Titre: Title:	A Forward On-The-Fly Approach for Safety and Reachability Controller Synthesis of Timed Systems
Auteur: Author:	Parisa Heidari
Date:	2012
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Heidari, P. (2012). A Forward On-The-Fly Approach for Safety and Reachability Controller Synthesis of Timed Systems [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/979/

UNIVERSITÉ DE MONTRÉAL

A FORWARD ON-THE-FLY APPROACH FOR SAFETY AND REACHABILITY CONTROLLER SYNTHESIS OF TIMED SYSTEMS

PARISA HEIDARI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
NOVEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

A FORWARD ON-THE-FLY APPROACH FOR SAFETY AND REACHABILITY CONTROLLER SYNTHESIS OF TIMED SYSTEMS

présentée par : HEIDARI Parisa
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. QUINTERO Alejandro, Doct., président.
Mme BOUCHENEB Hanifa, Doctorat, membre et directrice de recherche.
M. BELTRAME Giovanni, Ph.D., membre.
M. BENTAHAR Jamal, Ph.D., membre.

To Mom and Dad,
for their unfailing love...

To Fariba and Keivan
for being there!

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Hanifa Boucheneb, for having led me in this research. I appreciate her valuable directives and continuous moral support. She came with me step by step along the way. I'm honored to have worked with her and had the chance to profit from her experiences.

I would also like to thank Dr. Rachid Hadjidj for his collaboration in this research.

RÉSUMÉ

Cette thèse s'intéresse à la synthèse de contrôleurs pour des systèmes temps réel (systèmes temporisés). Partant d'un système temps réel modélisé par un réseau de Petri temporel composé de transitions contrôlables et non contrôlables (TPN), le contrôle vise à forcer, en restreignant les intervalles de franchissement des transitions contrôlables, le système à satisfaire les propriétés souhaitées.

Nous proposons, dans cette thèse, un algorithme pour synthétiser de tels contrôleurs pour des propriétés de sûreté et d'accessibilité. Cet algorithme, basé sur la méthode de graphe de classes d'états, calcule à la volée les classes d'états atteignables du TPN tout en collectant progressivement les sous-intervalles de tir à éviter, afin de satisfaire les propriétés souhaitées. Avec cet algorithme, il n'est plus nécessaire de calculer les prédécesseurs contrôlables et de partitionner récursivement les classes d'états jusqu'à atteindre un point fixe, comme c'est le cas dans les autres approches basées sur l'exploration, en avant et en arrière, de l'espace des états du système. Nous prouvons formellement la correction de l'algorithme, puis nous montrons que dans la catégorie des contrôleurs basés sur la restriction des intervalles de tir, l'algorithme, proposé dans cette thèse, synthétise un contrôleur optimal (le plus permissif possible).

Afin d'atténuer davantage le problème d'explosion combinatoire, nous montrons comment combiner cette approche avec une abstraction par l'inclusion, par union-convexe ou par enveloppe-convexe. Nous montrons également comment exploiter cet algorithme pour générer des contrôleurs décentralisés.

Enfin, nous proposons d'appliquer cet algorithme pour contrôler des TPN par des chronomètres. Notre algorithme permet de partitionner les intervalles des transitions en "bons" et "mauvais" sous-intervalles (à éviter). L'idée est d'utiliser des chronomètres pour suspendre les tâches (transitions) durant leurs mauvais sous-intervalles et les activer dans leurs "bons sous-intervalles". Il s'agit donc de contrôler les réseaux de Petri temporels en associant des chronomètres aux transitions contrôlables, pour obtenir ainsi des réseaux de Petri temporels contrôlés.

ABSTRACT

This thesis deals with controller synthesis for real time systems (timed systems). Given a real time system modeled as a Time Petri Net (TPN) with controllable and uncontrollable transitions, the control aims at forcing the system to satisfy properties of interest, by limiting the firing intervals of controllable transitions. We propose, in this thesis, an algorithm to synthesize such controllers for safety / reachability properties.

This algorithm, based on the state class graph method, computes on-the-fly the reachable state classes of the TPN while collecting progressively firing subintervals to be avoided so that the property is satisfied. It does not need to compute controllable predecessors and then split state classes until reaching a fixpoint, as it is the case for other approaches based on backward and forward exploration of state space of the system. We prove formally the correctness of the algorithm and show that, in the category of state dependent controllers based on the restriction of firing intervals, the algorithm proposed in this thesis, synthesizes maximally permissive controllers.

In order to attenuate the state explosion problem, we show how to combine efficiently this approach with an abstraction by inclusion, convex union or convex hull. Afterwards, we discuss the compatibility of this method with distributed systems and decentralized controllers.

Finally, we apply this algorithm to control TPN with controllable and uncontrollable transitions by stopwatch. In this approach, we find the subintervals violating the given properties and our objective is to suspend the tasks (transitions) during their bad subintervals and to resume them later. The controller is synthesized through the same algorithm already introduced. In this approach, we suggest to control time Petri nets by associating stopwatches to controllable transitions and to achieve a controlled time Petri nets.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY AND ABBREVIATIONS

BCFCF	Backward Conflict and Forward Concurrent Free
BCS	Backward Concurrent Structure
CTL	Computational Tree Logic
DBM	Difference Bound Matrix
DES	Discrete Event System
GMEC	Generalized Mutual Exclusion Constraints
IHTPN	Time Petri Nets with Inhibitor Hyperarcs
LTL	Linear Temporal Logic
SWA	Stopwatch Automata
SCG	State Class Graph
SWPN	Post and Pre-initialized Stopwatch Petri Nets
TA	Timed Automata
TCTL	Timed Computational Tree Logic
TL	Temporal Logic
TPN	Time Petri Nets
TS	Transition System
TTS	Timed Transition System
ZBG	Zone Based Graph

CHAPTER 1

INTRODUCTION

1.1 Motivation and problem statement

Real time systems are systems with specific timing requirements. In real time systems, well functionality is a two-sided concept where correctness is as important as respecting timing requirements. A request cannot be answered later than a given delay. These systems are widely used in our daily life from little electronic devices like a digital camera, to traffic lights, avionic systems and missile firing.

In the real time field, *critical systems* refer to the systems where a failure in functionality is too costly, critical or harmful. Considering the cost of a failure in such systems, their well functionality should be guaranteed at the design level, before implementation. In fact, a design needs to be verified formally before being implemented. What we mean by verification is to prove that a system is safe or to find a counterexample (i.e. an error or a failure). Verification is different from simulation. Simulation gives an idea about how a system works, it may show an error but it does not prove the absence of errors.

With this objective, the system is first modeled based on the mathematical expression of its behaviors and then, some techniques such as model-checkers are applied on the model to verify its correctness or give a counterexample (Fig.1.1). What we mean by correctness is satisfaction of the given properties and meeting the given timing requirements. Some known models are modeling languages, (timed) automata and (time or timed) Petri nets. Albeit having a mathematical nature, the two last ones are represented graphically, making them more user friendly and understandable. Automata and Petri nets are bi-similar most of the time. However, in some contexts one is more suitable than the other. For example, Petri nets are more convenient for modeling parallelism. An infinite system can be modeled by finite Petri nets. In general, the semantics of the models are defined by a transition system. Properties are formulas to declare requirements and desired specifications. Formulas are based on Temporal Logics (TL).

Model-checking is effective in locating bottlenecks. In the case a system does not guarantee to satisfy the properties of interest, a complementary object, a controller, is needed to

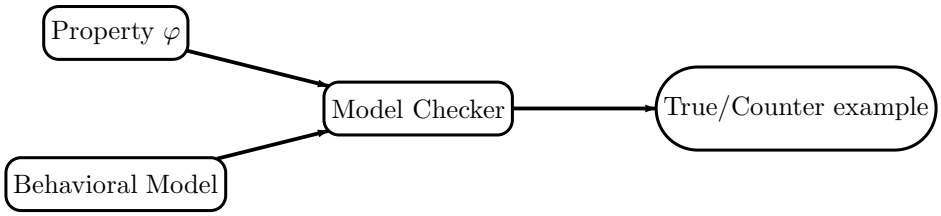


Figure 1.1 Model-checking.

fix the failures and guarantee the satisfaction of the given properties. Finding a controller for a system to force a given property is the subject of the *controller synthesis*. A model is analyzed to design the suitable controller using some algorithms. Many researches and case studies have been done to find an appropriate controller to guarantee the satisfaction of the property of interest. The objective is to have a controller running in parallel with the system under study and preventing any violation of given properties. In other words, another agent is added to the system such that the model-checker proves that the new compound system is safe and correct.

Model-checking detects the issues of an existing model and controller synthesis provides a solution to resolve them. In fact, a controller will make an open-loop system to be closed-loop. In an open-loop system, environment can affect the system in such a way that it violates the desired specifications. A closed-loop system adapts its reactions against the environmental undesired behaviors and keeps the requirements satisfied. Controller synthesis or supervisory controller, as in model checking, categorizes the properties and finds an algorithm to synthesize them.

In the concept of controller synthesis, two questions, *Control Problem* and *Controller Synthesis Problem* need to be addressed. The first question, Control Problem, investigates if for a given system $\mathcal{S}$ and a property φ, a controller C exists for the system $\mathcal{S}$ such that when $\mathcal{S}$ is running in parallel with C, the property φ is satisfied. The *Controller Synthesis Problem*, answers the question of if such controller exists, whether or not there is a solution to implement it.

The concept of controller synthesis is also described by game theory (Altisen *et al.*, 2005). Game theory considers a problem as a game with some players and a game *strategy*. A strategy is simply the actions to be played in each situation. Hence, game theory describes the

controller synthesis for a timed system, as a timed game with two players, the system under study against the environment. The control problem is then declared as: "Is there a strategy to make the controller the winner of the game?". In such a game, actions are partitioned into two disjoint sets, *controllable* and *uncontrollable*. Controllable actions are those that can be managed by the controller (forced to happen or prevented from happening). Uncontrollable actions are those that the controller has no control on. Players have equal chances to win. The control problem consists in searching a strategy such that the environment never wins the game against the controller. A strategy consists of the sequence of actions leading to the winning game outcome where the controller always wins against the environment.

In the controller synthesis, actions are observable or unobservable (Bouyer *et al.*, 2005). While an unobservable action is uncontrollable by default, an observable action is either controllable or uncontrollable. As an example, suppose a multitasking system with different processes, each of them having an execution time and period; and they access some possibly shared resources. Process arrival depends on its period and its termination depends on the execution time. Then, both process arrival and termination of execution are uncontrollable, whereas access to the shared resources and starting the execution are controllable. Unobservable actions are most internal actions with no available output event. An example of unobservable action is an internal counter in a digital system. With each clock cycle, the counter is modified, but it is unobservable, unless it is captured by a predefined value; at this time, an observable action happens and the external observer is aware.

There are various approaches in the literature to control different properties. Most of the available solutions do not take into account timing properties. In order to control these kinds of properties in timed models (Timed Automata and Time Petri Nets), several approaches have been proposed in the literature (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998). Two known methods are the backward fixpoint method and the backward-forward on-the-fly method. The first one is proposed for time Petri nets but it cannot guarantee to give a controller when it exists (Gardey *et al.*, 2006b). The second one is proposed for timed automata and guarantees to give the controller when it exists (Cassez *et al.*, 2005).

However, both above-mentioned approaches require some expensive operations such as calculating the difference between states. The difference between two states is not necessarily one state and then may result in several states which need to be handled separately. It would be interesting to investigate if there is an approach which does not need to split states.

In the context of controller synthesis, a scheduler is a kind of controller that manages shared resources and timing specifications of the system. In a multitasking system, a newly arrived task waits for accessing resources. After accessing the required resources, each task uses those resources for its execution time and then releases them. If the task is periodic, it stays in a passive state before its next arrival. A scheduler cannot modify timing specifications of the system such as period and execution time. In order to manage the shared resources among different periodic tasks with different levels of priority, the solution is to suspend a task with lower priority and let the other tasks with higher priorities to use the resources. Then, the suspended task is retrieved until it finishes its execution. When a task is suspended, the execution time is not progressing. It is interesting to see if controller synthesis can be used to synthesize a scheduler.

1.2 Thesis Contributions

This thesis consists of a number of contributions. We are interested in modeling the behavior of the real time systems by time Petri nets. Our main objective is to investigate controller synthesis of a real time system modeled by time Petri nets for safety and reachability properties.

Contribution 1: Safety controller synthesis

Our first contribution is an algorithm for controller synthesis of safety properties in time Petri nets. The goal is to have some general algorithms achieving suitable controllers and replacing case based solutions where the approach works for a particular case study. Our proposed algorithm is a forward on-the-fly semantic approach and is based on processing of the state class graph of the model. Our approach answers both control problem and controller synthesis problem mentioned above. It gives a controller if it exists. If the algorithm fails to give the controller, the controller does not exist. We prove that our approach gives a maximally permissive controller and apply the solution on some case studies.

Contribution 2: Reachability controller synthesis

In our next contribution, we study reachability controller synthesis. In our first contribution, we have suggested a forward on-the-fly algorithm for controller synthesis of real time systems modeled by time Petri nets. In the second contribution, we extend the algorithm to reachability properties. We prove the correctness of our algorithm and show that it is maximally permissive. We test it on some case studies.

Contribution 3: Optimization of the suggested approach by abstraction

In semantic approach, the algorithm processes the state space of the model and then, state space explosion is a common issue. In this thesis, we optimize the devised algorithm and investigate how to combine our approach with different methods of abstraction. Abstraction consists in agglomeration of some similar states and helps to have a more compact state space while decreasing the risk of state space explosion. During the construction of an abstraction, each newly computed abstract state is compared with the previously computed ones. We discuss the possibility of abstraction by inclusion and convex union. We also investigate if it is possible to use abstraction by convex hull. In the literature it was never proven if abstraction by convex hull preserve boundedness property. In this thesis, we present a new result. We give a counterexample to show that in our context, abstraction by convex hull is less appropriate as it does not preserve the boundedness property of TPN. Some case studies are also presented to show the scalability and effectiveness of each of the above-mentioned abstraction methods.

Contribution 4: Decentralized implementation on modular systems

Being optimized by abstraction, our algorithm is a good candidate for controller synthesis of large-scale systems. Large-scale systems are usually modular by their nature. In the other contribution, implementation of our centrally synthesized controllers on modular system is studied. We discuss how to implement the controller computed by the devised algorithm on modular systems and achieve a set of decentralized controllers. We answer the controller synthesis problem and investigate if our approach is implementable on modular, large-scale systems. Note that, we investigate how to implement a centralized computed controller on a decentralized system rather than synthesizing a "decentralized controller".

Contribution 5: Preemptive controller and scheduling

Our next contribution is to investigate if our controller synthesis approach can be used to synthesize a scheduler for managing shared resources and timing specifications. A scheduler needs to suspend and retrieve some tasks. Clocks are stopped during suspension. Our approach in its original form can initialize a clock but it cannot stop it. In this contribution, we synthesize the controller (scheduler) using the stopwatch which provides the possibility of suspending and retrieving the clocks. The solution we suggest in this thesis, is to synthesize the time Petri net model and to carry out the controller (scheduler). Then, the scheduler equips the appropriate transitions with stopwatch and controls the model by suspending the transitions. In other terms, we assume that every controllable action can be suspended (be

associated with stopwatch). This way considering, first, we calculate some subintervals in which the system violates the given properties and then the controller suspends the corresponding transition during those subintervals. This approach is very useful and interesting for preemptive scheduling purposes and managing shared resources.

The contributions of this thesis can be summarized as follows:

- Proposal of a forward on-the-fly algorithm for controller synthesis in time Petri nets. Formalization, proof of correctness and calculation of the complexity for the proposed algorithm. Providing some case studies.
- Extending the devised algorithm to reachability controllers. Formalization and proof of correctness. Providing some examples.
- Optimization of the proposed algorithm by state space abstraction. Investigating how to combine some known abstraction methods with the proposed approach. Providing some scalability analysis through some case studies.
- Investigation of the implementation of the proposed algorithms on large-scale modular systems. Discussing independent local controllers and local controllers with intercommunication among them.
- Extending the suggested controller synthesis approach to Petri nets equipped with stopwatch in order to support preemptive scheduling.

1.3 Impact and potentials of the thesis

In this thesis, we suggest an algorithm for controller synthesis of real time systems, modeled by time Petri nets. We propose a semantic approach which is not dependent on the specifications of a particular system. The available solutions in the literature for controller synthesis of real time systems are often case based and are dependent on the specifications of the system under study (Buy *et al.*, 2005; Iordache et Antsaklis, 2010; Wu *et al.*, 2008). Our approach is a completely forward method (not backward and forward), based on time Petri nets which restricts time intervals. Unlike previously existing general semantic approaches, our algorithm does not need to compute controllable predecessors, then split state classes and handle them separately (which is very costly). Therefore, our approach is less expensive and more efficient in comparison with the other general semantic approaches available in the literature. In the category of state dependent controllers based on the restriction of firing intervals, our algorithm synthesizes maximally permissive controllers. We prove the correctness of this algorithm and discuss its decentralized implementation. In addition, this

algorithm is extended to be used for preemptive scheduling purposes.

Considering the fact that controller synthesis of real time systems are widely used in different areas like web service applications, robotic manipulators, cooperative robotic, networked control systems, work flow applications, manufacturing, production chain, wireless sensing and actuators, a software tool for automatically controller synthesis can facilitate system design and verification. A generic solution is required to be implemented in a software tool then, case based solutions of (Buy *et al.*, 2005; Iordache et Antsaklis, 2010; Wu *et al.*, 2008) are less appropriate. The backward approaches (Cassez *et al.*, 2005; Gardey *et al.*, 2006b) are general but expensive. They require to calculate controllable predecessors and consequently need to compute the difference between states. This in turn may split states that should be handled separately. The approach suggested in this thesis does not split the states and is a good alternative to be implemented in a software tool with a graphical environment for automatically controller synthesis of time Petri net models. Such a tool is suitable for both professional and less professional users as it hides the complicated mathematical computations in the background.

1.4 Thesis organization

This document is organized as follows: in Chapter 2, we introduce some basic concepts and preliminaries. In Chapter 3, we focus on controller synthesis, and available approaches in the literature. We suggest our algorithm for safety properties and prove its correctness. We also discuss how to extend the devised approach to reachability controllers. In Chapter 4, we discuss different methods of state space abstraction and investigate the compatibility of our algorithm with those methods. Chapter 5 is dedicated to distributed systems and decentralized controllers; it includes a survey on the literature and shows that our algorithm is implementable on modular systems. Chapter 6 is devoted to controller synthesis using stopwatch for preemptive scheduling purposes. It includes a literature review on different Petri nets associated with stopwatch. Then, it suggests that the controller adds stopwatch to some appropriate controllable transitions to achieve a controlled system. Finally, we finish this document with the conclusion and future work in Chapter 7.

CHAPTER 2

Preliminaries and Basic Concepts

This chapter explains some preliminary concepts and definitions which will be used later in this document:

- **Transition system**: A *Transition System*(TS) is defined by $(\mathcal{Q}, \rightarrow, q_0)$ tuple where $\mathcal{Q}$ is the set of states, $q_0 \in \mathcal{Q}$ is the initial state and $\rightarrow$ is the transition relation between the states. The notion $q \rightarrow q'$ denotes that q is in relation with q' and $(q, q') \in \rightarrow$. A transition system is called finite (infinite) if $\rightarrow$ is finite (infinite), respectively.

- **Run**: A run is an execution path denoted by ρ. The notion ρ_q stands for an execution path starting from q:

$$\rho_q = q_0 \rightarrow q_1 \rightarrow q_2... \tag{2.1}$$

  where $q = q_0$.
  $q = \rho(i)$ is the i_{th} state of ρ, while ρ^i is a suffix of ρ starting from the state number i. Note that counting the states of an execution path starts from 1 (not 0).

- **Labeled Transition System**: A *labeled transition system* is defined by $Q = (\mathcal{Q},\ q_0,\ \Sigma,\ \rightarrow)$ where Σ is the set of labels and $\rightarrow \subseteq (\mathcal{Q} \times \Sigma \times \mathcal{Q})$. Labeled transition system is either *discrete* or *dense*; dense system is also called *continuous* (e.g. timed systems).

- **Discrete Transition System**: A discrete transition system is a labeled transition system where Σ is the set of actions also denoted by A. In these systems, $q \xrightarrow{\alpha} q'$ denotes that $(q, \alpha, q') \in \rightarrow$. And $q \xrightarrow{\alpha} q'$ with $\alpha \in A$ is called a discrete transition. In a discrete transition system a *run* is defined by:

$$\rho_q = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} ...q_i \xrightarrow{\alpha_i} ... \tag{2.2}$$

  where the action $\alpha_i \in A$ and $q = q_0$. If q is not determined, the corresponding run is denoted by ρ and the initial state of the model is considered as the starting state. Finally, $q = \rho(i)$ refers to the i^{th} state of a run and ρ^i stands for a suffix of ρ starting from the state number i.

– **Timed Transition System**: Let A be the set of actions. A *timed transition system* (TTS) is a labeled transition system, labeled by $\Sigma = A \cup \mathbb{R}^+$, such that $A \cap \mathbb{R}^+ = \emptyset$. A transition $q \xrightarrow{\theta} q'$ where $\theta \in \mathbb{R}^+$ is called a timed transition.
In a timed system, a transition is either a discrete action α, or time elapsing θ denoted by $q \xrightarrow{\theta} q'$. Consequently, in a timed system a run contains both discrete and timed transitions:

$$\rho_q = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\theta_0} ...q_i \xrightarrow{\alpha_i} q_{i+1} \xrightarrow{\theta_i} ... \tag{2.3}$$

A timed transition $q_1 \xrightarrow{\theta} q_2$ means $q_2 = q_1 + \theta$. A timed transition system is an example of dense or continuous transition system.
$Runs(q)$ signifies all runs starting from the state q.

– **Reachability**: For a given timed transition system $(\mathcal{Q}, q_0, \Sigma, \rightarrow)$, state q_n is reachable from q_0 if ρ_q exists, where $q = q_0$ such that, $q_0 \xrightarrow{\rho_q} q_n$. Let $Reach(q_0)$ be the set of all reachable states from q_0 then:

$$q' \in Reach(q_0)\ if\ \exists\ t,\ \exists\ \theta\ q \xrightarrow{\theta\ t} q'. \tag{2.4}$$

– **Formula**: Let Y be the set of real variables. A linear inequality over Y or every logic conjunction of linear inequalities are called *Formula*.

– **Atomic constraint**: An *atomic constraint* on Y is a linear inequality; it can come under two forms: $(x \prec c)$ or $(x - y \prec c)$, called *simple* and *triangular* respectively. In these inequalities $x, y \in Y$, $c \in \mathbb{Q} \cup \{\infty, -\infty\}$ and $\prec \in \{<, \leq, \geq, =, >\}$ where $\mathbb{Q}$ is the set of rational numbers. The set of all atomic constraints on Y is denoted by $\mathcal{C}(Y)$.

– **Hyperplane**: A *hyperplane* H on Y (a set of real variables), is a set of valuations satisfying an atomic constraint.

– **Polyhedron**: A *Polyhedron* P is the set of union or intersection of a finite number of hyperplanes.

– **Region**: A *Region* is represented by a set of constraints of the form $k < x < k + 1$ or $x = k$, where k is a constant on each dimension. Each variable can be equal with an integer or limited by two consecutive integers. For a two dimensional region represented by x and y, $(x = 2, y = 2)$, $(x = 1, 2 < y < 3)$, $(2 < x = y < 3)$ and $(1 < x < 2, 2 < y < 3)$ are all regions whereas, $(1 < x \leq 2, y = 2)$ and $(1 < x < 3, 1 < y < 2)$ are not regions.

– **Zone**: A *zone* is a convex polyhedron. A polyhedron P is called convex if:
$\forall v_1, v_2 \in P$ and $\lambda \in \mathbb{R}$, $0 < \lambda < 1, \lambda v_1 + (1-\lambda)v_2 \in P$.
Informally, each two points of a zone can make a line; if all points of such a line are in the zone, this zone is convex. If a polyhedron is non-convex, it can be broken into finite number of zones.

– **Bound**: Considering $c \in \mathbb{Q}$ and $\prec in\{<, \leq\}$, the couple $(c, \prec)$ is called a bound. Some operations are defined on bounds:
  – $(c, \prec) = (c', \prec') \Longleftrightarrow c = c'$and $\prec = \prec'$.
  – $(c, \prec) < (c', \prec') \Longleftrightarrow (c < c')$ or $(c = c'$ and $\prec < \prec')$.
  – $(c, \prec) \leq (c', \prec') \Longleftrightarrow (c, \prec) = (c', \prec')$ or $(c, \prec) < (c', \prec')$.
  – $(c, \prec) + (c', \prec') = (c + c', min(\prec, \prec'))$.

– **Difference Bound Matrix (DBM)**: Let Y be a set of real variables and F a set of atomic constraints (conjunction of atomic constraints) over Y. Consider $x_i, x_j \in Y$. Add an element 0 (denoted by x_0 or 0) to Y. Every atomic constraint from F on Y, is written as:
$x_i - x_j \prec_{i,j} c_{i,j}$ where $c_{i,j} \in \mathbb{Q} \cup \{\infty, -\infty\}, \prec_{i,j} \in \{<, \leq\}$. Thus, a matrix can represent all atomic constraints of F where each element of the matrix $(b_{i,j})$ is the bound of $x_i - x_j \prec_{i,j} c_{i,j}$. This matrix is called *Difference Bound matrix* of F (Bengtsson, 2002).

– **Reflexive, symmetrical and transitive relation**: A binary relation $\approx$ on the set $\mathcal{Q}$ is a subset of $(\mathcal{Q} \times \mathcal{Q})$. The relation $\approx$ is reflexive if and only if $q \approx q, \forall q \in \mathcal{Q}$. The relation $\approx$ is symmetrical if and only if $q \approx q' \Rightarrow q' \approx q, \forall q, q' \in \mathcal{Q}$. The relation $\approx$ is transitive if and only if

$$\forall q, q', q'' \in \mathcal{Q}, q \approx q' \wedge q' \approx q'' \Rightarrow q \approx q''.$$

– **Simulation**: Let $(\mathcal{Q}, q_0, \Sigma, \rightarrow)$ and $(\mathcal{Q}', q_0', \Sigma, \rightarrow)$ be two labeled transition systems, labeled by the set of labels Σ. Let $\approx \subseteq (\mathcal{Q} \times \mathcal{Q}')$ be a binary relation on the set of states of these two systems. The relation $\approx$ is a simulation if and only if $\forall (q, q') \in (\mathcal{Q} \times \mathcal{Q}')$ such that $q \approx q'$, the following condition holds:

$$q \xrightarrow{a} q_1 \Rightarrow \exists\ q_1'\ s.t.\ q' \xrightarrow{a} q_1' \wedge q_1 \approx q_1'.$$

– **Bi-simulation**: The relation $\approx$ is bi-simulation if and only if $\approx$ and its inverse are both

simulation. Consider two labeled transition systems $(\mathcal{Q}, q_0, \Sigma, \rightarrow)$ and $(\mathcal{Q}', q_0^{'}, \Sigma, \rightarrow)$, labeled by the set of labels Σ. The systems $\mathcal{Q}$ and $\mathcal{Q}'$ are bi-similar if and only if there exists a bi-similar relation $\approx$ such that $(q_0, q_0') \in \approx$.

2.1 Property-specification Languages

In this section, we present a brief survey on the languages declaring the properties and the behavior of the system. In general, temporal logics are used to express the properties and specifications of the system consist of three types of elements:

- a propositional variable from PV, the set of propositional variables.
- boolean logic operators $(\neg, \vee, \wedge)$; $\neg$ stands for not (e.g. $\neg\varphi$).
- some operators (A, E, U, X, R).

Let φ and ψ be two properties, the operators are defined as follows:

- **A**: signifies that the given property holds for all paths and is also denoted by $\forall$ (e.g. $A\varphi$ means that φ is held in all paths).
- **E**: signifies that the given property holds at least in a path and is also denoted by $\exists$ (e.g. $E\varphi$ means that φ is held at least in one path).
- **U**: stands for until (e.g. $\varphi \cup \psi$ denotes that φ is true until ψ happens).
- **X**: stands for next and means that the given property is true at the immediate next state of the computation.
- **R**: stands for release (e.g. $\varphi R \psi$ is read as φ releases ψ, and it denotes that ψ is true forever or it is true until φ happens for the first time).

Not, And, Until and *Next* are the basic operators. Others are defined using these three ones. A and E quantify the path while U and X are operators on the state.

Two major types of formalism exist: *linear time* and *branching time*. In linear time, properties are considered on runs as the set of executions, whereas in branching time properties are observed on the execution trees.

The following languages and their semantics are defined (Penczek et Polrola, 2004):

- **CTL***: CTL* is the most generalized language which contains all the operators named above. Other languages are subclasses of CTL* eliminating some operators. The words respect the grammar below where φ_s and φ_p represent the formula on the state and execution path respectively and $\wp \in PV$ is a propositional variable.

$\varphi_s := \ \wp | \ \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \varphi_s \vee \varphi_s \mid \forall\varphi_p \mid \exists\varphi_p.$

$\varphi_p := \ \varphi_s \mid \varphi_p \wedge \varphi_p \mid \varphi_p \vee \varphi_p \mid X\varphi_p \mid \varphi_p \cup \varphi_p \mid \varphi_p R \varphi_p.$

Two operators F (also denoted by $\diamondsuit$) and G (also denoted by $\square$) are defined as follows:

- $\exists \diamondsuit\varphi = \exists(true \cup \varphi)$.
- $\forall \diamondsuit\varphi = \forall(true \cup \varphi)$.
- $\exists \square\varphi = \neg\forall\diamondsuit\neg\varphi$.
- $\forall \square\varphi = \neg\exists\diamondsuit\neg\varphi$.

The semantics of CTL* is defined on a structure of Kripke $\mathcal{MD} = (Q, \mathcal{V})$, where:

- $Q = (\mathcal{Q}, \rightarrow, q_0)$ is a transition system,
- $\mathcal{V} : \mathcal{Q} \rightarrow 2^{PV}$ is a function that assigns to each state the set of atomic propositions it satisfies.

Let $\mathcal{MD}$ be a model of the system, φ and ψ two CTL* formulas, q a state, $Runs(q)$ the set of all runs starting from q and $\wp$ a propositional variable. In the following, the formal semantics of CTL^* is defined inductively, using the notation of *satisfaction*($\models$) (Hadjidj, 2006). The expression $\mathcal{MD}, x \models \varphi$ is read: in the model $\mathcal{MD}$, x satisfies the property φ.

- $\mathcal{MD}, q \models \wp$ iff $\wp \in \mathcal{V}(q)$.
- $\mathcal{MD}, x \models \neg\varphi$ iff $\mathcal{MD}, x \not\models \varphi$, for $x \in \{q, \rho\}$.
- $\mathcal{MD}, x \models \varphi \vee \psi$ iff $\mathcal{MD}, x \models \varphi$ or $\mathcal{MD}, x \models \psi$, for $x \in \{q, \rho\}$.
- $\mathcal{MD}, x \models \varphi \wedge \psi$ iff $\mathcal{MD}, x \models \varphi$ and $\mathcal{MD}, x \models \psi$, for $x \in \{q, \rho\}$.
- $\mathcal{MD}, q \models \forall\varphi$ iff $\forall\rho \in Runs(q), \mathcal{MD}, \rho \models \varphi$.
- $\mathcal{MD}, q \models \exists\varphi$ iff $\exists\rho \in Runs(q), \mathcal{MD}, \rho \models \varphi$.
- $\mathcal{MD}, \rho \models \varphi$ iff $\mathcal{MD}, \rho(1) \models \varphi$, φ is a state formula.
- $\mathcal{MD}, \rho \models X_\varphi$ iff $\mathcal{MD}, \rho^2 \models \varphi$.
- $\mathcal{MD}, \rho \models \psi U \varphi$ iff $(\exists j \geq 1)(\mathcal{MD}, \rho^j \models \varphi$ and$(\forall 1 \leq i < j), \mathcal{MD}, \rho^i \models \psi)$.

- **LTL**(Linear Temporal Logic): Words can be in the form of $\forall\varphi$ where φ does not include A and E. There is no quantifier.
- **CTL**(Computation Tree Logic): The words are positive subset of CTL* words. There is no $\neg$ operator in CTL. In CTL timed operators are combined with quantifiers e.g.

$AF, EF, AG, EG, AU, EU, AX, EX$ or their boolean combinations.

– $\mathbf{L}_{-X}$:There is no operator X in this language.

When we add the concept of time to the temporal logic, the above-mentioned languages change to TCTL* and TCTL. In such models, formulas are associated with time intervals or clocks.

Two known timed models widely used for modeling various systems are timed automata and timed Petri nets.

2.2 Timed Automata

A timed automaton (Alur, 1999) is an automaton extended with clocks and constraints on clocks. These constraints specify the stay time in each location and the timing condition of transitions.

Let Y be a finite set of real-valued variables called clocks. The notion $\prec$ stands for a binary relation and $\mathcal{C}(Y)$ denotes the set of constraints φ and it follows this grammar:

$$\varphi ::= x \prec k \mid x - y \ \prec k \mid \varphi \wedge \varphi, \text{ where } k \in \mathbb{Z} \text{ and } x, y \in Y.$$

The set $\mathcal{B}(Y)$ is a subset of $\mathcal{C}(Y)$ with only rectangular constraints $(x \prec k)$.

Then, timed automata are defined formally by a tuple (L, l_0, Y, A, E, I), where L is the set of locations, $l_0 \in L$ is the initial location, Y is the set of clocks, A is the set of actions, $E \subseteq L \times A \times \mathcal{B}(Y) \times 2^Y \times L$ is the set of edges between locations with an action, a guard (a condition serving as a label) and a set of clocks to be reset, and $I : L \to \mathcal{B}(Y)$ assigns invariants to locations.

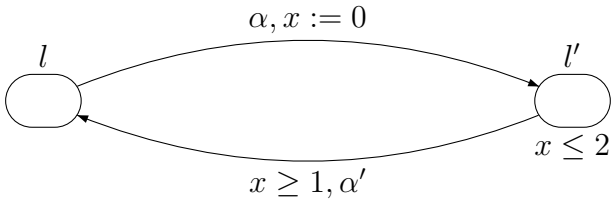


Figure 2.1 A simple example of timed automata from (Alur, 1999).

A clock valuation is a function $v : Y \to \mathbb{R}^+$ from the set of clocks to the non-negative reals. $\mathbb{R}^Y$ is the set of all clock valuations and $\forall x \in Y,\ v_0(x) = 0$. Let $x \in Y$ and $\theta \in \mathbb{R}^+$, $(v+\theta)(x) = v(x)+\theta$. We consider invariants and constraints as sets of clock valuations. For $g \in \mathcal{C}(Y)$ and $v \in (\mathbb{R}^+)^Y$, if v satisfies g we denote $v \models g$.

In timed automata, we define state $q = (l, v)$. Fig.2.1 taken from (Alur, 1999) shows a simple example of timed automata with the initial location l and the clock x. The system may remain in l infinitely as there is no invariant to restrict that. Once the action α happens and the system goes to l', x is reset. It may remain in l' until 2 time units and action α' may happen only after at least one time unit.

The semantics of timed automata is defined by a timed transition system $(\mathcal{Q}, q_0, \Sigma, \to)$ where $\mathcal{Q}$ is the set of locations $\mathcal{Q} = L \times (\mathbb{R}^+)^Y$ and $q_0 = (l_0, v_0)$:

- $(l, v) \xrightarrow{\theta} (l, v+\theta)$ if $\forall \theta' : 0 \leq \theta' \leq \theta \Rightarrow v + \theta' \in I(l)$.
- $(l, v) \xrightarrow{\alpha} (l', v')$ if there exists a transition $l \xrightarrow{g,\alpha,Y} l' \in E$ *s.t.* $v \models g, v' = v[Y]$ and, $v' \models I(l')$.

Kronos (Bozga *et al.*, 1998) and UPPAAL (Behrmann *et al.*, 2006b) are two known tools for formal verification of timed automata. UPPAAL-TIGA (Behrmann *et al.*, 2007) is an extension to UPPAAL that can be used for controller synthesis in timed automata.

2.3 Time Petri Nets

2.3.1 Definition and behavior

A time Petri net (TPN in short) is a Petri net augmented with time intervals associated with transitions (Merlin, 1974).

Formally, a time Petri net (TPN) is a tuple $(P, T, Pre, Post, M_0, Is)$ where:

- P and T are finite sets of places and transitions such that $(P \cap T = \emptyset)$,
- Pre and $Post$ are the backward and the forward incidence functions ($Pre, Post : P \times T \to \mathbb{N}, \mathbb{N}$ is the set of nonnegative integers),
- M_0 is the initial marking ($M_0 : P \to \mathbb{N}$), and
- Is is the static interval function ($Is : T \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$). $\mathbb{Q}^+$ is the set of nonnegative rational numbers. Is associates with each transition t an interval called the static firing interval of t. Bounds $\downarrow Is(t)$ and $\uparrow Is(t)$ of the interval $Is(t)$ are the minimum and maximum firing delays of t respectively.

In a controllable time Petri net, transitions are partitioned into controllable and uncontrollable transitions, denoted by T_c and T_u(with $T_c \cap T_u = \emptyset$ and $T = T_c \cup T_u$), respectively. For the sake of simplicity and clarification, in this manuscript the controllable transitions are depicted as white bars, while the uncontrollable ones as black bars.

In TPN, marking is a function of $P \rightarrow \mathbb{N}$. Let M be a marking and t be a transition. Transition t is enabled for M if and only if all the required tokens for firing t are present in M, i.e., $\forall p \in P, M(p) \geq Pre(p,t)$. In this case, the firing of t leads to the marking M' defined by: $\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t)$. We denote by $En(M)$ the set of transitions enabled for M, i.e., $En(M) = \{t \in T \mid \forall p \in P, Pre(p,t) \leq M(p)\}$. A TPN, is called bounded if:

$$\forall p \in P, M(p) \leq b.$$

For $t \in En(M)$, we denote $CF(M,t)$ to be the set of transitions enabled in M but in conflict with t, i.e., $CF(M,t) = \{t' \in En(M) \mid t' = t \vee \exists p \in P, M(p) < Pre(p,t') + Pre(p,t)\}$.

Let $t \in En(M)$ and M' be the successor marking of M by t, a transition t' is said to be newly enabled in M' iff t' is not enabled in the intermediate marking (i.e., $M - Pre(.,t)$) or $t' = t$. We denote $New(M',t)$ the set of transitions newly enabled M', by firing t from M, i.e., $New(M',t) = \{t' \in En(M') | t = t' \vee \exists p \in P, M'(p) - Post(p,t) < Pre(p,t')\}$.

Among the different semantics proposed for time Petri nets (Roux *et al.*, 2005), the research presented in this thesis focuses on the classical one, called intermediate semantics in (Roux *et al.*, 2005), in the context of mono-server and strong-semantics (Boyer et Vernadat, 2000).

There are two known characterizations for the TPN state. The first one, based on clocks, associates with each transition t_i of the model a *clock* to measure the time elapsed since t_i became enabled most recently. The TPN clock state is the pair (M, ν), where M is the marking and ν is the clock valuation function, $\nu : En(M) \rightarrow \mathbb{R}^+$. For a clock state (M, ν) and $t_i \in En(M)$, $\nu(t_i)$ is the value of the clock associated with transition t_i. The initial clock state is $q_0 = (M_0, \nu_0)$ where $\nu_0(t_i) = 0$, for all $t_i \in En(M_0)$. The TPN clock state evolves either by time progression or by firing transitions. When a transition t_i becomes enabled, its clock is initialized to zero. The value of this clock increases synchronously with time until t_i is fired or disabled by the firing of another transition. t_i can fire, if the value of its clock is inside its static firing interval $Is(t_i)$. It must be fired immediately, without any additional

delay, when the clock reaches $\uparrow Is(t_i)$. The firing of a transition takes no time, but may lead to another marking (required tokens disappear while produced ones appear).

Let $q = (M, \nu)$ and $q_0 = (M_0, \nu_0)$ be two clock states of the TPN model, $\theta \in \mathbb{R}^+$ and $t_f \in T$. We write $q \xrightarrow{\theta} q'$, also denoted by $q + \theta$, if and only if state q' is reachable from the state q after a time progression of θ time units, i.e.:
$\bigwedge_{t' \in En(M)} \nu(t) + \theta \leq \uparrow Id(t_i), M' = M$, and $\forall t_j \in En(M'), \nu'(t_j) = \nu(t_j) + \theta$.
We write $q \xrightarrow{t_f} q'$ if and only if state q' is immediately reachable from state q by firing transition t_f, i.e.: $t_f \in En(M)$, $\nu(t_f) \geq \downarrow Is(t_f)$, $\forall p \in P, M'(p) = M(p) - Pre(p, t_f) + Post(p, t_f)$, and $\forall t_i \in En(M')$, $\nu'(t_i) = 0$, if $t_i \in New(M', t_f)$, $\nu'(t_i) = \nu(t_i)$ otherwise.

The second characterization, based on intervals, defines the TPN state as a marking and a function which associates with each enabled transition the time interval in which the transition can fire (Berthomieu et Vernadat, 2003).

The TPN state is defined as a pair (M, Id), where M is a marking and Id is a firing interval function ($Id : En(M) \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$). The initial state is (M_0, Id_0) where M_0 is the initial marking and $Id_0(t) = Is(t)$, for $t \in En(M_0)$.

Let (M, Id) and (M', Id') be two states of the TPN model, $\theta \in \mathbb{R}^+$ and $t \in T$. The transition relation $\longrightarrow$ over states is defined as follows:

- $(M, Id) \xrightarrow{\theta} (M', Id')$, also denoted by $(M, Id) + \theta$, if and only if from state (M, Id), we will reach the state (M', Id') by a time progression of θ units, i.e., $\bigwedge_{t' \in En(M)} \theta \leq \uparrow Id(t'), M' = M$, and $\forall t'' \in En(M'), Id'(t'') = [Max(\downarrow Id(t'') - \theta, 0), \uparrow Id(t'') - \theta]$.
- $(M, Id) \xrightarrow{t} (M', Id')$ if and only if the state (M', Id') is reachable from state (M, Id) by immediately firing transition t , i.e., $t \in En(M), \downarrow Id(t) = 0$, $\forall p \in P, M'(p) = M(p) - Pre(p, t) + Post(p, t)$, and $\forall t' \in En(M'), Id'(t') = Is(t')$, if $t' \in New(M', t), Id'(t') = Id(t')$, otherwise.

The TPN state space is the labeled transition system $(\mathcal{Q}, q_0, \Sigma, \longrightarrow)$, where $q_0 = (M_0, Id_0)$ is the initial state of the TPN, $\Sigma = T \cup R^+$ and $\mathcal{Q} = \{q | q_0 \xrightarrow{*} q\}$ ($\xrightarrow{*}$ being the reflexive and transitive closure of the relation $\longrightarrow$ defined above) is the set of reachable states of the model. A *run* in the TPN state space $(\mathcal{Q}, q_0, \Sigma, \longrightarrow)$ of a state $q \in \mathcal{Q}$ is a maximal sequence $\rho = q_1 \xrightarrow{\theta_1} q_1 + \theta_1 \xrightarrow{t_1} q_2 \xrightarrow{\theta_2} q_2 + \theta_2 \xrightarrow{t_2} q_3 ...$, such that $q_1 = q$. By convention, for any state q_i, the relation $q_i \xrightarrow{0} q_i$ holds. The sequence $\theta_1 t_1 \theta_2 t_2 ...$ is called the timed trace of ρ. The

sequence $t_1 t_2 ...$ is called the firing sequence (untimed trace) of ρ. A marking M is reachable if and only if $\exists q \in \mathcal{Q}$ s.t. its marking is M. Runs (resp. timed / untimed traces) of the TPN are all runs (resp. timed / untimed traces) of the initial state q_0.

To use enumerative analysis techniques with time Petri nets, an extra effort is required to abstract their generally infinite state spaces. Abstraction techniques aim to construct a finite contraction of the state space of the model by removing some irrelevant details. This contraction of the state space preserves the properties of interest. For best performances, the contraction should be the smallest possible and computed with the minimal resources in terms of time and space. The preserved properties are usually verified using standard analysis techniques on the abstractions (Penczek et Polrola, 2004).

Several state space abstraction methods have been proposed, in the literature, for time Petri nets (e.g. the *state class graph (SCG)* (Berthomieu et Diaz, 1991), the *zone based graph (ZBG)* (Boucheneb *et al.*, 2009), etc). These abstractions may differ mainly in the characterization of the states (interval states or clock states), the agglomeration criteria of the states, the representation of the agglomerated states (abstract states), the kind of properties they preserve (markings, linear or branching properties) and their size.

Abstraction consists of agglomeration of some states with similar behaviors. Agglomeration methods are different in the type of properties they preserve (LTL,CTL,...), the time characterization they use (clock, interval or firing date), and also the agglomeration policy they consider. There are three levels for abstraction criteria:

- At the first level, states being the result of time elapsing are agglomerated and only states being the result of transitions are considered as distinct states.
- At the second level, all states being the result of a similar transition are regrouped.
- At the third level, all states being the result of the same transitions with different sequences are regrouped as equivalent nodes. For example, a state being the result of t_1, t_2 is agglomerated with a state being the result of t_2, t_1.

Fig.2.2 from (Boucheneb et Hadjidj, 2008) shows the three levels of abstraction.

Expressing the state of a model by either intervals or clocks are completely bi-similar. When time elapses in a model, its state is modified. The new state is declared by increasing clock values in clock characterization or decreasing the corresponding bounds in interval characterization. In some cases, using intervals instead of clocks helps to better abstract the

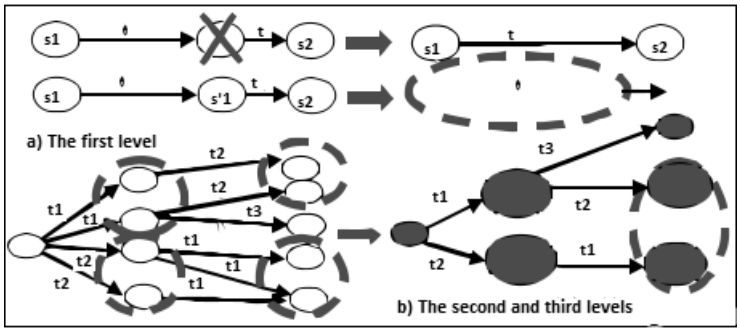


Figure 2.2 Three levels of abstraction (Boucheneb et Hadjidj, 2008).

state space, because bounds are positive values and whenever decreasing an interval yields to a negative value, it will be considered zero. In some other cases like construction of the state space graph, using clocks helps to better distinguish the states.

These abstractions are finite for all bounded time Petri nets. However, abstractions based on clocks are less interesting than the interval based abstractions when only linear properties are of interest. Indeed, abstractions based on intervals are finite for bounded TPN with unbounded intervals, while this is not true for abstraction based on clocks. The finiteness is enforced using an approximation operation, which may involve some overhead computation.

2.3.2 Zone Based Graph

In the Zone Based Graph (ZBG) (Boucheneb *et al.*, 2009), all clock states reachable by runs supporting the same firing sequence are agglomerated in the same node and considered modulo some over-approximation operation (Behrmann *et al.*, 2006a; Gardey *et al.*, 2006a). This operation is used to ensure the finiteness of the ZBG for the bounded TPNs with unbounded firing intervals. An abstract state, called state zone, is defined as a pair $\beta = (M, FZ)$ combining a marking M and a formula FZ which characterizes the clock domains of all the states agglomerated in the state zone. In FZ, the clock of each enabled transition for M is represented by a variable with the same name. The domain of FZ is convex and has a unique canonical form represented by the pair (M, Z), where Z is a DBM of the order $|En(M) \cup \{o\}$ defined by: $\forall (x, y) \in (En(M) \cup \{o\})^2$, $z_{xy} = Sup_{FZ}(x - y)$. Here o represents the value of 0. State zones of the ZBG are in the relaxed form.

The initial state zone is the pair $\beta_0 = (M_0, FZ_0)$, where M_0 is the initial marking and $FZ_0 = \bigwedge_{t_i,t_j \in En(M_0)} 0 \leq ti = tj \leq \uparrow_{t_u \in En(M_0)} I_s(t_u)$. As an example, consider the TPN given in (Gardey *et al.*, 2006b) and reported at Fig.2.4. Its state zone graph is reported at Fig.2.5 and its state zone graphs are reported in Table 2.1. More information about computing ZBG could be find in (Boucheneb *et al.*, 2009).

The ZBG of a bounded TPN is not necessarily bounded. Fig. 2.3 given in (Boucheneb *et al.*, 2009) shows a bounded TPN with an infinite ZBG. For this reason, zone based graphs require some over-approximation.

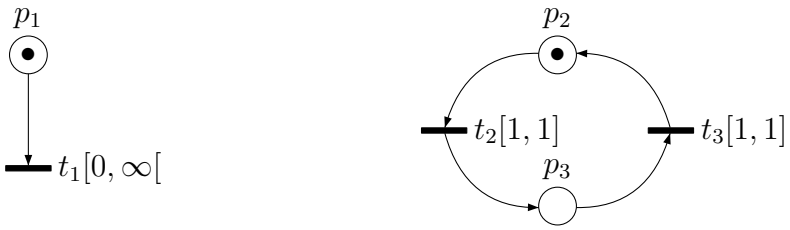


Figure 2.3 A bounded TPN with an infinite ZBG reported from (Boucheneb *et al.*, 2009).

In this work, we consider the state class method and study the possibility of enforcing the behavior of a given TPN in order to satisfy a safety / reachability property. The idea is to construct on-the-fly the reachable state classes of the TPN and at the same time collect progressively firing subintervals to be avoided so that the properties of interest are satisfied.

2.3.3 The state class graph method

In the state class graph method (Berthomieu et Diaz, 1991), all states reachable by the same firing sequence from the initial state are agglomerated in the same node and considered modulo the relation of the equivalence defined by: two sets of states are equivalent if and only if they have the same marking and the same firing domain. The firing domain of a set of states is the union of the firing domains of its states. All equivalent sets are agglomerated in the same node called a *state class* defined as a pair $\alpha = (M, F)$, where M is a marking and F is a formula which characterizes the firing domain of α. For each transition t_i enabled in M, there is a variable $\underline{t}_i$ in F representing its firing delay. F can be rewritten as the set of atomic constraints of the form [1]: $\underline{t}_i - \underline{t}_j \leq c$, $\underline{t}_i \leq c$ or $-\underline{t}_j \leq c$, where t_i, t_j are transitions, $c \in \mathbb{Q} \cup \{\infty\}$ and $\mathbb{Q}$ is the set of rational numbers.

1. For economy of notation, we use operator $\leq$ even if $c = \infty$.

Though the same domain can be expressed by different conjunctions of the atomic constraints (i.e., different formulas), all equivalent formulas have a unique form, called canonical form, that is usually encoded by a difference bound matrix (DBM) (Bengtsson, 2002). The canonical form of F is encoded by the DBM D (a square matrix) of the order $|En(M)|+1$ defined by: $\forall t_i, t_j \in En(M) \cup \{t_0\}, d_{ij} = (\leq, Sup_F(\underline{t}_i - \underline{t}_j))$. Here t_0 ($t_0 \notin T$) represents a fictitious transition whose delay is always equal to 0 and $Sup_F(\underline{t}_i - \underline{t}_j)$ is the largest value of $\underline{t}_i - \underline{t}_j$ in the domain of F. The computation of canonical form is based on the *Floyd-Warshall* shortest path algorithm and is considered as the most costly operation (cubic in the number of variables in F). The canonical form of a DBM makes some operations over formulas like the test of equivalence easier. Two formulas are equivalent if and only if the canonical forms of their DBMs are identical.

The initial state class is $\alpha_0 = (M_0, F_0)$, where $F_0 = \bigwedge_{t_i \in En(M_0)} \downarrow Is(t_i) \leq \underline{t}_i \leq \uparrow Is(t_i)$. Let $\alpha = (M, F)$ be a state class and t_f a transition and $succ(\alpha, t_f)$ the set of states defined by: $succ(\alpha, t_f) = \{q' \in \mathcal{Q} \mid \exists q \in \alpha, \exists \theta \in \mathbb{R}^+ \text{ s.t. } q \xrightarrow{\theta} q+\theta \xrightarrow{t_f} q'\}$. The state class α has a successor by t_f (i.e. $succ(\alpha, t_f) \neq \emptyset$), if and only if t_f is enabled in M and can be fired before any other enabled transition, i.e., the following formula is consistent [2]: $F \wedge (\bigwedge_{t_i \in En(M)} \underline{t}_f \leq \underline{t}_i)$. In this case, the firing of t_f leads to the state class $\alpha' = (M', F') = succ(\alpha, t_f)$ computed as follows (Berthomieu et Diaz, 1991):

1. $\forall p \in P, M'(p) = M(p) - Pre(p, t_f) + Post(p, t_f)$.
2. $F' = F \wedge (\bigwedge_{t_i \in En(M)} \underline{t}_f - \underline{t}_i \leq 0)$.
3. Replace in F' each $\underline{t}_i \neq \underline{t}_f$, by $(\underline{t}_i + \underline{t}_f)$.
4. Eliminate by substitution $\underline{t}_f$ and each $\underline{t}_i$ of transition conflicting with t_f in M.
5. Add constraint $\downarrow Is(t_n) \leq \underline{t}_n \leq \uparrow Is(t_n)$, for each transition $t_n \in New(M', t_f)$.

Formally, the SCG of a TPN model is a structure $(\mathcal{CC}, \longrightarrow, \alpha_0)$, where $\alpha_0 = (M_0, F_0)$ is the initial state class, $\forall t_i \in T, \alpha \xrightarrow{t_i} \alpha'$ iff $\alpha' = succ(\alpha, t_i) \neq \emptyset$ and $\mathcal{CC} = \{\alpha | \alpha_0 \xrightarrow{*} \alpha\}$. The complexity of computing each state class is $O(n^2)$, n being the number of transitions in the model (Boucheneb et Mullins, 2003). Reachability and boundedness problems are known to be undecidable for time Petri nets (Berthomieu et Menasche, 1983). However, there are some useful sufficient conditions to ensure the boundedness (Berthomieu et Menasche, 1983). The SCG is finite for all bounded TPNs and preserves linear properties (Berthomieu et Vernadat,

2. A formula F is consistent iff there is at least, one tuple of values that satisfies all constraints of F at once.

2003).

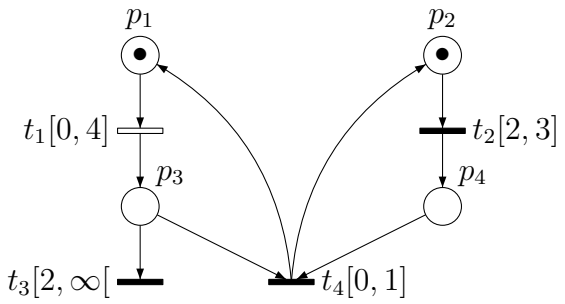


Figure 2.4 A simple Petri net with $T_c = \{t_1\}$.

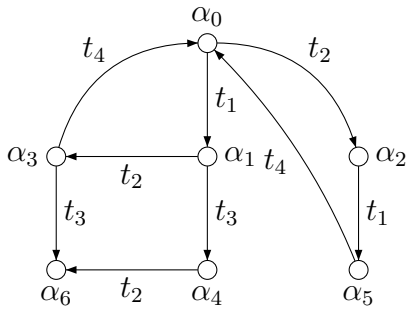


Figure 2.5 The state graph of the TPN presented at Fig.2.4.

As an example, Fig.2.4 shows a simple example of TPN with only one controllable transition t_1. At the beginning two places p_1 and p_2 are marked. In the case that t_3 fires, one token is lost and the model is blocked. In order to have a live and safe model, t_3 should be prevented. Fig.2.5 shows the state class graph of the TPN presented at Fig.2.4. Its state classes are reported in Table 2.2. For this example, state class graph and state zone based graph of the system are identical while classes and zones are different.

Table 2.1 State zones of the TPN presented at Fig.2.5.

$\beta_0 : p_1 + p_2$	$0 \leq \underline{t}_1 = \underline{t}_2 \leq 3$
$\beta_1 : p_2 + p_3$	$0 \leq \underline{t}_2 \leq 3 \wedge 0 \leq \underline{t}_3 \leq 3 \wedge 0 \leq \underline{t}_2 - \underline{t}_3 \leq 3$
$\beta_2 : p_1 + p_4$	$2 \leq \underline{t}_1 \leq 4$
$\beta_3 : p_3 + p_4$	$0 \leq \underline{t}_3 \leq 3 \wedge 0 \leq \underline{t}_4 \leq 1 \wedge 0 \leq \underline{t}_3 - \underline{t}_4 \leq 3$
$\beta_4 : p_2$	$2 \leq \underline{t}_2 \leq 3$
$\beta_5 : p_3 + p_4$	$0 \leq \underline{t}_3 = \underline{t}_4 \leq 2$
$\beta_6 : p_4$	

Table 2.2 The state classes of the TPN presented at Fig.2.5.

$\alpha_0 : p_1 + p_2$	$0 \leq \underline{t}_1 \leq 4 \wedge 2 \leq \underline{t}_2 \leq 3$
$\alpha_1 : p_2 + p_3$	$0 \leq \underline{t}_2 \leq 3 \wedge 2 \leq \underline{t}_3$
$\alpha_2 : p_1 + p_4$	$0 \leq \underline{t}_1 \leq 2$
$\alpha_3 : p_3 + p_4$	$0 \leq \underline{t}_3 \wedge 0 \leq \underline{t}_4 \leq 1$
$\alpha_4 : p_2$	$0 \leq \underline{t}_2 \leq 1$
$\alpha_5 : p_3 + p_4$	$2 \leq \underline{t}_3 < \infty \wedge 0 \leq \underline{t}_4 \leq 1$
$\alpha_6 : p_4$	

CHAPTER 3

Controller Synthesis in Real Time Systems

3.1 Introduction to controller synthesis

In controller synthesis, the objective is to find a controller running in parallel with the system under study to guarantee the satisfaction of some given properties, as discussed earlier in Chapter 1. A controller makes an open-loop system to be closed-loop. In such a framework, in general, the system consists of controllable and non controllable actions. The controller can only act on controllable actions. The objective is to force to meet some properties by selecting or delaying some controllable actions.

We mentioned earlier that two main questions arise for the controller: the existence and the possibility of implementation. The first question, *Control Problem* says, given a system $\mathcal{S}$ and a property φ, whether a controller C exists for the system $\mathcal{S}$ such that C running in parallel with $\mathcal{S}$ satisfies the property φ. This concept is formally represented as $\mathcal{S}||C \models \varphi$ where $||$ signifies parallel execution and $\models$ stands for satisfaction (Fig.3.1). The second question is the *Controller Synthesis Problem*, which in case the mentioned controller exists, investigates how to implement it. Another important aspect for the controller synthesis problem is the controller permissiveness. A maximally permissive controller is a controller that limits as little as possible the behavior of the system. The challenge is to find a controller ensuring a good compromise between the permissiveness and the implementation complexity.

There are different types of controllers: path dependent controllers, state dependent controllers, etc. In a path dependent controller, the controller keeps track of all the states traveled in each path; whereas, a state dependent controller acts only upon the current state of the system. In other terms, in contrary with a path dependent controller, a state dependent controller is memoryless. Implementation of path dependent controllers are more costly. The state dependent controllers are more interesting and are established as a good compromise between the permissiveness and the implementation complexity.

Among various models used to describe the behavior of $\mathcal{S}$, timed automata (TA in short) and time Petri nets (TPN in short) are the well-knowns. The properties studied in both models for control purposes are classified into two main categories:

1. Safety properties: Forbidden (bad) states will never be reached by the system.
2. Reachability properties: The system will eventually reach a state within a set of goal states.

In order to control these kinds of properties in timed models (TA and TPN), several approaches of state dependent controller synthesis have been proposed in the literature (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998). Two known methods are the backward *fixpoint* method and the backward-forward *on-the-fly* method. Both methods are based on computing controllable predecessors of abstract states (state zones). This computation involves some expensive operations such as computing differences between abstract states resulting split abstract states that are handled separately.

In this research, we propose an algorithm to synthesize on-the-fly a safety / reachability controller for time Petri nets. Unlike other mentioned solutions, our algorithm does not need to compute controllable predecessors and is based on the state class graph method. During the exploration of the state class graph, all sequences leading to undesired states are extracted and firing subintervals to be avoided are determined. The control is state dependent and consists of restricting the firing intervals of controllable transitions so as to avoid reaching the bad state classes. We show that for this category of control, our algorithm synthesizes maximally permissive controllers (Heidari et Boucheneb, 2012b,c; Heidari *et al.*, 2011).

This chapter is organized as follows: In Section 3.2, after a short survey on the control theory, previous algorithms and related work are discussed. Section 3.3 introduces a forward method for computing predecessors of the state classes. The algorithm proposed here for safety properties is developed in Section 3.4. The algorithm for reachability controller synthesis is developed in Section 3.5. Finally, Section 3.6 is devoted to the conclusion of the chapter.

Figure 3.1 Controller of a system.

3.2 Literature review

The theory of control was initially introduced in (Ramadge et Wonham, 1987). In this article,the authors have formalized, in terms of formal languages, the notion of control and the existence of a controller that enforces a discrete event system (DES) to behave as expected (legal languages). The system to be controlled is described in terms of a formal language which is generated by a finite automaton whose alphabet is a finite set of events. The control consists of enabling or disabling some controllable events so that to force or avoid some specific sequences of events of the system.

The concept of control has been afterwards extended to various models such as timed automata (Wong-Toi et Hoffmann, 1991) and time Petri nets (Sathaye et Krogh, 1993), where the control specification is expressed on the model states rather than the model language. Thus, for every system modeled by a controllable language, timed automata or time Petri nets, controller synthesis is used to restrict the behavior of the system making it to satisfy the desired properties (safety or reachability properties).

The typical procedure can be described as follows: First, a system is modeled and its desired properties are defined. Then, the existence (control problem as stated in Altisen *et al.*, 2005) and the implementation of the appropriate controller (controller synthesis problem as stated in Altisen *et al.*, 2005) are investigated. Several approaches of controller synthesis have been proposed in the literature. They may differ in the model they are working on (various types of Petri nets or automata), the approach they are based on (analytical (Wu *et al.*, 2008), structural (Buy *et al.*, 2005; Iordache et Antsaklis, 2010), semantic (Abid et Zouari, 2010b; Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998)), and finally the property to be controlled.

In (Wu *et al.*, 2008), the authors have considered a particular type of capacity time Petri net, where timing constraints are associated with transitions and some places, and all transitions are controllable. This timed Petri net is used to model a cluster tool with wafer residency time constraints. The wafers and their time constraints are represented by timed places. Using analytical approaches of schedulability and the particular structure of their model (model of the cluster tool), the authors have established an algorithm for finding, if it exists, an optimal periodic schedule which respects the residency time constraints of wafers. The control consists of limiting timing constraints of transitions and some places so as to respect the residency time constraints of the wafers.

In (Buy et Darabi, 2003; Buy *et al.*, 2005), the authors have considered safe and live time Petri nets where deadlines can be associated with some transition firings. The control consists of enforcing the model to meet deadlines of transition firings. The controller has the possibility of disabling any transition t which prevents the deadline of a transition t_d to be met. A transition t is allowed to fire only if its latency (the maximum delay between firing t and the next firing of t_d) is not greater than the current deadline of t_d. The latencies of transitions are computed by constructing an unfolding Petri net of the underlying untimed Petri net. This approach does not need to explore the state space. However, in general, the resulting controller is not maximally permissive, meaning that the controller may disable a net behavior that does not violate the properties of interest.

In (Iordache et Antsaklis, 2010), the authors have considered a mono-processor system. They have proposed an application of controller synthesis to automatically generate the part of the programs that deals with the coordination of concurrent processes. The programs are modeled by means of Petri nets in which places are labeled with instructions and transitions with conditions. The requirements are specified by means of constraints over markings. The control consists of enforcing the behavior so as to meet the requirements. In this application, the authors have used Petri nets without dealing with time constraints and the supervisor can disable a controllable transition.

Other controller synthesis approaches are based on the theory of regions and Petri nets (Abid et Zouari, 2010b; Ghaffari *et al.*, 2003). Basically, the theory of regions (Badouel *et al.*, 1995) is used to verify if an automaton (corresponding, for instance, to the expected behavior) is isomorphic to the reachability graph of a Petri net. In (Ghaffari *et al.*, 2003), the authors have adapted this theory to controller synthesis in Petri nets. The idea is to compute a convenient set of places and edges and add them to the original plant model. The completed model will avoid all forbidden markings. Their method first generates the reachability graph of the net. Then, the desired behavior of the net is derived from its reachability graph by eliminating forbidden markings, *dangerous markings* (i.e., the markings leading to a forbidden marking through an uncontrollable event) and blocking markings (i.e., the markings not leading to final states). Finally, the obtained graph is synthesized to calculate the control places and their parameters to be added to the plant.

In (Abid et Zouari, 2010b), the authors have discussed controller synthesis in a special type of colored Petri nets called well-formed nets (WF-nets) (Jensen et Rozenberg, 1991).

WF-nets are equivalent to the colored Petri nets in terms of expressiveness but, are enforcing some structuring restriction on color classes and firing functions. The authors have applied the theory of regions on an abstract version of the reachability graph of the plant called symbolic reachability graph (SRG). Timing characteristics are not taken into account.

In (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998), the authors have considered timed models (TA or TPN) with two kinds of transitions (controllable and uncontrollable) and investigated the control problem for safety or reachability properties. In their model, in order to prevent some undesired states, the controller can act on any firable and controllable transition. The controller can delay or force firing of controllable transitions but it cannot disable these transitions. The control problem is addressed by computing the winning states of the model, i.e. the states which will not lead by an uncontrollable transition to an undesired state. The computation of the winning states is based on the concept of controllable predecessors of states. In the literature, the set of controllable predecessors is usually denoted by $\pi(X)$, where X is the set of states satisfying the desired property (safe/goal states). The set $\pi(X)$ is defined by (Gardey *et al.*, 2006b):

$$\begin{array}{c} \pi(X) = \{q \in \mathcal{Q} | (\exists\ \delta \in \mathbb{R}_{\geq 0},\ q' \in X \text{ s.t. } (\exists\ t \in T\ q \xrightarrow{\delta\ t} q') \vee (q \xrightarrow{\delta} q')) \wedge \\ \forall\ \delta \in \mathbb{R}_{\geq 0},\ (\exists\ t \in T_u, q' \notin X\ q \xrightarrow{\delta\ t} q')\ \Rightarrow\ (\exists\ \delta_c < \delta, t_c \in T_c,\ q_c \in X\ q \xrightarrow{\delta_c\ t_c} q_c)\}. \end{array} \tag{3.1}$$

Intuitively, $\pi(X)$ is the set of predecessors of X which will not bring the system out of X. Fig.3.2 clarifies this concept. If the environment can execute an uncontrollable transition after δ time units, leading the system out of X (denoted by $\bar{X}$), then the controller should be able to execute a controllable action to keep the system in X before δ time units. In addition, in the context of timed models with strong semantics, the controller should not be forced to execute a controllable transition leading the system out of X. In strong semantics, a transition must be fired, without any additional delay, when the upper bound of its firing interval is reached.

Let $AG\ \phi$ be a safety property and let $X_0 = Sat(\phi)$ be the set of states which satisfy the property ϕ (safe states). The fixpoint of $X_{i+1} = h(X_i) = X_i \cap \pi(X_i), i \geq 0$ gives the largest set of safe states whose behaviors can be controlled so as to maintain the system inside this set of states (i.e., winning states). If the largest fixpoint of h includes the initial state then, it gives a controller which forces the system to stay in the safe states (i.e., a winning strategy).

Similarly, the fixpoint method is also used for reachability properties. Let $AF\ \psi$ be

a reachability property and $X_0 = Sat(\psi)$ the set of goal states. The least fixpoint of $X_{i+1} = h(X_i) = X_i \cup \pi(X_i), i \geq 0$ is the set of states whose behaviors can be controlled so as to reach one of the goal states (i.e., winning states) (Cassez *et al.*, 2005; Tripakis, 1998).

In the context of a timed model, this technique is applied on a state space abstraction of the timed model. In this case, X_i is a set of abstract states. If X_i is a finite set of abstract states, then the set of controllable predecessors of X_i is also a finite set of abstract states. The computation of the fixpoint of h will converge after a finite number of steps if the state space abstraction is finite (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998).

Note that the state space abstractions used in (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Tripakis, 1998) are based on clocks but the state space abstraction used in (Gardey *et al.*, 2006b) is not necessarily complete. A state space abstraction of a given model is sound if and only if it captures all firing sequences of the model. It is complete if and only if each firing sequence in the state space abstraction reflects a firing sequence of the model. The fixpoint method cannot guarantee to give the safety controller when it exists, unless the state space abstraction is both sound and complete. Indeed, a synthesis may fail because of some unreachable states, while for the reachable state space the safety controller exists. However, the cost of processing is increased because a sound and complete state space abstraction should be entirely calculated before applying the fixpoint algorithm.

Let us explain how to compute the fixpoint of h for a safety property through an example. Consider the TPN given in (Gardey *et al.*, 2006b) and reported in Fig.2.4. The state class graph (SCG) and the zone based graph (ZBG) of this TPN are equal, except that nodes are defined differently (state classes or state zones). The state class graph is depicted in Fig.2.5. Its state classes and state zones are reported in Table 2.1 and Table 2.2, respectively.

Suppose we are interested in forcing the following safety property: *AG not* $p_1 + p_3 = 0$,

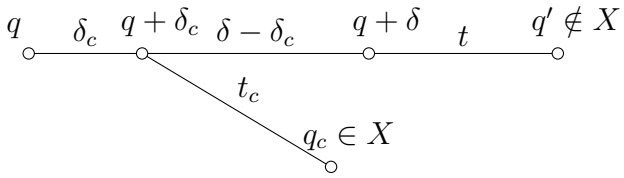


Figure 3.2 Controllable predecessors.

which means that places p_1 and p_3 are never both empty. The transition t_1 is the only controllable transition and the forbidden markings are p_2 and p_4. As the state class graph shows, if t_2 happens before t_1, the right path happens and does not lead to any bad markings. So, in this case, the controller has nothing to do. On the other hand, if t_1 happens before t_2, two state classes having forbidden markings may be reached (α_4, α_6).

To verify whether or not there is a controller for such a property, we compute the fixpoint of $X_{i+1} = h(X_i) = X_i \cap \pi(X_i)$, where $X_0 = \{\beta_0, \beta_1, \beta_2, \beta_3, \beta_5\}$ is the set of state zones which satisfy the property *not* $p_1 + p_3 = 0$. Such a controller exists if and only if the initial state of the model is a winning state (i.e., belongs to the fixpoint of h). The fixpoint is computed, in 3 iterations, as follows:

1. Iteration 1: $X_1 = X_0 \cap \pi(X_0) = \{\beta_0, \beta'_1, \beta_2, \beta'_3, \beta_5\}$. In this iteration, all states of β_1 and β_3, which are uncontrollable predecessors of bad state classes β_4 and β_6 are eliminated:
   $\beta'_1 = (p_2 + p_3, 1 < \underline{t}_2 \leq 3 \wedge 0 \leq \underline{t}_3 < 2 \wedge 1 < \underline{t}_2 - \underline{t}_3 \leq 3)$ and
   $\beta'_3 = (p_3 + p_4, 1 \leq \underline{t}_3 \leq 3 \wedge 0 < t_4 \leq 1 \wedge 1 \leq \underline{t}_3 - \underline{t}_4 \leq 3)$.
2. Iteration 2: $X_2 = X_1 \cap \pi(X_1) = \{\beta_0, \beta''_1, \beta_2, \beta'_3, \beta_5\}$. This iteration eliminates from β'_1 all states, which are uncontrollable predecessors of bad states of $\beta_3 - \beta'_3$:
   $\beta''_1 = \{p_2 + p_3, 2 < \underline{t}_2 \leq 3 \wedge 0 \leq \underline{t}_3 < 1 \wedge 2 \leq \underline{t}_2 - \underline{t}_3 \leq 3\}$.
3. Iteration 3: $X_2 = X_2 \cap \pi(X_2) = \{\beta_0, \beta''_1, \beta_2, \beta'_3, \beta_5\}$. The fixpoint X_2 is then the set of winning states. Since the initial state zone belongs to X_2, there is a controller for forcing the property *AG not* $p_1 + p_3 = 0$. To keep the model in safe states (in states of X_2), the controller must delay, in β_0, the firing of t_1 until its clock overpasses the value 2. Doing so, the successor of β_0 by t_1 will be β''_1.

This approach needs however to construct a state space abstraction before computing the winning states. To overcome this limitation, in (Cassez *et al.*, 2005; Tripakis, 1998), the authors have investigated the use of on-the-fly algorithms besides the fixpoint to compute the winning states for timed game automata [1]. The on-the-fly algorithm of (Cassez *et al.*, 2005) for the case of reachability properties is reported, in Fig.3.3. This algorithm uses the following three lists: *Passed*, containing all the state zones explored so far; *Waiting*, containing the set of edges to be processed; and *Depend*, indicating for each state zone S, the set of edges to be reevaluated in case the set of the winning states in S ($Win[S]$) is updated. Using this method, in each step, a part of the state zone graph is constructed and an edge $e = (S, a, S')$ of the *Waiting* list is processed. If the state zone S' is not in *Passed* and there

1. A timed game automata is a timed automata with two kinds of transitions: controllable and uncontrollable transitions.

are some states in S', which satisfy the desired reachability property, then these states are added to the winning states of S' ($Win[S']$). The winning states of S will be recomputed later (the edge e is added to the list $Waiting$). If S' is in $Passed$, $Win[S]$ and possibly those of its predecessors are recomputed and so on. In $Win[S]$, we get the controllable predecessors of the winning states of its successors.

Initialization:

$Passed \leftarrow \{S_0\}$ **where** $S_0 = \{(\ell_0, \vec{0})\}^{\nearrow}$;
$Waiting \leftarrow \{(S_0, \alpha, S') \mid S' = \mathsf{Post}_\alpha(S_0)^{\nearrow}\}$;
$Win[S_0] \leftarrow S_0 \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
$Depend[S_0] \leftarrow \emptyset$;

Main:

while $((Waiting \neq \emptyset) \wedge (s_0 \notin Win[S_0]))$ **do**
  $e = (S, \alpha, S') \leftarrow pop(Waiting)$;
  if $S' \notin Passed$ **then**
    $Passed \leftarrow Passed \cup \{S'\}$;
    $Depend[S'] \leftarrow \{(S, \alpha, S')\}$;
    $Win[S'] \leftarrow S' \cap (\{\mathsf{Goal}\} \times \mathbb{R}^X_{\geq 0})$;
    $Waiting \leftarrow Waiting \cup \{(S', \alpha, S'') \mid S'' = \mathsf{Post}_\alpha(S')^{\nearrow}\}$;
    if $Win[S'] \neq \emptyset$ **then** $Waiting \leftarrow Waiting \cup \{e\}$;
  else (* **reevaluate** *)[a]
    $Win^* \leftarrow \mathsf{Pred}_t(Win[S] \cup \bigcup_{S \xrightarrow{c} T} \mathsf{Pred}_c(Win[T]),$
      $\bigcup_{S \xrightarrow{u} T} \mathsf{Pred}_u(T \setminus Win[T])) \cap S$;
    if $(Win[S] \subsetneq Win^*)$ **then**
      $Waiting \leftarrow Waiting \cup Depend[S]$; $Win[S] \leftarrow Win^*$;
    $Depend[S'] \leftarrow Depend[S'] \cup \{e\}$;
  endif
endwhile

[a] When $T \notin Passed$,$Win[T] = \emptyset$

Figure 3.3 On-the-fly algorithm for timed game automata proposed in (Cassez *et al.*, 2005).

This on-the-fly algorithm is based on computing controllable predecessors and requires some expensive operations such as the difference between state zones. The difference between two state zones is not necessarily a state zone and then may result in several state zones, which need to be handled separately.

3.3 A forward method for computing predecessors of state classes

In this section, we discuss a forward method for computing predecessors of state classes. This method will be used later in our suggested algorithm. Let $\alpha = (M, F)$ be a state class

and $\omega \in T^+$ be a sequence of transitions firable from α. We denote $succ(\alpha, \omega)$ as the state class reachable from α by firing successively transitions of ω. We define inductively this set as follows: $succ(\alpha, \omega) = \alpha$, if $\omega = \epsilon$ and $succ(\alpha, \omega) = succ(succ(\alpha, \omega'), t_i)$, if $\omega = \omega'.t_i$.

During the firing of a sequence of transitions ω from α, the same transition may be newly enabled several times. To distinguish among different enablings of the same transition t_i, we denote t_i^k for $k > 0$ the transition t_i (newly) enabled by the k^{th} transition of the sequence; t_i^0 denotes the transition t_i enabled in M. Let $\omega = t_1^{k_1}....t_m^{k_m} \in T^+$ with $m > 0$ be a sequence of transitions firable from α (i.e., $succ(\alpha, \omega) \neq \emptyset$). We define $Fire(\alpha, \omega)$ the largest subclass α' of α (i.e., $\alpha' \subseteq \alpha$) s.t. ω is firable from all its states, i.e.,

$$Fire(\alpha, \omega) = \{q_1 \in \alpha \mid \exists \theta_1, ..., \theta_m, q_1 \xrightarrow{\theta_1} q_1 + \theta_1 \xrightarrow{t_1^{k_1}} q_2...q_m + \theta_m \xrightarrow{t_m^{k_m}} q_{m+1}\}. \tag{3.2}$$

Proposition 1 *$Fire(\alpha, \omega)$ is the state class (M', F') where $M' = M$ and F' can be computed as follows[2]: Let $M_1 = M$ and the marking reached from M by the subsequence $t_1^{k_1}....t_f^{k_f}$ of ω be denoted by M_{f+1}, where $f \in [1, m]$.*

1. *Initialize F' with the formula obtained from F by renaming all variables $\underline{t}_i$ in $\underline{t}_i^0$.*
2. *Add the following constraints:*

$$\bigwedge_{f \in [1,m]} \Big(\bigwedge_{t_i \in (En(M_1) - \bigcup_{j \in [1,f[} CF(M_j, t_j))} \underline{t}_f^{k_f} - \underline{t}_i^0 \leq 0 \quad \wedge$$

$$\bigwedge_{j \in [1,f[, t_n \in (New(M_{j+1}, t_j) - \bigcup_{k \in]j,f[} CF(M_k, t_k))} \underline{t}_f^{k_f} - \underline{t}_n^j \leq 0$$

$$\wedge \bigwedge_{t_n \in New(M_{f+1}, t_f)} \downarrow Is(t_n) \leq \underline{t}_n^f - \underline{t}_f^{k_f} \leq \uparrow Is(t_n)) \Big)$$

3. *Put the resulting formula in the canonical form and eliminate all variables $\underline{t}_i^j$ such that $j > 0$. Rename all variables $\underline{t}_i^0$ in $\underline{t}_i$.*

Note that $Fire(\alpha, \omega) \neq \emptyset$ (i.e., ω is firable from α) iff ω is feasible in the underlying untimed model and the formula obtained at step 2) above is consistent.

2. We suppose that the truth value of an empty set of constraints is always *true*.

Proof 1 *By first step all variables associated with the transitions of $En(M)$ are renamed ($\underline{t_i}$ is renamed in $\underline{t_i^0}$). This step allows us to distinguish delays of transitions enabled in M from those that are newly enabled by the transitions of the firing sequence.*

Second step adds the firing constraints of the transitions of the sequence (for $f \in [1, m]$). For each transition $t_f^{k_f}$ of the sequence, three blocks of constraints are added. The two first blocks mean that the delay of $t_f^{k_f}$ must be less than or equal to the delays of all transitions enabled in M_f (i.e., transitions of $En(M)$ and those enabled by t_j ($New(M_{j+1}, t_j), 1 \leq j < f$) that are maintained continuously enabled at least until firing $t_f^{k_f}$). Transitions of $En(M)$ that are maintained continuously enabled at least until firing $t_f^{k_f}$ are transitions of $En(M)$ which are not in conflict with $t_1^{k_1}$ in M_1, and, ..., and not in conflict with $t_{f-1}^{k_{f-1}}$ in M_{f-1}. Similarly, transitions of $New(M_j, t_j)$ (with $1 \leq j < f$) that are maintained continuously enabled at least until firing $t_f^{k_f}$ are transitions of $New(M_{j+1}, t_j)$ which are not in conflict with $t_{j+1}^{k_{j+1}}$ in M_{j+1}, and ..., and are not in conflict with $t_{f-1}^{k_{f-1}}$ in M_{f-1}. The third block of constraints specifies the firing delays of transitions that are newly enabled by $t_f^{k_f}$.

Third step isolates the largest subclass of α such that ω is firable from all its states.

As an example, consider the TPN depicted in Fig.2.4 and its state class graph shown at Fig.2.5. In the following, we show how to compute $Fire(\alpha_0, t_1^0 t_2^0 t_3^1)$. We have $En(M_0) = \{t_1, t_2\}$, $CF(M_0, t_1) = \{t_1\}$, $CF(M_1, t_2) = \{t_2\}$, $New(M_0, t_1) = \{t_3\}$ and $New(M_1, t_2) = \{t_4\}$. The subclass $(p_1 + p_2, F') = Fire(\alpha_0, t_1^0 t_2^0 t_3^1)$ is computed as follows:

1. Initialize F' with the formula obtained from $0 \leq \underline{t}_1 \leq 4 \wedge 2 \leq \underline{t}_2 \leq 3$ by renaming all variables $\underline{t}_i$ in $\underline{t}_i^0$:
$$0 \leq \underline{t}_1^0 \leq 4 \wedge 2 \leq \underline{t}_2^0 \leq 3.$$

2. Add the firing constraints of t_1 before t_2, t_2 before t_3 and constraints on the firing intervals of transitions enabled by these firings (i.e., t_3 and t_4):
$$\underline{t}_1^0 - \underline{t}_2^0 \leq 0 \wedge \underline{t}_2^0 - \underline{t}_3^1 \leq 0 \ \wedge \ 2 \leq \underline{t}_3^1 - \underline{t}_1^0 \ \wedge \ 0 \leq \underline{t}_4^2 - \underline{t}_2^0 \leq 1.$$

3. Put the resulting formula in the canonical form and eliminate all variables $\underline{t}_i^j$ such that $j > 0$. Rename all variables $\underline{t}_i^0$ in $\underline{t}_i$:
$$0 \leq \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge 1 \leq \underline{t}_2 - \underline{t}_1 \leq 3.$$

The subclass $Fire(\alpha_0, t_1^0 t_2^0 t_3^1)$ consists of all the states of α_0 from which the sequence $t_1 t_2 t_3$

is firable. If t_1 is controllable, to avoid reaching the marking p_4 by the sequence $t_1t_2t_3$, it suffices to choose the firing interval of t_1 in α_0 outside its firing interval in $Fire(\alpha_0, t_1^0 t_2^0 t_3^1)$ (i.e., $]2, 4]$).

Note that this forward method of computing predecessors can also be adapted and applied to the clock based abstractions. For instance, using the zone based graph, the initial state zone of the TPN shown at Fig.2.4 is $\beta_0 = (p_1 + p_2, 0 \leq \underline{t}_1 = \underline{t}_2 \leq 3)$. The sub-zone β_0' of β_0, from which the sequence $t_1t_2t_3$ is firable, can be computed in a similar way as the previous procedure where delay constraints are replaced by clock constraints,

$$\beta_0' = (p_1 + p_2, 0 \leq \underline{t}_1 = \underline{t}_2 \leq 2).$$

To avoid reaching the marking p_4 by the sequence $t_1t_2t_3$, it suffices to delay the firing of t_1 until when its clocks overpasses 2. This in turns means that its firing interval should be $]2, 4]$.

3.4 On-the-fly algorithm for safety controller synthesis

In this section, we propose an efficient forward on-the-fly method to synthesize a safety controller for time Petri nets. Our method differs from the previous ones, used for timed automata and time Petri nets by the fact that it computes the bad intervals instead of computing the winning states. Our proposed algorithm constructs a state class graph instead of a state zone graph and is not based on computing controllable predecessors (expensive operations). The state class graph is a good alternative for the on-the-fly algorithms as the exploration converges fast and does not need any over-approximation operation to enforce the convergence. In addition, the bad intervals are computed, using a forward approach, for only state classes containing at least a controllable transition.

Let us introduce informally the principle of our approach through the example given in the previous chapter. Consider the TPN of Fig.2.4, its state class graph depicted in Fig.2.5 and its state classes reported in Table 2.2. Our goal is to avoid to reach states where there is no token in both places p_1 and p_3 (i.e., state classes α_4 and α_6), by choosing appropriately the firing intervals of the controllable transition t_1.

From the initial state class α_0, there are two elementary paths $\alpha_0 t_1 \alpha_1 t_2 \alpha_3 t_3 \alpha_6$ and $\alpha_0 t_1 \alpha_1 t_3$ α_4 that lead to bad states. In both paths, there is only one state class (α_0) where t_1 is enabled. To avoid these bad paths, we propose to compute all states of α_0 from which t_1t_3 or

$t_1t_2t_3$ is firable, and denote it by B. Therefore:
$B(\alpha_0) = Fire(\alpha_0, t_1t_3) \cup Fire(\alpha_0, t_1t_2t_3)$, where:
$Fire(\alpha_0, t_1t_3) = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 1 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge 2 \leq \underline{t}_2 - \underline{t}_1 \leq 3)$ and
$Fire(\alpha_0, t_1t_2t_3) = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge 1 \leq \underline{t}_2 - \underline{t}_1 \leq 3)$.
Then, we replace in α_0, the firing interval of t_1 with $]2, 4]$. This interval is the complement of $[0, 2] \cup [0, 1]$ in the firing interval of t_1 in α_0 ($[0, 4]$). The approach we propose in the following is a combination of this principle with a forward on-the-fly method.

A controller for safety properties running in parallel with a system should force the system to satisfy the property '*AG not bad*', where 'bad' stands for the set of forbidden states. This property means that 'bad' states will never happen. In our context, such a system is described by means of a TPN and *bad* is a set of forbidden markings. In our approach, the idea is to explore, path by path, the state class graph of the TPN and look for sequences that lead the system to any forbidden marking (bad sequences or bad paths). Then, Proposition 1 is used to compute the subclasses that cause the bad states, happening later on through the found bad sequences (bad subclasses). The control consists of restricting the firing domain of the controllable transitions so as to avoid the bad subclasses. As we will show, our approach is based on operations over real intervals whereas other approaches are based on operations over time domains. Before presenting our algorithm, let us define some operations over intervals that are used in our algorithm.

3.4.1 Operations over real intervals

Let I and I' be two nonempty (real) intervals. We denote $I \oplus I'$ and $I \ominus I'$ intervals defined by: $\forall a \in \mathbb{R}$,
$a \in I \oplus I'$ iff $\exists b \in I, \exists c \in I', a = b + c$.
$a \in I \ominus I'$ iff $\exists b \in I, \exists c \in I', a = b - c$.
As an example, for $I = [1, 4]$ and $I' =]2, 5]$, $I \oplus I' =]3, 9]$ and $I \ominus I' = [-4, 2[$.

The union of intervals is not necessarily an interval (intervals are not closed under union). It is represented by a set of intervals. Let SI be a nonempty set of intervals. This set is in reduced form if and only if $\forall I, I' \in SI, I \cap I' = \emptyset$. In the rest of this document, we suppose that all sets of intervals are in reduced form. This form makes the operations over sets of intervals easier. For instance, a set of intervals is an interval if and only if its reduced form is a singleton.

Let SI be a nonempty set of real intervals, in reduced form and I be an interval. We

define straightforwardly the following operations over intervals and sets of intervals:

- For $op_1 \in \{\oplus, \ominus\}$, $(I\ op_1\ SI)$ and $(SI\ op_1\ I)$ are the reduced form of $\{I\ op_1\ I' | I' \in SI\}$ and $\{I'\ op_1\ I | I' \in SI\}$, respectively.
- For $op_2 \in \{\subset, \subseteq\}$, $(SI\ op_2\ I)$ iff $(\forall I' \in SI, I'\ op_2\ I)$ and $(I\ op_2\ SI)$ iff $(\exists I' \in SI, I\ op_2\ I')$.
- $I - SI$ is the reduced form of $\{I' \subseteq I | \forall I'' \in SI \wedge I' \cap I'' = \emptyset\}$.

For example, for $I = [1, 4]$, $I' =]2, 5]$ and $I'' = [9, 10]$, we have:

1. $I' \cup I'' = \{]2, 5], [9, 10]\}$,
2. $I \oplus (I' \cup I'') = \{]3, 9], [10, 14]\}$,
3. $I \ominus (I' \cup I'') = \{[-4, 2[, [-9, -6]\}$,
4. $I \not\subseteq (I' \cup I'')$,
5. $I \cup I' \cup I'' = \{[1, 5], [9, 10]\}$
6. $[0, 15] - (I \cup I' \cup I'') = \{[0, 1[,]5, 9[,]10, 15]\}$.

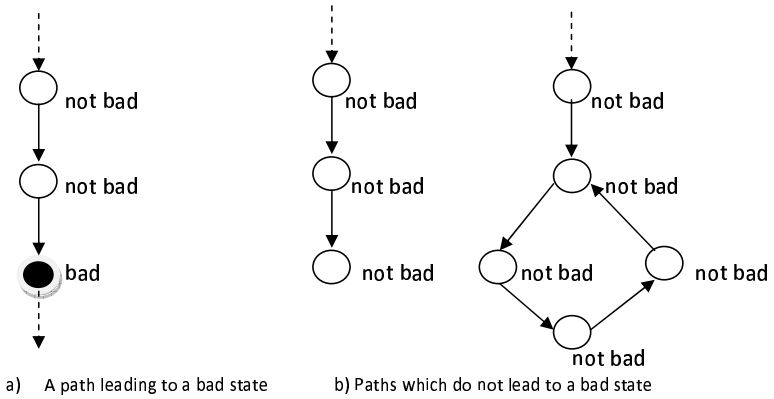


Figure 3.4 Paths satisfying or not a safety property. Black state is to be avoided.

Algorithm 1 On-the-fly algorithm for the safety control of TPN - Part A.

Function $main(TPN\ \mathcal{N},\ Markings\ bad)$
Where $\mathcal{N}$ is a TPN
bad **is a set of bad markings.**
Let T_c be the set of controllable transitions of $\mathcal{N}$ and
α_0 the initial state class of $\mathcal{N}$.
$Passed = \emptyset$
if $(explore(\alpha_0, \epsilon, \{\alpha_0\}) \neq \emptyset)$ **then**
  {Controller does not exist}
  return
end if
for all $((\alpha, \Omega, LI) \in Passed)$ **do**
  $Ctrl[\alpha] = \bigcup_{(t_c, t_s, BI) \in LI} \{(t_c, t_s, INT(\alpha, \underline{t}_c - \underline{t}_s) - BI)\}$
end for

(*)
$\alpha = (M, F)$;
$En_c(M) = En(M) \cap T_c$;
$En_c^0(M) = En_c(M) \cup \{t_0\}$;
$New_c(M, t) = New(M, t) \cap T_c$;
$New(M_0, \epsilon) = En(M_0)$;
t_0 is a fictitious transition whose time variable is fixed at 0.
$Dep(\alpha, t, LI) \equiv \exists (t_c, t_s, BI) \in LI, t_c \notin New(M, t) \wedge (t_s \notin New(M, t) \vee$
$INT(\alpha, \underline{t}_c - \underline{t}_0) \not\subseteq \bigcap_{I \in BI} (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)))$.

Algorithm On-the-fly algorithm for the safety control of TPN - Part B.

Function $Traces\ explore(Class\ \alpha, Trans\ t,\ Classes\ \mathcal{C})$
if $(\exists \Omega, LI$ **s.t.** $(\alpha, \Omega, LI) \in Passed)$ **then**
  if $(\Omega \neq \emptyset \wedge Dep(\alpha, t, LI))$ **then**
    return $\{t.\omega | \omega \in \Omega\}$
  end if
  return $\emptyset$
end if
if $(M \in bad)$ **then**
  return $\{t\}$
end if
$Traces\ \Omega = \emptyset$;
for all $t' \in En(M)$ **s.t** $succ(\alpha, t') \neq \emptyset \wedge succ(\alpha, t') \notin \mathcal{C}$ **do**
  $\Omega = \Omega \cup explore(succ(\alpha, t'), t', \mathcal{C} \cup \{succ(\alpha, t')\})$
end for{Ω **contains all bad sequences of** α**.}**
if $(\Omega = \emptyset)$ **then**
  $Passed = Passed \cup \{(\alpha, \emptyset, \emptyset)\}$
  return $\emptyset$
end if
$LI = \{(t_c, t_s, BI) | (t_c, t_s) \in En_c(M) \times En_c^0(M) \wedge$

$$BI = \bigcup_{\omega \in \Omega} INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \subset INT(\alpha, \underline{t}_c - \underline{t}_s)\}$$

$Passed = Passed \cup \{(\alpha, \Omega, LI)\}$
if $(Dep(\alpha, t, LI))$ **then**
  return $\{t.\omega | \omega \in \Omega\}$
end if
return $\emptyset$

3.4.2 Our algorithm

The method proposed here, is presented in Algorithm 1. The list *Passed* is used to retrieve the set of state classes processed so far, their bad sequences, and the bad intervals of controllable transitions (their domains in bad subclasses). Function *main* consists of an initialization step and a calling to the recursive function *explore*. The call $explore(\alpha_0, \emptyset, \{\alpha_0\})$ returns the set of bad sequences from α_0 that cannot be avoided by restricting firing domains of controllable transitions. If this set is nonempty, it means that such a controller does not exist. Otherwise, it exists and the algorithm guarantees that for each state class α with some bad sequences, there is a possibility to choose appropriately the firing intervals of some controllable transitions so as to avoid all bad subclasses of α. The control of α consists of eliminating, from the firing intervals of such controllable transitions, all parts figuring in its bad subclasses. The restriction of domains is also applied on firing delays between two controllable transitions of α. We get in $Ctrl$, all possibilities of controlling each state class.

The function *explore* receives parameters α being the class under process, t the transition leading to α and $\mathcal{C}$ the set of traveled classes in the current path. It uses functions $succ(\alpha, t)$ and $Fire(\alpha, \omega)$ already explained in sections 2.3.3 and 3.3, respectively. It distinguishes 3 cases:

1. The state class α has been already processed (i.e., α is in *Passed*): In this case, there is no need to explore it again. However, its bad sequences have to be propagated to its predecessor by t, in case the control needs to be started before reaching α in order to avoid bad states of its predecessors. The control of α is independent of its predecessors along the path if all possibilities of control in α are limited to the newly enabled transitions. In case there is, in α, a possibility of control, which limits the firing interval of some controllable transition not newly enabled in α, it means that the predecessor of α by t has some bad states that must be avoided. The condition $Dep(\alpha, t, LI)$, used in Algorithm 1, is derived from the necessary and sufficient conditions, established in the next section, to control α independently from its predecessor by t.

2. The state class α has a forbidden marking (i.e., α is a bad state class): In this case, the transition t is returned, which means that this sequence needs to be avoided before reaching α.

3. In other cases, the function *explore* is called for each successor of α, not already en-

countered in the current path (see Fig.3.4), to collect, in Ω, the bad sequences of its successors. Once all successors are processed, Ω is checked:

- 3.1) If $\Omega = \emptyset$, it means that α does not lead to any bad state class or its bad sequences can be avoided later by controlling its successors. Then, $(\alpha, \emptyset, \emptyset)$ is added to $Passed$ and the function returns with $\emptyset$.
- 3.2) If $\Omega \neq \emptyset$, the function *explore* determines intervals of controllable transitions in bad subclasses, which do not cover their intervals in α. It gets such intervals, identifying states to be avoided, in LI (bad intervals). It adds (α, Ω, LI) to $Passed$ and then verifies whether or not α is controllable independently from its predecessor state class in the current path. In such a case, there is no need to start the control before reaching α and then the empty set is returned by the function. Otherwise, there is a need to propagate the control to its predecessor by t. The set of sequences, obtained by prefixing with t sequences of Ω, is then returned by the function.

3.4.3 Definitions: Bad sequences, bad/winning intervals, losing/winning subclasses

Let $\alpha = (M, F)$ be a state class s.t. $M \notin bad$. The sets $\Omega(\alpha)$, $LI(\alpha)$ and $Ctrl(\alpha)$ are recursively defined as follows. Note that our algorithm computes the bad and winning intervals for only state classes with some bad sequences and controllable transitions. In the following, we suppose that α has some bad sequences and some controllable transitions.

- Bad sequences of α:

$$\Omega(\alpha) = \bigcup_{\alpha \xrightarrow{t} \alpha' = (M',F'), \alpha' \neq \alpha, M' \notin bad} \{t.\omega \mid \omega \in \Omega(\alpha') \wedge Dep(\alpha', t)\} \quad \cup \quad \bigcup_{\alpha \xrightarrow{t} \alpha' = (M',F'), M' \in bad} \{t\} \tag{3.3}$$

where, $Dep(\alpha', t) \equiv \exists (t_c, t_s, BI) \in LI(\alpha)$,
$t_c \notin New(M, t) \wedge (t_s \notin New(M, t) \vee INT(\alpha, \underline{t}_c - \underline{t}_0) \not\subseteq \bigcap_{I \in BI} (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)))$.

This condition states that either the transition is not newly enabled or if it is newly enabled, its interval is not completely included in the set of bad intervals. In other words, for an enabled transition (which is not newly enabled), there exists a sub-interval to safely control the system.

As we will show later, $\neg Dep(\alpha', t)$ is a necessary and sufficient condition for ensuring a control of α, independently from its predecessor by t. In other words, if $Dep(\alpha', t)$ holds then the control must be started before reaching α' by t.

– Bad intervals of α:

$$LI(\alpha) = \{(t_c, t_s, BI) | t_c \in En_c(M), t_s \in En_c^0(M), \\ BI = \bigcup_{\omega \in \Omega(\alpha)} INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \neq INT(\alpha, \underline{t}_c - \underline{t}_s)\}. \tag{3.4}$$

The set BI of each element (t_c, t_s, BI) of $LI(\alpha)$ is the union of intervals of $\underline{t}_c - \underline{t}_s$, in bad subclasses, which does not cover the interval of $\underline{t}_c - \underline{t}_s$ in α. The idea is to use the complement of these intervals in α to characterize states outside the bad subclasses.

– Winning intervals of α:

$$Ctrl(\alpha) = \{(t_c, t_s, WI) | (t_c, t_s, BI) \in LI(\alpha) \wedge WI = INT(\alpha, \underline{t}_c - \underline{t}_s) - BI\}. \tag{3.5}$$

In each element (t_c, t_s, WI) of $Ctrl(\alpha)$, WI is the set of all subintervals of $INT(\alpha, \underline{t}_c - \underline{t}_s)$ that are outside the intervals of $\underline{t}_c - \underline{t}_s$ in $LI(\alpha)$. The control of α consists of choosing an interval I from an element (t_c, t_s, WI) of $Ctrl(\alpha)$ and then restricting the domain of $\underline{t}_c - \underline{t}_s$ in α to I.

– Winning states of α: According to our approach, if $Ctrl(\alpha) \neq \emptyset$, the set of winning states of α, denoted $W(\alpha)$, is defined by:

$$W(\alpha) = \bigcup_{(t_c, t_s, WI) \in Ctrl(\alpha), I \in WI} (M, F \wedge \underline{t}_c - \underline{t}_s \in I). \tag{3.6}$$

– Losing states of α: Similarly, the set of the losing states, denoted $L(\alpha)$ is defined by:

$$L(\alpha) = \bigcup_{(t_c, t_s, BI) \in LI(\alpha), I \in BI} (M, F \wedge \underline{t}_c - \underline{t}_s \in I). \tag{3.7}$$

Note that $\alpha = W(\alpha) \cup L(\alpha)$ since intervals in $Ctrl(\alpha)$ are complements, in α of those in $LI(\alpha)$. Let $B(\alpha) = \bigcup_{\omega \in \Omega(\alpha)} Fire(\alpha, \omega)$ be the set of subclasses of α from which at least one of the bad sequences of $\Omega(\alpha)$ is firable. The following example shows that $L(\alpha)$ is not necessarily equal to $B(\alpha)$. Consider the initial state class α_0 of the previous example (see Fig.3.5). We have:

– $\alpha_0 = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 4 \wedge 2 \leq \underline{t}_2 \leq 3)$,
– $B(\alpha_0) = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge -3 \leq \underline{t}_1 - \underline{t}_2 \leq -1)$,
– $BI = \{(t_1, t_0, [0, 2])\}$,
– $WI = \{(t_1, t_0,]2, 4])\}$,
– $W(\alpha_0) = (p_1 + p_2, 2 < \underline{t}_1 \leq 4 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge -1 < \underline{t}_1 - \underline{t}_2 \leq 2)$,
– $L(\alpha_0) = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge -3 \leq \underline{t}_1 - \underline{t}_2 \leq 0)$, and

– $L(\alpha_0) - B(\alpha_0) = (p_1 + p_2, 1 < \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 < 3 \wedge -1 < \underline{t}_1 - \underline{t}_2 \leq 0)$.

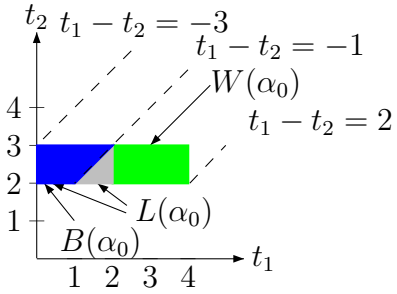


Figure 3.5 The winning and losing subclasses of α_0 in TPN of Fig.2.4 for *AG not* $p_1 + p_3 = 0$.

Note that we cannot limit the firing interval of t_1 in the later subclasses so as to avoid the bad markings, since $INT(L(\alpha_0) - B(\alpha_0), \underline{t}_1 - \underline{t}_0) =]1, 2] \subset INT(B(\alpha_0), \underline{t}_1 - \underline{t}_0) = [0, 2]$.

Note also that our algorithm computes neither $W(\alpha)$ nor $L(\alpha)$. Instead, it computes the bad and winning intervals of the state classes with bad sequences and controllable transitions.

3.4.4 Formalization and proof of the correctness Algorithm1 for safety controller synthesis

Let $\mathcal{N}$ be a time Petri net and *bad* a set of forbidden markings. To synthesize a safety controller for $\mathcal{N}$, our algorithm explores, path by path, its state class graph looking for bad paths. It tries to control the system behavior starting from the last to the first state class of each bad path, so as to avoid reaching forbidden markings. If it fails to control a state class of a path, independently from its previous state classes, the algorithm tries to control its previous state classes and so on. If it succeeds, there is no need to propagate the control to its predecessors in the path. The list $Passed$ is used to store the explored state classes with no bad markings. Each element of $Passed$ is a triplet $(\alpha, \Omega(\alpha), LI(\alpha))$ where $\alpha = (M, F)$ is a state class s.t. $M \notin bad$, $\Omega(\alpha)$ is the set of bad sequences of α, which cannot be avoided, independently from α, from its successors, and $LI(\alpha)$ gives the intervals of controllable transitions in bad subclasses of α (bad intervals). The set $LI(\alpha)$ allows to retrieve the safe intervals of controllable transitions, by computing the complements, in α, of the forbidden intervals (i.e., all possibilities of controlling α, $Ctrl(\alpha)$).

3.4.5 Independent controllable state classes

In our algorithm, to delay the control as much as possible, the control is processed from the last to the first state class of each bad path. The propagation of the control along the path is stopped as soon as we reach a state class whose control has no effect on its predecessors. The control of the path is then initiated at such a state class.

Definition 1 *Let α be a state class reached by some transition t. Formally, α is controllable independently from its predecessor by t if and only if all states of α are made safe by restricting intervals of controllable and newly enabled transitions, i.e., $\forall (t_c, t_s, WI) \in Ctrl(\alpha), \forall I \in WI$,*

$$(M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)} = \alpha_{|En(M)-New_c(M,t)}.$$

Intuitively, this condition means that all possibilities of controlling α affect only the domains of transitions that are controllable and newly enabled. Theorem 1 establishes two necessary and sufficient conditions to control α independently from its predecessor in the path.

Theorem 1 *Let $\alpha = (M, F)$ be a state class reached by some transition t.*

1. *α is controllable independently from its predecessor by t, if and only if $\forall (t_c, t_s, BI) \in LI(\alpha), \forall I \in BI$,*

$$(M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)} = \alpha_{|En(M)-New_c(M,t)}. \tag{3.8}$$

2. *α is controllable independently from its predecessor by t, if and only if*

$$\begin{aligned} &\forall (t_c, t_s, BI) \in LI(\alpha) \text{ s.t. } t_c \notin New(M,t), \\ &\quad (t_s \in New(M,t) \wedge INT(\alpha, \underline{t}_c - \underline{t}_0) \subseteq \bigcap_{I \in BI} (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)). \end{aligned} \tag{3.9}$$

Proof 2 *By definition, α is controllable independently from its predecessor by t if and only if $\forall (t_c, t_s, WI) \in Ctrl(\alpha), \forall I \in WI, (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)} = \alpha_{|En(M)-New_c(M,t)}$. This condition means that the effect of the different possibilities of control is limited to the controllable transitions that are newly enabled. Then, the losing and winning state sets of α are only distinguishable by the firing intervals of transitions that are controllable and newly*

enabled.

1. *By definition of $Ctrl(\alpha)$, intervals of WI are complements of those of the element (t_c, t_s, BI) of $LI(\alpha)$. It follows that $WI \cup BI = INT(\alpha, \underline{t}_c - \underline{t}_s)$ and $WI \cap BI = \emptyset$. If for some interval I of WI, $F \wedge \underline{t}_c - \underline{t}_s \in I$ splits the domain of t_c in α then there is some interval I' of BI such that $F \wedge \underline{t}_c - \underline{t}_s \in I'$ results in splitting the interval of t_c in α. The reverse is also true. Then, we can state that α is controllable independently from its predecessor by t, if and only if*

$$\forall (t_c, t_s, BI) \in LI(\alpha), \forall I \in BI, \\ (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)} = \alpha_{|En(M)-New_c(M,t)}. \tag{3.10}$$

2. *Let us now show that the condition 3.10 above, is equivalent to the condition 3.9. We consider separately different cases:*

   (a) *$t_c \in New_c(M,t)$ and $t_s \in New_c^0(M,t)$: According to the firing rule of transitions from state classes, F can be rewritten as a conjunction of two independent sets of constraints: $F_1 \wedge \bigwedge_{t_n \in New(M,t)} \downarrow Is(t_n) \leq \underline{t}_n \leq \uparrow Is(t_n)$, where F_1 has no constraint over transitions of $New(M,t)$.*
   Then, $\alpha_{|En(M)-New_c(M,t)} = (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)}$.

   (b) *$t_c \notin New_c(M,t)$ and $t_s \notin New_c(M,t)$: Since both t_c and t_s are in $En(M) - New_c(M,t)$ and $I \subset INT(\alpha, \underline{t}_c - \underline{t}_s)$, it follows that $\alpha_{|En(M)-New_c(M,t)} \neq (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)}$.*

   (c) *$t_c \notin New_c(M,t)$ and $t_s \in New_c(M,t)$: In this case, $F \wedge \underline{t}_c - \underline{t}_s \in I$ adds the constraint $\underline{t}_c \in I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)$ to the domain of t_c. This domain is not affected if and only if $INT(\alpha, \underline{t}_c - \underline{t}_0) \subseteq (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0))$. Since $t_c \notin New(M,t)$ and $t_s \in New(M,t)$, if the domain of $\underline{t}_c$ is affected then $\alpha_{|En(M)-New_c(M,t)} \neq (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)}$. Otherwise, the domains of all transitions, except the one of $\underline{t}_s$, are consequently not affected by the added constraint. Therefore, $\alpha_{|En(M)-New_c(M,t)} = (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)}$ if and only if $INT(\alpha, \underline{t}_c - \underline{t}_0) \subseteq (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0))$. This condition must be satisfied for all I in BI s.t.*

$(t_c, t_s, BI) \in LI(\alpha)$. It is then equivalent to:

$$INT(\alpha, \underline{t}_c - \underline{t}_0) \subseteq (\bigcap_{I \in BI} (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0))).$$

(d) $t_c \in New_c(M,t)$ and $t_s \notin New_c(M,t)$: This case is covered by the previous one, since $F \wedge \underline{t}_c - \underline{t}_s \in I$ is equivalent to $F \wedge \underline{t}_s - \underline{t}_c \in I'$, where $I' = [0,0] \ominus I$ belongs to the set BI' of the tuple (t_s, t_c, BI') of $LI(\alpha)$. Following the same steps as the previous case, we show that:
$\alpha_{|En(M)-New_c(M,t)} = (M, F \wedge \underline{t}_c - \underline{t}_s \in I)_{|En(M)-New_c(M,t)}$ if and only if $INT(\alpha, \underline{t}_s - \underline{t}_0) \subseteq (\bigcap_{I' \in BI'} I' \oplus INT(\alpha, \underline{t}_c - \underline{t}_0))$.
To sum up, α is controllable independently from its predecessor by t if and only if $\forall (t_c, t_s, BI) \in LI(\alpha)$ s.t. $t_c \notin New(M,t)$, $t_s \in New(M,t) \wedge INT(\alpha, \underline{t}_c - \underline{t}_0) \subseteq \bigcap_{I \in BI} I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)$.

The second necessary and sufficient condition established in Theorem 1 is very useful as it does not need computing the losing subclasses of α and consists of simple operations over intervals of controllable transitions.

3.4.6 Legal safety controllers

A safety controller is legal if it ensures that the system under control is maintained inside safe states (i.e., states satisfying the safety property of interest).

Let α be a state class, $\Omega(\alpha)$ the set of bad sequences of α to be avoided from α and $B(\alpha) = \bigcup_{\omega \in \Omega(\alpha)} Fire(\alpha, \omega)$. Lemma 1 establishes that $W(\alpha) \subseteq \alpha - B(\alpha)$, which means that all bad sequences of $\Omega(\alpha)$ are not firable from $W(\alpha)$. Other bad sequences of α (not in $\Omega(\alpha)$) are avoided from its successors independently from α. The restrictions made by its successors to avoid these bad sequences have no effect on α. Therefore, our algorithm synthesizes safety legal controllers of α.

Lemma 1 $W(\alpha) \subseteq \alpha - B(\alpha)$.

Proof 3 *By definition, $W(\alpha) \subseteq \alpha$. Let s be a state of $W(\alpha)$. It means that:*
$\exists (t_c, t_s, WI) \in Ctrl(\alpha), \exists I \in WI$ s.t. $Id(t_c) \ominus Id(t_s) \subseteq I$.
Since I is outside all intervals of $(\underline{t}_c - \underline{t}_s)$ in $B(\alpha)$, it follows that $s \notin B(\alpha)$.

3.4.7 Maximally permissive controllers

In our context, to synthesize a maximally permissive controller, the winning states of each state class must be as large as possible. In other terms, the set of bad sequences must be reduced as much as possible (by delaying the control as much as possible) and the winning states w.r.t. the reduced set of bad sequences must be the largest one.

Lemma 2 proves that, for a given set of bad sequences $\Omega(\alpha)$, the firing intervals of controllable transitions cannot be used to distinguish between $B(\alpha)$ and $L(\alpha) = \alpha - W(\alpha)$. For each controllable transition of $En_c(M)$ and a pair of controllable transitions, there are some states in $B(\alpha)$, which share some firing delays with states of $L(\alpha)$. In other words, it means that $W(\alpha)$ is the largest set of winning states in α w.r.t. $\Omega(\alpha)$.

Let $\alpha = (M, F)$ be a state class and $s = (M, Id)$ a state. Then $s \in \alpha$ iff $\forall t \in En(M), \forall t' \in En(M) \cup \{t_0\}, Id(t) \ominus Id(t') \subseteq INT(\alpha, \underline{t} - \underline{t'})$. Recall that t_0 is a fictitious transition which it is never fired and whose delay is fixed at 0. By convention, $Id(t_0) = [0, 0]$. This transition is used for economy of notations.

Lemma 2 $\forall s = (M, Id) \in L(\alpha), \forall t_c \in En_c(M), \forall t_s \in En_c(M) \cup \{t_0\}, \exists \omega \in \Omega(\alpha),$

$$(Id(t_c) \ominus Id(t_s)) \cap INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \neq \emptyset.$$

Proof 4 *Let* $s = (M, Id) \in \alpha - W(\alpha) = L(\alpha)$. *Using the definition of* $W(\alpha)$, *we can state that* $\forall (t_c, t_s, WI) \in Ctrl(\alpha), \forall I \in WI, Id(t_c) \ominus Id(t_s) \not\subseteq I$. *The relationship between* LI *and* WI *allows to rewrite the previous relation as follows:* $\forall (t_c, t_s, BI) \in LI(\alpha), \forall I \in INT(\alpha, \underline{t}_c - \underline{t}_s) - \bigcup_{\omega \in \Omega(\alpha)} INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s), Id(t_c) \ominus Id(t_s) \not\subseteq I$.
Then, $\forall (t_c, t_s, BI) \in LI(\alpha), \exists \omega \in \Omega(\alpha),\ INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \cap Id(t_c) \ominus Id(t_s) \neq \emptyset$.
For $(t_c, t_s) \in En_c(M) \times En_c^0(M)$ *which does not appear in* $LI(\alpha)$, *the following relation holds:* $\exists \omega \in \Omega(\alpha), INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) = INT(\alpha, \underline{t}_c - \underline{t}_s)$.
In conclusion, we can state that:
$\forall s = (M, Id) \in \alpha - W(\alpha), \forall t_c \in En_c(M), \forall t_s \in En_c(M) \cup \{t_0\},$
$\exists\ \omega \in \Omega(\alpha), \quad (Id(t_c) \ominus Id(t_s)) \cap INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \neq \emptyset.$

Using Lemma 2 and Theorem 1, in the following, we prove that our algorithm gives, if it exists, a maximally permissive state dependent controller for a given bounded TPN and a safety property over markings.

Theorem 2 *1. Algorithm 1 guarantees to find a maximally permissive controller, based on the restriction of firing intervals of controllable transitions, if it exists.*

2. Our approach is decidable for all bounded TPNs.

Proof 5 *1. This approach explores the state class graph and collects bad paths. The state class graph preserves markings and firing sequences. Therefore, all bad sequences and bad markings of the state class graph are really bad sequences and reachable bad markings of the model. This approach concludes that the controller does not exist for the considered property if and only if the function explore returns a nonempty set for the initial state class α_0. This in turn means that α_0 has some bad sequences that cannot be avoided by restricting intervals of controllable transitions (i.e., $W(\alpha_0) = \emptyset$). If $W(\alpha_0) \neq \emptyset$ then we can avoid all bad paths by restricting firing intervals of controllable transitions (such a controller exists). Otherwise, these restrictions are not sufficient to avoid bad paths (such a controller does not exist). Lemma 1 shows that for a given set of bad sequences $\Omega(\alpha)$ of α, $W(\alpha)$ is the largest winning states. According to Definition 1, if α is controllable independently from its predecessor state class α' by some t, then α and $W(\alpha)$ have the same set of predecessor states by t in α'. Otherwise, there are in α' some losing states which must be avoided to maintain the system inside the winning states. Indeed, by construction, each state of α has a predecessor by t in α. Therefore, there is a subclass in the predecessor state class of α by t s.t. its successor by t is outside $W(\alpha)_{|En(M)-New_c(M,t)}$. It follows that $\Omega(\alpha)$ is the smallest set of bad sequences which cannot be avoided independently from α from its successors. Therefore, our algorithm gives maximally permissive controllers.*

2. This approach is decidable for any bounded TPN because its state class graph is finite and the approach explores, path by path, the state class graph. The exploration of a path is abandoned as soon as a loop is detected or a bad marking is reached.

Remark:

Let us investigate the hardness complexity of the solution step by step:

- The complexity of computing BI of a state class (in worst case) is $O(K \times L \times {n_t}^2)$ where K is the number of paths starting from the state class and leading to bad markings, L is the maximal length of such paths and n_t is the number of transitions in the model.
- The complexity of relation 3.8 is $O(|LI| \times |BI|)$. The test of equality is $O(1)$.

– The complexity for relation 3.9 is $O(|LI| \times m \times |BI|)$, where m is the number of places in the model (complexity of testing $New(M, t)$).

In (Boucheneb *et al.*, 2009), the authors have discussed the completeness complexity for bounded TPNs. They have proven that model checking of TPN-TCTL formula on a bounded TPN is PSPACE-complete.

3.4.8 Illustrative examples

Case of a TPN with one controllable transition

Let us first apply our approach on the TPN at Fig.2.4 for the safety property *AG not* $p_1 + p_3 = 0$. Its SCG and its state classes are reported in Fig.2.5 and Table 2.2, respectively. For this example, we have $T_c = \{t_1\}$, $bad = \{p_2, p_4\}$, $Passed = \emptyset$ and $\alpha_0 = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 4 \wedge 2 \leq \underline{t}_2 \leq 3)$.

The process starts by calling $explore(\alpha_0, \epsilon, \{\alpha_0\})$ (see Fig.3.6). Since α_0 is not in $Passed$ and its marking is not forbidden, *explore* is successively called for the successors of α_0: $explore(\alpha_1, t_1, \{\alpha_0, \alpha_1\})$ and $explore(\alpha_2, t_2, \{\alpha_0, \alpha_2\})$. In *explore* of α_1, function *explore* is successively called for α_3 and α_4. In *explore* of α_3, function *explore* is called for the successor α_6 of α_3 by t_3: $explore(\alpha_6, t_3, \{\alpha_0, \alpha_1, \alpha_3, \alpha_6\})$. For the successor of α_3 by t_4 (i.e., α_0), there is no need to call *explore* as it belongs to the current path. Since α_6 has a forbidden marking, *explore* of α_6 returns to *explore* of α_3 with $\{t_3\}$, which, in turn, adds $(\alpha_3, \{t_2t_3\}, \emptyset)$ to $Passed$ and returns to *explore* of α_1 with $\{t_2t_3\}$.

In *explore* of α_1, function *explore* is called for α_4 ($explore(\alpha_4, t_3, \{\alpha_0, \alpha_1, \alpha_4\})$). This call returns, to *explore* of α_1, with $\{t_3\}$, since α_4 has a forbidden marking. In *explore* of α_1, the tuple $(\alpha_1, \{t_2t_3, t_3\}, \emptyset)$ is added to $Passed$ and $\{t_1t_2t_3, t_1t_3\}$ is returned to *explore* of α_0. Then, *explore* of α_0 calls $explore(\alpha_2, t_2, \{\alpha_0, \alpha_2\})$, which in turn calls $explore(\alpha_5, t_1, \{\alpha_0, \alpha_2, \alpha_5\})$. Since α_5 has only one successor (α_0) and this successor belongs to the current path, the call of *explore* for α_5 adds $(\alpha_5, \emptyset, \emptyset)$ to $Passed$ and returns to *explore* of α_2 with $\emptyset$, which, in turn, returns to *explore* of α_0.

After exploring both successors of α_0, in *explore* of α_0, we get in $\Omega = \{t_1t_2t_3, t_1t_3\}$ the set of bad paths of α_0. As the state class α_0 has a controllable transition t_1, its bad subclasses are computed: $Fire(\alpha_0, t_1t_2t_3) = \{(p_1 + p_2, 0 \leq \underline{t}_1 \leq 2 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge 1 \leq \underline{t}_2 - \underline{t}_1 \leq 3)$ and $Fire(\alpha_0, t_1t_3) = (p_1 + p_2, 0 \leq \underline{t}_1 \leq 1 \wedge 2 \leq \underline{t}_2 \leq 3 \wedge 2 \leq \underline{t}_2 - \underline{t}_1 \leq 3)\}$. The fir-

ing interval of t_1 in α_0 ($[0,4]$) is not covered by the union of intervals of t_1 in bad subclasses of α_0 ($[0,2] \cup [0,1] \neq [0,4]$). Then, $(\alpha_0, \{t_1t_2t_3, t_1t_3\}, \{(t_1, t_0, \{[0,2]\})\})$ is added to *Passed*. As t_1 is newly enabled, the empty set is returned to the function *main*, which concludes that a controller exists. According with the list *Passed*, α_0 needs to be controlled ($Ctrl[\alpha_0] = \{(t_1, t_0, \{[0,4] - [0,2]\})\}$). For all others, there is nothing to do.

Note that for this example, it is possible to carry out a static controller, which is in this case, a mapping over controllable transitions. Indeed, it suffices to replace the static interval of t_1 with $]2,4]$. Such a controller is in general less permissive than the state dependent controller. However, its implementation is static and very simple as if the model is corrected rather than controlled.

It is also possible to carry out a marking dependent controller (a mapping over markings). Such a controller can be represented by duplicating t_1, each of them being associated with an interval and conditioned to a marking (see Table 3.1 and Fig.3.7).

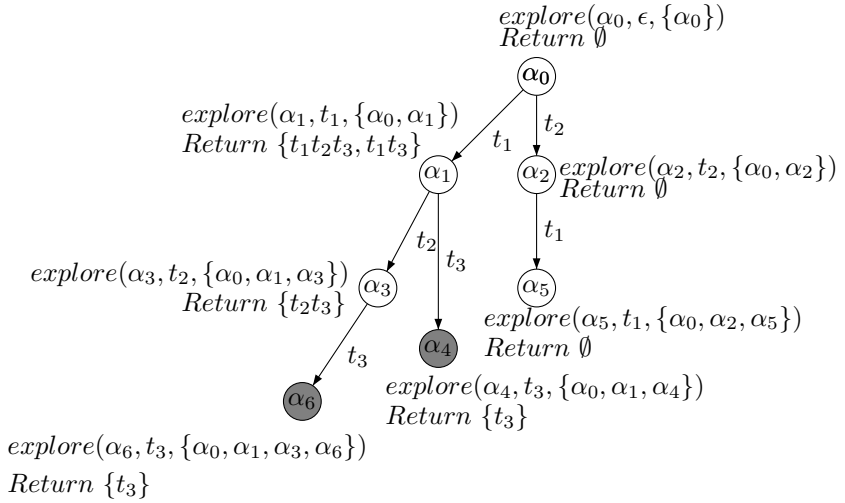


Figure 3.6 Applying Algorithm 1 on the TPN of Fig.2.4 for *AG not* $p_1 + p_3 = 0$.

Case of a TPN with two controllable transitions

As a second example, consider the model depicted in Fig.3.8, which describes the assembly section of a manufacturing line, where two pieces from types A and B are assembled and stored. The assembly section consists of two robotic arms, a conveyor and an assembly tray. Each robotic arm is assigned to bring a part from corresponding type and put it on the

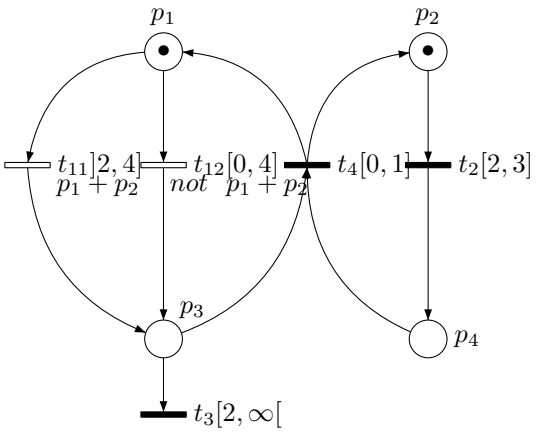


Figure 3.7 The controlled TPN obtained for the TPN of Fig.2.4 for *AG not* $p_1 + p_3 = 0$.

Table 3.1 A marking dependent controller for the TPN of Fig.2.4.

Marking	Constraint to be applied on t_1
$p_1 + p_2$	$2 < \underline{t}_1 \leq 4$
Others	$0 \leq \underline{t}_1 \leq 4$

conveyor (specified by transitions t_1 and t_2). The conveyor runs and places the parts on the tray where they are assembled and stored in the boxes (specified by transitions t_3 and t_4). The conveyor is always on and moves on non-stop. The assembling operation is done on the tray and is uncontrollable. Storing an assembled part is also uncontrollable. The assembly tray should receive two parts from two different types consecutively. It is not allowed to accept more than one part on the conveyor. Then, a controller is needed for the robotic arms to manage the system so as to meet the previous requirement (i.e., the safety property *AG* $Conveyor < 2$).

Table 3.2 State classes of the TPN at Fig.3.8.

$\alpha_0 : A + B + Conv.ON$	$1 \leq \underline{t}_1 \leq 7 \wedge 2 \leq \underline{t}_2 \leq 6$
$\alpha_1 : Conveyor + B + Conv.ON$	$1 \leq \underline{t}_2 \leq 5 \wedge 2 \leq \underline{t}_3 \leq 4$
$\alpha_2 : Conveyor + A + Conv.ON$	$0 \leq \underline{t}_1 \leq 5 \wedge 2 \leq \underline{t}_3 \leq 4$
$\alpha_3 : Tray + B + Conv.ON$	$0 \leq \underline{t}_2 \leq 3$
$\alpha_4 : 2Conveyor + Conv.ON$	$0 \leq \underline{t}_3 \leq 4$
$\alpha_5 : Tray + A + Conv.ON$	$0 \leq \underline{t}_1 < 3$
$\alpha_6 : Tray + Conveyor + Conv.ON$	$2 \leq \underline{t}_3 \leq 4$
$\alpha_7 : 2Tray + Conv.ON$	$10 \leq \underline{t}_4 \leq 10$

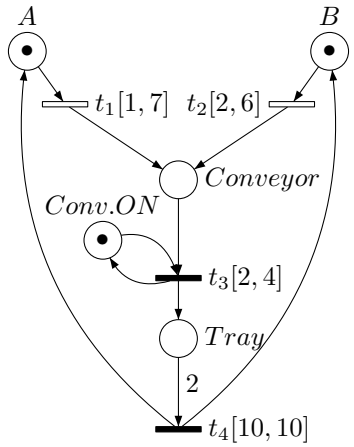


Figure 3.8 The TPN model of the assembling section in a manufacturing line.

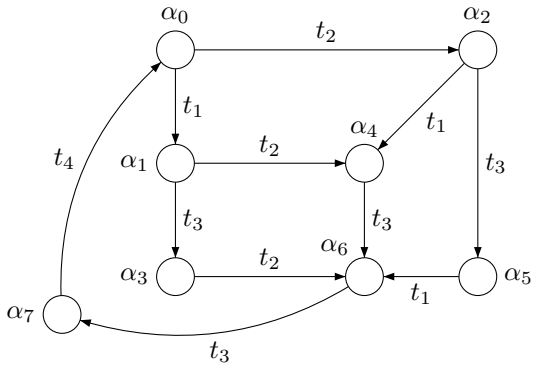


Figure 3.9 The state graph of the TPN presented at Fig.3.8.

The state class graph of the model and its state classes are given in Fig.3.9 and Table 3.2, respectively. Only two transitions t_1 and t_2 are controllable. The transition t_1 models the operation, 'Bring A' while t_2 models 'Bring B' respectively. As the state classes of Fig.3.9 and Table 3.2 show, the state class α_4 violates this property and then is a forbidden state class. The controller should prevent the system entering this state class. We trace our algorithm to find a controller. The sequences are reported in Table 3.3.

Our algorithm concludes that there is a controller. All possibilities of control of state classes are given in $Ctrl$: $Ctrl(\alpha_0) = \{(t_1, t_2, \{[-5, -4[,]4, 5]\})\}$, $Ctrl(\alpha_1) = \{(t_2, t_0, \{]4, 5]\})\}$,

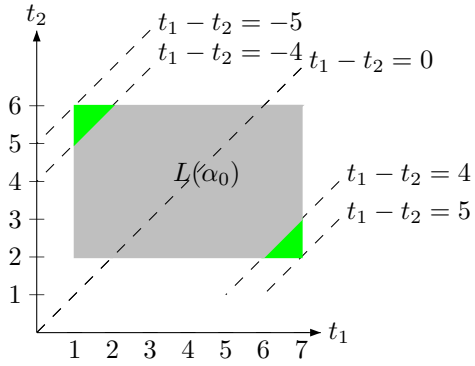


Figure 3.10 The winning and bad subclasses of α_0 in the TPN of Fig.3.8, w.r.t. $AG\ Convoyer < 2$.

Table 3.3 Tracing Algorithm 1 on the example of Fig.3.8.

Class	Condition / Decision
α_0	call *explore* for α_1
α_1	call *explore* for α_4
α_4	$M_4 \in bad \rightarrow$ return $\{t_2\}$
α_1	call *explore* for α_3
α_3	call *explore* for α_6
α_6	call *explore* for α_7
α_7	$succ(\alpha_7, t_4) = \alpha_0 \in \mathcal{C} \rightarrow$ add $(\alpha_7, \emptyset, \emptyset)$ to $Passed$, return $\emptyset$
α_6	add $(\alpha_6, \emptyset, \emptyset)$ to $Passed$, return $\emptyset$
α_3	add $(\alpha_3, \emptyset, \emptyset)$ to $Passed$, return $\emptyset$
α_1	add $(\alpha_1, \{t_2\}, \{(t_2, t_0, \{[0,4]\}\})$ to $Passed$, $t_2 \notin New(M_1, t_1) \rightarrow$ return $\{t_1 t_2\}$
α_0	call *explore* for α_2
α_2	call *explore* for α_4
α_4	$M_4 \in bad \rightarrow$ return $\{t_1\}$
α_2	call *explore* for α_5
α_5	add $(\alpha_5, \emptyset, \emptyset)$ to $Passed$, $succ(\alpha_6, t_3) = \alpha_7$ is in $Passed \rightarrow$ return $\emptyset$
α_2	add $(\alpha_2, \{t_1\}, \{(t_1, t_0, \{[0,4]\}\})$ to $Passed$, $t_1 \notin New(M_2, t_2) \rightarrow$ return $\{t_2.t_1\}$
α_0	$LI = \{(t_1, t_2, \{[-4,4]\})\}$, add $(\alpha_0, \{t_1 t_2, t_2 t_1\}, LI)$ to $Passed$, return $\emptyset$
main	$Ctrl(\alpha_0) = \{(t_1, t_2, \{[-5,-4[,]4,5]\})\}$ $Ctrl(\alpha_1) = \{(t_2, t_0, \{]4,5]\})\}$ $Ctrl(\alpha_2) = \{(t_1, t_0, \{]4,5]\})\}$

$Ctrl(\alpha_2) = \{(t_1, t_0, \{]4,5]\})\}$, $Ctrl(\alpha_i) = \emptyset$, for $i \in \{3,5,6,7\}$. Note that the control must be started before reaching α_1 (resp. α_2) because the controllable transition of α_1 (resp. α_2) is not newly enabled, which means that there are some bad states in α_0 (see Fig.3.10).

For this example, a static controller exists and consists of replacing the static firing intervals of t_1 and t_2 with $[1,2[$ and $]5,6]$ (or $]6,7]$ and $[2,3[$), respectively (see Fig.3.10). The marking dependent controller is identical to the state dependent controller because each state class has its own marking.

3.5 Controller for reachability properties

The algorithm proposed here for the safety properties, is also adaptable to reachability properties. A reachability controller running in parallel with the system should satisfy the property $AF\ goal$, where $goal$ is a set of goal markings. This property means that a marking of $goal$ will eventually be reached (see Fig.3.11). For reachability properties, the controller should prevent all paths which terminate without reaching a goal marking, or contains a loop on none goal markings (see Fig.3.11). Then, if we define state classes leading to such cases as bad states, a safety controller is able to control this system in order to satisfy the given reachability property. Thus, the algorithm proposed for safety properties is extensible to the case of reachability properties with some minor modifications (see Algorithm 3). The main function is the same for the both kinds of properties except that in this case, the set of bad markings (bad) is replaced with the set of goal markings ($goal$). In the function $explore$, a bad path is computed for two cases:

- a terminating node, not leading to a $goal$ state. This condition is imposed by verifying whether $(En(M) = \emptyset)$.
- a loop, not including a $goal$ state. This condition is imposed by verifying whether: $succ(\alpha, t') \in \mathcal{C}$.

By preventing these cases, the controller allows only the paths reaching to $goal$ states.

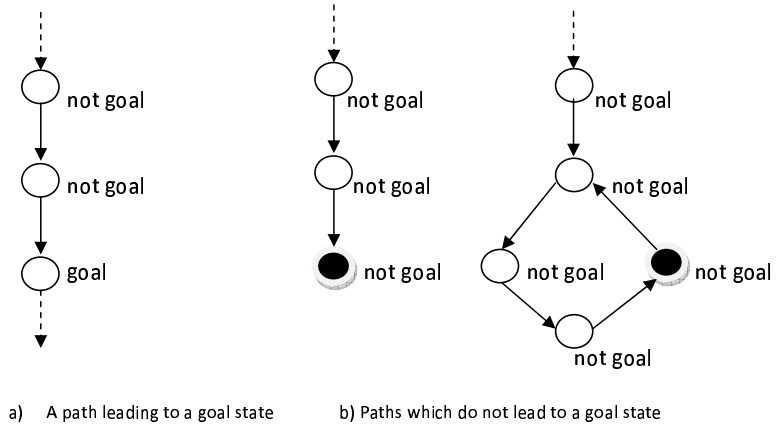


Figure 3.11 Paths satisfying or not a reachability property. Black states are to be avoided.

Algorithm 3 On-the-fly algorithm for the reachability control of TPN - Part A.

Function $main(TPN\ \mathcal{N}, Markings\ goal)$
Where $\mathcal{N}$ is a TPN and
$goal$ **is a set of goal markings.**
Let T_c be the set of controllable transitions of $\mathcal{N}$ and
α_0 **the initial state class of $\mathcal{N}$.**

$Passed = \emptyset$
if $(explore(\alpha_0, \epsilon, \{\alpha_0\}) \neq \emptyset)$ **then**
  {Controller does not exist}
  return
end if

for all $((\alpha, \Omega, LI) \in Passed)$ **do**
  $Ctrl[\alpha] = \bigcup_{(t_c, t_s, BI) \in LI} \{(t_c, t_s, INT(\alpha, \underline{t}_c - \underline{t}_s) - BI)\}$
end for
return

(*)
$\alpha = (M, F)$;
$En_c(M) = En(M) \cap T_c$;
$En_c^0(M) = En_c(M) \cup \{t_0\}$;
$New_c(M, t) = New(M, t) \cap T_c$;
$New(M_0, \epsilon) = En(M_0)$; $New^0(M_0, \epsilon) = En(M_0) \cup \{t_0\}$;
t_0 is a fictitious transition whose time variable is fixed at 0.
$Dep(\alpha, t, LI) \equiv \exists (t_c, t_s, BI) \in LI, t_c \notin New(M, t) \wedge (t_s \notin New(M, t) \vee$
$INT(\alpha, \underline{t}_c - \underline{t}_0) \not\subseteq \bigcap_{I \in BI} (I \oplus INT(\alpha, \underline{t}_s - \underline{t}_0)))$.

Algorithm On-the-fly algorithm for the reachability control of TPN - Part B.

Function $Traces\ explore(Class\ \alpha, Trans\ t,\ Classes\ \mathcal{C})$
if $(\exists \Omega, LI$ **s.t.** $(\alpha, \Omega, LI) \in Passed)$ **then**
  if $(\Omega \neq \emptyset \wedge Dep(\alpha, t, LI))$ **then**
    return $\{t.\omega | \omega \in \Omega\}$
  end if
  return $\emptyset$
end if
if $(M \in goal)$ **then**
  return $\emptyset$
end if
if $(En(M) = \emptyset)$ **then**
  return $\{t\}$
end if
$Traces\ \Omega = \emptyset$
for all $t' \in En(M)$ **s.t** $succ(\alpha, t') \neq \emptyset$ **do**
  if $succ(\alpha, t') \in \mathcal{C}$ **then**
    $\Omega = \Omega \cup \{t'\}$
  else
    $\Omega = \Omega \cup explore(succ(\alpha, t'), t', \mathcal{C} \cup \{succ(\alpha, t')\})$
  end if
end for
if $(\Omega = \emptyset)$ **then**
  $Passed = Passed \cup \{(\alpha, \Omega, \emptyset)\}$
  return $\emptyset$
end if
$LI = \{(t_c, t_s, BI) | (t_c, t_s) \in En_c(M) \times En_c^0(M) \wedge$

$$BI = \bigcup_{\omega \in \Omega} INT(Fire(\alpha, \omega), \underline{t}_c - \underline{t}_s) \subset INT(\alpha, \underline{t}_c - \underline{t}_s)\}$$

$Passed = Passed \cup \{(\alpha, \Omega, LI)\}$
if $(Dep(\alpha, t, LI))$ **then**
  return $\{t.\omega | \omega \in \Omega\}$
end if
return $\emptyset$

3.5.1 Formalization and proof of the correctness of Algorithm3 for reachability controller synthesis

Let $\mathcal{N}$ be a time Petri net and *goal* a set of desired markings to be reached. To synthesize a safety controller for $\mathcal{N}$, our algorithm explores, path by path, its state class graph looking for the nodes terminating a path without reaching a *goal* state and the states terminating a loop on non *goal* states. Then, the controller has to avoid these states. Similar to a safety controller, the reachability controller also tries to control the system behavior starting from the last to the first state class of each bad path, so as to avoid reaching forbidden markings. In this case forbidden markings are from the two categories mentioned above (the states presented in black at Fig.3.11). If the algorithm fails to control a state class of a path, independently from its previous states, the algorithm tries to control its previous state classes and so on. If it succeeds, there is no need to propagate the control to its predecessors in the path. The list $Passed$ is used to store the explored state classes leading to *goal* markings.

The same as for safety controllers, each element of $Passed$ is a triplet $(\alpha, \Omega(\alpha), LI(\alpha))$ where:

- either $\alpha = (M, F)$ is a state class s.t. $M \in goal$,
- or $\exists \alpha' \in Reach(\alpha)$ where $\alpha' = (M', F')$ s.t. $M' \in goal$.

The set $\Omega(\alpha)$ is the set of bad sequences of α, which cannot be avoided from its successors, independently from α. Here, $LI(\alpha)$ gives the intervals of controllable transitions in bad subclasses of α (bad intervals). The set $LI(\alpha)$ allows to retrieve the safe intervals of controllable transitions, by computing the complements, in α, of the forbidden intervals (i.e., all possibilities of controlling α, $Ctrl(\alpha)$).

3.5.2 State dependent controller

Let $\alpha = (M, F)$ be a state class reached by some transition t and $Ctrl(\alpha)$ be the reachability controller in the state α computed by Algorithm 3. Remember the definition of independent state classes given in Definition 1. The conditions of theorem 1 hold for reachability controller synthesis. Consequently, α is controllable independently from its predecessors by t.

3.5.3 Legal reachability controllers

A reachability controller is legal if it ensures that the system under control leads to the goal states (i.e., states satisfying the reachability property of interest).

Lemma 3

$$\forall s \in W(\alpha); s \in goal \vee \exists s' \in Reach(s)\ s.t.\ s' \in goal.$$

Proof 6 *If Lemma 3 is not true and has a counterexample, we should have a state s' such that:*

$$\exists s' \in W(\alpha),\ s.t.\ s' \notin goal \wedge (Reach(s') \cap goal = \emptyset).$$

Look at Fig.3.11. The final states not leading to goal states are supposed to be avoided. Based on the conditions imposed on Algorithm 3 (Part B) for the calculation of bad paths to be avoided, the set bad is defined as:

$$bad = \{s | s \notin goal \wedge Reach(s) \cap goal = \emptyset\}.$$

Then, $s' \in bad$ and is avoided by the controller. On the other hand, from Lemma 1 we have $W(\alpha) \subseteq \alpha - B(\alpha)$. This is impossible to find a state $s' \in W(\alpha)$ s.t. $s' \in bad$.

3.5.4 Maximally permissive reachability controllers

In Section 3.4.7, we discussed maximally permissive controllers and using Theorem 2 proved that Algorithm 1 is maximally permissive and is decidable for bounded TPNs. Let us see if the same result can be carried out for reachability controller synthesis of the Algorithm 3.

Theorem 3 *1. Algorithm 3 guarantees to find a maximally permissive controller, based on the restriction of firing intervals of controllable transitions, if it exists.*

2. Our approach is decidable for all bounded TPNs.

Proof 7 *1. We have considered the terminating states of none goal paths and states closing cycles of non goal states as forbidden states. We have forced the system to avoid them and to allow only the paths reaching a state with goal markings. Then, the same proof as that of Theorem 2 is applicable. This approach explores the state class graph and collects bad paths. The state class graph preserves markings and firing sequences. Therefore, all bad sequences and bad markings of the state class graph are really bad sequences and reachable bad markings of the model. We have proven that the algorithm for safety controllers is maximally permissive. If the controller is restricting the set of forbidden states (i.e. terminating states) as little as possible, then the set of winning states of the system leading to the goal states is as large as possible. The proof of Theorem 2 applies here as well and we conclude that our algorithm gives the maximally permissive controllers.*

2. The proof is completely the same as that of Theorem 2. The proposed approach for reachability controller is decidable for any bounded TPN because its state class graph is finite and the approach explores, path by path, the state class graph. The exploration of a path is abandoned as soon as a goal state is detected. The algorithm aims to collect the paths leading to a terminating state or a state closing a cycle of non goal states. Paths are abandoned as soon as such states are reached.

3.5.5 An example of reachability controller synthesis

In this section, we follow the solution on a simple example to illustrate further the procedure. Inspired from the example of timed game with imperfect information of (Cassez *et al.*, 2007), we have chosen an application of box painting production system. In this system, boxes are placed on a conveyor belt. The corresponding sensor needs between $[1, 5]$ time units to distinguish the presence of a new box. It takes 8 time units to have the box painted. A robotic arm takes the box from the conveyor in $[6, 10]$ time units. If the box is not picked within an appropriate duration, it may fall down from the conveyor in $[8, 10]$. Dropped boxes are damaged and considered as refurbished. Fig.3.12 models this system. The only controllable action is that of the robotic arm named *pick*. We need a reachability controller to push the system towards the place *picked*.

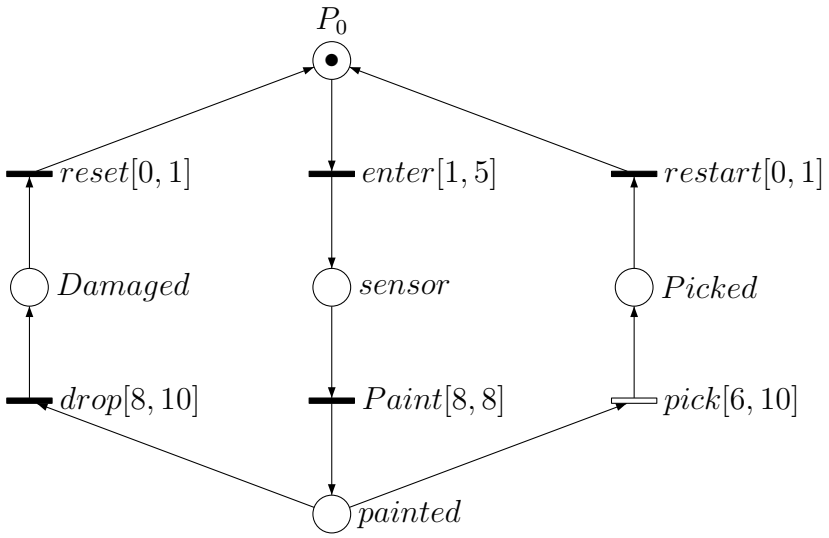


Figure 3.12 A box painting production system.

The state class graph of the system and class information are given in Fig.3.13 and Fig.3.14. From Fig.3.13 and Fig.3.14, we conclude that $goal = \{\alpha_4\}$. On the other hand,

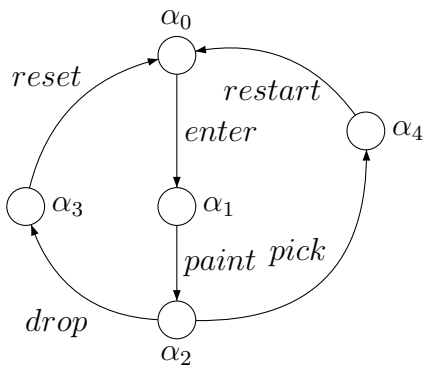


Figure 3.13 The state class graph of the TPN presented at Fig.3.12.

$\alpha_0 : P_0$	$1 \leq enter \leq 5$
$\alpha_1 : sensor$	$8 \leq paint \leq 8$
$\alpha_2 : painted$	$8 \leq drop \leq 10 \wedge$ $6 \leq pick \leq 10$
$\alpha_3 : damaged$	$0 \leq reset \leq 1$
$\alpha_4 : Picked$	$0 \leq restart \leq 1$

Figure 3.14 The state class information of the TPN presented at Fig.3.12.

the state class α_3 is the last state in a loop that does not include a goal state. Remember Fig.3.11. The controller should avoid α_3. Now, let us trace Algorithm 3 on the state class graph. The algorithm starts from α_0 and continues recursively to α_1, α_2 and α_3. At α_3, the condition $succ(\alpha, t') \in \mathcal{C}$ holds. Then, the algorithm returns back to α_2 with $\Omega = \{drop\}$. The other successor available from α_2 is α_4 which is included in *goal*, then the controller returns back to α_2 with $\emptyset$. In α_2, there is a newly enabled controllable transition, *pick* which should be fired before firing of the transition *drop*. The condition $pick < drop$ and consequently, $6 \leq pick < 8$ is imposed. Finally, we obtain: $Ctrl(\alpha_2) = \{(pick, t_0, \{[6, 8[\})\}$. Fig. 3.15 shows the trace of Algorithm 3 on the state class graph depicted in Fig. 3.14.

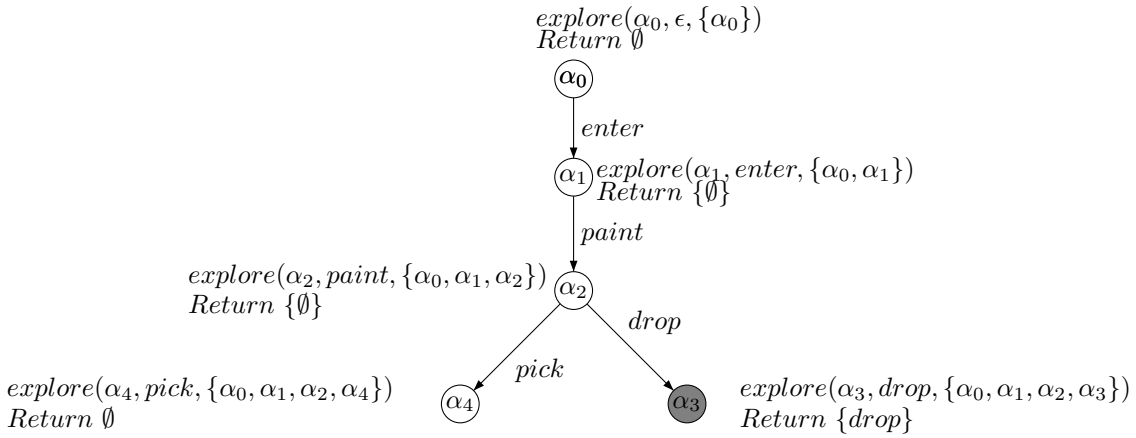


Figure 3.15 Applying Algorithm 3 on the TPN of Fig.3.12 for *AF picked*.

3.6 Conclusion

In this chapter, we have proposed a completely forward on-the-fly algorithm for synthesizing safety and reachability controllers for time Petri nets. Our algorithm explores the state class graph. Meanwhile, in order to make the system satisfy the given property, the algorithm collects for each state class bad paths (i.e. the paths, which do not satisfy the property of interest), and bad intervals (i.e. the intervals of controllable transitions to be avoided) .

The proposed algorithm guarantees to find a state dependent controller based on restriction of firing intervals of controllable transitions, if it exists. Moreover, the synthesized controller is maximally permissive in the category of state dependent controllers based on restriction of firing intervals.

Our algorithm does not need to compute any controllable predecessor as it is the case in the other existing approaches. Computing controllable predecessors involves some expensive operations such as the difference between time domains and then may result in several subclasses, which need to be handled separately. The nice feature of our algorithm is that it is based on simple operations over intervals and classical operations used to compute successor state classes.

The algorithm proposed here is decidable for a bounded TPN because the state class graph is finite and the algorithm explores, path by path, the state class graph (the exploration of a path is abandoned as soon as a loop is detected or a bad state class is reached).

However, one limitation of this approach is combinatorial state explosion. To attenuate the state explosion problem, we will investigate in the next chapter if it is possible to combine this approach with abstraction by inclusion, convex union or convex hull.

CHAPTER 4

Abstraction

4.1 Introduction to the state space abstraction

In order to verify a given property in a TPN model, its state space should be explored. The state space is usually introduced by a transition system. The state space is often very large and difficult to be explored. Working on an abstraction of the state space is a suitable solution.

With the proliferation of large-scale systems, combinatory explosion and state space explosion are challenges that arise while modeling and verification of these systems. Using different methods of abstraction reduces combinatory explosion and state space explosion.

4.2 Abstraction by inclusion, convex union or convex hull

To further attenuate the state explosion problem, the forward on-the-fly reachability algorithm is usually combined with an abstraction by inclusion, convex union or convex hull.

During the construction of an abstraction, each newly computed abstract state is compared with the previously computed ones. In an abstraction by inclusion, two abstract states, with the same untimed information, that their time domains are such that one is included in the other are grouped into one node.

In an abstraction by convex union, two abstract states, with the same untimed information, that their time domains are such that their union is convex are grouped into one node. The convex union abstractions are more compact than inclusion abstractions. However, the convex union test is a very expensive operation relatively to the test of inclusion. The convex union test of n (with $n > 1$) abstract states $\alpha_1, \alpha_2, ..., \alpha_n$ involves computing the smallest enclosing abstract state of their union α, the difference between α and $\alpha_1, \alpha_2, ...\alpha_{n-1}$, and finally checking that this difference is included in α_n.

In an abstraction by convex hull, all abstract states with the same untimed information, are grouped into one node and represented by the smallest enclosing abstract state of their union. The convex hull abstraction requires less costly operations relative to the convex

union abstraction. Unfortunately, the convex hull abstraction is not the most compact abstraction. We give here a simple example of TPN, which is bounded (i.e.: has a finite number of reachable markings) and whose abstraction by convex hull is infinite. It means that this abstraction does not preserve the boundedness property and then markings. Both abstractions by inclusion and convex union preserve markings and then boundedness property.

4.2.1 Inclusion test, convex union test and convex hull of two state classes

Let $\alpha = (M, F)$ and $\alpha' = (M', F')$ be two state classes, D and D' the canonical forms of F and F', respectively. The state class α is included in the state class α' (i.e., $\alpha \subseteq \alpha'$) if and only if $M = M'$ and $\forall t_i, t_j \in En(M) \cup \{t_0\}, d_{ij} \leq d'_{ij}$.

If $M = M'$, the convex hull of state classes α and α', denoted by $\alpha \sqcup \alpha'$, is defined here as in (Daws et Tripakis, 1998), as the smallest state class $\alpha'' = (M, F'')$ such that $\alpha \subseteq \alpha''$ and $\alpha' \subseteq \alpha''$. The canonical form of F'' can be obtained as follows:

$$\forall t_i, t_j \in En(M) \cup \{t_0\}, d''_{ij} = Max(d_{ij}, d'_{ij}).$$

The union of two state classes is not necessarily a state class. If $M = M'$ then the union of α and α', denoted by $\alpha \cup \alpha'$, is convex if and only if $\alpha \cup \alpha' = \alpha \sqcup \alpha'$. So, the test of whether or not the union of α and α' is convex can be obtained using inclusion test and subtract operation as follows: $((\alpha \sqcup \alpha') - \alpha) \subseteq \alpha'$.
This test can be generalized to n state classes α_1,..., α_n as:

$$((\alpha_1 \sqcup \alpha_2 .... \sqcup \alpha_n) - \alpha_1 - \alpha_2 .... - \alpha_{n-1}) \subseteq \alpha_n.$$

4.2.2 How to use an abstraction by inclusion?

To further attenuate the state explosion problem, we investigate how to combine the controller synthesis approach of Section 3.4 with an abstraction by inclusion. This combination has been successfully used in many verification tools. The aim of this abstraction is to avoid to explore successors of a state class in the case we have already explored that state class or a more larger state class. To use this abstraction, in the context of the controller synthesis algorithm proposed in Section 3.4, we need to establish a relationship between the bad sequences of state classes related by inclusion. We must be able to retrieve, using this relationship, the bad sequences of a state class from those of any larger state class.

Let α and α' be two state classes s.t. $\alpha \subseteq \alpha'$. The following lemma establishes that:

1. each bad sequence of α is also a bad sequence of α'.
2. the bad sequences of α can be computed from those of α'.

Lemma 4 *Let $\alpha = (M, F)$ and $\alpha' = (M, F')$ be two state classes s.t. $\alpha \subseteq \alpha'$, $\Omega(\alpha)$ and $\Omega(\alpha')$ their sets of bad sequences. Then:*

1. $\Omega(\alpha) \subseteq \Omega(\alpha')$.

2. $\Omega(\alpha) = \{\omega \in \Omega(\alpha') \mid Fire(\alpha, \omega) \neq \emptyset\}$.

Proof 8 *1. $\Omega(\alpha) \subseteq \Omega(\alpha')$ is trivial, since $\alpha \subseteq \alpha'$.*

2. $\Omega(\alpha) = \{\omega \in \Omega(\alpha') \mid Fire(\alpha, \omega) \neq \emptyset\}$ is also trivial, since $\alpha \subseteq \alpha'$ and $\Omega(\alpha) \subseteq \Omega(\alpha')$.

According to Lemma 4, if a state class α is included in another state class α', we can compute the bad sequences of α from those of α'. Therefore, when the function *explore* is called for a state class α, it is not necessary to explore its successors in case there exists in $Passed$ a state class α' which includes α (i.e., $\exists(\alpha', A', LI') \in Passed$ s.t. $\alpha \subseteq \alpha'$). Indeed, if $\alpha \subseteq \alpha'$, then (according to Lemma 4), the bad sequences of α can be obtained from those of α' as follows: $\Omega(\alpha) = \{\omega \in A' \mid Fire(\alpha, \omega) \neq \emptyset\}$.

Moreover, for every state class reachable from α, there is a state class reachable from α' s.t. the former is included or is equal to the latter. The bad sequences of state classes reachable from α can then be computed using those of state classes reachable from α'.

For the same reasons as above, the abstraction by inclusion can be applied, when we test whether or not a state class has been previously encountered in the current path.

- In function *explore*, the set $\mathcal{C}$ contains the visited state classes of the current path. At this level, if a successor α of the current state class of the path $\mathcal{C}$ is larger than a state class α' of $\mathcal{C}$, there is no need to continue the exploration of α'. We can go back to α', replace in the path α' by α and then explore α.
- Finally, before inserting a state class α in the list $Passed$, all state classes of $Passed$ included in α are removed from $Passed$.

With this abstraction, the on-the-fly algorithm explores, path by path, the state class graph. During the exploration of a path, the current path is abandoned as soon as we reach a state class α such that $\exists(\alpha', A', LI') \in Passed, \alpha \subseteq \alpha'$.

When the exploration is completed, the list *Passed* contains the largest reachable state classes of the model that are not forbidden. In case a controller exists, for each state class of *Passed* with controllable transitions, *Ctrl* indicates how to restrict the firing intervals of its controllable transitions so as to avoid the forbidden markings. The implementation of the controller consists in following the evolution of the model and restricting the firing intervals of the controllable transitions of the current state so as to avoid the forbidden markings. If the current state s belongs to a state class α containing some losing states, the firing interval of an arbitrary controllable transition t_c of s is restricted so as to be included or equal to a firing interval of $AI(\alpha, t_c)$ where $AI(\alpha, t_c) = \bigcup_{(t_c, t_0, GI)} GI$ *s.t.* $(t_c, t_0, GI) \in Ctrl(\alpha)$. As an example, consider the TPN at Fig.2.4 studied previously, its initial state $s_0 = (p_1 + p_2, 0 \leq t_1 \leq 4 \wedge 2 \leq t_2 \leq 3)$ belongs to the initial state class. So, the firing interval of t_1 has to be restricted to the firing interval of t_1 in $AI(\alpha_0, t_1)$ (i.e.,]2,4]).

Note that even if there is no state class α' in *Passed*, which includes the current state class α, it may be included in the union of some state classes of *Passed*. In this case, it is not necessary to explore α since it is possible to determine the bad sequences of α using bad sequences of state classes in the list *Passed*. More precisely, suppose that there exists n elements $(\alpha_1, A_1, LI_1), ..., (\alpha_n, A_n, LI_n)$ in *Passed* s.t. $\alpha \subseteq \alpha_1 \cup .... \cup \alpha_n$. The bad sequences $\Omega(\alpha)$ of α can be obtained as follows:

$$\Omega = \{\omega \in A_1 \cup A_2 ... \cup A_n \mid Fire(\alpha, \omega) \neq \emptyset\}.$$

From a practical point of view, clock difference diagrams (CDDs) (Larsen *et al.*, 1999) can be used to represent the list *Passed*. CDDs allow to represent in a very concise way the union of convex domains. The list *Passed* is handled using only two basic operations (union and inclusion), which are well supported by CDDs.

4.2.3 Can we use an abstraction by convex union?

The convex union abstraction aims at grouping together all state classes such that their union is a state class. This agglomeration principle can be applied to the list *Passed*. All state classes in *Passed* such that their union is a state class are grouped and represented by their convex hull. In this case, their convex hull is exactly their union. Therefore, the bad sequences of their convex hull is the union of their bad sequences.

The test of convexity is a very expensive operation relative to the test of inclusion. More-

over, for a set of state classes, the convex test may fail when taken two by two, but succeed if taken all together. To achieve a high degree of abstraction, we need to test all possible combinations of state classes sharing the same marking and having states in common.

Another limitation of this abstraction is that state classes which share the same marking but their union not being convex are not grouped together. To overcome this limitation, we can use, as for the inclusion abstraction, CDDs to represent non convex unions of state classes.

4.2.4 Can we use an abstraction by convex hull?

With the convex hull abstraction, all state classes with the same marking are grouped together and represented by the smallest enclosing state class (their convex hull). This abstraction might appear the best alternative to attenuate the state explosion problem. It was never investigated in the literature whether abstraction by convex hull preserves the properties like boundedness, reachability and etc. In the following, we show that abstraction by convex hull does not preserve the boundedness property. The TPN at Fig.4.1 is a simple counterexample. The state class graph of this TPN, reported at Fig.4.2.a, is finite and consists of seven states classes and five markings (see Table 4.1). Its abstraction by convex hull, shown at Fig.4.2.b, is infinite. Indeed, the firing sequence $t_1t_2t_6t_6$.... is not feasible in the SCG but feasible in the convex hull abstraction (see Fig.4.2.b and Table 4.1). It produces an infinite number of markings. Therefore, the convex hull abstraction does not preserve reachability, safety and boundedness properties of the TPN.

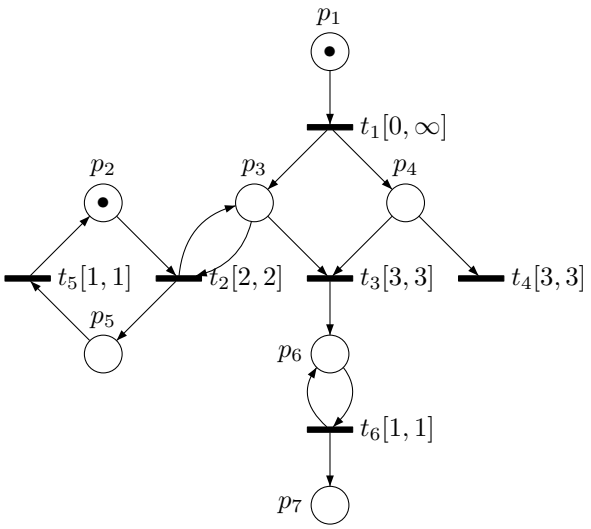


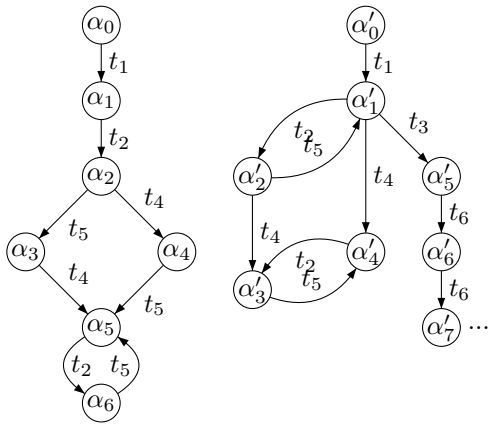
Figure 4.1 A TPN with finite SCG and infinite convex hull abstraction.

In the context of the controller synthesis approach proposed in this manuscript, in the list *Passed*, replacing all the state classes that share the same marking by their convex hull may add some extra bad sequences. The set of bad sequences of their convex hull cannot be obtained from the bad sequences of the grouped state classes. It is then necessary to explore their convex hull and then compute its bad sequences. However, this exploration may never terminate, even if the model is bounded. The convex hull abstraction is then less appropriate than inclusion and convex union abstractions in this context.

4.3 Experiments

We benchmark the above-mentioned abstraction methods through some case studies. We investigate various behaviors involving parallelization and sequentiality. For each case, the state class graphs with and without abstraction methods are constructed and the number of computed state classes (vertices), the number of links and the time taken for state class graph construction is measured. The results are presented in three categories:

1. First category named *aa* is for simple state class graph construction without any abstraction.
2. The category *bb* is for state class graph construction using abstraction by inclusion.
3. The third category *cc* corresponds to state class graph construction with abstraction by convex union.



a) Finite SCG b) Infinite convex hull abstraction

Figure 4.2 SCG and abstraction by convex hull of the TPN at Fig.4.1.

Table 4.1 State classes of the SCG at Fig.4.2.a.

α_0 $p_1 + p_2$ $0 \leq t_1$	α_1 $p_2 + p_3 + p_4$ $t_2 = 2$ $t_3 = t_4 = 3$	α_2 $p_3 + p_4 + p_5$ $t_3 = 3$ $t_4 = t_5 = 1$	α_3 $p_2 + p_3 + p_4$ $t_2 = t_3 = 2$ $t_4 = 0$
α_4 $p_3 + p_5$ $t_5 = 0$	α_5 $p_2 + p_3$ $t_2 = 2$	α_6 $p_3 + p_5$ $t_5 = 1$	
α'_0 $p_1 + p_2$ $0 \leq t_1$	α'_1 $p_2 + p_3 + p_4$ $t_2 = 2$ $2 \leq t_3 \leq 3$ $0 \leq t_4 \leq 3$ $0 \leq t_3 - t_4 \leq 2$	α'_2 $p_3 + p_4 + p_5$ $t_3 = 3$ $0 \leq t_4 \leq 1$ $t_5 = 1$	α'_3 $p_3 + p_5$ $0 \leq t_5 \leq 1$
α'_4 $p_2 + p_3$ $0 \leq t_2 \leq 2$	α'_5 $p_2 + p_6$ $t_6 = 1$	α'_6 $p_2 + p_6 + p_7$ $t_6 = 1$	α'_7 $p_2 + p_6 + 2p_7$ $t_6 = 1$

Even though we have shown that abstraction by convex hull is less appropriate in our context, we also give the results for this abstraction to give a better point of view (category *dd*). Timing values are means of three iterations. The notion $'-'$ shows that computation was impossible because of state explosion. The experiments have been run on a Corei7, 2.2 GHz, equipped with 4 GB of RAM.

4.3.1 Production cell

Inspired from the Production Cell of (Cassez *et al.*, 2005; Lewerentz et Lindner, 1995; Melcher et Winkelmann, 1998), we define the following specifications for our system under study:
Unprocessed plates arrive through a conveyor. A robot with two arms A and B takes the unprocessed plates to a press (by arm A) and places them after being processed on the departure belt (by arm B). All actions of the robot are controllable. The robot should not put more than one plate on the press. The behavior of the system is modeled in Fig.4.3. In our first essay, just two plates exist on the arriving conveyor. The plates are available for the robot within $[0, 1]$ time units. Pressing process is instantly. The robot needs $[2, 4]$ time units

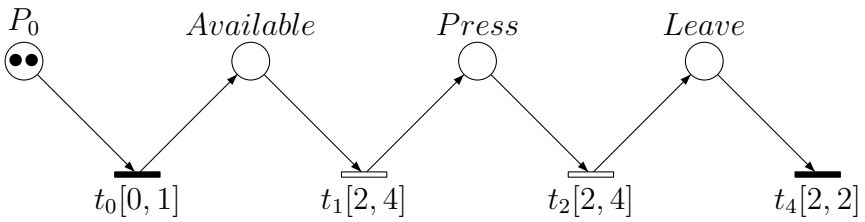


Figure 4.3 A production cell system.

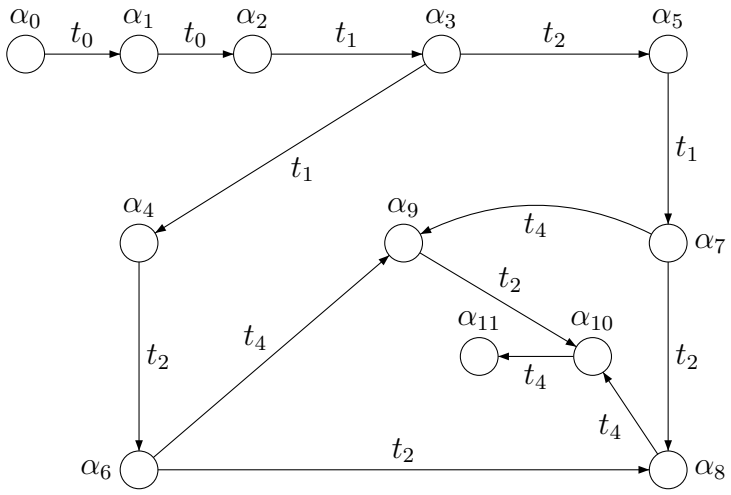


Figure 4.4 The state class graph of the production cell system of Fig.4.3.

to move the plates to/from the press. The processed plates leave the system in $[2, 2]$ time units.

The state class graph of the system before abstraction is composed of 15 state classes while the abstracted state space by inclusion or convex union is composed of only 12 states. We process the abstracted state class graph by inclusion presented at Fig.4.4. Table 4.2 gives the class information of the state class graph of Fig.4.4. The only forbidden state is α_4 and the controller should act at α_3 to fire the transition t_2 before the transition t_1. The condition $2 \leq t_1 < 3 \wedge t_2 = 2$ is imposed at α_3.

Now, suppose the number of unprocessed plates available on the arriving conveyor is increased. The state class graph becomes larger and efficient application of the abstraction methods reduces the number of states during the class graph construction. The obtained

Table 4.2 State classes of the TPN presented at Fig.4.3.

$\alpha_0 : 2P_0$	$0 \leq t_0 \leq 1 \wedge 0 \leq t_0 \leq 1$
$\alpha_1 : P_0 + Available$	$0 \leq t_0 \leq 1 \wedge 2 \leq t_1 \leq 4$
$\alpha_2 : 2Available$	$1 \leq t_1 \leq 4 \wedge 2 \leq t_1 \leq 4$
$\alpha_3 : Available + Press$	$0 \leq t_1 \leq 3 \wedge 2 \leq t_2 \leq 4$
$\alpha_4 : 2Press$	$0 \leq t_2 \leq 4 \wedge 2 \leq t_2 \leq 4$
$\alpha_5 : Available + Leave$	$0 \leq t_1 \leq 1 \wedge 2 \leq t_4 \leq 2$
$\alpha_6 : Press + Leave$	$0 \leq t_2 \leq 4 \wedge 2 \leq t_4 \leq 2$
$\alpha_7 : Press + Leave$	$2 \leq t_2 \leq 4 \wedge 1 \leq t_4 \leq 2$
$\alpha_8 : 2Leave$	$0 \leq t_4 \leq 2 \wedge 2 \leq t_4 \leq 2$
$\alpha_9 : Press$	$0 \leq t_2 \leq 3$
$\alpha_{10} : Leave$	$0 \leq t_4 \leq 2$
$\alpha_{11} : -$	$-$

results for abstraction by inclusion and abstraction by convex union are presented in Table 4.3. Although the abstraction by convex hull is less appropriate in our context, we have presented the results for this abstraction in Table 4.4 to give a better point of view. The second line of each row contains the improvement (reduction) ratio over the state class graph construction in percentage.

The results show that the above-mentioned abstraction methods significantly reduce the number of computed states. State space explosion happens for 12 plates and more without applying any abstraction during the construction of the state class graph. For abstraction by inclusion, state space explosion is distinguished for 15 plates and more. Abstraction by convex union is very efficient in these tests. For larger state spaces, the cost of calculations in terms of the state space construction time, is increased but the state space explosion problem is effectively attenuated.

Although using abstraction methods increases the complexity of computations, it highly reduces the number of state classes. The complexity of controller synthesis approach has a linear relationship with the number of the state classes to be processed. Therefore, the total controller synthesis computations are reduced after applying abstraction methods.

4.3.2 Producer/consumer model

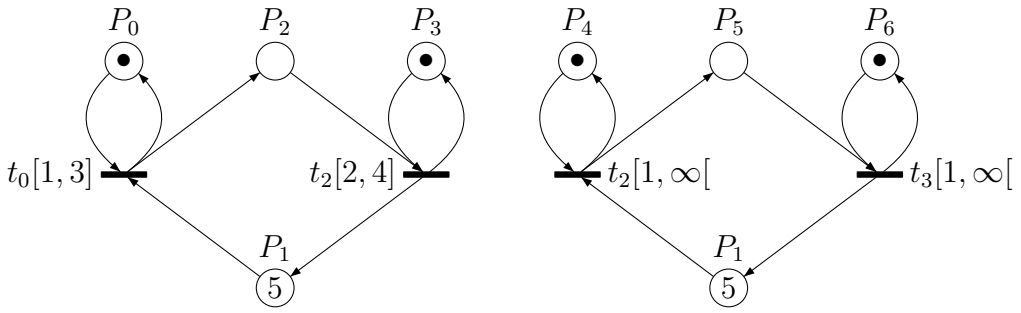
The next study is the example of producer/consumer depicted in Fig.4.5. We have considered pure parallel composition of $n-1$ copies of the model presented in Fig.4.5.a and one copy of the model presented in Fig.4.5.b. We denote this composition by $P(n)$. All places

Table 4.3 Results for different abstraction implementations and different number of available plates of the system of Fig.4.3. The first category *aa* is for non-abstracted state class graph, *bb* is for abstraction by inclusion, *cc* is for abstraction by convex union. The second line of each row is the reduction percentage over SCG construction.

Plates	aa			bb			cc		
Ratio	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*
1	5	4	0ms	5	4	0ms	5	4	0ms
%									
2	15	17	0ms	12	14	0ms	12	14	0ms
%				20	17		20	17	
3	42	56	0ms	25	35	0ms	22	29	0ms
%				40	37		48	48	
4	115	171	0ms	50	78	0ms	35	50	0ms
%				56	54		69	71	
5	304	483	0ms	99	165	0ms	51	77	0ms
%				67	66		83	84	
6	765	1270	10ms	196	340	0ms	70	110	10ms
%				74	73		91	91	
7	1844	3161	20ms	389	691	0ms	92	149	30ms
%				79	78		95	95	
8	4286	7533	50ms	774	1394	0ms	117	194	60ms
%				82	81		97	97	
9	9675	17350	180ms	1543	2801	80ms	145	245	130ms
%				84	84		98	98	
10	21342	38903	390ms	3080	5616	270ms	176	302	210ms
%				85	85		99	99	
11	46218	85380	851ms	6153	11247	941ms	210	365	360ms
%				86	87		99	99	
12	-	-	-	12298	22510	3785ms	247	434	681ms
%									
13	-	-	-	24587	450307	15762ms	287	509	1141ms
%									
14	-	-	-	49164	90092	64683ms	330	590	1772ms
%									
15	-	-	-	-	-	-	376	677	2954ms
%									

Table 4.4 Results for different number of available plates of the system of Fig.4.3. The first category *aa* is for non-abstracted state class graph, *dd* is for abstraction by convex hull. The second line of each row is the reduction percentage over SCG construction.

Plates	aa			dd		
Ratio	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*
1	5	4	0ms	5	4	0ms
%						
2	15	17	0ms	11	12	0ms
%				27	29	
3	42	56	0ms	19	24	0ms
%				47	57	
4	115	171	0ms	29	40	0ms
%				75	77	
5	304	483	0ms	41	60	0ms
%				86	87	
6	765	1270	10ms	55	84	0ms
%				93	93	
7	1844	3161	20ms	71	112	0ms
%				96	64	
8	4286	7533	50ms	89	144	0ms
%				98	98	
9	9675	17350	180ms	109	180	0ms
%				99	99	
10	21342	38903	390ms	131	220	0ms
%				99	99	
11	46218	85380	851ms	155	264	0ms
%				99.6	99.7	
12	-	-	-	181	312	10ms
%						
13	-	-	-	209	364	10ms
%						
14	-	-	-	239	420	10ms
%						
15	-	-	-	271	480	10ms
%						



a) ProdCons1 b) ProdCons2

Figure 4.5 Producer/consumer model.

Table 4.5 Results for different abstraction implementations and different configurations of the system of Fig.4.5. The first category *aa* is for non-abstracted state class graph, *bb* is for abstraction by inclusion, *cc* is for abstraction by convex union. The second line of each row is the reduction percentage over SCG construction.

$P(n)$	aa			bb			cc		
Ratio	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*
P(1)	11	19	0ms	8	14	0ms	8	14	0ms
%				27	26		27	26	
P(2)	748	2460	10ms	41	138	0ms	36	123	0ms
%				94	94		95	95	
P(3)	25135	116940	971ms	338	1853	20ms	252	1374	120ms
%				99	98		99	99	
P(4)	309158	1744288	15181ms	2021	14884	470ms	1466	10687	5788ms
%				99	99		99.5	99	
P(5)	-	-	-	7735	69107	4206ms	5959	51751	116417ms
%									

called P_1 are merged in one single place. Table 4.5 represents the results obtained while constructing the state class graph without any abstraction, with the abstraction by inclusion and the abstraction by convex union. Table 4.6 represents the results without abstraction and the results with the abstraction by convex hull. The second line of each row shows the improvement (reduction) ratio over the state class construction in percentage. Based on the experimental results presented here, these abstraction methods can considerably reduce the number of computed states (up to 99% in this example).

4.4 Conclusion

In this chapter, we have investigated how to combine the approach presented in Section 3.4 with an abstraction by inclusion or convex union to attenuate the state explosion prob-

Table 4.6 Results for different abstraction implementations and different configurations of the system of Fig.4.5. The first category *aa* is for non-abstracted state class graph, *dd* is for abstraction by convex hull. The second line of each row is the reduction percentage over SCG construction.

$P(n)$	aa			dd		
Ratio	*Vertices*	*Links*	*time*	*Vertices*	*Links*	*time*
P(1)	11	19	0ms	6	14	10ms
%				45	26	
P(2)	748	2460	10ms	21	60	0ms
%				97	97.5	
P(3)	25135	116940	971ms	56	210	0ms
%				99.7	99.8	
P(4)	309158	1744288	15181ms	126	560	10ms
%				99.9	99.9	
P(5)	-	-	-	252	1260	30ms
%						

lem. We have shown that the abstraction by convex hull is less appropriate for this approach, since it does not preserve the boundedness properties and then markings.

Experimental results show that abstraction methods can significantly reduce the number of state classes. We have seen that the complexity of the controller synthesis approach suggested in Section 3.4 has a linear relationship with the number of states in the state class graph. Therefore, the application of abstraction methods can highly reduce the time needed for controller synthesis.

In the next chapter, we will discuss how to implement our devised algorithm on large-scale and modular systems.

CHAPTER 5

Decentralized Controller for Modular Systems

5.1 Introduction to decentralized controller

In controller synthesis two main questions should be investigated : existence and implementation of the controller. In chapter 3, we discussed in detail an algorithm to answer the first question, existence of the controller. In this chapter, we answer the second question and investigate the decentralized implementation of a controller on large-scale systems. With the proliferation of large-scale systems, their controller synthesis is an important issue. A large-scale system is often modular by its nature and consists of different modules with the possibility of inter-communication among the modules.

Suppose a controller exists and is synthesized for a large-scale system. Implementing a centralized controller on a modular system is not always easy. In such a case, a decentralized controller is more feasible and preferable. Then, an important challenge in controller synthesis of large-scale systems is how to implement a controller in modular systems. In other words, how to adapt an already synthesized controller to a distributed system and achieve a decentralized controller.

In the literature, we distinguish the following configurations for a controller running in parallel with the system under study:

1. A centralized controller is controlling a plant model (Cassez *et al.*, 2005; Gardey *et al.*, 2006b; Heidari et Boucheneb, 2010, 2012c).
2. A set of decentralized controllers are controlling a large system such that the system is considered as a set of similar modules and is controlled by a set of similar local controllers (Hillah, 2009).
3. A set of decentralized controllers are controlling a system such that the local property in each module can be different from that of the other modules. There is no global property in this case and every local controller controls its own local specification. A local controllable action in one module can be an uncontrollable action in the others. In other words, the set of controllable/uncontrollable actions and the property(ies) to be controlled are all defined locally (Shengbing *et al.*, 2010).

4. A system consists of different modules. However, the local controllers and the local properties are the same. Thus, there is a global property (Abid et Zouari, 2010a).

The first configuration is the first option coming to mind and is the one considered in Section 3.4. As explained earlier, large-scale systems are usually modular by their nature and consequently, this configuration is not directly applicable.

The second option is suitable if all infrastructures are similar. For example in (Hillah, 2009), in order to control the road traffic, a route is divided into multiple similar sections. Although this configuration simplifies both synthesis and implementation, it is not applicable unless all of the modules are completely uniform.

In the third option, the system consists of some independent modules collaborating with each other. Thus, the input of a module can be the output of another one; meaning that, a controllable action in a module can be uncontrollable in another one. Consequently, desired property(ies) of one module may differ from that of the others. An example of this configuration is when some already implemented and controlled modules are assembled together and then, the global specification needs to be verified. Verification of global specifications after implementation is discussed in (David *et al.*, 2010).

Finally, the last configuration is applicable to a system which is modular by its nature with a unique property. In such a system, the modular system is considered as a whole to synthesize a controller. However, in order to implement the synthesized controller, we need to take into account the challenges of a modular systems like intercommunications.

Our objective in this chapter is to investigate the use of the approach suggested in Section 3.4 to derive a set of decentralized controllers to be used in large-scale systems. In other words, our objective is to "think globally, act locally" as stated in (Rudie et Wonham, 1992). We have a unique property and a modular system, so the fourth configuration is convenient in our case (Heidari *et al.*, 2012). In addition, among the four above-mentioned configurations, the last one is more appropriate for real system considerations. We discuss in the following how to adapt a controller to a distributed system with a unique property.

This chapter is organized as follows: Section 5.2 is a survey on decentralized controllers already existing in the literature, Section 5.3 explains how the algorithm introduced in Section 3.4 is adapted to distributed systems as a decentralized controller. Finally, Section 5.4

presents the conclusion and future work.

5.2 Literature review

In Section 3.2, we have given a survey on controller synthesis with different models (TA/TPN), different approaches (analytical, structural and semantic) and different properties (reachability/safety). All of the solutions discussed there match with the first configuration where a plant model is controlled by a centralized controller. In the following, we have a survey on the three other configurations and their available solutions in the literature.

5.2.1 Case of uniform modules and uniform local controllers

In (Hillah, 2009), the main objective is to integrate the formal methods with the distributed system modeling. A modular system and its appropriate controller is investigated as a case study which is related to our objective. In that framework, the controller synthesis is used to study an intelligent transportation system; the idea is to improve the safety and security of a transportation system and to prevent the accidents while increasing the traffic flow.

In (Hillah, 2009), a particular type of Petri nets, called *Instantiable Petri Nets* (IPN), is introduced. IPN provides object oriented modeling capabilities (e.g. hierarchy, encapsulation, etc.), a useful feature in expressing specifications of large-scale systems. IPN is a good alternative for UML with the advantage that it allows automatic verification of properties. However, we focus on their contribution in applying controller synthesis on a distributed system.

The road is divided into sections and each section is composed of a number of cells. A cell is referred to by a tuple (x, y) where y signifies the number of lanes and x corresponds to the horizontal position in the y_{th} lane. In this framework, a decentralized controller consists of similar instances of a local controller, each of them controlling a section (infrastructure). To preserve decidability of the suggested solution, there is up to one uncontrollable vehicle in each section. In each section the minimum and maximum of permitted velocities and accelerations are determined. Besides, two consecutive vehicles shall keep a minimum delay predefined in each section. This delay depends on the least allowed velocity of the section.

A vehicle a is described by a tuple $a =< a.x, a.y, a.v, a.c >$ where x and y show the position of the vehicle in its section, v is its current velocity and c is a flag to indicate if

the vehicle is controllable. Let a and b be two vehicles in a road and assume the relative position of b in horizontal axis relative to a is represented by $b.\Delta x = 0$. The relation $a < b$ holds if : $[a.x < b.x \vee (a.x = b.x \wedge a.y < b.y)]$. For example, if a and b are two successive controllable vehicles and $b.\Delta x = 0$, then the condition $b.y > a.y$ shall be held in order to prevent a collision. Note that in this model, the notion of time is considered implicitly by the fact that it considers the velocity.

Forbidden states describe an accident. The set of forbidden states are defined as those states for which the difference between the velocities of two consecutive vehicles are more than the difference between their positions. Failing states are determined and the controller should prevent them.

A local controller consists of two blocks: $BlackSpotMonitor$ and $ScenarioManager$. The former monitors risky states while the latter searches the solution. Scenario Manager repeats as long as there is a risky event and stops if the risk is null. To prevent an infinite loop, once BlackSpotMonitor detects a risky event, it first checks if a scenario is found for this case; otherwise an object of scenario manager is created for investigation and finding a solution (new_scm). If a scenario already exists for the case, this step is bypassed and the event is simply added to the list.

Note that research presented in (Hillah, 2009) is a case-based solution and is applicable only if all the modules are completely the same. Instances of local controllers are also similar. Although this configuration is very useful for the particular considered approach, it is not a general solution and is less interesting in our case. However, controller synthesis in distributed systems is not their main objective. Our interest is to have a more generic solution which does not require exactly similar modules.

5.2.2 Case of various local properties

In (Shengbing *et al.*, 2010), the authors have discussed a decentralized controller in the modular discrete-event systems where the given property is not global and the local property of interest differs in each module. Solving the control problem in the global system requires solving some other local control problems with multiple local properties. In addition, the set of controllable/uncontrollable (observable/non-observable respectively) actions are not global; meaning that, a controllable transition in the module k may be an uncontrollable action in the module k' (the same for the observability).

The set of events occurring in each module is a subset of events happening in the entire system. In this approach, the authors have presented necessary and sufficient conditions for the existence of such a controller. Meanwhile, they have shown that the satisfaction of a global property is a necessary condition for the satisfaction of every local property (but, not the sufficient condition).

As explained earlier, such configuration is suitable when the objective is to put together already implemented modules which is different from our objective in this thesis.

5.2.3 Case of an identical global property

In (Luo, 2009), the modular synthesis is introduced as a solution to reduce the complexity of the controller synthesis in the Petri nets. The author has suggested to decompose the simple Petri nets into some modules while transforming the control problem into a series of new generalized mutual exclusion constraints (GMEC)(Giua *et al.*, 1992) each of them associated with a module and then, solve the new control problems. In this paper, DES system is modeled by simple Petri nets while the property of interest is formalized by GMECs. In a GMEC formalization, a weighted sum of the markings needs to be less than a constant. Markings satisfying the given constraint are safe; otherwise, they are forbidden.

As reported in (Luo, 2009) (from Lien, 1976), ordinary Petri nets are categorized by limiting their input/output of place/transition to:

- forward conflict free: Petri nets with only one output for each place.
- backward conflict free: Petri nets with just one input for each place.
- forward concurrent free: Petri nets with only one input for each transition.
- backward concurrent free: Petri nets with exactly one output for each transition.
- backward conflict and forward concurrent free (BCFCF): Petri nets in which each node has no more than one input.

Also, two places sharing a common output transition constitute (with their shared transition) a backward concurrent structure (BCS) (Luo, 2009).

The suggested algorithm, exists if the uncontrollable influence subnet is a BCFCF. A BCS is decomposed if its shared transition is uncontrollable but is not connected to a forbidden place. Then, in each module a place is added such that the model becomes closed-loop and

is called a 'monitor' place (see Moody et Antsaklis, 2000). The local controllers work in the "or" logic manner meaning that at least one of the local controllers should be able to control the system.

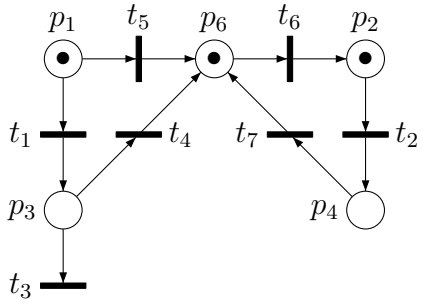


Figure 5.1 An example of the overlapped Petri nets taken from (Aydin et Altug, 2009).

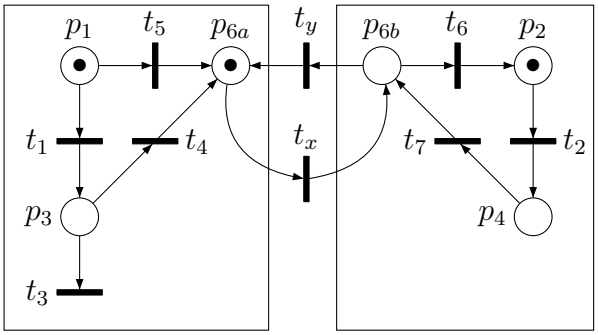


Figure 5.2 Expanded Petri nets of Fig.5.1.

The author has considered forbidden places as a restriction of forbidden markings. With this assumption this algorithm is permissive. In this solution, timing constraints are not considered.

Modeling and controller synthesis of the complex, large-scale systems for a safety property (e.g. deadlock free) in simple Petri nets is the objective of (Aydin et Altug, 2009). In this approach, the property of interest is identical in all of the modules. A large-scale system modeled by simple Petri nets is broken into modules using decomposition overlapping. In (Aydin et Altug, 2009), only overlapping places are allowed while no common transition is permitted. Interconnections between modules is also modeled by the overlapped places. A common place is repeated n times, n being the number of the resulting decomposed modules. If the place is initially marked, the token is only assigned to one of the repeated places. A set of extra transitions are added to facilitate token exchanges among the overlapped places. Suppose the Petri nets presented in Fig.5.1. The overlapping decomposition of the model is shown in Fig.5.2. Two figures 5.1 and 5.2 are both taken from (Aydin et Altug, 2009).

Let us assume a plant system modeled by Petri nets with P being the set of places, as of (Aydin et Altug, 2009) a non-empty set $S \subset P$ is called *siphon* if $Pre(S) \subset Post(S)$. (Pre and $Post$ are already defined in Section 2.3). S is a *minimal siphon* if no other siphon is included in S. In addition, S is called a *controlled siphon* if $M(p) \geq 1$ holds for at least one $p \in S$. On the other hand, if the condition $M_0(p) \geq 1$ holds for at least one $p \in S$, adding a

control place p_c makes S a controlled siphon. The authors have shown that S is a controlled siphon of the global plant if and only if it is a controlled siphon of its corresponding module.

In this model, the controller synthesis consists in computing the overlapping decomposition of the original Petri nets and achieving a set of disjoint sub-Petri nets; finding all minimal siphons in each module and making sure they are all controlled siphons (adding control places if necessary). In this procedure, the control places calculated in each module are finally added to the original Petri nets.

This approach is structural and state space calculation is not required. Besides, local controllers have no intercommunication. The solution is extensible for reachability properties such as enforcing liveness or boundedness. Timing constraints are not considered in this solution.

In (Abid et Zouari, 2010a), the authors have discussed a decentralized active controller for the colored Petri nets with a unique property in all of the modules. First, the reachability graph of the global system is created in order to determine the forbidden markings and the forbidden states (the states having forbidden markings). The controller synthesis aims to prevent these forbidden states by disabling some transitions. The markings (states respectively) from which a transition leads to a forbidden marking (forbidden state respectively) are named *dangerous markings (dangerous states* respectively). The controller should act in the dangerous markings and disable the appropriate controllable transitions in order to prevent the forbidden states.

In this approach, a decentralized controller consists of some local controllers, one for each module. The typical procedure is that each local controller watches the actual state of its associated module, communicates with the other modules to extract the actual global marking of the system, and acts to prevent forbidden states when necessary. Every local controller in its turn is composed of two submodules: an executor submodule and a communication submodule. Whenever a module enters a new state, the communication submodule sends the current local marking to all of the executor submodules and let them extract the actual global marking.

The executor submodule acts upon the information gathered from all other local controllers if the system global state is dangerous. In this way, the executor submodules of the local controllers are aware of the system global marking and the local controllers act

appropriately in the suitable state. To minimize the communications, the communication submodule announces the state of its corresponding module in two cases: either its corresponding module exits from a dangerous state or it enters into a dangerous state (whether from a dangerous or non-dangerous state).

Let us assume a system consisting of n modules, and k be an index ($1 \leq k \leq n$). In each communication submodule, the following transitions and places are defined:

- Place LaS_k holding the latest communicated local marking. Tokens in this place are chosen from the color class C_{flag} containing (d_flag) for the dangerous markings and (o_flag) otherwise.
- Place CLM_k holding the current local marking of the corresponding module.
- Place DLM_k holding the set of dangerous markings.
- Synchronized transition $S - DLM_k$ to send a dangerous local marking.
- Synchronized transition $S - CLM_k$ to send a non dangerous actual local marking when reached from a dangerous global marking.

In the executor submodule the following places are defined:

- Place GM_k holding the actual global marking of the plant system.
- Place DM_k containing the set of dangerous global markings.
- Place AS_k being an alert state of the local controller in each module.
- Place AT_k holding the set of authorizations for the forbidden transitions of the module k. An input/output arc connects this state to every forbidden transition in order to verify the authorization of a firing.

The approach mentioned above is based on the fourth configuration (i.e. a unique global property is controlled in different modules). The modules are not necessarily similar (unlike Hillah, 2009) while the property of interest is identical in all the modules (in contrast with Shengbing *et al.*, 2010). The research presented in (Abid et Zouari, 2010a) is based on a semantic approach without structural or analytical calculations need (in contrary to Aydin et Altug, 2009; Luo, 2009). The modules intercommunicate through synchronized places and transitions. More precisely, places hold messages which are transferred by synchronized transitions and only shared places are allowed. However, the solution of (Abid et Zouari, 2010a) is based on the colored Petri nets and the timing constraints are not considered.

5.3 A decentralized controller for TPN models

In this section, we consider a distributed system consisting of multiple modules, modeled by time Petri nets with a unique property. We investigate if it is possible to break the safety/reachability controller (output of Algorithm 1 or 3) into some modules and have a set of decentralized controllers. Note that the objective is to implement an already synthesized global controller in a modular system, "Think globally, Act locally" as stated in (Rudie et Wonham, 1992).

In the following, we investigate if it is possible to implement and adapt the global controller synthesis approach of Section 3.4 with a modular system modeled by TPN. Some of the possible solutions and their limitations are also discussed. And then, we suggest a decentralized implementation for the solution proposed in Section 3.4. We explain the approach for the output of Algorithm 1. The same procedure is applicable for reachability controller obtained from Algorithm 3.

Remember that Algorithm 1 calculates $Ctrl$, a function of the state classes where the controller is supposed to limit the intervals of some enabled controllable transitions. When the state dependent controller is implemented decentrally as a set of local controllers, then each local controller needs to be state dependent. In order to have the same result as that of Algorithm 1, each local controller should also be aware of the global state. As such, the whole information of each module including local markings and local timing domains should be announced to other modules. This in turn means that, in this case, we cannot have independent local controllers and modules should have intercommunication among them.

A deeper look at Algorithm 1 shows that three levels of control with different levels of dependency and permissiveness can be carried out. The first level which is discussed in detail in Chapter 3 is state dependent and is the most permissive one. The second level is marking dependent. The controller should act upon the global marking of the system. Recall the example of Fig.2.4. We have shown the obtained marking dependent controller in Table 3.1 and Fig.3.7. This level is less permissive but is reasonably applicable. There is always a trade-off between the permissiveness and the implementation cost. The third level is a static controller which is the least permissive. At this level, we calculate a safe interval for each controllable transition as if we are correcting the system before execution. An example of such scenario is to replace the interval of the transition t_1 in the example of Fig.2.4 with $]2, 4]$. As discussed earlier, in the example of Fig.2.4 if t_2 is fired before t_1, there is no need

to limit the behavior of t_1. In case of a static controller, the interval associated with t_1 is replaced with $]2, 4]$ and the permissiveness is sacrificed.

Let us consider a distributed system with multiple modules and a unique property (the fourth configuration). We discuss in the following if each of the above-mentioned levels of control are implementable on a modular system with and without the possibility of intercommunication.

5.3.1 Case of static local controllers

One of the control levels obtained from Algorithm 1 is the static controller which is the least permissive and consequently, the simplest option in terms of implementation. It consists in modifying the intervals associated with the controllable transitions regardless of the system modifications. As mentioned before, the static controller is like correcting the system before its execution. Then considering decentralized static controllers, the local controllers are independent and there is no need to intercommunicate among the modules. The static controller can be implemented as a set of decentralized controllers.

5.3.2 Case of marking dependent local controllers

A marking dependent controller acts according to the global marking of the system. The marking dependent controller acts similarly in the states with the same marking. In fact, the intersection of safe intervals in the states with similar markings are considered to extract the marking dependent controller of the system. Consequently, there is a risk of loss of permissiveness in comparison with the state dependent controller. If the markings are different in each state (i.e. there is not at least two states with the same marking), the state dependent controller is identical with the marking dependent controller. Consider the function $Ctrl$ the state dependent controller obtained from Algorithm 1, we denote the marking dependent controller extracted from $Ctrl$ by $CtrlM$ and investigate its decentralized implementation.

In our first step, we discuss independent local controllers without any intercommunication among them. In this case, each local controller decides according to the local marking of its own module. The above-mentioned marking dependent controller $CtrlM$ is post-processed to extract the local marking dependent controller. For example, suppose a system is composed of two modules: mod_1 and mod_2. Let $CtrlM$ be the obtained marking dependent

controller. For a global marking M and a controllable transition t_c, the set of safe intervals of t_c at marking M is computed as: $CtrlM(M, t_c) = \{int_1, int_2, ..., int_n\}$. A global marking is composed of local markings i.e. $M = (M_{mod_1}, M_{mod_2})$. A post-processing step is needed to extract the local controllers $CtrlM_{mod_i}$ dependent to the local markings, where i stands for the index number associated with each module.

Let us discuss more in detail the implementation of local marking dependent controllers without any intercommunication among them which is less costly because the local controllers do not need to exchange any information. However, there are some limitations. Each local controller considers only its own local markings. Remember the above mentioned modular system composed of mod_1 and mod_2. Suppose two forbidden global markings $M_{f1} = (M_{mod1}, M_{mod2})$ and $M_{f2} = (M'_{mod1}, M_{mod2})$ where M_{mod_1}, M'_{mod_1} are the local markings of mod_1 and M_{mod_2} is the local markings of mod_2. The global marking dependent control function gives two different safe intervals for these markings i.e. $CtrlM(M_{f1})$ and $CtrlM(M_{f2})$. Consequently, we have two different safe intervals for M_{mod2}. In order to have local marking dependent controllers without intercommunication, the intersection of these safe intervals should be considered. It is possible that the intersection of the safe intervals for similar local markings is empty. This in turn means that even if a global marking dependent controller exists, sometimes it is not possible to extract a set of decentralized controllers where each local controller is dependent to the corresponding local markings.

For the same system, even if the set of local controllers dependent to the local markings exists (the intersection of safe intervals for the states with similar markings is not empty), still we risk to obtain a less permissive controller in comparison with the calculated global marking dependent controller. Suppose a forbidden global marking $M_f = (M_{mod_1}, M_{mod_2})$ where M_{mod_1} is the local marking of mod_1 and M_{mod_2} is the local marking of mod_2. On the other hand, suppose a global safe marking $M_s = (M_{mod_1}, M'_{mod_2})$ where M_{mod_1} is the local marking of mod_1 and M'_{mod_2} is the local marking of mod_2. What may happen in decentralized implementation with independent local controllers is that, local controller of mod_1 prevents M_{mod_1} and consequently, M_s is prevented (even if M_s is safe). We provide an example in the following.

Remember the example of the assembly section of a manufacturing line discussed earlier in Section 3.4.8. As described earlier, the assembly section depicted in Fig.3.8 consists of two robotic arms, a conveyor and an assembly tray. Each robotic arm is assigned to bring a part from the corresponding type (A or B) and put it on the conveyor (specified by transitions t_1

Table 5.1 A marking dependent controller for the TPN of Fig.3.8. The chosen scenario forces t_1 to fire before t_2.

Marking	Constraint to be applied on t_1 and/or t_2
$\alpha_0 : A + B + Conv.ON$	$1 \leq t_1 < 2 \wedge 5 < t_2 \leq 6$
$\alpha_1 : Conveyor + B + Conv.ON$	$5 < t_2 \leq 6$

and t_2). The conveyor runs and places the parts on the tray where they are assembled and stored in the boxes (modeled by transitions t_3 and t_4, respectively). The conveyor is always on and moves non-stop. The assembling operation is done on the tray and is uncontrollable. Storing an assembled part is also uncontrollable. The assembly tray should receive two parts from two different types consecutively. It is not allowed to accept more than one part on the conveyor. Then, a controller is needed for the robotic arms to manage the system so as to meet the safety property $AG\ Conveyor < 2$. As discussed earlier in Section 3.4.8, the obtained controller provides two scenarios: either forcing t_1 to fire at $[1, 2[$ while delaying t_2 until $]5, 6]$ or delaying t_1 until $]6, 7]$ while forcing t_2 at $[2, 3[$. Suppose we have chosen the first scenario. The marking dependent controller for this scenario is given in Table 5.1.

Now let us break the system into modules and extract the local controllers dependent to the local markings. The assembly section is composed of three modules. First module, named a consists of the robotic arm which moves parts of type A, second module named b includes the conveyor and the assembling tray and the third one, named c consists of the robotic arm which moves parts of type B. The local controller of module b is always idle and has nothing to do as there is no controllable transition in this module. The local marking dependent controllers of the module a and c are called $CtrlM_a$ and $CtrlM_c$, respectively and are extracted as follows:

- $CtrlM_a$: For marking A force t_1 to fire at $[1, 2[$.
- $CtrlM_c$: For marking B delay t_2 until $]5, 6]$.

With the chosen scenario, t_1 is fired at $[1, 2[$. The local controller of the module B delays the controllable transition t_2 until $]5, 6]$. Suppose that just 2 time units after firing t_1, the transition t_3 is fired in the module b, the system enters in the global state α_3 (refer to Fig.3.9). Based on the calculation of the global controller, there is no need to prevent firing of t_2; whereas, in our current implementation $CtrlM_c$ continues to delay t_2 because it is not aware of what is going on in the other modules.

In our second step, we investigate the implementation of decentralized marking dependent

controllers with the possibility of intercommunication. If the intercommunication delay is not negligible in comparison to the state evolution of the system or if the intercommunication delay is not included in the time intervals associated with the transitions, this delay should be considered in the global model of the system before synthesizing the controller by Algorithm 1.

In practice, mechanical movement and evolutions are usually slower than digital evolutions. For example, suppose a mechanical system controlled by a digital microprocessor system. Intercommunications among local microprocessors take a few milliseconds while discrete evolutions of mechanical equipments cannot be done in less than some seconds. In this case, intercommunication delay is not comparable with the state evolution frequency. This delay is negligible or included in the time intervals associated with the system transitions. In other words, intercommunications are supposed to be fully synchronized. In this case, data exchanges among the modules do not cause any problem though they are costly in terms of implementation.

Now suppose a digital system where intercommunication delay is comparable with the system evolution frequency and cannot be discarded. For example, consider a modular system composed of two modules. Each module has a controllable transition (i.e. t_1 and t_2). The intercommunication delay between two modules is equal with d time unites. Let α be a state class where both t_1 and t_2 are newly enabled. The output of Algorithm 1 for this class is to impose the condition of $|t_1 - t_2| > a$. In its modular implementation, when t_1 becomes newly enabled in the first module, it takes d time unites for the second module to distinguish this event and become aware of. Then, when the second module receives the value of the interval associated with t_1, this value is actually $t_1 + d$ in the first module. Hence, a post processing is required for $Ctrl$ (similarly for $CtrlM$) to reconsider this delay, i.e.:

$$|t_1 - t_2| > a + d. \tag{5.1}$$

Let us look at this problem from another angle. When intercommunication delay is comparable with the state evolution frequency, this actually means that the specification of the global system is changed. Then, if the intercommunication delay is comparable to the state evolution frequency, this delay should be considered in the global model of the system. After each discrete event, an intermediate place, $Sync$ followed by an uncontrollable transition t_{sync} is required. To simplify the model, we can add this delay to the interval associated with an uncontrollable transition but, a controllable transition has to be followed by the set of intermediate $Sync$ place and uncontrollable transition t_{sync} modeling the intercommunication

delay (See Fig.5.3 and Fig.5.4).

One important point in this case is that, during the data transmission, recipient modules are not aware of the real global marking (state) of the system. The local controller of some module may decide upon the new data while the local controller of some other receiver modules decide according to the previous state as the data is not still transfered. One solution is that the controller cannot act once a transmission is established. In fact, $CtrlM$ (similarly for $Ctrl$) cannot act in a class where its marking includes a transmission (i.e. a $Sync$ place is marked). This condition imposes that the controllers cannot make a new decision unless the new data is received and the global state of the system is properly recognized. In case intercommunication delay makes any problem in the control procedure, Algorithm 1 fails to

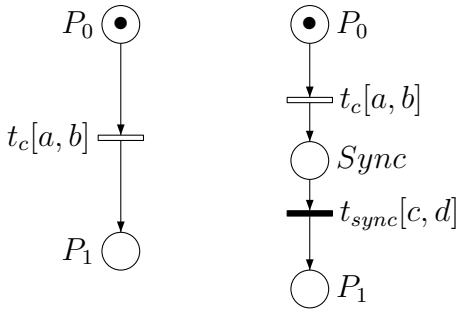


Figure 5.3 A controllable transition, considering synchronization delay.

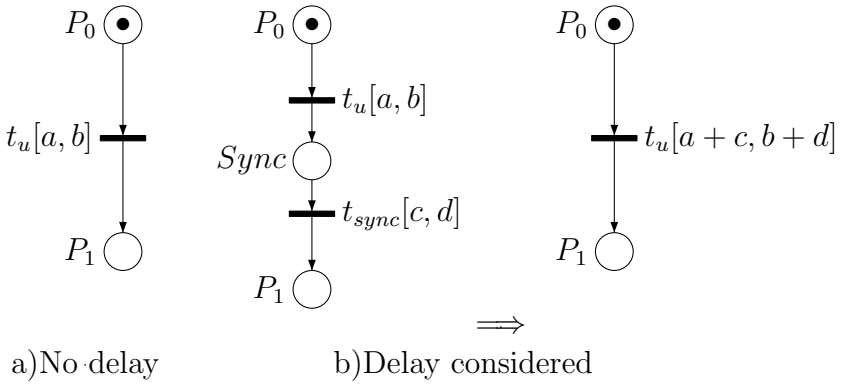


Figure 5.4 An uncontrollable transition, considering synchronization delay.

compute a controller considering the new specification of the system and the supplementary conditions of the Algorithm. In other words, if intercommunication delay is such significant that causes a failure in the control procedure then, Algorithm 1 is not able to find a controller for the model after its new modifications.

Let us investigate how it works in practice. Remember the system controlled by local microprocessors. The modules are connected directly to each other. When one module is willing to announce its new marking/state, the local controller establishes a connection with the other modules. After receiving the corresponding signal (communication request) each module is waiting to receive the new data.

Note that in this configuration each local controller should be able to distinguish that a *Sync* place (in the global model) is marked. Suppose a distributed system where each module has direct connection only with some of the other modules. A message is transfered through some intermediate modules before being received by all of the other modules. For example, in an Internet application a message passes through multiple intermediate ip addresses before the final recipient receive it. In this case, some of the receiver modules cannot distinguish that there is a transmission under execution. The local controllers of such receiver modules might make their decision according to the obsolete information (marking state) because they are aware of neither the actual global state, nor an under process message transmission.

As a conclusion, consider a modular system where all of the modules have direct interconnections among them. The system is controlled by a set of decentralized marking dependent local controllers. The local controllers decide according to the global marking of the system. The local controllers should announce their marking after each new discrete event and all of the modules should update and reconstruct their copy of global marking. As explained earlier, the local controllers cannot make any decision during a transmission, (when a *Sync* place is marked). Note should be taken that, if the modules are not directly connected (each local controller cannot distinguish if a communication is established) the local controllers may work inappropriately and fail to control the system.

5.3.3 Case of state dependent local controllers

Let us investigate if it is possible to implement state dependent controllers as a set of decentralized local controllers. First, consider the case of independent local controllers without any intercommunication. In this case, local controllers should act upon their local states.

Therefore, a post processing step is required to extract local state dependent controllers. On the other hand, in Algorithm 1 the difference between two enabled controllable transitions is considered to compute $Ctrl$ of the appropriate classes. If two controllable transitions are from two different modules, independent decentralized implementation is not feasible. Otherwise, the same as marking dependent controllers, a post-processing step is required to compute local state dependent controllers of each module.

The same as local marking dependent controllers, the intersection of safe intervals for same local states should be considered. This intersection can be empty. This in turn means that, it is possible to have a global state dependent controller while the set of local state dependent controllers do not exist. And also, considering local state dependent controllers without any intercommunication among them can cause the loss of permissiveness.

In the next step, consider the decentralized implementation with the possibility of intercommunication. In order to have a state dependent controller, each local controller should keep track of the global state of the system. The modules should intercommunicate among them to extract the global state (i.e. global marking and timing domain) of the system. Then, each module should announce its new local state after each new discrete event. In this case, whenever a controllable transition becomes newly enabled its corresponding local controller should propagate the timing information of the newly enabled controllable transition to all other modules.

In summary, in a decentralized implementation, having a set of state dependent local controllers is costly because it requires a lot of information exchanges about both new markings and time information. If the intercommunication among modules is comparable with the state evolution frequency of the system, this delay should be considered in the global model of the system. As explained earlier, each transition is followed by a place and an uncontrollable transition to model the intercommunication delay. Note that if the modules are not directly communicating with each other, there is a risk that the decentralized implementation does not follow the actual synthesized controller of Algorithm 1. In fact, if all of the modules are not directly connected to each other and the local controllers cannot distinguish that a communication is established, there is no guarantee that decentralized implementation follows exactly the centralized computed controller.

5.3.4 Decentralized implementation with the possibility of intercommunication

We have discussed different possibilities for decentralized implementation of the output of Algorithm 1. We stated that static controllers and marking dependent controllers can be implemented on modular systems independently without any intercommunication. Although independent local controllers are less costly and easier to be implemented, they may sacrifice the permissiveness of the approach. We concluded that state dependent controller is very costly and less appropriate to be implemented on modular systems as a set of decentralized controllers. In this section, we suggest an algorithm for decentralized implementation of a marking dependent controller with the possibility of intercommunication. In this case, we want to keep track of the global marking of the system in order to save the permissiveness of the solution. Then, at this level, our objective is to have a set of local controllers dependent to the global markings of the system.

We categorize state classes into forbidden, safe and dangerous (those leading to a forbidden state) as in (Abid et Zouari, 2010a). Each modular controller should be aware of the global marking to be able to track the global marking graph of the system. For this purpose, modules will exchange their local markings after each state evolution. Each local controller has a communication submodule for data exchanges and an executer submodule to perform the control procedure obtained from Algorithm 1. In each module, we consider a local variable, called status, that keeps track of the current status of the module (safe or dangerous). This status is determined based on the global marking of the system. As long as the global status is dangerous, local controllers are active. Otherwise, they are idle and have nothing to do. If there is no enabled controllable transition in a module, then its controller is idle but communication threads are always active.

The system is divided into modules by finding the overlapped sub Petri nets. As in (Abid et Zouari, 2010a) and in contrary with (Aydin et Altug, 2009), in our case it is possible to have both overlapped places and transitions. Once a marking is changed (a discrete event happens), the corresponding local controller will announce this modification to all other modules and then, all receivers will reconstruct and extract the global marking. Note that we have one sender at a time and all others are recipient. In other words, each transition firing (and its consecutive marking modification) is propagated only by one local controller while others are receivers. Algorithm 5 formalizes this discussion.

Algorithm 5 Decentralized Implementation of the Centralized Controller Synthesized by the Algorithm 1.

Function *Implementing Local Controller (State Class Graph SCG, Global Dangerous Marking DM)*
status = dangerous;
M=M_0;
while true do
  M=getGlobalMarking(SCG);
  if local event is occurred then
    if status != getCurrentStatus(M,DM) or getCurrentStatus(M,DM)= dangerous then
      Announce the current local marking to all other controllers and get consensus
    end if
  end if
  status=getCurrentStatus(M,DM);
  if status= dangerous and $T_{c(i)} \cap En(M) \neq \emptyset$ then
    Apply the synthesized strategy;
  end if
  Wait until a modification (new local or external marking);
end while

($^*Remark$)
$T_{c(i)}$ **is the set of local controllable transitions of module** i**.**
M **is the current global marking.**
$En(M)$ **is the set of enabled transitions at global marking** M**.**

At the beginning, the status is potentially dangerous in all of the modules; so each local controller initializes its corresponding status to dangerous. Then, local controllers extract the global marking of the system. This way, every module can keep track of the marking of the entire system. At this point, based on the global marking, if the status is not dangerous, each local controller can update its status. This modification (from dangerous to safe) is announced to all other modules. Now, if status is safe, local controllers become idle. They just listen to the new modifications, if any. On the other hand, in the modules with a dangerous state, if there exists an enabled controllable transition, the corresponding local controller is active. In fact, the value of the variable status is particularly interesting in the modules with enabled controllable transitions. A local controller either active or idle, is always listening to the new modifications (internal or external).

If a transition is fired in a module, its corresponding local controller verifies the new global marking and updates the local status, if needed. In the case of a dangerous state or any state modification, the new marking is announced to all others. This way considering, all of the modules are able to keep track of the global state class graph.

In summary, local controllers intercommunicate. When a discrete event happens in a module, if its local status is dangerous or is changed from dangerous to safe, then its local controller announces the new marking. Otherwise, local controllers are listening to the modification of other modules. Algorithm 5 formalizes this discussion.

We have shown in Algorithm 5 that, if Algorithm 1 is able to give a centralized marking dependent controller (a controller exists), then its decentralized implementation is also feasible. If intercommunication delay is not negligible or included in the intervals associated with the transitions, this delay should be considered in the global model of the system. Note that in this case Algorithm 1 should consider an extra condition. The controller cannot act during a data transmission (i.e. in the classes where a $Sync$ place is marked). As discussed earlier, this approach is not appropriate if each module cannot distinguish whether or not data is being transfered (like message transmission in web applications).

In contrary with (Aydin et Altug, 2009) our approach does not add any place or transition to make a system controlled. In our approach, the behavior of the controller is implicitly added to the same model. We are dealing with TPNs and the controller modifies time intervals of controllable transitions to guarantee the satisfaction of properties (if the controller exists).

5.3.5 Illustrative examples

This section provides some examples of decentralized implementation of the controller synthesized by Algorithm 1 in two categories. In the first category, we assume that intercommunication delay is negligible in comparison to the state evolution of the system or is included in the intervals associated with the transitions. In the second category, we take into account the intercommunication delay among modules and follow the procedure to achieve a decentralized implementation. Note that in the examples of the second category, the modules have direct connection among them and distinguish the communication process as soon as it is established.

Case of local controllers with fully-synchronized intercommunications

Let's consider the same example of Fig.2.4 for the case of fully synchronized modules. As of Section 3.4, the controller shall act in the state class α_0 to prevent t_1 from firing until $]2, 4]$ unless t_2 is fired. The corresponding marking dependent controller is reported in Table 3.1. First, let us break the system into modules by finding the corresponding overlapped Petri nets. The obtained modular system is presented at Fig.5.5 and two transitions t_{4-1} and t_{4-2} are overlapped. Considering the dependency between modules and overlapped transitions, the state class graph of each module cannot be processed individually. Intercommunication between modules is inevitable to have the same global state class graph of Fig.2.5 and the same result. In this example, there is just one controllable transition.

The state class graph of Fig.2.5 determines that α_4 and α_6 are the forbidden states; α_1, α_3 and α_0 may lead to these forbidden states and are dangerous, while α_2 and α_5 are safe.

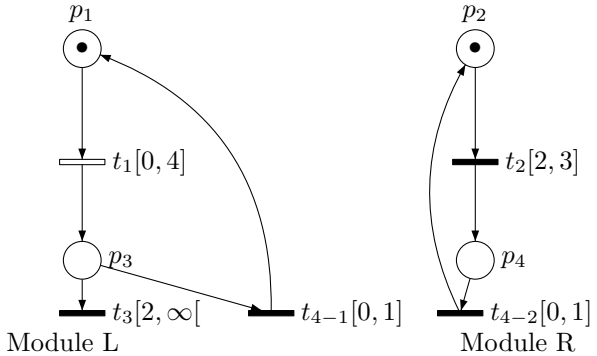


Figure 5.5 The example of Fig.2.4 in modules.

Tables 5.2 and 5.3 represent the trace of Algorithm 5 on Fig.5.5. The notions $T_{c(L)}$ and $T_{c(R)}$ stand for the set of local controllable transitions in the modules L and R respectively. Note that state change from α_2 to α_5 is not announced to Module L of Fig.5.5 as the status is always safe and no announcement is required.

Case of local controllers with non-synchronized intercommunications

In the previous section, we have shown an example with fully synchronized local controllers where intercommunication delay is negligible in comparison to the state evolution of the system or this delay is included in the intervals associated with the transitions. In this section, we take into account the communication delay among the modules. In other words, we assume that the intercommunication delay is comparable with the frequency of discrete state evolutions. In the following, we study two examples. In the first one intercommunication delay is problematic and prevents the control procedure while in the second one the

Table 5.2 Trace of Algorithm 5 on Fig.5.5 (Module L).

state	Action
α_0	Extract the initial marking; Status = dangerous. $T_{c(L)} \cap En(M) = t_1$; The controller, delays t_1 and listens to the modifications.
α_1	t_1 is fired; Status = dangerous. Marking announcement; $T_{c(L)} \cap En(M) = \emptyset$.
α_3	t_2 is fired in module R; Status = dangerous. $T_{c(L)} \cap En(M) = \emptyset$.
α_2	t_2 is fired in module R; Status = safe.
α_5	t_1 is fired; Status = safe. Unchanged safe status; no announcement.

Table 5.3 Trace of Algorithm 5 on Fig.5.5 (Module R).

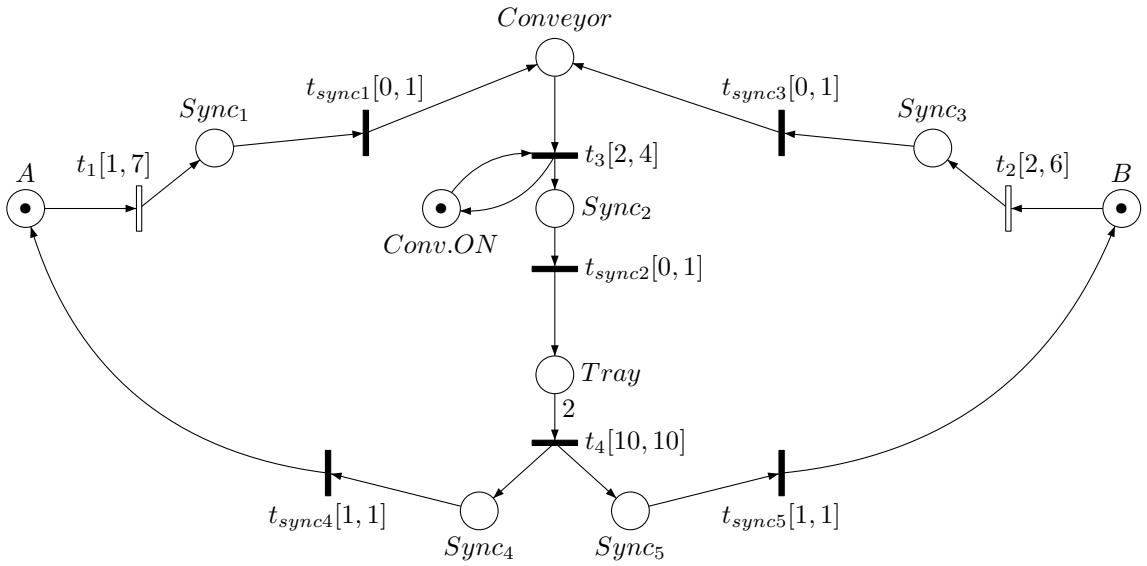
state	Action
α_0	Extract the initial marking. Status = dangerous; $T_{c(R)} = \emptyset$.
α_1	t_1 is fired in module L; Status = dangerous. $T_{c(R)} = \emptyset$.
α_3	t_2 is fired; Status = dangerous. Marking announcement.
α_2	t_2 is fired; Status = safe. Marking announcement.

controller succeeds to control the system. We will see that Algorithm 1 shows the failure in the first example.

Example 1:

We consider the same example of assembly section of a manufacturing line depicted in Fig.3.8 and discussed in Section 3.4.8. As mentioned earlier, we consider intercommunication among modules in the global model and we revise our modeling accordingly. In this approach local transitions resulting a data transmission are followed by a set of place ($Sync$) and a transition (t_{sync}). The global model should provide a solution for these exchanges and their corresponding delays.

The assembly section is composed of three modules. First module, named a consists of the robotic arm which moves parts of type A, second module named b includes the conveyor and the assembling tray and the third one, named c consists of the robotic arm which moves parts of type B. The events exchanged between modules are:



a.Robotic arm A b.Putting down and Assembling c.Robotic arm B

Figure 5.6 The TPN model of the assembly section of a manufacturing line considering intercommunication delay among the modules.

1. When a piece of type A is placed on the conveyor, a token from the module a is sent to the module b. Consequently, a message is sent to the module c to announce this event.
2. When a piece of type B is placed on the conveyor, a token from the module c is sent to the module b. Consequently, a message is sent to the module a to announce this event.
3. When Conveyor delivers a part to Tray, a message is sent to the modules a and c to announce this event and the consequent change of status.
4. Two parts are completely assembled on Tray and the robotic arms are again ready. Two tokens from the module b are sent to the modules a and c (one for each).

Considering intercommunication delay, the specification of the system is modified and the model of Fig.3.8 is no more valid. We should revise the global model to reconsider these new timing constraints. In this example, intercommunication delay is considered one time unit where applicable. Note that once a part of type A or B is placed on the conveyor, the event is detected simultaneously in the module b (i.e. delay is zero). On the other hand, transferring a message from the module a to the module c and vice versa for announcing new local markings or state evolutions will take a while (i.e. delay is one). Similarly, when a part is delivered from Conveyor to Tray, this event is distinguished simultaneously in the module b, but takes a while to be recognized by the other modules. Therefore, for these cases intercommunication delay is equal with $[0, 1]$. But, when assembling procedure is finished, there is no internal action. A message is sent to the modules a and c then intercommunication delay in this case is $[1, 1]$.

The model of Fig.5.6 represents this system with its new specifications. Two places $Sync_1$ and $Sync_3$ model transmissions among the modules a, b and c. Two transitions t_{sync1} and t_{sync3} represent the corresponding intercommunication delay which is considered equal with $[0, 1]$ time units. The messages being transfered between the modules contain local markings. Firing of t_3 in the module b is announced to the modules a and c through the place $Sync_2$ and the transition t_{sync2}. Once two parts are assembled on Tray and the system is reinitialized, a token is transfered from module b to each of the other modules within a delay. Two intermediate places $Sync_4$, $Sync_5$ and two transitions t_{sync4}, t_{sync5} are added to model this behavior.

The state class graph of the model comparing to the state class graph of Fig.3.8 becomes larger and consists of 28 state classes. The abstracted state space after abstraction by inclusion consists of 24 state classes. We apply the algorithm on the compact abstracted state graph given at Fig.5.7 and Table 5.4. Considering the information of classes provided in

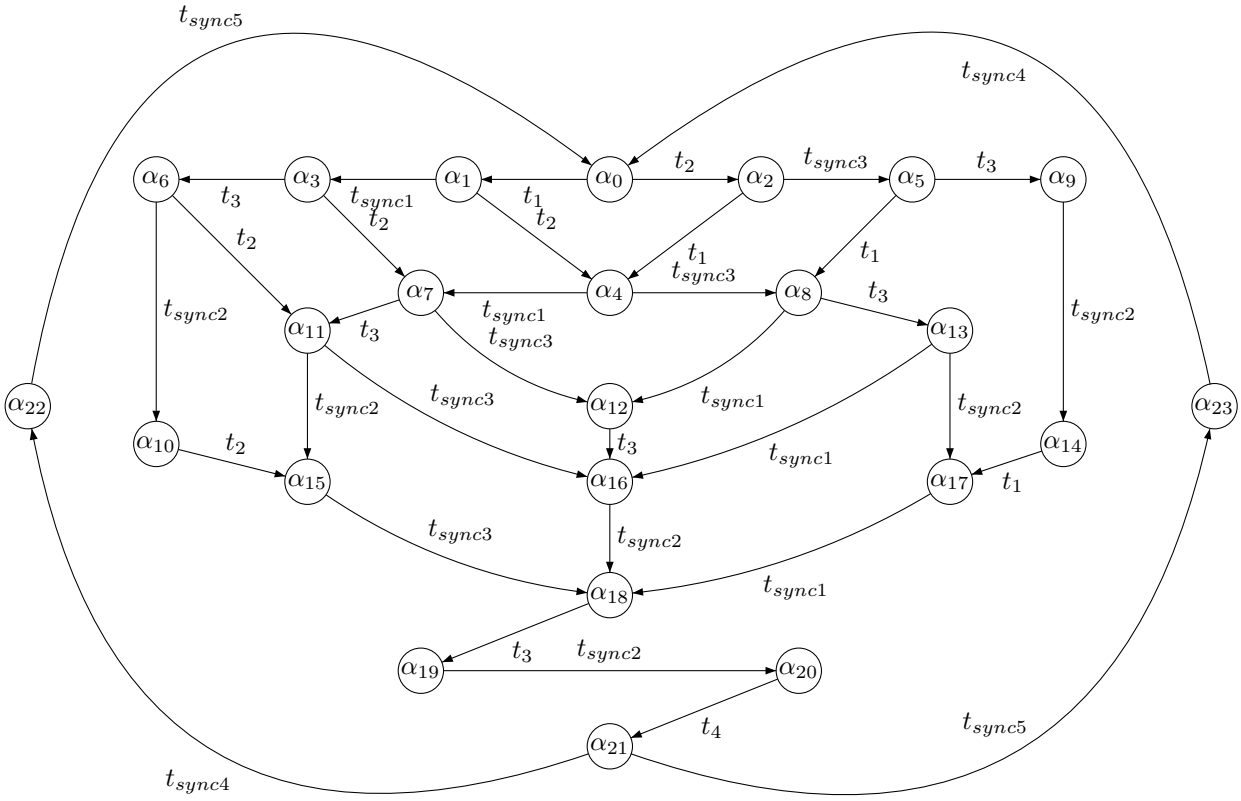


Figure 5.7 The state class graph of the model depicted in Fig.5.6.

Table 5.4, the set of forbidden state classes contains: $\{\alpha_{12}\}$.

Let us trace Algorithm 1 on Fig.5.7. The algorithm starts from α_0 and continues recursively exploring the graph through α_2, α_5, α_8 and finally reaches to α_{12} with forbidden markings. The algorithm returns back to α_8 with $\{t_{sync1}\}$ and to α_5 with $\{t_1 t_{sync1}\}$. The other successors available from α_5 are safe. There is an enabled controllable transition in α_5 but it is not newly enabled and the controller should have started earlier. The algorithm returns back to α_2 with $\{t_{sync3} t_1 t_{sync1}\}$. The other path available from α_2 leads also to a forbidden state and is computed similarly. The algorithm returns back to α_0 with $\{t_2 t_1 t_{sync3} t_{sync1}, t_2 t_{sync3} t_1 t_{sync1}\}$. In addition, the other path available from α_0 through α_1 leads also to the forbidden state α_{12}. Finally, the set of bad paths at α_0 includes: $\{t_1 t_{sync1} t_2 t_{sync3}, t_1 t_2 t_{sync1} t_{sync3}, t_2 t_1 t_{sync3} t_{sync1}, t_2 t_{sync3} t_1 t_{sync1}\}$. Then, the condition $|t_1 - t_2| > 5$ should be imposed in α_0 to obtain a safe system. Considering the timing specification of the current system, this condition cannot be satisfied. Thus, due to the intercommunication delay in this system, the controller fails to control the system and Algorithm

Table 5.4 State classes of Fig.5.7 for $t_3 = [2, 4]$.

$\alpha_0 : A + B + Conv.ON$	$1 \leq t_1 \leq 7 \wedge 2 \leq t_2 \leq 6$
$\alpha_1 : B + Sync_1 + Conv.ON$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_2 \leq 5$
$\alpha_2 : A + Sync_3 + Conv.ON$	$0 \leq t_1 \leq 5 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_3 : Conveyor + Conv.ON + B$	$2 \leq t_3 \leq 4 \wedge 0 \leq t_2 \leq 5$
$\alpha_4 : Sync_1 + Conv.ON + Sync_3$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_5 : Conveyor + Conv.ON + A$	$2 \leq t_3 \leq 4 \wedge 0 \leq t_1 \leq 5$
$\alpha_6 : B + Sync_2 + Conv.ON$	$0 \leq t_{sync2} \leq 1 \wedge 0 \leq t_2 \leq 3$
$\alpha_7 : Conveyor + Conv.ON + Sync_3$	$0 \leq t_3 \leq 4 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_8 : Conveyor + Conv.ON + Sync_1$	$0 \leq t_3 \leq 4 \wedge 0 \leq t_{sync1} \leq 1$
$\alpha_9 : A + Conv.ON + Sync_2$	$0 \leq t_1 \leq 3 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{10} : Tray + Conv.ON + B$	$0 \leq t_2 \leq 3$
$\alpha_{11} : Sync_2 + Conv.ON + Sync_3$	$0 \leq t_{sync2} \leq 1 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_{12} : 2Conveyor + Conv.ON$	$0 \leq t_3 \leq 4$
$\alpha_{13} : Sync_1 + Conv.ON + Sync_2$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{14} : A + Conv.ON + Tray$	$0 \leq t_1 \leq 3$
$\alpha_{15} : Tray + Conv.ON + Sync_3$	$0 \leq t_{sync3} \leq 1$
$\alpha_{16} : Conveyor + Conv.ON + Sync_2$	$2 \leq t_3 \leq 4 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{17} : Sync_1 + Conv.ON + Tray$	$0 \leq t_{sync1} \leq 1$
$\alpha_{18} : Conveyor + Conv.ON + Tray$	$1 \leq t_3 \leq 4$
$\alpha_{19} : Sync_2 + Conv.ON + Tray$	$0 \leq t_{sync2} \leq 1$
$\alpha_{20} : Conv.ON + 2Tray$	$10 \leq t_4 \leq 10$
$\alpha_{21} : Sync_4 + Conv.ON + Sync_5$	$0 \leq t_{sync4} \leq 1 \wedge 0 \leq t_{sync5} \leq 1$
$\alpha_{22} : A + Conv.ON + Sync_5$	$1 \leq t_1 \leq 7 \wedge 0 \leq t_{sync5} \leq 0$
$\alpha_{23} : B + Sync_4 + Conv.ON$	$0 \leq t_{sync4} \leq 0 \wedge 0 \leq t_2 \leq 6$

1 also shows this failure.

Example 2:

Now let us follow another example where in presence of the intercommunication delay, Algorithm 1 is able to give a controller. To better clarify the approach, we consider the same example of the assembly section but this time Conveyor needs $[2, 3]$ time units to deliver a part to Tray. In other terms, the interval associated to t_3 is modified from $[2, 4]$ to $[2, 3]$ time units. The state class graph remains the same but the classes are different. The state classes are given in Table 5.5. The same as before, the set of bad paths available from α_0 contains $\{t_1 t_{sync1} t_2 t_{sync3}, t_1 t_2 t_{sync_1} t_{sync3}, t_2 t_1 t_{sync3} t_{sync1}, t_2 t_{sync3} t_1 t_{sync1}\}$. The condition $|t_1 - t_2| > 4$

should be imposed in α_0 in order to have a safe system. Note that if the model with $t_3 = [2, 3]$ is synthesized regardless of intercommunication delay, the condition would be $|t_1 - t_2| > 3$. The results here confirm our previous discussions for the equation 5.1.

We insist here that although there is an enabled controllable transition in the state classes α_1 and α_2, the controller cannot make a new decision because of active data transmission in these states. After synthesizing the model (considering $t_3 = [2, 3]$), the final result is:

- $Ctrl(\alpha_0) = \{(t_1, t_2, \{[-5, -4[,]4, 5]\})\}$.
- $Ctrl(\alpha_3) = \{(t_2, t_0,]3, 5])\}$.
- $Ctrl(\alpha_5) = \{(t_1, t_0,]3, 5])\}$.

Now that delay of intercommunication is considered in the model and the computed controller can guarantee the satisfaction of the given properties, the modular implementation of

Table 5.5 State classes of Fig.5.7 for $t_3 = [2, 3]$.

$\alpha_0 : A + B + Conv.ON$	$1 \leq t_1 \leq 7 \wedge 2 \leq t_2 \leq 6$
$\alpha_1 : B + Sync_1 + Conv.ON$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_2 \leq 5$
$\alpha_2 : A + Sync_3 + Conv.ON$	$0 \leq t_1 \leq 5 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_3 : Conveyor + Conv.ON + B$	$2 \leq t_3 \leq 3 \wedge 0 \leq t_2 \leq 5$
$\alpha_4 : Sync_1 + Conv.ON + Sync_3$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_5 : Conveyor + Conv.ON + A$	$2 \leq t_3 \leq 3 \wedge 0 \leq t_1 \leq 5$
$\alpha_6 : B + Sync_2 + Conv.ON$	$0 \leq t_{sync2} \leq 1 \wedge 0 \leq t_2 \leq 3$
$\alpha_7 : Conveyor + Conv.ON + Sync_3$	$0 \leq t_3 \leq 3 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_8 : Conveyor + Conv.ON + Sync_1$	$0 \leq t_3 \leq 3 \wedge 0 \leq t_{sync1} \leq 1$
$\alpha_9 : A + Conv.ON + Sync_2$	$0 \leq t_1 \leq 3 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{10} : Tray + Conv.ON + B$	$0 \leq t_2 \leq 3$
$\alpha_{11} : Sync_2 + Conv.ON + Sync_3$	$0 \leq t_{sync2} \leq 1 \wedge 0 \leq t_{sync3} \leq 1$
$\alpha_{12} : 2Conveyor + Conv.ON$	$0 \leq t_3 \leq 3$
$\alpha_{13} : Sync_1 + Conv.ON + Sync_2$	$0 \leq t_{sync1} \leq 1 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{14} : A + Conv.ON + Tray$	$0 \leq t_1 \leq 3$
$\alpha_{15} : Tray + Conv.ON + Sync_3$	$0 \leq t_{sync3} \leq 1$
$\alpha_{16} : Conveyor + Conv.ON + Sync_2$	$2 \leq t_3 \leq 3 \wedge 0 \leq t_{sync2} \leq 1$
$\alpha_{17} : Sync_1 + Conv.ON + Tray$	$0 \leq t_{sync1} \leq 1$
$\alpha_{18} : Conveyor + Conv.ON + Tray$	$1 \leq t_3 \leq 3$
$\alpha_{19} : Sync_2 + Conv.ON + Tray$	$0 \leq t_{sync2} \leq 1$
$\alpha_{20} : Conv.ON + 2Tray$	$10 \leq t_4 \leq 10$
$\alpha_{21} : Sync_4 + Conv.ON + Sync_5$	$0 \leq t_{sync4} \leq 1 \wedge 0 \leq t_{sync5} \leq 1$
$\alpha_{22} : A + Conv.ON + Sync_5$	$1 \leq t_1 \leq 7 \wedge 0 \leq t_{sync5} \leq 0$
$\alpha_{23} : B + Sync_4 + Conv.ON$	$0 \leq t_{sync4} \leq 0 \wedge 0 \leq t_2 \leq 6$

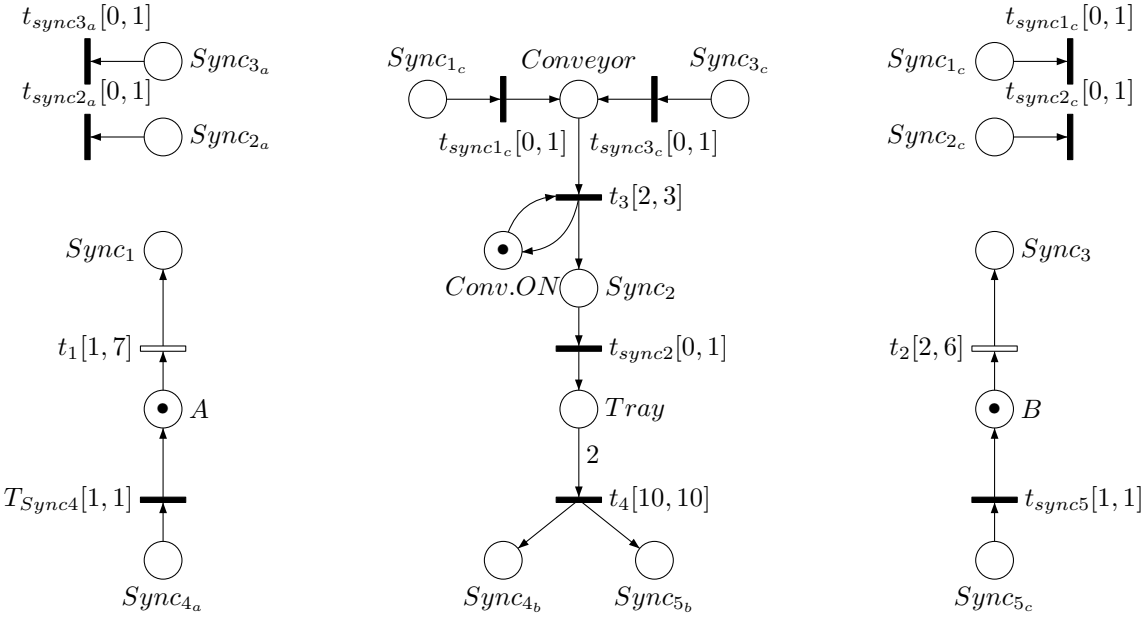
the controller in presence of the intercommunication delay will work as well.

Now, it's time to break the system into modules. The synchronization places and transitions are overlapped which include the places $Sync_1$ to $Sync_5$ and the transitions t_{sync1} to t_{sync5}. These places and transitions model data transmission with their corresponding delays. The modular system is given in Fig.5.8. Each overlapped place or transition is suffixed with the name of its corresponding module (e.g. $Sync_{3_a}$). The places $Sync_1$, $Sync_2$ and $Sync_3$ are overlapped in all of the three modules to show the direct connections among the modules. As mentioned earlier, if a module cannot distinguish that a transmission is in progress (like web applications message passing) the controller will not work properly.

Applying Algorithm 1 on the state class graph of the entire system shows that there are two options to safely control this system: either force t_1 to fire at $[1, 2[$ while delaying t_2 until $]5, 6]$ or, force t_2 to fire at $[2, 3[$ while delaying t_1 until $]6, 7]$. Suppose we have chosen the first option, forcing t_1 in $[1, 2[$ while delaying t_2 until $]5, 6]$. Consider Fig.5.8: first, all the modules are in α_0. The status is dangerous and local controllers of a and c are active. There is no controllable transition in the module b and then, its local controller is idle. However, communication submodules are always active everywhere and listen to the new modifications. The local controller of c is active and delays t_2. Meanwhile, t_1 is fired by the local controller of a. The part is sensed on the conveyor of module b. The new marking is sent to the module c. The status is yet dangerous in all of the modules. From now on, there is no enabled controllable transition in the module a and its local controller is idle while the local controller of c is active and delays t_2. Then, t_3 is fired in the module b and its local marking is announced to the others and the status becomes safe in all of the modules. The local controller of the module c cannot make a new decision until the data transmission is terminated. Afterwards, the status becomes safe in all of the modules. Consequently, there is no need to delay t_2 anymore and the local controller of the module c also becomes idle.

As explained earlier, the number of data exchanges can be optimized in practice. There is no need to announce change of state when both previous and new states are safe. For example, suppose t_1 is fired in the module a. When t_3 is fired in the module b the status becomes safe. Then, it is not necessary to announce firing of t_2 (in the module c) to the module a. Once the parts are assembled on Tray and the system is reinitialized, the status becomes dangerous again and then, this modification should be announced to others. Tables 5.6, 5.7 and 5.8 show this trace after this optimization. Remember that the intercommunication delay is considered $[0, 1]$ in some cases and then, this optimization does not violate the

state class graph. Bypassing a data transmission is as if it takes 0 time units.



a.Robotic arm A b.Putting down and Assembling c.Robotic arm B

Figure 5.8 The modular TPN model of the system depicted in Fig.5.6.

Table 5.6 Trace of Algorithm 5 on Fig.5.8 (block a).

state	Action
α_0	Extract the initial marking; Status = dangerous; $T_{c(a)} \cap En(M) = t_1$. The controller forces t_1 to fire at $[1,2[$(based on the chosen scenario)
α_1	t_1 is fired; Status = dangerous. $Sync_1$ is marked to announce the marking; $T_{c(a)} \cap En(M) = \emptyset$.
α_3	t_{sync1} is fired; recalculate the global marking; Status = dangerous.
α_6	$Sync_2$ is marked by module b; Status = dangerous.
α_{10}	t_{sync2} is fired; recalculate the global marking; Status = safe.
α_{21}	$Sync_4$ and $Sync_5$ are marked; Status = safe.
α_{22}	t_{sync4} is fired; recalculate the global marking; Status = dangerous.

The procedure is similar if we choose to delay t_1 until $]6,7]$ while forcing t_2 to fire at $[2,3[$.

The important result concluded here is that, considering intercommunication delay among modules changes the model of the global system, not the typical procedure. The typical pro-

Table 5.7 Trace of Algorithm 5 on Fig.5.8 (block b).

State	Action
α_0	Extract the initial marking; Status = dangerous; $T_{c(b)} \cap En(M) = \emptyset$. The controller is idle and listens to the modifications.
α_1	t_1 is fired in module a; $Sync_1$ is marked; Status = dangerous.
α_3	t_{sync1} is fired; recalculate the global marking; Status = dangerous.
α_6	t_3 is fired; $Sync_2$ is marked to announce the new marking.
α_{10}	t_{sync2} is fired; recalculate the global marking; Status = safe.
α_{15}	t_2 is fired in the module c; $Sync_3$ is marked; Status = safe.
α_{18}	t_{sync3} is fired; recalculate the global marking; Status = safe.
α_{19}	t_3 is fired and $Sync_2$ is marked; Status = safe.
α_{20}	t_{sync2} is fired; recalculate the global marking; Status = safe.
α_{21}	t_4 is fired; Status = safe; $Sync_4$ and $Sync_5$ are marked.
α_{22}	t_{sync4} is fired; recalculate the global marking; Status = dangerous.

Table 5.8 Trace of Algorithm 5 on Fig.5.8 (block c).

state	Action
α_0	Extract the initial marking; Status = dangerous; $T_{c(c)} \cap En(M) = t_2$. The controller delays t_2(based on the chosen scenario).
α_1	t_1 is fired in the module a; $Sync_1$ is marked. Status = dangerous; $T_{c(c)} \cap En(M) = t_2$.
α_3	t_{sync1} is fired, recalculate the global marking. Status = dangerous; $T_{c(c)} \cap En(M) = t_2$; the controller delays t_2.
α_6	$Sync_2$ is marked as t_3 is fired in the module b Status = dangerous; $T_{c(c)} \cap En(M) = t_2$.
α_{10}	t_{sync2} is fired , recalculate the global marking Status = safe; $T_{c(c)} \cap En(M) = t_2$.
α_{15}	t_2 is fired; status= safe; $Sync_3$ is marked. $T_{c(c)} \cap En(M) = \emptyset$; No announcement to the module a.
α_{18}	t_{sync3} is fired; recalculate the global marking; Status= safe.
α_{21}	t_4 is fired in module b; $Sync_5$ is marked; Status=safe.
α_{23}	t_{sync5} is fired; recalculate the global marking; Status = dangerous.

cedure is always making a model of the system, applying the algorithms of Section 3.4. If the global controller exists, we can apply Algorithm 5 to implement the decentralized controller. Message passing between modules will take a while which is already considered in the computations. If our decentralized controllers are marking dependent, only information of new local markings are exchanged (i.e. tokens representing messages include local markings). If the decentralized controllers are state dependent, information of newly enabled transitions are

also propagated which is more costly (i.e. tokens representing messages include local markings and timing information of controllable transitions). The state dependent controllers in decentralized implementation are not recommended.

5.4 Conclusion

In this chapter, we have studied how the forward on-the-fly approach presented in Section 3.4 for controller synthesis of TPN is adapted with distributed and modular systems. We have shown how to implement this approach on large-scale systems consisting of different components. The obtained decentralized controller can be state dependent, marking dependent or static.

In this research, we have investigated decentralized independent local controllers and local controllers with the possibility of intercommunications. In case local controllers have non-synchronized intercommunications among them, the intercommunication delay should be considered in the global model of the system and the model with its new specification is synthesized with Algorithm 1.

When the intercommunication delay is not negligible, the model should correspond exactly to the actual specification of the system. The intercommunication delay should be considered in the model before being synthesized. If the intercommunication delay is causing any problem then, Algorithm 1 fails to find the controller. If it succeeds to calculate a controller considering intercommunication delay, then it will work also in its decentralized implementation.

We have also shown that a controller cannot make a new decision during a data transmission process. Each module should be able to distinguish that a communication is established otherwise this implementation may fail to work properly. In fact, if the configuration of the system is such that modules do not have direct connections (like a web application where a message passes through intermediate modules) then, this approach is less appropriate.

CHAPTER 6

Controller Synthesis with Stopwatch

6.1 Introduction to timed models associated with stopwatch

In some applications such as scheduling, it is necessary to suspend a task and retrieve it again from where it was suspended. The execution time does not include the time spent in suspension. Once a task is retrieved, the corresponding clock continues to progress from where is was stopped. This behavior cannot be modeled with the ordinary timed automata and time Petri nets. In the ordinary timed automata and time Petri nets all of the enabled clocks progress with a unified rate. It is possible to reinitialize a clock but it cannot be stopped and retrieved again.

Stopwatch is an extension of timed models to facilitate modeling of interruption and resumption of a job. Once an interrupt happens a task is suspended. Later, it is retrieved and continues from where it was interrupted. During the interruption, the clock of interrupted task is stopped while other clocks progress normally. The idea of stopwatch has been discussed and extended to Timed automata (TA) as well as Time Petri Nets (TPN) and some types of TA and TPN associated with stopwatch are already introduced (Allahham et Alla, 2007, 2008; Cassez et Larsen, 2000; Roux et Lime, 2004). In stopwatch, we may have some states whose clocks keep progressing, while in some other state the clock is stopped and keeps the value it had before being stopped as if the clock has a memory and after being retrieved it continues with its previous value.

In a multitasking real time system with interruptible tasks and shared resources, a suitable scheduler is necessary to manage the resources and prevent blocking and deadlock. The scheduler guarantees the well functionality of the system in terms of respecting deadline, priority and similar constraints. Each task cannot start execution until the required resources are available. A task releases the occupied resources after finishing its execution.

In multitasking systems, tasks are categorized to periodic, aperiodic and sporadic (Isovic et Fohler, 2000):

– **Periodic task**: These tasks arrive in regular intervals. Worst case and best case execution times are of the specifications defined for periodic tasks. Deadline is the

other characteristic defined for these tasks with respect to their worst case execution time. Deadline in periodic tasks can be either soft or hard. A critical deadline is called hard deadline whereas meeting a soft deadline is not critical.
- **Aperiodic task**: Tasks with irregular inter arrivals are called aperiodic. Aperiodic tasks are usually associated with soft deadlines.
- **Sporadic task**: Aperiodic tasks with a minimum inter arrival are called sporadic. Sporadic tasks are associated with hard deadlines.

A periodic task may have three states (Altisen *et al.*, 2000):
- **Waiting**: Once a periodic task arrives, it should wait for required resources.
- **Execution**: When all required resources are available, the task starts its execution and spends its execution time in this state.
- **Passive**: After being executed, the task releases the required resources and waits until its next arrival. The time between two consecutive task arrivals is equal with the given period.

In a multitasking system with periodic tasks, the scheduler should manage the shared resources in a way that the predefined period for the tasks are respected. On the other hand, some of the tasks may be associated with a deadline or priority. In that case, the scheduler should suspend the execution of a task with lower priority and let the higher priority tasks to use the resources. Once the tasks with higher priority are executed, the other tasks with lower priorities continue their execution. This behavior is called preemptive scheduling.

In the field of controller synthesis, a scheduler is seen as a controller (Altisen *et al.*, 1999, 2000). The scheduler should manage the shared resources and starts execution of each task in a way that the timing constraints, deadlines and priorities are respected. Arrival of each task is due to its period and then is uncontrollable. Execution time is also predefined and then is uncontrollable. The only controllable action is the starting of an execution. In preemptive scheduling, a task can be suspended during its execution and leave the resources to some other task with a higher priority. The solution discussed in Chapter 3 in its original form cannot be used for preemptive scheduler synthesis.

One solution is to model the system with time Petri nets associated with stopwatch. In this context, each task execution is modeled by a transition equipped with stopwatch and then the corresponding controller is synthesized to manage suspension and resumption of each task such that the corresponding timing constraints are respected. As we will see in

this chapter, one problem with this solution is that calculating the state space graph of time Petri nets associated with stopwatch requires some over approximation.

The other problem is that time Petri nets associated with stopwatch do not necessarily preserve the boundedness property. Let $\mathcal{N}_1$ be a time Petri net and $\mathcal{N}_2$ be a time Petri net associated with stopwatch, the following relation holds:

$$\mathcal{N}_1 \text{ is bounded } \Leftrightarrow \text{SCG of } \mathcal{N}_1 \text{ is finite.}$$

This condition is not always true for a time Petri net associated with stopwatch. In other terms:

$$\mathcal{N}_2 \text{ is bounded } \not\Rightarrow \text{ SCG of } \mathcal{N}_2 \text{ is finite.}$$

Indeed, if the number of reachable markings in a time Petri net associated with stopwatch is finite, the number of state classes is not necessarily finite (Roux et Lime, 2004). These limitations hinder the controller synthesis of time Petri nets associated with stopwatch.

In this thesis, we suggest an alternative solution. We propose to synthesize a time Petri net without stopwatch where the transitions corresponding to the task executions are considered controllable. We extract the controller (i.e. scheduling strategy) and then add the stopwatch to model the controlled system (the system and its scheduler). In fact, we suggest to implement the synthesized controller by means of stopwatch (Heidari et Boucheneb, 2012a).

In the following, we discuss how the algorithm suggested in Section 3.4 is used for scheduling purposes to synthesize a controller for interruptible tasks. Algorithm 1 is permissive and the computed controller is restricting time intervals making the system to satisfy the given properties. In this chapter, we suggest to suspend a task during its bad subinterval. This approach is useful for preemptive scheduling purposes where safety properties correspond to meeting deadlines, priorities, preventing deadlock for shared resources and etc.

In this chapter, we use a synthesized controller (output of the algorithm of Section 3.4 in particular), and we show how to control the system using stopwatch. In order to implement the controller by means of stopwatch, we assume that it is possible to associate each controllable transition with a stopwatch so as to suspend or resume it whenever needed.

The rest of this chapter is organized as follows: Section 6.2 presents a literature review on different stopwatch Petri nets. Section 6.3 synthesizes a controller using stopwatch. Section

6.4 presents an example of a multitasking system. Finally, Section 6.5 gives the conclusion and future work.

6.2 Literature review

The idea of suspension and resumption of a task in a system, is modeled in different ways in timed automata and time Petri nets (Allahham et Alla, 2008; Cassez et Larsen, 2000; Roux et Lime, 2004). In (Cassez et Larsen, 2000), the authors have introduced stopwatch automata (SWA) as a sub-class of timed linear hybrid automata. In stopwatch automata, an additional binary variable is defined to show the rate of time progression. The clocks may have two velocities (time derivation): zero or one. Zero signifies stopped while one signifies normal progress. If clocks are running, they progress with a global rate, identical to all non-stopped clocks of the model. The authors have shown that stopwatch automata is as expressive as timed languages.

In (Roux et Lime, 2004), the authors have introduced inhibitor hyperarcs to interrupt an enabled transition. Once a place p connected to a transition t via an inhibitor arc is marked, the transition t is suspended and stops firing. In other words, once an inhibitor arc is enabled, the corresponding transition is interrupted. They have called these nets IHTPN (Time Petri Nets with Inhibitor Hyperarcs). Fig.6.1 represents a simple example of time Petri nets with an inhibitor arc. In this figure, t_1 is enabled. As soon as p_2 is marked, firing t_1 is suspended. Note that an inhibitor hyperarc is not graphically presented by a classical arrow; instead, it is shown by an empty circle at the extreme of the edge.

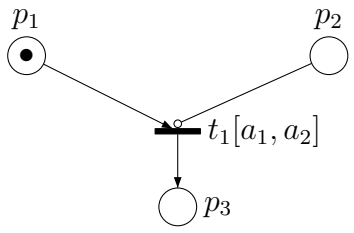


Figure 6.1 A simple example of time Petri nets with inhibitor hyperarc. t_1 is active if p_2 is not marked, otherwise it is suspended.

Another stopwatch model comes in (Allahham et Alla, 2007, 2008). Post and Pre-initialized stopwatch Petri nets (SWPN)(Allahham et Alla, 2007, 2008) also model interruption and retrieving of tasks. In such stopwatch Petri nets, transitions are partitioned into

two disjoint subclasses $T = T_{int} \cup T_{no-int}$ where T_{int} is the set of interruptible transitions and T_{no-int} is the set of non-interruptible transitions. Stopwatches can suspend enabled interruptible transitions. For each transition t_i, there is a function $v(t_i)$ representing the value of its associated stopwatch. If $t_i \in T_{int}$, $v(t_i)$ signifies the time elapsed since t_i was first enabled, whereas if $t_i \in T_{no-int}$, $v(t_i)$ represents the time elapsed since t_i was last enabled. A transition is called $firable$ if it is enabled ($M > Pre(t_i)$) and $\downarrow Is(t_i) \leq v(t_i) \leq \uparrow Is(t_i)$.

In contrary with IHTPN where a marked place suspended an enabled transition, in this model stopwatch consumes some tokens to disable an enabled transition. It is implemented by means of some extra places/transitions. A stopwatch transition is in conflict with an interruptible transition. Thus, consuming a token can suspend an interruptible enabled transition. With this model, an interrupt is unpredictable and we don't know when it happens. In fact at any time, the interruptible task and the interrupt both have equal chance to happen. The simple Petri nets of Fig.6.2 reported from (Allahham et Alla, 2008) is a simple example of SWPN of an interruptible task.

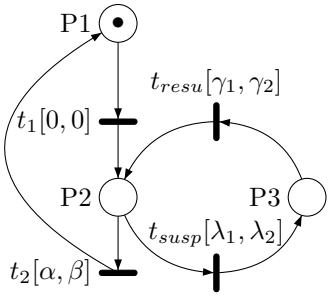


Figure 6.2 An interruptible task modeled by SWPN, reported from (Allahham et Alla, 2008).

Let t_i be a transition and M a marking. t_i is called:
– **Enabled**: if $M \geq Pre(t_i)$ denoted by $t_i \in En(M)$.
– **Firable**: if $t_i \in En(M) \wedge \downarrow Is(t_i) \leq v(t_i) \leq \uparrow Is(t_i)$, denoted by $t_i \in Firable(M, v)$.
– **Suspended**: if $Pre(t_i) > M \wedge v(t_i) > 0$, denoted by $t_i \in susp(M, v)$.
$\uparrow enabled(t_i, M, t_k)$ signifies that t_i is newly enabled by firing t_k at marking M.

The new concept $Pre - initialization$ defined in this model refers to non-interruptible transitions. For every $t_i \in T_{no-int}$, t_i is firable if $v(t_i)$ is already initialized when t_i becomes

enabled. In other words, $v(t_i) = 0$ when $t_i \in \uparrow enabled(t_i, M, t_k)$.

The concept, $Post - initialization$ is defined for interruptible transitions. When an interruptible transition is fired the associated $v(t_i)$ is initialized ($v(t_i) = 0$). Fig.6.3 and Fig.6.4 reported from (Allahham et Alla, 2008) show the difference between time evolution in an interruptible transition and a non-interruptible transition. In these figures, the notions E, D and F stand for enabling, disabling and firing of a transition respectively.

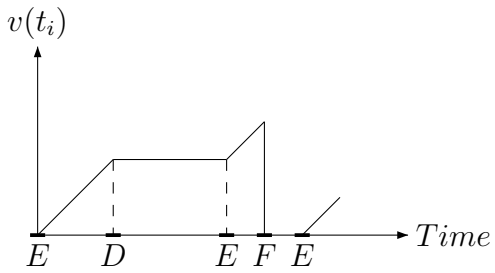


Figure 6.3 Time elapses, t_i is an interruptible transition.

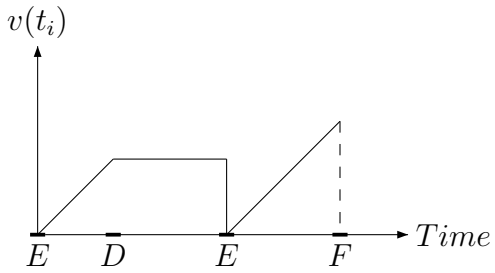


Figure 6.4 Time elapses, t_i is a non-interruptible transition.

In brief, according to the types of transitions, two types of clock initialization is defined:

– **Pre-initialization**: Given a transition $t_i \in T_{no-int}$ where T_{no-int} is the set of non-interruptible transitions, clock initialization is called pre-initialization happening when

t_i is recently enabled. As soon as a transition becomes enabled, its associated clock is initialized.

- **Post-initialization**: Given a transition $t_i \in T_{int}$ where T_{int} is the set of interruptible transitions, clock initialization is called post-initialization happening after firing t_i; it means that the transition initializes the clock after being fired. Post-initialization is dependent to transition firing.

In (Allahham et Alla, 2008), continuous and discrete transitions are defined as follows:
Let $d \in \mathbb{R}^+$, a continuous transition is denoted by $(M, v) \xrightarrow{d} (M, v')$ if and only if $\forall t_i \in T$:

- $v'(t_i) = v(t_i)$, if the transition t_i is not enabled.
- $v'(t_i) = v(t_i) + d$ if t_i is enabled.
- $M \geq Pre(t_i) \Rightarrow v' \leq \uparrow Is(t_i)$.

And a discrete transition is denoted by $(M, v) \xrightarrow{t_i} (M', v')$ if and only if $\forall t_i \in T$:

- $t_i \in Firable(M, v)$.
- $M' = M - Pre(t_i) + Post(t_i)$.
- $v'(t_i) = 0$ if $t_i \in T_{int}$ $(Post\text{-}initialization)$.
- $\forall t_k \in T$,$v'(t_k) =$.
  - 0 if $t_k \in \uparrow enabled(t_k, M, t_i)$;$(Pre\text{-}initialization)$.
  - $v(t_i)$ Otherwise.

In order to perform further timing analysis, the SWPN is transformed to hybrid automaton. In addition, a forward algorithm is suggested to compute the reachable states.

If the number of stopwatches increases, for example if all of the transitions are interruptible, the complexity of calculation will highly increase. An ordinary TPN is also a SWPN where $T_{int} = \emptyset$. After having a survey on different stopwatch Petri nets available in the literature, in the following we will investigate how to integrate stopwatch in controller synthesis approach suggested in Chapter 3.

6.3 Controller synthesis and stopwatch

In this section, we consider time Petri nets with controllable/uncontrollable transitions and show how such a system is controlled by associating some of the controllable transitions with stopwatch. First, we apply the on-the-fly algorithm of Section 3.4 and calculate the appropriate controller. Then, instead of restricting time intervals of corresponding controllable transitions, we associate them with stopwatch. In fact, we suspend some transition during its bad subinterval to control a system so as to satisfy a given property.

We consider the inhibitor hyperarcs of (Roux et Lime, 2004). The controller shall suspend a controllable transition in its bad subinterval and retrieve it otherwise. Let us remember the example of Fig.2.4 with the state class graph of Fig.2.5 and Table 2.2. We have seen that in order to prevent the system entering forbidden state classes α_4 and α_6, the controller should prevent t_1 from firing before $]2, 4]$. Thus, an inhibitor hyperarc is added to the transition t_1. This inhibitor hyperarc connects t_1 to a place called p_{susp}. At the beginning, this place is marked and then, t_1 is suspended. At $[2, 2]$ the token of p_{susp} is consumed and the corresponding hyperarc becomes disabled. The controllable transition t_1 is now firable again. The token is returned to place p_{susp} after t_1 is fired. Thus, the controller suspends t_1 during its bad subinterval and resumes it after then.

At the beginning, all clocks are initialized to zero. Then, time elapses but in $[0, 2]$, this transition is suspended and hence, its clock does not elapse anymore. Later at time $[2, 2]$, this transition is retrieved and becomes active. We don't need to delay it anymore, that's why the lower bound of the interval associated with t_1 is modified to 0. Now the clock starts elapsing. Within 2 time units t_1 should be fired (i.e. when the actual time of the general clock of the system reaches 4). Then, the interval associated with t_1 in a controlled TPN associated with stopwatch is $[0, 2]$. Fig.6.5 represents the clock evaluation of the transition t_1. The controlled Petri nets of this example is presented in Fig.6.6.

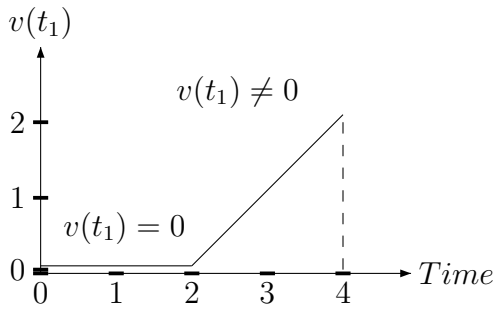


Figure 6.5 Clock evaluation of t_1 in the controlled TPN of Fig.6.6.

In summary, our goal is to use inhibitor hyperarcs and suspend the appropriate controllable transitions during their bad subintervals. The bad subintervals are calculated as in Section 3.4. Note that in Section 3.4, we were restricting and limiting time intervals while

in this approach the idea is to suspend and delay them. In this research, we suppose every controllable transition can be associated with a stopwatch if needed.

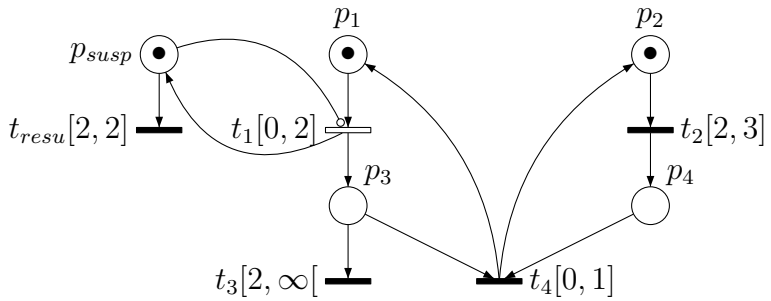


Figure 6.6 Time Petri net of Fig.2.4, controlled by inhibitor hyperarcs ($T_c = \{t_1\}$).

Above, we have shown how to control a system when the bad subinterval is at the beginning of the firing interval. What if the acceptable subinterval is at the beginning and bad subintervals are after? Consider the example of Fig.6.7. A controllable transition t_1 is associated with firing interval $[a, b]$. Suppose that, based on Algorithm 1, the subinterval $[a, \alpha_1]$ is acceptable while $]\alpha_1, b]$ is a bad subinterval, where $a \leq \alpha_1 < b$. The controlled time Petri nets is presented in Fig.6.8. Fig.6.9 represents the clock evaluation of t_1 and shows how the new interval associated with t_1 is calculated. In $[0, a[$, p_{susp} is marked and t_1 is neither active nor enabled. Meanwhile, at the time a, t_{r1} becomes enabled and is fired consuming the token in p_{susp}. Thus, right at a, the transition t_1 becomes active and its associated clock starts elapsing. Note that, associated stopwatch delays t_1 for a time units and we do not want to delay it anymore; then, the lower bound of the new associated interval in controlled TPN is 0. After α_1 time units, t_{r2} is fired and p_{susp} becomes marked again. Consequently, t_1 is suspended.

Consider the same example of Fig.6.7 and this time suppose that the algorithm 1 gives the following output:
$[a, \alpha_1[$ is acceptable, $[\alpha_1, \alpha_2[$ is a bad subinterval, $[\alpha_2, b]$ is acceptable where the condition $a < \alpha_1 < \alpha_2 \leq b$ holds. With the same idea, the controlled Petri nets is presented in Fig.6.10 and Fig.6.11 shows clock evaluation of t_1.

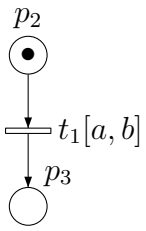


Figure 6.7 A simple time Petri net ($T_c = \{t_1\}$).

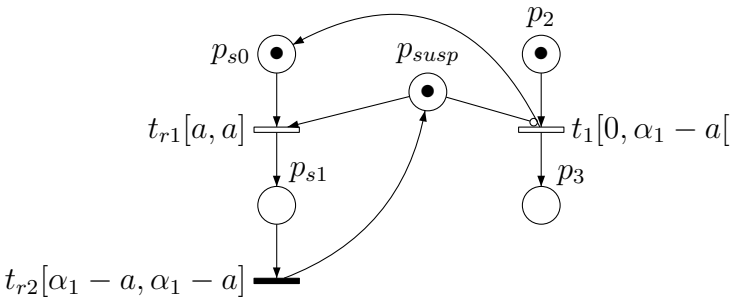


Figure 6.8 The controlled TPN of Fig.6.7 using inhibitor hyperarcs ($T_c = \{t_1\}$). Forbidden interval is $]\alpha_1, b]$ where $a \leq \alpha_1 < b$.

6.3.1 Why inhibitor hyperarcs?

We have shown how to achieve a controlled model from an uncontrolled time Petri nets by adding inhibitor hyperarcs. One question is why among different types of stopwatch inhibitor hyperarcs are chosen. Is it possible to use another stopwatch model? The answer is yes. Moreover, inhibitor hyperarcs provide more flexibility and less complication. Let us see if this idea is feasible using other types of Petri nets with stopwatch.

We consider the post and pre-initialized Petri nets proposed in (Allahham et Alla, 2007, 2008) for stopwatch and try to control a model where a controllable transition associated with time interval $[a, b]$ has a bad subinterval $[a, \alpha]$. Based on our hypothesis, the controller should suspend the controllable transition at $[a, a]$ and resume it at $[\alpha, \alpha]$. The characteristic of post and pre-initialized Petri nets is that they consume the same token of the original model and in addition, the exact time when the model becomes suspended is unknown as stopwatch

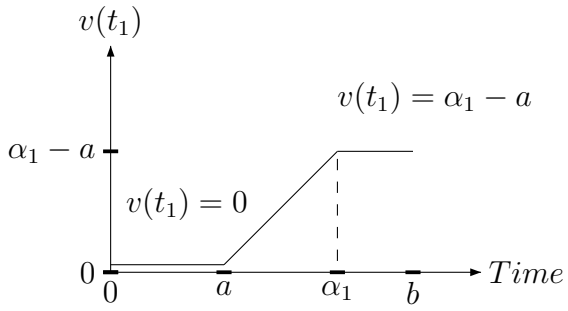


Figure 6.9 Clock evaluation of t_1 in the controlled TPN of Fig.6.7. Forbidden interval is $]\alpha_1, b]$ where $a \leq \alpha_1 < b$.

transition and original transition have the same chance of firing. At the first glance, the idea is feasible by adding one place and two transitions. However, there are some issues. We explain the problem through an example.

We consider the same example of Fig.2.4. The suggested controlled model using post and pre-initialized Petri nets comes in Fig.6.12. The controllable transition t_1 is associated with stopwatch. The stopwatch interrupts the task at $[0, 0]$ and resume it at $[2, 2]$. The problem is: at $[0, 0]$, both transitions t_1 and t_{sus} are enabled with equal chance of firing whereas, in order to have a controlled model, t_{sus} should fire at $[0, 0]$ before t_1. Thus, the controller fails unless if we modify the interval associated to t_1 to $]0, 2]$ which is not of interest in this approach.

The other alternative is presented at Fig.6.13. An auxiliary place p_s solves the problem. Hence, using the stopwatch Petri nets suggested in (Allahham et Alla, 2007, 2008), it is not easy to give a general solution to control a model considering the output of Algorithm 1. In addition, the resulting controlled nets are more complicated. For example, in case the controllable transition is associated with the interval $[a, b]$ where its bad subinterval is $[\alpha, b]$ and $a < \alpha$ then, the controlled model becomes complicated.

6.4 Illustrative example

Suppose a multitasking system with the following specifications: Four tasks t_1, t_2, t_3 and t_4 are being executed. Only the two first tasks are controllable. The task t_1 is periodic with

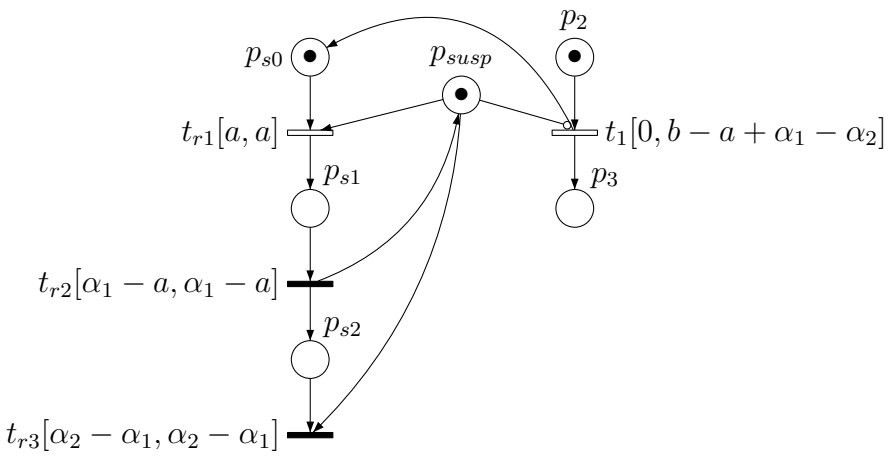


Figure 6.10 A simple time Petri net controlled by inhibitor hyperarcs ($T_c = \{t_1\}$); bad subinterval is $[\alpha_1, \alpha_2[$ where $a < \alpha_1 < \alpha_2 \leq b$.

a period of $[6, 6]$, t_4 is dependent on t_2 and t_3. Two tasks t_1 and t_2 share a common resource. On the other hand, t_1 enters through a single capacity buffer. It means that if t_1 is not executed by the end of its period, the system will block. The model of this system is depicted in Fig.6.14. If t_1 is not executed by the end of its period the system enters the state *Block* and, t_5 is the transition to be avoided. *Delay* is a place modeling the behavior of the buffer considering the period of the task t_1. And finally, t_1 executes within $[2, 2]$ time units, t_2 in $[2, 3]$, t_3 within $[5, 6]$ and t_4 in $[0, 1]$.

The state class graph of the model is given in Fig.6.15 and Table 6.1 shows the state class informations. Based on the state class graph of the system, the state classes $\alpha_4, \alpha_8, \alpha_{13}, \alpha_{14}$, α_{18}, α_{20} and α_{23} are forbidden. Let us briefly trace the algorithm on the state class graph. First, we follow the algorithm on the left branch where the state classes α_4, α_8 and α_{13} are located. The algorithm starts from α_0 and is executed recursively to α_1, then α_3, α_7 and finally reaches to α_{13} with a forbidden marking. Then it comes back to α_7 with $\{t_5\}$ and to α_3 with $\{t_4t_5\}$. The other successor available from α_3 is α_8 which is also forbidden. Till now, there is no enabled controllable transition to avoid these classes and the algorithm continues returning back to α_1 with $\{t_3t_4t_5, t_3t_5\}$. The other successor available from α_1 is α_4 which is also forbidden. Thus, it returns back to α_0 with $\{t_2t_3t_4t_5, t_2t_3t_5, t_2t_5\}$. The other successor available from α_0 is α_2 which is safe and the algorithm continues to $\alpha_5, \alpha_9, \alpha_{15}, \alpha_{19}$ and finally α_{18} which is a forbidden state. Then, it goes back to α_{19} with $\{t_5\}$ and consequently to α_{15} with $\{t_2t_5\}$. The other successors available from α_{15} are the forbidden state α_{20} and the state α_2 which is already under processing. Note that two enabled controllable transitions are available at α_{15} but the controller cannot act at this level because t_5 may fire before t_1

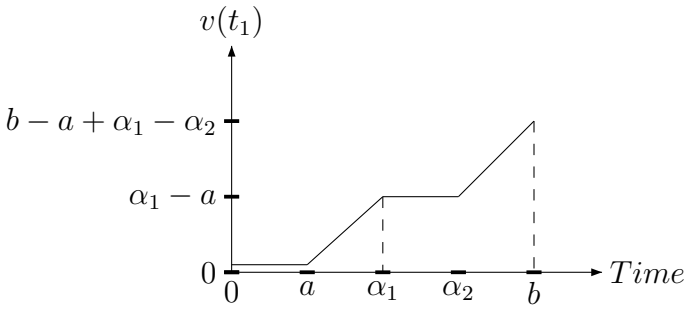


Figure 6.11 Clock evaluation of t_1 in a controlled TPN of Fig.6.7. Forbidden interval is $]\alpha_1, \alpha_2]$ where $a < \alpha_1 < \alpha_2 < b$.

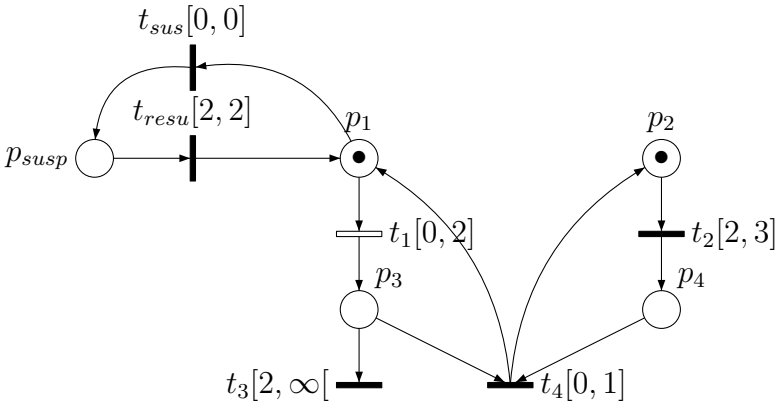


Figure 6.12 Controlling the example of Fig.2.4 using stopwatch of (Allahham et Alla, 2008). The controller fails.

or t_2 and lead to a forbidden state. Then, the algorithm continues and returns back to α_9 with $\{t_4t_2t_5, t_2t_5\}$ where no controllable transition is available. Consequently, the algorithm continues and returns back to α_5 and finally reaches to α_2 where newly enabled controllable transitions are available. Till now the path to be avoided at α_2 includes $\{t_2t_3t_4t_2t_5, t_2t_3t_4t_5\}$.

The procedure is similar for the other path available from α_2 including α_6 and its successors. The forbidden state reachable through this path is α_{23} and in order to avoid this forbidden state, the controller can act at α_{21} with two enabled controllable transitions. So, nothing is returned to α_2 from this path. Having a closer look at the state class graph and the output of the algorithm, we conclude that: at each state where t_1 and t_2 are newly enabled and the controller should act (i.e.α_0, α_2, and α_{21}), the controller should force t_1 to fire before t_2 (i.e.$t_2 - t_1 > 0$). In other words, t_2 should not be active at $[2, 2]$. It is sufficient to add an

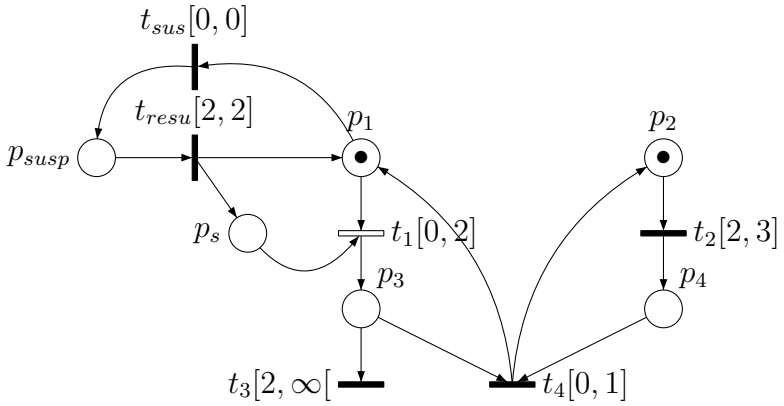


Figure 6.13 Controlled model of Fig.2.4 using stopwatch of (Allahham et Alla, 2008).

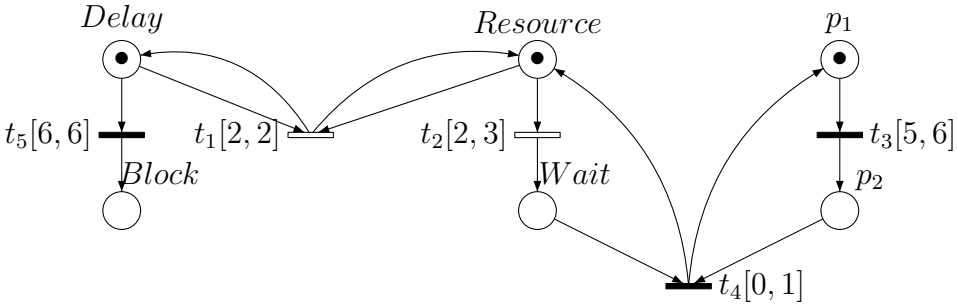


Figure 6.14 A Periodic system with $T_c = \{t_1, t_2\}$.

inhibitor hyperarc to deactivate t_2 at $[2,2]$. The controller cannot act at the state class α_{15} as discussed earlier and has nothing to do at the state class α_6 (the permissive controller will act later at the state class α_{21}). Note that, a state class like α_{14} which is reachable through a forbidden state (α_8) is not processed by the algorithm as it is supposed to be avoided in the controlled system. The controlled model using inhibitor hyperarcs is given at Fig.6.16.

Our model is now safe. Yet some challenges exist. Look at the state class graph of the model presented at Fig.6.15. In some paths one task is executed frequently while others are still waiting. For example, see the paths leading to α_{17}. Although the system is not blocked, some of the tasks cannot be executed. A good scheduler had better to perform different tasks alternatively. We may want to consider alternation, particular sequences, add a deadline for each task or other possible policies. And finally, we may ask if it is possible to restrict the execution time of a task even if it is in its safe subinterval. Then, while synthesizing a scheduler, we can add more constraints and scheduling policies beyond "safe states". Further

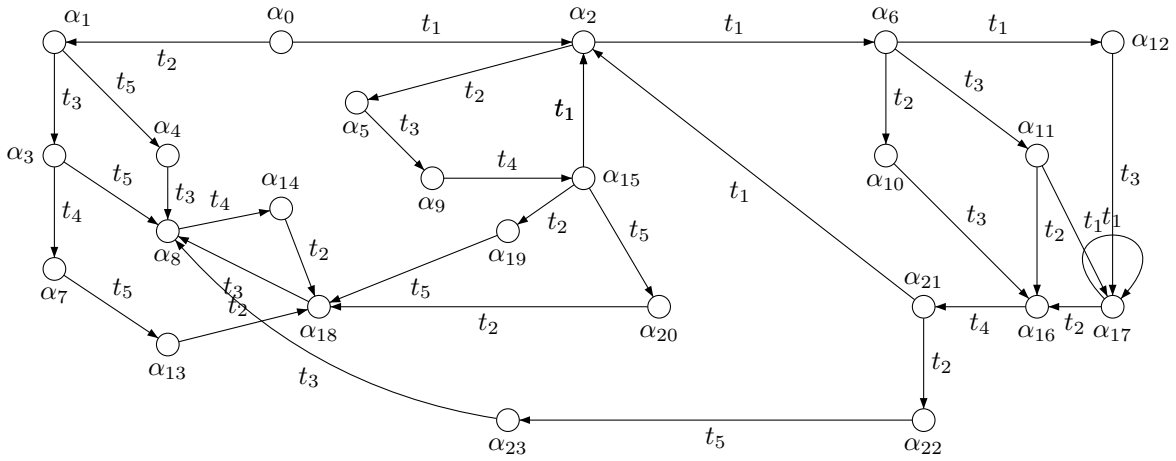


Figure 6.15 The state class graph of the TPN presented at Fig.6.14.

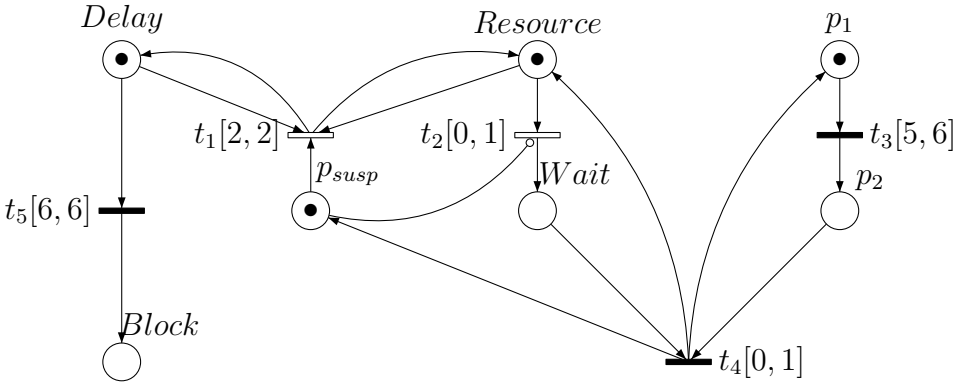


Figure 6.16 Controlled model of Fig.6.14 using inhibitor hyperarcs.

researches are required to answer these questions.

6.5 Conclusion

In this chapter, we have discussed that a scheduler is seen as a controller. We have also discussed some challenges of using controller synthesis for preemptive scheduling. We have adapted the approach explained in Section 3.4 to preemptive scheduling purposes by using time Petri nets associated with stopwatch. Considering the over approximation required for computing the state class graph of a time Petri nets associated with stopwatch, we have suggested an alternative solution to synthesize the time Petri nets without stopwatch, extract the appropriate controller and then, associate stopwatch to controllable transitions in order

Table 6.1 State classes of the TPN presented at Fig.6.14.

$\alpha_0 : Delay + P_1 + Resource$	$2 \leq t_2 \leq 3 \wedge 5 \leq t_3 \leq 6 \wedge 2 \leq t_1 \leq 2 \wedge 6 \leq t_5 \leq 6$
$\alpha_1 : Delay + P_1 + Wait$	$3 \leq t_3 \leq 4 \wedge 4 \leq t_5 \leq 4$
$\alpha_2 : Delay + P_1 + Resource$	$2 \leq t_2 \leq 3 \wedge 3 \leq t_3 \leq 4 \wedge 2 \leq t_1 \leq 2 \wedge 6 \leq t_5 \leq 6$
$\alpha_3 : Delay + P_3 + Wait$	$0 \leq t_4 \leq 1 \wedge 0 \leq t_5 \leq 1$
$\alpha_4 : P_1 + Wait + Block$	$0 \leq t_3 \leq 0$
$\alpha_5 : Delay + P_1 + Wait$	$1 \leq t_3 \leq 2 \wedge 4 \leq t_5 \leq 4$
$\alpha_6 : Delay + P_1 + Resource$	$2 \leq t_2 \leq 3 \wedge 1 \leq t_3 \leq 2 \wedge 2 \leq t_1 \leq 2 \wedge 6 \leq t_5 \leq 6$
$\alpha_7 : Delay + Resource + P_1$	$2 \leq t_2 \leq 3 \wedge 5 \leq t_3 \leq 6 \wedge 2 \leq t_1 \leq 2 \wedge 0 \leq t_5 \leq 1$
$\alpha_8 : P_3 + Wait + Block$	$0 \leq t_4 \leq 1$
$\alpha_9 : Delay + P_3 + Wait$	$0 \leq t_4 \leq 1 \wedge 2 \leq t_5 \leq 3$
$\alpha_{10} : Delay + P_1 + Wait$	$0 \leq t_3 \leq 0 \wedge 4 \leq t_5 \leq 4$
$\alpha_{11} : Delay + Resource + P_3$	$0 \leq t_2 \leq 2 \wedge 0 \leq t_1 \leq 1 \wedge 4 \leq t_5 \leq 5 \wedge$ $0 \leq t_2 - t_1 \leq 1 \wedge -4 \leq t_2 - t_5 \leq -3$ $\wedge -4 \leq t_1 - t_5 \leq -4$
$\alpha_{12} : Delay + Resource + P_1$	$2 \leq t_2 \leq 3 \wedge 0 \leq t_3 \leq 0 \wedge 2 \leq t_1 \leq 2 \wedge 6 \leq t_5 \leq 6$
$\alpha_{13} : Resource + P_1 + Block$	$1 \leq t_2 \leq 3 \wedge 4 \leq t_3 \leq 6 \wedge -4 \leq t_2 - t_3 \leq -2$
$\alpha_{14} : Resource + P_1 + Block$	$2 \leq t_2 \leq 3 \wedge 5 \leq t_3 \leq 6$
$\alpha_{15} : Delay + Resource + P_1$	$2 \leq t_2 \leq 3 \wedge 5 \leq t_3 \leq 6 \wedge 2 \leq t_1 \leq 2 \wedge 1 \leq t_5 \leq 3$
$\alpha_{16} : Delay + P_3 + Wait$	$0 \leq t_4 \leq 1 \wedge 4 \leq t_5 \leq 4$
$\alpha_{17} : Delay + Resource + P_3$	$2 \leq t_2 \leq 3 \wedge 2 \leq t_1 \leq 2 \wedge 6 \leq t_5 \leq 6$
$\alpha_{18} : P_1 + Wait + Block$	$2 \leq t_3 \leq 4$
$\alpha_{19} : Delay + P_1 + Wait$	$3 \leq t_3 \leq 4 \wedge 0 \leq t_5 \leq 1$
$\alpha_{20} : Resource + P_1 + Block$	$0 \leq t_2 \leq 2 \wedge 3 \leq t_3 \leq 5 \wedge -4 \leq t_2 - t_3 \leq -2$
$\alpha_{21} : Delay + Resource + P_1$	$2 \leq t_2 \leq 3 \wedge 5 \leq t_3 \leq 6 \wedge 2 \leq t_1 \leq 2 \wedge 3 \leq t_5 \leq 4$
$\alpha_{22} : Delay + P_1 + Wait$	$3 \leq t_3 \leq 4 \wedge 1 \leq t_5 \leq 2$
$\alpha_{23} : P_1 + Wait + Block$	$1 \leq t_3 \leq 3$

to prevent their bad subintervals. We have used inhibitor hyperarcs to suspend a controllable transition in its bad subinterval.

The approach suggested in this chapter is useful for preemptive scheduling purposes to manage critical sections and shared resources. Further studies could be interesting to see how different scheduling policies or constraints could be combined to this approach.

CHAPTER 7

CONCLUSION

7.1 Analysis of the achievements

In this thesis, we have proposed a forward on-the-fly algorithm for controller synthesis of safety properties in a system modeled by time Petri nets. We have also extended our algorithm for controller synthesis of reachability properties. The proposed algorithm investigates the control problem and answers to the question of existence of a controller. In other words, the algorithm guarantees to give a controller if it exists. If this algorithm fails to calculate a controller then, the controller does not exist. The proposed approach is decidable for bounded time Petri nets. This algorithm does not need to calculate the controllable predecessors of the states which is a costly operation and is used in former methods.

Our algorithm explores the state class graph of a system on-the-fly and collects the paths leading to forbidden states. Then, it restricts the time intervals of controllable transitions to prevent these paths happening. This approach calculates a state based controller. However, a marking dependent controller or a static controller can be extracted. We have shown that the state based controller is the maximally permissive. We have also proven the correctness of the algorithm for both safety and reachability properties.

We have optimized the algorithm and suggested to use a more abstracted state space for controller synthesis. We have investigated different methods of abstraction on our algorithm. We have concluded that abstraction by inclusion and abstraction by convex union are convenient solutions to attenuate the state space explosion. Abstraction by convex hull is less appropriate because it does not preserve the boundedness property and then marking boundedness. With this optimization, our algorithm is more convenient for controller synthesis of large-scale and complex systems.

We have shown through some case studies that although abstraction methods increase the complexity of calculations, they reduce efficiently the number of state classes. Considering the linear relation between the complexity of our controller synthesis algorithm with the number of available states, using abstraction is very effective. The global time of processing is reduced and state space explosion is attenuated.

We have investigated the implementation of our devised algorithm on modular systems to answer the question of controller synthesis problem. We have explained how to adapt the suggested algorithm to modular systems and achieve a set of decentralized controllers. In summary, the idea has been to synthesis a central controller and then implement it as a set of decentralized local controllers on a distributed system. We have shown that if a controller exists and the proposed algorithm succeeds to calculate a controller, it is possible to implement it on a distributed and modular system. First, the behavior of all modules are modeled as a whole and our suggested algorithm is applied on. Then, we implement the obtained controller on the modules. Each module has a local controller. We have considered a global property which is unique in all modules.

We have studied independent local controllers as well as local controllers with the possibility of intercommunications. We have concluded that independent local controllers are less permissive. If intercommunication delay is comparable to the state evolution frequency and is not negligible, this delay should be considered in the global model, before synthesizing the controller. In that case, a place and an uncontrollable transition model the synchronization and intercommunication delay.

Finally, we have proposed that in case a controller (like a scheduler) cannot restrict time intervals associated with controllable transitions, the controllable transitions can be suspended in their bad subintervals. In time Petri nets associated with stopwatch, it is possible to suspend a task for a while. Synthesizing a time Petri net associated with stopwatch is complicated and needs some over approximations. Besides, time Petri nets associated with stopwatch do not preserve the boundedness property. In this thesis, we have suggested a more effective alternative. We synthesize a time Petri net without stopwatch and then, equip the controllable transitions with stopwatch where necessary. With this assumption, every controllable transition can be suspended in its bad subinterval and resumed otherwise. Amongst different types of stopwatch, our solution is based on inhibitor hyperarcs. This approach is particularly useful at design level for preemptive scheduling purposes and managing shared resources and critical sections.

7.2 Limitations of the approach

We have shown how to implement an already synthesized centralized controller on a modular system which is particularly interesting for controller synthesis of large-scale systems.

Although we have discussed some methods of abstraction, the state space explosion is always a challenge and/or limitation in the analysis of large-scale systems.

We have discussed that in order to have a state dependent decentralized controller, timing information of each state should be announced when a controllable transition is newly enabled. These exchanges are costly. Therefore, this approach is less interesting if the number of inter module exchanges is significant.

In the approach discussed in this thesis, we have assumed that the controller can only react on the transitions and modify their associated timing intervals. The controller cannot modify places, the available markings or the specifications of the arcs. In fact in our approach, the controller is limited to the given input data.

7.3 Future work

Our first perspective is to implement the suggested algorithm as a tool with a graphical interface. Model checking and controller synthesis are both very useful when applied in practical applications in the industry. A user friendly graphical interface facilitates these synthesis for both common users and professionals. Such a tool can hide complex mathematical analysis and can facilitate application of the method in the industry.

In the control synthesis of modular systems, we have considered that the whole system is synthesized and a centralized controller is obtained. We have answered to the controller synthesis problem and shown how to implement an already synthesized centralized controller on a modular system. It is interesting to study if a decentralized controller can be achieved directly through parallel controller synthesis in each module.

Controller synthesis is also applied in preemptive scheduling purposes where a controller is in fact a scheduler. Here, we have suggested to control a TPN using stopwatch to suspend and resume a task when needed. This is a good starting point for further researches on scheduling and to calculate automatically a scheduler in a multitasking system. Considering different scheduling policies is challenging and is worth to be considered in further studies.

In this thesis, our controller reacts on the timing intervals associated with the transitions. It is interesting to investigate other controller synthesis approaches where the controller can react on the places, markings and/or arcs. Further researches are worth to be done to extract

other controllers by adding new places, modifying available markings or manipulating the arcs.

REFERENCES

ABID, C. et ZOUARI, B. (2010a). Decentralised active controller. *7th International Conference on Informatics in Control, Automation and Robotics , ICINCO*, Volume 2, 252–59.

ABID, C. A. et ZOUARI, B. (2010b). Synthesis of controllers for symmetric systems. *International Journal of Control*, Volume 83, 2354–2367.

ALLAHHAM, A. et ALLA, H. (2007). Réseaux de Petri à chronomètres post et pré-initialisés. *6ème Colloque Francophone sur la Modélisation des Systèmes Réactifs(MSR'07)*. 263–280.

ALLAHHAM, A. et ALLA, H. (2008). Post and pre-initialized stopwatch petri nets: Formal semantics and state space computation. *Nonlinear Analysis: Hybrid Systems*, Volume 2, 1175 – 1186.

ALTISEN, K., BOUYER, P., CACHAT, T., CASSEZ, F. et GARDEY, G. (2005). Introduction au contrôle des systèmes temps-réel. *Journal Européen des Systèmes Automatisés*, Volume 39, 367–380.

ALTISEN, K., GOSSLER, G., PNUELI, A., SIFAKIS, J., TRIPAKIS, S. et YOVINE, S. (1999). A framework for scheduler synthesis. *20th IEEE Symposium of Real-Time Systems.* 154–63.

ALTISEN, K., GOSSLER, G. et SIFAKIS, J. (2000). A methodology for the construction of scheduled systems. *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems(FTRTFT)*. 106–20.

ALUR, R. (1999). Timed automata. *International Conference on Computer Aided Verification: CAV'99*. 8–22.

AYDIN, A. et ALTUG, I. (2009). Decentralized structural controller design for large scale discrete event systems modelled by Petri nets. *KYBERNETIKA*, Volume 45, 3–14.

BADOUEL, E., BERNARDINELLO, L. et DARONDEAU, P. (1995). Polynomial algorithms for the synthesis of bounded nets. *6th International Joint Conference CAA/FASE.* 364–378.

BEHRMANN, G., BOUYER, P., LARSEN, K. et PELANEK, R. (2006a). Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, Volume 8, 204 – 15.

BEHRMANN, G., COUGNARD, A., DAVID, A., FLEURY, E., LARSEN, K. G. et LIME, D. (2007). Uppaal-tiga: Time for playing games! *17th International Conference on Computer Aided Verification*, Volume 4590 of LNCS, 121–125.

BEHRMANN, G., DAVID, A., LARSEN, K. G., HAKANSSON, J., PETTERSON, P., WANG, Y. et HENDRIKS, M. (2006b). Uppaal 4.0 verification tool for timed automata. *3rd International Conference on the Quantitative Evaluation of Systems*. 125–6.

BENGTSSON, J. (2002). *Clocks, DBMs and states in timed systems.* PhD dissertation, Uppsala Universitet (Sweden).

BERTHOMIEU, B. et DIAZ, M. (1991). Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, Volume 17, 259–273.

BERTHOMIEU, B. et MENASCHE, M. (1983). An enumerative approach for analyzing time petri nets. *9th World Computer Congress (IFIP)*. 41 – 6.

BERTHOMIEU, B. et VERNADAT, F. (2003). State class constructions for branching analysis of time Petri nets. *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 442–457.

BOUCHENEB, H., GARDEY, G. et ROUX, O. H. (2009). TCTL model checking of time Petri nets. *Journal of Logic and Computation*, Volume 6, 1509–1540.

BOUCHENEB, H. et HADJIDJ, R. (2008). *Model checking of time Petri nets*, vol. Petri Net:Theory and Application. I-Tech Publishing, Vienna, Austria.

BOUCHENEB, H. et MULLINS, J. (2003). Analysis of temporal nets: calculation of o(n2) classes and o(mn) path times. *Technique et Science Informatiques*, Volume 22, 435 – 59.

BOUYER, P., CHEVALIER, F., KRICHEN, M. et TRIPAKIS, S. (2005). Observation partielle des systèmes temporisés title of translation:partial observation of timed systems. *Journal Europeen des Systemes Automatises*, Volume 39, 381–393.

BOYER, M. et VERNADAT, F. (2000). Language and bisimulation relations between subclasses of timed petri nets with strong timing semantic. *Technical report, LAAS*.

BOZGA, M., DAWS, C., MALER, O., OLIVERO, A., TRIPAKIS, S. et YOVINE, S. (1998). Kronos: a model-checking tool for real-time systems. *Lecture Notes in Computer Science*, Volume 1486, 298.

BUY, U. et DARABI, H. (2003). Deadline-enforcing supervisory control for time Petri nets. *IMACS Multiconference on Computational Engineering in Systems Applications (CESA)*.

BUY, U., DARABI, H., LEHENE, M. et VENEPALLY, V. (2005). Supervisory control of time Petri nets using net unfolding. *29th Annual International Computer Software and Applications Conference (COMPSAC)*, Volume 2, 97–100.

CASSEZ, F., DAVID, A., FLEURY, E., LARSEN, K. G. et LIME, D. (2005). Efficient on-the-fly algorithms for the analysis of timed games. *16th International Conference on concurrency theory*. 66–80.

CASSEZ, F. et LARSEN, K. (2000). The impressive power of stopwatches automata. *11th International Conference on Concurrency Theory. Proceedings of CONCUR 2000.* Springer-Verlag, 138–152.

CASSEZ, P., DAVID, A., LARSEN, K., LIME, D. et RASKIN, J.-F. (2007). Timed control with observation based and stuttering invariant strategies. *5th International Symposium of Automated Technology for Verification and Analysis, ATVA.*, Volume 4762 of LNCS, 192 – 206.

DAVID, A., KIM, G. L., LEGAY, A., NYMAN, U. et WASOWSKI, A. (2010). ECDAR: An environment for compositional design and analysis of real time systems. *8th International Symposium of Automated Technology for Verification and Analysis (ATVA)*, Volume 6252 of LNCS, 365–370.

DAWS, C. et TRIPAKIS, S. (1998). Model checking of real-time reachability properties using abstractions. *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems(TACAS)*, Volume 1384 of LNCS, 313–329.

GARDEY, G., ROUX, O. et ROUX, O. (2006a). State space computation and analysis of time petri nets. *Theory and Practice of Logic Programming*, Volume 6, 301 – 20.

GARDEY, G., ROUX, O. E. et ROUX, O. H. (2006b). Safety control synthesis for time Petri nets. *8th International Workshop on Discrete Event Systems*. 22–28.

GHAFFARI, A., REZG, N. et XIE, X. (2003). Design of a live and maximally permissive Petri net controller using the theory of regions. *IEEE Trans. Robot. Autom.*, Volume 19, 137–141.

GIUA, A., DICESARE, F. et SILVA, M. (1992). Generalized mutual exclusion contraints on nets with uncontrollable transitions.

HADJIDJ, R. (2006). *Analyse et validation formelle des systèmes temps réel*. PhD dissertation, École Polytechnique de Montréal.

HEIDARI, P. et BOUCHENEB, H. (2010). Efficient method for checking the existence of a safety/ reachability controller for time Petri nets. *10th International Conference on Application of Concurrency to System Design(ACSD)*. 201–210.

HEIDARI, P. et BOUCHENEB, H. (2012a). Controller synthesis of time Petri nets using stopwatch. *Journal of Engineering*. Submitted.

HEIDARI, P. et BOUCHENEB, H. (2012b). A forward on-the-fly approach in controller synthesis of time Petri nets. *InTech - open science - open minds*. Book chapter.

HEIDARI, P. et BOUCHENEB, H. (2012c). Maximally permissive controller synthesis for time Petri nets. *International Journal of Control*. Accepted.

HEIDARI, P., BOUCHENEB, H. et HADJIDJ, R. (2011). A forward on-the-fly method for controller synthesis in time Petri nets. *IEEE Transactions on Automatic Control*. Under revision.

HEIDARI, P., BOUCHENEB, H. et HADJIDJ, R. (2012). A decentralized controller for distributed systems modelled by time Petri nets. *IEEE Transactions on Control Systems Technology*. Under revision.

HILLAH, L. (2009). *Intégration des méthodes formelles au développement dirigé par les modèles, pour la conception et la vérification des systèmes et applications répartis*. PhD dissertation, University of Pierre & Marie Curie (France).

IORDACHE, M. V. et ANTSAKLIS, P. J. (2010). Concurrent program synthesis based on supervisory control. *The American Control Conference (ACC)*. 3378–3383.

ISOVIC, D. et FOHLER, G. (2000). Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. *Proceedings - Real-Time Systems Symposium*. 207 – 216.

JENSEN, K. et ROZENBERG, G. (1991). High-level Petri nets: Theory and application.

LARSEN, K., WEISE, C., YI, W. et PEARSON, J. (1999). Clock difference diagrams. *Nordic Journal of Computing*, 271–298.

LEWERENTZ, C. et LINDNER, T. (1995). *Case study 'Production Cell': A comparative study in formal specification and verification*. No. 1009. 388 – 388.

LIEN, Y. (1976). Termination properties of generalised Petri nets. *SIAM Journal on Computing*, Volume 5, 251–65.

LUO, J. (2009). Decentralized control approach of Petri nets based on net structure decomposition methods. *International Conference on Computers and Industrial Engineering, CIE*. 1560–67.

MELCHER, H. et WINKELMANN, K. (1998). Controller synthesis for the 'Production Cell' case study. *2nd Workshop on Formal Methods in Software Practice (FMSP)*. 24 – 33.

MERLIN, P. M. (1974). *A study of the recoverability of computing systems*. PhD dissertation, University of California, Irvine, United States.

MOODY, J. et ANTSAKLIS, P. (2000). Petri net supervisors for DES with uncontrollable and unobservable transitions. *IEEE Transactions on Automatic Control*, Volume 45, 462–76.

PENCZEK, W. et POLROLA, A. (2004). Specification and model checking of temporal properties in time Petri nets and timed automata. *25th International conference on application and theory of Petri nets*, Volume 3099 of LNCS, 37–76.

RAMADGE, P. J. et WONHAM, W. M. (1987). Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, Volume 25, 206–230.

ROUX, O. H. et LIME, D. (2004). Time Petri nets with inhibitor hyperarcs. formal semantics and state space computation. *ICATPN'04*. 371–390.

ROUX, O.-H., LIME, D., HADDAD, S., CASSEZ, F. et BÉRARD, B. (2005). Comparison of different semantics for time Petri nets. *3rd Automated Technology for Verification and Analysis*, Volume 3707 of LNCS, 293–307.

RUDIE, K. et WONHAM, W. (1992). Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, Volume 37, 1692 – 708.

SATHAYE, A. S. et KROGH, B. H. (1993). Synthesis of real-time supervisors for controlled time Petri nets. *32nd Conference on Decision and Control*, Volume 1, 235–236.

SHENGBING, J., KUMAR, R., TAKAI, S. et WENBIN, Q. (2010). Decentralized control of discrete-event systems with multiple local specifications. *IEEE Transactions on Automation Science and Engineering*, Volume 7, 512 – 22.

TRIPAKIS, S. (1998). *L'Analyse Formelle des Systèmes Temporisés en Pratique*. PhD dissertation, Université Joseph Fourier - Grenoble 1 Sciences et Geographie.

WONG-TOI, H. et HOFFMANN, G. (1991). The control of dense real-time discrete event systems. *30th IEEE Conference on Decision and Control Part 2 (of 3)*, Volume 2, 1527–1528.

WU, N., CHU, C., CHU, F. et ZHOU, M. (2008). Modeling and schedulability analysis of single-arm cluster tools with wafer residency time constraints using Petri net. *International Conference on Networking, Sensing and Control, (ICNSC)*. 84–89.