Synchronous Product of Time Petri Nets and its Applications to Fault-Diagnosis

Eric Lubat

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *30/11/2021* **par :**

ERIC LUBAT

Synchronous Product of Time Petri Nets and its Applications to Fault-Diagnosis

---

JURY

ARMAND TOGUYENI	Professeur des Universités	Président du Jury
CLAIRE PAGETTI	Ingénieure de recherche	Membre du Jury
AUDINE SUBIAS	Maître de Conférences	Membre du Jury
DIDIER LIME	Professeur des Universités	Membre du Jury
MOHAMED GHAZEL	Directeur de recherche	Membre du Jury
PIERRE-EMMANUEL HLADIK	Maître de Conférences	Membre du Jury

École doctorale et spécialité :
EDSYS : Systèmes embarqués 4200046

Unité de Recherche :
Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS)

Directeur de Thèse :
Pierre-Emmanuel HLADIK

Rapporteurs :
Didier LIME et *Mohamed GHAZEL*

To the best there is, the best there was and the best there ever will be.
My architect and brother, Matthieu LUBAT.

Acknowledgements

> Nobody is gonna hit as hard as life. But it ain't about how hard ya hit. It's about how hard you can get hit and keep moving forward.
>
> *Rocky Balboa*

The beginning is a very delicate time. Know that my journey began in the summer of 1993, and for these 28 years of life, I mainly have one thing to say: I am grateful.

First of all, I would like to thank my supervisor, Dr. Pierre-Emmanuel Hladik. Kind, funny, always there when needed, Dr. Hladik was the perfect match for an atypical PhD Student and I will always consider him as my master and example in life. I was impulsive when arriving in my first year and he taught me patience by being an example of it. If every teacher was more like him, engineering school would be a funnier place.

I would like to pay special regards to Dr. Silvano Dal Zilio, who was there when I needed advice and gave me the chance to proceed in a PhD. During the first year he taught me everything I needed to work on the field of critical engineering despite his commitments and without him, it would have been much harder to proceed to the end of this journey.

I want to pay homage to Dr. Didier Le Botlan who was like a third supervisor to me. Every team needs an intriguing hooded figure who grants wonderful advice in a cynical way. He was this and so much more. He was the first to give me a class to teach, he was there during lockdown, checking on me by email, wondering if I was ok. Through all these years, he was always another example to follow and the perfect combination between fun and pedagogic teaching.

I also want to acknowledge Dr. Yannick Pencole and Dr. Audine Subias. Their view on the field of diagnosis was essential to this work and their advice helped me as much in class than in my work.

I want to thank INSA, LAAS-CNRS and the VERTICS team for the opportunity to become a PhD student. After two years of PhD being the sole student of the VERTICS team I also want to thank my two co-workers, Thomas and Nicolas. It was a pleasure getting advice from Thomas and transmitting them to Nicolas.

As I was saying, after all these years, I am grateful. I want to pay respect to my close circle of friends who helped me go through this bumpy ride that has been this thesis. I want to thank Yoann, my room-mate, who I have known since kindergarten. He was always there during the past three years. I want to thank my parents, my brothers, my family and my friends (Rémi, Brice, Ernesto, Ben, Benji, Vicente and all those whom I do not mention there).

Teaching was a great part of this PhD and I want to pay special regard to the student I helped the most, Keyro, whom I now consider as a little brother. You

grew up good and wonderful. It was great just watching you evolving, every day was like a privilege.

I want to thank the two groups who helped me the most during the lockdown. First of all, the Fantasy Sphere store which let me work on my manuscript in their warehouse while they were selling cards. Secondly, I want to thank the Violent Squirrels, my friends on the internet. They were an essential part of getting through the pandemics by being there for advices and video games.

I also want to acknowledge the people who wrong me during this thesis. It was not only a bumpy ride because of the lockdown or because of the work, but sometimes because of the people who wanted to impact me in a negative way. To my enemies, to my exes and to every person I do not know who wants to hurt me, I want to say one thing. I forgive but I do not forget. And after this thesis, I forgive all of you and I wish you a long and happy life without me.

I have been down. I have been up. I have felt like the first and the last in a race that was not taking place. But after all this time, I am one thing. I am grateful.

I am grateful for the friends I have made and the enemies I have lost.

I am grateful for the adventure of a lifetime this thesis has been. It was as much a research work on computer science than a work on myself for the past three years.

I have gone through all these complex emotions in this last little bit. I have been angry, I have been sad, I have been frustrated, I have been all of that, but today, when I woke up this morning, I felt nothing but gratitude, because I have gotten to do what I love for nearly 3 years. It may be my thesis but it is our accomplishment. I will conclude by saying it again.

I am grateful for all of you.

Contents

Chapter 1

Introduction

The work performed during my PhD thesis focus on the formal verification of Discrete Event Systems (DES), and more precisely the study of properties on the timed languages defined in the context of Time Petri Nets. This work was accomplished at the LAAS-CNRS (Laboratory for Analysis and Architecture of Systems) in the Verification of Time Critical Systems team (Verification de Systèmes Temporisés Critiques – VERTICS) and at the "Institut National des Sciences Appliquées" of Toulouse (INSA-Toulouse).

This introductory chapter gives a quick overview of this thesis manuscript. We first introduce different notions related to DES and also provide some motivations behind our work.

1.1 Context and Motivations

Over the past 40 years, formal verification (on all kind of systems) has held an important place in computer science. Computing systems are present all around us and have a substantial impact in our daily life. Formal verification is a field of computer science that focuses on providing theories, methods and tools for checking that systems fulfill the requirements drafted by their designers. Formal Verification is strongly applied in the context of *safety critical systems* (in the aerospace sector for example) where the term *safety* is used to imply catastrophic consequences (injuries, death, ...) in case of a failure.

We can list some outstanding examples of catastrophic failures, where problems with a computer program or a controller had a preponderant role. Each of those major incidents have attracted the attention of the general public at their time (see for instance [Neumann 1994] for a list of such incidents):

- *Therac-25* (1985–87) : Between June 85 and January 87, a computer-controlled radiation therapy machine, the Therac-25, severely overdosed six patients due to a software coding issue.
- *Ariane 5* (1996) : The inaugural launch of the European Ariane 5 ended in a blast. This failure was caused by an internal software exception that was not handled during the execution of a data conversion from 64-bits floating point to a 16-bits signed integer value.
- *NASA Mars Pathfinder* (1997) : The Martian rover started losing information due to several system resets. The system was restarted due to a problem

of priority inversion and resulted in delays in relaying data, shortening the duration of the mission.

- *Charles Schwab Corporation* (April 13, 2021) : In April of 2021, the Charles Schwab Corporation transferred 1.2 millions of dollars to a 33 years old American due to a computer bug. The bank managed to get back three quarters of the money before all was spent.

But formal verification is not here only to show how we could have adverted failures. We can also present some examples where developing safety critical applications using formal methods was a success (a survey on this topic is available in [Garavel 2012]):

- *FM8501* (1985) : Formal verification of the 16-bit FM8501 microprocessor using the NQTHM theorem prover. This was the first verified microprocessor, followed by many others [Hunt 1994].
- *Four color theorem* (2005) : Computer-checked proof of the "four color theorem", using the Coq proof assistant, a complex problem in discrete mathematics with a long history of flawed and fallacious proofs [Gonthier 2007].
- *Formal modelling of the EMV (Europay-MasterCard-Visa) protocol* (2011) : Formal modelling of the EMV protocol suite in the F# language [de Ruiter 2011] and automated analysis of these protocols by joint use of the FS2PV [Bhargavan 2006] and ProVerif tools [Blanchet 2004].

In my thesis, I focus on the formal verification of reactive, communicating systems, with a particular emphasis on properties that rely on the respect of hard, real-time constraints.

The market demands for more efficient and automated solutions has pushed the complexity of embedded systems to levels never imagined before. Model-checking became then a natural solution, since it proposes a *push-button* solution to check the safety and property on such complex systems. Model-checking catches errors early in the system design phase, before they become very expensive to fix and it can be easily integrated into a standard development cycle.

One of the fields we explore to tackle the verification of DES is *fault diagnosis*. Fault diagnosis plays an essential role in the safe operation of industrial systems. We focus on a particular property, based on the analysis of the behaviour of DES, called diagnosability. First and foremost, we have to define two different notions, *diagnosis* and *diagnosability*. Diagnosis (or *online diagnosis*) is the method performed to detect and localize the cause of a fault. Diagnosability (or *diagnosability analysis*) is a property that is true for systems such that it is always possible to detect and locate any specific fault after its occurrence. We also often ask that detection occurs within a finite delay, or after a finite number of "observable events". Diagnosability is analysed offline, on a model of the system, and can be defined as a property on the language of DES.

To summarize, this thesis deals with Discrete Event Systems, and more precisely with the formal verification of properties about their temporal and timed behaviour.

1.2 Discrete Event Systems

Discrete Event Systems are model-based specifications that describe the behaviour of a system. A DES is often described as a set of discrete elements (distinct states and distinct events linking them). For example, a simple door can be abstracted as a DES system with two possible discrete states, *open* and *closed*. An event in this case is the occurrence of an action in the system that can change its state. In our case, we could consider two possible events, *closing* and *opening*.

Recently, we have seen the rapid and complex evolution of DES all around us. Several of these new systems rely strongly on new methods of programming ("machine learning" for example). However, the validation and verification of such algorithms is still undergoing research to reach the robustness of classical validation and verification methods [Hand 2020]. Therefore, we still need robust systems not relying on new methods for a lot of safety critical system projects (aeronautics, automotive, etc) and some of the robust methods need to be extended to the field of new algorithms.

DES can also be defined as a probabilistic model to convey more information regarding the behaviour of the system in the form of a probabilistic event. However, in this thesis, we mainly focus on DES with timed and deterministic behaviour.

In this work we strongly rely on the notion of model, which are much deeply presented in the Chapter 2 of this thesis.

1.3 Observable Events and Sequence of Executions

One of the core concepts in the formal verification of DES is *observability*. Even if a DES requires a large set of events E to describe its dynamics, it is often not the case that a "bystander" (or outside observer) can *observe* all of them. Sometimes, only a fraction of E is observable. This set of observable events, that we denote E_o in the following, limits the set of properties that an observer can monitor. It also implies the notion of unobservable events, E_u, such that $E_u = E \setminus E_o$.

In the following, we should also use the notion of *label*, that is a tag or marker that can be used to represent a group of related events; such that all events with the same label are indiscernible from each other by an observer. Like with events, we will have observable and unobservable labels.

Finally, any reasonable observer should be able to record the order (and the dates) at which actions occur. This leads to the notion of a *sequence (or chain) of events*—that we should also call an *execution*—that will be the central notion that we study in this work. The set of all labelled sequences of events in a system is called its *language*. By looking at the "words" (labelled sequences of events) in this language, we can express some properties of our DES. Like for instance reachability

properties, that answers questions such as "can we observe an occurrence of the event a ?", temporal properties (properties about the order in which events can occur in the system) of the form "it is true that the event a is always eventually followed by the event b ?"

I will not focus on the formal verification of temporal properties in this work, but rather on the verification that a system is diagnosable or not. Like in the work of a detective, it is possible to infer a property about unobservable events by looking at the clues given by the observable events in an execution, and the order and date at which they occur. This problem is related to the notion of *diagnosability*, meaning the property, for a system, that it is always possible to decide whether some (given) unobservable behaviour occurred by looking only at the sequence of observable events.

It turns out that many verification methods and techniques rely on the notion of *intersection* between languages to express properties on a system; and also checking whether such intersection is empty or not. This is for example the case with model-checking [Clarke 1999], when using automata-theoretic approaches. Another example is with the *theory of supervisory control* [Ramadge 1989], for deciding whether it is possible to synthesize a supervisor given a discrete-event dynamic system.

One distinctive characteristic of our approach is that we use an indirect method and do not directly compute a product between "state graphs". Instead, we replace the use of intersections, at the level of (sets of) behaviours, by a notion of *composition*, at the model-level. Meaning that we want to provide an effective method where the (language) intersection of two models—for instance a system and its observer—can be defined or computed using the "product" of these systems. In the next chapters (see for example Section 2.2), we show that such a notion of product is not easy to define in the presence of timing constraints. This is what motivates most of our definitions and is at the basis of most of our results.

1.4 Petri Nets

In this work, we choose Petri nets as the main formalism used to define the specification of a DES. They will be the syntax we use to describe the possible behaviours of a DES.

A Petri Net (PN) [Petri 1962] is a discrete device that defines when events in a system can occur and how they interact with each other; it is a calculus to reason about concepts such as concurrency and causality. To stick with the problem we want to address in this work, we will apply Petri nets to reason about observability.

Petri nets can be understood as a *calculus* because they are defined from a deliberately small set of elements, interacting by using a very limited set of rules (or operations). Actually, we only use four elements: transitions, places, arcs and tokens. The events in a PN are associated with *transitions*, while states are associated with *places*. In its most basic form, all places contain the same kind of resources, called *tokens* and the global state of a net is given by the amount of

tokens in each of its places. Therefore the state of a system can be interpreted as a mapping between places and the number of tokens that they contain. Finally, places and transitions are connected together using (directed) *arcs*, in a graph-like fashion, that expresses the conditions and the effects of each transition. We define Petri nets more precisely in Chapter 3, as well as how we can extend this model to take into account time.

Another interesting feature of Petri nets, that we should make use of, is the ability to describe a net using a graphical syntax. In the remainder, we use a standard graph-like representation in which places are depicted with "circles", transitions with "boxes", and names and labels appear as decorations of these elements. For example, the DES for the *door* system described in section 1.2 could be modelled with the net in Figure 1.1.

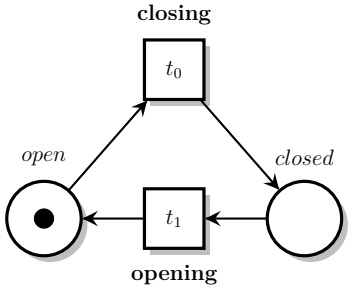


Figure 1.1: PN of a door

Finally, Petri nets also define an *algebra*, because it is possible to define a set of operations for combining nets together and inferring properties on the composition of several systems from the behaviour of each of them. One such operation is the *synchronous product* of Petri nets, sometimes also called *parallel composition*, that is essential to the study of this model [Cassandras 2009]. This concerns for the notion of *compositionality*, and the fact that it works well with labels, is what motivates (in part) our choice of PN for the specification of DES.

Since we are interested in the study of timed systems, we should consider an extension of PN, called Time Petri Nets (TPN), where we can also express constraints on the time needed before firing a transition. Basically, all transitions are associated with a timing constraint (a time interval). In order to fire, we add a new condition; the transition must stay enabled enough time to fulfil its timing constraint. We should see that time can restrict the set of possible behaviours of a system and, by doing so, it may greatly complicate formal verification.

In this work, we focus on the *diagnosability* of single faults in a TPN (see Chapter 6). The idea is to check if it is possible to infer that a faulty event—an unobservable event f—has occurred by looking only at the observable events in the system. We will also consider the problem of diagnosability for "patterns of events". Our approach to this problem is based on an extension of the notion of synchronous product that works well with TPN and the verification methods that have been developed for their analysis (typically the State Class Graphs of [Berthomieu 1983]).

1.5 Problematics

The main problematic of our work can be summarized by the following question: *how can we analyse the synchronous product of two TPN in an efficient way?*

To explain why this problem is complex, we need first to explain what are the two main problems that hinder the definition of a synchronous product between Time Petri Nets.

A first problem has to do with the use of a dense time model. In this case, we may have to deal with infinitely small time delays, which in turn may create infinitely many states and (time-)transitions in our systems. This problem has been tackled for TPN with the definition of *State Class Graphs* [Berthomieu 1983], in which the timing information is abstracted using system of inequalities between "time"-variables (or clocks). Actually the same problem occurs when we use a discrete notion of time, and when we have large time constants. In this case, even though the state space may be finite, we may still be faced with a scalability problem. In my work, I propose an extension of the notion of State Class Graph that will be useful for checking the diagnosability of TPN.

The second problem, and one of the main focus in my work, is the difficulty to define a composition operation between TPN that is "compositional"; meaning that it preserves the product of the behaviours of each net in the product. There exists some solutions on the composition of Time Petri Nets, see [Peres 2011] in particular, where the authors extend the TPN model with a new notion of priorities between transitions (see the notion of IPTPN that we describe in Section 3.3.2.2). A problem with this approach is that priorities add a lot of complexity when analysing the behaviour of TPN, and we would like to avoid adding them if not necessary. In my work, I will propose a new extension, based on an "internal" notion of product, to solve this problem.

1.6 Contributions

My main contribution is the definition of a new formal model, an extension of Time Petri Nets [Lubat 2019], that helps us solve the problems described in the previous section.

Our extension is obtained by integrating a notion of "product of transitions" directly in the model, with the idea that transitions with a common label must fire synchronously, but without syntactically "merging" them or their timing constraints together. This idea is more extensively explained in Chapter 4.

A motivation, and the main application of our extension to TPN, is to propose a direct extension of the twin-plant construction [Jiang 2001a] to the case of TPN, without any post-processing of traces (see additional information in Section 3.2). We use this method to decide the single fault diagnosability problem on TPN and to show that it can be simply extended to decide the diagnosability of more complex behaviours; what is commonly called a *pattern* of behaviours.

An advantage of our approach is that we can easily adapt it to several extensions of TPN: adding priorities between transitions; inhibitor arcs; capacity arcs; etc.

In a more schematic way, the contributions of this thesis can be described by the following itemized list of problems, for which we give solutions based on our novel product construction:

1. Define a notion of product between labelled TPN that preserves composability, meaning that the behaviour of the synchronous product of two TPN is exactly the synchronization of the behaviours of each net taken separately (and on their observable labels).

2. Define a good data structure and an algorithm for checking the diagnosability of a single fault in a TPN.

3. Define a method to check the diagnosability of untimed patterns in a TPN.

4. Define a method to check the diagnosability of timed patterns in TPN. This means that we can also add constraints on the date at which "faults" must occur. We give a positive answer in the case of a restricted category of timed patterns, that are interesting in practice. Our solution gives some indication on the high complexity of this problem in a more general case.

One last contribution of this thesis (contribution number 5) could be interpreted as an answer to the (natural) question "is your notion of product useful at something else than diagnosability ?". While our initial publication on the subject also includes "(observer-based) model-checking" as a potential application [Lubat 2019], we decided not to include this direction of research in this manuscript, since it is not that different from what we describe regarding diagnosability. We propose instead a chapter that describes how we can transpose our notion of product to the HIPPO framework [Hladik 2021], a specification language and a real-time execution engine built as an extension of FIACRE [Berthomieu 2008a] with executable tasks. We also mention a possible application to a problem known as *opacity*, that corresponds to another class of "observability properties", dual to diagnosability.

1.7 Outline of the Thesis

The thesis manuscript is decomposed into nine different chapters, including this one. We briefly describe the purpose and content of each chapter in the list below. To facilitate the reading of the manuscript, we recapitulate the contributions made at the end of each chapter, in a dedicated section called "summary".

- *Chapter 1*: This chapter presents a quick overview of the problems addressed during the thesis and of the context of our work. We introduce the notion of DES and the motivation behind our work.

- *Chapter 2*: This chapter presents the different models used and a brief overview of methods for the formal verification of systems. We make a focus on model-checking techniques, which corresponds to the approach followed in my work. The chapter also tackles the notion of diagnosability we use through this thesis.

- *Chapter 3*: Here we present the technical details regarding the syntax and semantics of Time Petri nets. We then provide a quick overview on the notion of products, first on TTS, then on TPN with some ad-hoc solutions. We conclude by presenting the notion of (discrete-event and continuous-time) state space graph, which are used for defining the semantics of TPN.

- *Chapter 4*: This chapter is devoted to the definition of Product TPN (or PTPN for short). We define our product operator and give the semantics of PTPN. We also describe a new kind of behaviour, called *timelocks*, that can occur with PTPN but not with TPN.

- *Chapter 5*: This chapter describes the notion of State Class Graphs (SCG). We start by going over the definition of "classical" TPN, then we show how it can be naturally extended to PTPN. We also discuss the differences between Weak and Strong SCG.

- *Chapter 6*: This chapter describes an application of PTPN (and State Class Graphs) to check the diagnosability of systems. After a brief overview of the problem, we describe the notion of *critical pairs* and the concept of *twin plant* in order to detect them. Then, we propose a method for the diagnosability of a single fault and finally an extension of this method for the case of untimed patterns.

- *Chapter 7*: This chapter focuses on experimental results and on the tool that was developed specifically to implement our different constructs and methods. We describe the different experimental tests and benchmarks we used to test the applicability of our methods. We use these experiments to compare performances between our approach, with PTPN, with an approach based on IPTPN, that is also new, but that uses verification tools that were already available at the start of my thesis. On the second section of this chapter we also test the scalability of our approach and focus on more complex benchmarks. Finally, we present a benchmark for the diagnosability of patterns.

- *Chapter 8*: Before concluding, we discuss the problem of checking diagnosability for a very specific example of timed pattern that cannot be addressed with the method defined previously. We also discuss about what is needed in order to apply this method on more general patterns and how our approach could be automated.

- *Chapter 9*: We use the conclusion as an opportunity to discuss two extensions that are currently being investigated: first concerning another notion

of observability, called *opacity*, and another concerned with an application of synchronous product in the context of the HIPPO execution engine.

Chapter 2

State of the Art

Although we focus on the property of *diagnosability* in our work, it is interesting to look at other verification problems related to the study of DES. We use this as an opportunity to study the state of the art about the modelling of DES and the different formal verification techniques that could be applied.

Our main technique of interest is *model-checking* [Clarke 1981, Queille 1982], which is the approach we used during the PhD thesis. *Model-checking* is a set of automated techniques to check whether a systems meets its requirements. This verification method can also return an counter-example (a scenario) in the case when one of the requirements is not met.

These techniques are useful during the development of *safety critical systems* as described in Chapter 1. We give more details in Section 2.3 of this chapter. Model-Checking, as indicated by its name, requires models. We start by an overview of some formalisms that can be used to define a model, and their semantics, in Section 2.1. Next we give some state of the art on the formal verification for DES in Section 2.3.

2.1 Modelling of DES

This section is largely based on *Introduction to Discrete Event Systems* by Lafortune and Cassandras [Cassandras 2009]. As a first approximation, a DES is a *model* of a system in which we can represent the possible states using a finite set of elements $s_0, s_1, s_2, ...$ and such that the current state of the system can change only using one of a finite number of events. To better define the concept of DES, we need to give more details about the two notions of *events* and *discrete*.

An *event* may simply be identified as a specific action (e.g. closing a door) leading to a possible change in the state of the system (e.g. the door is closed). It may occur spontaneously or when some conditions are all met (such as conditions on the duration of an action for example). We should use the notation E to refer to the finite set of possible events and we use e_i to refer to elements in E.

The word *discrete* refers to the fact that the dynamics of the system is made up of events. In a DES, state only changes at certain points in time through instantaneous transitions. At each "moment" we can either select a particular event (if its conditions are met) or we can select a *null-event* ε, to simply let time elapse, without changing the discrete state of the system. In our case, the set of states (with the initial state and all the reachable states) is called the *state-space*.

It is a discrete automaton in which the transitions are tagged with events in E. We also say that we have an *event-driven system* [Cohen 2000].

Time, just like events and states, is also generally thought of as discrete in this kind of approaches. It means that, just like with a mechanical clock, time increases in a discrete way (like the tick of a clock) and it can be expressed as an integer number in a certain, fixed unit of time (which can be arbitrarily small). As said before, in a DES, the evolution depends on the events of the system, which can be active or not regarding the time of the systems or some other conditions (for instance a probe detecting a change in the environment). In this context, time can be a useful information since it can constrain the occurrence of some sequence of events; for example to limit the duration between the occurrence of a signal and the raise of an alarm.

In summary, DES have three main characteristics:

- The set of states is a discrete set.
- The current state can change only depending on events (which can be *null*).
- Time and other continuous data types can be added (a DES can be augmented with more complex data) and may be used as conditions in the choice of events.

A word about Time : In DES, Time, as said before, is a useful information. Some events may only occur after a certain amount of time. We usually store time in two different kinds of data, *clocks* or *domains*. We will come back to these notions later, this is why I should try to give some intuitions about the differences between clocks and domains here:

- *Clocks* are simple counters which count how many units of time have elapsed since they have been last reset. Usually, each event has its own clock which is reset after its occurrence.
- *Domains* are systems of inequalities, updated when events occur. They usually represent constraints on (virtual) clocks, such as upper and lower bounds on the time the system can stay in a given state.

A word about semantics : we use the notion of *semantics* as the method used to describe the possible behaviours of the DES. A central notion in this context is the one of executions; meaning sequences of events that can be observed in a run of the system. We can combine time and events to have a better description of the behaviour, which leads to a notion of timed sequence, such as:

$$(e, 3), (\varepsilon, 3), (e, 0), \ldots$$

We call this sequence a (timed) *trace*. A trace describes a behaviour of the system as follows: e occurs at 3 units of time, 3 units of time elapse and then another event e occurs immediately after that.

Traces of the form given above, where events are associated with their occurrence date, describe what is called a *signal* semantics. Another possible semantics is based on *timed words*, where we consider "time elapsing" as a special kind of event, such as: (for more information about semantics of traces see [Popova 1991]):

$$3, e, 3, \varepsilon, e, \ldots \qquad \text{or} \qquad 3, e, 1.2, 1.8, \varepsilon, 0, e, \ldots$$

With the usual convention that a sequence of two delays θ, θ' can always be replaced with a single delay (of value $\theta+\theta'$), and that null delays can be omitted or arbitrarily added.

These two choices of semantics can lead to slightly different results when we define the properties of a system. It is the case, for instance, when studying the decidability of the model-checking problem for some timed temporal logics. But this will not be the case with the properties studied in my work and we will stick with the timed words semantics for the remainder of this work.

A trace consisting of no events is called the empty trace and is denoted by ε and the length of a trace is the number of events contained in it (counting multiple occurrences of the same event separately). By convention, the length of the empty trace is zero.

In the following, we give three examples of formal models (equipped with a notion of time) that can be used to specify DES.

2.1.1 Henzinger's Timed Transitions Systems

Our first formal model is the Timed Transitions Systems (TTS) defined by Henzinger et al. [Henzinger 1992], which we will abbreviate by TTS-Henzinger (or even simply by H-TTS) to avoid possible confusions. A H-TTS is composed of a set of states and a set of discrete events, both evolving depending on the event occurring or time elapsing. State can change based on two rules:

- An event e occurs and changes the current state.
- Time elapses; the discrete state does not change but we update the time spent in the current state.

H-TTS is amongst the simplest models for timed DES and does not add more rules than the one given in our general definition of DES, at the beginning of the chapter. It is possible to extend H-TTS with a notion of *labels* and to define a Labelled Transitions System as a result, see for example [Keller 1976] where this notion is called "Named Transitions Systems".

We give an example of H-TTS in Fig. 2.1. We use the semantics given in [Henzinger 1992] to explain the behaviour of this system.

In a H-TTS each event has its own timing constraint, defined using an interval, that indicates at which times the event can occur. Let's take the example in figure 2.1:

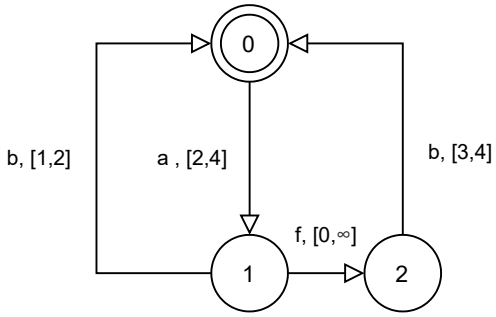


Figure 2.1: Example of a H-TTS

- States: the system has only three different states, $\{0, 1, 2\}$, with 0 the initial state.

- Events: the system has four possible events/transitions, defined by the initial state, the resulting state, and a label (in this case one of a, b or f).

- Timing Constraints: give the lower and upper bound for the duration one can stay in a state before "firing" an event.

This H-TTS has four possible events with three respective labels: a, b and f. In the following, we will often take the convention that f is the label of *faults* (an example of unobservable action). Some of the events are constrained by timing constraints (which are reset every time you leave the previous state). For instance, after arriving in state 0, the system must transit to state 1, with event a, after a time θ in the interval $[2, 4]$.

H-TTS is a very simple model that is interesting for historical reasons, but also because it is very close to the semantics that we should use to describe the behaviour of our systems. It is basically a Finite State Machine (FSM) with labels on the transitions, extended with timing constraints. We give two other examples of timed models: Timed Automata and Time Petri Nets. The first adds the possibility to have multiple clocks (and not only the time since we entered a state). It also adds the possibility to express constraints and "invariants" using expression over clocks.

Time Petri nets is to Petri nets what H-TTS are to FSM. We should see that the addition of timing constraints on (Petri) transitions is not totally straightforward; in particular because it complicates composition. We give more details about each of these models below

Both models can adequately describe the behaviour of timed DES and provide a framework to analyse and implement DES efficiently. In each case, it is also possible to find verification tools that can be used to automatically prove properties about a model.

2.1.2 Timed Automata

Automata are the most basic form of DES models (almost like a TTS). Timed Automata (TA) were first introduced in [Alur 1990]. It is an extension of FSM in which transitions are decorated with expressions over clocks and where the state is a pair consisting of the current, discrete state, as well as a valuation for the clocks (we use Q for the set of clocks).

In a timed automaton, each event has a guard (a constraint over clock value) which indicates when such event can be fired and a set of clocks to be reset when the transition is fired. Timed Automata are defined by their set of states, their set of events (labelled or not), their initial state and their set of clocks. In a TA, like in the H-TTS model, the evolution of the system still depends on two main rules:

- An event e occurs and changes the state.
- Time elapses and the set of discrete clocks evolves. A specific condition of TA, that makes formal verification feasible, is that all clocks evolve (increase) with the same constant rate.

We can explain the semantics of TA using simple examples.

Example 1: Our first example is a Timed Automata (figure 2.2) that corresponds exactly to the H-TTS given in Figure 2.1. This TA is composed as follows:

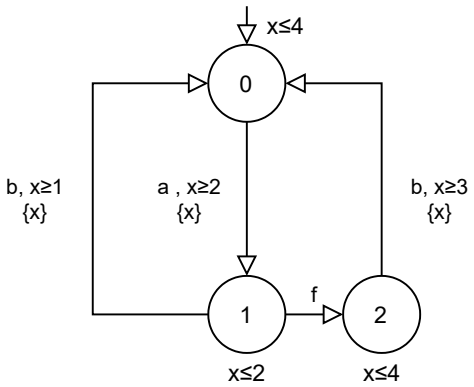


Figure 2.2: Example of a TA

- States: $\{0, 1, 2\}$, where 0 is the initial state..
- Events: $\{0$ to $1(a), 1$ to $2(f), 2$ to $0(b), 1$ to $0(b)\}$.
- Clocks : a single clock, x, that is reset each time we follow a transition after occurrence (this is the meaning of "inscription" $\{x\}$).

The difference is in the timing constraint. In TA, timing constraints can be found on events and on states, which are called *invariants*. Invariants are not from the classical TA model. Here, for example, you have to leave the state 0 before 4 unit of time and you can only go through the first event after an unit of time (which is exactly the behaviour of our previous example).

Example 2: The dissociation between states and clocks generates new behaviours. This is the case in our second example, Figure 2.3. In this TA, the clock y is not reset until the transition from 1 to 2. If the system does not instan-

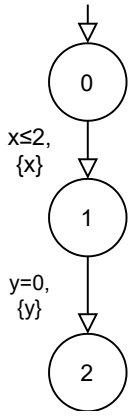


Figure 2.3: Example of a TA with a possible Timelock

taneously go from state 0 to state 1, it is not possible to go from state 1 to state 2 because of the impossible solution between the invariant (on y) and the condition on the event.

This behaviour creates a locked situation called a *deadlock* or more precisely a *timelock* in this situation.

As you can see, TA are almost like TTS in their construction. They are intuitive, easy to use, easily combined and there is a large body of research on methods for analysing them.

2.1.3 Time Petri Nets

Our final example of timed model is an extension of Petri Nets (PN) [Petri 1962] with timing constraints over the transitions. We will focus on this model, due to [Merlin 1974], that we simply call Time Petri Net (TPN) in the following. We should not consider other models, such as Petri nets with timing conditions over the places or the arcs. See [Boyer 2008, Bérard 2013] for more information about how to compare these different timed extensions of Petri Nets.

In a Petri net, events are associated with *transitions*. For an event t to occur (we say that transition t is *fired*), the condition is that there are enough tokens in the places connected as input to t. We say also in this case that the transition is

enabled. After firing a transition, we remove tokens from the input places and add tokens to the output places.

In a TPN, we also add a condition on the time the transition stays enabled before firing it. Like in the transition of a H-TTS, this condition is a time interval. In practice, it means that for every enabled transition we need to have either a "timer", or clock, that captures the delay since the transition was enabled (without interruption); or a "firing domain", that captures the possible dates in the future at which the transition can fire.

The choice of clocks versus domains is not really meaningful when defining the semantics of TPN, but they will later lead to two different ways of abstracting the timing behaviour. In particular, the original approach defined in [Berthomieu 1983] is based on firing domains and has resulted on the definition of a notion of "Weak" State Class Graph (SCG). This is the approach we will follow in the next chapter. By contrast, the use of clocks lead to a notion of Strong SCG, that we should also address in the following.

The condition on time for firing an enabled transition, t, is that t can be fired immediately: its firing domain includes 0 or, dually, the value of its timer is in the time interval associated with t. We also need to define when the timer associated with a transition is updated/reset. We will rather say that the transition is *reinitialized* when its timing constraints are reset and, in the opposite case, we say that the transition is *persistent*. Basically, a transition is persistent when it was not just fired, or when it did not suddenly become enabled as the result of firing a transition. We also have several possible choices in the semantics of reinitialization, see e.g. [Bérard 2013]. We will define more precisely our notion of persistent and reinitialized transitions in Chapter 3.

Like with the H-TTS and TA models, the state of a TPN evolves depending on two main rules:

- A transition t is fired and changes the markings of places. In a TPN, the firing of a transition is immediate (takes 0-time) and atomic. In particular, all transitions that are enabled but not persistent are reset at the same time.

- Time elapses and we update the firing domains of all enabled transitions.

While this is globally similar to the behaviour of H-TTS and TA, we can stress some important differences concerning the rule for letting time elapse. These differences will have an impact on some of our results.

Since no transitions are fired when time elapses, the set of enabled transitions stays the same. This is one difference with TA, since the expression associated with an event in a TA may change value when clocks increase. Another difference is that the timing constraints of enabled transitions are strict; meaning that it is not possible to wait for a duration $\theta > 0$ if, for some enabled transition t, the value of θ is not in the firing domain of t. In short, it is not possible in a TPN to gain or lose events by simply letting time elapse.

Example of TPN: We give, Figure 2.4, an example of TPN similar to our first example of TA (see Figure 2.2).

You may notice that this is a very restricted example of TPN since, at any given times, there is exactly one token in the whole net. We chose this example to underline the similarities between the three models considered so far. More precisely, we are in a special case, of a very restricted class of nets, called *Marked Graphs*, such that every place has exactly one incoming and one outgoing arc. We consider more complex examples later in this chapter, with synchronizations and possible conflicts between several places; see also our examples used in the "Experimental Results" (Chapter 7).

Another difference between automata and Petri nets is the ability, in the latter, to model an unbounded number of resources; this is the case when a sequence of transitions may strictly increase the markings of a net. We should not study the case of unbounded nets in our work, even though it will not change most of our results (except when we consider the complexity and/or decidability of some of our methods).

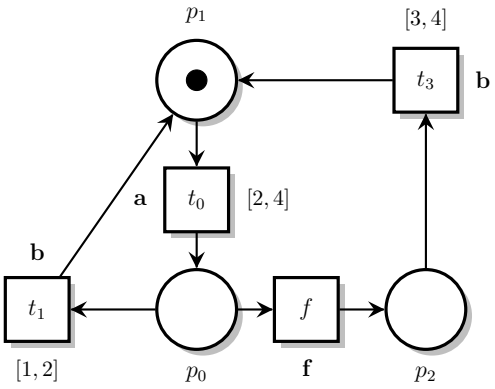


Figure 2.4: Example of a TPN

2.1.4 Synchronous Networks of DES

It is possible to enrich models for DES by defining a notion of "networks of models", in which different components can synchronize on common observable events. An emblematic example of such model, in the untimed case, is Arnold and Nivat's synchronized network of automata [Arnold 2002].

We can define an abstract operation of product between DES as follows, which can be interpreted as some kind of Cartesian product between the behaviours of components. This definition can be extended to more than two systems and/or with a notion of synchronization vectors.

Definition 1 (Synchronous product of DES)**.** *Given two DES, N_1 and N_2, with observable events $E_{o,1}, E_{o,2}$, the synchronous product $N_1 \| N_2$ is a DES with set of*

states $S = S_1 \times S_2$, events: $E = E_1 \cup E_2$; and clocks : $Q = Q_1 \cup Q_2$, such that (s_1, s_2) can transition to (s'_1, s'_2) in $N_1 \| N_2$ if and only if one of the following three conditions occurs:

- *s_1 transitions to s'_1 in N_1 and s_2 transitions to s'_2 in N_2 on a common, observable event.*
- *s_1 transitions to s'_1 in N_1 on an event that is not in $E_{o,2}$ and $s'_2 = s_2$.*
- *s_2 transitions to s'_2 in N_2 on an event that is not in $E_{o,1}$ and $s'_1 = s_1$.*

In the following, we use the product of two copies of the same system, $N \| N$, in order to check properties about pairs of executions that have the same observable events; in the same order and at the same dates. This is the reason why this product operation is central in my work.

While we can define the synchronization product in an abstract way, it behaves quite differently depending on the actual choice of formalism we use to model the DES. We review the situation for the three formalisms we listed in the previous sections.

In the case of Timed Automata, a network of TA can always be "compiled into" (interpreted as) a single TA by statically computing all possible interactions (at the cost of increasing the size of the model). In this case it is necessary to consider the conjunction of invariants on each state, and the conjunction of the clock expressions on the synchronized transitions. This is the reason why most formal definitions of TA avoid the complexity of directly handling networks of TA.

The same is not possible for H-TTS; since different TTS (automata) in a network may enter their new state at a different time, we may need to keep track of several "timing intervals" to check which vector of transition can fire at a given date.

The situation with Petri nets lies in between these two cases. In the absence of time constraints, it is always possible to replace a network of nets by simply fusing copies of transitions that have the same, observable event. This is a conventional operation on Petri nets, also called *synchronous product*. (We define this operation more formally in the next chapter.) We can illustrate the synchronous product between two Petri nets using a simple example, see Figures 2.5 and 2.6. Basically, we create new transitions by fusing together copies of transitions that have the same label.

One of the main contribution of my work is to define an extension to TPN that allow to build the synchronous product of two or more systems together.

2.2 Comparing Expressiveness

We have briefly described three possible formalisms to specify timed DES, with TA and TPN taking the upper hand. For the sake of brevity, we did not mention other possible choices, such as: timed extension of process calculi, for instance

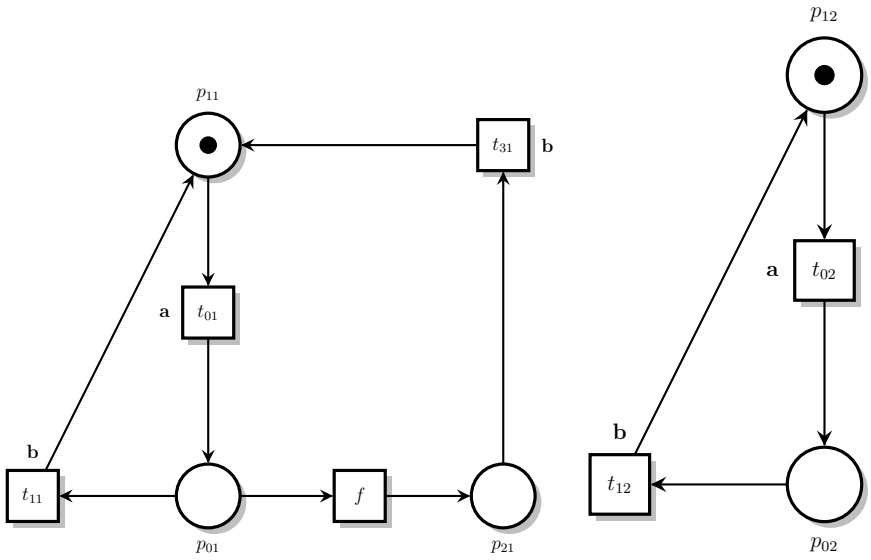


Figure 2.5: Two PN before their product

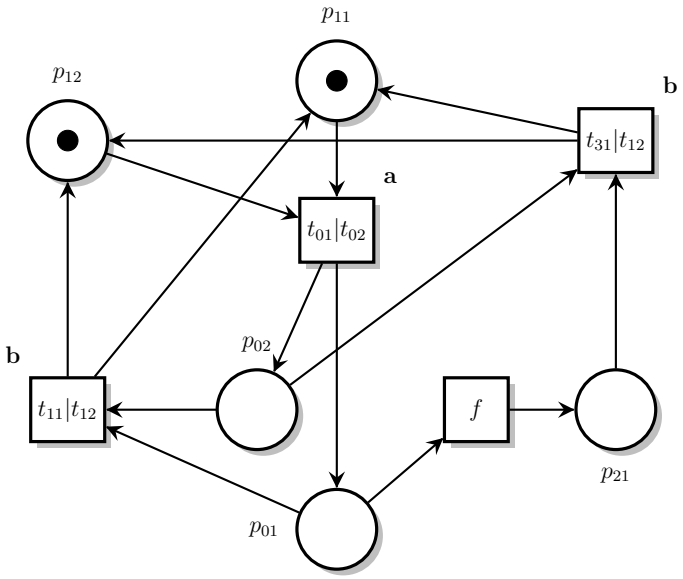


Figure 2.6: Synchronous product of the two PN in Figure 2.5

tCCS [Moller 1990]; synchronous languages; etc. A natural question to ask is: "Which among these models is better ?"

An argument often used in favour of TA is that of conciseness and "primitiveness", since all the (discrete) states in the TA are explicitly given in its definition; whereas with TPN we must work at two different levels: places in the syntax of the net and markings in its "state graph" (its semantics). Put differently, assuming we want to study the impact of adding time to a formal model, we may as well choose to start with the most basic formalism.

This is not a clear cut point. In practice, tools and models based on TA rather use networks of Time Automata, with synchronization on channels. This is the case, for instance, with the tool Uppaal [Larsen 1997]. Comparatively, the notion of synchronization is intrinsic of TPN. Moreover, TPN do not use a separate (syntactical) category for clocks and do not need to add separate notions for (time) invariants and clock resets.

A more formal way to approach the problem is to study the question of *expressiveness* between models; meaning whether it is always possible to "simulate" or "interpret" a system expressed using a model in language A, using a model in language B. Other features that can be taken into account here are the complexity of computing the interpretation from a model; the size of this interpretation; and the notion of equivalence used to compare models.

It is known that TA are strictly more expressive than TPN [Lime 2003a, Bérard 2008] and there are methods to build an equivalent TA from a TPN. In this context, timed bisimulation is often considered as the right notion of equivalence but, in our case and since we are mainly focused on properties about traces, language equivalence would be a more interesting choice. We will use these results in more details in Chapter 4 (see also the equivalence results given in [Lubat 2019]).

Working with a less expressive model may actually have its advantages. In particular, it could be the case that some classes of problems are easier to deal with using TPN rather than TA. Also, irrespective of the "theoretical complexity" of a problem, it may be the case that some problems are easier to handle, in practice, using tools designed for TPN rather than with TA. For instance, we have not been able to adapt our approach for checking diagnosability over TPN using tools developed for TA, such as Uppaal (https://uppaal.org/) or TChecker (https://www.labri.fr/perso/herbrete/tchecker/), due to limitations of these tools. In brief, none of these tools support the verification of a specific LTL formula, of the kind $\Diamond dead$, meaning that every execution must eventually end with a deadlock or a timelock. (We give more information on these notions later on.) Yet this is precisely the kind of property we use with the "twin plant" approach to check diagnosability [Boussif 2016].

In this PhD thesis, I did not attempt to prove deep theoretical results about the limitations of using TPN for checking diagnosability, when compared with TA. The focus is more on developing pragmatic methods for checking diagnosability, practical enough to serve as a basis for implementing a dedicated verification tool. As a result, one outcome of my work is to provide some support to the idea that

TPN offer a good compromise between expressiveness (it is good enough to model the systems we need to check) and performance of the verification when it comes to analysing diagnosability. This is in line with the observations made in [Lai 2008]—a paper about diagnosis, not diagnosability, and in the untimed case—where the authors conclude that while automata-based approaches are more general, Petri-based approaches present "significant advantages in terms of computational complexity".

Now that we have presented the two main models for our DES representation, and a quick comparison between them, we will focus on the notion of *verification*.

2.3 Formal Verification and Model Checking

Formal verification is a field of computer science concerned with developing techniques for checking that a system, or protocol, satisfies the requirements defined by its designers. These requirements can be expressed using informal, natural language specifications. But more formal approaches are possible, like for instance the use of modal logics (see for instance our use of temporal logic in Chapter 6) or the use of a "golden model", that is a formal model representing the good, expected behaviours of the system. In the latter case, verification often amounts to checking equivalence between the *implementation* and *specification* models.

Checking the diagnosability property on TPN will require a slightly different approach, in which we analyse the common observable behaviours in two copies of the same model.

There is a large collection of verification techniques, that can be classified based on the methods used to abstract the semantics of the system. We can cite *deductive techniques*, in which the abstract behaviour is defined from requirements expressed as inductive properties—these properties must be defined manually by the user and their proof may rely on the use of proof assistants—; *static analysis*, in which the abstract behaviour is automatically derived from the actual system, for instance from programming code, using predefined approximations (possibly using some inputs or parameters from the user); and *model-checking*, where the user must provide a model specification of the system and the property (for example a finite automaton or a Petri net).

2.3.1 Model Checking

The main verification method used in my work is Model-Checking. It is a collection of automated techniques first introduced independently by Joseph Sifakis and Jean-Pierre Queille [Queille 1982] and Edmund M.Clarke and Allen Emerson [Clarke 1981]. In the most basic way, model checking operates on a discrete representation of the model's "state space", usually described as *Kripke structures* or *Labelled Transition Systems* (LTS) [Clarke 1999]. Both cases are graph-like data structure, with a notion of transition relation between states and of initial state. The only difference between these models is that, in Kripke structures, information is stored on the states/nodes, whereas it is stored on the transitions in a LTS.

More formally, a Kripke structure is equipped with a function that associates sets of (atomic) properties to each state/node in the graph, whereas a LTS is equipped with a function that associates properties to transitions.

In my work, I will use a combination of LTS and Kripke structure—a Labelled Kripke Structure—in which states are labelled with markings of a Petri net and transitions are labelled with (vectors of) transitions, or labels. We should also work on an abstracted version of the state space, where we eliminate timing information and time elapsing transitions. This is exactly the notion of *State Class Graphs* (SCG) introduced by Berthomieu et al. [Berthomieu 1983] (see Chapter 5 of this thesis).

2.3.2 Temporal Logic

An important class of model checking methods rely on the use of temporal logic to specify the property that we want to check on a model. Temporal Logic is a special case of modal logics that includes operators to express constraints about the order in which events can occur. It was defined by Arthur Prior in the 1950s [PRIOR 1957] and has proven to be a good candidate to express properties about concurrent, reactive systems.

We can distinguish two main branches of temporal logics, depending on how they interpret "executions" in a system [Clarke 1988]. In "linear" temporal logics, the evolution of a system is treated as if each state has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences (traces) and describe the behaviour of single executions of the system, independently from each other. In "branching" temporal logics, such as CTL, we consider that states may lead to various possible futures. Hence, in this case, formulas are interpreted over infinite computation trees, describing the possible nondeterministic behaviour of the system.

There exist many different temporal logics, that can be compared upon their *expressiveness* (which properties can be specified ?) and *complexity* (how complex it is to check a given property ?). For the sake of brevity, we will only mention very briefly two examples, LTL and CTL, using only examples. You can find many textbooks, for instance [Clarke 1999], that provide comprehensive information on this topic. We will also mention "timed" extensions of temporal logics, where it is possible to also express constraints on the duration of an action, or the time separating the occurrence of two events. I will give a single example of such timed, temporal logics, namely Metric Interval Temporal Logic (MITL).

Linear Temporal Logic (LTL) is the archetype of linear temporal logics. It was first presented by Pnueli in 1977 [Pnueli 1977]. It is an extension of propositional logics with two modalities, *finally* ($\Diamond$) and *globally* ($\Box$), to express conditions about the future in an execution path. Basically, given an execution path and a formula ϕ of LTL:

- *Finally* ϕ (or $\Diamond\phi$) holds if ϕ holds sometimes in the future of the path.

- *Globally* ϕ (or $\Box\phi$) holds if ϕ holds everywhere, on the entire subsequent path.

A formula is valid for a system if it holds for every *maximal executions*; meaning all infinite executions or executions that end with a deadlock.

We can take the example of a LTL formula about events occurring in our main TPN example (see Figure 2.4).

$$\Box\,(a \Rightarrow \Diamond\,(b \vee f))$$

This formula expresses the condition that, every time an event with label a occurs, it must be the case that an event with label f or b occurs in the future. This formula is satisfied on our particular example of nets. This is an instance of "leadsto" formula, a pattern that will arise in our work on diagnosability.

Note that we can also define versions of LTL that deal with states, instead of events, or that deals equally with both. In point of fact, I will use a LTL model-checker in my work, called *selt* [Berthomieu 2008b], part of the Tina toolbox, that combines both states and events in LTL.

Computational Tree Logic (CTL) is a branching-time logic first introduced in 1986 [Clarke 1986]. CTL is a modal logics that includes both "path operators", that talks about occurrences of events given one execution π (like in LTL), and path "quantifiers", that constraint some or all paths starting from a given state s. Unlike with LTL, where a formula is true if it is satisfied on all (maximal) executions of the system, we define the satisfiability of a CTL formula ϕ on states. Hence, in the same system, we can have states that satisfy and other that dissatisfy the same property.

- *All* ϕ (or $A\,\phi$) holds for an event s if ϕ holds on all the path starting from s.
- *Exist* ϕ (or $E\,\phi$) holds for an event s if there is at least one path starting from s where ϕ holds.
- *Globally* ϕ (or $G\,\phi$) holds for a path π if ϕ hold everywhere on π.
- *Finally* ϕ (or $F\,\phi$) holds for a path π if ϕ eventually holds (somewhere) in a state of π.

With our choice of syntax, a CTL formula equivalent to our previous LTL example is:

$$A\,G\,(a \Rightarrow A\,F\,(b \vee f))$$

LTL and CTL are not comparable, in the sense that some properties can be expressed in one logic but not the other. For instance, the LTL property $\Diamond\Box\,a$ holds if event a always occurs infinitely often. This property cannot be expressed in CTL.

In the case where the property is expressible in both logics, one may be inclined to use tools for LTL instead of CTL. A pragmatic reason is that there exist efficient

algorithms for LTL model-checking, that typically require less memory than with CTL, and that can benefit from on-the-fly computations. The situation is even more conclusive in the case of TPN, when using SCG approach. Indeed, while it is possible to compute a SCG abstraction that preserve branching, the result is often much more complex than with SCGs that only preserve traces (and are therefore enough when checking linear time properties). In my work, I will show that diagnosability can be reduced to the problem of checking a "leadsto" property, that is a class of formulas expressible in both LTL and CTL. Also, instead of reusing a general LTL model-checker, I will define a specific model-checking algorithm, specialized for this single formula.

Metric Interval Temporal Logic (MITL) is a fragment of Metric Temporal Logic (MTL), a logic in which the temporal modalities are replaced with time-bounded versions, such as $\Diamond_{[1,3]}\phi$, that constrain the date at which the event must occur [Henzinger 1998]. In MITL, we add the requirement that every time interval in a time-bounded operator must not be punctual. Hence MITL rules out the possibility to enforce that two different events must occur at the same date. One example of MITL formula is $\Box\Diamond_{[0,1]}a$, meaning that event a must occur at least once every unit of time.

We will not use or refer to timed logics in the following, but mentioning MTL and MITL gives us the opportunity to make some interesting remarks. First, contrary to LTL and CTL, the model-checking problem for full MTL is undecidable [Ouaknine 2005]. On the opposite, while MITL is decidable, its theoretical complexity is very high; it is EXPSPACE, whereas model-checking LTL is in PSPACE. The decidability of MITL with respect to MTL is an indication that checking synchronicity between events is difficult. Hence an indication that the problem I address is difficult. Also, the complexity of MITL indicates that it may be a better idea to use "untimed" verification methods on the SCG (an abstract semantics of TPN where timing information has been discarded). Finally, the MITL model-checking algorithm is complex and there are no mature verification tool that implements it (see MightyL [Brihaye 2017] for a prototype implementation). This is another reason to avoid using timed logics.

2.4 Fault Diagnosis

Fault diagnosis play an essential role in the safe operation of industrial systems. Diagnosability was introduced in the context of DES by *Sampath* [Sampath 1995], where the authors define the notion of *diagnoser* using a property over the observable language of a system. An extension of this problem in the context of timed DES is due to Tripakis [Tripakis 2002], that defines a method for checking the diagnosability on Timed Automata. It took several years to see the problem addressed in the context of Time Petri nets [Basile 2018, Liu 2014, Ghazel 2009], which may indicate that the problem is more complex in this case.

First and foremost, we have to define two different notions, *diagnosis* and *diagnosability* [Lin 1994]. Diagnosis (or *online diagnosis*) is the method performed to detect and localize the cause of a fault. Diagnosability (or *diagnosability analysis*) is the ability to detect and locate any fault within a finite delay after its occurrence. Diagnosability is analysed offline.

We mainly focus on diagnosability in this work and also focus on the use of labels to define observable events. The properties of diagnosability can also be studied using stochastic models [Bérard 2017]. However we will not address this approach in my thesis.

2.4.1 Diagnosability for DES

Diagnosability was first introduced as a property over the accepted language of automata [Sampath 1995]. In this work, the authors give necessary and sufficient conditions for diagnosability and introduce a notion of *diagnoser*, a model which is mapped on the online observation of a system in order to detect the occurrence of a faulty event. Since the diagnoser-based approach has to check all the states, it suffers from state explosion problem.

This definition of diagnosability can easily be transferred to Petri nets [Ushio 1998], where the net marking is observable and all transitions are unobservable. In this paper, a simple diagnoser and sufficient conditions for diagnosability are proposed.

In [Jiang 2001b], an algorithm based on the synchronous product (or *parallel composition*) of a DES with itself, called a *twin-plant*, is proposed. The idea is to check properties by comparing different behaviors in the same DES. In [Yoo 2002] a comparable polynomial-time algorithm for deciding diagnosability is presented. The idea here is to synchronize two copies of the DES, one including the faulty behaviour and the other without any faults. The product of this modified copy and the original is called a *verifier* and may be simpler to analyse than the twin plant. In each case, an algorithm is proposed to decide diagnosability based on finding specific "cycles", or infinite behaviours, in the execution of the system. The system is diagnosable when no such cycle can be found.

In [Xue 2004], the authors compose a net called *verifier net* to analyse the diagnosability of PN. The idea is to check the verifier net, with a reachability analysis to conclude on the diagnosability.

In all these previous works, We can identify two main variants of the diagnosability problem: *K-diagnosability* and Δ-*diagnosability*.

- *K-diagnosability* is a qualitative analysis where K is an (integer) bound on the number of events that can occur in a system between the occurrence of a fault and its detection [Basile 2012].

- Δ-*diagnosability* is also a qualitative analysis where Δ is a bound on the time needed to detect the fault [Tripakis 2002].

Note that we focus on the diagnosability of "permanent faults", meaning faults for which no recovery is made. A notion of intermittent faults also exists, see for example [Contant 2004],[Jiang 2003] or [Boussif 2021].

2.4.2 Diagnosability in the Presence of Time

Diagnosability analysis between *timed* and *untimed* models are quite different. The information added by the presence of time can be useful to detect faults, for example because we find inconsistencies in the date at which some event occurs. This means that a system may not be diagnosable if we disregard timing information but may become diagnosable when we consider time. However, the addition of time also brings an extra layer of complexity when analysing a model.

Some works have addressed the diagnosability of TPN [Liu 2014, Wang 2015, Basile 2017]. While they propose substantially different methods, they all rely on a variation of the SCG construction of [Berthomieu 1983].

A notion commonly used is the notion of *critical pair* [Jiang 2001a], meaning a pair of maximal executions in the model, that have the same observable, at the same dates. A pair is critical if one execution has a fault and not the other; and a system is diagnosable if it has no critical pairs. The *twin-plant* method of [Jiang 2001a] is representative of this group. The drawback of this approach is that we may have more states in the twin-plant than in the system. An advantage is that this method is conceptually simple.

The approach in [Basile 2017] starts by building a Modified SCG that over-approximates the possible (timed) executions of a system. The system is diagnosable if no critical pair is found at this point. Indeed, time can only limit executions, not add new behaviour. If a candidate critical pair is found, it is necessary to solve a number of Linear Programming problems (LPP) to check whether this scenario is feasible (whether it is possible to find consistent times for the occurrence of events). This approach has several limitations, in particular, it may require to solve a large number of LPP.

In [Liu 2014], the authors define a notion of Augmented State Class (ASC) graphs, which are SCG augmented with diagnosability information, and use a method to split time intervals in order to only keep deterministic paths in the ASC graph. The interval splitting phase may create a large number of new active states that can lead to a state explosion problem. The approach in [Wang 2015] relies on a combination of SCG and an enumeration of all the firing sequences between active states.

In [Pencolé 2021] the authors define the notion of pattern and the diagnosability of this new behaviour in a TPN. This notion of pattern is more explored in the Section 6.4 of this thesis.

Our approach is quite different and relies on the use of the SCG construction for an extension of TPN that integrates the notion of "twin-product".

2.5 Summary

This Chapter introduced most of the state of the art concerning my work. To summarize:

- *Discrete Event Systems* are model-based description of real systems that describes its behaviour. We mainly focus on *Petri Nets* to model DES in this work.
- *Formal verification* is a domain of computer science that describes methods for *checking* (or *proving*) that a system follows correctly the requirements defined by its designers.
- *Properties* used during formal verification can often be expressed as a constraint over the language of DES. Our approach relies on checking properties over the intersection of two or more languages.
- *Diagnosability* (or *diagnosability analysis*) is the ability to detect and locate any fault within a finite delay after its occurrence. It is a property which can be checked via the intersection of language using a method known as the *twin-plant construct* [Yoo 2002].

In Chapter 3, we focus on TPN and introduce most of the technical details that we need to describe our extension of TPN.

Chapter 3

Time Petri Nets and other Technical Background

In this chapter we present Time Petri nets model. This formalism is classically used to model DES with time. First, we introduce all the basis of the TPN models and specifically timing constraints. We then quickly focus on the notions of execution and trace which are used to define equivalence between models. We also introduce the product of TPN and different ways to process it.

3.1 Definition

A *Time Petri Net* (TPN) is a Petri Net where each transition t is decorated with a (static) time interval $\mathbf{I}_s(t)$ that constrains the time at which it can fire. A transition is still enabled when there are enough tokens in its input places as for a classical net. Once enabled, transition t can fire if it stays enabled for a duration θ that is in the interval $\mathbf{I}_s(t)$. In this case, t is said *time enabled*. We can define more formally a TPN.

Definition 2 (Time Petri Net)**.** *A Time Petri Net (TPN) is a tuple* $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ *in which:* $\langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$ *is a Petri Net with* P *and* T *the set of places and transitions;* $\mathbf{Pre}$, $\mathbf{Post} : T \to P \to \mathbb{N}$ *are the precondition and postcondition functions;* $m_0 : P \to \mathbb{N}$ *is the initial marking; and* $\mathbf{I}_s : T \to \mathbb{I}$ *is its* static interval function *(with* $\mathbb{I} = \mathbb{Q}_{\geq 0} \times (\mathbb{Q}_{\geq 0} \cup \{\infty\})$ *for the set of all possible time intervals).*

To simplify our presentation, only the case of closed intervals of the form $[l, h]$ or $[l, +\infty[$ is considered. Moreover, for a time interval $\mathbf{I}$ its lower bound is denoted $\downarrow\mathbf{I}$ and its upper bound $\uparrow\mathbf{I}$. For a transition t with its static interval function $\mathbf{I}_s(t)$, its earliest firing time is denoted $\alpha_t^s = \downarrow\mathbf{I}_s(t)$ and its latest firing time $\beta_t^s = \uparrow\mathbf{I}_s(t)$, *i.e* $\mathbf{I}_s(t) = [\alpha_t^s, \beta_t^s]$.

In the following, it is considered that transitions can be tagged using a countable set of labels $\Sigma = \{a, b, \dots\}$. The special constant ε (not in Σ) is also distinguished for internal, silent transitions. A global labelling function $\mathcal{L} : T \to \Sigma \cup \{\varepsilon\}$ that associates a unique label with every transition. It is assumed that there is a countable set of all possible transitions (identifiers) and that different nets have distinct transitions. The *alphabet* of a net is the collection of labels (Σ) associated

with its transitions.

Example The figure 3.1 uses the same example as the one used in Chapter 2.

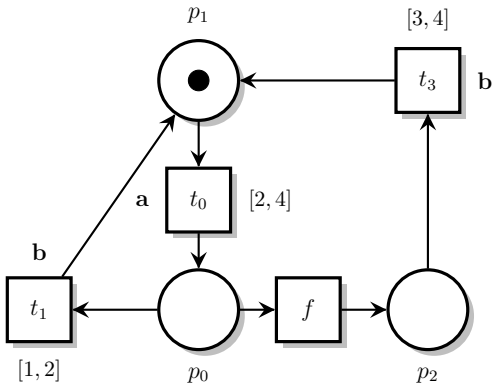


Figure 3.1: Example of a TPN

This TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ is composed of:

- Places: $P = \{p_0, p_1, p_2\}$.
- Transitions: $T = \{t_0, t_1, t_3, f\}$.
- Precondition and postcondition functions:

$$\mathbf{Pre} = \begin{array}{c} \begin{matrix} p_0 & p_1 & p_2 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} t_0 \\ t_1 \\ f \\ t_3 \end{matrix}$$

$$\mathbf{Post} = \begin{array}{c} \begin{matrix} t_0 & t_1 & f & t_3 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \begin{matrix} p_0 \\ p_1 \\ p_2 \end{matrix}$$

- Initial marking: $m_0 = \{0, 1, 0\}$
- Static Intervals for each transition:

$$\mathbf{I}_s(t) = \begin{cases} [2, 4] & \text{if } t = t_0 \\ [1, 2] & \text{if } t = t_1 \\ [3, 4] & \text{if } t = t_3 \\ [0, +\infty] & \text{if } t = f \end{cases}$$

Moreover, a labelling function has been added to this model such that:

$$\mathcal{L}(t_0) = a, \mathcal{L}(t_1) = b, \mathcal{L}(t_3) = b, \mathcal{L}(f) = \varepsilon$$

with the alphabet $\Sigma = \{a, b\}$.

3.2 Semantics of a TPN

To define the behaviour of a TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, I_s \rangle$ a *marking* m and the set $\mathcal{E}(m)$ of enabled transitions are defined as:

Definition 3 (Marking)**.** *A marking m is a (total) function $m : P \to \mathbb{N}$ from places in P to natural numbers.*

Definition 4 (Enabled transition)**.** *For a marking m, a transition t in T is* enabled *if and only if $m \dot{\geqslant} \mathbf{Pre}(t)$ (the pointwise comparison between functions is used).*

Definition 5 (Set of enabled transitions)**.** *$\mathcal{E}(m)$ is the set of transitions enabled for a marking m, i.e. $\mathcal{E}(m) = \{t \mid t \in T, m \dot{\geqslant} \mathbf{Pre}(t)\}$.*

Now it is possible to define a *state* of a TPN:

Definition 6 (State)**.** *A state s of a TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$ is a pair $s = (m, \varphi)$ in which m is a marking, and $\varphi : T \to \mathbb{I}$ is a mapping from transitions to time intervals, also called* firing domains *such that: $\forall t \in \mathcal{E}(m), \varphi(t) \in \mathbb{I} \wedge \uparrow\varphi(t) \leq \beta_t^s$.*

Intuitively, if t is enabled for a marking m, then $\varphi(t)$ contains the dates at which t can possibly fire in the future. For instance, when t is newly enabled, it is associated to its static time interval $\varphi(t) = \mathbf{I}_s(t)$. Likewise, a transition t can fire immediately only when 0 is in $\varphi(t)$ and it cannot remain enabled for more than its timespan, *i.e.* the maximal value in $\varphi(t)$.

For a given delay θ in $\mathbb{Q}_{\geq 0}$, we denote $\iota - \theta$ the time interval ι shifted (to the left) by θ: e.g. $[l, h] - \theta = [\max(0, l - \theta), \max(0, h - \theta)]$. By extension, $\varphi \dot{-} \theta$ is used for the partial function that associates the transition t the value $\varphi(t) - \theta$. This operation is useful to model the effect of time passage on the enabled transitions of a net.

We now go into more details about the semantics and behaviour of TPN. To do this, we must begin by formally defining what a TTS is. Our definition of a TTS came from [Bérard 2005a] where the authors define more representations of a TPN (which is different from the H-TTS in Chapter 2).

Definition 7 (Timed Transition Systems)**.** *A Timed Transition System (TTS) over the set of actions A is a tuple $[\![N]\!] = \langle S, s_0, A, \to \rangle$ where S is the set of states, $s_0 \in S$ is the initial state, $\to \subseteq S \times (A \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ is the set of edges. If $(s, \alpha, s') \in \to$, it is written $s \xrightarrow{\alpha} s'$.*

The following definition of the semantics of a TPN is quite standard, see for instance [Bérard 2005b, Berthomieu 2006]. In general terms, the semantics of a TPN is a TTS structure $\langle S, S_0, \to \rangle$ with only two possible kinds of actions: either a transition t is fired, or a time delay θ elapses. A transition t can fire from the state (m, φ) if t is enabled at m and firable instantly. More formally, the semantics of the TPN is defined as:

Definition 8 (TPN semantics)**.** *The semantics of a TPN* $N \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ *with the labelling function* $\mathcal{L} : T \to \Sigma \cup \{\varepsilon\}$ *is the Timed Transition System (TTS)* $[\![N]\!] = \langle S, s_0, \Sigma, \to \rangle$ *where* S *is the smallest set containing* s_0 *and closed by* $\to$*, where:*

- $s_0 = (m_0, \varphi_0)$ *is the initial state, with* m_0 *the initial marking and* $\varphi_0(t) = \mathbf{I}_s(t)$ *for every* t *in* $\mathcal{E}(m_0)$*;*
- *the state transition relation* $\to \subseteq S \times (\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ *is the relation such that for all states* (m, φ) *in* S*:*

  (i) $(m, \varphi) \xrightarrow{\mathcal{L}(t)} (m', \varphi')$ *iff :*
  - $t \in \mathcal{E}(m)$
  - $0 \in \varphi(t)$
  - $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
  - $\forall k \in T, k \in \mathcal{E}(m') \Rightarrow$
    $$\varphi'(k) = \begin{cases} \varphi(k) & \text{if } k \neq t \wedge m - \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(k) \\ \mathbf{I}_s(k) & \text{otherwise} \end{cases}$$

  (ii) if $\theta \dot{\leqslant} \varphi(t)$ *then* $(m, \varphi) \xrightarrow{\theta} (m, \varphi \dot{-} \theta)$*.*

Transitions in the case (i) above are called *discrete transitions* and mean that if a transition t is enabled and is ready to fire ($0 \in \varphi(t)$) then there is a transition labelled with $\mathcal{L}(t)$ in the TTS from the state (m, φ) to the state (m', φ') where $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$ and φ' is a firing function such that $\varphi'(k) = \varphi(k)$ for any persistent transition and $\varphi'(k) = \mathbf{I}_s(k)$ elsewhere; in the case (ii) transitions labelled with delays are the *continuous*, or time elapsing, transitions. Like with nets, the alphabet of a TTS is the set of labels, in Σ, associated to discrete actions.

3.2.1 Executions and traces

An *execution* of a net N is a sequence in its semantics $[\![N]\!]$. It is a time-event word over the alphabet containing both labels (in $\Sigma \cup \{\varepsilon\}$) and delays. Continuous transitions can always be grouped together, meaning that when $(m, \varphi) \xrightarrow{\theta} (m, \varphi')$ and $(m, \varphi') \xrightarrow{\theta'} (m, \varphi'')$ then necessarily $(m, \varphi) \xrightarrow{\theta+\theta'} (m, \varphi'')$ (and the firing domain φ' is uniquely defined from φ and θ). Based on this observation, executions of the form $\sigma \stackrel{\text{def}}{=} \theta_0\, a_0\, \theta_1\, a_1\, \ldots$ where each discrete transition is preceded by a single time delay can always be considered.

By contrast, a *trace* is the untimed word obtained from an execution when only the discrete actions without ε are kept. Then the *language* of a TPN N, denoted $L(N)$, is the set of all its (finite) traces.

By definition, the language of a TPN is prefix-closed; and it is regular when the net is bounded [Berthomieu 1983].

Example: For our example in Figure 3.1:

- An execution of the TPN could be $2a1\varepsilon3b$ and so the related trace is ab.
- The language is a repetition of a and b such $L(N) = (ab)^*$.

3.2.2 Firing sequences and runs

Let $N = \langle S, s_0, \Sigma, \rightarrow \rangle$ be a TPN and $\sigma = t_1...t_n$ be a *transition sequence* in T. Let $\tau = \tau_0...\tau_n$ with $\tau_i \in \mathbb{Q}_{\geq 0}$ be a sequence of times. Then the sequence $\sigma(\tau) = \tau_0 t_1 \tau_1 ... t_n \tau_n$ is called a *run of* σ.

Definition 9 (feasible run)**.** *Let a TTS $[\![N]\!] = \langle S, s_0, \Sigma, \rightarrow \rangle$, $s = (m, \varphi)$ a state in S and $\sigma(\tau) = \tau_0 t_1 \tau_1 ... t_n \tau_n$ a run of σ. It is said that $\sigma(\tau)$ fires from s into s' ($s \xrightarrow{\sigma(\tau)} s'$) if there are states $(s_0, s'_0..., s_n)$ in S such $s \xrightarrow{\tau_0} s_0 \xrightarrow{\mathcal{L}(t_1)} s'_0 ... s'_{n-1} \xrightarrow{\tau_n} s_n \xrightarrow{\mathcal{L}(t_n)} s'$.*

The run $\sigma(\tau)$ is a feasible run from state s in S, if there is a state s' such that $\sigma(\tau)$ can fire from s to s'.

The run $\sigma(\tau)$ is said feasible in $[\![N]\!]$, if $\sigma(\tau)$ is a feasible run from s_0.

Definition 10 (firing sequence)**.** *A transition sequence σ is a firing sequence in the TTS $[\![N]\!]$ if it exits a sequence of time τ such σ has a feasible run $\sigma(\tau)$ in $[\![N]\!]$.*

Definition 11 (reachable state)**.** *A state s is reachable in a TTS $[\![N]\!]$ if there exists a firing sequence σ in $[\![N]\!]$ with $s_0 \xrightarrow{\sigma(\tau)} s$.*

Definition 12 (state space)**.** *The set RS_N of all reachable states in a TTS $[\![N]\!]$ is called the state space of $[\![N]\!]$.*

Definition 13 (reachable marking)**.** *A marking m is reachable in a TTS $[\![N]\!]$ if there is a reachable state s in $[\![N]\!]$ with $s = (m, \varphi)$.*

3.2.3 Equivalence

The notion of bisimulation is a useful concept for the comparison of behaviours. In a general way, it allows to verify that two behaviours are "similar": if a system does an action, then the other system also does this action and vice versa. Moreover, it is also possible to use a "weak" variant of this property, taking into account only the non-silent (observable) actions. Since a formal language can be defined by the set of behaviours it can express, this equivalence can also be used to compare two formalisms. This equivalence has been extended to the comparison of TTS. The definition of timed bisimulation is thus obtained.

Definition 14 (Timed bisimulation). *Assume* $G_1 = \langle S_1, s_1^0, \Sigma_1, \rightarrow_1 \rangle$ *and* $G_2 = \langle S_2, s_2^0, \Sigma_2, \rightarrow_2 \rangle$ *are two TTS and the binary relation* $\sim \subseteq S_1 \times S_2$. G_1 *and* G_2 *are said strongly timed bisimilar iff* $s_1^0 \sim s_2^0$ *and, whenever* $s_1 \sim s_2$ *and* $a \in \Sigma_1 \cup \Sigma_2 \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}$*:*

- $s_1 \xrightarrow{a}_1 s_1' \Rightarrow \exists s_2', s_2 \xrightarrow{a}_2 s_2' \wedge s_1' \sim s_2'$
- $s_2 \xrightarrow{a}_1 s_2' \Rightarrow \exists s_1', s_1 \xrightarrow{a}_1 s_1' \wedge s_1' \sim s_2'$

Strong timed bisimilarity could be a too strong equivalence. Thus, a weak version of timed bisimulation is preferred. It relies on a weak version of the transition relation $s \stackrel{\alpha}{\Rightarrow} s'$ (with α an action in $\Sigma \cup \mathbb{Q}_{\geq 0}$) where silent transitions are hidden. The weak transition relation is defined from $\xrightarrow{\alpha}$ as follows:

Definition 15 (Weak transition relation). *From a transition relation* $\rightarrow \subseteq S \times (\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ *the weak transition relation* $\Rightarrow \subseteq S \times (\Sigma \cup \mathbb{Q}_{\geq 0}) \times S$ *is defined for an action* $\alpha \in \Sigma \cup \mathbb{Q}_{\geq 0}$ *as:*

The weak transition relation $s \stackrel{\alpha}{\Rightarrow} s'$ *is defined from the following set of rules:*

$$\frac{s \xrightarrow{\alpha} s'}{s \stackrel{\alpha}{\Rightarrow} s'} \quad \frac{s \stackrel{\alpha}{\Rightarrow} s' \;\; s' \xrightarrow{\epsilon} s''}{s \stackrel{\alpha}{\Rightarrow} s''} \quad \frac{s \stackrel{\epsilon}{\Rightarrow} s' \;\; s' \xrightarrow{\alpha} s''}{s \stackrel{\alpha}{\Rightarrow} s''} \quad \frac{s \stackrel{\theta}{\Rightarrow} s' \;\; s' \stackrel{\theta'}{\Rightarrow} s''}{s \stackrel{\theta+\theta'}{\Longrightarrow} s''}$$

It is then possible to define a *weak timed bisimilation* between two TTS as follows:

Definition 16 (Weak timed bisimulation). *Assume* $G_1 = \langle S_1, s_1^0, \Sigma_1, \rightarrow_1 \rangle$ *and* $G_2 = \langle S_2, s_2^0, \Sigma_2, \rightarrow_2 \rangle$ *are two TTS with the weak relations* $\Rightarrow_i$ *and the binary relation* $\approx \subseteq S_1 \times S_2$. G_1 *and* G_2 *are said weak timed bisimilar iff* $s_1^0 \approx s_2^0$ *and, whenever* $s_1 \sim s_2$ *and* $a \in \Sigma_1 \cup \Sigma_2 \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}$*:*

- $s_1 \stackrel{a}{\Rightarrow}_1 s_1' \Rightarrow \exists s_2', s_2 \stackrel{a}{\Rightarrow}_2 s_2' \wedge s_1' \approx s_2'$
- $s_2 \stackrel{a}{\Rightarrow}_1 s_2' \Rightarrow \exists s_1', s_1 \stackrel{a}{\Rightarrow}_1 s_1' \wedge s_1' \approx s_2'$

In the following, two nets denoted $N_1 \approx N_2$, are bisimilar when $[\![N_1]\!] \approx [\![N_2]\!]$.

Example: Consider the two TTS described in Figure 3.2. They are not timed bisimilar due to the ϵ transition in the left TTS, but they are weak timed bisimilar.

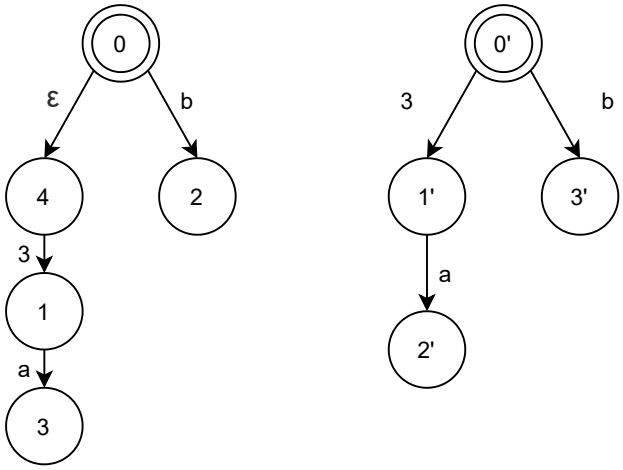


Figure 3.2: Example of a two simple TTS

3.3 Synchronous Products

As stated in the previous chapter, the problem we address in this thesis is based on the synchronous product of TPN. In this section, we will go through various ways of realizing this product, starting by defining the TTS product, and then describing two ad-hoc methods.

3.3.1 TTS products

Definition 17 (Product of TTS)**.** *Assume* $[\![N_1]\!] = \langle S_1, s_1^0, \Sigma_1, \to_1 \rangle$ *and* $[\![N_2]\!] = \langle S_2, s_2^0, \Sigma_2, \to_2 \rangle$ *are two TTS. The product of* $[\![N_1]\!]$ *by* $[\![N_2]\!]$ *is the TTS* $[\![N_1]\!] \| [\![N_2]\!] = \langle S_1 \times S_2, (s_1^0, s_2^0), \Sigma, \to \rangle$ *with* $\Sigma = \Sigma_1 \cup \Sigma_2$ *and* $\to$ *the smallest relation obeying the following rules* ($\alpha \in \Sigma_1 \cup \Sigma_2 \cup \{\epsilon\} \cup \mathbb{Q}_{\geq 0}$)*:*

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \ \alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\varepsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2' \ \alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\varepsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \ s_2 \xrightarrow{\alpha}_2 s_2' \ \alpha \neq \varepsilon}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

Timed bisimilarity (strong and weak) is preserved by product [Peres 2011], meaning that for all TTS G, G_1 and G_2 we have $G_1 \approx G_2 \Rightarrow (G \| G_1) \approx (G \| G_2)$.

Example: Consider the two TTS shown in Fig. 3.2, then their product synchronized using the properties in Definition 17 is the TTS given by the Fig. 3.3.

In this example, the product of the two TTS with common labelled events (a and b) is made. The evolution of the system is represented as a pair of state (with $(0, 0')$ as its origin). This product can also be conducted on TA with similar results (even if invariant can sometimes be cumbersome if the product is not automatized).

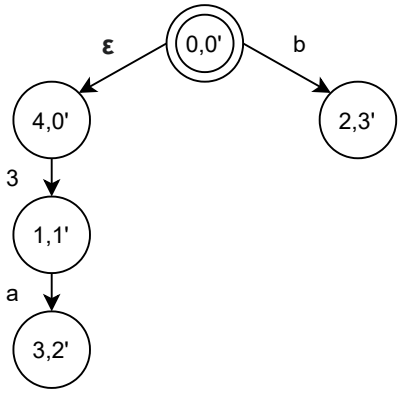


Figure 3.3: Synchronous product of two TTS

3.3.2 Synchronous product of TPN

However, the Definition 17 only considers *TTS* and not TPN. The synchronous product of two TPN represents the core of this thesis. To our knowledge, two methods were proposed to process the product of TPN. In [Lime 2003b] a specific State Class Graph (see Chapter 5 for more details on SCG) based on timed automata is computed and in [Peres 2011] an ad-hoc transformation is conducted to an Inhibit-Permit-TPN (IPTPN).

3.3.2.1 State Class Timed Automaton

Lime and Roux proposed an extension in [Lime 2003b] of the state class graph construction that allows to build the state class graph (see Chapter 5 for more information) of a bounded TPN as a timed automaton. They prove that this timed automaton and the TPN are timed-bisimilar and they also prove a relative minimality result of the number of clocks needed in the obtained automaton. We use this method to compute a TA from our TPN models and compare it with our contributions in the Chapter 7.

This first approach is structural but limited to Petri nets whose underlying net is 1-safe.

This method is augmented by Cassez and Roux [Cassez 2006] who propose a structural encoding of TPN into TA that preserves the semantics in the sense of timed bisimulation, and therefore that preserves timed language acceptance. This encoding generates one automata, and one clock, for every transition in the TPN and it can be extended in order to accommodate strict timing constraints; that is static time intervals that have a finite, open bound.

Since we only tackle bounded TPN in this thesis, we remained with the first method which is more straightforward.

3.3.2.2 IPTPN

Time Petri nets are extended with a priority relation in Berthomieu et al. [Berthomieu 2006] where it is shown that priorities strictly increase the expressiveness of TPN. TPN was further extended with a second relation over transitions, the permit relation. The priority relation is kept but renamed the inhibit relation.

Definition 18 (Inhibits Permits TPN)**.** *An IPTPN is a labelled TPN augmented with two relations over transitions:*

- *I is the inhibit relation written $x - \circ y$. It is spelled x inhibits y.*
- *P is the permit relation written $x - \bullet y$. It is spelled x permits y.*

Both the inhibit and the permit arcs are activated when a token sensibilized then, just like a classical transition.

The inhibit arc forbid the firing of the affected transitions (which means to the x transition blocks the firing of the y transition). The permits arc allows the firing of the affected transitions (which means to the x transition allows the firing of the y transition).

Using these properties of inhibiting and allowing the firing of a transition we separate the timing constraint from the label of a timed transitions.

These relations allow the decomposition of TPN into IPTPN, which are composable with a synchronous product operation. The idea is to mimic the timing behaviour of the transitions without having the timing constraint on them (by inhibiting and permitting the firing of transitions). Let's take a quick example:

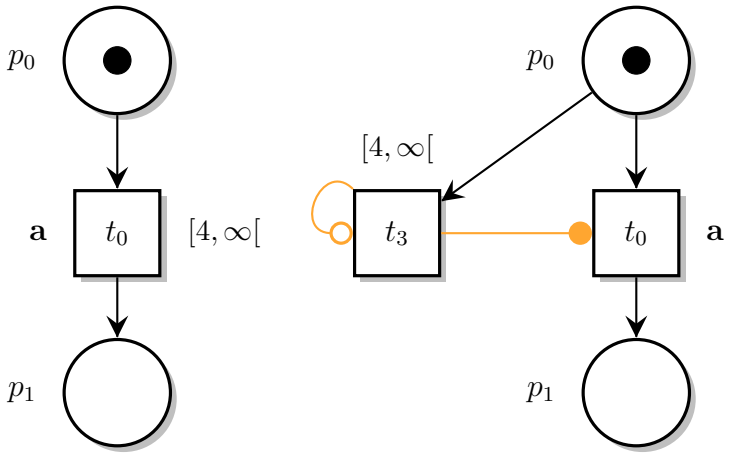


Figure 3.4: TPN with a single transition (left) decomposed as an IPTPN (right)

Here, the idea is to decorrelate the transition from its timing constraint but still keeping the timing behaviour. In this case, the synchronous product is only conducted on the labelled transitions and it keeps their *timing constraint* with

the Inhibit and Permit arc on the fused transitions (hence keeping both timing behaviour).

However, this solution uses *priorities* arcs (with the Inhibit and Permit arc) and duplicates every transitions with timing constraints. This leads to much bigger systems, especially the use of priorities which requires special state classes to keep the priority information. The idea with our contribution is also to tackle the problem of scalability for a synchronous product of TPN.

3.4 Parametric state and parametric run

This Section introduces parametric states and runs and is entirely based on the book *Time and Petri Nets* of *Popova-Zeugmann* [Popova-Zeugmann 2013].

In a TPN, for the same visible actions (label) there may be an infinity of possible executions for a same trace: trace of $1a4b$ is the same as $0a5b$. The main idea of the parametric state representation is then to define a notation capable of representing multiple behaviours regarding the timing constraints. This "execution" is called *parametric run* and is based on *parametric states.*

We need to define the t-marking function as follows:

Definition 19 (t-marking)**.** *Let T be the set of all transitions in a TPN N. Any (total) function $h : T \to \mathbb{Q}_{\geq 0} \cup \{\sharp\}$ is a t-marking in N.*

For Popova-Zeugmann, a state is a pair $z = (m, h)$ of a marking with a t-marking where $h(t)$ can be interpreted as the clock of t, meaning that it measures the time elapsed since the transition last became enabled. The value $\sharp$ represents that the clock of the transition has stopped (t not enabled), while any transition t with $h(t) \in \mathbb{Q}_{\geq 0}$ is running and shows the time of t in the state.

Remind that we also denote β_t^s as the latest firing time of t and α_t^s as its earliest firing time, *i.e* $\mathbf{I}_s(t) = [\alpha_t^s, \beta_t^s]$.

Definition 20 (Popova-Zeugmann TTS)**.** *For a TPN $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ with the labelling function $\mathcal{L} : T \to \Sigma \cup \{\varepsilon\}$ a TTS $[\![N]\!]_p = \langle Z, z_0, \Sigma, \to \rangle$ is defined where Z is the smallest set containing z_0 and closed by $\to$, where:*

- *$z_0 = (m_0, h_o)$ is the initial state, with m_0 the initial marking and*

$$h_0 = \begin{cases} 0 & \text{if } t \in \mathcal{E}(m_0) \\ \sharp & \text{otherwise} \end{cases}$$

- *the state transition relation $\to \subseteq S \times (\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ is the relation such that for all states (m, h) in S:*

  (i) $(m, h) \xrightarrow{\mathcal{L}(t)} (m', h')$ iff :

$* \ t \in \mathcal{E}(m)$

$* \ h(t) \geq \alpha_t^s$

$* \ \forall k \in \mathcal{E}(m)$
$h(k) \leq \beta_k$

$* \ m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$

$* \ \forall k \in T \Rightarrow$
$$h'(k) = \begin{cases} \sharp & \text{if } k \notin \mathcal{E}(m') \\ h(k) & \text{if } k \in \mathcal{E}(m') \wedge k \neq t \wedge m - \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(k) \\ 0 & \text{otherwise} \end{cases}$$

(ii) if $\forall t \in \mathcal{E}(m), \theta \dot{\leqslant} \mathbf{I}_s(t) \dot{-} h(t)$ *then* $(m,h) \xrightarrow{\theta} (m,h')$, $h'(t) = h(t) - \theta$.

Theorem 1. *[Popova-Zeugmann 2013] For a TPN N,* $[\![N]\!]$ *is isomorphic to* $[\![N]\!]_p$.

Proof. It is easy to show that for a feasible execution $\sigma = \tau_0 t_1 \tau_1 t_2 ... t_n \tau_n$ such that $z_0 \xrightarrow{\sigma} z$ with $z = (m_z, h)$ in $[\![N]\!]_p$ we have $s_0 \xrightarrow{\sigma} s$ in $[\![N]\!]$ with $s = (m_s, \varphi)$ such that $m_z = m_s$ and for $t \in \mathcal{E}(m)$, $\varphi(t) = \mathbf{I}_s(t) - h(t)$. □

Definition 21 (Parametric state and parametric run)**.** *Let* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ *be a TPN and let* $\sigma = t_1 \dots t_n$ *be a firing sequence in N. Then, the* parametric run $(\sigma(x), B_\sigma)$ *of* σ *in* $[\![N]\!]$ *with* $\sigma(x) = x_0 t_1 x_1 \dots x_{n-1} t_n x_n$ *and the* parametric state (z_σ, B_σ) *in N are recursively defined as follows:*

Basis:

- $\sigma = \varepsilon$, *i.e,* $\sigma(x) = x_0$.

- $z_\sigma = (m_\sigma, h_\sigma)$ *and* B_σ *are defined as follows:*
  - $m_\sigma = m_0$
  - $h_\sigma(t) = \begin{cases} x_0 & \text{if} \quad t \in \mathcal{E}(m_\sigma) \\ \sharp & \text{otherwise} \end{cases}$
  - $B_\sigma = \{0 \leq h_\sigma(t) \leq \beta_t^s \mid t \in \mathcal{E}(m_\sigma)\}$

Step:

Assume that z_σ *and* B_σ *are already defined for the sequence* $\sigma = t_1 \dots t_n$. *For* $\sigma = t_1 \dots t_n t_{n+1}$, $w = t_1 \dots t_n$ *(we have* $\sigma = w t_{n+1}$*) is defined and set:*

- $m_\sigma = m_w - \mathbf{Pre}(t_{n+1}) + \mathbf{Post}(t_{n+1})$

- $h_\sigma(t) = \begin{cases} \sharp & \text{if } t \notin \mathcal{E}(m_\sigma) \\ h_w(t) + x_{n+1} & \text{if } t \in \mathcal{E}(m_\sigma) \cap \mathcal{E}(m_w) \wedge m - \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(t_{n+1}) \wedge t \neq t_{n+1} \\ x_{n+1} & \text{otherwise.} \end{cases}$

- $B_\sigma = B_w \cup \left\{ \alpha_{t_{n+1}}^s \leq h_w(t_{n+1}) \right\} \cup \{0 \leq h_\sigma(t) \leq \beta_t^s \mid t \in \mathcal{E}(m_\sigma)\}$

Remark that for a firing sequence $\sigma = t_1..t_n$, the set S_σ of all solutions for $x = (x_1...x_n)$ satisfying the inequalities in B_σ is a polyhedron. The t-marking $h_\sigma(t)$ is a vector of linear functions.

A parametric state represents the set of all states which can be reached by firing a feasible run of σ so a state is the combination of all firing sequence σ with their solutions x in B_σ.

Definition 22 (set of all reachable states)**.** *The set of all states which can be reached by firing a feasible run is :*

$$K_\sigma = \{z_\sigma \,|\, B_\sigma\} = \{z_{\sigma(\beta(x))} \mid \beta(x) \textit{ is a solution of } B_\sigma\}$$

Example: Let's consider the TPN described in Fig. 3.1, where :

$$K_\epsilon = \{\{0,1,0\}, (x_0, \sharp, \sharp, \sharp) \mid \{0 \leq x_0 \leq 4\}\}$$

After firing the sequence $\sigma = t_0$, the state K_{t_0} becomes:

$$K_{t_0} = \{\{1,0,0\}, (\sharp, x_1, \sharp, x_1) \mid \{2 \leq x_0 \leq 4, 0 \leq x_1 \leq 2\}\}$$

where the set of conditions B_{t_0} is the union of B_ε, $\{\alpha^s_{t_0} \leq h_\varepsilon(t_0) \leq \beta^s_{t_0}\} = \{2 \leq x_0 \leq 4\}$ and $\{0 \leq h_\sigma(t) \leq \beta^s_t \mid t \in \mathcal{E}(m_\sigma)\} = \{0 \leq x_1 \leq 2\}$.

Definition 23 (Popova state class)**.** *Let N be a TPN and σ a feasible transition sequence. The Popova state class (PSC) C_σ of σ is defined as follows :*

Basis: $C_\varepsilon = \{s \mid \exists \tau\, (\tau \in \mathbb{Q}_{\geq 0} \wedge z_0 \xrightarrow{\tau} z)\}$

Step: *If C_σ is already defined then $C_{\sigma t}$ is derived from C_σ by firing t:*
$C_{\sigma t} = \{z \mid \exists z_1 \exists z_2 \exists \theta (s_1 \in C_{tr(\sigma)} \wedge \tau \in \mathbb{Q}_{\geq 0} \wedge z_1 \xrightarrow{t} z_2 \xrightarrow{\tau} z)\}$

The PSC C_ε is the set of all states in $[\![N]\!]$ reachable from the initial state by elapsing time but without firing transitions. And the class C_σ contains all states that are reachable by firing any feasible runs of σ.

Popova-Zeugmann claims that:

Theorem 2. *For every TPN N and for every firing sequence σ it holds that:*

(i) $RS_N = \bigcup_\sigma C_\sigma$*, i.e. the state space of N is the union of all (Popova) state classes.*

(ii) $\{z_\sigma \,|\, B_\sigma\} = C_\sigma$.

3.5 Summary

In this Chapter 3 we have presented the bases of our work, the TPN and its technicalities. To summarize:

- *TPN* are composed of places, transitions and timing constraint. Their semantics is based on their labels and they can be combined with labels and time-events to create *chains* (or *patterns*).
- *Equivalence* and *weak-bisimilarity* are necessary tools to keep in check the properties and behaviour after operations on TPN. They are mainly used to check our TTS still keeps its properties after the synchronous product.
- *Synchronous Product* are based on the language of DES. Combination is based on the label of transitions and they can be merged if they have the same labels. This operation is trivial in Automata and TTS because their timing constraints can be separated from their labels.
- *TPN* are not well suited for a classical synchronous product (and it is not formally defined).
- *SCTA* and *IPTPN* are solutions to process the composition of TPN but the first relies on a transformation into a TA and the second relies on priorities and creating more transitions which can lead to a problem in term of scalability.

The main new model to create a synchronous product is depicted in the Chapter 4 and it constitutes the basis for all our contributions.

In Chapter 4 we will focus on PTPN and all its properties and technicalities.

Chapter 4

Product TPN and their Semantics

In this chapter we present our first contribution in the field of synchronous product of TPN, the PTPN. First, we define our product before going into more detail about the state created from a PTPN. In the second section of this Chapter, we present the new semantics provided with PTPN and a new behaviour, the timelock.

Thirdly, we go more precisely into the operation ongoing in a PTPN, in terms of execution and semantics. In the fourth section, we explain more precisely the synchronous product between two TTS representing our TPN to compare it to our PTPN operation.

Finally, we conclude via a summary.

4.1 Definition

Product TPN (PTPN) were created to tackle the problem of synchronous product on a timed context in a Petri Net model. Indeed, the synchronous product of two time transitions is not clearly defined (contrary to the Timed Automaton).

A PTPN proposes an extension of TPN with a *synchronous product* operation between TPN in the style of Arnold-Nivat synchronization of processes [Arnold 2002]. The main idea of a PTPN is to force transitions with the same labels to fire synchronously.

A PTPN is the composition (N, R) of a TPN N, with a set of transitions T, and a *product relation* R that is a collection of *firing sets* $r_1, ..., r_n$ included in T (hence $R \subseteq P(T)$, the powerset of T). The idea is that all the transitions in an element r of R must be fired at the exact same time. As a consequence, two transitions in r should have the same labels ($\mathcal{L}(r) = a$ should be used to say they have a common label a) and not interfere with each other (they should not share a common input place).

Definition 24 (Product TPN)**.** *A* product TPN *(PTPN)* (N, R) *is a pair of a TPN* $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ *and a product relation* $R \subseteq P(T)$ *such that for every firing set* r *in* R*, transitions in* r *are independent and compatible :*

$$(t_1, t_2 \in r) \Rightarrow (\mathcal{L}(t_1) = \mathcal{L}(t_2)) \wedge (\forall p \in P, \mathbf{Pre}(t_1)(p) > 0 \Rightarrow \mathbf{Pre}(t_2)(p) = 0)$$

Example: Let's take a simple example of PTPN (N, R) composed of two single transitions TPN (figure 4.1). Here N is a classical TPN with $P = \{p_0, p_1, p_2, p_3\}$, $T = \{t_0, t_1\}$, etc., extended with the product relation $R = \{\{t_0, t_1\}\}$.

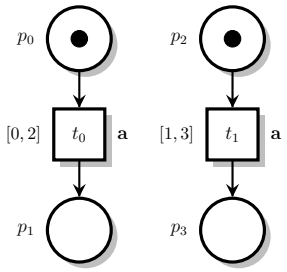


Figure 4.1: A TPTN with a product relation $R = \{\{t_0, t_1\}\}$

4.2 Semantics of a PTPN and Timelock

The semantics for PTPN relies largely on the semantics of TPN but makes a particular use of labels to synchronize transitions with same label. As we will see, this behaviour can conduct to a situation where time constraints induce a blocking situation that we call time deadlock.

Definition 25 (Semantics of a PTPN)**.** *The semantics of a PTPN (N, R) is the TTS $[\![(N, R)]\!] = \langle S, s_0, \Sigma, \rightarrow \rangle$, where S is the smallest set containing s_0 and closed by $\rightarrow$ such that:*

- *$s_0 = (m_0, \varphi_0)$ is the initial state, with m_0 the initial marking and $\varphi_0(t) = \mathbf{I}_s(t)$ for every t in $\mathcal{E}(m_0)$;*
- *the state transition relation $\rightarrow \subseteq S \times (\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ is the relation such that for all states (m, φ) in S:*

  (i) $(m, \varphi) \xrightarrow{a} (m', \varphi')$ iff :

  - *$\exists r \in R$ with labels a*
  - *$\forall t \in r, t \in \mathcal{E}(m)$*
  - *$\forall t \in r, 0 \in \varphi(t)$*
  - *$m' = m - \sum_{t \in r} \mathbf{Pre}(t) + \sum_{t \in r} \mathbf{Post}(t)$*
  - *$\forall k \in T, k \in \mathcal{E}(m') \Rightarrow$*

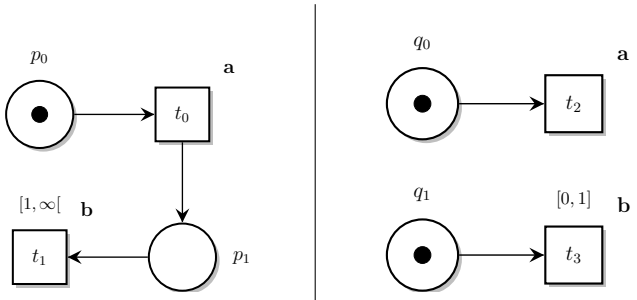
    $$\varphi'(k) = \begin{cases} \varphi(k) & \text{if } k \neq t \wedge m - \sum_{t \in r} \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(k) \\ \mathbf{I}_s(k) & \text{otherwise} \end{cases}$$

  (ii) if $\theta \dot{\leqslant} \varphi$ then $(m, \varphi) \xrightarrow{\theta} (m, \varphi \dot{-} \theta)$.

The only new case (compared to a TPN) is for transitions with the same label ($t \in r$) to fire synchronously and the effect if firing both of them simultaneously. When a set of transitions $r = \{t_1, ..., t_n\}$ is fired from state (m, φ), a transition k (with $k \neq t$) is said to be persistent if k is also enabled in the marking $m - \sum_{t \in r} \mathbf{Pre}(t)$, only if $m - \sum_{t \in r} \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(k)$. The other transitions enabled after firing r are called newly enabled.

TPN form a natural subset of PTPN, where every firing set has only one transition. More precisely, a TPN N with transitions $\{t_1, ..., t_n\}$ can always be interpreted as the PTPN (N, R_N), where R_N is the collection of singletons $\{\{t_1\}, ..., \{t_n\}\}$. In the following, we often omit the product relation in a PTPN when it is not needed, or obvious from the context. We should also simply use the term net, or the symbol N, to refer to a Product TPN.

Figure 4.2 shows an example where we can consider two TPN N_1 and N_2 as a PTPN.



(a) $(N_1, \{\{t_0\}, \{t_1\}\})$ (b) $(N_2, \{\{t_3\}, \{t_2\}\})$

Figure 4.2: Two examples of PTPN

As a side effect, our choice of semantics entails that a transition on a "shared label" is blocked until a fireable transition with the same label on the other component is found. This may introduce a new kind of *time deadlock* that has no direct equivalent in a TPN: when a transition that shares a label has to fire urgently (hence time cannot progress) while there are no matching transition that is time-enabled. For a state (m, φ), it happens when a transition in $t \in r$ is enabled with $\varphi(t) = [0, 0]$ and $\exists t' \in r, (t' \notin \mathcal{E}(m)) \vee (0 \notin \varphi(t'))$.

Example: Consider a PTPN (N, R) composed by the two TPN N_1 and N_2 of the Figure 4.2 and with a relation $R = \{\{t_0, t_2\}, \{t_1, t_3\}\}$. The first synchronized event which can occur in the PTPN $N_1 \times N_2$ is transition $\{t_0, t_2\}$. Since time elapses as in a classical TPN, the t_3 transition is enabled and its timing constraint evolves independently of t_1 which is not enabled. The only common solution for the timing constraint in $\{t_1, t_3\}$ is 1 and they have to be fired synchronously. This means that if any amount of time is elapsed before firing $\{t_0, t_2\}$ (which is possible), the $\{t_1, t_3\}$ transitions will never be able to fire (because of timing constraints). This is the primary example of a *timelock*.

Time deadlocks are important in the context of our work. They model the case of two executions that start with the same observation but that cannot be reconciled after some point; meaning that observable events are enough to eventually discriminate them. Such situations are common during diagnosis and indicate that there is a way to distinguish two partial observations.

Remark that the reachable states in $[\![(N, R)]\!]$ are a subset of the states in $[\![N]\!]$. This is because a synchronization on a shared label may be forbidden, but it never creates a new opportunities to fire a transition.

4.3 Executions and traces

An *execution* of a PTPN (N, R) is a sequence of actions in its semantics $[\![(N, R)]\!]$ that starts from its initial state. It is a time-event word $\alpha_1 \dots \alpha_n$ over the alphabet containing both labels $(a, b, \dots \in \Sigma)$ and delays $(\theta \in \mathbb{Q}_{\geq 0})$, where silent transitions are not recorded. Since labelled transitions are synchronized there exists only one label for a pair of synchronized transitions. In the following, executions are simplified in order to avoid the occurrence of two successive delays; just like in Section 3.2.1.

By contrast, a *trace* is the untimed word obtained from an execution when only the discrete actions are kept. Then the language of a PTPN is the set of all its (finite) traces. As we will see in the Chapter 5, the State Class Graph construction of [Berthomieu 1983] provides an effective method for computing a finite representation of the traces in a bounded TPN. We can do the same with Product TPN using the SCG construction.

Example: We can illustrate our definitions by considering the PTPN (N, R) composed by the two TPN N_1 and N_2 of the Figure 4.2 and with a relation $R = \{\{t_0, t_2\}, \{t_1, t_3\}\}$. As explained before, the only possibles executions are $\theta_0 a$ with $\theta_0 > 0$ or $0a1b$. So the set of traces is $\{a, ab\}$.

4.4 Synchronous product of PTPN

We define in this Section the product of two PTPN.

Definition 26 (Synchronous product of PTPN)**.** *Given two PTPN (N_1, R_1) and (N_2, R_2) with sets of places P_1, P_2 and transitions T_1, T_2, their product $(N_1, R_1) \times (N_2, R_2)$ is the PTPN (N, R) where N is the concurrent composition (juxtaposition) of N_1 with N_2 the net $\langle P_1 \cup P_2, T_1 \cup T_2, \mathbf{Pre}, \mathbf{Post}, m_0^1 \uplus m_0^2, \mathbf{I}_s \rangle$ with $\mathbf{Pre}(t)(p) =$*

$\mathbf{Pre}_i(t)(p)$ *if and only if* $t \in T_i$ *and* $p \in P_i$ *with* $i \in 1..2$*, and 0 otherwise (same with* **Post***); and the product relation* R *is such that:*

$$R = \bigcup_{a \in \Sigma_{1,2}} \{r_1 \cup r_2 \mid r_i \in R_i, \mathcal{L}(r_i) = a, i \in 1..2\} \cup \bigcup_{a \in \Sigma \setminus \Sigma_{1,2} \cup \{\varepsilon\}} \{r \mid r \in R_1 \cup R_2, \mathcal{L}(r) = a\}$$

with $\Sigma_{1,2}$ *the set of shared labels and* $\Sigma = \Sigma_1 \cup \Sigma_2$*.*

Unlike the conventional synchronous composition operator between Petri nets, transitions with the same labels are not merged but, instead, relations are composed. But like with synchronization, our goal is to define an operation that is a congruence, meaning that $[\![(N_1, R_1) \times (N_2, R_2)]\!]$ is equivalent to $[\![(N_1, R_1)]\!] \parallel [\![(N_2, R_2)]\!]$.

Theorem 3. *For two PTPN* (N_1, R_1) *and* (N_2, R_2)*, the TTS* $[\![(N_1, R_1) \times (N_2, R_2)]\!]$ *is isomorphic to* $[\![(N_1, R_1)]\!] \parallel [\![(N_2, R_2)]\!]$*.*

Proof. The Theorem can be proved by induction. Assume two PTPN (N_1, R_1) and (N_2, R_2) and $(N, R) = (N_1, R_1) \times (N_2, R_2)$. We denote:

- $[\![(N, R)]\!] = \langle S, s_0, \mapsto, \Sigma \rangle$
- $[\![(N_i, R_i)]\!] = \langle S_i, s_i^0, \Sigma_i, \to_i \rangle$ and $i = 1, 2$
- $[\![(N_1, R_1)]\!] \parallel [\![(N_2, R_2)]\!] = \langle (S_1 \times S_2), (s_1^0, s_2^0), \Sigma_\parallel, \to \rangle$.

By definition we have $\Sigma = \Sigma_1 \cup \Sigma_2 = \Sigma_\parallel$ and $s_0 = (s_1^0, s_2^0)$ that initializes our induction.

Remark that two TTS $\langle S_1, s_1^0, \to_1, \Sigma \rangle$ and $\langle S_2, s_2^0, \to_2, \Sigma \rangle$ over the same set of labels Σ are isomorphic if there is a bijection $B : S_1 \to S_2$ with $B(s_1^0) = s_2^0$ and $(s, t, s') \in \to_1 \Leftrightarrow (B(s), t, B(s')) \in \to_2$ for all $s, s' \in S_1$.

Suppose an execution σ in $[\![(N, R)]\!]$ from the initial state s_0 to a reachable state $s = (m, \varphi)$, *i.e.* $s_0 \overset{\sigma}{\mapsto} s$, with (s_1, s_2) a state in $S_1 \times S_2$ reachable with the same execution, *i.e.* $(s_1^0, s_2^0) \overset{\sigma}{\to} (s_1, s_2)$.

Consider an action α in $\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}$ such $s \overset{\alpha}{\mapsto} s'$ exists with $s' = (m', \varphi')$. We have three cases (Def. 25):

1. For $\alpha \in \mathbb{Q}_{\geq 0}$, we have $s' = (m, \varphi \dot{-} \alpha)$ and so $\alpha \dot{\leqslant} \varphi_i$. By definition (see Def. 8), if $(m_i, \varphi_i) \in S_i$ and $\alpha \dot{\leqslant} \varphi_i$ then $s_i \overset{\alpha}{\to}_i s_i'$ exists. Then by Definition 17 $(s_1, s_2) \overset{\alpha}{\to} (s_1', s_2')$ exists and $s' = (s_1', s_2')$.

2. For $\alpha \in \Sigma_{1,2}$, we have by definition transitions $t_i \in T_i, i \in 1..2$ such that $\mathcal{L}(t_i) = \alpha$, $t_i \in \mathcal{E}(m)$ and $0 \in \varphi(t_i)$. Thus, $s_i \overset{\alpha}{\to}_i (m_i', \varphi_i')$ exists and $m' = m_1' \uplus m_2'$ and $\varphi' = \varphi_1' \uplus \varphi_2'$ (see properties on pre and post-conditions in Def. 24). Then (Def. 17) $(s_1, s_2) \overset{\alpha}{\to} (s_1', s_2')$ exists and $s' = (s_1', s_2')$.

3. For $\alpha \in (\Sigma \setminus \Sigma_{1,2}) \cup \{\varepsilon\}$, it means that there exists a relation $r \in R_1$ (resp. in R_2) such $\mathcal{L}(r) = \alpha$ and $\nexists r' \in R_2, \mathcal{L}(r') = \alpha$ (resp. R_1). We have also $\forall t \in r, t \in \mathcal{E}(m) \wedge 0 \in \varphi(t)$. Thus $s_1 \xrightarrow{\alpha}_1 (m'_1, \varphi'_1)$ (resp. $s_2 \xrightarrow{\alpha}_2 (m'_2, \varphi'_2)$) exists with $m' = m'_1 \uplus m_2$ and $\varphi' = \varphi'_1 \uplus \varphi_2$, (resp. $m' = m_1 \uplus m'_2$ and $\varphi' = \varphi_1 \uplus \varphi'_2$). Then (Def. 17) $(s_1, s_2) \xrightarrow{\alpha} (s'_1, s_2)$ (resp. $(s_1, s_2) \xrightarrow{\alpha} (s_1, s'_2)$) exists and $s' = (s'_1, s'_2)$.

Now, consider an action α in $\Sigma \cup \{\varepsilon\} \cup \mathbb{Q}_{\geq 0}$ such $(s_1, s_2) \xrightarrow{\alpha} (s'_1, s'_2)$ in $[\![(N_1, R_1)]\!] \| [\![(N_2, R_2)]\!]$. We have also three cases (Def. 17):

1. For $\alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\varepsilon\}$ and $s_1 \xrightarrow{\alpha}_1 s'_1$ thus there is a relation $r \in R_1$ such $\mathcal{L}(r) = \alpha$, $\forall t \in r$, t is enabled at m, and $0 \in \varphi(t)$. By Definition 26 the relation r is also in R. Thus, it exists a state $s \in S$ and a state $s' = s'_1 \uplus s_2$ such $s \xmapsto{\alpha} s'$ (Def. 25).

2. For $\alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\varepsilon\}$ and $s_2 \xrightarrow{\alpha}_2 s'_2$, we have exactly the same reasoning.

3. For $\alpha \neq \varepsilon$ and $s_i \xrightarrow{\alpha}_i s'_i$, we can distinguish two other cases:

   - For $\alpha \in \mathbb{Q}_{\geq 0}$ we have $s'_i = (m_i, \varphi_i \dot{-} \alpha), i \in 1..2$ and so we have a relation in $[\![(N, R)]\!]$ between s and a state $s' = (m, \varphi'_1 \uplus \varphi'_2) = (s'_1, s'_2)$ and so $s \xmapsto{\alpha} s'$ exists.
   - For $\alpha \in \Sigma_{1,2}$, we have by definition two relations $r_1 \in R_1$ and $r_2 \in R_2$, such $\mathcal{L}(r_1) = \mathcal{L}(r_1) = \alpha$ and $\forall t \in r_1 \cup r_2$, t is enabled at m, and $0 \in \varphi(t)$. By Definition 26, it exists a relation $r = r_1 \cup r_2$ in R, so a relation in $[\![(N, R)]\!]$ between s and a state $s' = (m'_1 \uplus m'_2, \varphi'_1 \uplus \varphi'_2) = (s'_1, s'_2)$ and so $s \xmapsto{\alpha} s'$ exists.

□

4.5 L-observable Executions

For a PTPN with a set of labels Σ, a set of observable labels $L \subseteq \Sigma$ is defined, and the *L-observation* for an execution $\sigma = \alpha_1 \dots \alpha_k$ is defined as the sequence $\text{obs}_L(\alpha_1) \dots \text{obs}_L(\alpha_k)$ such that $\text{obs}_L(\alpha) = \alpha$ when $\alpha \in \mathbb{Q}_{\geq 0} \cup L$ and $\text{obs}_L(\alpha) = 0$ otherwise. Hence $\text{obs}_L(\sigma)$ is an execution that contains only the observable events in σ, in the same order and at the same dates than in σ.

We will now define two new products over a set of observations.

Definition 27 (L-observable product of two TTS)**.** *Assume $K_1 = \langle S_1, s_1^0, \Sigma_1, \rightarrow_1 \rangle$ and $K_2 = \langle S_2, s_2^0, \Sigma_2, \rightarrow_2 \rangle$ are two TTS and L is a set of (observable) labels ($L \subseteq \Sigma_1 \cap \Sigma_2$). The product of two TTS over L, denoted $K_1 \|_L K_2$, is the TTS $\langle (S_1 \times S_2), (s_1^0, s_2^0), \Sigma; \rightarrow \rangle$ with $\Sigma = \Sigma_1 \cap \Sigma_2$ such that $\rightarrow$ is the smallest relation*

obeying the following rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad \alpha \in (\Sigma \setminus L) \cup \{\varepsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2' \quad \alpha \in (\Sigma \setminus L) \cup \{\varepsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \qquad s_2 \xrightarrow{\alpha}_2 s_2' \qquad \alpha \in \mathbb{Q}_{\geq 0} \cup L}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

So the "common observations" in the two nets N_1 and N_2 relative to common observations L are exactly the observations in the TTS product $[\![N_1]\!] \|_L [\![N_2]\!]$. This has a direct application when there is the intention to find a critical pair in a TPN N, since it amounts to finding an observation in $[\![N_1]\!] \|_L [\![N_2]\!]$ where the first component had an occurrence of a fault and not the second.

Theorem 4. *There is an execution σ in $K_1 \|_L K_2$ if and only if there are two executions, σ_1 in K_1 and σ_2 in K_2, with the same observations:* $\text{obs}_L(\sigma) \equiv \text{obs}_L(\sigma_1) \equiv \text{obs}_L(\sigma_2)$.

Proof. Given an execution $\sigma = \alpha_1 ... \alpha_n$ in $K_1 \|_L K_2$, we define two new sequences $\sigma_i = \#_i(\sigma)$ $(i \in \{1, 2\})$ as $\#_i(\alpha_1) ... \#_i(\alpha_n)$ such $\#_i(\alpha) = \alpha$ if $s_i \xrightarrow{\alpha}_i s_i'$ and $\#_i(\alpha) = 0$ otherwise.

Suppose $\#_1(\alpha_1 ... \alpha_k)$ in K_1, and $K_1 \|_L K_2$ is in state (s_1, s_2) after the execution of $\alpha_1 ... \alpha_k$. If $\#_1(\alpha_{k+1}) = \alpha_{k+1}$, then the execution $...\alpha_k\alpha_{k+1}$ is in K_1, else $\#_1(\alpha_{k+1}) = 0$ and by Definition 27 the new state is (s_1, s_2'), then $...\alpha_k\alpha_{k+1}$ is also in K_1. The reasoning is the same with $\#_2$ and K_2. Recursively, for an execution σ from $K_1 \|_L K_2$, we have $\sigma_i = \#_i(\sigma)$ in K_i.

Moreover, from the rules in Definition 27 we have $p_i(\alpha) = \alpha$ for all $\alpha \in \mathbb{Q}_{\geq 0} \cup L$ and thus $\text{obs}_L(\alpha) = \text{obs}_L(\#_i(\alpha))$. So we have $\text{obs}_L(\sigma) \equiv \text{obs}_L(\sigma_i)$.

Reciprocally, consider any pair of executions σ_1 and σ_2 with $\text{obs}_L(\sigma_1) \equiv \text{obs}_L(\sigma_2)$. An execution σ_i can be decomposed in $\sigma_{i,1}\alpha_1\sigma_{i,2}\alpha_2 ... \sigma_{i,n}\alpha_n\sigma_{i,n+1}$ where $\sigma_{i,k}$ are executions such $\text{obs}(\sigma_{i,k}) \equiv 0$ and $\alpha_k \in \mathbb{Q}_{\geq 0} \cup L$. From the Definition 27, the execution $\sigma = \sigma_{1,1}\sigma_{2,1}\alpha_1\sigma_{1,2}\sigma_{2,2}\alpha_2 ... \sigma_{1,n}\sigma_{2,n}\alpha_n\sigma_{1,n+1}\sigma_{2,n+1}$ is feasible for $K_1 \|_L K_2$ and $\text{obs}_L(\sigma) \equiv \text{obs}_L(\sigma_1) \equiv \text{obs}_L(\sigma_2)$. □

As for the product of a TTS we now define the synchronous product of TTS over a set of observables.

Definition 28 (L-observable product of PTPN)**.** *Given two PTPN (N_1, R_1) and (N_2, R_2) with sets of places P_1, P_2 and transitions T_1, T_2, their product over a set of observables $L \subseteq \Sigma_1 \cap \Sigma_2$, $(N_1, R_1) \times_L (N_2, R_2)$, is the PTPN (N, R) where N is the concurrent composition (juxtaposition) of N_1 with N_2 the net $\langle P_1 \cup P_2, T_1 \cup T_2, \mathbf{Pre}, \mathbf{Post}, m_0^1 \uplus m_0^2, \mathbf{I}_s \rangle$ with $\mathbf{Pre}(t)(p) = \mathbf{Pre}_i(t)(p)$ if and only if $t \in T_i$ and $p \in P_i$ with $i \in 1..2$, and 0 otherwise (same with $\mathbf{Post}$); and the product*

relation R is such that:

$$R = \bigcup_{a \in L} \{r_1 \cup r_2 \mid r_i \in R_i, \mathcal{L}(r_i) = a, i \in 1..2\} \cup \bigcup_{a \in \Sigma \setminus L \cup \{\varepsilon\}} \{r \mid r \in R_1 \cup R_2, \mathcal{L}(r) = a\}$$

with $\Sigma = \Sigma_1 \cup \Sigma_2$.

Theorem 5. *The State graph of $[\![(N_1, R_1) \times_L (N_2, R_2)]\!]$ is isomorphic to $[\![(N_1, R_1)]\!] \|_L [\![(N_2, R_2)]\!]$.*

Proof. The proof is exactly the same than Theo. 3 and can be done by induction on the shortest path from the initial state, s_0, to a reachable state s in $[\![N_1 \times_L N_2]\!]$, then a case analysis on the possible transitions from s. □

4.6 Parametric run of a PTPN

As for TPN, a more compact representation of the state space of a PTPN than with a TTS can be defined by using parametric state. Here, the definition is adapted to the PTPN.

Definition 29 (Parametric state and parametric run for PTPN)**.** *Let (N, R) a PTPN with $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ be a TPN and let $\sigma = t_1 \dots t_n$ be a firing sequence in N. Then, the* parametric run *$(\sigma(x), B_\sigma)$ of σ in N with $\sigma(x) = x_0 t_1 x_1 \dots x_{n-1} t_n x_n$ and the* parametric state *(z_σ, B_σ) in N are recursively defined as follows:*

Basis:

- *$\sigma = \varepsilon$, i.e, $\sigma(x) = x_0$.*
- *$z_\sigma = (m_\sigma, h_\sigma)$ and B_σ are defined as followed:*
  - $m_\sigma = m_0$
  - $h_\sigma(t) = \begin{cases} x_0 & \text{if} \quad t \in \mathcal{E}(m_\sigma) \\ \sharp & \text{otherwise} \end{cases}$
  - $B_\sigma = \{0 \leq h_\sigma(t) \leq \beta_t^s \mid t \in \mathcal{E}(m_\sigma)\}$

Step: *Assume that z_σ and B_σ are already defined for the sequence $\sigma = t_1 \dots t_n$. For $\sigma = t_1 \dots t_n t_{n+1}$ with $t_{n+1} \in r, r \in R$, $w = t_1 \dots t_n$ (we have $\sigma = w t_{n+1}$) is defined and set:*

- $m_\sigma = m_w - \sum_{t \in r} \mathbf{Pre}(t) + \sum_{t \in r} \mathbf{Post}(t)$
- $h_\sigma(t) = \begin{cases} \sharp & \text{if } t \notin \mathcal{E}(m_\sigma) \\ h_w(t) + x_{n+1} & \text{if } t \in \mathcal{E}(m_\sigma) \cap \mathcal{E}(m_w) \wedge m - \sum_{k \in r} \mathbf{Pre}(k) \dot{\geqslant} \mathbf{Pre}(t) \wedge t \notin r \\ x_{n+1} & \text{otherwise.} \end{cases}$
- $B_\sigma = B_w \cup \{\alpha_t^s \leq h_w(t) \mid t \in r\} \cup \{0 \leq h_\sigma(t) \leq \beta_t^s \mid t \in \mathcal{E}(m_\sigma)\}$

4.7 Summary

In this Chapter 4 we presented our first contribution, the PTPN, explaining its behaviour regarding the synchronous product of TPN.

- *PTPN* is composed of places, transitions and timing constraint just like TPN. The idea is to compose TPN by synchronizing transitions on their *labels*. If a label was in two (or more) original TPN, there will be a synchronized firing of the transitions regardless of their timing constraints.
- *Timelock* is a result of this synchronization. If the timing constraints of two (or more) transitions do not have a common solution, it will end up in a time deadlock.
- *TTS* product and *PTPN* are equivalent in terms of executions.
- *Diagnosability* will be based on the product and the new *timelock* behaviour which shows timing constraint differences between the original TPN of the PTPN.

The Chapter 5 will be focused on the state class graph resulting from a TPN and from a PTPN since we mainly check properties on the state graph of our DES.

Chapter 5

SCG and SSCG

We have seen through the two last chapters that TPN and PTPN have slight differences in terms of semantics and behaviour. When the objective is to study a properties based on a DES, its State Class Abstraction is analysed. The State Class is a representation of a TPN in terms of the marking and the firing domain in a finite abstraction.

State Classes can be divided in two kinds, the classical representation of a State Class (based on the firing domain) and the Strong State Class (based on clock representation). They both have their value in terms of representation and properties that can be checked, but the classical representation is generally smaller than the Strong representation.

In the following section we discuss the State Class Abstraction, first focusing on TPN then on PTPN, which have a slightly different behaviour because of the synchronization.

5.1 The State Class Abstraction for TPN

In the remainder of this section, the notation α_t^s and β_t^s are for the left and right endpoints of $\mathbf{I}_s(t)$ (that is $\downarrow\mathbf{I}_s(t)$ and $\uparrow\mathbf{I}_s(t)$ respectively, see Section 3.1 for details). By definition: $0 \leq \alpha_t^s \leq \beta_t^s$. As a convention, it is considered that $\beta_t^s - \alpha_t^s = \infty$ if β_t^s is infinite. Likewise, inequalities of the form $x \leq \infty$ (which is a tautology) are implicitly accepted. The notation of α_{t_i} into α_i is also simplified when it is non-ambiguous.

In this section, the results on the state class abstraction method for TPN defined by Berthomieu et al. [Berthomieu 1983, Berthomieu 1991] are recalled. A State Class Graph (SCG) is a finite abstraction of the timed transition system of a net that preserves the markings and the traces. The construction is based on the idea that temporal information in states (the firing domain φ) can be conveniently represented using systems of difference constraints [Ramalingam 1995].

5.1.1 State classes

Definition 30 (State class)**.** *A state class C is defined by a tuple (m, D) where m is a marking and D is the firing domain described by a finite system of linear inequalities.*

The domain D is defined by a set of difference constraints in reduced form*, that*

are inequalities of the kind:

$$\begin{cases} \alpha_i \leq x_i & t_i \in \mathcal{E}(m) \\ x_i \leq \beta_i & t_i \in \mathcal{E}(m) \\ x_i - x_j \leq \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \end{cases}$$

and the coefficients α_i, β_i and γ_{ij} are rational numbers.

In a domain D, the variables x_i denote the firing time of the enabled transition t_i relatively to the time when the marking of the class occurred. Consequently, they denote a constraint on the value of $\varphi(t_i)$ (see Section 3.1 for details).

The system D is *consistent* by construction in the case of a TPN. This means that the system needs to follow the following conditions:

$$\begin{array}{rcll} \alpha_i & \leq & \beta_i & (C1) \\ -\gamma_{ji} & \leq & \gamma_{ij} & (C2) \end{array}$$

with (i, j) ranging over the set of transitions $\mathcal{E}(m)$ (with $i \neq j$).

Moreover, the form of D can be improved by choosing the tightest possible bounds that do not change its associated solutions set. In this case, D is in *closure form.*

Definition 31 (Closure form)**.** *A difference system is in closure form iff it is closed by recursivity, that is iff it is in normal form and its constraints are the tightest preserving its solution set.*

The closure form of a firing domain D of a class C is [Berthomieu 1983]:

$$D^* = \begin{cases} \alpha_i^* \leq x_i & t_i \in \mathcal{E}(m) \\ x_i \leq \beta_i^* & t_i \in \mathcal{E}(m) \\ x_i - x_j \leq \gamma_{ij}^* & i \neq j, t_i, t_j \in \mathcal{E}(m) \end{cases}$$

where

- *α_i^* is the smallest possible value of variable x_i solution of D, i.e. $\alpha_i^* = \inf\{x_i \mid x_i$ a solution of $D\}$,*
- *β_i^* is the largest possible value of variable x_i solution of D, i.e. $\beta_i^* = \sup\{x_i \mid x_i$ a solution of $D\}$,*
- *γ_{ij}^* is the largest possible value of the difference $x_i - x_j$, i.e. $\gamma_{ij}^* = \sup\{x_i - x_j \mid x_i, x_j$ a solution of $D\}$.*

A result in [Aspvall 1979] implies that this closure form can be computed in polynomial time with a shortest-path graph algorithm, for example with the Floyd-Warshall algorithm that has an $O(n^3)$ time cost.

Lemma 1. *A system D is in closure form iff $\forall i, i, k$:*

$$\begin{array}{rcll} \alpha_i & \geq & \alpha_k - \gamma_{ki} & \textit{(C3)} \\ \beta_i & \leq & \gamma_{ik} + \beta_k & \textit{(C4)} \\ \gamma_{ij} & \leq & \beta_i - \alpha_j & \textit{(C5)} \\ \gamma_{ij} & \leq & \gamma_{ik} + \gamma_{kj} & \textit{(C6)} \end{array}$$

Proof. Suppose the system D in closure form, with;

$$D = \left\{ \begin{array}{rcl} \alpha_i & \leq & x_i \\ x_i & \leq & \beta_i \\ x_i - x_j & \leq & \gamma_{ij} \end{array} \right.$$

From this system we have the following relations:

$$\begin{array}{rcccl} \alpha_k - \gamma_{ki} & \leq & x_k - \gamma_{ki} & \leq & x_i \\ x_i & \leq & x_k + \gamma_{ik} & \leq & \beta_k + \gamma_{ik} \\ x_i - x_j & \leq & x_i - x_k + x_k - x_j & \leq & \gamma_{ik} + \gamma_{kj} \\ x_i - x_j & \leq & \beta_i - \alpha_j & & \end{array}$$

and by definition of α, β and γ as infimum or supremum for a system in closure form, we have directly $D \Rightarrow (C3), (C4), (C5)$ and $(C6)$.

Reciprocally, as for Difference-Bound Matrix (DBM), we introduce a variable $x_0 = 0$ to rewrite inequalities of D:

$$D = \left\{ \begin{array}{rcl} x_0 - x_i & \leq & -\alpha_i \\ x_i - x_0 & \leq & \beta_i \\ x_i - x_j & \leq & \gamma_{ij} \end{array} \right.$$

and so by denoting $\gamma_{0i} = -\alpha_i$ and $\gamma_{i0} = \beta_i$ the system can be viewed as a set of inequalities $x_i - x_j \leq \gamma_{ij}$ and condition (C3), (C4), (C5) and (C6) can be summarized with (C6b) $\gamma_{ij} \leq \gamma_{ik} + \gamma_{kj}$ extended with x_0.

So now suppose that we have (C6b) $\gamma_{ij} \leq \gamma_{ik} + \gamma_{kj}$ and that D is not in closure form. It means $\exists \gamma, x_i - x_j \leq \gamma < \gamma_{ij}$ and so:

$$\begin{array}{rcl} x_i - x_k + x_k - x_j & < & \gamma_{ij} \\ \gamma_{ik} + \gamma_{kj} & < & \gamma_{ij} \\ \gamma_{ij} & < & \gamma_{ij} \end{array}$$

So $(C6b) \Rightarrow D$ in closure form. □

During the process to compute a SCG it is necessary to compare two classes. We introduce the notation L_D that represents the set of all solutions for x that satisfies the inequalities of D and, so, we can define the equality of two classes.

Definition 32 (Equality of two classes)**.** *Two classes* $C_1 = (m_1, D_1)$ *and* $C_2 = (m_2, D_2)$ *are equal if* $m_1 = m_2$ *and* $L_{D_1} = L_{D_2}$.

Remark that checking firing domains for equality is improved by the fact that the domains are in closure form (see [Berthomieu 1983] for details).

5.1.2 Transitions between state classes

We quickly explain here how the domain of a class can be obtained after firing a transition. These steps are taken from [Berthomieu 1983, Berthomieu 1991].

Initial state class

The initial class C_ε is (m_0, D_0) where m_0 is the initial marking and D_0 is the domain defined by the set of static time constraints:

$$D_0 = \begin{cases} \alpha_i^s & \leq & x_i & t_i \in \mathcal{E}(m_0) \\ x_i & \leq & \beta_i^s & t_i \in \mathcal{E}(m_0) \\ x_i - x_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m_0) \end{cases}$$

where $\mathcal{E}(m_0)$ is the set of enabled transitions at m_0 (see Section 3.1 for details) and $\gamma_{ij} = \beta_i^s - \alpha_j^s, i \neq j, t_i, t_j \in \mathcal{E}(m_0)$

Firability condition

Theorem 6. *Assume* $C_\sigma = (m, D)$ *is defined and* D *is consistent and in closure form and that a transition* t_j *is enabled at* m*. A transition* t_f *is firable iff*

$$\forall t_j \in \mathcal{E}(m), t_j \neq t_f \quad \gamma_{jf} \geq 0 \tag{FIRE}$$

Proof. A transition t_f can be fired with the constraints in D iff there exists a time $\theta \geq 0$ such that the system D is consistent and augmented with the constraints:

(1) $x_f = \theta$

(2) $\theta \leq x_j$ for all transitions $t_j \in \mathcal{E}(m), t_j \neq t_f$

Condition (1) means that the transition t_f is time-enabled after a time θ has elapsed, while condition (2) states that the deadline of any enabled transition is not exceeded.

By eliminating the variable θ, t_f can be fired iff the constraints $x_f \leq x_j$ are consistent for all transitions $t_j \neq t_f$ that is enabled at m. Since the system is in closure form $x_j - x_f \leq \gamma_{jf}$:

$$\forall t_j \in \mathcal{E}(m), j \neq f \quad \gamma_{jf} \geq 0$$

□

Successor class

From a class $C_\sigma = (m, D)$, if condition (FIRE) is true for the transition t_f, $C_{\sigma.t_f} = (m', D')$ is added as the successor class from C_σ, where m' is the result of firing t_f from m:

$$m' = m - \mathbf{Pre}(t_f) + \mathbf{Post}(t_f)$$

Below, the procedure to compute D' from D is briefly described in four steps:

1. The (FIRE) conditions for t_f stated above are added to D.

2. New variables $x'_k = x_k - x_f$ are used in the set of inequalities. The variable x'_k matches the earliest firing date of k at the time t_f fires, that is, the possible values of the time interval $\varphi(k)$ used in Definition 8, case (i). A set of inequalities is obtained where all occurrences of the variables x_k (and x_f in consequence) can be eliminated.

3. The variables for transitions in conflict with t_f are removed, so that the variables only range over transitions enabled at m' (transition t_f is included).

4. For every transition t_k newly enabled at m' after firing t_f, (we denote this set as $nnbl(m, t_f)$) the constraint $\alpha^s_k \leq x'_k \leq \beta^s_k$ is added and further inequalities are provided to take into account the relationship between these new variables and the persistent ones. These constraints match the fact that the firing interval of a newly enabled transition t_k is equal to $\mathbf{I}_s(k)$.

As a result, we obtain a set of inequalities where we can eliminate all occurrences of the variables x_k and x_f. After removing redundant inequalities and simplifying the constraints on transitions in conflict with t_f—so that the variables only range over transitions enabled at m'—we obtain a domain D' that is also in closure form. The coefficients α'_i, β'_i and γ'_{ij} of this set of difference constraints can be defined directly from the coefficients of D.

$$\alpha'_i = \begin{cases} \alpha^s_i & \text{if } t_i \in \text{nnbl}(m, t_f) \\ \max(\{0\} \cup \{-\gamma_{ki} \mid t_k \in \mathcal{E}(m)\}) & \text{otherwise} \end{cases}$$

$$\beta'_i = \begin{cases} \beta^s_i & \text{if } t_i \in \text{nnbl}(m, t_f) \\ \gamma_{if} & \text{otherwise} \end{cases}$$

$$\gamma'_{ij} = \begin{cases} \beta'_i - \alpha'_j & \text{if either } t_i \text{ or } t_j \text{ in nnbl}(m, t_f) \\ \min(\gamma_{ij}, \beta'_i - \alpha'_j) & \text{otherwise} \end{cases}$$

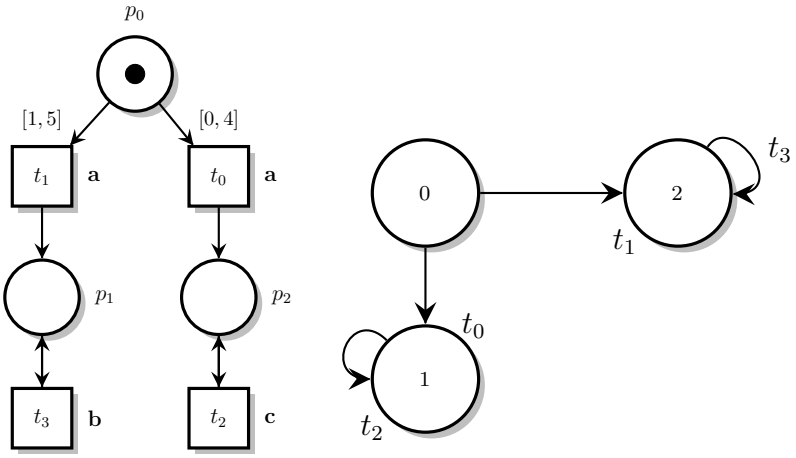
We do not give the demonstration of this calculation for a PTN because we will detail this calculation for a PTPN in the next section. The result has been proved by H. Bouchened. As we have seen before, a TPN can be represented by a PTPN, so the computation of the SCG for a PTPN is a generalization.

For the same reason we do not give here properties attached to SCG for a TPN such as reachability or trace preservation. All these properties will be given in the

next Section for PTPN.

To compute a SCG from a TPN, we can use the software TINA [Berthomieu 2004] [1]. TINA (TIme petri Net Analyzer) is a toolbox for the editing and analysis of Petri Nets. TINA can construct various state space abstractions. Depending on the selected option, the construction preserves markings, states, LTL properties, or CTL properties of the concrete state space of the Time Petri net. To compute a SCG, we use the default option and, as we will see in Chapter 8, some other tools from the TINA toolbox to analyse and transform TPN.

Example: Through this chapter we will use the TPN described in Fig. 5.1a to show the differences between the different State Class models.



(a) TPN (b) SCG

Figure 5.1: Exemple of a TPN N and its SCG

Fig. 5.1b shows the SCG obtained for the TPN. It can be seen that the different transitions are being fired, without timing information on the SCG.

The initial state $s_0 = (m_0, D_0)$ is composed as follow:

- $m_0 = \{1, 0, 0\}$
- $D_0 = \begin{cases} 0 \leq x_0 \\ 1 \leq x_1 \\ x_0 \leq 4 \\ x_1 \leq 5 \\ x_0 - x_1 \leq 3 \\ x_1 - x_0 \leq 5 \end{cases}$

[1] TINA Website : http://projects.laas.fr/tina/

Transitions t_0 or t_1 may fire if their firing constraints are fulfilled in terms of time conditions. By firing either one, a trivial domain is obtained (in either $\{1\}$ or $\{2\}$).

By firing transition t_0, a new class is obtained with :

- $m_1 = \{0, 0, 1\}$
- $D_1 = \begin{cases} 0 & \leq x_2 \\ x_2 & \leq \infty \end{cases}$

By using TINA on this example, we can obtain exactly the same classes:

```
class 0
    marking
p0
    domain
0 <= t0 <= 4
1 <= t1 <= 5

class 1
    marking
p2
    domain
0 <= t2

class 2
    marking
p1
    domain
0 <= t3
```

5.2 The State Class Abstraction Revisited

In this section, we generalize the SCG abstraction to the case of PTPN.

5.2.1 Definition of a state class for a PTPN

The adaptation is quite trivial since a PTPN (N, R) structure is really similar to a TPN structure N (see Chapter 4 for more details). In this section, the idea is to focus on the main difference in terms of behaviour between PTPN and TPN, the subset of transitions which have to fire synchronously. Each step will be repeated and adapted to the PTPN.

Initial state class

The initial class C_0 is (m_0, D_0) where m_0 is the initial marking and D_0 is the same than for TPN :

$$D_0 = \begin{cases} \alpha_i^s & \leq & x_i & t_i \in \mathcal{E}(m_0) \\ x_i & \leq & \beta_i^s & t_i \in \mathcal{E}(m_0) \\ x_i - x_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m_0) \end{cases}$$

where $\mathcal{E}(m_0)$ is the set of enabled transitions at m_0 with $\gamma_{ij} = \beta_i^s - \alpha_j^s, i \neq j, t_i, t_j \in \mathcal{E}(m_0)$.

Firability condition

Theorem 7. *Consider a PTPN (N, R) and assume $C_\sigma = (m, D)$ defined and D consistent and in closure form, a set of transitions $r \in R$ is firable at m iff $r \subseteq \mathcal{E}(m)$ and:*

$$\forall t_j \in \mathcal{E}(m) \setminus r, \min\{\gamma_{jf} \mid t_f \in r\} \geq 0 \qquad \text{(FIRE)}$$

Proof. A set of transitions $r \in R$ can be fired with the constraints in D iff all transitions in r are enabled (i.e. $\forall t_f \in r, t_f \in \mathcal{E}(m)$) and there exists a time $\theta \geq 0$ such that the system D is consistent and augmented with the constraints

$$\begin{cases} x_f & = & \theta & t_f \in r \\ \theta & \leq & x_j & t_j \in \mathcal{E}(m) \setminus r \end{cases}$$

And so :

$$\begin{cases} x_f & = & x_{f'} & t_f, t_{f'} \in r \\ x_f & \leq & x_j & t_j \in \mathcal{E}(m) \setminus r, t_f \in r \end{cases}$$

Since the system is in a closure form, i.e. $x_j - x_f \leq \gamma_{jf}, \forall t_j \in \mathcal{E}(m) \setminus r, \forall t_f \in r$, the set of transitions r is firable from the state (m, D) iff:

$$\forall t_j \in \mathcal{E}(m) \setminus r, \min\{\gamma_{jf} \mid t_f \in r\} \geq 0$$

□

Successor class

From a class $C_\sigma = (m, D)$, if the condition (FIRE) is true for the set of synchronous transitions r, $C_{\sigma.r} = (m', D')$ is added as the successor class from C_σ, where m' is the result of firing transitions of r from m:

$$m' = m - \sum_{t \in r} \mathbf{Pre}(t) + \sum_{t \in r} \mathbf{Post}(t)$$

Now we will proceed with the four same steps as with TPN to construct D'.

1. The conditions to fire transitions in r are added to D:

$$\bar{D} = \left\{ \begin{array}{rcll} x_f & = & x_{f'} & t_f, t_{f'} \in r \\ x_f & \leq & x_j & t_j \in \mathcal{E}(m) \setminus r, t_f \in r \\ \alpha_j & \leq & x_j & t_j \in \mathcal{E}(m) \\ x_j & \leq & \beta_j & t_j \in \mathcal{E}(m) \\ x_i - x_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

By introducing a new variable x_t to replace all variables $x_f, t_f \in r$, a new system is obtained:

$$\bar{D} = \left\{ \begin{array}{rcll} x_t & = & x_f & t_f \in r \\ x_t & \leq & x_j & t_j \in \mathcal{E}(m) \setminus r \\ \max\{\alpha_f \mid t_f \in r\} & \leq & x_t & \\ x_t & \leq & \min\{\beta_f \mid t_f \in r\} & \\ 0 & \leq & \gamma_{ff'} & f \neq f', t_f, t_{f'} \in r \\ x_t - x_j & \leq & \min\{\gamma_{fj} \mid t_f \in r\} & t_j \in \mathcal{E}(m) \setminus r \\ x_i - x_t & \leq & \min\{\gamma_{if} \mid t_f \in r\} & t_i \in \mathcal{E}(m) \setminus r \\ \alpha_j & \leq & x_j & t_j \in \mathcal{E}(m) \setminus r \\ x_j & \leq & \beta_j & t_j \in \mathcal{E}(m) \setminus r \\ x_i - x_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array} \right.$$

Remark that we have for all transitions $t_k \in \mathcal{E}(m) \setminus r$:

$$x_t - x_j \leq x_k - x_j \leq \gamma_{kj}$$

and so

$$x_t - x_j \leq \min\{\gamma_{kj} \mid t_k \in \mathcal{E}(m)\}$$

2. Consider now a change of variable such as:

$$x'_j = x_j - x_t$$

$$\bar{D} = \left\{ \begin{array}{rcll} 0 & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ \max\{\alpha_f \mid t_f \in r\} & \leq & x_t & \\ x_t & \leq & \min\{\beta_f \mid t_f \in r\} & \\ 0 & \leq & \gamma_{kl} & k \neq l, t_k, t_l \in r \\ \max\{-\gamma_{kj} \mid t_k \in \mathcal{E}(m)\} & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ x'_j & \leq & \min\{\gamma_{jf} \mid t_f \in r\} & t_j \in \mathcal{E}(m) \setminus r \\ \alpha_j - x_t & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ x'_j & \leq & \beta_j - x_t & t_j \in \mathcal{E}(m) \setminus r \\ x'_i - x'_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array} \right.$$

Remark that

$$\begin{array}{rcl} x_t & \leq & \min\{\beta_f \mid t_f \in r\} \\ -x_t & \geq & -\min\{\beta_f \mid t_f \in r\} \\ \alpha_j - x_t & \geq & \alpha_j - \min\{\beta_f \mid t_f \in r\} \\ \alpha_j - x_t & \geq & \alpha_j + \max\{-\beta_f \mid t_f \in r\} \\ \alpha_j - x_t & \geq & \max\{\alpha_j - \beta_f \mid t_f \in r\} \end{array}$$

and

$$\begin{array}{rcl} \max\{\alpha_f \mid t_f \in r\} & \leq & x_t \\ -\max\{\alpha_f \mid t_f \in r\} & \geq & -x_t \\ \beta_j - \max\{\alpha_f \mid t_f \in r\} & \geq & \beta_j - x_t \\ \beta_j + \min\{-\alpha_f \mid t_f \in r\} & \geq & \beta_j - x_t \\ \min\{\beta_j - \alpha_f \mid t_f \in r\} & \geq & \beta_j - x_t \end{array}$$

And we have:

$$\begin{array}{c} \alpha_i \leq x'_i + x_t \leq \beta_i \\ \alpha_i - x'_i \leq x_t \leq \beta_i - x'_i \\ \alpha_j - x'_j \leq x_t \leq \beta_j - x'_j \end{array}$$

and so:

$$\begin{array}{rcl} x'_i - x'_j & \leq & \beta_i - \alpha_j \\ x'_j - x'_i & \leq & \beta_j - \alpha_i \end{array}$$

By eliminating x_t the system $\bar{D}$ becomes:

$$\bar{D} = \left\{ \begin{array}{rcll} 0 & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ \max\{-\gamma_{kj} \mid t_k \in \mathcal{E}(m)\} & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ \max\{\alpha_j - \beta_f \mid t_f \in r\} & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ x'_j & \leq & \min\{\gamma_{jf} \mid t_f \in r\} & t_j \in \mathcal{E}(m) \setminus r \\ x'_j & \leq & \min\{\beta_j - \alpha_f \mid t_f \in r\} & t_j \in \mathcal{E}(m) \setminus r \\ x'_i - x'_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \\ x'_i - x'_j & \leq & \beta_i - \alpha_j & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array} \right.$$

By considering that the system D is in closure form we have from (C5) $\gamma_{ij} \leq \beta_i - \alpha_j$, and so we can eliminate:

$$\begin{array}{rcll} \max\{\alpha_j - \beta_f \mid t_f \in r\} & \leq & x'_j & t_j \in \mathcal{E}(m) \setminus r \\ x'_j & \leq & \min\{\beta_j - \alpha_f \mid t_f \in r\} & t_j \in \mathcal{E}(m) \setminus r \\ x'_i - x'_j & \leq & \beta_i - \alpha_j & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array}$$

and so, we have a new system:

$$\bar{D} = \left\{ \begin{array}{rcll} \alpha'_i & \leq & x'_i & t_i \in \mathcal{E}(m) \setminus r \\ x'_i & \leq & \beta'_i & t_i \in \mathcal{E}(m) \setminus r \\ x'_i - x'_j & \leq & \gamma_{ij'} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array} \right.$$

where:

$$\begin{array}{rcl} \alpha'_i & = & \max(\{0\} \cup \{-\gamma_{ki} \mid t_k \in \mathcal{E}(m)\}) \\ \beta'_i & = & \min\{\gamma_{if} \mid t_f \in r\} \\ \gamma'_{ij} & = & \gamma_{ij} \end{array}$$

3. In this new step, we eliminate variables for transitions in conflict with t_r in r when transitions in r are fired. We denote $\text{cflt}(m,r) = \mathcal{E}(m) \setminus \{t_i \in T \mid m - \sum_{t_f \in r} \mathbf{Pre}(t_f) \dot{\geqslant} \mathbf{Pre}(t_i)\}$ the set of transitions in conflict with all transitions in r for a marking m (this set includes transitions in r).

Consider a transition $t_e \in \text{cflt}(m,r)$ and $i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r)$, we have:

$$\left\{ \begin{array}{rcl} \alpha'_e & \leq & x'_e \\ x'_e & \leq & \beta'_e \\ x'_e - \gamma_{ej} & \leq & x'_j \\ x'_j & \leq & \gamma'_{je} + x'_e \\ x'_i - x'_j & \leq & \gamma'_{ie} + \gamma'_{ej} \end{array} \right.$$

and so by eliminating x'_e, $\bar{D}$ becomes for all transitions $i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r)$:

$$\bar{D} = \left\{ \begin{array}{rcl} \alpha'_j & \leq & x'_j \\ \max\{\alpha'_e - \gamma'_{ej} \mid t_e \in \text{cflt}(m,r)\} & \leq & x'_j \\ x'_j & \leq & \beta'_j \\ x'_j & \leq & \min\{\gamma'_{je} + \beta'_e \mid t_e \in \text{cflt}(m,r)\} \\ x'_i - x'_j & \leq & \gamma'_{ij} \\ x'_i - x'_j & \leq & \min\{\gamma'_{ie} + \gamma'_{ej} \mid t_e \in \text{cflt}(m,r)\} \end{array} \right.$$

We obtain a new system:

$$\bar{D} = \left\{ \begin{array}{rcll} \alpha''_j & \leq & x_i & t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ x_i & \leq & \beta''_j & t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ x_i - x_j & \leq & \gamma''_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \end{array} \right.$$

where:

$$\begin{array}{rcl} \alpha''_i & = & \max(\{\alpha'_i\} \cup \{\alpha'_e - \gamma'_{ei} \mid t_e \in \text{cflt}(m,r)\}) \\ \beta''_i & = & \min(\{\beta'_i\} \cup \{\beta'_e + \gamma'_{ie} \mid t_e \in \text{cflt}(m,r)\}) \\ \gamma''_{ij} & = & \min(\{\gamma'_{ij}\} \cup \{\gamma'_{ie} + \gamma'_{ej} \mid t_e \in \text{cflt}(m,r)\}) \end{array}$$

or

$$\begin{array}{rcl} \alpha''_i & = & \max\{0, -\gamma_{ki}, -\gamma_{ei}, -\gamma_{ke} - \gamma_{ei} \mid t_k \in \mathcal{E}(m), t_e \in \text{cflt}(m,r)\} \\ \beta''_i & = & \min\{\gamma_{ik}, \gamma_{ek} + \gamma_{ie} \mid t_k \in r, t_e \in \text{cflt}(m,r)\} \\ \gamma''_{ij} & = & \min\{\gamma_{ij}, \gamma_{ie} + \gamma_{ej} \mid t_e \in \text{cflt}(m,r)\} \end{array}$$

As the system D is in closure form, we have (C6) $\gamma_{jk} \leq \gamma_{ji} + \gamma_{ik}$, then

$$\begin{array}{rcl} \alpha''_i & = & \max(\{0\} \cup \{-\gamma_{ki} \mid t_k \in \mathcal{E}(m)\}) \\ \beta''_i & = & \min\{\gamma_{ik} \mid t_k \in r\} \\ \gamma''_{ij} & = & \gamma_{ij} \end{array}$$

4. For every newly enabled transition at m', the set is denoted as $nnbl(m,r) = \mathcal{E}(m') \setminus \{t_k \mid m - \sum_{t\in r} \mathbf{Pre}(t) \dot{\geqslant} \mathbf{Pre}(t_k)\}$.

   The constraints $\alpha^s_n \leq x'_n \leq \beta^s_n$ and $x'_n - x'_m \leq \beta^s_n - \alpha^s_m$ are added for each $n \neq m, t_n, t_m \in nnbl(m,r)$ and further inequalities are provided to take into account the relationship between these new variables and the others :

$$\bar{D} = \left\{ \begin{array}{rcll} \alpha''_j & \leq & x_j & t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ \alpha^s_n & \leq & x_n & t_n \in \text{nnbl}(m,r) \\ x_j & \leq & \beta''_j & t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ x_n & \leq & \beta^s_n & t_n \in \text{nnbl}(m,r) \\ x_i - x_j & \leq & \gamma''_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ x_n - x_m & \leq & \beta^s_n - \alpha^s_m & n \neq m, t_n, t_m \in \text{nnbl}(m,r) \\ x_n - x_j & \leq & \beta^s_n - \alpha''_j & t_n \in \text{nnbl}(m,r), t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \\ x_i - x_n & \leq & \beta''_i - \alpha^s_n & t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r), t_n \in \text{nnbl}(m,r) \end{array} \right.$$

   Remark that $x_i - x_j \leq \beta''_i - \alpha''_j$ for all transitions $t_i, t_j \in \mathcal{E}(m) \setminus \text{cflt}(m,r), i \neq j$.

   This system can be seen as:

$$\bar{D} = \left\{ \begin{array}{rcll} \alpha'''_i & \leq & x_i & t_i \in \mathcal{E}(m) \setminus r \\ x_i & \leq & \beta'''_i & t_i \in \mathcal{E}(m) \setminus r \\ x_i - x_j & \leq & \gamma'''_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \setminus r \end{array} \right.$$

   with:

$$\begin{array}{rcl} \alpha'''_i & = & \left\{ \begin{array}{ll} \alpha^s_i & \text{if } t_i \in \text{nnbl}(m,t) \\ \max(\{0\} \cup \{-\gamma_{ki} \mid t_k \in \mathcal{E}(m)\}) & \text{otherwise} \end{array} \right. \\ \beta'''_i & = & \left\{ \begin{array}{ll} \beta^s_i & \text{if } t_i \in \text{nnbl}(m,t) \\ \min\{\gamma_{ik} \mid t_k \in r\} & \text{otherwise} \end{array} \right. \\ \gamma'''_{ij} & = & \left\{ \begin{array}{ll} \beta'''_i - \alpha'''_j & \text{if either } t_i \text{ or } t_j \text{ in nnbl}(m,t) \\ \min(\gamma_{ij}, \beta'''_i - \alpha'''_j) & \text{otherwise} \end{array} \right. \end{array}$$

Lemma 2. *Assume $C = (m, D)$ is a state class with D in closure form. Then, for every set of transitions r such that condition (FIRE) holds, there is a unique class (m', D') obtained from C by firing transitions of r. The domain D' is also in closure form and can be computed incrementally as described above.*

Proof. To prove that the domain D' is in closure form we will evaluate:

$$\gamma'''_{ij} \leq \gamma'''_{ik} + \gamma'''_{kj}$$

Consider that:

- $t_i, t_j, t_k \in \text{nnbl}(m, r)$

$$\begin{cases} \gamma'''_{ij} &= \beta^s_i - \alpha^s_j \\ \gamma'''_{ik} &= \beta^s_i - \alpha^s_k \\ \gamma'''_{kj} &= \beta^s_k - \alpha^s_j \end{cases}$$

$$\gamma'''_{ik} + \gamma'''_{kj} = \beta^s_i - \alpha^s_k + \beta^s_k - \alpha^s_j \overset{(C1:\alpha^s_k \leq \beta^s_k)}{\geq} \beta^s_i - \alpha^s_j$$

- $t_i, t_j \in \text{nnbl}(m, r)$ and $t_k \in \mathcal{E}(m') \setminus \text{nnbl}(m, r)$

$$\begin{cases} \gamma'''_{ij} &= \beta^s_i - \alpha^s_j \\ \gamma'''_{ik} &= \begin{cases} \beta^s_i & \text{if } -\gamma_{lk} \leq 0, \forall t_l \in \mathcal{E}(m) \\ \beta^s_i + \gamma_{lk} & \text{otherwise} \end{cases} \\ \gamma'''_{kj} &= \gamma_{kt} - \alpha^s_j \end{cases}$$

$$\gamma'''_{ik} + \gamma'''_{kj} = \begin{cases} \beta^s_i + \gamma_{kt} - \alpha^s_j \overset{(FIRE:\gamma_{lt} \geq 0)}{\geq} \beta^s_i - \alpha^s_j \\ \beta^s_i + \gamma_{lk} + \gamma_{kt} - \alpha^s_j \overset{(C6:\gamma_{lk} + \gamma_{kt} \geq \gamma_{lr})}{\geq} \beta^s_i + \gamma_{lt} - \alpha^s_j \overset{(FIRE:\gamma_{lr} \geq 0)}{\geq} \beta^s_i - \alpha^s_j \end{cases}$$

- $t_i, t_k \in \text{nnbl}(m, r)$ and $t_j \in \mathcal{E}(m) \setminus \text{nnbl}(m, r)$

$$\begin{cases} \gamma'''_{ij} &= \begin{cases} \beta^s_i & \text{if } -\gamma_{lj} \leq 0, \forall t_l \in \mathcal{E}(m) \\ \beta^s_i + \gamma_{lj} & \text{otherwise} \end{cases} \\ \gamma'''_{ik} &= \beta^s_i - \alpha^s_k \\ \gamma'''_{kj} &= \begin{cases} \beta^s_k & \text{if } -\gamma_{lj} \leq 0, \forall t_l \in \mathcal{E}(m) \\ \beta^s_k + \gamma_{lj} & \text{otherwise} \end{cases} \end{cases}$$

$$\gamma'''_{ik} + \gamma'''_{kj} = \begin{cases} \beta^s_i - \alpha^s_k + \beta^s_k \overset{(C1:\alpha^s_k \leq \beta^s_k)}{\geq} \beta^s_i & \text{if } -\gamma_{lj} \leq 0 \\ \beta^s_i - \alpha^s_k + \beta^s_k + \gamma_{lj} \overset{(C1:\alpha^s_k \leq \beta^s_k)}{\geq} \beta^s_i + \gamma_{lj} & \text{otherwise} \end{cases}$$

- $t_k, t_j \in \text{nnbl}(m, r)$ and $t_i \in \mathcal{E}(m) \setminus \text{nnbl}(m, r)$

$$\begin{cases} \gamma'''_{ij} &= \gamma_{it} - \alpha^s_j \\ \gamma'''_{ik} &= \gamma_{it} - \alpha^s_k \\ \gamma'''_{kj} &= \beta^s_k - \alpha^s_j \end{cases}$$

$$\gamma'''_{ik} + \gamma'''_{kj} = \gamma_{it} - \alpha^s_k + \beta^s_k - \alpha^s_j \overset{(C1:\alpha^s_k \leq \beta^s_k)}{\geq} \gamma_{it} - \alpha^s_j$$

- $t_i \in \text{nnbl}(m,r)$ and $t_k, t_j \in \mathcal{E}(m) \setminus \text{nnbl}(m,r)$

$$\left\{\begin{array}{rcl} \gamma'''_{ij} & = & \left\{\begin{array}{ll} \beta^s_i & \text{if } -\gamma_{lj} \leq 0, \forall t_l \in \mathcal{E}(m) \\ \beta^s_i + \gamma_{lj} & \text{otherwise} \end{array}\right. \\ \gamma'''_{ik} & = & \left\{\begin{array}{ll} \beta^s_i & \text{if } -\gamma_{lk} \leq 0, \forall t_l \in \mathcal{E}(m) \\ \beta^s_i + \gamma_{l'k} & \text{otherwise} \end{array}\right. \\ \gamma'''_{kj} & = & \min(\gamma_{kj}, \beta'''_k - \alpha'''_j) \end{array}\right.$$

  - Suppose $\gamma_{nj} \geq 0, \forall t_n \in \mathcal{E}(m)$, then $\gamma'''_{ij} = \beta^s_i$ and $\gamma'''_{kj} = \min(\gamma_{kj}, \gamma_{kt}) \geq 0$.
    - Suppose that $\gamma_{nk} \geq 0, \forall t_n \in \mathcal{E}(m)$ then $\gamma'''_{ik} = \beta^s_i$ and so we have $\gamma'''_{ik} + \gamma'''_{kj} = \beta^s_i + \min(\gamma_{kj}, \gamma_{kt}) \geq \gamma'''_{ij}$.
    - Suppose that $\gamma'''_{ik} = \beta^s_i + \gamma_{l'k}$, then $\gamma'''_{ik} + \gamma'''_{kj} = \beta^s_i + \gamma_{l'k} + \min(\gamma_{kj}, \gamma_{kt}) \geq \beta^s_i + \min(\gamma_{l'j}, \gamma_{l't}) \geq \beta^s_i = \gamma'''_{ij}$.
  - Suppose $\gamma'''_{ij} = \beta^s_i + \gamma_{lj}$ then $\gamma_{lj} \leq 0$, $\gamma_{lj} \leq \gamma_{nj}, l \neq n, t_l, t_n \in \mathcal{E}(m)$ and $\gamma'''_{ij} = \beta^s_i$ and $\gamma'''_{kj} = \min(\gamma_{kj}, \gamma_{kt} + \gamma_{lj}) \geq \gamma_{lj}$.
    - Suppose that $-\gamma_{nk} \leq 0, \forall t_n \in \mathcal{E}(m)$ then $\gamma'''_{ik} = \beta^s_i$ and so we have $\gamma'''_{ik} + \gamma'''_{kj} = \beta^s_i + \min(\gamma_{kj}, \gamma_{kt} + \gamma_{lj}) \geq \gamma'''_{ij}$.
    - Suppose that $\gamma'''_{ik} = \beta^s_i + \gamma_{l'k}$, then $\gamma'''_{ik} + \gamma'''_{kj} = \beta^s_i + \gamma_{l'k} + \min(\gamma_{kj}, \gamma_{kt} + \gamma_{lj}) \geq \beta^s_i + \min(\gamma_{l'j}, \gamma_{l't} + \gamma_{lj}) \geq \beta^s_i + \gamma_{lj} = \gamma'''_{ij}$.

- $t_j \in \text{nnbl}(m,r)$ and $t_i, t_k \in \mathcal{E}(m) \setminus \text{nnbl}(m,r)$

$$\left\{\begin{array}{rcl} \gamma'''_{ij} & = & \gamma_{it} - \alpha^s_j \\ \gamma'''_{ik} & = & \min(\gamma_{ik}, \beta'''_i - \alpha'''_k) \\ \gamma'''_{kj} & = & \gamma_{kt'} - \alpha^s_j \end{array}\right.$$

  - Suppose $\gamma'''_{ik} = \gamma_{ik}$ then $\gamma'''_{ij} + \gamma'''_{kj} = \gamma_{ik} + \gamma_{kt'} - \alpha^s_j \overset{(C6:\gamma_{ik}+\gamma_{kt'} \geq \gamma_{it'})}{\geq} \gamma_{it'} - \alpha^s_j \overset{(\gamma_{it} = \min\{\gamma_{ik} \mid t_k \in r\})}{\geq} \gamma_{it} - \alpha^s_j$
  - Suppose $\gamma'''_{ik} = \beta'''_i - \alpha'''_k = \gamma_{it} + \min(0, \gamma_{lk})$ then $\gamma'''_{ij} + \gamma'''_{kj} = \gamma_{it} + \min(0, \gamma_{lk}) + \gamma_{kt'} - \alpha^s_j = \gamma_{it} + \min(\gamma_{kt'}, \gamma_{lk} + \gamma_{kt'}) - \alpha^s_j$ and so $\gamma'''_{ij} + \gamma'''_{kj} \overset{(C6:\gamma_{lk}+\gamma_{kt'} \geq \gamma_{lt'})}{\geq} \gamma_{it} - \alpha^s_j + \min(\gamma_{kt'}, \gamma_{lt'}) \overset{(FIRE)}{\geq} \gamma_{it} - \alpha^s_j$

- $t_k \in \text{nnbl}(m,r)$ and $t_i, t_j \in \mathcal{E}(m) \setminus \text{nnbl}(m,r)$

$$\left\{\begin{array}{rcl} \gamma'''_{ij} & = & \min(\gamma_{ij}, \beta'''_i - \alpha'''_j) \leq \beta'''_i - \alpha'''_j \\ \gamma'''_{ik} & = & \beta'''_i - \alpha^s_k \\ \gamma'''_{kj} & = \beta^s_k - \alpha'''_j & \end{array}\right.$$

$$\gamma'''_{ik} + \gamma'''_{kj} = \beta'''_i - \alpha^s_k + \beta^s_k - \alpha'''_j \overset{(C1:\alpha^s_k \leq \beta^s_k)}{\geq} \beta'''_i - \alpha'''_j \geq \gamma'''_{ij}$$

- $t_i, t_j, t_k \in \mathcal{E}(m) \setminus \text{nnbl}(m, r)$

$$\left\{ \begin{array}{lcl} \gamma'''_{ij} & = & \min(\gamma_{ij}, \gamma_{it}, \gamma_{it} + \gamma_{lj}) \\ \gamma'''_{ik} & = & \min(\gamma_{ik}, \gamma_{it}, \gamma_{it} + \gamma_{l'k}) \\ \gamma'''_{kj} & = & \min(\gamma_{kj}, \gamma_{kt'}, \gamma_{kt'} + \gamma_{lj}) \end{array} \right.$$

with $\gamma_{it} = \min\{\gamma_{if} \mid t_f \in r\} \leq \gamma_{it'}$ and $\gamma_{lj} = \min\{\gamma_{nj} \mid t_n \in \mathcal{E}(m)\} \leq \gamma_{l'j}$.
Consider the different values of $\gamma'''_{ik} + \gamma'''_{kj}$:

$$\begin{array}{l} - \ \gamma_{ik} + \gamma_{kj} \overset{(C6:\gamma_{ik}+\gamma_{kj} \geq \gamma_{ij})}{\geq} \gamma_{ij} \\ - \ \gamma_{ik} + \gamma_{kt'} \overset{(C6:\gamma_{ik}+\gamma_{kt'} \geq \gamma_{ij})}{\geq} \gamma_{it'} \overset{(\gamma_{it'} \geq \gamma_{it})}{\geq} \gamma_{it} \\ - \ \gamma_{ik} + \gamma_{kt'} + \gamma_{lj} \overset{(C6:\gamma_{ik}+\gamma_{kt'} \geq \gamma_{ij})}{\geq} \gamma_{it'} + \gamma_{lj} \overset{(\gamma_{it'} \geq \gamma_{it})}{\geq} \gamma_{it} + \gamma_{lj} \\ - \ \gamma_{it} + \gamma_{kj} \overset{(\gamma_{kj} \geq \gamma_{lj})}{\geq} \gamma_{it} + \gamma_{lj} \\ - \ \gamma_{it} + \gamma_{kt'} \overset{(FIRE)}{\geq} \gamma_{it} \\ - \ \gamma_{it} + \gamma_{kt'} + \gamma_{lj} \overset{(FIRE)}{\geq} \gamma_{it} + \gamma_{lj} \\ - \ \gamma_{it} + \gamma_{l'k} + \gamma_{kj} \overset{(C6:\gamma_{l'k}+\gamma_{kj} \geq \gamma_{l'j})}{\geq} \gamma_{it} + \gamma_{l'j} \overset{(\gamma_{l'j} \geq \gamma_{lj})}{\geq} \gamma_{it} + \gamma_{lj} \\ - \ \gamma_{it} + \gamma_{l'k} + \gamma_{kt'} \overset{(C6:\gamma_{l'k}+\gamma_{kt'} \geq \gamma_{l't'})}{\geq} \gamma_{it} + \gamma_{l't'} \overset{(FIRE)}{\geq} \gamma_{it} \\ - \ \gamma_{it} + \gamma_{l'k} + \gamma_{kt'} + \gamma_{lj} \overset{(C6:\gamma_{l'k}+\gamma_{kt'} \geq \gamma_{l't'})}{\geq} \gamma_{it} + \gamma_{l't'} + \gamma_{lj} \overset{(FIRE)}{\geq} \gamma_{it} + \gamma_{lj} \end{array}$$

□

5.2.2 Graph of state classes

Definition 33. *(state class graph) The state class graph (SCG) of a PTPN is obtained by constructing a tree, that we call tree of state classes, with the initial class as root and arcs labelled with transitions in r, going from class C to class C' iff transitions in r are firable from class C and if firing them leads to class C' (Lemma. 2). The SCG is then the merging of equal classes in the tree of state classes.*

Theorem 8. *For a PTPN (N, R), if the state (m, φ) is reachable in the state graph of (N, R), and $(m, \varphi) \xrightarrow{(\theta,t)} (m', \varphi')$ then there are two classes $C = (m, D)$ and $C' = (m', D')$ reachable in the SCG computed for (N, R) with $\varphi \in D$ and $\varphi' \in D'$.*

Proof. We can rewrite a state (m, φ) as a pair (m, A) with A a system of inequalities for $t_i \in \mathcal{E}(m)$ such that:

$$A = \left\{ \downarrow\varphi(t_i) \ \leq \ y_i \ \leq \ \uparrow\varphi(t_i) \quad t_i \in \mathcal{E}(m) \right.$$

and if $(m,\varphi) \xrightarrow{(\theta,t)} (m',\varphi')$ then (m',φ') is equivalent to (m',A') with:

$$A' = \begin{cases} \downarrow\varphi^s(t_i) \leq y'_i \leq \uparrow\varphi^s(t_i) & \text{if } t_i \in \text{nnbl}(m,r) \\ \max(0,\downarrow\varphi(t_i)-\theta) \leq y'_i \leq \uparrow\varphi(t_i)-\theta & \text{if } t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r) \end{cases}$$

and by definition $\max\{\alpha_f \mid t_f \in r\} \leq \theta \leq \min\{\beta_f \mid t_f \in r\}$.

If we suppose that for a state (m,φ) we have a state class $C = (m,D)$ such that $\varphi \in D$ (i.e. y_i is a solution of D) then:

$$\alpha_i \leq \downarrow\varphi(t_i) \leq y_i \leq \uparrow\varphi(t_i) \leq \beta_i$$

For $t_i \in \text{nnbl}(m,r)$, we have $\alpha'''_i = \downarrow\varphi^s(t_i)$ and $\beta'''_i = \uparrow\varphi^s(t_i)$, so:

$$\alpha'''_i \leq y'_i \leq \beta'''_i \text{ if } t_i \in \text{nnbl}(m,r)$$

For $t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r)$ we have:

$$\begin{array}{rcl} \alpha'''_i & = & \max\{0,\gamma_{li}\} = \max(\{0\} \cup \{-\gamma_{ki} \mid t_k \in \text{cflt}(m,t)\}) \\ \beta'''_i & = & \gamma_{il'} = \min\{\gamma_{if} \mid t_f \in r\} \end{array}$$

We have $y'_i \leq \uparrow\varphi(t_i) - \theta \leq \beta_i - \theta$, and as D is in closure form, from (C5):

$$\begin{array}{rcl} \beta_i - \theta & \leq & \gamma_{il'} + \beta_{l'} - \theta \\ & \leq & \gamma_{il'} + \beta_{l'} - \max\{\alpha_f \mid t_f \in r\} \\ & \leq & \gamma_{il'} + \beta_{l'} - \alpha_{l'} \\ & \leq & \gamma_{il'} \end{array}$$

and $y'_i \geq \max(0,\downarrow\varphi(t_i) - \theta) \geq \max(0,\alpha_i - \theta)$, and from (C3):

$$\begin{array}{rcl} \max(0,\alpha_i - \theta) & \geq & \max(0,\alpha_l - \gamma_{li} - \theta) \\ & \geq & \max(0,\alpha_l - \gamma_{li} - \min\{\beta_f \mid t_f \in r\}) \\ & \geq & \max(0,\alpha_l - \gamma_{li} - \beta_l) \\ & \geq & \max(0,\gamma_{li}) \end{array}$$

and so we have $\alpha'''_i \leq y'_i \leq \beta'''_i$ if $t_i \in \mathcal{E}(m) \setminus \text{cflt}(m,r)$.

□

Theorem 9. *For a PTPN (N,R), its TTS $[\![(N,R)]\!]$ has the same set of firing sequences than its SCG.*

Proof. From the definition of the state classes, any sequence of firable transitions from the initial state will be a path in the tree of state classes.

Existence of a path σ from the initial class to a class C' implies that a feasible run with firing sequence σ is feasible from the initial state. □

It can be proved that the *SCG* construction preserves the set of linear time properties.

Theorem 10. *For a PTPN (N, R) with a finite number of reachable markings, a finite SCG can be defined with the same set of reachable markings and the same set of traces than the timed transition system $[\![(N, R)]\!]$.*

Proof. From Th. 9 we have immediately the same set of traces.

The proof on the number of classes in a *SCG* is similar to the Theorem 2 in [Berthomieu 1991]. We give here only the main steps of the reasoning.

From the definition of the firing rule, the constants α_i, β_i and γ_{ij} of any domain are linear combinations with integer coefficients of α_i^s and β_i^s, i.e. :

$$\forall i, \exists \lambda_1, ..., \lambda_{2n} \in \mathbb{Z}, \alpha_i = \lambda_1 \alpha_1^s + ... + \lambda_n \alpha_n^s + \lambda_{n+1} \beta_1^s + ... + \lambda_{2n} \beta_n^s$$

and similarly for β_i and γ_{ij}.

Moreover, from the firing rule, we have bounds on α_i, β_i and γ_{ij} computed from the initial class:

$$\begin{array}{ccccc} 0 & \leq & \alpha_i & \leq & \alpha_i^s \\ 0 & \leq & \beta_i & \leq & \beta_i^s \\ -\alpha_k^s & \leq & \gamma_{jk} & \leq & \beta_j^s \end{array}$$

It is shown in [Berthomieu 1991] (Lemma 4) that for two constants A and B and a finite set of rational constants $Q_1, ..., Q_n$, there is only a bounded number of linear combinations of numbers $Q_1, ..., Q_n$ with integer coefficients, between A and B, i.e., for $\lambda_1, ..., \lambda_n \in \mathbb{Z}$ and $Q_1, ..., Q_n \in \mathbb{Q}$, the number of rational numbers x such that

$$x = \lambda_1 Q_1 + ... + \lambda_n Q_n \text{ and } A \leq x \leq B$$

is bounded.

We have seen that the possible values α_i, β_i and γ_{ij} for systems that define the state classes are linear combinations with integer coefficients of α_i^s and β_i^s and have upper and lower bounds. Immediately, from the previous result, for a given marking there exists only a bounded number of domains related to this marking. So if the PTPN has a finite number of reachable markings then the number of classes, i.e. a pair of marking and domain, is finite. □

5.2.3 Example of a SCG for a PTPN

For the purpose of this example, we will make a PTPN using the same TPN from the example 5.1b, twice. We call the original TPN N and its twin N'.

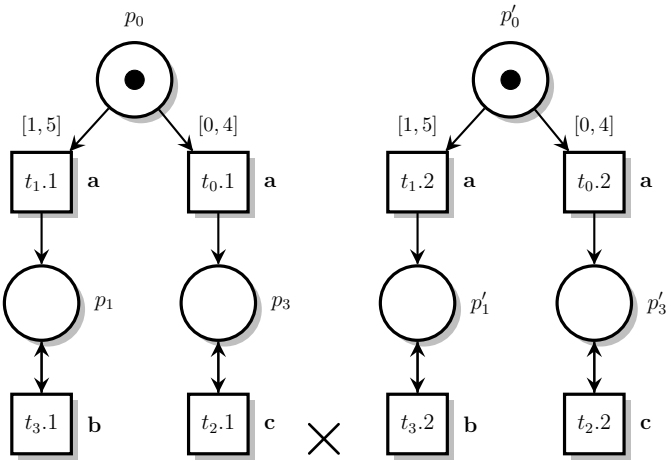


Figure 5.2: PTPN of $N \times N'$

With set of transitions for m_0:

$$\{(t_0.1|t_0.2), (t_1.1|t_0.2), (t_0.1|t_1.2), (t_1.1|t_1.2)\}$$

Using this PTPN we have the following SCG creation.

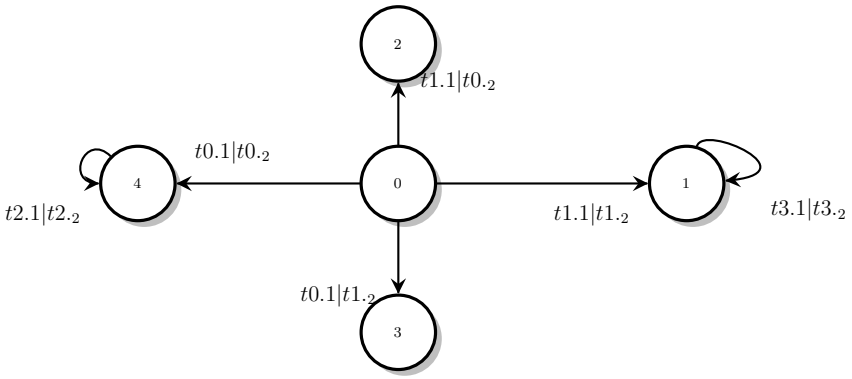


Figure 5.3: SCG of the PTPN 5.2

Here, the four markings, still representing the possible markings in the PTPN, can be seen. As a clear example of a PTPN behaviour, the synchronous firing of the transitions with the same labels can be seen.

The SCG at $\{0\}$ is composed as follow:

- $m_0 = \{p_0, p'_0\}$

- $D = \left\{ \begin{array}{rcl} 0 & \leq x_{t0.1|t0.2} \leq & 4 \\ 1 & \leq x_{t1.1|t1.2} \leq & 5 \\ 1 & \leq x_{t0.1|t1.2} \leq & 4 \\ 1 & \leq x_{t1.1|t0.2} \leq & 4 \\ & \gamma_{t0.1|t0.2,t1.1|t1.2} \leq & 3 \\ & \gamma_{t0.1|t0.2,t0.1|t1.2} \leq & 3 \\ & \gamma_{t0.1|t0.2,t1.1|t0.2} \leq & 3 \\ & \gamma_{t1.1|t1.2,t0.1|t0.2} \leq & 5 \\ \ldots & & \end{array} \right.$

D is of course much bigger because of the synchronous behaviour between the two TPN. This conclude our study of the SCG and we now proceed to the SSCG, another abstraction used for verification purposes.

5.3 The Strong State Class Abstraction for TPN

In this section we present the Strong State Class abstraction introduced by Berthomieu et al. [Berthomieu 1983].

This class relies on clock for time modelling and is defined as *strong* because it keeps much more information (priorities for example) than the general representation of state class. The preservation of branching can be added in a SSCG. Contrary to the SCG we will directly give the form of the SSCG for the PTPN without discussing that of the TPN (a construction of the SSCG for a TPN is given by Berthomieu and Vernadat in [Berthomieu 2003]).

5.3.1 Definition of a strong state class

A strong state class R is represented by a pair (m, Q), where m is a marking and the clock domain Q is described by a (finite) system of linear inequalities. Q is the equivalent of a domain D, but it represents clock counting for every enabled transition.

Initial state class

For a PTPN with initial state $s_0 = (m_0, C_0)$, the initial strong state class is (m_0, Q_0) with:

$$Q_0 = \left\{ \; 0 \;\; \leq \;\; \psi_i \;\; \leq \;\; 0 \;\;\; t_i \in \mathcal{E}(m_0) \right.$$

Firability condition

For a PTPN (N, R) and from a strong state class $R = (m, Q)$ with

$$Q = \left\{ \begin{array}{rcll} \alpha_i & \leq & \psi_i & t_i \in \mathcal{E}(m) \\ \psi_i & \leq & \beta_i & t_i \in \mathcal{E}(m) \\ \psi_i - \psi_j & \leq & \gamma_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

consistent, a set of transitions $r \in R$ is firable iff:

- All transitions of r are enabled at m, i.e. $t_f \in \mathcal{E}(m) \cup r$
- The system $Q_f = Q \cap F$ with

$$F = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s & \leq & \psi_f + \theta & t_f \in r \\ \psi_i + \theta & \leq & \beta_i^s & t_i \in \mathcal{E}(m) \end{array} \right.$$

  is consistent.

Theorem 11. *A set of transitions r of a PTPN (N, R) is firable from a strong state class $R = (m, Q)$ in closure form iff $\forall t_i \in \mathcal{E}(m), \forall t_f \in r, f \neq i, \alpha_f^s - \beta_i^s \leq \gamma_{fi}$*

Proof. Eliminating θ in F yields the system:

$$\begin{array}{rcll} 0 & \leq & \beta_i^s - \psi_i & t_i \in \mathcal{E}(m) \\ \alpha_f^s - \psi_f & \leq & \beta_i^s - \psi_i & i \neq f, t_f \in r, t_i \in \mathcal{E}(m) \end{array}$$

By definition $\psi_i \leq \beta_i^s$ is true in Q and further, $\psi_f - \psi_i \leq \gamma_{fi}$ exists in Q, so a necessary condition is that, for each $i \neq f$ there is $\alpha_f^s - \beta_i^s \leq \gamma_{fi}$. □

Successor class

If the set of transitions r is firable from (m, Q) then we have $(m, Q) \xrightarrow{t} (m', Q')$ with:

$$m' = m - \sum_{t \in r} \mathbf{Pre}(t) + \sum_{t \in r} \mathbf{Post}(t)$$

and Q' is obtained by:

1. Adding a new variable θ constrained by the previous firability condition;
2. For each transition t_i enabled at m', a new variable ψ_i' is introduced with constraints:

$$\begin{array}{ll} \psi_i' = \psi_i + \theta & \text{if } t_i \notin r, m - \sum_{t \in r} \mathbf{Pre}(t) \geq \mathbf{Pre}(t_i) \\ 0 \leq \psi_i' \leq 0 & \text{if } t_i \in \text{nnbl}(m, r) \end{array}$$

3. variables ψ and θ are eliminated.

Now we will detail the computation. Consider the clock domain Q represented by the following system, assumed consistent and in closure form:

$$Q = \left\{ \begin{array}{rcll} \alpha_i & \leq & \psi_i & t_i \in \mathcal{E}(m) \\ \psi_i & \leq & \beta_i & t_i \in \mathcal{E}(m) \\ \psi_i - \psi_j & \leq & \gamma_{ij} & t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

Adding the firability conditions:

$$Q' = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s & \leq & \psi_f + \theta & t_f \in r \\ \psi_i + \theta & \leq & \beta_i^s & t_i \in \mathcal{E}(m) \\ \alpha_i & \leq & \psi_i & t_i \in \mathcal{E}(m) \\ \psi_i & \leq & \beta_i & t_i \in \mathcal{E}(m) \\ \psi_i - \psi_j & \leq & \gamma_{ij} & t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

Consider a transition $t_f \in r$ and making ψ_f explicit in Q':

$$Q' = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_f & \\ \alpha_f & \leq & \psi_f & \\ \psi_i - \gamma_{if} & \leq & \psi_f & i \neq f, t_i \in \mathcal{E}(m) \\ \psi_f & \leq & \psi_j + \gamma_{fj} & j \neq f, t_j \in \mathcal{E}(m) \\ \psi_f & \leq & \beta_f & \\ \psi_f & \leq & \beta_f^s - \theta & \\ \theta & \leq & \beta_i^s - \psi_i & t_i \in \mathcal{E}(m) \\ \alpha_i & \leq & \psi_i & t_i \in \mathcal{E}(m) \\ \psi_i & \leq & \beta_i & t_i \in \mathcal{E}(m) \\ \psi_i - \psi_j & \leq & \gamma_{ij} & t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

We now eliminate ψ_f:

$$Q' = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} & j \neq f, t_j \in \mathcal{E}(m) \\ \alpha_f^s - \theta & \leq & \beta_f & \\ \alpha_f^s - \theta & \leq & \beta_f^s - \theta & (4) \\ \alpha_f & \leq & \psi_j + \gamma_{fj} & j \neq f, t_j \in \mathcal{E}(m) \quad (5) \\ \alpha_f & \leq & \beta_f & (6) \\ \alpha_f & \leq & \beta_f^s - \theta & \\ \psi_i - \gamma_{if} & \leq & \psi_j + \gamma_{fj} & j \neq i, t_j, t_i \in \mathcal{E}(m) \quad (8) \\ \psi_i - \gamma_{if} & \leq & \beta_f & i \neq f, t_i \in \mathcal{E}(m) \quad (9) \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta & i \neq f, t_i \in \mathcal{E}(m) \\ \theta & \leq & \beta_i^s - \psi_i & t_i \in \mathcal{E}(m) \\ \alpha_i & \leq & \psi_i & t_i \in \mathcal{E}(m) \quad (12) \\ \psi_i & \leq & \beta_i & t_i \in \mathcal{E}(m) \\ \psi_i - \psi_j & \leq & \gamma_{ij} & t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

(4) and (6) are true by hypothesis. Since Q is in closure form, (8) is redundant with (C6) $\gamma_{ij} \leq \gamma_{if} + \gamma_{fj}$, and (C3) $\alpha_f - \gamma_{fi} \leq \alpha_i$ makes (5) redundant with (12). There is also: $\beta_i \leq \gamma_i t + \beta_t$ which makes (9) redundant with (12). So finally we

have for all transitions $t_i, t_j \in \mathcal{E}(m) \setminus \{t_f\}$:

$$Q' = \begin{cases} 0 & \leq & \theta \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} \\ \alpha_f^s - \theta & \leq & \beta_f \\ \alpha_f & \leq & \beta_f^s - \theta \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta \\ \theta & \leq & \beta_i^s - \psi_i \\ \alpha_i & \leq & \psi_i \\ \psi_i & \leq & \beta_i \\ \psi_i - \psi_j & \leq & \gamma_{ij} \end{cases}$$

If we proceed in the same manner to eliminate all transitions in r, we obtain a set of inequalities for $t_i, t_j \in \mathcal{E}(m) \setminus r$ such:

$$Q' = \begin{cases} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} & t_f \in r \\ \alpha_f^s - \theta & \leq & \beta_f & t_f \in r \\ \alpha_f & \leq & \beta_f^s - \theta & t_f \in r \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta & t_f \in r \\ \theta & \leq & \beta_i^s - \psi_i & \\ \alpha_i & \leq & \psi_i & \\ \psi_i & \leq & \beta_i & \\ \psi_i - \psi_j & \leq & \gamma_{ij} & \end{cases}$$

Now consider a transition t_c in conflict with r and rewriting Q' to make the conflict variable ψ_c explicit:

$$Q' = \begin{cases} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} & t_f \in r \\ \alpha_f^s - \theta & \leq & \beta_f & t_f \in r \\ \alpha_f & \leq & \beta_f^s - \theta & t_f \in r \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta & t_f \in r \\ \theta & \leq & \beta_i^s - \psi_i & \\ \alpha_i & \leq & \psi_i & \\ \psi_i & \leq & \beta_i & \\ \psi_i - \psi_j & \leq & \gamma_{ij} & \\ \alpha_f^s - \theta - \gamma_{fc} & \leq & \psi_c & \\ \alpha_c & \leq & \psi_c & \\ \psi_i - \gamma_{ic} & \leq & \psi_c & \\ \psi_c & \leq & \beta_f^s - \theta + \gamma_{cf} & \\ \psi_c & \leq & \beta_c & \\ \psi_c & \leq & \gamma_{cj} + \psi_j & \\ \psi_c & \leq & \beta_c^s - \theta & \end{cases}$$

And by eliminating ψ_c we have:

$$Q' = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} & t_f \in r \quad (2) \\ \alpha_f^s - \theta & \leq & \beta_f & t_f \in r \quad (3) \\ \alpha_f & \leq & \beta_f^s - \theta & t_f \in r \quad (4) \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta & t_f \in r \quad (5) \\ \theta & \leq & \beta_i^s - \psi_i & \\ \alpha_i & \leq & \psi_i & \\ \psi_i & \leq & \beta_i & \\ \psi_i - \psi_j & \leq & \gamma_{ij} & \\ \alpha_f^s - \gamma_{fc} & \leq & \beta_f^s + \gamma_{cf} & (9) \\ \alpha_f^s - \theta - \gamma_{fc} & \leq & \beta_c & (10) \\ \alpha_f^s - \theta - \gamma_{fc} & \leq & \gamma_{cj} + \psi_j & (11) \\ \alpha_f^s - \gamma_{fc} & \leq & \beta_c^s & (12) \\ \alpha_c & \leq & \beta_f^s - \theta + \gamma_{cf} & (13) \\ \alpha_c & \leq & \beta_c & (14) \\ \alpha_c & \leq & \gamma_{cj} + \psi_j & (15) \\ \alpha_c & \leq & \beta_c^s - \theta & \\ \psi_i - \gamma_{ic} & \leq & \beta_f^s - \theta + \gamma_{cf} & (17) \\ \psi_i - \gamma_{ic} & \leq & \beta_c & (18) \\ \psi_i - \gamma_{ic} & \leq & \gamma_{cj} + \psi_j & (19) \\ \psi_i - \gamma_{ic} & \leq & \beta_c^s - \theta & \end{array} \right.$$

(9),(12) and (14) (involving constants only) must hold since the system is consistent. (15),(18) and (19) are redundant. (13) is redundant since $\alpha_c - \gamma_{cf} \leq \alpha_f$ via (3). (11) is redundant via (2). (10) is redundant via (3). (17) is redundant via (5).

So by iterating the same process for all transitions in conflict we have for transitions $t_i, t_j \in \mathcal{E}(m) \setminus (r \cup cnflt(m,r))$

$$Q' = \left\{ \begin{array}{rcll} 0 & \leq & \theta & \\ \alpha_f^s - \theta & \leq & \psi_j + \gamma_{fj} & t_f \in r \\ \alpha_f^s - \theta & \leq & \beta_f & t_f \in r \\ \alpha_f & \leq & \beta_f^s - \theta & t_f \in r \\ \psi_i - \gamma_{if} & \leq & \beta_f^s - \theta & t_f \in r \\ \theta & \leq & \beta_i^s - \psi_i & \\ \alpha_i & \leq & \psi_i & \\ \psi_i & \leq & \beta_i & \\ \psi_i - \psi_j & \leq & \gamma_{ij} & \\ \alpha_c & \leq & \beta_c^s - \theta & t_c \in cflt(m,r) \\ \psi_i - \gamma_{ic} & \leq & \beta_c^s - \theta & t_c \in cflt(m,r) \end{array} \right.$$

We now introduce new variables ψ'_i for all persistent transitions t_i with $\psi'_i =$

$\psi_i + \theta$, and so:

$$Q' = \begin{cases} 0 \leq \theta & \\ \alpha_f^s \leq \psi'_j + \gamma_{fj} & t_f \in r \\ \alpha_f^s - \theta \leq \beta_f & t_f \in r \\ \alpha_f \leq \beta_f^s - \theta & t_f \in r \\ \psi'_i - \gamma_{if} \leq \beta_f^s & t_f \in r \\ 0 \leq \beta_i^s - \psi'_i & \\ \alpha_i \leq \psi'_i - \theta & \\ \psi'_i - \theta \leq \beta_i & \\ \psi'_i - \psi'_j \leq \gamma_{ij} & \\ \alpha_c \leq \beta_c^s - \theta & t_c \in cflt(m,r) \\ \psi'_i - \gamma_{ic} \leq \beta_c^s & t_c \in cflt(m,r) \end{cases}$$

or

$$Q' = \begin{cases} \psi'_i \leq \beta_i^s & \\ \alpha_f^s \leq \psi'_j + \gamma_{fj} & t_f \in r \\ \psi'_i - \gamma_{if} \leq \beta_f^s & t_f \in r \\ \psi'_i - \psi'_j \leq \gamma_{ij} & \\ \psi'_i - \gamma_{ic} \leq \beta_c^s & t_c \in cflt(m,r) \\ 0 \leq \theta & \\ \alpha_f^s - \beta_f \leq \theta & t_f \in r \\ \psi'_i - \beta_i \leq \theta & \\ \theta \leq \beta_f^s - \alpha_f & t_f \in r \\ \theta \leq \psi'_i - \alpha_i & \\ \theta \leq \beta_c^s - \alpha_c & t_c \in cflt(m,r) \end{cases}$$

Elimination of θ produces

$$Q' = \begin{cases} \alpha_i \leq \psi'_i & & \\ \alpha_f^s - \gamma_{fj} \leq \psi'_j & t_f \in r & (2) \\ \alpha_f^s - \beta_f + \alpha_i \leq \psi'_i & t_f \in r & (3) \\ \psi'_i \leq \beta_f^s + \gamma_{if} & t_f \in r & (4) \\ \psi'_i \leq \beta_c^s + \gamma_{ic} & t_c \in cflt(m,r) & (5) \\ \psi'_i \leq \beta_i^s & & \\ \psi'_i \leq \beta_f^s - \alpha_f + \beta_i & t_f \in r & (7) \\ \psi'_i \leq \beta_c^s - \alpha_c + \beta_i & t_c \in cflt(m,r) & (8) \\ \psi'_i - \psi'_j \leq \gamma_{ij} & & (9) \\ \psi'_i - \psi'_j \leq \beta_i - \alpha_j & & (10) \\ \alpha_f \leq \beta_f^s & t_f \in r & (11) \\ \alpha_c \leq \beta_c^s & t_c \in cflt(m,r) & (12) \\ \alpha_f^s - \beta_f \leq \beta_f^s - \alpha_f & t_f \in r & (13) \\ \alpha_f^s - \beta_f \leq \beta_c^s - \alpha_c & t_c \in cflt(m,r), t_f \in r & (14) \end{cases}$$

(3) is redundant with (2) since $\gamma_{fi} \leq \beta_f - \alpha_i$. Similarly, (7) is redundant with

(4). (8) is redundant with (5) and (10) is redundant with (9). (11)-(14) (involving constants only) must hold since the system is assumed consistent.

What is left is then:

$$Q' = \left\{ \begin{array}{rcll} \alpha_i & \leq & \psi'_i & \\ \alpha^s_f - \gamma_{fj} & \leq & \psi'_j & t_f \in r \quad (2) \\ \psi'_i & \leq & \beta^s_f + \gamma_{if} & t_f \in r \quad (4) \\ \psi'_i & \leq & \beta^s_c + \gamma_{ic} & t_c \in cflt(m,r) \quad (5) \\ \psi'_i & \leq & \beta^s_i & \\ \psi'_i - \psi'_j & \leq & \gamma_{ij} & \quad (9) \end{array} \right.$$

The last step consists to add variables and constraints corresponding to the newly enabled transitions yielding system:

$$Q' = \left\{ \begin{array}{rcll} \alpha'_i & \leq & \psi'_i & t_i \in \mathcal{E}(m) \\ \psi'_i & \leq & \beta'_i & t_i \in \mathcal{E}(m) \\ \psi'_i - \psi'_j & \leq & \gamma'_{ij} & i \neq j, t_i, t_j \in \mathcal{E}(m) \end{array} \right.$$

with

$$\alpha'_i = \left\{ \begin{array}{ll} 0 & \text{if } t_i \in \text{nnbl}(m, t_f) \\ \max(\{\alpha_i\} \cup \{\alpha_f - \gamma_{fi} \mid t_f \in r\}) & \text{otherwise} \end{array} \right.$$

$$\beta'_i = \left\{ \begin{array}{ll} 0 & \text{if } t_i \in \text{nnbl}(m, t_f) \\ \min(\{\beta_i\} \cup \{\beta_k + \gamma_{ik} \mid t_k \in r \cup \text{cflt}(m,r)\}) & \text{otherwise} \end{array} \right.$$

$$\gamma'_{ij} = \left\{ \begin{array}{ll} \beta'_i - \alpha'_j & \text{if either } t_i \text{ or } t_j \text{ in nnbl}(m,r) \\ \min(\gamma_{ij}, \beta'_i - \alpha'_j) & \text{otherwise} \end{array} \right.$$

5.3.2 Example of a SSCG for a PTPN

For the purpose of this example, we will make a PTPN using the same TPN from the example 5.1b, twice. We call the original TPN N and its twin N'.

With set of transitions for m_0:

$$\{(t_0.1|t_0.2), (t_1.1|t_0.2), (t_0.1|t_1.2), (t_1.1|t_1.2)\}$$

Here, the SSCG obtained from the PTPN is the same as for a SCG, because on such small example, the added clocks informations is not significant.

This SSCG is decomposed as follow for state $\{0\}$:

- $m_0 = \{p_0, p_0\}$
- $D = \left\{ \begin{array}{rcl} 0 & \leq x_{t0.1|t0._2} \leq & 4 \\ 1 & \leq x_{t1.1|t1._2} \leq & 5 \\ \dots & & \end{array} \right.$

The idea is still to have a clock for each possible pair of transitions firing from the original state. This ends ups with a big clock domain Q.

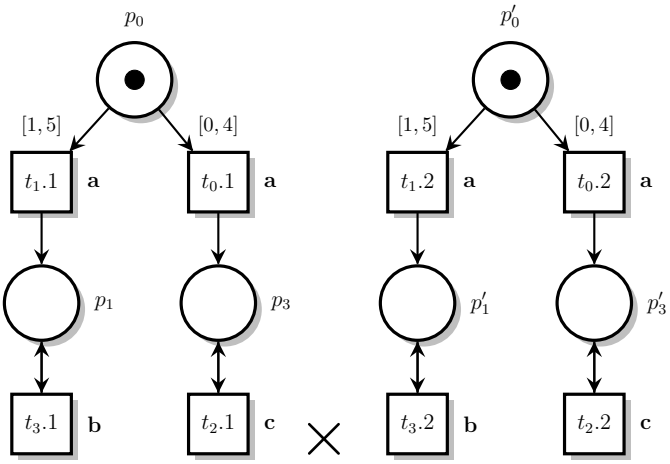


Figure 5.4: PTPN of $N \times N'$

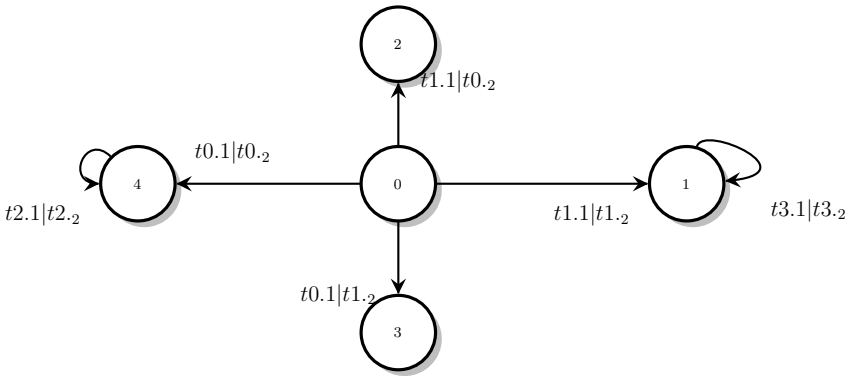


Figure 5.5: SSCG of $N \times N'$

Now that we have seen the SCG and the SSCG, adapted to PTPN, we summarize this chapter before going to the next one.

5.4 Summary

In this Chapter 5 we presented the SCG and SSCG, two constructions used in the analysis of TPN and PTPN behaviour.

- *SCG* are composed of states and a domain (representing the timing behaviour of the TPN or PTPN). They are easily used to analyse the behaviour of the modelled TPN or PTPN and we will mainly use them trough this thesis.
- *SSCG* are composed of states and a clock domain (representing the timing behaviour of the TPN or PTPN). They are bigger than SCG and we use a particular kind of SSCG, with timing clearly displayed as i events.
- SCG and SSCG are easily adapted to *PTPN* by adding the information about the *pair* of transitions firing synchronously and taking into account the more restrictive timing constraints.

The Chapter 6 will be focused on the property of *Diagnosability*, the core property we studied during this thesis.

Chapter 6

Diagnosability

After presenting our new models, the PTPN, its computation and the SCG, we aim to explain an algorithm regarding the diagnosability in TPN models. This Chapter will be decomposed in four sections. First, we introduce the basis of our algorithm and we make a quick summary of the needed tools for our models. Secondly, we present the twin plant method, quickly adapted to our PTPN, and how to use it to check diagnosability. We then proceed to our two analysis, first an algorithm to check the diagnosability of a single fault, then, a method to check the diagnosability of a *pattern* (a chain of labels). Finally, we conclude this chapter via a quick summary.

6.1 Problematic

In this chapter, we aim to use our new model, the PTPN, to process an ad-hoc synchronous product and to check a property (diagnosability) on a TPN. Diagnosability (or *diagnosability analysis*) is the ability to detect and locate any fault within a finite delay after its occurrence.

Diagnosability of faults in Petri nets can be decomposed in two sets of techniques: graph-based techniques (diagnoser, verifier/twin-plant) or via the solution of optimization problems (Integer Linear Programming or ILP) (see [Basile 2018] for example). Graph-based techniques are based on the analysis of the net reachability or coverability graph (with an LTL checker for example). The second approach tackles the mathematical representation of the system itself to specify and solve optimization problems (usually expressed as ILP). Our approach is based on a graph-based method, but we also use optimization methods in the Chapter 8 of this thesis.

A notion commonly used is the notion of *critical pair* [Jiang 2001a]. A critical pair is a trace where one copy has a fault and not the other; and a system is diagnosable if it has no critical pairs. To check the diagnosability of a DES, a product composed of the DES with a *faultless* copy (i.e. a copy where transition that model the fault is removed) is made and a search for critical pairs is realized.

We want to check those critical pairs in the context of our SCG (from a PTPN). The problematic of the first section of this Chapter is then: *Can we check for critical pairs in PTPN-SCG?*

We then want to extend the notion of diagnosability to the diagnosability of a pattern, also via a PTPN-SCG. The second problematic of this chapter is then: *Can we check for diagnosability of a pattern in a PTPN-SCG?*

For the first problematic we create an ad-hoc algorithm, aiming at discovering the critical pairs in our processed SCG, optimizing the exploration of the SCG for a single fault diagnosability analysis. The second problematic is treated as a more general problem where a single fault can be expressed as a pattern (which leads to less optimized results than our ad-hoc algorithm).

We first present the twin-plant method, an algorithm aiming to discover the critical pairs in a Petri Net.

6.2 Verifier method

One of the first algorithms we were inspired from was the Verifier Methods [Yoo 2002] to check the diagnosability of a single fault event. The idea is to compare a TPN with a copy of itself without the faulty transitions. To make this comparison, a PTPN is constructed between these two TPN and a check is realized to find a cycle after a faulty transition (hence comparing faulty behaviour and faultless behaviour).

Let's take a quick example with the following Twin-TPN in figure 6.1. The original system N and its faultless copy N' is shown.

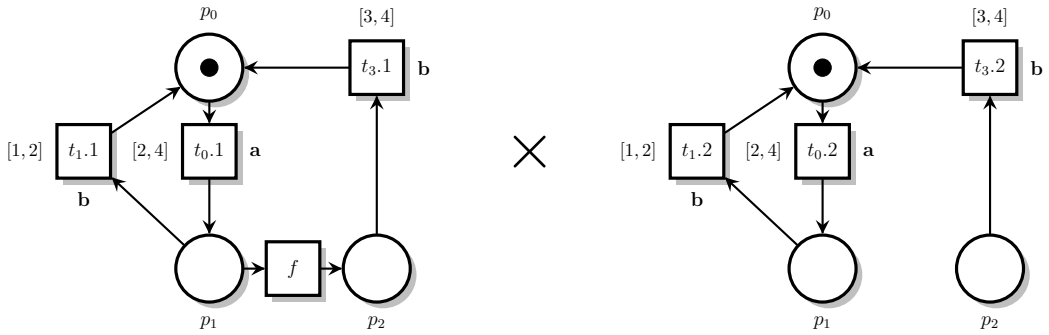


Figure 6.1: Composition of N and its faultless twin N'

This system is diagnosable because of the difference in timing constraints between t_1 and t_3 which creates a *timelock*. The SCG of the product of N and N' (the two TPN are considered as two PTPN) is given in Figure 6.2. In this SCG, after the faulty transition, the system cannot fire others transitions.

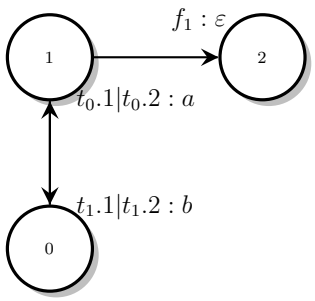


Figure 6.2: Example of a SCG.

To detect a cycle after a faulty transition, the SCG of the PTPN is explored. In our case of *diagnosability analysis*, the aim is to detect the *cycle of functioning after a fault*. To do that, a Tarjan's [Tarjan 1972] like *depth-first search* (DFS) algorithm is designed. A DFS algorithm is an algorithm made to explore tree or graph data structures. The concept is to explore as far as possible along each branch before backtracking. *Tarjan's strongly connected components* algorithm is an algorithm in graph theory to find the strongly connected components (SCCs) of a directed graph. *Strongly connected components*, in our case, are states which are connected in each direction through their transitions (hence the existence of a cycle between them).

If we take the example in Fig 6.2, two SCC: $SCC_1 = \{0, 1\}$ and $SCC_2 = \{2\}$ are represented. A DFS algorithm would go from the original state 0 and explore the path $t0.1|t0.2$ and $t1.1|t1.2$ as a cycle (a SCC) before backtracking to explore $f.1$ and conclude that 2 is an alone state in its own SCC. The idea, to summarize, is to detect SCC after a faulty event, to conclude on the diagnosability of our system.

We first explain the diagnosability analysis for a single fault event in the following Section 6.3.

6.3 Single Fault

In the following, to simplify the notations and when it is not ambiguous, we will denote a PTPN with the simple TPN without relations. We also assume that all transitions with a label are observable so $L = \Sigma$.

The *twin-plant* construction of a net N can easily be defined as the composition of two copies of N, say $N.1 \times_L N.2$. In the following, failures are considered as transitions on a common unobservable label, denoted f. The single fault f is diagnosable when a (critical) pair of executions cannot be found such that: (1) they have the same L-observations; and (2) only one of them eventually exhibits a failure (contains a transition labelled with f).

Just like in figure 6.2, a deadlock occurs after the faulty event, showing the difference between faulty and faultless behaviour and concluding on the diagnosability of the TPN N in figure 6.1.

The general assumption that systems are *ultimately observable* is used; meaning

that they do not block and that, on every execution, an observable event is always eventually found after a bounded number of transitions and within a bounded delay (which entails the absence of Zeno traces, like in [Tripakis 2002]). Hence the fault f is diagnosable when all the executions with a faulty transition are blocked. This means these executions cannot progress due to a observation difference between faulty behaviour and faultless behaviour.

6.3.1 Algorithm

Using the features of the PTPN modelling, we have developed an algorithm to check the diagnosability of a TPN (see Algo. 1). This Algorithm is based on the Twin Machine algorithm and its purpose is to find a cycle after (or with) a fault transition. If a fault transition leads to a cycle, it means that the TPN is not diagnosable. The algorithm is based on the Tarjan's Algorithm [Tarjan 1972] to find SCC via a DFS method.

The algorithm uses as input a PTPN-SCG (see Chapter 5) created from the PTPN of the product between N and N', its copy without the fault. Each state of the SCG is associated with an integer value. By default this value is 0 and means that the state is not in a cycle or a deadlock. If the value of a state is 1 then the state is in a cycle and if the value is -1 the state is a deadlock. Remark that the function $isCycle(m)$ (line 33) returns *True* if the value associated with the state m is 1 and *False* otherwise. The function $isTransitionFaulty(m, m')$ (line 23) returns *True* if the transition from m to m' is labelled as a fault.

The main data structure used in Algo. 1 is a stack called *stack*. It is a memory stack where the states explored are stocked. States can be popped (with the $MultiPopAndMark$ function) or pushed on it (with *push* function). The function $MultiPopAndMark(m, val)$ pops the state m and all states in the stack below and marks each popped state with the value val. However, since this function is also popping the stack depending on the cycle, it should also detect when a faulty transition is found and put the global variable $aFaultIsOccured$ to *False*.

Using the stack, cycles are found. The idea is to explore all states and their successors to detect the cycles. The function stops when it concludes on the diagnosability and returns 1 if it is diagnosable and -1 otherwise (line 32). The algorithm is followed by a quick example.

Example: For our following example we took again our TPN from 6.1 which is diagnosable. In this particular case, the fault in the TPN N is diagnosable because of the timing constraint in t_3 and t_1 (the transitions in the faultless copy N'). They do not have a *common solution*, hence the deadlock because of the timing constraints.

If this example is modified just a little bit to end up with figure 6.3, different results are obtained. In this example the $t_3.1$ transition has different timing constraint with $[2, 3]$ as its static interval.

Algorithm 1 Find cycle after a fault

```
1: //Input is the SCG from PTPN N after the intersection with N' the faultless copy
Require: SCG scg
2: stack := empty stack
3: aFaultIsOccured := False
4: notDiag := False
5: cycleAlgorithm(scg.origin)
6:
7: Function rec cycleAlgorithm(m)
8: stack.push(m)
9: if isDeadlock(m) then
10:   stack.MultiPopAndMark(m, −1) // Mark the deadlock
11: end if
12: while notDiag == False and stack.isEmpty == False do
13:   if m.next is in the stack then
14:     if aFaultIsOccured then
15:       notDiag := True // There is a cycle.
16:     end if
17:     // If there is no fault, we still have a cycle.
18:     stack.MultiPopAndMark(m.next, 1)
19:   else
20:     if aFaultIsOccured and isCycle(m.next) then
21:       // If the fault has occurred and the next state leads to a cycle, it is not diagnosable.
22:       notDiag := True
23:     else if isTransitionFaulty(m, m.next) then
24:       // Indicate the occurrence of a fault
25:       aFaultIsOccured := True
26:     end if
27:     m' = nextUnexploredState(m)
28:     cycleAlgorithm(m')
29:   end if
30: end while
31: if notDiag then
32:   return(−1) // The fault is diagnosable.
33: else if stack.isEmpty() then
34:   // If we have check each state the system is diagnosable.
35:   return(1)
36: end if
37: EndFunction
```

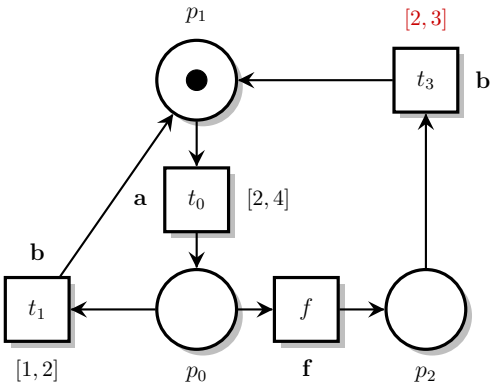


Figure 6.3: Example of a TPN N_{ex2}

If the SCG of $N_{ex2} \times_L N'_{ex2}$ is processed, the SCG in figure 6.4 is obtained.

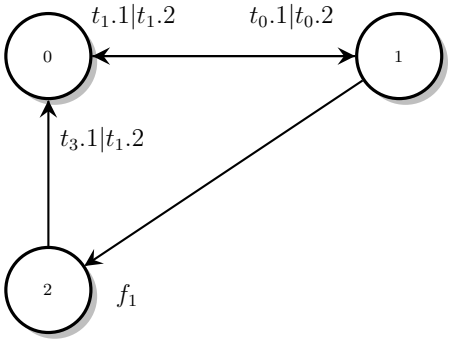


Figure 6.4: SCG of $N_{ex2} \times_L N'_{ex2}$

In this example you can quickly conclude that it is possible to have a *common solution* in the timing constraint of $t_3.1|t_1.2$ and continue to run the TPN after the faulty event (since the fault leads to the state $\{2\}$ which is not a deadlock any more).

If this SCG is run through our algorithm, the result show in Table 6.1 is obtained. Beginning at state $\{0\}$ and by following the transitions, a cycle between $\{0\}$ and $\{1\}$ is quickly found. However, since all of $\{1\}$ next states are not explored, the algorithm goes back to $\{1\}$ and the faulty transition f_1 is followed. After exploring $\{2\}$, the existence of a single SCC $\{0, 1, 2\}$ is observed, which contains a faulty transition concluding on the undiagnosability of the faulty event f_1.

Stack	$\{0\}$	$\{0,1\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,2\}$
Followed Transitions	$t_0.1\|t_0.2$	$t_1.1\|t_1.2$	f_1	$t_3.1\|t_1.2$	

Table 6.1: Processing of the Stack for SCC detection

This example focuses on the idea that timing is an important information to conclude on the diagnosability of events in a TPN. More examples will be available on Chapter 7 of this thesis, focusing on the scalability of our method.

Now that we have presented the diagnosability analysis for a single fault, we will present our extensions of this algorithm to the case of *patterns*.

6.4 Patterns

Our method can be extended to check the diagnosability of *patterns of unobservable events* [Jéron 2006] (a chain of labels). The idea here is to process the diagnosability of a much more complex behaviour represented as a chain of labelled transitions (usually unobservable) by using our PTPN model. We want to rely on our ad-hoc synchronous behaviour to synchronize and detect patterns in TPN systems.

In our case, a *pattern* M is a special instance of TPN. The set of labels occurring in M is denoted F and a place in M, said *found*, is distinguished as a witness for detection. For instance, the pattern in Fig. 6.5 detects executions that have three consecutive occurrences of b without any f in-between.

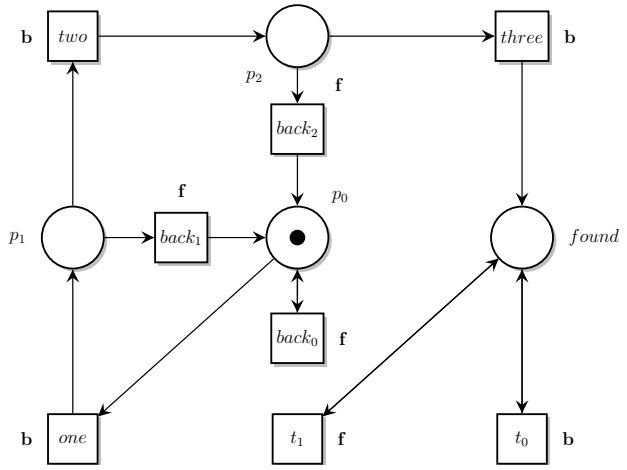


Figure 6.5: pattern for "three consecutive b without f"

A pattern M *detects* the execution σ if $\text{obs}_F(\sigma)$, the F-observation of σ, is an execution of M that "marks" the place *found*: the execution $\text{obs}_F(\sigma)$ can be run in $[\![M]\!]$ and it satisfies the linear property $\Diamond found$. It means that the pattern behaviour can be executed in the system **and** that the *found* place is detected.

More generally, we say that pattern M is *detected* in N when we reach a state where place *found* is marked in $N \times_F M$. This can be inferred from the marking graph of the pattern (see Fig. 6.6) or from the product $N \times_F M$.

Instead of defining a pattern as a regular language [Jéron 2006], or as a set of timed sequences [Gougam 2017, Pencolé 2018], we use a (prefix-closed) set of executions in $[\![M]\!]$. Hence we restrict ourselves to *time regular* languages, meaning sets of executions that can be "realized" with a TPN. This is enough to model every regular set of (untimed) traces in a TPN model. The process of a *timed pattern* is explored in Chapter 8.

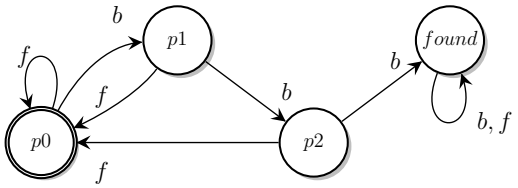


Figure 6.6: marking graph for the pattern in Fig. 6.5

6.4.1 Composition of a Pattern

We also want to make sure that a pattern does not interfere with the system it interacts with. For example, it should not prevent some executions of the system. To this end, three well-formedness conditions on patterns are imposed.

Definition 34 (Well-formed Pattern)**.** *A well-formed pattern is defined as followed:*

1. *Patterns are* total*: they should always allow transitions on the labels in F, at any time (they never block or delay a transition).*
2. *Patterns are* deterministic*: the same observations should lead to the same states.*
3. *Labels in F are unobservable: $F \cap L = \emptyset$.*

Constraints (1) and (2) can be expressed as a property over all states in $[\![M]\!]$, namely $\forall s \in [\![M]\!], a \in F, \theta \in \mathbb{Q}_{\geq 0}. \nexists t \in T, s', s'' \in [\![M]\!], (\mathcal{L}(t) = a \wedge s \xrightarrow{\theta} s' \xrightarrow{t} s'')$.

By analogy with our previous definition of diagnosability, pattern M is diagnosable if it is not possible to find a (critical) pair of executions such that M is detected in one but never in the other. This question can be reduced to a model-checking problem on a twin-plant; this time on the product of the system with the pattern. Generally, N_1 and N_2 are used to denote a system and its copy and M_1 and M_2 for the pattern and its copy.

Theorem 12. *Given a well-formed pattern M, with labels F, the net N, with label L, is diagnosable for pattern M if and only if all the maximal executions of the product $(N_1 \times_F M_1) \times_L (N_2 \times_F M_2)$ satisfy $(\Diamond found.1) \Rightarrow \Diamond (found.2 \vee dead)$.*

Proof. Since M is total (constraint 1 in Definition 34), the pattern is detected for an execution σ of N iff (Th. 4) there is an (equivalent) execution σ_M in $[\![N]\!] \|_F [\![M]\!]$ and the property $\Diamond found$ is valid for σ_M.

Moreover, since M is deterministic (constraint 2 in Definition 34), σ_M is unique, so it is not possible to find another execution in σ_M in $[\![N]\!] \|_F [\![M]\!]$, compatible with σ, where *found* is not marked.

Hence (Th. 5) the diagnosability of a pattern M is equivalent to check the diagnosability of the event *found* in the PTPN $N \times_F M$.

By Th. 4 and $F \cap L = \emptyset$ (constraint 3 in Definition 34), a critical pair in $(N \times_F M)$ corresponds to an execution in the state graph of $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$. In the following, we denote the atomic proposition *found.i* the faults stemming from M_i. Let us now consider an execution σ in $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$ and distinguish the cases where σ is a blocked or infinite execution:

(1) Since the system is ultimately observable and by Th. 4, an execution σ in $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$ is blocked if and only if all infinite executions $\sigma_1 = \sigma'_1 \sigma''_1$ in $\llbracket N_1 \times_F M_1 \rrbracket$ and $\sigma_2 = \sigma'_2 \sigma''_2$ in $\llbracket N_2 \times_F M_2 \rrbracket$ with $obs_L(\sigma'_1) \equiv obs_L(\sigma'_2) \equiv obs_L(\sigma)$ have diffente L-observations for σ''_1 and σ''_2, i.e. $obs_L(\sigma''_1) \not\equiv obs_L(\sigma''_2)$. It means that the property $(\Diamond found.1 \Rightarrow \Diamond dead) \wedge (\Diamond found.2 \Rightarrow \Diamond dead)$ is valid for σ iff $obs_L(\sigma_1) \not\equiv obs_L(\sigma_2)$.

(2) For an infinite execution σ in $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$, they are two executions, (Th. 4) σ_1 in $\llbracket N_1 \times_F M_1 \rrbracket$ and σ_2 in $\llbracket N_2 \times_F M_2 \rrbracket$, such $obs_L(\sigma) \equiv obs_L(\sigma_1) \equiv obs_L(\sigma_2)$. If the LTL formula $(\Diamond found.1) \Rightarrow (\Diamond found.2)$ is not valid for σ, then it exists two executions with the same L-observables σ_1 in $\llbracket N_1 \times_F M_1 \rrbracket$ and σ_2 in $\llbracket N_2 \times_F M_2 \rrbracket$ such the pattern is detected in σ_1 and not in σ_2 (i.e. the system is not diagnosable). And reciprocally, if we have two executions σ_1 and σ_2 with $obs_L(\sigma_1) \equiv obs_L(\sigma_2)$ and where the pattern is detected in σ_1 and not in σ_2, then it exits an execution σ in $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$ where $(\Diamond found.1) \Rightarrow (\Diamond found.2)$ is not valid (reasoning is the same if we consider $(\Diamond found.2) \Rightarrow (\Diamond found.1)$).

From the previous cases (1) and (2) we can conclude that if properties $(\Diamond found.1) \Rightarrow (\Diamond found.2 \vee dead)$ and $(\Diamond found.2) \Rightarrow (\Diamond found.1 \vee dead)$ are valid for all maximal executions in $(\llbracket N_1 \times_F M_1 \rrbracket) \|_L (\llbracket N_2 \times_F M_2 \rrbracket)$ then it not exists two executions in $\llbracket N_1 \times_F M_1 \rrbracket$ and in $\llbracket N_2 \times_F M_2 \rrbracket$ with the same L-observables and where only one detected the pattern. So, by definition, the system is diagnosable iff the properties are valid.

These properties can be optimized by taking into account the inherent symmetry of the problem (If one of the properties is valid, the second is also valid) and so we have only to verify that $(\Diamond found.1) \Rightarrow \Diamond(found.2 \vee dead)$. □

Some of the well-formedness constraints can be relaxed in the proof of Theorem 12. For instance, "deterministic" can be replaced with the weaker property: "detection is unambiguous". This means that it is not possible to find an execution in M that leads to two markings, one where *found* is marked and the other is not. Nonetheless, this presentation has some merits. For instance each condition can be checked automatically on the marking graph of M when the net is bounded and has no timing constraints.

But a question which remains is, *How do the product is processed?*.

6.4.2 Synchronization with the Pattern

The idea is to first process the product of the TPN N and its pattern M.

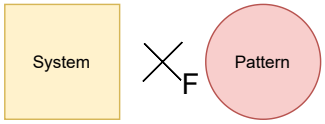


Figure 6.7: First step of our Pattern-Product

When the product described in figure 6.7 is done (with a PTPN synchronizing the two TPN as $N \times_F M$), the occurrence of the *found* place has to be checked. For this, the diagnosability of the transitions just before *found* is tested. To do this, another ad-hoc twin-plant method for a single fault is applied. So, a copy of $N \times_F M$ is processed and synchronized with itself (with $\times_L$) just like in figure 6.8.

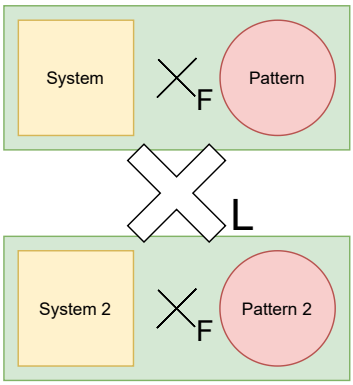


Figure 6.8: Second step of our Pattern-Product

Contrary to the methods in Section 6.3, the faulty transitions in the copy are not erased and the system is checked via an LTL formula.

This method, however, creates a much bigger system than the previous one (Section 6.3) because of the synchronized pattern added to the system N.

6.4.3 Single Fault Pattern

A pattern can also be defined as a *single fault pattern*, which means that the pattern is only composed of a single transition representing the faulty event. This method is efficient for a pattern, but in the case of a single fault, it will create a much bigger SCG at the end of the process than our first twin-plant method in Section 6.3, because the fault is erased in the previous method and place *found* is added.

Therefore, to check if N is diagnosable for a single fault with the pattern method, the SCG can be generated from $N.1 \times_L N.2$ then a LTL model-checker can be used to check property $(\Diamond f.1) \Rightarrow \Diamond (f.2 \vee \mathit{dead})$.

6.4.4 Algorithm

The diagnosability of a net, N, in relation to an F-pattern, M, can be checked by first generating the SCG from a script processing the SCG from subsection 6.4.2 ; and then use a LTL model-checker with the property of Th. 12 :

$$(\Diamond found.1) \Rightarrow \Diamond\,(found.2 \vee dead)$$

The idea is to detect the occurrence of a $found$ marking or a deadlock, indicating the diagnosability of the detected pattern. The model-checker *selt* is used to check on the SCG if the pattern is diagnosable. For an example of this method, which creates a larger system, you may refer to the Chapter 7 for our benchmark about the diagnosability of patterns.

Example: Our system remains the TPN N in Figure 6.3 and the pattern M is the single fault pattern shown in Figure 6.9.

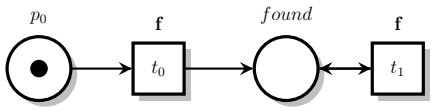


Figure 6.9: Pattern M

The software Twina (see Section 7.1) is used to process the PTPN from N and M. Twina can also give a counter-example if the system is not diagnosable. The following result are obtained for a single fault diagnosability analysis on N.

```
# net {}, 16 places, 15 transitions, 74 arcs                              #
# bounded, not live, possibly reversible                                  #
# abstraction        count      props      psets       dead       live #
#      states           14         16         13          0          6 #
# transitions           26         15         15          6          7 #
FALSE
  state 0:
  L.scc*3 {p0.1.1} {p0.1.2} {p1.2.1} {p1.2.2} {ptest_t0.2.1}
   {ptest_t0.2.2} {ptest_t1.2.1} {ptest_t1.2.2} {ptest_t3.2.1} {ptest_t3.2.2}
  -{t0.2.1|t0.2.2} ... (preserving T)->
* [accepting] state 11:
 L.scc {found.1.1} {p0.1.2} {p0.2.2} {p2.2.1} {ptest_t0.2.1}
 {ptest_t0.2.2} {ptest_t1.2.1} {ptest_t1.2.2} {ptest_t3.2.1} {ptest_t3.2.2}
  -{t3.2.1|t1.2.2} ... (preserving - dead /\ - {found.1.2})->
  state 11:
  L.scc {found.1.1} {p0.1.2} {p0.2.2} {p2.2.1} {ptest_t0.2.1}
   {ptest_t0.2.2} {ptest_t1.2.1} {ptest_t1.2.2} {ptest_t3.2.1} {ptest_t3.2.2}
```

The counter-example can be processed in a trace which can be input on the Tina tool.

6.5 Summary

In this Chapter 6 we present a diagnosability analysis for a single fault and a pattern based on PTPN.

- *Single Fault* is detected with the help of the *twin-plant* method, adapted to a PTPN. The idea is to detect if there is a difference in behaviour between a faulty system and a faultless system.
- *Patterns* are more complex behaviours to detect. They are represented as a TPN. They are synchronized to the original system to detect the occurrence of the pattern in the system. This produces bigger systems than the previous methods but more complex behaviour can be analysed than a single fault.

The Chapter 7 will be focused on our experimental result, compared with the *IPTPN* methods presented in Section 3.3.2.2.

Chapter 7

Experimental Results

In this Chapter, we focus on the different tests and benchmarks that we used to evaluate the feasibility of our methods. First, we mainly compare the PTPN methods to the IPTPN methods (see subsection 3.3.2.2 for more information). We also want to test the scalability of our method and we focus on more complex benchmarks on the second section of this chapter.

Finally we present a benchmark for the diagnosability of patterns by using a well-known example from a previous paper by Gougam and al. [Gougam 2017].

Before going into our results (mainly used to prove the scalability of our methods), we quickly present our software, TWINA, which is now part of the TINA toolbox [Berthomieu 2004].

7.1 TWINA

TWINA [Dal Zilio 2019] is a tool for analyzing the *product* of two Time Petri Nets (PTPN), with possibly inhibitor and read arcs [Peres 2011]. Its main objective is to compute a usable representation of the intersection of two net languages; meaning the intersection of the (timed) languages obtained from the executions of two TPNs, in which transitions with the same labels are fired at the same time.

The tool is based on a new extension of the State Class Graph construction, the method used in the TINA (TIme petri Net Analyzer) toolbox. Like for TINA, this tool is maintained by the Verification of Time Critical Systems (VERTICS) group at LAAS-CNRS, which develops new verification methods and tools for checking properties of critical systems having strong temporal and timing requirements.

TWINA is made to compute PTPNs in different forms (SCG or SSCG for example) and data computable in the rest of the TINA tool-chain (selt for example). Let's quickly go through some of its options:

```
%twina -h
twina -W system.net
twina -F system.net
twina -W -v system.net
```

- *-W* process the SCG of the system in input by preserving markings and maximal executions.
- *-F* process the SSCG of the system in input with time delay (in i transition).

- *-W -v* process the LSCG of the system in input and output all the marking and domain of each class of the system.

For more information about this software, you may refer to its website (https://projects.laas.fr/twina/). Let's now take our first example.

7.2 Single Fault analysis

We now focus on our single fault diagnosability analysis. We first use our *twin-plant* algorithm to conclude on the diagnosability of our system. For this, we use an option in the tool *TWINA* with the following command:

```
twina -f --fault f system.net
```

This command simply processes a twin-plant with a fault f and concludes with our algorithm on the diagnosability of our system. In our experiments, we want to compare the results obtained with PTPN (using TWINA) and an encoding into IPTPN [Peres 2011]. By default, TWINA uses option -W, that computes the Linear SCG of a net. We also provide an option -I to compute the LSCG for the product of two nets.

Let's take a more direct example.

7.2.1 Example of single fault analysis

We take again the example from Chapter 2 in figure 2.4. We process a twin-plant construct and check the diagnosability of our system with the first algorithm (see Algorithm 1).

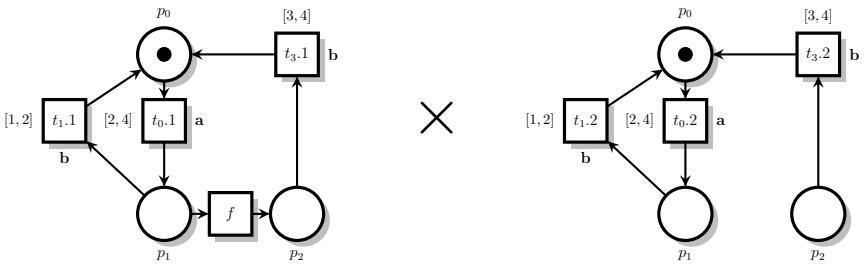


Figure 7.1: Composition of N and its faultless twin N'

We compare our method with the software *sift* (construction and checking of reachability graph), another software in the Tina toolbox and the SCTA methods (see subsection 3.3.2.1) to process TA from TPN [Lime 2003a] (and then model-check the resulting twin-plant TA). We process the SCG used in sift with an IPTPN [Peres 2011] and a classical TPN. Since the IPTPN models process a SSCG (and not a SCG) and sift was not made specifically for diagnosability analysis, we end up with slightly better results.

	Twina	IPTPN/sift	TPN/sift
Places	6	6	25
Trans.	7	12 $(2 + 5 * 2)$	211
Classes	3	3	1389

Figure 7.2: Results of the analysis of a single fault

In figure 7.2, it can be clearly seen that the Twina method is the most efficient. Since IPTPN must create transitions to process the decomposition of timing constraints from their labelled transitions, it is naturally bigger in terms of transitions. Another way to analyse the diagnosability is to process a SSCG from a TA computed from a TPN, which was our idea for the *TPN/sift* solution. As you can see, it creates the biggest SCG possible in terms of places, so this method is certainly not efficient in terms of scalability.

We also compared our methods via bigger models, found in the diagnosis worlds, in the next Subsection.

7.2.2 Scalability example

We use several models, some adapted to a timed context, to check the feasibility and the scalability of our method. If you would like to reproduce our benchmark, you can refer to our manual on the TWINA webpage (https://projects.laas.fr/twina/) in the reproducibility section. In the following, we compare the size of the LSCG obtained on different models with the results obtained using IPTPN and Tina. The results are reported in Section 7.3 below, with the sizes of the SCG in number of classes and edges; we also give the ratio of classes saved between the SCG and the SSCG (where a ratio of 2 means that we have twice as much classes in the SSCG than in the LSCG). We use the *sift* tool to compute the SSCG from an IPTPN; sift is an optimized version of TINA that provides fewer options (state class abstractions) but that is much faster.

We use different models for our benchmarks (in each case we state the name of the fault option used in the twin product construction):

- *plant* is the model of a complex automated manufacturing system from [Wang 2015] (-fault=F);
- *jdeds* is an example taken from [Gougam 2017] extended with time (-fault=f);
- *train* is a modified version of the train controller example in the Tina distribution with an additional transition that corresponds to a fault in the gate (we have examples with 3 and 4 trains) (-fault=F1);
- *wodes* is the WODES diagnosis benchmark of Giua (found for instance in [Gougam 2017]) with added timing constraints (-fault=F1).

For each model, we give the result of three experiments: *plain* where we compute the SCG of the net, alone; *twin* where we compute the intersection between the TPN and a copy of itself with some transitions removed; and *obs* where we compute the intersection of the net with a copy of the observer.

Model	Exp.	Twina Classes	IPTPN/sift Classes	Ratio
jdeds	plain	26	28	$\times 1.1$
jdeds	twin	544	706	$\times 1.3$
jdeds	obs	57	64	$\times 1.1$
train3	plain	$3.10 \cdot 10^3$	$5.05 \cdot 10^3$	$\times 1.6$
train3	twin	$1.45 \cdot 10^6$	$4.02 \cdot 10^6$	$\times 2.8$
train3	obs	$6.20 \cdot 10^3$	$1.01 \cdot 10^4$	$\times 1.6$
train4	plain	$1.03 \cdot 10^4$	$1.68 \cdot 10^4$	$\times 1.6$
train4	twin	$2.10 \cdot 10^7$	$5.76 \cdot 10^7$	$\times 2.7$
train4	obs	$2.06 \cdot 10^4$	$3.37 \cdot 10^4$	$\times 1.6$
plant	plain	$2.70 \cdot 10^6$	$4.63 \cdot 10^6$	$\times 1.7$
plant	twin	$1.30 \cdot 10^3$	$1.63 \cdot 10^3$	$\times 1.3$
plant	obs	$5.72 \cdot 10^6$	$9.79 \cdot 10^6$	$\times 1.7$
wodes	plain	$2.55 \cdot 10^3$	$5.36 \cdot 10^3$	$\times 2.1$
wodes	twin	$5.54 \cdot 10^4$	$1.51 \cdot 10^5$	$\times 2.7$
wodes	obs	$5.77 \cdot 10^3$	$1.47 \cdot 10^4$	$\times 2.5$
wodes232	plain	$2.04 \cdot 10^4$	$3.24 \cdot 10^4$	$\times 1.6$
wodes232	twin	$3.96 \cdot 10^7$	$3.39 \cdot 10^8$	$\times 8.6$
wodes232	obs	$1.06 \cdot 10^5$	$2.26 \cdot 10^5$	$\times 2.1$

Figure 7.3: Results of the two methods

As expected, *TWINA* is generally more efficient because it relies on the SCG model, smaller compared to the SSCG models from an IPTPN. For smaller systems, such as *jdeds*, the IPTPN method still remains efficient, however, for bigger system such as *wodes232*, the IPTPN method produces almost ten times the classes of the PTPN method. You can also see that Figure 7.3 only focus on classes and not on transitions. In an IPTPN method, the number of transitions is generally doubled in timed systems since every timed transition is decomposed into two transitions.

We now focus on our example for *pattern diagnosability analysis*.

7.3 Pattern analysis

We applied our methods to the following example from [Gougam 2017]. The system is defined in figure 7.4. It is the modelling of a transport system, timed. You can refer to the TWINA webpage on the benchmark patterns if more information about the pattern is wanted.

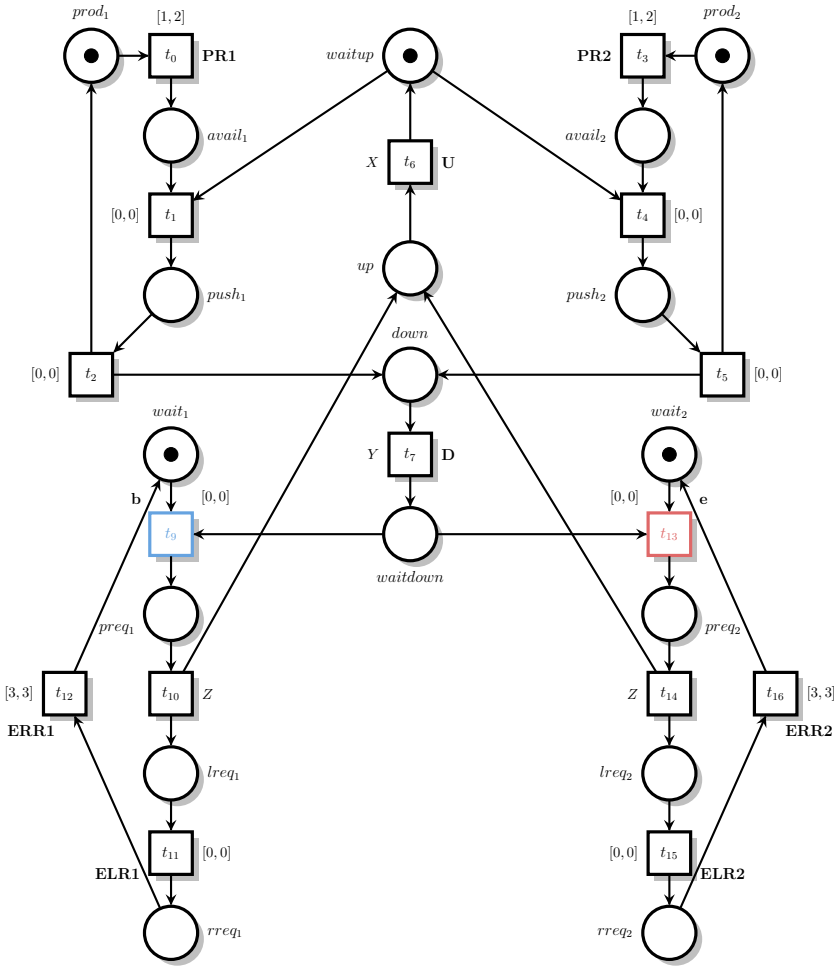


Figure 7.4: Transport timed - [Gougam 17]

We modified slightly this model to give it a b and a e transitions and created the following *3b without e* pattern (figure 7.5). Every time an e label occurs, we go back to the original place of the pattern. We still want to detect the $found$ place just like in Section 6.4.

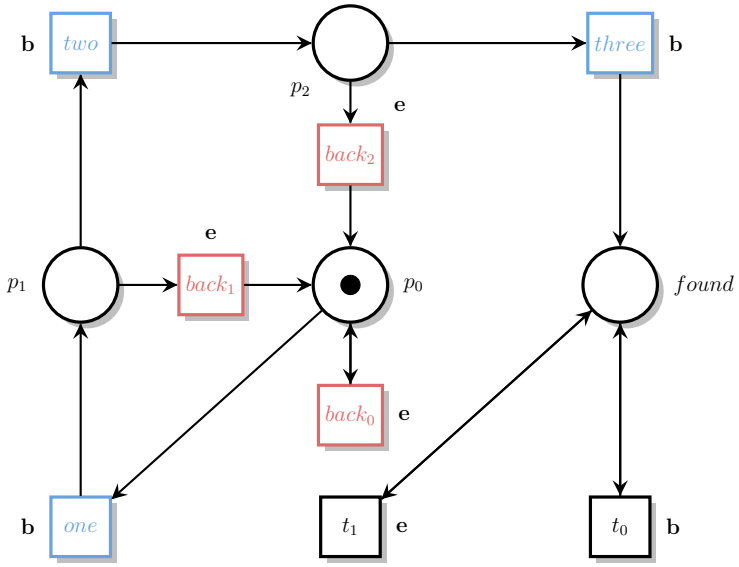


Figure 7.5: pattern for "three consecutive b without e"

The result depends on the timing constraints: let X be the (static) time interval for t_6, Y for t_7 and Z for t_{10} and t_{14}.

- $X = Y = Z = [0, \infty[$: the system is **not diagnosable** and has 126548 classes. The system without any timing constraint leads to a state space with 14270 markings
- $X = Y = [0, \infty[$ and $Z = [0, 6]$: the system is **not diagnosable** and has 129096 classes
- $X = Y = [0, \infty[$ and $Z = [1, 6]$: the system is **diagnosable** and has 15848 classes
- $X = Y = [0, 10]$ and $Z = [0, \infty[$: the system is **diagnosable** and has only 2186 classes

Depending on the timing constraints, the pattern is diagnosable or not. If the system is not diagnosable, we end up with almost 130000 classes, which is still scalable regarding the original size of the system and patterns.

We now quickly summarize our results.

7.4 Summary

In this Chapter 7 we presented our results, compared with a previous method (IPTPN).

- *PTPN* is generally more efficient than IPTPN in terms of size because it relies on the SCG construction contrary to the IPTPN which relies on the SSCG construction.
- *Untimed Patterns* can be detected on timed system with our method which is scalable.

The Chapter 8 will be focused on the diagnosability of timed patterns, which is an undergoing work.

Chapter 8

Timed Pattern Diagnosability

In the Chapter 6, the studied patterns have no time in their models. The question that naturally arises is: *is it possible to extend this method to patterns with time?*

To do this, we must first define the way in which time will be taken into account in the pattern, and then study how to perform the model-checking operation.

This Chapter is decomposed as follows: First, we introduce the problematic encountered for the analysis of timed patterns, with an example. We quickly go into one of our solutions and the needed hypothesis for it to work. The idea is also to detail our process, since this method will be automatized in future works. Finally, we conclude via a quick summary of this Chapter.

8.1 Problematic

One of the problematic we encounter when augmenting the notion of *pattern* was the idea to implement time inside of these modelled behaviour. When trying to check a *timed pattern*, we asked ourselves two main questions:

- Should time be relative to the previous event or absolute to the launch of the system?
- What do we do when the system does not behave on a path with the pattern? How do we take it into account?

For the first question we decided to tackle time as a relative value (the first event being the *start* of the system). We can still tackle a pattern without considering a *timed* first event with a pattern such as $a4b$ which means "*b 4 units of time after a*".

Let's take the following system in figure 8.1 as an example of a *wanted timed pattern*. The only observable labels are the o labels in this example.

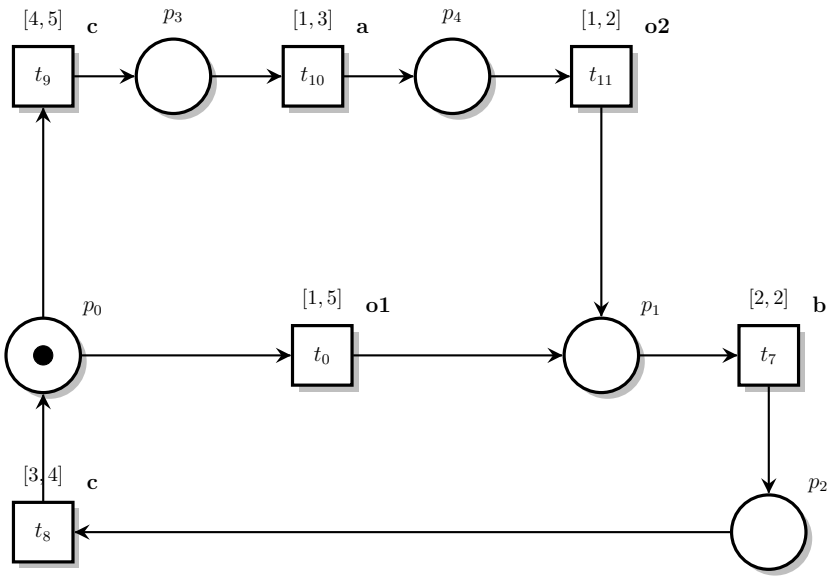


Figure 8.1: TPN "sys3"

In this system, we can easily show that there exists only one possible path to obtain the pattern $8a4b$ which means "*b 4 units of time after a and a 8 units of time after the start of the system*". Let's decompose the only possible path:

Marking :	p_0	p_3	p_4	p_1	p_2
Taken transitions :	$5t_9$	$5t_93t_{10}$	$5t_93t_{10}2t_{11}$	$5t_93t_{10}2t_{11}2t_7$	
Observation :			$10o_2$	$10o_2$	$10o_2$

Table 8.1: Decomposition of the path for $8a4b$

As you can see, the only possible outcome of an observation of $10o_2$ is the occurrence of b 4 units of time after an a label which occurs at 8 units of time after the start of the system. We can then conclude, since $10o_2$ is the only possible observable behaviour for this pattern, that the pattern $8a4b$ is diagnosable. Every time we would see a $10o_2$, we would conclude on the occurrence of the unobservable labels.

Now, for the second question: What happens if the system begins with the t_0 transition? Technically, it would create a difference of behaviour between the system and its pattern which would always result in a deadlock (since the first timing of the pattern needs an a to occur at 8).

If the system does not take the path with the timed event we want, how do we process it? Previously, in untimed patterns, we ended up in *deadlock* if the pattern was not found, but here, we would have a problem regarding the timed behaviour (which is related to the beginning of the system or the previous event). In an ideal system, the path would always try to have the wanted behaviour and we would conclude on the diagnosability of such behaviour.

This problem of *unwanted behaviour* was considered by the processing of a special kind of pattern for the diagnosability of timed behaviour.

One of our first ideas for a $8a$ pattern is the following figure 8.2. We wanted to end up with a more general approach with a classical *twin product* in the form $(N \times E) \times_L (N \times E)'$ (with N the system and E the pattern and N',E' their copy). In this pattern, we let the system act freely and we prioritize the occurrence of the first a transitions at 8. The orange arcs are priorities arcs which prioritize the origin transition (for example t_{p2} is more prioritize than t_{p3}).

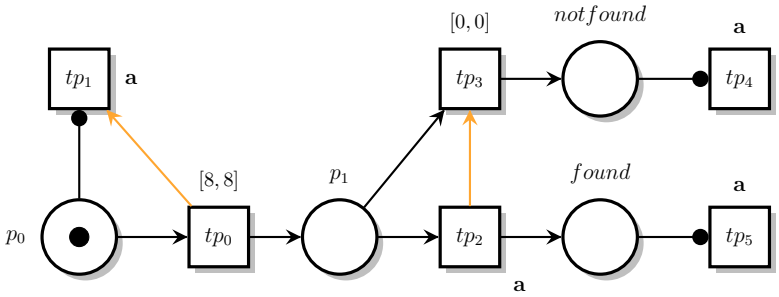


Figure 8.2: Prototype of a $8a$ pattern

In this case of a pattern E, we want to take the first a occurring at 8, but we do not force the system to absolutely take this behaviour. If it is possible to have $8a$, the pattern will indeed prioritize the occurrence of the pattern. We did not want to inferred on the system behaviour, but we did not mind the idea of prioritizing a possible occurrence.

However, this method to model-check the system was lacking in terms of feasibility. The idea behind a twin-product $(N \times E) \times_L (N \times E)'$ is to compare the system (with the pattern occurring if possible) and a copy of itself (without the pattern). In this case, the pattern is prioritized in the copy and the original, so it is impossible to compare with this product (since both will have the pattern if possible).

Another idea we had was to process a stronger pattern, in terms of imposing its behaviour on the system, and to compare the forced system and an unforced copy, such as the following operation $(N \times E) \times_L N'$. However, since the priorities in E were a downgrade in terms of memory, we processed a new kind of time pattern which forced its behaviour through the PTPN operation $\times$ between N and E.

Let's say we want to detect the following pattern of $8a4b$. An a label 8 after the start of the system precisely and a b after 4 units of time. We have the following pattern in figure 8.3:

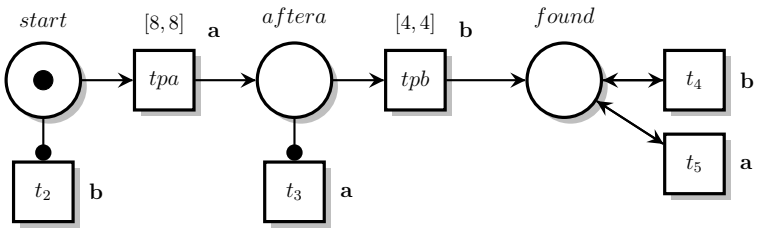


Figure 8.3: Pattern 8*a*4*b*

This pattern breaks several rules defined in section 6.4. This pattern is not total and it is interfering with the system behaviour (which is something we want here). The idea is to force the system N to synchronize and to follow the pattern E if possible. Then, we have to compare and to analyse all the paths that the system N takes with the pattern E and compare them with an unsynchronized copy of N.

In this case, we will treat the deadlock from the *unwanted behaviour* in the process of analysing this product.

We present this process in the following section 8.2.

8.2 Analysis process

First, we quickly describe all the tools used in the process chain before going into the details of the process for the operation of model-checking.

8.2.1 Tools used on the process chain

To process our previous idea of a timed pattern, we used several tools in the process chain.

- TINA : TINA builds various state space abstractions for Petri nets and Time Petri nets. We used it to build our original net and its pattern.
- TWINA : TWINA is a tool for analysing the "product" of two Time Petri Nets (TPN), with possibly inhibitor and read arcs. We used it to process our product (with $\times$ and $\times_L$).
- Muse : Muse model-checks state-event on a Kripke transition system given in ktz format. We used it to find the states where *found* became true.
- Pathto : Given a Kripke transition system (KTS) in a ktz file, a target state and a source state (default 0), computes a path in the ktz from the source to the destination state. We used it to find the path to the state where *found* became true.
- Plan : From a Time Petri net or Time Transition System and a firing sequence in .scn format, plan computes an inequality system characterizing all the times at which transitions in the sequence may fire. We used it to create the inequalities systems representing our path to the *found* places.

- Z3 : Z3 Theorem Prover is a cross-platform satisfiability modulo theories (SMT) solver. We used it to solve our inequations (and see if multiple observations can be found for the same pattern).

Now that we have quickly described our tools and their usage, we proceed to our process.

8.2.2 Process

The process we used is summarized in the following figure 8.4.

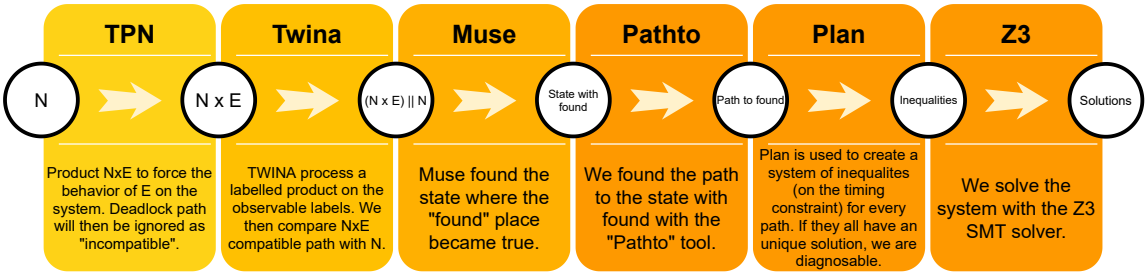


Figure 8.4: Process to analyses a Timed Pattern

The question asked is *"How do we tackle a path without* $8a$*?"*. To process this event we ended up creating a new product, ad-hoc, for this problematic. First, we process a PTPN of the system N and its pattern E and we want to find the *acceptable path* which tried to behave in the pattern. The process is decomposed as follows:

- We process the PTPN $N \times E$ to find the *acceptable path*. If the system cannot follow E behaviour, we will only obtain a deadlock. We know the desired behaviour impossible, hence its diagnosability.

- We now process and study the synchronous product $((N \times E) \times_L N)$ to search the state where we obtain a $found$ place;

- The analysis of the product $((N \times E) \times_L N)$ is done with the *muse* model-checker. The *twin* product $\times$ is done on labels, and $\times_L$ is done on observables labels (o_1 and o_2); We obtain a chain of states if there is any just as follows:

```
[31 , 32]
```

- With every state found, we can use the tool *pathto* to get all the paths leading to this particular state. For a state we can obtain a path as follows:

```
0 i 1 i 3 i 5 i 7 i 11 t9.1 12 t9.2.2 15 i 16 i 18 i
20 t10.1 21 tpa.1.2 t10.2.2 23 i 24 i 26 t11.1 t11.2.2
27 i 28 i 29 tpb.1.2 t7.2.2 31
```

With every i representing time and each number representing the state evolving.

- With these obtained paths, we can use the tool *plan* to get a system of inequalities representing all the static intervals of this path. We can obtain a system as follows:

```
{t9.2.2}$z1
{t10.2.2}$z4
{t11.2.2}$z6
{t7.2.2}$z8

where

0 <= start
-----------------
4 <= z1 - start <= 5
0 <= z4 - start
1 <= z4 - z1 <= 3
0 <= z6 - start
1 <= z6 - z4 <= 2
0 <= z8 - start
2 <= z8 - z6 <= 2
```

With each variable z_x representing the clock of transition (with the start as the reference).

- Finally, we can use a SMT (we use *z3*) to check if there is only a unique solution to the observable of these paths compatible with E.

This method however, still has issues. First of all, the created system from the product is big in terms of classes, transitions and places. The main problem of this method, in terms of scalability, is the creation of all of the different inequalities systems which have to be solved to conclude on the diagnosability of the timed pattern.

This method was then automated by linking all of the different tools with a *Python* parser called *Pollux*.

8.2.3 The Pollux parser

Pollux is a parser programmed in Python which is capable of processing the previously mentioned steps to analyse the diagnosability of timed patterns.

The process is explained step-by-step just like in the previous Subsection. First and foremost, the two inputs of Pollux are: The acceptable states with a $found$ place and the pattern in the form of the time associated to its transitions (or labels).

- The input, in the case of our example (in figure 8.1) is decomposed in two files:
  The state with $found$ becoming true:

  ```
  [31 , 32]
  ```

  The pattern:

  ```
  t10=8,t7=12
  ```

  For the next step, Pollux parses the state file to process the state one by one.

- Using the *pathto* tool, with the states processed one by one, the following results are obtained:
  Path0 for state 31:

  ```
  0 i 1 i 3 i 5 i 7 i 11 t9.1 12 t9.2.2 15 i 16 i 18 i
  20 t10.1 21 tpa.1.2 t10.2.2 23 i 24 i 26 t11.1 t11.2.2
  27 i 28 i 29 tpb.1.2 t7.2.2 31
  ```

  Path1 for state 32:

  ```
  0 i 1 i 3 i 5 i 7 i 11 t9.1 12 t9.2.2 15 i 16 i 18 i
  20 t10.1 21 tpa.1.2 t10.2.2 23 i 24 i 26 t11.1 t11.2.2
  27 i 28 i 29 t7.1 30 tpb.1.2 t7.2.2 32
  ```

With every i representing time and each number representing the state evolving. Those paths need to be processed in a form usable by the *plan* tool. The new files are called SCNPath.

- After the parsing of the two previous paths, the following results are obtained:
  SCNPath0 for state 31:

```
{t9.1}$0{t9.2.2}{t10.1}$0{tpa.1.2}$0{t10.2.2}{t11.1}
$0{t11.2.2}{tpb.1.2}$0{t7.2.2}
```

  SCNPath1 for state 32:

```
{t9.1}$0{t9.2.2}{t10.1}$0{tpa.1.2}$0{t10.2.2}{t11.1}
$0{t11.2.2}{t7.1}$0{tpb.1.2}$0{t7.2.2}
```

  Here, the idea in a SCNPath is to keep only the necessary transitions to create the inequalities with the *plan* tool.

- With these obtained paths, we can use the tool *plan* to get a system of inequalities representing all the static intervals of this path. We obtain systems as follows:
  System0 for SCNPath0:

```
{t9.1}$z0
{t9.2.2}$z1
{t10.1}$z2
{tpa.1.2}$z3
{t10.2.2}$z4
{t11.1}$z5
{t11.2.2}$z6
{tpb.1.2}$z7
{t7.2.2}$z8

where

0 <= start
-----------------
4 <= z0 - start <= 5
4 <= z1 - start <= 5
0 <= z1 - z0 <= 1
5 <= z2 - start <= 8
1 <= z2 - z0 <= 3
0 <= z2 - z1 <= 3
```

```
8 <= z3 - start <= 8
3 <= z3 - z1 <= 3
0 <= z3 - z2 <= 2
0 <= z4 - start
1 <= z4 - z1 <= 3
0 <= z4 - z2 <= 2
0 <= z4 - z3 <= 0
0 <= z5 - start
1 <= z5 - z2 <= 2
0 <= z5 - z3 <= 2
0 <= z5 - z4 <= 2
0 <= z6 - start
1 <= z6 - z3 <= 2
1 <= z6 - z4 <= 2
0 <= z6 - z5 <= 2
0 <= z7 - start
4 <= z7 - z3 <= 4
2 <= z7 - z5 <= 2
2 <= z7 - z6 <= 2
0 <= z8 - start
2 <= z8 - z5 <= 2
2 <= z8 - z6 <= 2
0 <= z8 - z7 <= 0
```

System1 for SCNPath1:

```
{t9.1}$z0
{t9.2.2}$z1
{t10.1}$z2
{tpa.1.2}$z3
{t10.2.2}$z4
{t11.1}$z5
{t11.2.2}$z6
{tpb.1.2}$z7
{t7.2.2}$z8

where

0 <= start
-----------------
4 <= z0 - start <= 5
4 <= z1 - start <= 5
0 <= z1 - z0 <= 1
```

```
5 <= z2 - start <= 8
1 <= z2 - z0 <= 3
0 <= z2 - z1 <= 3
8 <= z3 - start <= 8
3 <= z3 - z1 <= 3
0 <= z3 - z2 <= 2
```

...

The main issue here was to process these systems into file understandable by the *z3* solver. We also had to compute these with the pattern file to add the information into the system of inequalities.

- Just before using *z3* we have to process the new inequalities systems. The following results are obtained:
  Inequatilies0 for System0:

```
(declare-const z1 Int)
(assert (>= z1 4))
(assert (<= z1 5))
(declare-const z4 Int)
(assert (>= z4 0))
(assert (>= z4 (+ 1 z1)))
(assert (<= z4 (+ 3 z1)))
(declare-const z6 Int)
(assert (>= z6 0))
(assert (>= z6 (+ 1 z4)))
(assert (<= z6 (+ 2 z4)))
(declare-const z8 Int)
(assert (>= z8 0))
(assert (>= z8 (+ 2 z6)))
(assert (<= z8 (+ 2 z6)))
(assert (= z4 8))
(assert (= z8 12))
(declare-const y4 Int)
(assert (not (= y4 z4)))
(assert (>= y4 0))
(assert (>= y4 (+ 1 z1)))
(assert (<= y4 (+ 3 z1)))
(declare-const y8 Int)
(assert (not (= y8 z8)))
(assert (>= y8 0))
(assert (>= y8 (+ 2 z6)))
(assert (<= y8 (+ 2 z6)))
(check-sat)
(get-model)
```

Inequatilies1 for System1:

```
(declare-const z1 Int)
(assert (>= z1 4))
(assert (<= z1 5))
(declare-const z4 Int)
(assert (>= z4 0))
(assert (>= z4 (+ 1 z1)))
(assert (<= z4 (+ 3 z1)))
(declare-const z6 Int)
(assert (>= z6 0))
(assert (>= z6 (+ 1 z4)))
(assert (<= z6 (+ 2 z4)))
(declare-const z9 Int)
(assert (>= z9 0))
(assert (>= z9 (+ 2 z6)))
(assert (<= z9 (+ 2 z6)))
(assert (= z4 8))
(assert (= z9 12))
(declare-const y4 Int)
(assert (not (= y4 z4)))
(assert (>= y4 0))
(assert (>= y4 (+ 1 z1)))
(assert (<= y4 (+ 3 z1)))
(declare-const y9 Int)
(assert (not (= y9 z9)))
(assert (>= y9 0))
(assert (>= y9 (+ 2 z6)))
(assert (<= y9 (+ 2 z6)))
(check-sat)
(get-model)
```

- Finally, we can use a SMT solver (we use *z3*) to check if there is only a unique solution to the observable of these paths compatible with E. In this case, both systems are said *unsat* (for *unsatisfiable*), hence the diagnosability of this pattern.

Still, the process of all of the inequalities files is an issue in terms of memory and the scalability of Pollux is still untested. For this particular example, which is not that big, we process all the steps in $0,10$ seconds. We would also want to process a more general method which is presented in Chapter 9.

We now do a quick summary of this chapter.

8.3 Summary

In this Chapter 8 we presented an extension of the diagnosability of a pattern by adding timing constraints on it.

- *Timed Patterns* are relative to the start of the system in our case. After the first event we are relative to the previous event in terms of time.
- *Diagnosability of a timed pattern* relies on several tools to conclude by checking the timing constraints of the possible paths for the timed pattern behaviour.

This concludes the current work we had done on the diagnosability of timed patterns. We now proceed to the conclusion of this thesis.

Chapter 9

Conclusion

9.1 Overview

This Chapter 9 concludes our work. This thesis describes my contributions to the synchronous product of TPN and its applications in the domain of diagnosis (more precisely diagnosability analysis). I propose a new model, called PTPN, to create and ad-hoc synchronization between TPN and mimic the synchronous product behaviour in a TPN context. During this thesis, I also propose an algorithm to conduct our diagnosability analysis on our PTPN model. This approach is implemented in the tool TWINA which is tested on several benchmarks.

In Chapter 4, I have defined our new model, the Product TPN, which allows a synchronization between several TPN via their common labels. The idea is to force transitions with common labels to fire synchronously. With this in mind, I also tackle some new behaviours encountered in the PTPN models, such as the timelock.

In Chapter 5, I also process a new SCG model for our PTPN, to analyse directly the state classes of our PTPN model. The new behaviour is the firing of several transitions synchronized at the same time.

In Chapter 6, I tackle the property of diagnosability. The idea is to use the PTPN to create an ad-hoc synchronous product and to check the property of diagnosability (of a single fault or of a pattern).

In Chapter 7, I propose a test for our algorithm, compared with previous methods of analysis. I also use several benchmarks to test the scalability of our method and I conclude on its feasibility.

In Chapter 8, I explain one of the extension of our diagnosability of patterns. The diagnosability of timed patterns is explained with all its problematic regarding timing in the pattern. An ad-hoc solution for an example is then proposed.

Several publications are the result of the work done during this thesis:

- A State Class Construction for Computing the Intersection of Time Petri Nets Languages - August 2019
  Lubat, Éric and Dal Zilio, Silvano and Le Botlan, Didier and Pencolé, Yannick and Subias, Audine
  17th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)
  Amsterdam, Netherlands

- The tool TWINA construction d'espaces d'états abstrait pour l'intersection de Time Petri nets - November 2019
  Lubat, Éric
  12ème Colloque sur la Modélisation des Systèmes Réactifs (MSR 2019)
  Angers, France

- A Short Overview on Diagnosability of Patterns in Timed Petri Net - June 2020
  Lubat, Éric and Dal Zilio, Silvano
  14th Summer School on Modelling and Verification of Parallel Processes (MOVEP 2020)
  Grenoble (on line), France

- A New Product Construction for the Diagnosability of Patterns in Time Petri Net - December 2020
  Lubat, Éric and Dal Zilio, Silvano and Le Botlan, Didier and Pencolé, Yannick and Subias, Audine
  59th Conference on Decision and Control (CDC 2020)
  Jeju Island (on line), South-Korea

- Détection de Pattern temporisé dans les réseaux de Petri temporels - November 2021(Proceeding)
  Lubat, Éric and Hladik, Pierre-Emmanuel
  Modélisation des Systèmes Réactifs (MSR'21)
  Paris, France

I also wanted to talk about some future work and perspective for this thesis. I present some extensions we worked on, the *opacity*, the *hippo add-on*, *comparison with Uppaal*, *Prognosability* and *Pollux*.

9.2 Future works

Opacity: Our first extension would be to study more thoroughly the property of *Opacity*. Opacity is a basic property of Discrete Event Systems that relates to the "anonymity" of concealed events. It means that every secret event (which is usually unobservable) cannot be detected by an outside observer. Actually, many definitions for opacity only ask for the secret to stay undetected after a bounded number of operations.

This property can be expressed, as many properties in a DES, as a property on formal languages. Opacity has several possible uses: as a tool to express anonymity constraints; as a requirement for voting systems; as a security property in some military systems; etc.

We are interested by the study of opacity because it shares a lot of similarity with diagnosability, see for instance the work of Bérard [Bérard 2017] where the author directly connects the two properties.

However, during our study of this property we ended up on the paper *The dark side of Timed Opacity* [Cassez 2009]. In this paper, the author extends the notion of opacity, defined for DES, to a dense-time system. He also defines and studies the problem of timed opacity in TA. From the point of view of an attacker, time measurement gives a more accurate and realistic model of the system.

However, the author concludes that for a very restrictive class of TA, the opacity problem is already *undecidable*, leaving no hope for a decidable solution on a less restrictive model. Notice that his result carries over to other reasonable models of dense-time systems like Time Petri Nets (TPN), because TPN and TA are weakly timed bisimilar.

This problem was not tackled in our TPN study, but it was interesting to focus more on the time information and the branching information of our models. To extend the study of opacity we would have to create a weaker version of this property to conclude on a possibility of opacity for example. We now talk about the undergoing work on a tool called *HIPPO*.

HIPPO: The design of embedded real-time systems requires specific toolchains to guarantee time constraints and safe behaviour. These tools and their artefacts need to address timing constraints and execution semantics in a robust way during the modelling, verification and implementation of the system. HIPPO is a toolchain, that integrates tools for design, verification and execution built around a common formalism.

HIPPO is based on an extension of the Fiacre specification language with run-time features, such as asynchronous function calls and synchronization with events. We formally define the behaviour of these additions and describe a compiler to generate both an executable code and a verifiable model from the same high-level specification. The execution of the resulting code is supported by a dedicated execution engine that guarantees real-time behaviour and that reduces the semantic gap between high-level models and executable code.

HIPPO gets a Fiacre model and processes a real-time executable. Our contribution remains in the field of PTPN. We did work on HIPPO to create a PTPN behaviour on it. Since HIPPO is already an efficient toolbox for the simulation of real-time systems, we want to create an add-on which does not impact the overall behaviour of the HIPPO tool.

The idea is to create a PTPN using the HIPPO toolbox, by synchronizing transitions with the same label, just like in a classical PTPN. For this, we process a *C* library to add to the original HIPPO code. This was made with the idea to process an observer, with the goal of checking properties on the system defined in the HIPPO environment. An ad-hoc test was made to detect the occurrence of an event on the system *double-click* but the automation of the process is still undergoing development. This new feature of the HIPPO tools would need to be tested in terms of scalability.

Comparison with UPPAAL: One of the works we tried to process during the beginning of this thesis was a comparison, in terms of memory, speed and results between a twin-plant methods made with PTPN and one made with UPPAAL. The idea was to use a classical example, called *trains3*(https://projects.laas.fr/twina/post/examples/) which is a level-crossing example with 3 trains, to compare the two methods.

The synchronization between the different elements composing the trains system were made via PTPN and via a classical synchronous product for UPPAAL. However, to this day, the UPPAAL verifier cannot handle implication with a deadlock (which is a core idea to check diagnosability via a twin-plant methods), so we did not had the opportunity to finish this comparison.

Prognosability: A key property for the safety of system is the property of *prognosability*. Prognosability represents the ability for a system to be prognosed, or in other terms, it represents the ability to predict a failure (before its occurrence). This property is directly linked to the property of diagnosability studied through this thesis (for an overview on Diagnosability and Prognosability see [Vignolles 2020]). Indeed, if a failure is to be prognosed it need to be a diagnosable failure (since you cannot predict future behaviors which are not diagnosable), you can see more information about the necessity of diagnosability in a prognosis field in [Genc 2009].

Just like our study of diagnosability, we can process the property of prognosability by checking twin-plant algorithms or by solving optimization problems. Some works exist regarding the prognosability of extended Petri Nets, with upper and lower bound regarding timing of the possible prognosed behavior [Kanazy 2019], but to our knowledge, there is not a generalized method to verify the prognosability on Time Petri Nets.

A key future work would be to adapt first the analysis of single fault prognosability with PTPN.

Pollux: Finally, one of the possible future works would be to generalize the process in Chapter 8. The *Pollux* parser would also need a scalability test with a configurable example where you could decide the size of the system for a same pattern.

Bibliography

[Alur 1990] Rajeev Alur and David L. Dill. *Automata For Modeling Real-Time Systems.* In Mike Paterson, editor, Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990. (Cited in page 15.)

[Arnold 2002] André Arnold. *Nivat's processes and their synchronization.* Theoretical Computer Science, vol. 281, no. 1, 2002. (Cited in pages 18 and 43.)

[Aspvall 1979] Bengt Aspvall and Yossi Shiloach. *A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two Variables per Inequality.* In 20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 29-31 October 1979, pages 205–217. IEEE Computer Society, 1979. (Cited in page 54.)

[Basile 2012] Francesco Basile, Pasquale Chiacchio and Gianmaria De Tommasi. *On K-diagnosability of Petri nets via integer linear programming.* Automatica, vol. 48, no. 9, 9 2012. (Cited in page 26.)

[Basile 2017] F. Basile, M. P. Cabasino and C. Seatzu. *Diagnosability Analysis of Labeled Time Petri Net Systems.* IEEE Transactions on Automatic Control, vol. 62, no. 3, 2017. (Cited in page 27.)

[Basile 2018] Francesco Basile, Gianmaria De Tommasi, Claudio Sterle, Abderraouf Boussif and Mohamed Ghazel. *Efficient diagnosability assessment via ILP optimization: a railway benchmark.* In 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, pages 441–448, 2018. (Cited in pages 25 and 81.)

[Bérard 2005a] B. Bérard, F. Cassez, S. Haddad, Didier Lime and O. H. Roux. *Comparison of Different Semantics for Time Petri Nets.* In Doron A. Peled and Yih-Kuen Tsay, editors, Automated Technology for Verification and Analysis, pages 293–307, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. (Cited in page 31.)

[Bérard 2005b] Béatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime and Olivier H. Roux. *Comparison of the Expressiveness of Timed Automata and Time Petri Nets.* In Paul Pettersson and Wang Yi, editors, Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings, volume 3829 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005. (Cited in page 32.)

[Bérard 2008] Beatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime and Olivier H. Roux. *When are timed automata weakly timed bisimilar to time Petri nets?* Theoretical Computer Science, vol. 403, no. 2-3, 2008. (Cited in page 21.)

[Bérard 2013] Béatrice Bérard, Maria Paola Cabasino, Angela Di Febbraro, Alessandro Giua and Carla Seatzu. *Petri Nets with Time.* In Carla Seatzu, Manuel Silva and Jan H. van Schuppen, editors, Control of Discrete-Event Systems, volume 433 of *Lecture Notes in Control and Information Sciences*, pages 319–341. Springer, 2013. (Cited in pages 16 and 17.)

[Bérard 2017] Beatrice Bérard, Stefan Haar, Sylvain Schmitz and Stefan Schwoon. *The Complexity of Diagnosability and Opacity Verification for Petri Nets.* Application and Theory of Petri Nets and Concurrency, 2017. (Cited in pages 26 and 114.)

[Berthomieu 1983] B. Berthomieu and M. Menasche. *An enumerative approach for analyzing time Petri nets.* In Proceedings IFIP, 1983. (Cited in pages 5, 6, 17, 23, 27, 33, 46, 53, 54, 56 and 71.)

[Berthomieu 1991] B. Berthomieu and M. Diaz. *Modeling and Verification of Time Dependent Systems Using Time Petri Nets.* IEEE Trans. on Software Engineering, vol. 17, no. 3, 1991. (Cited in pages 53, 56 and 69.)

[Berthomieu 2003] Bernard Berthomieu and François Vernadat. *State Class Constructions for Branching Analysis of Time Petri Nets.* In Hubert Garavel and John Hatcliff, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 442–457. Springer Berlin Heidelberg, 2003. (Cited in page 71.)

[Berthomieu 2004] B. Berthomieu, P.-O. Ribet and F. Vernadat. *The tool TINA–Construction of Abstract State Spaces for Petri Nets and Time Petri Nets.* International Journal of Production Research, vol. 42, no. 14, 2004. (Cited in pages 58 and 93.)

[Berthomieu 2006] B. Berthomieu, F. Peres and F. Vernadat. *Bridging the Gap Between Timed Automata and Bounded Time Petri Nets.* In Formal Modeling and Analysis of Timed Systems (FORMATS), volume 4202 of *LNCS*. Springer, 2006. (Cited in pages 32 and 37.)

[Berthomieu 2008a] Bernard Berthomieu, Hubert Garavel, Frédéric Lang and François Vernadat. *Verifying Dynamic Properties of Industrial Critical Systems Using TOPCASED/FIACRE.* ERCIM News, vol. 2008, no. 75, 2008. (Cited in page 7.)

[Berthomieu 2008b] Bernard Berthomieu, Florent Peres and François Vernadat. *Abstract State Spaces for Time Petri Nets Analysis.* In 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing

(ISORC 2008), 5-7 May 2008, Orlando, Florida, USA, pages 298–304. IEEE Computer Society, 2008. (Cited in page 24.)

[Bhargavan 2006] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon and Stephen Tse. *Verified Interoperable Implementations of Security Protocols*. In 19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy, pages 139–152. IEEE Computer Society, 2006. (Cited in page 2.)

[Blanchet 2004] Bruno Blanchet. *Automatic Proof of Strong Secrecy for Security Protocols*. In 2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA, page 86. IEEE Computer Society, 2004. (Cited in page 2.)

[Boussif 2016] Abderraouf Boussif, Baisi Liu and Mohamed Ghazel. *A twin-plant based approach for diagnosability analysis of intermittent failures*. In Christos G. Cassandras, Alessandro Giua and Zhiwu Li, editors, 13th International Workshop on Discrete Event Systems, WODES 2016, Xi'an, China, May 30 - June 1, 2016, pages 237–244. IEEE, 2016. (Cited in page 21.)

[Boussif 2021] Abderraouf Boussif, Mohamed Ghazel and João Carlos Basilio. *Intermittent fault diagnosability of discrete event systems: an overview of automaton-based approaches*. Discret. Event Dyn. Syst., vol. 31, no. 1, pages 59–102, 2021. (Cited in page 27.)

[Boyer 2008] Marc Boyer and Olivier H. Roux. *On the Compared Expressiveness of Arc, Place and Transition Time Petri Nets*. Fundam. Informaticae, vol. 88, no. 3, pages 225–249, 2008. (Cited in page 16.)

[Brihaye 2017] Thomas Brihaye, Gilles Geeraerts, Hsi-Ming Ho and Benjamin Monmege. *MightyL: A Compositional Translation from MITL to Timed Automata*. In International Conference on Computer Aided Verification, pages 421–440. Springer, 2017. (Cited in page 25.)

[Cassandras 2009] Christos G. Cassandras and Stephane Lafortune. Introduction to discrete event systems. Springer-Verlag, 2009. (Cited in pages 5 and 11.)

[Cassez 2006] Franck Cassez and Olivier H Roux. *Structural translation from time Petri nets to timed automata*. Journal of Systems and Software, vol. 79, no. 10, 2006. (Cited in page 36.)

[Cassez 2009] Franck Cassez. *The Dark Side of Timed Opacity*. In Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim and Sang-Soo Yeo, editors, Advances in Information Security and Assurance, Third International Conference and Workshops, ISA 2009, Seoul, Korea, June 25-27, 2009. Proceedings, volume 5576 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2009. (Cited in page 115.)

[Clarke 1981] Edmund M. Clarke and E. Allen Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In Dexter Kozen, editor, Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981. (Cited in pages 11 and 22.)

[Clarke 1986] Edmund M. Clarke, E. Allen Emerson and A. Prasad Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Trans. Program. Lang. Syst., vol. 8, no. 2, pages 244–263, 1986. (Cited in page 24.)

[Clarke 1988] Edmund M. Clarke and I. A. Draghicescu. *Expressibility results for linear-time and branching-time logics*. In J. W. de Bakker, Willem P. de Roever and Grzegorz Rozenberg, editors, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 1988. (Cited in page 23.)

[Clarke 1999] Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model checking. MIT press, 1999. (Cited in pages 4, 22 and 23.)

[Cohen 2000] Guy Cohen. *Discrete Event Systems*. https://www.rocq.inria.fr/metalau/cohen/SED/index-e.html, Jan 2000. (Cited in page 12.)

[Contant 2004] Olivier Contant, Stéphane Lafortune and Demosthenis Teneketzis. *Diagnosis of Intermittent Faults*. Discret. Event Dyn. Syst., vol. 14, no. 2, pages 171–202, 2004. (Cited in page 27.)

[Dal Zilio 2019] Silvano Dal Zilio and Éric Lubat. *TWINA - A realtime model-checker for analyzing Twin-TPN*, 2019. (Cited in page 93.)

[de Ruiter 2011] Joeri de Ruiter and Erik Poll. *Formal Analysis of the EMV Protocol Suite*. In Sebastian Mödersheim and Catuscia Palamidessi, editors, Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers, volume 6993 of *Lecture Notes in Computer Science*, pages 113–129. Springer, 2011. (Cited in page 2.)

[Garavel 2012] Hubert Garavel. *Three Decades of Success Stories in Formal Methods*. In International Conference on Formal Methods for Industrial Critical Systems (FMICS), 2012. (Cited in page 2.)

[Genc 2009] Sahika Genc and Stéphane Lafortune. *Predictability of event occurrences in partially-observed discrete-event systems*. Autom., vol. 45, no. 2, pages 301–311, 2009. (Cited in page 116.)

[Ghazel 2009] Mohamed Ghazel, Armand Toguyéni and Pascal Yim. *State Observer for DES Under Partial Observation with Time Petri Nets.* Discrete Event Dynamic Systems, vol. 19, no. 2, 2009. (Cited in page 25.)

[Gonthier 2007] Georges Gonthier. *The Four Colour Theorem: Engineering of a Formal Proof.* In Deepak Kapur, editor, Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. (Cited in page 2.)

[Gougam 2017] Houssam-Eddine Gougam, Yannick Pencolé and Audine Subias. *Diagnosability analysis of patterns on bounded labeled prioritized Petri nets.* Discrete Event Dynamic Systems, vol. 27, no. 1, 2017. (Cited in pages 87, 93, 95 and 96.)

[Hand 2020] David J. Hand and Shakeel Khan. *Validating and Verifying AI Systems.* Patterns, vol. 1, no. 3, page 100037, 2020. (Cited in page 3.)

[Henzinger 1992] Thomas A. Henzinger, Zohar Manna and Amir Pnueli. *Timed transition systems.* In J. W. de Bakker, C. Huizing, W. P. de Roever and G. Rozenberg, editors, Real-Time: Theory in Practice, pages 226–251, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. (Cited in page 13.)

[Henzinger 1998] T. A. Henzinger, J.-F. Raskin and P.-Y. Schobbens. *The regular real-time languages.* In Kim G. Larsen, Sven Skyum and Glynn Winskel, editors, Automata, Languages and Programming, pages 580–591, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. (Cited in page 25.)

[Hladik 2021] Pierre-Emmanuel Hladik, Félix Ingrand, Silvano Dal Zilio and Reyyan Tekin. Hippo: A Formal-Model Execution Engine to Control and Verify Critical Real-Time Systems. working paper or preprint, April 2021. (Cited in page 7.)

[Hunt 1994] Warren A. Hunt. FM8501: A verified microprocessor, volume 795 of *Lecture Notes in Computer Science.* Springer, 1994. (Cited in page 2.)

[Jéron 2006] Thierry Jéron, Hervé Marchand, Sophie Pinchinat and Marie-Odile Cordier. *Supervision patterns in discrete event systems diagnosis.* In International Workshop on Discrete Event Systems, 2006. (Cited in page 87.)

[Jiang 2001a] S. Jiang, Z. Huang, V. Chandra and R. Kumar. *A polynomial algorithm for testing diagnosability of discrete-event systems.* IEEE Transactions on Automatic Control, vol. 46, no. 8, 2001. (Cited in pages 6, 27 and 81.)

[Jiang 2001b] Shengbing Jiang, Zhongdong Huang, Vigyan Chandra and Ratnesh Kumar. *A polynomial algorithm for testing diagnosability of discrete-event systems.* Transactions on Automatic Control, vol. 46, no. 8, 8 2001. (Cited in page 26.)

[Jiang 2003] Shengbing Jiang, Ratnesh Kumar and Humberto E. Garcia. *Diagnosis of repeated/intermittent failures in discrete event systems.* IEEE Trans. Robotics Autom., vol. 19, no. 2, pages 310–323, 2003. (Cited in page 27.)

[Kanazy 2019] Redouane Kanazy, Samir Chafik, Éric Niel and Mohamed Zouagui. *Failure prognosis in discrete events systems based on extended Time petri nets: example of an electric car battery cell.* In 4th Conference on Control and Fault Tolerant Systems, SysTol 2019, Casablanca, Morocco, September 18-20, 2019, pages 276–281. IEEE, 2019. (Cited in page 116.)

[Keller 1976] Robert M. Keller. *Formal Verification of Parallel Programs.* Commun. ACM, vol. 19, no. 7, pages 371–384, July 1976. (Cited in page 13.)

[Lai 2008] Stefano Lai, Davide Nessi, Maria Paola Cabasino, Alessandro Giua and Carla Seatzu. *A comparison between two diagnostic tools based on automata and Petri nets.* In 9th International Workshop on Discrete Event Systems, Göteborg, Sweden, 5 2008. (Cited in page 22.)

[Larsen 1997] Kim G Larsen, Paul Pettersson and Wang Yi. *UPPAAL in a nutshell.* International journal on software tools for technology transfer, vol. 1, no. 1, pages 134–152, 1997. (Cited in page 21.)

[Lime 2003a] Didier Lime and Olivier H. Roux. *State class timed automaton of a time Petri net.* In Proceedings of the 10th International Workshop on Petri Nets and Performance Models, PNPM 2003, Urbana-Champaign, IL, USA, September 2-5, 2003, pages 124–133. IEEE Computer Society, 2003. (Cited in pages 21 and 94.)

[Lime 2003b] Didier Lime and Olivier Henri Roux. *State class timed automaton of a time Petri net.* In The 10th International Workshop on Petri Nets and Performance Models,(PNPM'03), 2003. (Cited in page 36.)

[Lin 1994] Feng Lin. *Diagnosability of discrete event systems and its applications.* Journal of Discrete Event Dynamic Systems: Theory and Applications, vol. 4, no. 2, 5 1994. (Cited in page 26.)

[Liu 2014] Baisi Liu, Mohamed Ghazel and Armand Toguyeni. *Diagnosis of Labeled Time Petri Nets Using Time Interval Splitting.* IFAC Proceedings Volumes, vol. 47, no. 3, 2014. (Cited in pages 25 and 27.)

[Lubat 2019] Éric Lubat, Silvano Dal Zilio, Didier Le Botlan, Yannick Pencolé and Audine Subias. *A State Class Construction for Computing the Intersection of Time Petri Nets Languages.* In Formal Modeling and Analysis of Timed Systems (FORMATS), volume 11750 of *LNCS*. Springer, 2019. (Cited in pages 6, 7 and 21.)

[Merlin 1974] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems.* PhD thesis, University of California, Irvine, 1974. (Cited in page 16.)

[Moller 1990] Faron Moller and Chris Tofts. *TCCS : A Temporal Calculus of Communicating Systems (DRAFT).* 09 1990. (Cited in page 21.)

[Neumann 1994] Peter G. Neumann. *Illustrative Risks to the Public in the Use of Computer Systems and Related Technology.* SIGSOFT Softw. Eng. Notes, vol. 19, no. 1, pages 16–29, January 1994. (Cited in page 1.)

[Ouaknine 2005] Joël Ouaknine and James Worrell. *On the decidability of metric temporal logic.* In 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05), pages 188–197. IEEE, 2005. (Cited in page 25.)

[Pencolé 2018] Yannick Pencolé and Audine Subias. *Timed pattern diagnosis in timed workflows: a model checking approach.* IFAC-PapersOnLine, vol. 51, no. 7, 2018. (Cited in page 87.)

[Pencolé 2021] Yannick Pencolé and Audine Subias. *Diagnosability of event patterns in safe labeled time Petri nets: a model-checking approach.* IEEE Transactions on Automation Science and Engineering, 2021. (Cited in page 27.)

[Peres 2011] F. Peres, B. Berthomieu and F. Vernadat. *On the Composition of Time Petri Nets.* Discrete Event Dynamic Systems, vol. 21, no. 3, 2011. (Cited in pages 6, 35, 36, 93 and 94.)

[Petri 1962] C. A. Petri. *Fundamentals of a Theory of Asynchronous Information Flow.* In Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962, pages 386–390. North-Holland, 1962. (Cited in pages 4 and 16.)

[Pnueli 1977] Amir Pnueli. *The Temporal Logic of Programs.* In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pages 46–57. IEEE Computer Society, 1977. (Cited in page 23.)

[Popova-Zeugmann 2013] Louchka Popova-Zeugmann. Time and petri nets. Springer, 2013. (Cited in pages 38 and 39.)

[Popova 1991] L. Popova. *On Time Petri Nets.* J. Information Processing and Cybernetics EIK, vol. 27, no. 4, 1991. (Cited in page 13.)

[PRIOR 1957] A. N. PRIOR. *Time and Modality.* Zeitschrift für Philosophische Forschung, vol. 13, no. 3, pages 477–479, 1957. (Cited in page 23.)

[Queille 1982] Jean-Pierre Queille and Joseph Sifakis. *Specification and verification of concurrent systems in CESAR.* In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. (Cited in pages 11 and 22.)

[Ramadge 1989] Peter JG Ramadge and W Murray Wonham. *The control of discrete event systems.* Proceedings of the IEEE, vol. 77, no. 1, 1989. (Cited in page 4.)

[Ramalingam 1995] G. Ramalingam, J. Song, L. Joscovicz and R. E. Miller. *Solving Difference Constraints Incrementally.* Algorithmica, vol. 23, 1995. (Cited in page 53.)

[Sampath 1995] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen and Demosthenis Teneketzis. *Diagnosability of discrete-event systems.* IEEE Transactions on automatic control, vol. 40, no. 9, 1995. (Cited in pages 25 and 26.)

[Tarjan 1972] Robert Tarjan. *Depth-first search and linear graph algorithms.* SIAM journal on computing, vol. 1, no. 2, 1972. (Cited in pages 83 and 84.)

[Tripakis 2002] Stavros Tripakis. *Fault Diagnosis for Timed Automata.* In 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), 2002. (Cited in pages 25, 26 and 84.)

[Ushio 1998] Toshimitsu Ushio, Isao Onishi and Koji Okuda. *Fault detection based on Petri net models with faulty behaviors.* In Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on, volume 1. IEEE, 1998. (Cited in page 26.)

[Vignolles 2020] Amaury Vignolles, Elodie Chanthery and Pauline Ribot. *An overview on diagnosability and prognosability for system monitoring.* In EUROPEAN CONFERENCE OF THE PROGNOSTICS AND HEALTH MANAGEMENT SOCIETY (PHM Europe), (Virtual conference), Italy, July 2020. (Cited in page 116.)

[Wang 2015] Xu Wang, Cristian Mahulea and Manuel Silva. *Diagnosis of time Petri nets using fault diagnosis graph.* IEEE Transactions on Automatic Control, vol. 60, no. 9, 2015. (Cited in pages 27 and 95.)

[Xue 2004] Fei Xue and Da-Zhong Zheng. *Diagnosability for discrete event systems based on Petri net language.* In 8th International Conference on Control, Automation, Robotics and Vision, ICARCV 2004, Kunming, China, 6-9 December 2004, Proceedings, pages 2111–2116. IEEE, 2004. (Cited in page 26.)

[Yoo 2002] Tae-Sic Yoo and Stéphane Lafortune. *Polynomial-time verification of diagnosability of partially observed discrete-event systems.* IEEE Transactions on automatic control, vol. 47, no. 9, 2002. (Cited in pages 26, 28 and 82.)

Abstract: We study the behaviour of *Discrete Event Systems* (DES) subject to strong temporal constraints. We are more particularly interested in the formal verification of properties on the timed languages associated with their executions. In this context, we focus on DES modelled using *Time Petri Nets* (TPN), an extension of classical Petri nets in which we can constrain the time during which transitions stay enabled.

Our goal is to use and extend techniques borrowed from model-checking in order to check properties related to the *diagnosability* of a system. To this end, we study properties on the *intersection* of the timed languages of systems. Our approach is based on the definition of a new composition operator, that we call *synchronous product*, that constrain different transitions to fire at the same time. This allows us to analyse the product of systems more directly, without the need to compute the intersection of their language at the level of their state spaces.

Our main contribution is the definition of a new formal model, called *Product TPN* (PTPN), that includes our notion of synchronous product in its syntax. We show how to extend the notion of *State Class Graphs* to PTPN and use this construction to check the *diagnosability* of single faults on TPN. We also study the diagnosability of more complex behaviours, expressed using *patterns* of events, and explore a restricted case of *timed pattern*.

Keywords: Discrete Event Systems, Verification, Model-checking, Time Petri Nets, Synchronous Product, Diagnosability, Pattern

Résumé : Cette thèse porte sur l'étude des *Systèmes à Événements Discrets* (SED) soumis à des contraintes temporelles fortes, et plus précisément sur la vérification de propriétés liées aux langages associés à leurs exécutions. Dans ce contexte, nous nous concentrons à l'étude des *réseaux de Petri temporels* (TPN) comme modèle pour la spécification des SED.

L'objectif général est d'utiliser et d'étendre des méthodes issues du domaine du model-checking afin de répondre à des questions de diagnosticabilité. Pour ce faire, nous cherchons à vérifier des propriétés liées à *l'intersection* entre les langages temporels (le comportement) de différent systèmes. Notre approche repose sur la définition d'une nouvelle opération de *produit synchrone* entre TPN qui nous permet d'utiliser des techniques d'analyse plus directes. Ceci nous permet, en particulier, d'éviter de devoir calculer directement l'intersection entre langages au niveau des espaces d'état des systèmes.

Notre contribution principale est la définition d'un nouveau modèle, les *Product TPN* (PTPN), qui internalise notre concept de produit synchrone entre transitions. Nous proposons une extension de la notion de graphes de classes au cas des PTPN et utilisons ce modèle pour vérifier la propriété de *diagnosabilité* sur les TPN dans le cas de fautes simples, mais également pour la diagnosticabilité de scénarios plus complexes, décrit sous la forme de *motifs*.

Mots clés : Systèmes à événements discrets, Vérification formelle, Model-checking, réseaux de Petri temporels, Produit synchrone, diagnosticabilité, motif