 7.6).

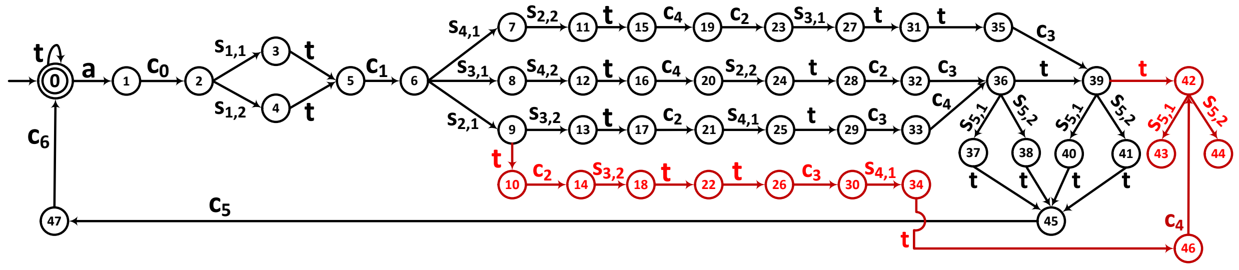


Figure 7.9: *Supervisor candidate: $M^0 = T||H$, Supervisor: M^1 (shown in black colour)*

7.3 Fault-tolerant Scheduler Synthesis

In this section, we extend the presented scheduler synthesis scheme to handle transient faults that may affect task execution. We make the following *assumptions*: (1) Any instance of PTG G' encounters at most w transient faults during its execution. (2) A task node may be affected by multiple faults during its execution. (3) A transient fault that occurs during the execution of task node τ_i is detected at the end of its execution and must be re-executed to tolerate the fault [72]. (4) Faults may be detected using hardware [16,98] or software [83,84] based error detection methods. The overhead corresponding to this detection mechanism is considered to be accounted as part of the execution time of each task node [52]. (5) The recovery overhead associated with task re-execution is assumed to be negligible.

Problem Statement-2: *Given a PTG G' to be executed on m homogeneous processing cores, design a supervisor containing the exhaustive set of all deadline, sporadicity, resource and precedence constraints satisfying scheduling sequences for G' that has the ability to tolerate the occurrence of at most w transient processor faults.*

The event set Σ: The fault-tolerant models discussed next includes all the events used for the fault-free models discussed earlier (Section 7.2.1). In addition, we now introduce two more events as part of Σ_i: (i) d_i: fault detection event for τ_i, and (ii) f_i: fault notification event corresponding to τ_i. Since, faults are induced by the environment, both d_i and f_i are considered to be *uncontrollable*. Therefore, the updated event set becomes: $\Sigma_0 = \{a, c_0\}$, $\Sigma_i = \{\{s_{i,j}|1 \leq i \leq n, 1 \leq j \leq m\} \cup \{c_i, d_i, f_i|1 \leq i \leq n\}\}$, $\Sigma_{n+1} = \{c_{n+1}\}$ and $\Sigma = \cup_{i=0}^{n+1}\Sigma_i \cup \{t\}$.

7.3.1 Fault-tolerant Task Execution Model

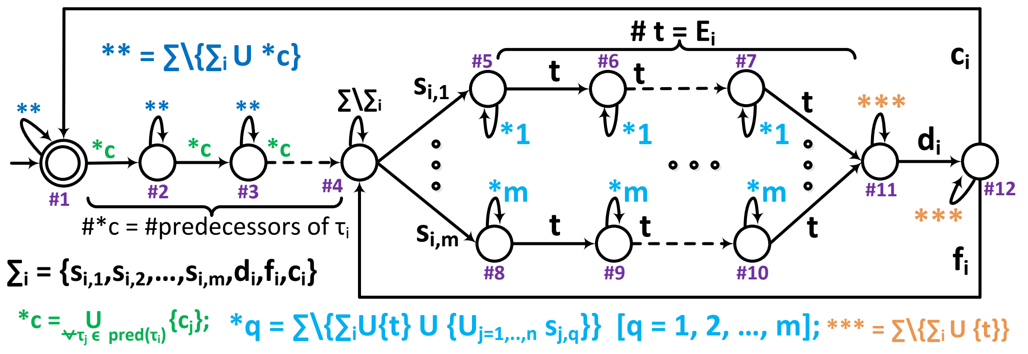

** = ∑\{∑ᵢ U *c}
#1
#2
#3
#4
**
*c
∑\∑ᵢ
#*c = #predecessors of τᵢ
∑ᵢ = {sᵢ,₁,sᵢ,₂,…,sᵢ,ₘ,dᵢ,fᵢ,cᵢ}
# t = Eᵢ
#5
#6
#7
#8
#9
#10
#11
#12
sᵢ,₁
sᵢ,ₘ
t
*1
*m
***
dᵢ
cᵢ
fᵢ
*c = U ∀τⱼ ∈ pred(τᵢ) {cⱼ}; *q = ∑\{∑ᵢU{t} U {Uⱼ₌₁,..,ₙ sⱼ,q}} [q = 1, 2, …, m]; *** = ∑\{∑ᵢ U {t}}

Figure 7.10: *Fault-tolerant Task execution model FT_i for intermediate node τ_i*

In order to design a fault-tolerant scheduler, the task execution model associated with intermediate task nodes (shown in Figure 7.3) has been modified. The TDES models for the source (T_0) and sink nodes (T_{n+1}) remain structurally same (although, the event set Σ now includes events d_i and f_i in it). T_0 and T_{n+1} are renamed as FT_0 and FT_{n+1}, respectively. The modified model FT_i for any intermediate task node τ_i is shown in Figure 7.10. States #1 to #11 in the modified model are same as the initial fault-free model T_i. We have newly introduced *State* #12. Subsequent to the execution of τ_i for E_i ticks on anyone of the processing cores, model FT_i reaches *State* #11. At this state, fault detection event d_i takes FT_i to *State* #12. If τ_i did not suffer a fault as FT_i moved from *State* #4 to *State* #11, then FT_i transits back to the initial state on c_i where it waits for the completion of its predecessor nodes corresponding to the next instance of G'. On the contrary, if the outcome of the fault detection process is positive, then τ_i must be re-executed to tolerate the fault. Hence, FT_i moves to *State* #4 from *State* #12 on the occurrence of the fault notification event f_i. It may be observed that model FT_i represents a behavior which allows τ_i to tolerate an arbitrary number of transient faults disregarding the deadline constraint. Using this modified FT_i for intermediate task nodes, we can obtain the composite fault-tolerant task execution model $FT = ||_0^{n+1} FT_i$.

Example: Let us consider the PTG G shown in Figure 7.11(a). It consists of three task nodes τ_1, τ_2 and τ_3 with execution times 1, 2 and 1, respectively. The minimum inter-arrival time between two consecutive instances of G is 10 time units. We assume

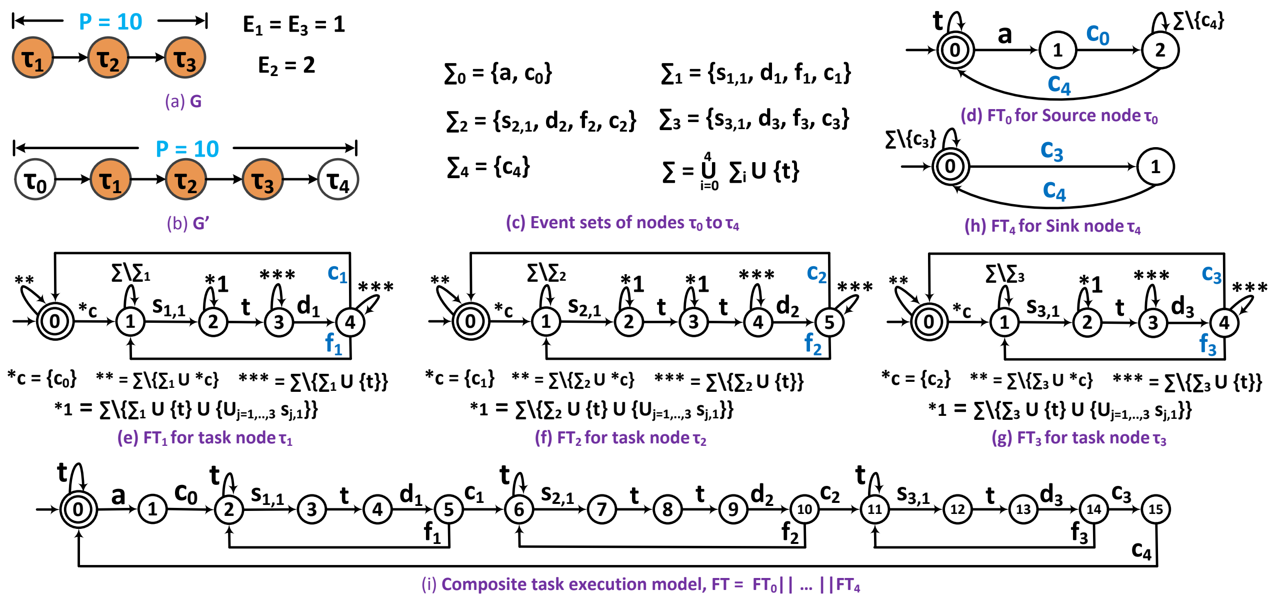


Figure 7.11: *The models associated with PTG G executing on a single core platform*

that G is executed on a single core and must tolerate at most one transient fault during the execution of any instance. The transformed model G', the event set Σ and the individual task execution models are shown in Figures 7.11(b) to (h). The composite task execution model FT $(= ||_{i=0}^{4} FT_i)$ is shown in Figure 7.11(i).

7.3.2 Fault-tolerance Specification

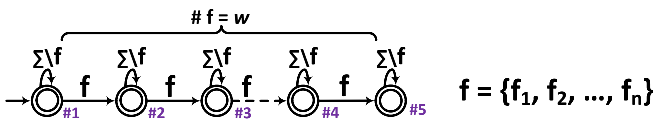


Figure 7.12: *Fault specification model F_w for w faults*

It may be noted that as FT_i can tolerate an arbitrary number of faults by ignoring deadlines, there composition FT also preserves the same property. However, our objective is to tolerate at most w faults in any instance of G'. Hence, we introduce a fault-tolerance specification model which captures this bound on the number of faults to be tolerated. Figure 7.12 shows the transition structure of the fault specification model F_w. This model contains $w + 1$ distinct states to count w faults. Model F_w allows transition to a distinct state whenever a fault event from the set $\{f_1, f_2, \ldots, f_n\}$ is encountered. All states in F_w are marked since the design must be equipped to tolerate any number of faults between 0 and w.

Given the composite fault-tolerant task execution model FT and fault-tolerance specification model F_w, we obtain the model FTw, by conducting the *synchronous product* operation, i.e., $FTw = FT || F_w$. *The marked behavior $L_m(FTw)$ contains all execution sequences that can tolerate at most w transient faults during the execution of a single instance of G'.* However, these sequences may violate the timing constraints related to deadline and minimum inter-arrival time.

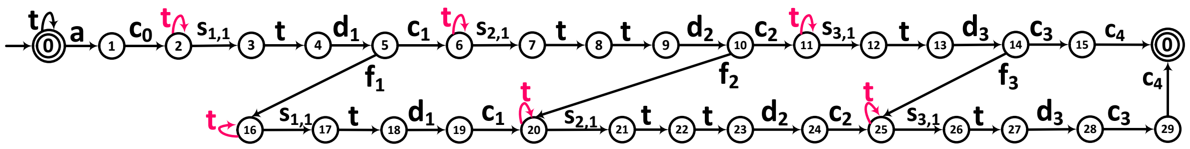


Figure 7.13: *Single fault-tolerant composite task execution model $FT1$*

Example (cntd.): In order to tolerate a single transient fault, fault specification model F_1 can be constructed according to Figure 7.12. Next, we compose FT (in Figure 7.11(i)) with F_1 to obtain the single fault-tolerant composite task execution model $FT1$ ($= FT || F_1$), as depicted in Figure 7.13. It may be noted that $L_m(FT1)$ contains all execution sequences that have the ability to tolerate at most one transient fault.

7.3.3 Scheduler Synthesis

In order to determine all sequences in $L_m(FTw)$ that satisfy the stipulated specifications related to timing, we construct the finite state automaton $MF^0 = FTw || H$. The TDES model for the timing specification H remain structurally same (although, the event set Σ now includes events d_i and f_i in it). Since $L_m(MF^0) = L_m(FTw) \cap L_m(H)$, $L_m(MF^0)$ contains all sequences in $L_m(FTw)$ that satisfy specifications on deadline, sporadicity, fault-tolerance, resource and precedence constraints. The sequences in $L_m(FTw)$ that do not satisfy at least one of the above constraints, lead to *deadlock* states in MF^0, i.e., M^0 is *blocking*. Hence, we may apply *trim* operation to remove the minimal number of states from MF^0 such that the resulting model (which we refers to MF^1) becomes *non-blocking*. That is, $MF^1 = trim(MF^0)$. Although, the resulting automaton MF^1 is *non-blocking*, it is not controllable, as proved in Theorem 7.3.1.

Theorem 7.3.1. *$L_m(MF^1)$ is not controllable with respect to $L(MF^0)$.*

Proof. Let us consider the set of fault events $\{f_1, f_2, \ldots, f_n\}$. After executing a common prefix s on both MF^0 and MF^1 from their respective initial states, if f_i is defined at a given state of MF^0, then f_i must also be defined in MF^1. According to Problem Statement 2, the scheduler must tolerate the occurrence of at most w faults per instance of G'. It may be noted that there is no restriction on the pattern in which faults can occur. Therefore, in the worst-case, all w faults may affect the execution of a single task node (say, τ_i) in G'. $L_m(MF^1)$ can be controllable only if all sequences in $L(MF^0)$ which represent scenarios where all w faults affect a single task node lead to marked states in MF^0. On the contrary, if any of such sequences in $L(MF^0)$ end up in a deadlock state subsequent to the occurrence of a fault event f_i, then the suffix that leads to the deadlock state in MF^0 will be removed during the construction of MF^1 through the *trim* operation. Now, $L_m(MF^1)$ does not contain a controllable sequence corresponding to the sequence in $L(MF^0)$ which represented the valid occurrence of the fault pattern which lead to the deadlock state in MF^0. Therefore, $sf_i \in \overline{L(MF^0)}$ does not necessarily imply $sf_i \in \overline{L_m(MF^1)}$. Hence, $L_m(MF^1)$ is not controllable with respect to $L(MF^0)$. $\square$

Since $L_m(MF^1)$ is not controllable, we cannot apply *trim* operation on MF^0 to obtain MF^1. Rather, we compute the minimally restrictive (or maximally permissive) supervisor that restricts the system behavior to the *supremal controllable sub-language of* $L_m(MF^0)$. We can compute this supremal controllable sub-part of MF^0 using the *Safe-State-Synthesis* algorithm [104]. This algorithm removes the minimal number of states from MF^0 until it becomes *controllable* as well as *non-blocking*. Let us denote the resulting model to be $MF^1 = supC(L_m(MF^0))$. The resulting behavior $L_m(MF^1)$ *is the optimal fault-tolerant schedulable language*.

Example (cntd.): Let us assume the deadline D of PTG G' (shown in Figure 7.11(b)) to be 5 time units. The timing specification model H for PTG G' is shown in Figure 7.11(j). Next, we compose $FT1$ (in Figure 7.13) and H to obtain MF^0 (shown in Figure 7.14(a)). Given MF^0, let us analyze the controllability of sequences in $L(MF^0)$ by considering four distinct example sequences. The first sequence seq_4 ($= ac_0s_{1,1}td_1c_1s_{2,1}ttd_2c_2s_{3,1}td_3c_3c_4 \in L_m(MF^0)$) is fault-free. The other three sequences seq_5 ($= ac_0s_{1,1}td_1\ f_1s_{1,1}td_1c_1\ s_{2,1}ttd_2c_2s_{3,1}td_3c_3c_4 \in L_m(MF^0)$), seq_6 ($= ac_0s_{1,1}td_1c_1s_{2,1}ttd_2c_2$

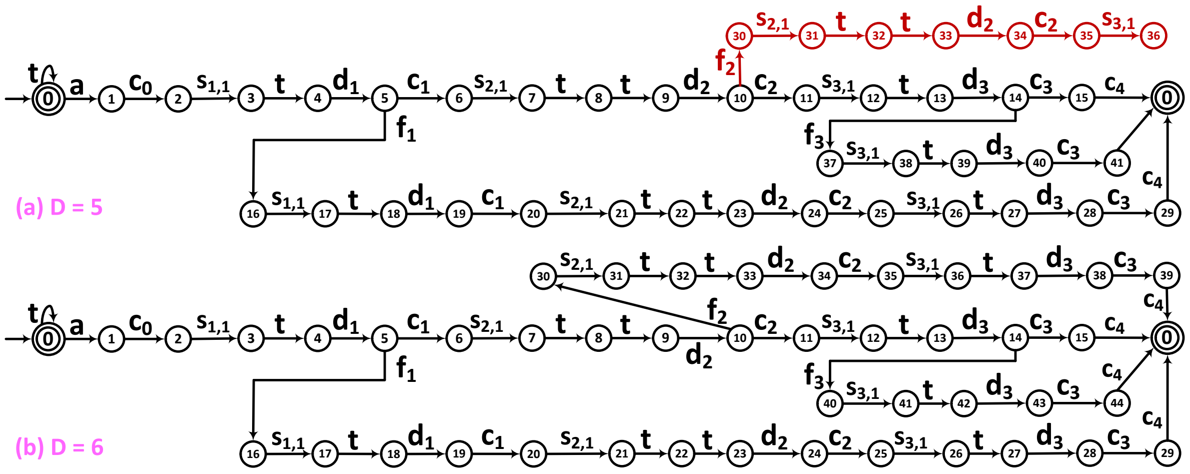


Figure 7.14: *Partial diagram of MF^0 $(= FT1||H)$ for Deadlines (a) $D = 5$, (b) $D = 6$.*

$s_{3,1}td_3$ $f_3s_{3,1}td_3c_3$ $c_4 \in L_m(MF^0)$) and seq_7 $(= ac_0s_{1,1}td_1c_1s_{2,1}ttd_2$ $f_2s_{2,1}ttd_2c_2$ $s_{3,1} \notin L_m(MF^0))$ represent scenarios where tasks τ_1, τ_3 and τ_2 is affected by a fault, respectively. The subsequences in seq_5, seq_6 and seq_7 marked in red represent the occurrence of the fault and subsequent re-execution of the fault affected task. It may be observed that while sequences seq_4, seq_5 and seq_6 are deadline-meeting, seq_7 is deadline-missing and lead to a deadlock state (*State* 36). Thus, $seq_7 \in L(MF^0)$ is uncontrollable.

If a *trim* operation is applied over MF^0 to obtain MF^1, then the suffix $f_2s_{2,1}ttd_2c_2s_{3,1}$ (marked with red color in Figure 7.14(a)) of $seq_7 \in L(MF^0)$ is discarded, with the remaining part of MF^0 representing MF^1. However, as the prefix $(ac_0s_{1,1}td_1c_1s_{2,1}ttd_2)$ of seq_7 is retained in MF^1 and there is no way to prevent the uncontrollable occurrence of f_2 at *State* 10, MF^1 becomes uncontrollable with respect to MF^0. Therefore, the operation $supC(L_m(MF^0))$ is used instead of *trim*, in order to obtain MF^1 by determining the maximally accessible part of MF^0 subsequent to the removal of prefixes from it which may lead to states where transitions representing the potential occurrence of uncontrollable events will eventually reach deadlock states. For the example under consideration, the application of $supC(L_m(MF^0))$ determines the accessible part of MF^0 subsequent to the removal of the prefix $ac_0s_{1,1}td_1c_1s_{2,1}ttd_2$, and this results in an empty set. This implies that the given PTG G' is not schedulable under the given constraints.

Now, suppose the deadline D of PTG G' (shown in Figure 7.11(b)) is increased from 5 to 6 time units. Under such a modified scenario, we compute the model MF^0 (Figure 7.14(b)) and $supC(L_m(MF^0))$. Here, $L_m(MF^1)$ $(= supC(L_m(MF^0)))$ is non-empty and contains execution sequences that can tolerate single transient faults while meeting timing requirements of G'.

7.4 Satisfying Performance Objectives

The above sections present a systematic methodology for determining the exhaustive set of all feasible scheduling sequences that satisfy a set of hard constraints related to the satisfaction of deadline, sporadicity, resource, precedence and fault-tolerance, associated with a given PTG. This set of feasible scheduling sequences may be further filtered to obtain the best schedule with respect to one or a combination of chosen performance parameters such as schedule length, power-dissipation, degree of fault-tolerance, resource consumption etc. In the following two subsections, we discuss mechanisms to optimize two such performance objectives: (i) makespan minimization, and (ii) maximizing fault-tolerance.

7.4.1 Makespan Minimization

We conduct makespan minimization on the set of scheduling sequences obtained as outcome from the fault-free scheduler synthesis scheme presented in subsection 7.2.1. For this purpose, we develop a state-space search and refinement procedure which takes M^1 as input and produces the model M^2 that contains the set of execution sequences in $L_m(M^1)$ whose makespan is minimum. This algorithm is essentially based on the idea of Breadth-First Search (BFS) and proceeds as follows:

1. Initialize each state q $(\in Q)$ of M^1 with elapsed time (denoted by $q.ET$) to be 0. Start the search operation from the initial state q_0 of M^1.

2. Find the set of states that can be reached by a single transition from the states that have been visited during last iteration.

3. For each newly reached state q_x, find the set of all immediate predecessors of q_x and denote this set as $Pred(q_x)$.

3a. If $|Pred(q_x)| = 1$: Let q_y be the only predecessor of q_x (i.e., $\delta(q_y, \sigma) = q_x$). Compute elapsed time at q_x based on whether or not q_x is reached on tick event (t):

$$q_x.ET = \begin{cases} q_y.ET + 1 & \text{if } \sigma = t \\ q_y.ET & \text{Otherwise} \end{cases} \tag{7.1}$$

3b. If $|Pred(q_x)| > 1$: Determine,

$$\forall q_y \in Pred(q_x), ET_y = \begin{cases} q_y.ET + 1 & \text{if } \sigma = t \\ q_y.ET & \text{Otherwise} \end{cases}$$

Now, $q_x.ET$ is obtained as the minimum over the ET_y values corresponding to all predecessors of q_x. That is, $q_x.ET = \min_y \{ET_y\}$. For all $q_y \in Pred(q_x)$, if $ET_y > q_x.ET$, remove the transition from q_y to q_x on σ, i.e., set $\delta(q_y, \sigma) = \emptyset$.

4. Repeat steps (2) and (3) until all states in M^1 are visited.

5. Since, we have removed some of the transitions in M^1, we perform reachability operation starting from the initial state q_0 of M^1 (transformed) to obtain the set of states that are reachable from q_0. The resulting model consisting only of safe reachable states is called M^2.

The marked behavior $L_m(M^2)$ contains all and only the execution sequences whose makespan is minimal. *Anyone of the execution sequences in $L_m(M^2)$ can be used to schedule tasks on-line.*

Example (cntd.): Application of the makespan minimization procedure on M^1 shown in Figure 7.9 results in M^2 with contains sequences with maximal makespan of 4 time units. For example, the sequence seq_3 ($= ac_0s_{1,1}tc_1s_{2,1}s_{3,2}tc_2s_{4,1}tc_3c_4s_{5,1}tc_5c_6$) with makespan 4 is retained in M^2 while seq_2 ($= ac_0s_{1,1}tc_1s_{2,1}s_{3,2}tc_2s_{4,1}tc_3c_4ts_{5,1}tc_5c_6$) having makespan 5 is eliminated. The gantt chart representation of the schedule using seq_3 (Figure 7.15(a)) as the scheduler, is shown in Figure 7.15(b).

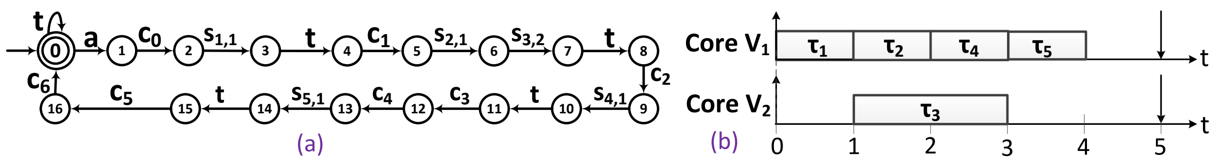


Figure 7.15: *(a) Supervisor, (b) Gantt chart representation*

7.4.2 Maximizing Fault-tolerance

The makespan minimization objective is achieved through an adaptation of BFS on the fault-free version of the final supervisor M^1. On the other hand, the procedure for deriving the maximally fault-tolerant supervisor is conducted through a systematic iterative application of the fault-tolerant supervisor synthesis mechanism discussed in Section 7.3. It may be noted that the synthesis mechanism in Section 7.3 produces a supervisor containing sequences which are tolerant to a stipulated number of faults. The maximization procedure starts by determining lower (w^{min}) and upper bounds (w^{max}) on the number of faults that may possibly be tolerated in the system at hand. Initially w^{min} is set to 0. The upper bound w^{max} is computed using the following two observations: (1) Given E_{cri}, the length of the critical path, and deadline D of PTG G', the maximum spare time for re-executing fault affected tasks, is given by: $D - E_{cri}$, (2) All sequences in the maximally fault-tolerant supervisor must be controllable even when all faults affect the task having maximal execution time among nodes in G'. Therefore, w^{max} is obtained as: $w^{max} = \lfloor \frac{D-E_{cri}}{E_{max}} \rfloor$, where, $E_{max} = \max\limits_{i} E_i$. In order to find the maximally fault-tolerant supervisor that can tolerate w^{opt} faults ($w^{min} \leq w^{opt} \leq w^{max}$), we apply an *interval bisection* based iterative sequence filtering technique presented in Algorithm 11. Finally, the resulting model MF^1 contains all possible execution sequences in which each instance of PTG G' can tolerate at most w^{opt} faults.

Example (cntd.): Let us consider the example PTG G' (Figure 7.11(b)) with D as 10 time units. Under this scenario, initial value of w^{min} is 0 and w^{max} is ($\lfloor \frac{10-4}{2} \rfloor =$) 3. Application of Algorithm 11 on G' returns a supervisor MF^1 with sequences which can tolerate at most ($w^{opt} =$) 3 faults.

ALGORITHM 11: MAXIMIZE FAULT-TOLERANCE

```
    Input: The composite fault-tolerant task execution model $FT$
    Output: $MF^1$, $w^{opt}$
 1  Initialize $w^{min}$ to 0;
 2  Initialize $w^{max}$ to $\lfloor \frac{D-E_{cri}}{E_{max}} \rfloor$;
 3  while $w^{min} \leq w^{max}$ do
 4      $w^{opt} = \lfloor (w^{min} + w^{max})/2 \rfloor$ ;
 5      Build TDES model $F_{w^{opt}}$ for fault specification to tolerate $w^{opt}$ faults (refer
        subsection 7.3.2);
 6      Compute $MFw = FT || F_{w^{opt}}$;
 7      Compute the initial supervisor candidate, $MF^0 = MFw || H$;
 8      $MF^1$ = Computer supremal controllable sup-part of $MF^0$;
 9      if the state set $Q$ of $MF^1$ is non-empty then
10          $w^{max} = w^{opt} - 1$;
11      else
12          $w^{min} = w^{opt} + 1$;
13  return $MF^1$, $w^{opt}$;
```

7.5 Complexity Analysis

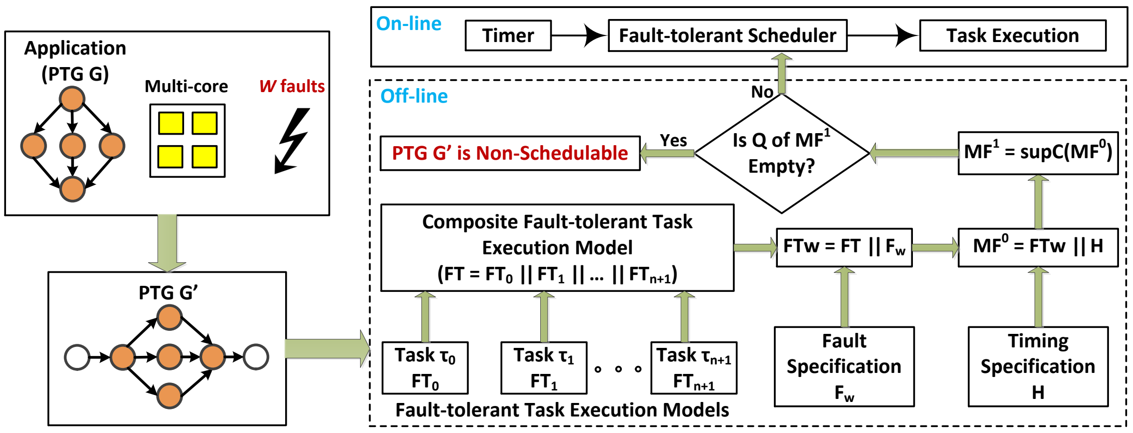


Figure 7.16: *Proposed Scheduler Synthesis Framework.*

A schematic diagram representing the overall flow of the fault-tolerant scheduler synthesis framework has been summarized in Figure 7.16. We now present a step-wise discussion on the complexity of the proposed (fault-tolerant scheduler) synthesis scheme.

1. The state-space complexity of FT_i (shown in Figure 7.10) is computed as follows: *State* #4 of FT_i has m branches emanating from it based on the events

$\{s_{i,1}, s_{i,2}, ..., s_{i,m}\}$ representing the start of task τ_i's execution on anyone of the m processors. With execution time being E_i, each of these m branches contain E_i states due to transitions on *tick* events. Therefore, the state-space complexity of FT_i becomes $\mathcal{O}(mE_i)$.

2. The state-space complexity of models FT_0, FT_{n+1} and F_w are constant.

3. The state-space of H (in Figure 7.7) is $\mathcal{O}(P)$ because distinct states are used to count the occurrence of each tick starting from the arrival to the minimum inter-arrival time of G'.

4. Given n DESs FT_0, FT_1, ..., FT_{n+1}, an upper bound on the number of states in the composite fault-tolerant task execution model FT is given by: $\prod_{i=0}^{n+1} |Q^{FT_i}|$, where $|Q^{FT_i}|$ $(= \mathcal{O}(mE_i))$ is the total number of states in FT_i. Similarly, the total number of states in the composite model FTw is: $|Q^{FT}| \times |Q^{F_w}|$. Thus, the total number of states in MF^0 is: $|Q^{FTw}| \times |Q^H|$.

5. The time complexity of the *Safe-State-Synthesis* algorithm [77] which is used for computing $L_m(MF^1)$ $(= supC(L_m(MF^0)))$ is polynomial in the size of MF^0.

It may be seen that the number of states in the composite model MF^0 grows *exponentially* as the number of tasks, processing cores and faults to be tolerated, increases. So, the proposed scheme may be highly time consuming and unacceptably memory intensive even for moderately large systems, thus severely restricting scalability, especially for industrial applications with many tasks. Over the years, *BDD* (*Binary Decision Diagram*) based symbolic synthesis mechanisms have proved to be a key technique towards the efficient computation of large finite state machine models including SCTDES based supervisors [77]. The steps to symbolically compute the composite model from a given set of individual models, and final supervisor can be found in the literature [38, 41, 77].

7.6 Experimental Results

In this section, we evaluate our proposed scheme through simulation based experiments.

Real-time Applications: We have considered three real-world parallel applications namely, *Fast Fourier Transform (FFT)*, *Gaussian Elimination (GaussElim)* and *Epigenomics* [55,103,109]. Figure 7.17(a) shows a commonly used sample PTG representation of the *FFT* application with $\rho = 4$. Here, ρ is used as the size of *FFT* and the total number of nodes $n = (2 \times \rho - 1) + \rho \times log_2\rho$, where $\rho = 2^y$ for some integer y [109]. The first part of the RHS expression gives a number of recursively invoked task nodes, and the second part gives the number of butterfly operation task nodes. Figure 7.17(b) shows an example of the *GaussElim* application with $\rho = 5$, and the total number of nodes is $n = (\rho^2 + \rho - 2)/2$ [109]. Figure 7.17(c) shows an example of the *Epigenomics* application with $\rho = 4$, in which the total number of nodes is given by $n = (4\times\rho)+4$ [55].

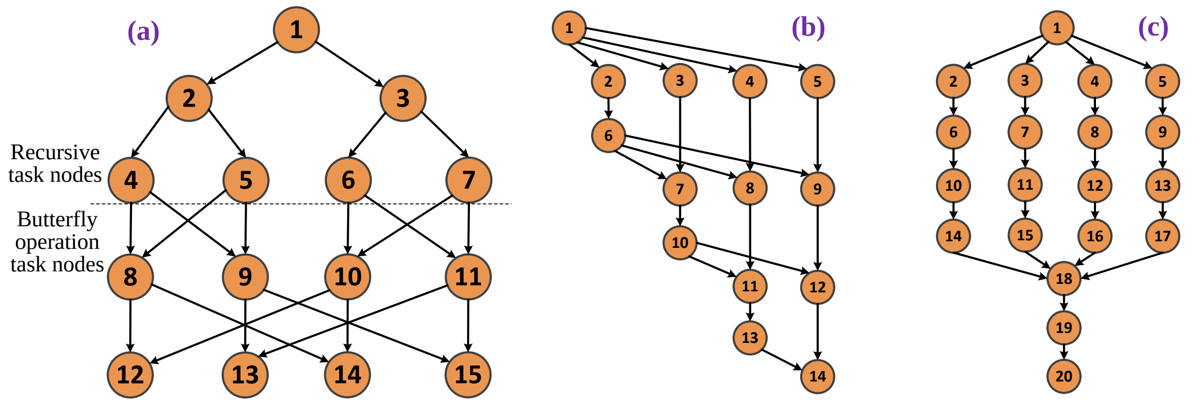


Figure 7.17: *(a) FFT Graph [103], (b) Gaussian Elimination Graph [109], and (c) Epigenomics [55]*

Simulation Setup: Execution times of task nodes (E_i) are randomly generated from a normal distribution with mean $\mu = 25$ and standard deviation $\lambda = 10$. For each PTG, we computed the summation of execution times of all nodes along its critical path, E_{cri}. Then, we set the deadline D such that $D > E_{cri}$ and $D > \lceil \frac{\sum_{n_i \in G'} E_i}{m} \rceil$. We also set $D = P$. Given a PTG and m cores, we have constructed their execution and timing specification models, using our Python scripts which are tailored according to the generalized models presented in our framework. The scheduler synthesis steps have been implemented using TTCT tool [2]. The computations were performed using a 24 Core Intel(R) Xeon(R) CPU E5-2420 v2 @ 2.2 GHz with 64 GB RAM running linux kernel 2.6.32.

Performance Metrics:

- *Schedule Length Ratio* (SLR) is obtained as the ratio of the actual schedule length to the summation of execution times of all task nodes in the critical path (CP). i.e., $SLR = \frac{makespan}{\sum\limits_{n_i \in CP} E_i}$.

- $Efficiency = \frac{Speedup}{m}$, where, $Speedup = \frac{\sum\limits_{\tau_i \in G'} E_i}{makespan}$.

- *Fault-tolerance* measures the number of transient faults to be tolerated per instance of G'.

Related works considered for comparison: In the context of production scheduling, several heuristic strategies have been proposed to solve the makespan minimization problem of PTGs on parallel machines [89, 91]. These strategies consider different *priority rules* to choose a task node for execution from the available ready nodes.

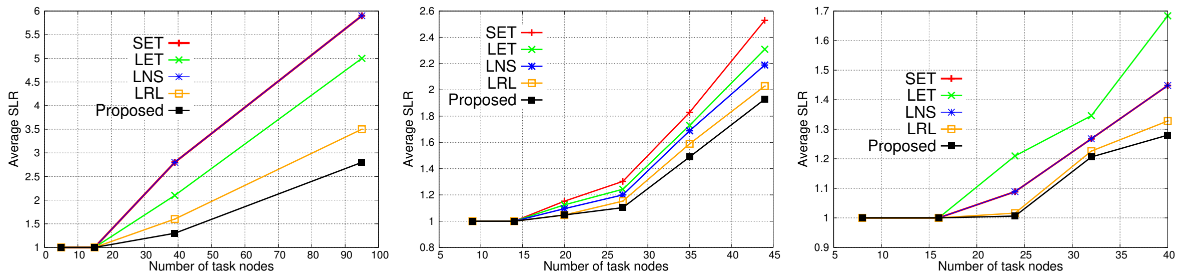
- *Longest Execution Time* (LET): the task node with the maximum execution time is selected.

- *Shortest Execution Time* (SET): the task node with the minimum execution time is selected.

- *Largest Number of Successors* (LNS): the task node with the largest number of immediate successors is selected.

- *Largest Remaining Load* (LRL): the task node with the largest load to be executed by its successors (starting from the immediate successors to the sink node) is selected.

In case of ties among multiple ready nodes, we have considered the node with the smaller execution time. We now compare our proposed work with the above approaches.

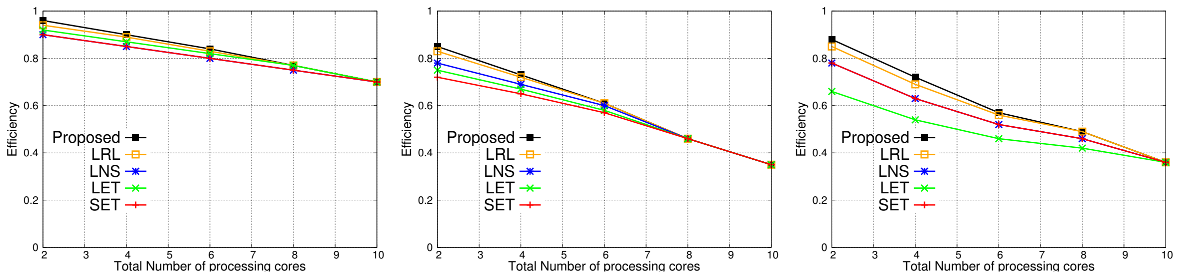


(a) *FFT* **(b)** *GaussElim* **(c)** *Epigenomics*

Figure 7.18: *Average Schedule Length Ratio.*

Experiment 1: ***Measuring Schedule Length Ratio***: This experiment measures the average schedule length ratios of applications under the different scheduling schemes mentioned above. For this purpose, we set the number of processing cores m to 4 and varied the number of task nodes n in the PTG. The results are reported in Figure 7.18. It may be observed that the average SLR is almost same when the number of nodes in the PTG is small. As the number of nodes increases, our proposed approach is able to minimize the makespan which results in significantly lower average SLR compared to other schemes. Among the other scheduling schemes, *LRL* performs better due to its ability to always choose the subset of ready task nodes whose execution will minimize makespan.

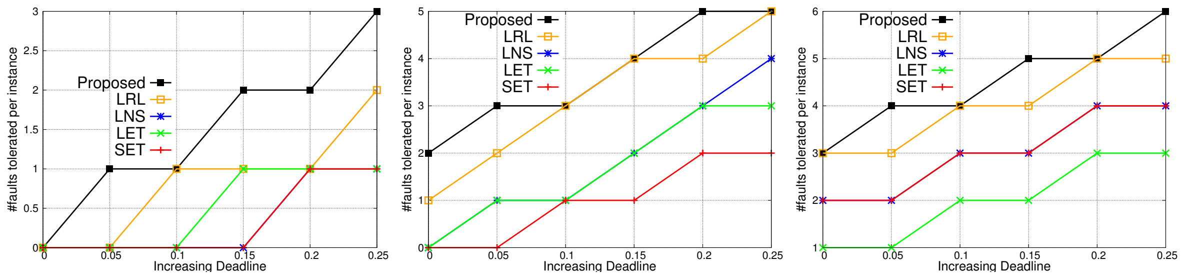
The performance of the remaining schemes (*SET*, *LET*, *LNS*) varies depending on the PTG structure and execution times of nodes. In case of the *FFT* application (Figure 7.18a), *LET* outperforms both *SET* and *LNS*. With respect to *GaussElim* application (Figure 7.18b), *LNS* performs better than *SET* and *LET*. This is because *GaussElim* contains a substantial number of nodes with multiple outgoing edges and *LNS* selects such nodes for execution to minimize the makespan. On the other hand, *SET* and *LNS* peform well compared to *LET* for the *Epigenomics* application (Figure 7.18c). This is due to the presence of task nodes with smaller execution times at the start of some of the parallel branches in *Epigenomics*. Since, *LET* delays the execution of such nodes in the presence of other ready task nodes with higher execution times, the execution of entire branch gets delayed. Consequently, makespan of the resulting schedule increases.



(a) *FFT* (b) *GaussElim* (c) *Epigenomics*

Figure 7.19: *Measuring efficiency by varying the number of processing cores*

Experiment 2: ***Measuring Efficiency***: This experiment measures and compares the efficiency of the different scheduling schemes. We set the number of task nodes n to 39, 44 and 40 for *FFT*, *GaussElim* and *Epigenomics*, respectively. The number of processing cores m is varied from 2 to 10. The results are presented in Figure 7.19. It may be observed that efficiency of our proposed scheme is better than other schemes when m is relatively small. As the number of cores increases, the processing capacity of the platform increases. Since, the computation demand of an application remains constant, efficiency of all schemes decreases.



(a) *FFT* (b) *GaussElim* (c) *Epigenomics*

Figure 7.20: *Measuring fault-tolerance by varying deadline*

Experiment 3: ***Measuring Fault-tolerance***: This experiment measures and compares the fault-tolerance capability of our proposed scheme along with other scheduling schemes. We set the number of task nodes n to 39, 44 and 40 for *FFT*, *GaussElim* and *Epigenomics*, respectively. The number of processing cores m is set to 4. Here, we have varied the deadlines associated with the applications. Specifically, we have increased the deadline by up to 25% (denoted by 0.25 in x-axis of Figure 7.20) from its initial value (which is represented by 0 in the x-axis of Figure 7.20). The results are presented in Figure 7.20. It may be observed that our scheme is able to tolerate a higher number of faults compared to other schemes. This is due to the ability of our scheme to minimize makespan so that the additional remaining time before deadline may be more effectively used to re-execute task nodes affected by faults.

7.7 Summary

In this work, first we have developed an offline scheduler synthesis framework for multicores executing real-time applications modeled as precedence-constrained task graphs. Then, we have extended our proposed scheme to handle transient processor faults that may affect task execution. The synthesized scheduler not only controls task execution but also drives the system to safe execution states such that all timing requirements associated with a given PTG is satisfied, even in the presence of uncontrollable transient processor fault events induced by the environment. Next, we have upgraded the scheduler synthesis framework to incorporate performance objectives such as makespan minimization and fault-tolerance maximization. Conducted experiments reveal the practical efficacy of our scheme. The next chapter summarizes the contributions of this dissertation and discusses a few possible extensions this research.

Chapter 8 Conclusions and Future Perspectives

8.1 Summarization

In spite of the progressively increasing computing capabilities of hardware platforms, effective allocation of computing resources to diverse competing applications is set to remain a daunting problem in safety-critical real-time embedded systems. This is because, these systems impose stringent timing, resource, performance and safety related constraints which must be accurately captured in the design process in order to ensure proper functioning of the system. In this context, *scheduling* acts as a vital design component which determines an appropriate co-execution order for the application tasks such that the desired performance objectives may be achieved while satisfying all constraints. This thesis deals with the scheduler synthesis for safety-critical real-time systems, using SCTDES as an underlying formalism. Scheduler synthesis based on the SCTDES framework starts with the modeling of individual components and their associated constraints. Such model-based design of safety-critical systems is very complex and challenging. Here, we enumerate a few challenging constraints which must be considered during the development of models for a given system configuration.

- The first challenge relates to the *modeling of timing constraints* associated with the various types of computation activities (i.e., tasks) that occur in such systems. The timing constraints of tasks are captured by their execution requirements, deadlines and arrival patterns (i.e., time-triggered, event-triggered).

- *Modeling of the execution behavior* of tasks (whether preemptive or non-preemptive) on a given processing platform. Also, the tasks may be *independent* or *precedence-constrained.*

- *Modeling of resource constraints* imposed by the underlying hardware platform on which the system is implemented. Over the years, the nature of execution platforms is witnessing a shift from single core to homogeneous and heterogeneous multi-cores, where several applications share the same platform [4].

- *Modeling of safety and performance related constraints* such as fault-tolerance, peak power minimization, makespan minimization etc.

In recent years, many researchers have shown that SCTDES can be used for the design of real-time schedulers [29, 53, 85, 87, 105–107]. Most of these approaches are targetted towards the scheduling of time-triggered (i.e., periodic) applications on a uniprocessor platform. A few of them have also attempted to handle event-triggered (i.e., aperiodic, sporadic) applications. However, these works have not been able to correctly model the characteristics / constraints associated with event-triggered tasks. Subsequently, the applicability of the above schemes have been limited only to a certain smaller subset of safety-critical systems. In this thesis, we have attempted to develop models which can address some of the major challenges highlighted above. In particular, scheduler synthesis schemes presented in Chapter 3 considered both the non-preemptive as well as preemptive scheduling of real-time sporadic tasks on uniprocessor platforms while, the remaining chapters (from 4 to 7) dealt with the scheduling of tasks on multiprocessor platforms. Specifically, Chapter 4 presents the fault-tolerant scheduler synthesis scheme which can tolerate a single/multiple permanent processor faults that may happen during the preemptive execution of set of dynamically arriving aperiodic tasks. In Chapter 5, we have attempted to synthesize a chip level peak power aware scheduler for a set of periodic tasks executing non-preemptively on a homogeneous multi-core platform. Next, we have considered the scheduling of periodic tasks on a heterogeneous processing platform in Chapter 6. Then in Chapter 7, our last contributory chapter, we proposed models that can be used to synthesize scheduler for real-time applications that are modeled as precedence-constrained task graphs. We now present brief summaries of these works in more detail.

In Chapter 3, our first contributory chapter, we have presented scheduler synthesis frameworks for real-time sporadic tasks executing (non-preemptively / preemptively) on uniprocessors. Although, in recent years, there has been a few significant works dealing with real-time scheduling using SCTDES, this is possibly the first work which addresses the scheduler synthesis problem for sporadic tasks. Our first framework considered the scheduling of dynamically arriving aperiodic tasks. Then, proposed models have been extended towards sporadic tasks. We have also illustrated the scheduler synthesis process using a motor network example. Next, our second framework proposed the synthesis of preemptive scheduler for sporadic tasks. The synthesized scheduler is guaranteed to be *work-conserving*, i.e., it will never keep the processor idle in the presence of ready to execute tasks. The scheduler is also able to support concurrent execution of multiple accepted tasks and correctly model the inter-arrival time requirement of a sporadic task. The practical applicability of our proposed framework has been illustrated using an industrial control system example.

In Chapter 4, we have presented a systematic way of synthesizing an *optimal fault-tolerant scheduler* for multiprocessor systems which processes a set of dynamically arriving aperiodic tasks. First, we have developed models that can be used to synthesize a single permanent processor fault-tolerant scheduler and then, extended it towards tolerating multiple faults. It may be noted that the state-space of the fault-tolerant scheduler synthesized using our proposed models increases exponentially as the number of tasks, processors, faults to tolerated increases. Hence, we have devised a mechanism to obtain a non-blocking supervisor using BDD based symbolic computation. This has helped us to control the exponential state space complexity of the optimal exhaustive enumeration oriented synthesis methodology.

Apart from providing tolerance against processor faults, safety-critical systems implemented on modern multi-core chips with high gate densities, must adhere to a strict power budget called TDP constraint, in order to control functional unreliability due to temperature hot-spots [78]. Hence, in Chapter 5, we have presented a systematic way of synthesizing an *optimal scheduler* for non-preemptive real-time tasks on multi-core systems with pre-specified chip level peak power constraints. The synthesis process starts with the development of the execution models for tasks and resource-constraint models for processing cores. Composition over these models ultimately provides the deadline and resource constraint satisfying supervisor. Further, the power-constraint violating sequences are filtered-out from the initial supervisor through a search and refinement mechanism. Subsequently, the targeted supervisor containing only scheduling sequences that dissipate minimal power is retained. To control state-space complexity involved in the synthesis process, a BDD based computation flow has been designed corresponding to the TDES oriented construction steps. Finally, we presented the experimental evaluation of our proposed framework using real-world benchmark programs. With respect to the state-of-the-art related works [67, 78], our framework is able to minimize the peak-power and improve the acceptance ratio of task sets.

In the earlier chapters, we have assumed the processing cores in a multi-core platform to be *identical* (i.e., *homogeneous*). However, the nature of processors in embedded systems is changing over the years. Therefore, in Chapter 6, we dealt with the synthesis of scheduler for a set of independent, non-preemptive periodic tasks executing on *heterogeneous* multi-cores. The synthesis process begins by developing the task execution models for each periodic task which effectively captures a task's, (i) *distinct execution requirements* on different processing cores, (ii) *deadline requirement* and (iii) *fixed inter-arrival time* constraint between any two consecutive instances. Then, we developed for each processor the specification models in order to enforce *resource-constraints*. Through a *synchronous product* on the individual models, we obtained the composite task execution model and specification model. Finally, the scheduler which contains the set of valid execution sequences, that can be used during on-line execution, is synthesized. We have also discussed the optimality and working of the scheduler synthesized using our proposed models.

In the earlier chapters, we have assumed the real-time tasks to be *independent*. However, they are also often modeled as PTG where nodes represent tasks and edges represent inter-task dependencies. Hence, in Chapter 7, as our last contributory chapter, we have delved towards the synthesis of scheduler for multi-cores executing real-time applications modeled as PTGs. We have also extended our proposed scheme to handle transient processor faults that may affect task execution. The synthesized scheduler not only controls task execution but also drives the system to safe execution states such that all timing requirements associated with a given PTG is satisfied, even in the presence of uncontrollable transient processor fault events induced by the environment. Next, we have upgraded the scheduler synthesis framework to incorporate performance objectives such as makespan minimization and fault-tolerance maximization. Simulation results revealed that the proposed scheme is able to *minimize the makespan* and *improve the fault-tolerance* capability of real-time systems executing PTGs, compared to the other existing schemes.

In summary, the work conducted as part of this thesis deals with the development of formal correct-by-construction scheduling methodologies for different types of real-time applications (including aperiodic, periodic, sporadic; with / without preemption etc.) on platforms ranging from uniprocessors to homogeneous as well as heterogeneous multiprocessors. Techniques necessary to imbibe fault-tolerance, power awareness etc. within the designed scheduler synthesis mechanisms, have also been proposed. BDD-based symbolic computation techniques have been devised to control the inherent state-space complexities associated with the scheduler synthesis mechanism.

8.2 Future Works

The work presented in this thesis leaves several open directions and there is ample scope for future research in this area. In this section, we present three such future perspectives.

- **Verification of the individual TDES models for correctness:** Given a system and its specification model, the supervisor synthesized using the SCTDES framework is provably *correct-by-construction* [18]. However, this correctness property is based on the assumption that the designs of individual system and specfication models are sound and complete. It is obvious that adhoc mechanisms cannot be employed to make such strong guarantees on the correctness of the individual models. Handcrafted models developed by the system engineers based on their domain knowledge and experience may often be prone to design flaws as depicted by us in [40]. For example, Park and Cho [86] presented a systematic way of computing a largest fault-tolerant and schedulable language that provides information on whether the scheduler (i.e., supervisor) should accept or reject a newly arrived aperiodic task. The computation of such a language is mainly dependent on the task execution model presented in their paper. However, *the task execution model is unable to capture the situation when the fault of a processor occurs even before the task has arrived.* Consequently, a task execution model that does not capture this fact may possibly be assigned for execution on a faulty processor. This problem has been illustrated with an appropriate example in [40].

To avoid the issues associated with individual models, we may apply automated verification techniques to identify and correct the presence of possible errors in the developed models. For this purpose, formal approaches such as *model checking*[1] seem to be an attractive alternative. In order to apply model-checking, the properties of interest, the correctness of which we desire to check, must first be identified. Given these properties, model checking through the following three steps: (1) The model $\mathcal{M}$ must be specified using the description language of a model checker; (2) The specification language must then be used to code the properties and this will produce a temporal logic formula ϕ for each specification; (3) Run the model checker with inputs $\mathcal{M}$ and ϕ. The model checker outputs *YES* if $\mathcal{M}$ satisfies ϕ (represented by $\mathcal{M} \models \phi$) and *NO* otherwise; in the later case, a counter-example in the form of a trace of the system behavior exhibiting the violation of the property is produced. The automatic generation of such *counter traces* is an important tool in the design and debugging of safety-critical systems. When a model check-

[1] *Model checking* [35] is an automatic, model-based, property-verification technique for finite state concurrent systems.

ing mechanism is applied on the handcrafter system and specification models, we can ensure the soundness and completeness of these models against our chosen properties of interest.

- **Scheduler synthesis for distributed processing platforms:** In the current research work, we have assumed a tightly-coupled interconnection of individual processing elements. Hence, inter-processor communication delays have been ignored. However, in a loosely-coupled distributed processing platform, such an assumption does not hold [103]. In addition, depending on the type and structure of the interconnection network between processing elements, the interprocessor message transmission time may significantly vary. To match the requirements of the distributed systems mentioned above, the research presented here must be suitably adapted.

- **Minimizing temporal overheads related to supervisor computation:** Given the initial supervisor candidate S_0, *SAFE STATE SYNTHESIS* (Algorithm 1) removes the minimal number of states from S_0 such that it becomes both *controllable* and *non-blocking*. This technique essentially conducts a search over the entire state-space of S_0 to remove blocking states. This search can be parallelized to minimize the total amount of time taken to compute the final supervisor [74].

Disseminations out of this Work

Journal Papers

1. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. “A design fix to supervisory control for fault- tolerant scheduling of real-time multiprocessor systems with aperiodic tasks.”International Journal of Control. Volume 88, Issue 11, Pages 2211-2216, 2015.

2. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. “Fault-Tolerant Preemptive Aperiodic RT Scheduling by Supervisory Control of TDES on Multiprocessors.”ACM Transactions on Embedded Computing Systems (TECS). 16(3), 87:1 - 87:25, 2017.

3. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. “Comments on Supervisory control for real-time scheduling of periodic and sporadic tasks with resource constraints [Automatica 45 (2009) 2597-2604].”Automatica, Volume 82, Pages 332-334, 2017.

4. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. “Supervisory Control Approach and its Symbolic Computation for Power-aware RT Scheduling.”IEEE Transactions on Industrial Informatics (TII), 2018. (Accepted).

5. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. “Optimal Scheduling of Nonpreemptive Periodic Tasks on Heterogeneous Multi-cores using Supervisory Control.”Automatica. (Provisionally accepted; Under third round of review).

6. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Multiprocessor Scheduling of Non-preemptive Sporadic Tasks using Supervisory Control of TDES - A BDD Based Approach." Journal of Parallel and Distributed Computing. (Under review).

7. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Optimal Work-conserving Scheduler Synthesis for Real-time Sporadic Tasks using Supervisory Control of Timed DES." Journal of Scheduling. (Under review).

Conference Papers

1. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Real-time Scheduling of Non-preemptive Sporadic Tasks on Uniprocessor Systems using Supervisory Control of Timed DES." American Control Conference (ACC). Pages 3212-3217, 2017.

2. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Fault-Tolerant Scheduling of Non-preemptive Periodic Tasks using SCT of Timed DES on Uniprocessor Systems." IFAC-PapersOnLine. Volume 50, Issue 1, Pages 9315-9320, 2017.

3. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Exact Task Completion Time Aware Real-Time Scheduling Based on Supervisory Control Theory of Timed DES." European Control Conference (ECC). Pages 1908-1913, 2018.

4. Rajesh Devaraj, Arnab Sarkar and Santosh Biswas. "Static Scheduling of Parallel Real-time Tasks using Supervisory Control of Timed Discrete Event Systems." American Control Conference (ACC). (Under review).

Other Publications

The following publications, co-authored by the author of this thesis, are relevant but not included in the thesis:

1. Satish Kumar, *Rajesh Devaraj* and Arnab Sarkar. "A Hybrid Offline-Online Approach to Adaptive Downlink Resource Allocation Over LTE." IEEE/CAA Journal of Automatica Sinica (JAS), 2018. (Accepted).

2. Piyoosh P Nair, *Rajesh Devaraj*, Argha Sen, Arnab Sarkar and Santosh Biswas. “DES based Modeling and Fault Diagnosis in Safety-critical Semi-Partitioned Real-time Systems.”IFAC-PapersOnLine. Volume 50, Issue 1, Pages 5029-5034, 2017.

3. Sanjay Moulik, *Rajesh Devaraj*, Arnab Sarkar and Arijit Shaw. “A Deadline-Partition Oriented Heterogeneous Multi-core Scheduler for Periodic Tasks.”The 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Pages 204-210, 2017.

4. Sanjay Moulik, *Rajesh Devaraj* and Arnab Sarkar. “HETERO-SCHED: A Low-overhead Heterogeneous Multi-core Scheduler for Real-time Periodic Tasks.”The 20th IEEE International Conference on High Performance Computing and Communications (HPCC), 2017 (Accepted).

5. Sanjay Moulik, *Rajesh Devaraj* and Arnab Sarkar. “COST: A Cluster-oriented Scheduling Technique for Heterogeneous Multi-cores.”IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2018 (Accepted).

6. Piyoosh P Nair, *Rajesh Devaraj* and Arnab Sarkar. “FEST: Fault-tolerant Energy-aware Scheduling on Two-core Heterogeneous Platform.”The 8th International Conference on Embedded Computing and System Design (ISED), 2018 (Accepted).

7. Sanjay Moulik, *Rajesh Devaraj* and Arnab Sarkar. “HEART: A Heterogeneous Energy-Aware Real-Time Scheduler.”The 32nd International Conference on VLSI Design and The 18th International Conference on Embedded Systems (VLSID), 2019 (Accepted).

8. Sanjit Kumar Roy, *Rajesh Devaraj* and Arnab Sarkar. “Optimal Scheduling of PTGs with Multiple Service Levels on Heterogeneous Distributed Systems.”American Control Conference (ACC), 2019 (Under review).

9. Sanjit Kumar Roy, *Rajesh Devaraj* and Arnab Sarkar. “Service Level Aware Optimal Scheduling of Task Graphs on Heterogeneous Distributed Systems.”IEEE Transactions on Industrial Informatics (TII) (Under review).

10. Satish Kumar, *Rajesh Devaraj*, Arnab Sarkar, Arijit Sur and Santosh Biswas. "Client-side QoE Management for SVC-DASH Video Streaming: A FSM Supported Design Approach." IEEE Transactions on Network and Service Management (TNSM) (Under review).

References

[1] "Intel Xeon Processor E5 v4 Product Family Datasheet, volume one: Electrical: [online] https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v4-datasheet-vol-1.pdf, June 2016." [Pg.6], [Pg.109], [Pg.133]

[2] "TTCT Software," [online]. Available: http://www.control.utoronto.ca/DES, 2017. [Pg.15], [Pg.31], [Pg.133], [Pg.145], [Pg.146], [Pg.183]

[3] S. Acharya and R. Mahapatra, "A dynamic slack management technique for real-time distributed embedded systems," *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 215–230, 2008. [Pg.74]

[4] A. Aminifar, P. Eles, Z. Peng, A. Cervin, and K.-E. Årzén, "Control-Quality Driven Design of Embedded Control Systems with Stability Guarantees," *IEEE Design & Test*, 2017. [Pg.190]

[5] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A predictable framework for safety-critical embedded systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600–1612, 2014. [Pg.1]

[6] J. H. Anderson, J. P. Erickson, U. C. Devi, and B. N. Casses, "Optimal semi-partitioned scheduling in soft real-time systems," *Journal of Signal Processing Systems*, vol. 84, no. 1, pp. 3–23, 2016. [Pg.2]

[7] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Euromicro Conference on Real-Time Systems*. IEEE, 2000, pp. 35–43. [Pg.24]

[8] S. Banerjee, G. Surendra, and S. Nandy, "On the effectiveness of phase based regression models to trade power and performance using dynamic processor adaptation," *Journal of Systems Architecture*, vol. 54, no. 8, pp. 797–815, 2008. [Pg.127]

[9] S. Baruah, "The federated scheduling of systems of conditional sporadic DAG tasks," in *International Conference on Embedded Software.* IEEE, 2015, pp. 1–10. [Pg.157]

[10] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems.* Springer, 2015. [Pg.3], [Pg.24], [Pg.141], [Pg.143], [Pg.156]

[11] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *Real-Time Systems Symposium.* IEEE, 2012, pp. 63–72. [Pg.157]

[12] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1-2, pp. 9–20, 2006. [Pg.141]

[13] S. K. Baruah, V. Bonifaci, R. Bruni, and A. Marchetti-Spaccamela, "ILP-based approaches to partitioning recurrent workloads upon heterogeneous multiprocessors," in *Euromicro Conference on Real-Time Systems.* IEEE, 2016, pp. 215–225. [Pg.23]

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996. [Pg.24]

[15] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Real-Time Systems Symposium, 1990. Proceedings., 11th.* IEEE, 1990, pp. 182–190. [Pg.54]

[16] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *On-Line Testing Symposium.* IEEE, 2003, pp. 144–148. [Pg.172]

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. [Pg.133]

[18] B. A. 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. [Pg.4], [Pg.8], [Pg.25], [Pg.26], [Pg.38], [Pg.142], [Pg.149], [Pg.193]

[19] R. Brandt, V. Garg, R. Kumar, F. Lin, S. Marcus, and W. Wonham, “Formulas for calculating supremal controllable and normal sublanguages,” *Systems & Control Letters*, vol. 15, no. 2, pp. 111–117, 1990. [Pg.95]

[20] D. Brooks, V. Tiwari, and M. Martonosi, *Wattch: A framework for architectural-level power analysis and optimizations.* ACM, 2000, vol. 28, no. 2. [Pg.133]

[21] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677–691, 1986. [Pg.9], [Pg.10], [Pg.95]

[22] A. Burns, “Scheduling hard real-time systems: a review,” *Software engineering journal*, vol. 6, no. 3, pp. 116–128, 1991. [Pg.2]

[23] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer, 2011, vol. 24. [Pg.2], [Pg.4], [Pg.17], [Pg.21], [Pg.42], [Pg.48], [Pg.54], [Pg.57], [Pg.70], [Pg.83], [Pg.141]

[24] G. C. Buttazzo, M. Bertogna, and G. Yao, “Limited preemptive scheduling for real-time systems: A survey,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013. [Pg.156]

[25] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* Springer, 2008. [Pg.25]

[26] B. Chattopadhyay and S. Baruah, “A Lookup-Table Driven Approach to Partitioned Scheduling,” in *Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 257–265. [Pg.23]

[27] J. Chen, C. Du, F. Xie, and B. Lin, “Scheduling non-preemptive tasks with strict periods in multi-core real-time systems,” *Journal of Systems Architecture*, vol. 90, pp. 72–84, 2018. [Pg.141]

[28] J. Chen, C. Du, F. Xie, and Z. Yang, “Schedulability analysis of non-preemptive strictly periodic tasks in multi-core real-time systems,” *Real-Time Systems*, vol. 52, no. 3, pp. 239–271, 2016. [Pg.141]

[29] P. C. 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. [Pg.xviii], [Pg.4], [Pg.28], [Pg.38], [Pg.39], [Pg.51], [Pg.52], [Pg.70], [Pg.142], [Pg.143], [Pg.157], [Pg.190]

[30] H. Chetto, M. Silly, and T. Bouchentouf, “Dynamic scheduling of real-time tasks under precedence constraints,” *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990. [Pg.39], [Pg.155]

[31] M. Chetto, “Optimal scheduling for real-time jobs in energy harvesting computing systems,” *IEEE Trans. Emerging Topics Comput.*, vol. 2, no. 2, pp. 122–133, 2014. [Online]. Available: https://doi.org/10.1109/TETC.2013.2296537 [Pg.5], [Pg.39]

[32] M. Chetto, *Real-time Systems Scheduling 1: Fundamentals.* John Wiley & Sons, 2014, vol. 1. [Pg.2]

[33] H. S. Chwa, J. Seo, J. Lee, and I. Shin, “Optimal Real-Time Scheduling on Two-Type Heterogeneous Multicore Platforms,” in *Real-Time Systems Symposium*, Dec 2015, pp. 119–129. [Pg.24]

[34] H. S. Chwa, J. Lee, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, “Global EDF schedulability analysis for parallel tasks on multi-core platforms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1331–1345, 2017. [Pg.12], [Pg.155], [Pg.157]

[35] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT press, 1999. [Pg.194]

[36] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM computing surveys*, vol. 43, no. 4, p. 35, 2011. [Pg.3], [Pg.23]

[37] M. L. Dertouzos, “Control Robotics: the Procedural Control of Physical Processes,” in *Information Processing.* North-Holland Publishing Company, 1974. [Pg.22]

[38] R. Devaraj, A. Sarkar, and B. Santosh, “Supervisory Control Approach and its Symbolic Computation for Power-aware RT Scheduling,” *IEEE Transactions on Industrial Informatics*, pp. 1–13, 2018. [Pg.182]

[39] R. Devaraj, A. Sarkar, and S. Biswas, “Real-time scheduling of non-preemptive sporadic tasks on uniprocessor systems using Supervisory Control of Timed DES,” in *American Control Conference*, pp. 3212–3217, 2017. [Pg.143], [Pg.157]

[40] R. Devaraj, A. Sarkar, and S. Biswas, “A design fix to supervisory control for fault-tolerant scheduling of real-time multiprocessor systems with aperiodic tasks,” *International Journal of Control*, vol. 88, no. 11, pp. 2211–2216, 2015. [Pg.194]

[41] R. Devaraj, A. Sarkar, and S. Biswas, “Fault-Tolerant Preemptive Aperiodic RT Scheduling by Supervisory Control of TDES on Multiprocessors,” *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 3, pp. 87:1–87:25, Apr. 2017. [Pg.110], [Pg.111], [Pg.182]

[42] F. Fauberteau, S. Midonnet, and M. Qamhieh, “Partitioned scheduling of parallel real-time tasks on multiprocessor systems,” *ACM SIGBED Review*, vol. 8, no. 3, pp. 28–31, 2011. [Pg.156]

[43] Z. Fei, *Symbolic supervisory control of resource allocation systems.* Chalmers University of Technology, 2014. [Pg.124]

[44] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, “A real-time scheduling service for parallel tasks,” in *Real-Time and Embedded Technology and Applications Symposium.* IEEE, 2013, pp. 261–272. [Pg.11], [Pg.155]

[45] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, “Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling,” *Real-Time Systems*, vol. 47, no. 5, p. 389, 2011. [Pg.24]

[46] R. Garibay-Martínez, G. Nelissen, L. L. Ferreira, P. Pedreiras, and L. M. Pinho, "Improved Holistic Analysis for Fork–Join Distributed Real-Time Tasks Supported by the FTT-SE Protocol," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1865–1876, 2016. [Pg.156]

[47] L. George, N. Rivierre, and M. Spuri, "Preemptive and non-preemptive real-time uniprocessor scheduling," Ph.D. dissertation, Inria, 1996. [Pg.39], [Pg.54]

[48] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, pp. 272–284, 1997. [Pg.74], [Pg.75]

[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *International Workshop on Workload Characterization.* IEEE, 2001, pp. 3–14. [Pg.132]

[50] E. Hjelm, "Safety-critical control in mixedcriticality embedded systems: A study on mixed criticality in the automotiveindustry," 2017. [Pg.1]

[51] W. Housseyni, O. Mosbahi, M. Khalgui, and M. Chetto, "Real-time scheduling of sporadic tasks in energy harvesting distributed reconfigurable embedded systems," in *13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November 29 - December 2, 2016*, 2016, pp. 1–8. [Online]. Available: https://doi.org/10.1109/AICCSA.2016.7945682 [Pg.48]

[52] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs," *ACM Trans. on Embedded Computing Systems*, vol. 11, no. 3, p. 61, 2012. [Pg.172]

[53] V. Janarthanan, P. Gohari, and A. Saffar, "Formalizing real-time scheduling using priority-based supervisory control of discrete-event systems," *IEEE Trans. on Automatic Control*, vol. 51, no. 6, pp. 1053–1058, 2006. [Pg.4], [Pg.38], [Pg.39], [Pg.142], [Pg.143], [Pg.157], [Pg.190]

[54] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *Real-Time Systems Symposium, 1991. Proceedings., Twelfth.* IEEE, 1991, pp. 129–139. [Pg.39], [Pg.141]

[55] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013. [Pg.xx], [Pg.183]

[56] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Transparent recovery from intermittent faults in time-triggered distributed systems," *IEEE Transactions on Computers*, vol. 52, no. 2, pp. 113–125, 2003. [Pg.155]

[57] A. Kanduri, M.-H. Haghbayan, A. M. Rahmani, P. Liljeberg, A. Jantsch, H. Tenhunen, and N. Dutt, "Accuracy-aware power management for many-core systems running error-resilient applications," *Transactions on Very Large Scale Integration Systems*, vol. 25, no. 10, pp. 2749–2762, 2017. [Pg.110], [Pg.111]

[58] J. C. Knight, "Safety critical systems: challenges and directions," in *International Conference on Software Engineering.* IEEE, 2002, pp. 547–550. [Pg.1]

[59] V. Kontorinis, A. Shayan, D. M. Tullsen, and R. Kumar, "Reducing peak power with a table-driven adaptive processor core," in *International symposium on microarchitecture.* ACM, 2009, pp. 189–200. [Pg.110], [Pg.111]

[60] H. Kopetz, "From a federated to an integrated architecture for dependable embedded systems," DTIC Document, Tech. Rep., 2004. [Pg.155]

[61] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, and Y. Halgand, "A survey of approaches combining safety and security for industrial control systems," *Reliability Engineering & System Safety*, vol. 139, pp. 156–178, 2015. [Pg.1], [Pg.5]

[62] C. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Computing Surveys*, vol. 46, no. 4, p. 48, 2014. [Pg.5], [Pg.6], [Pg.73], [Pg.155]

[63] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Real-Time Systems Symposium.* IEEE, 2010, pp. 259–268. [Pg.156]

[64] T. Langer, L. Osinski, and J. Mottok, “A Survey of Parallel Hard-Real Time Scheduling on Task Models and Scheduling Approaches,” in *International Conference on Architecture of Computing Systems.* VDE, 2017, pp. 1–8. [Pg.156]

[65] E. L. Lawler and J. Labetoulle, “On preemptive scheduling of unrelated parallel processors by linear programming,” *Journal of the ACM*, vol. 25, no. 4, pp. 612–619, 1978. [Pg.24]

[66] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach.* Mit Press, 2016. [Pg.1]

[67] J. Lee, B. Yun, and K. G. Shin, “Reducing peak power consumption inmulti-core systems without violatingreal-time constraints,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 1024–1033, 2014. [Pg.xix], [Pg.xxiii], [Pg.6], [Pg.11], [Pg.110], [Pg.111], [Pg.127], [Pg.128], [Pg.132], [Pg.135], [Pg.136], [Pg.192]

[68] J. Y.-T. Leung and J. Whitehead, “On the complexity of fixed-priority scheduling of periodic, real-time tasks,” *Performance evaluation*, vol. 2, no. 4, pp. 237–250, 1982. [Pg.23]

[69] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, “Global EDF scheduling for parallel real-time tasks,” *Real-Time Systems*, vol. 51, no. 4, pp. 395–439, 2015. [Pg.157]

[70] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *International Symposium on Microarchitecture.* IEEE, 2009, pp. 469–480. [Pg.133]

[71] F. Liberato, S. Lauzac, R. Melhem, and D. Mossé, “Fault tolerant real-time global scheduling on multiprocessors,” in *Euromicro Conference on Real-Time Systems.* IEEE, 1999, pp. 252–259. [Pg.74], [Pg.75]

[72] F. Liberato, R. Melhem, and D. Mossé, “Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems,” *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, 2000. [Pg.74], [Pg.75], [Pg.172]

[73] C. L. Liu and 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. [Pg.2], [Pg.21], [Pg.22], [Pg.23], [Pg.39], [Pg.70], [Pg.143]

[74] J. Markovski and H. Hu, “Is multicore supervisory controller synthesis in the Ramadge-Wonham framework feasible?” in *International Conference on Automation Science and Engineering.* IEEE, 2015, pp. 521–525. [Pg.29], [Pg.195]

[75] K. Meng, R. Joseph, R. P. Dick, and L. Shang, “Multi-optimization power management for chip multiprocessors,” in *International conference on Parallel architectures and compilation techniques.* ACM, 2008, pp. 177–186. [Pg.110], [Pg.111]

[76] S. Miremadi, Z. Fei, K. Åkesson, and B. Lennartson, “Symbolic Supervisory Control of Timed Discrete Event Systems,” *IEEE Transactions on Control Systems Technology*, vol. 23, no. 2, pp. 584–597, 2015. [Pg.10]

[77] S. Miremadi, B. Lennartson, and K. Akesson, “A BDD-based approach for modeling plant and supervisor by extended finite automata,” *IEEE Transactions on Control Systems Technology*, vol. 20, no. 6, pp. 1421–1435, 2012. [Pg.9], [Pg.10], [Pg.29], [Pg.96], [Pg.182]

[78] W. Munawar, H. Khdr, S. Pagani, M. Shafique, J.-J. Chen, and J. Henkel, “Peak Power Management for scheduling real-time tasks on heterogeneous many-core systems,” in *International Conference on Parallel and Distributed Systems.* IEEE, 2014, pp. 200–209. [Pg.6], [Pg.109], [Pg.110], [Pg.111], [Pg.132], [Pg.135], [Pg.136], [Pg.192]

[79] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, “Hierarchical power management for asymmetric multi-core in dark silicon era,” in *Annual Design Automation Conference.* ACM, 2013, p. 174. [Pg.110], [Pg.111]

[80] G. Nelissen, “Efficient optimal multiprocessor scheduling algorithms for real-time systems,” Ph.D. dissertation, Université libre de Bruxelles, 2012. [Pg.16]

[81] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic, “Techniques optimizing the number of processors to schedule multi-threaded tasks,” in *Euromicro Conference on Real-Time Systems.* IEEE, 2012, pp. 321–330. [Pg.156]

[82] G. Nelissen, H. Carvalho, D. Pereira, and E. Tovar, “Run-time monitoring environments for real-time and safety critical systems,” in *Demo Session, 22nd IEEE Real-Time Embedded Technology & Applications Symposium*, 2016. [Pg.2]

[83] B. Nicolescu, Y. Savaria, and R. Velazco, “Software detection mechanisms providing full coverage against single bit-flip faults,” *IEEE Transactions on Nuclear science*, vol. 51, no. 6, pp. 3510–3518, 2004. [Pg.172]

[84] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002. [Pg.172]

[85] S.-J. Park and K.-H. Cho, “Real-time preemptive scheduling of sporadic tasks based on supervisory control of discrete event systems,” *Information Sciences*, vol. 178, no. 17, pp. 3393–3401, 2008. [Pg.4], [Pg.38], [Pg.39], [Pg.190]

[86] S.-J. Park and K.-H. Cho, “Supervisory control for fault-tolerant scheduling of real-time multiprocessor systems with aperiodic tasks,” *International Journal of Control*, vol. 82, no. 2, pp. 217–227, 2009. [Pg.xviii], [Pg.73], [Pg.75], [Pg.80], [Pg.85], [Pg.106], [Pg.107], [Pg.108], [Pg.194]

[87] S.-J. Park and J.-M. Yang, “Supervisory control for real-time scheduling of periodic and sporadic tasks with resource constraints,” *Automatica*, vol. 45, no. 11, pp. 2597–2604, 2009. [Pg.4], [Pg.38], [Pg.39], [Pg.142], [Pg.143], [Pg.157], [Pg.190]

[88] R. M. Pathan, “Real-time scheduling algorithm for safety-critical systems on faulty multicore environments,” *Real-Time Systems*, vol. 53, no. 1, pp. 45–81, 2017. [Pg.71]

[89] M. L. Pinedo, *Scheduling: theory, algorithms, and systems.* Springer, 2016. [Pg.2], [Pg.54], [Pg.70], [Pg.155], [Pg.184]

[90] M. Qamhieh and S. Midonnet, “An experimental analysis of dag scheduling methods in hard real-time multiprocessor systems,” in *Conference on Research in Adaptive and Convergent Systems.* ACM, 2014, pp. 284–290. [Pg.155]

[91] K. E. Raheb, C. T. Kiranoudis, P. P. Repoussis, and C. D. Tarantilis, “Production scheduling with complex precedence constraints in parallel machines,” *Computing and Informatics*, vol. 24, no. 3, pp. 297–319, 2012. [Pg.184]

[92] A. M. Rahmani, M.-H. Haghbayan, A. Miele, P. Liljeberg, A. Jantsch, and H. Tenhunen, “Reliability-aware runtime power management for many-core systems in the dark silicon era,” *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 2, pp. 427–440, 2017. [Pg.110], [Pg.111]

[93] G. Raravi, B. Andersson, V. Nélis, and K. Bletsas, “Task assignment algorithms for two-type heterogeneous multiprocessors,” *Real-Time Systems*, vol. 50, no. 1, pp. 87–141, Jan 2014. [Pg.5], [Pg.24], [Pg.141]

[94] I. Ripoll, A. Crespo, and A. García-Fornes, “An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems,” *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 388–400, 1997. [Pg.75]

[95] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, “The shift to multicores in real-time and safety-critical systems,” in *International Conference on Hardware/Software Codesign and System Synthesis.* IEEE, 2015, pp. 220–229. [Pg.3]

[96] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013. [Pg.156], [Pg.157]

[97] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, *et al.*, “T-crest: Time-predictable multi-core architecture for embedded systems,” *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015. [Pg.2]

[98] D. Sciuto, C. Silvano, and R. Stefanelli, “Systematic AUED codes for self-checking architectures,” in *International Symposium on Defect and Fault Tolerance in VLSI Systems.* IEEE, 1998, pp. 183–191. [Pg.172]

[99] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, “The eda challenges in the dark silicon era: Temperature, reliability, and variability perspectives,” in *Annual Design Automation Conference.* ACM, 2014, pp. 1–6. [Pg.6]

[100] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” in *International Conference on Dependable Systems and Networks.* IEEE, 2002, pp. 389–398. [Pg.155]

[101] F. Somenzi, “CUDD: CU Decision Diagram Package 3.0.0,” 2015. [Pg.11], [Pg.98], [Pg.133]

[102] J. Sun, N. Guan, Y. Wang, Q. Deng, P. Zeng, and W. Yi, “Feasibility of fork-join real-time task graph models: Hardness and algorithms,” *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, p. 14, 2016. [Pg.156]

[103] H. Topcuoglu, S. Hariri, and M.-y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002. [Pg.xx], [Pg.183], [Pg.195]

[104] A. Vahidi, M. Fabian, and B. Lennartson, “Efficient supervisory synthesis of large systems,” *Control Engineering Practice*, vol. 14, no. 10, pp. 1157–1167, 2006. [Pg.28], [Pg.29], [Pg.117], [Pg.123], [Pg.176]

[105] X. Wang, Z. Li, and W. M. Wonham, “Dynamic multiple-period reconfiguration of real-time scheduling based on timed DES supervisory control,” *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 101–111, 2016. [Pg.4], [Pg.38], [Pg.39], [Pg.110], [Pg.111], [Pg.142], [Pg.143], [Pg.157], [Pg.190]

[106] X. Wang, Z. Li, and W. M. Wonham, “Optimal priority-free conditionally-preemptive real-time scheduling of periodic tasks based on DES supervisory control,” *IEEE Transactions on Systems, Man and Cybernetics: Systems*, vol. 47, no. 7, pp. 1082–1098, 2017. [Pg.4], [Pg.38], [Pg.39], [Pg.110], [Pg.111], [Pg.142], [Pg.143], [Pg.157], [Pg.190]

[107] X. Wang, Z. Li, and W. M. Wonham, “Priority-free conditionally-preemptive scheduling of modular sporadic real-time systems,” *Automatica*, vol. 89, pp. 392–397, 2018. [Pg.4], [Pg.143], [Pg.157], [Pg.190]

[108] W. M. Wonham and P. J. Ramadge, “On the supremal controllable sublanguage of a given language,” *SIAM Journal on Control and Optimization*, vol. 25, no. 3, pp. 637–659, 1987. [Pg.29], [Pg.32]

[109] G. Xie, J. Jiang, Y. Liu, R. Li, and K. Li, “Minimizing Energy Consumption of Real-Time Parallel Applications Using Downward and Upward Approaches on Heterogeneous Systems,” *IEEE Transactions on Industrial Informatics*, vol. 13, no. 3, pp. 1068–1078, 2017. [Pg.xx], [Pg.183]

[110] H. Zheng, Y. He, L. Zhou, Y. Chen, and X. Ling, “Scheduling of non-preemptive strictly periodic tasks in multi-core systems,” in *2017 International Conference on Circuits, Devices and Systems (ICCDS).* IEEE, 2017, pp. 195–200. [Pg.141]