



THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Modeling Distributed Systems with Petri Nets and Temporal Logic

Ligny, Bernard; Raskin, Jean-François

Award date:
1995

Awarding institution:
University of Namur

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

**Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique - Rue Grandgagnage, 21
5000 NAMUR**

**Modeling Distributed Systems with
Petri Nets and Temporal Logic**

Bernard Ligny et Jean-François Raskin
Troisième licence et maîtrise en informatique

*Mémoire présenté le 28 juin 1995 en vue de l'obtention
du titre de licencié et maître en informatique*

Promoteur : Mr Eric DUBOIS

Co-promoteur : Mr Pierre-Yves SCHOBENS

Remerciements

Nous tenons tout d'abord à exprimer nos plus vifs remerciements aux chercheurs et membres du personnel du Département Euridis de l'Université de Rotterdam. Durant notre stage, nous avons bénéficié d'une excellente ambiance propice au travail. Nos remerciements vont plus particulièrement à Yao-Hua Tan et Leendert van der Torre pour leurs conseils judicieux et l'attention qu'ils ont portée à nos travaux et ainsi qu'à Marianne Oostdijk pour son dévouement et sa disponibilité.

Nous sommes également reconnaissants envers nos deux promoteurs, Eric Dubois et Pierre-Yves Schobbens, pour le temps qu'ils ont consacré aux réunions et à la lecture de notre mémoire. De plus, nous avons pu bénéficier, tout au long de notre travail, de leur grande expertise dans le domaine des langages de spécification formels.

Enfin, nous tenons à remercier nos proches pour le soutien tant moral que matériel qu'ils nous ont apporté et sans lequel rien n'aurait été pareil.

Résumé

Dans ce mémoire, nous définissons un langage de spécification formel pour systèmes coopératifs, dynamiques et distribués. Ce langage (PNRTL) est basé sur les réseaux de Petri, la logique temporelle temps-réel et la logique déontique.

Une spécification dans notre langage est composée de trois objets. Un réseau représente les actions et aspects dynamiques du système modélisé. Un ensemble de formules de la logique temporelle a pour but de restreindre les exécutions possibles du réseau à l'ensemble de ses exécutions désirées. Enfin, un système de poids est associé aux transitions du réseau pour modéliser des préférences entre les exécutions du système, cela permet de représenter la notion déontique de sub-idéalité.

Une étude de cas met en évidence la capacité de notre langage à exprimer des contraintes opérationnelles, déclaratives, temps-réels, ... et des aspects déontiques.

Nous proposons également deux approches pour réaliser des preuves formelles sur les spécifications PNRTL : la première est basée sur la théorie des automates de Büchi et la seconde sur les systèmes de preuves logiques.

Abstract

In this master thesis, we define a formal specification language for cooperative, dynamic and distributed systems. This language (PNRTL) is based on the formalisms of Petri nets, real-time temporal logic and deontic logic.

A specification in our language is composed of three objects. A net represents the dynamic aspects of the modeled system. A set of temporal logic formulae prunes the set of possible executions of the net to the set of desired executions of the system. A weight system associated with the transitions of the net models a preference ordering on the behaviors of the system, this allows to represent the notion of deontic sub-ideality.

A case study shows the ability of our language to express operational, declarative, real-time, ... constraints and deontic aspects.

We also propose two different approaches to conduct formal proof of properties of the PNRTL specifications : the first one is based on the Büchi automata theory while the second one rests on logical proof systems.

Table of Contents

INTRODUCTION.....	0.1
-------------------	-----

PART ONE : FORMALISMS

CHAPTER 1 : THE PETRI NET FORMALISM.....	1.1
--	-----

1.1 INTRODUCTION	1.1
1.2 BASIC PETRI NETS	1.2
1.2.1 <i>A few definitions</i>	1.4
1.2.2 <i>Verification of properties of Petri nets</i>	1.7
1.3 PREDICATE/TRANSITION NETS	1.10
1.4 COLORED PETRI NETS	1.12
1.5 TIMED PETRI NETS.....	1.15
1.6 DOCUMENTARY PETRI NETS.....	1.18

CHAPTER 2 : THE TEMPORAL LOGIC FORMALISM.....	2.1
---	-----

2.1 INTRODUCTION	2.1
2.2 TEMPORAL LOGIC	2.1
2.2.1 <i>Modal Logic</i>	2.1
2.2.2 <i>Linear Temporal Frame</i>	2.3
2.2.3 <i>Temporal Logic in Computer Science</i>	2.3
2.3 REAL-TIME TEMPORAL LOGIC	2.4
2.4 A TEMPORAL LOGIC BASED LANGUAGE : ALBERT	2.5
2.4.1 <i>The ALBERT Language</i>	2.5
2.4.2 <i>Main concepts</i>	2.6
2.4.3 <i>Agent declaration</i>	2.7
2.4.4 <i>ALBERT phrases</i>	2.8
2.4.5 <i>Constraints</i>	2.9

CHAPTER 3 : TEMPORAL LOGIC VERSUS PETRI NETS	3.1
--	-----

3.1 INTRODUCTION	3.1
3.2 COMPARISON OF BOTH APPROACHES.....	3.2
3.2.1 <i>Concurrency, choice and synchronization</i>	3.2
3.2.2 <i>Causality</i>	3.5
3.2.3 <i>Determinism and non-determinism</i>	3.7
3.2.4 <i>Conclusion</i>	3.7
3.3 TRANSFORMATION OF PETRI NETS INTO ALBERT.....	3.8
3.3.1 <i>The mapping of the places and tokens</i>	3.8
3.3.2 <i>The mapping of the transitions</i>	3.9
3.3.3 <i>The mapping of the arcs</i>	3.9
3.3.4 <i>The simultaneity</i>	3.11
3.3.5 <i>Illustration of the transformation rule R</i>	3.13
3.3.6 <i>Summary and conclusion</i>	3.15

CHAPTER 4 : DEONTIC LOGIC	4.1
4.1 INTRODUCTION	4.1
4.2 SDL : A MODAL LOGIC FOR DEONTIC REASONING	4.2
4.3 DEONTIC ASPECTS IN OUR WORK	4.4

PART TWO : A NEW INTEGRATED LANGUAGE

CHAPTER 5 : PETRI NETS AND DEONTIC ASPECTS	5.1
5.1 INTRODUCTION	5.1
5.2 STRICT OBLIGATIONS/PROHIBITIONS	5.1
5.2.1 <i>Strict obligations</i>	5.2
5.2.2 <i>Strict prohibitions</i>	5.6
5.2.3 <i>Limitations of the approach</i>	5.8
5.3 MODELING VARYING SUB-IDEALITY IN PETRI NETS	5.9
5.3.1 <i>Introduction</i>	5.9
5.3.2 <i>The extended Petri net formalism</i>	5.11
5.4 CONCLUSION	5.15

CHAPTER 6 : PETRI NETS AND TEMPORAL LOGIC	6.1
6.1 INTRODUCTION	6.1
6.2 OPERATIONAL SEMANTICS OF PETRI NETS	6.3
6.3 THE LOGIC FORMULAE OF THE LANGUAGE	6.6
6.3.1 <i>State formulae</i>	6.6
6.3.2 <i>Temporal formulae</i>	6.10
6.3.3 <i>The desired executions</i>	6.12
6.4 A SPECIFICATION IN PNTL	6.13
6.4.1 <i>A model of this case in the common Petri nets</i>	6.13
6.4.2 <i>The PNTL specification of the case</i>	6.14
6.4.3 <i>Evaluations</i>	6.15

CHAPTER 7 : THE PNRTL LANGUAGE	6.1
7.1 MOTIVATIONS	7.1
7.2 NEW CONCEPTS	7.2
7.3 REAL TIME PR/TR NETS	7.5
7.3.1 <i>Introduction</i>	7.5
7.3.2 <i>Introduction of Real-Time</i>	7.5
7.3.3 <i>Firing rule and operational semantics</i>	7.9
7.4 LOGICAL FORMULAE OF PNRTL	7.16
7.4.1 <i>State formulae</i>	7.16
7.4.2 <i>Real Time Temporal formulae</i>	7.18
7.5 MANY-SORTED STRUCTURES	7.23
7.6 A SPECIFICATION IN SEPARATE SUB-NETS	7.26

PART THREE : APPLICATIONS AND TOOLS

CHAPTER 8 : A CASE STUDY IN PNRTL.....	8.1
8.1 DESCRIPTION OF THE CASE.....	8.1
8.2 SPECIFICATION OF THE CASE.....	8.2
8.3 CONCLUSION.....	8.12
CHAPTER 9 : DESIRED EXECUTIONS OF A PNTL SPECIFICATION.....	9.1
9.1 INTRODUCTION.....	9.1
9.2 INFINITE EXECUTIONS ON BOUNDED PETRI NETS.....	9.2
9.3 OVERVIEW OF THE AUTOMATA THEORY.....	9.2
9.3.1 <i>Finite automata on finite words</i>	9.2
9.3.2 <i>Finite automata on infinite words</i>	9.3
9.3.3 <i>Generalized Büchi automata</i>	9.5
9.3.4 <i>Closure properties of ω-regular languages</i>	9.6
9.4 BÜCHI AUTOMATA AND BOUNDED PETRI NETS.....	9.6
9.4.1 <i>From Petri nets to automata</i>	9.6
9.4.2 <i>The set of possible executions</i>	9.8
9.4.3 <i>The library example</i>	9.9
9.4.4 <i>An algorithm</i>	9.10
9.4.5 <i>A more general algorithm</i>	9.13
9.4.6 <i>Mapping Petri nets into Büchi automata</i>	9.14
9.5 REDUCTION OF THE SET OF POSSIBLE EXECUTIONS.....	9.16
9.5.1 <i>Hypothesis</i>	9.16
9.5.2 <i>Constraints on states and constraints on words</i>	9.17
9.5.3 <i>Reduction when dealing with a constraint on words</i>	9.17
9.5.4 <i>Reduction when dealing with a constraint on states</i>	9.20
9.5.5 <i>Calculating the intersection of two languages</i>	9.22
9.6 TESTING OF PROPERTIES ON THE SET OF DESIRED EXECUTIONS.....	9.24
9.7 EXTENSION TO THE LINEAR TEMPORAL LOGIC.....	9.26
9.7.1 <i>About the expressiveness of the linear temporal logic</i>	9.26
9.7.2 <i>Extended temporal logic</i>	9.27
CHAPTER 10 : LOGICAL PROOFS IN PNTL AND PNRTL.....	10.1
10.1 INTRODUCTION.....	10.1
10.2 LOGICAL PROOFS IN PNTL.....	10.1
10.2.1 <i>Part A : the pure logic part</i>	10.2
10.2.2 <i>Part B : the domain part</i>	10.3
10.2.3 <i>Part C : the net part</i>	10.4
10.2.4 <i>A proof example in PNTL</i>	10.5
10.3 A PROOF SYSTEM FOR PNRTL.....	10.7
10.3.1 <i>Formal characterization of the distance function d</i>	10.7
10.3.2 <i>Axiomatization of the PNRTL nets</i>	10.8
10.3.3 <i>Axioms for real-time temporal operators</i>	10.10
10.3.4 <i>Two proof examples in PNRTL</i>	10.12
10.4 CONCLUSION.....	10.13
CONCLUSION.....	11.1
BIBLIOGRAPHICAL REFERENCES.....	B.1

PART ONE
FORMALISMS

Introduction

Modeling distributed systems...

Distributed systems are becoming increasingly widespread. But methodologies for building distributed composite systems are lacking¹, specifications languages aimed at the modeling of such systems are seldom or have a weak expressiveness limiting their scope of application. There is a growing need for formal languages capable to manage the interactions between heterogeneous components (human, software, robot, ...) within a system, and capable to express various kinds of constraints (e.g. synchronization, response times, obligations, ...). Furthermore, those languages should offer analysis, validation and/or verification methods or tools.

...with Petri nets and temporal logic

Among the few candidates satisfying those requirements, two approaches are emerging. On the one hand, we have the Petri nets based languages for which numerous simulation and analysis methods have been developed. They provide an easy graphical representation of systems. On the other hand, temporal logic based languages probably offer a greater expressiveness, but they lack of "operational" tools. We think that it would be useful to

¹ The failure rate for building concurrent real-time systems is still estimated at 75% !

combine both approaches in order to take advantage at one and the same time of the great expressiveness of the temporal logic, and also of the analysis power and simple principles of Petri nets.

Overview

Part one : formalisms

The first part introduces the formalisms, the basic concepts and definitions that we will need all along this work. In that sense, it can be considered as the theoretical framework. We first present the Petri net theory and make a brief survey of the existing extended models inspired from the original Petri net model (chapter 1). We will see successively the predicate/transition nets, the colored Petri nets, the timed Petri nets and the documentary Petri nets. The second chapter recalls the main principles of temporal and real-time temporal logic (chapter 2). An example of specification language (ALBERT) for distributed systems is then presented. Next (chapter 3), we compare these two ways of modelling by analysing their respective capabilities for expressing various mechanisms (concurrency, synchronization, ...) that are inherent to such systems, and complex constraints like deontic aspects or real-time features. This will lead us to the definition of a semantics-preserving transformation of Petri nets into the ALBERT language. Finally, we introduce the formalism of a new branch of logic that deals with the modeling of permissions, prohibitions, etc... : the deontic logic (chapter 4).

Part two : a new integrated language

In the second part, we propose a new integrated language (PNRTL) combining the different advantages - or avoiding the disadvantages - of both approaches. Firstly (chapter 5), we investigate the possible ways of modeling deontic aspects in Petri nets, especially the obligations and prohibitions, and suggest to distinguish among the executions of the net, the ones that are ideal and sub-ideal. This will allow us to obtain a preference order on the set of executions. The two next chapters (chapters 6 and 7) really present the new specification language we propose. Roughly speaking, we attach to the Petri net model a set of temporal logic formulas (representing constraints) that restrict the set of allowable executions. It becomes now much easier to express, for instance, time or performance constraints.

Part three : application and tools

The last part of this work - part we have wanted to be more practical - illustrates the different ideas and suggestions previously exposed, by applying them to a case study (chapter 8), namely a library network. Thereafter (chapter 9), we present some techniques and tools to reduce the set of *possible* executions into the one of *desired* executions, that is to say the set whose all executions respect the additional constraints. Those techniques which are based on the automata and formal language theories, can also be used for the testing of invariants or properties but they only work for a specification without real-time features. Finally (chapter 10), we explain how proofs can be done by translating a net and the constraints that accompany it, into an axiomatic system allowing us to derive - and so to proof - invariants.

Chapter 1

The Petri net formalism

1.1 Introduction

Petri nets [PETERSON81] are a popular graphical formalism for the study, modeling and analysis of discrete dynamic systems. These nets are particularly appropriate for modeling distributed systems, because they can be used to represent parallelism and synchronization. They have been developed from the early work of Carl Adam Petri [PETRI62] who formulated the basis for a theory of communication between asynchronous components of a system in 1962. The use and study of Petri nets has spread widely in the last few years.

A Petri net is a mathematical representation of a system. The usual approach considers Petri nets as an auxiliary tool (see figure 1.1) : conventional design techniques are used to specify a system which is then modeled as a Petri net. This Petri net model can now be analyzed and leads to a better system (any problem encountered in the analysis forces to revize the system). This cycle is repeated until the analysis reveals no problem - or only acceptable ones. In the alternative - and more recent - approach, the entire design and specification process is carried out in terms of Petri nets. Then the problem is to transform the Petri net representation into an actual working system.

Many modeling tools or languages are inspired from the original Petri net theory. They add some extensions to the Petri nets in order to facilitate the modeling of concurrent and/or distributed systems. The best known extended Petri nets are the colored (section 1.3 and 1.4)

and the timed Petri nets (section 1.5). Finally, we will see a third extended model : the documentary Petri nets (section 1.6).

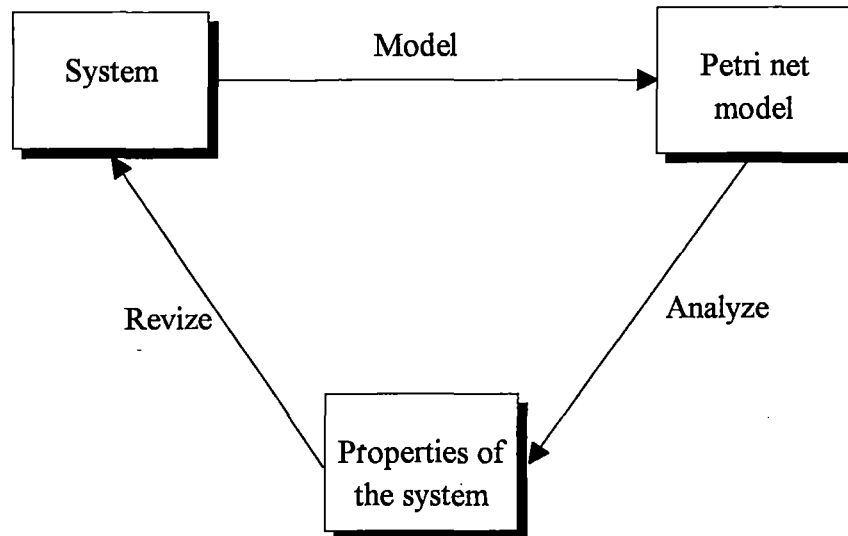


Figure 1.1 (the use of Petri nets)

1.2 Basic Petri nets

A Petri net is composed of two types of objects : the **places** and the **transitions**. The set of places represents the state of the modeled system; the set of transitions represents events, actions or phenomena that alter the state of the system. The places may contain several **tokens**. The presence of tokens in a place can be interpreted as the presence of a resource of a certain kind, or as the satisfaction of certain preconditions. To **fire** a transition (that is to perform the corresponding action), some preconditions have to be satisfied, i.e. some places must contain a specified number of tokens. Those places are called the **input places** of the transition. The **firing** of a transition has the effect that the **marking** of the net, that is to say the token distribution, is modified : the specified number of tokens are removed from its input places and, at the same time, tokens are added to some places. These are the **output places** of the transition. So, transitions consume and produce tokens. The dynamic behavior of the system is represented by the flowing of the tokens through the net.

Graphically, a Petri net is depicted as a directed graph (see figure 1.2) which consists of two disjunct sets of nodes : the places (represented as circles) and the transitions (represented as bars). Places and transitions are connected by arcs. It is not allowed to connect two places

or two transitions. The number of arcs between a transition and a place indicates how many tokens are requested or produced by the transition. A token is represented by a dot.

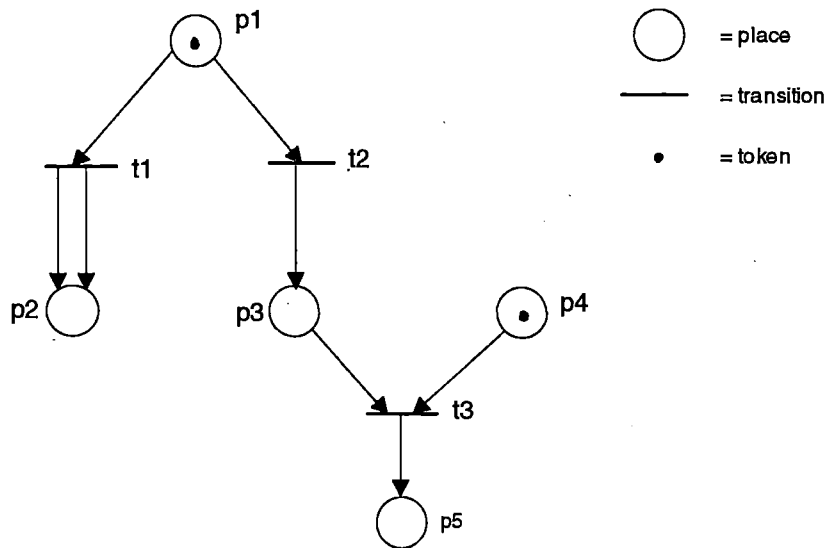


Figure 1.2 (example of a Petri net)

A transition is **enabled** if all its input places contain the specified number of tokens (i.e. as many tokens as arcs from the place to the transition). A transition may fire whenever it is enabled. Note that we say “may”, because when two transitions are enabled at the same marking, we have to choose - and this choice is **undeterministic** - to fire one of the two, and the second will not necessarily still be enabled in the marking which results from the firing of the first transition. For instance, in the net of figure 1.2, t_1 and t_2 are both enabled, but the firing of one transition will disable the other one, since the token in the place p_1 will be removed.

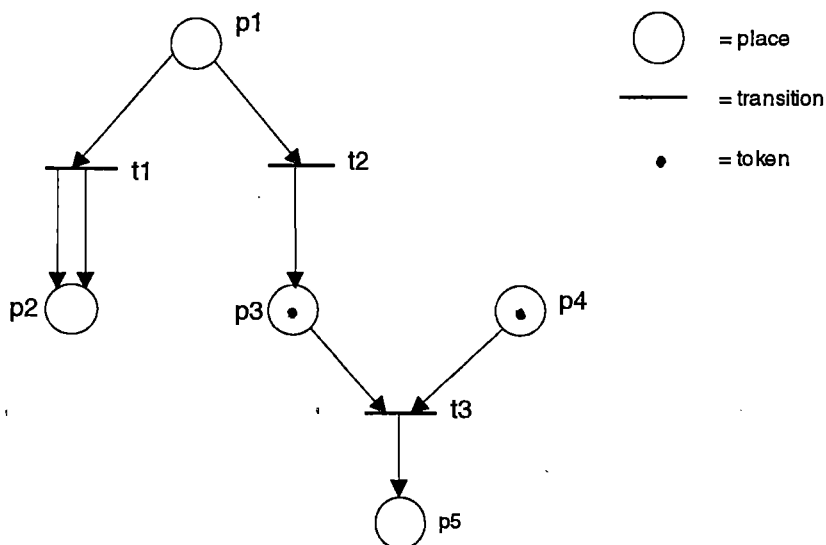


Figure 1.3 (the state of the net after the firing of t_2)

1.2.1 A few definitions

We now give the basic definitions of the Petri net formalism. See [BRAMS83a, PETERSON81] for more details.

Definition 1.1 (Petri net)

A Petri net structure is a four-tuple $N=(P, T, Pre, Post)$.

where $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places ($n \geq 0$)

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions ($m \geq 0$), with $P \cap T = \emptyset$

$Pre: P \times T \rightarrow \mathbb{N}$ is the input function, giving the number of tokens needed in a place to fire a transition

$Post: P \times T \rightarrow \mathbb{N}$ is the output function, giving the number of tokens produced in a place when firing a transition

Very often, the functions Pre and $Post$ are represented by a matrix $n \times m$ (see figure 1.4).

Definition 1.2 (Marking)

A marking M is an assignment of tokens to the places of a Petri net. A marking M is a function giving for each place the number of tokens it contains:

$$M: P \rightarrow \mathbb{N}; p \rightarrow M(p)$$

Very often, this function is represented by a vector of n elements (see figure 1.4).

We write (N, M) for a Petri net N with marking M .

Definition 1.3 (Enabled transition)

An enabled transition in a marking M is a transition whose all input places contain at least the required number of tokens:

$$t \in T \text{ is enabled} \Leftrightarrow \forall p_i \in P: M(p_i) \geq Pre(p_i, t)$$

Or in matrix notations :

$$t \in T \text{ is enabled} \Leftrightarrow M \geq Pre(*, t) \quad \text{where } Pre(*, t) \text{ is the column related to } t, \text{ in the matrix } Pre.$$

Definition 1.4 (Firing a transition)

In a marking M , only enabled transitions can fire. Firing an enabled transition t results in a new marking \bar{M} - which we write $M \xrightarrow{t} \bar{M}$ - defined¹ by :

¹ Firing a transition is thus deterministic.

$$\forall p_i \in P: \bar{M}(p_i) = M(p_i) - Pre(p_i, t) + Post(p_i, t)$$

This means that the vector \bar{M} resulting from the firing of t , is defined by the matricial equation:

$$\bar{M} = M - Pre(*, t) + Post(*, t)$$

where $Pre(*, t)$ is the column related to the transition t in the matrix Pre .

$Post(*, t)$ is the column related to the transition t in the matrix $Post$.

Definition 1.5 (Reachability set)

The reachability set R of a marked Petri net (N, M) is the smallest set of markings defined by:

1. $M \in R(N, M)$
2. $M' \in R(N, M) \wedge \exists t \in T: M' \xrightarrow{t} M'' \Rightarrow M'' \in R(N, M)$

The execution of a Petri net from an initial marking is a sequence of transitions. This firing sequence can be represented by the concatenation of symbols, each of them being a transition:

Definition 1.6 (Execution/Sequence of transitions)

Let's consider T as an alphabet composed of the symbols t . A sequence s of transitions is a word of T^* :

$$s = t_{i_1} t_{i_2} \dots t_{i_k} \quad (t_{i_j} \in T)$$

Definition 1.7 (Firing a sequence of transitions)

The firing of a sequence s of transitions in the marked net (N, M) leads to the marking \bar{M} - which we will write $M \xrightarrow{s} \bar{M}$ - if and only if²:

either $s = \lambda$ (empty sequence), then $\bar{M} = M$

either $s = s' t$ ($s' \in T^*$, $t \in T$), then $\exists M': M \xrightarrow{s'} M' \wedge M' \xrightarrow{t} \bar{M}$

The set of possible executions can be defined in terms of firing sequences:

Definition 1.8 (Set of possible executions from a marking)

The set E of possible executions of a marked Petri net (N, M) is defined as

$$E(N, M) = \{s \in T^*: M \xrightarrow{s} M' \wedge M' \in R(N, M)\}$$

² A fireable sequence of transitions also represents a path that leads from the marking M to the marking in the reachability graph of the Petri net.

Definition 1.9 (Set of possible executions between two markings)

The set E of possible executions of a marked Petri net (N, M) leading to the marking \bar{M} is defined as:

$$E(N, M, \bar{M}) = \{s \in T^* : M \xrightarrow{s} \bar{M}\}$$

Finally, we introduce the notion of boundedness of Petri nets :

Definition 1.10 (Bounded place)

A place of a marked Petri net (N, M) is bounded if the number of tokens it contains is bounded:

$$p \in P \text{ is } k\text{-bounded} \Leftrightarrow \forall M' \in R(N, M) : M'(p) \leq k$$

Definition 1.11 (Bounded Petri net)

A Petri net (N, M) is bounded if and only iff all the places are bounded :

$$(N, M) \text{ is } k\text{-bounded} \Leftrightarrow \forall p \in P, \forall M' \in R(N, M) : M'(p) \leq k$$

Example :

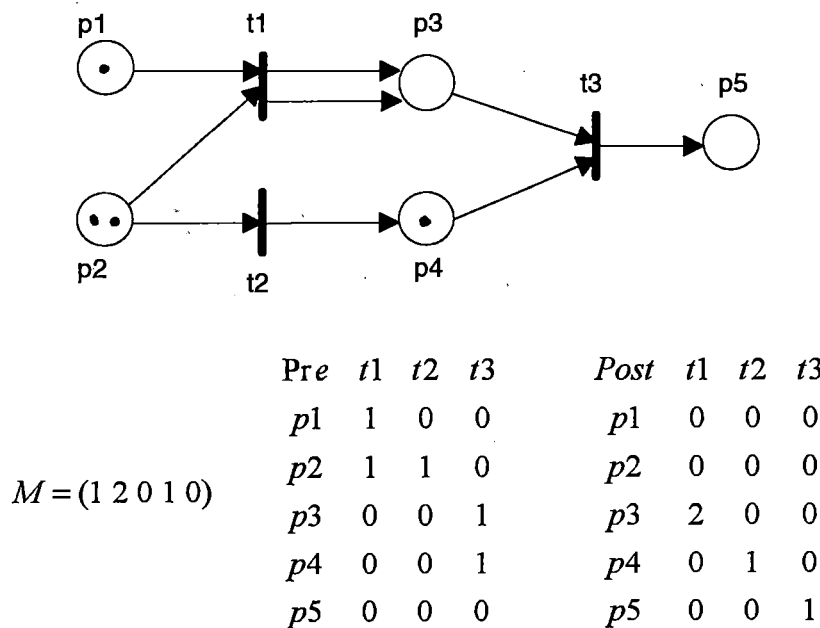


Figure 1.4 (another Petri net)

- In this net, t_1 and t_2 are enabled, because :

$$M=(1\ 2\ 0\ 1\ 0) \geq Pre(*,t_1)=(1\ 1\ 0\ 0\ 0)$$

$$M=(1\ 2\ 0\ 1\ 0) \geq Pre(*,t_2)=(0\ 1\ 0\ 0\ 0)$$
- The firing of t_1 results in the marking M_1
 where $M_1 = M - Pre(*,t_1) + Post(*,t_1)$

$$= (1\ 2\ 0\ 1\ 0) - (1\ 1\ 0\ 0\ 0) + (0\ 0\ 2\ 0\ 0)$$

$$= (0\ 1\ 2\ 1\ 0)$$
- The firing of t_2 results in the marking M_2
 where $M_2 = M - Pre(*,t_2) + Post(*,t_2)$

$$= (1\ 2\ 0\ 1\ 0) - (0\ 1\ 0\ 0\ 0) + (0\ 0\ 0\ 1\ 0)$$

$$= (1\ 1\ 0\ 2\ 0)$$
- The sequences $s_1=t_1t_2t_3$ and $s_2=t_2t_1t_3$ belong to $E(N,M)$, but $s_3=t_2t_3t_1$ does not.

1.2.2 Verification of properties of Petri nets

In order to ensure the correctness of the model (that is the Petri net) and maybe to revize it, we need to analyze this model and see if the wanted properties are satisfied (cfr figure 1.1). There are mainly three different but complementary approaches to analyze Petri nets :

- approach based on the coverability tree and graph
- approach based on the linear algebra
- approach based on the graph theory

Below, we give a brief survey of the first two approaches. See [FICHEFET88] for more about verification of properties.

A. The coverability tree and graph

Basically, the central idea of this technique is to build a tree of all the markings that are reachable from the initial marking, and then to analyze it. The root of the tree is thus the initial marking, while the sons of a node M are the markings that can be obtained by firing any

enabled transition at marking M . In other words, such a tree indicates what are the reachable markings and what are the possible executions of a Petri net (see figure 1.5).

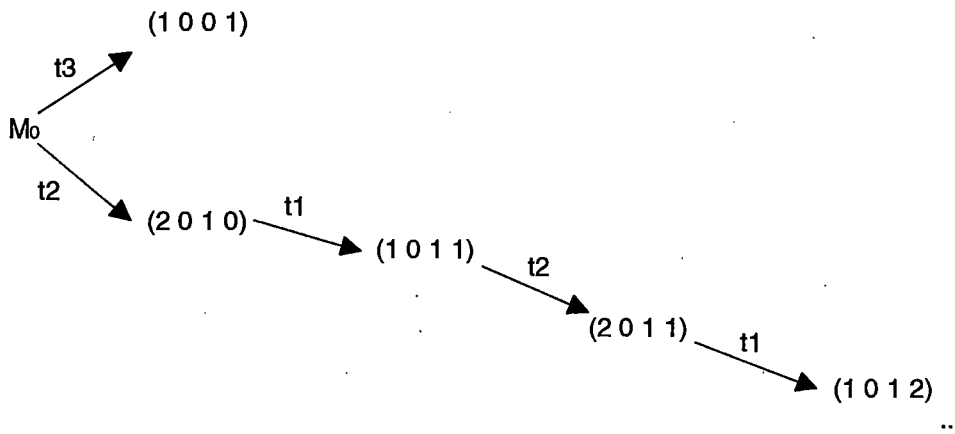


Figure 1.5 (a coverability tree)

The analysis of the coverability tree can provide some interesting indications, like the unboundedness of some places (if the marking of those places increases periodically) or the unboundedness of the net (if there is an infinite number of markings in the tree). To avoid the handling of infinite trees, one often prefers the coverability graph, in which the marking of some places is replaced by the symbol ' ω ' which denotes an arbitrary large number. For instance, the marking $M=(2\ 0\ 1\ \omega)$ stands for $(2\ 0\ 1\ 0)$, $(2\ 0\ 1\ 1)$, $(2\ 0\ 1\ 2)$, etc... Since it is no more a tree, loops or cycles are now permitted in the graph (see figure 1.6). It is clear that a ω associated with a place means that this place, and thereby the net, is not bounded.

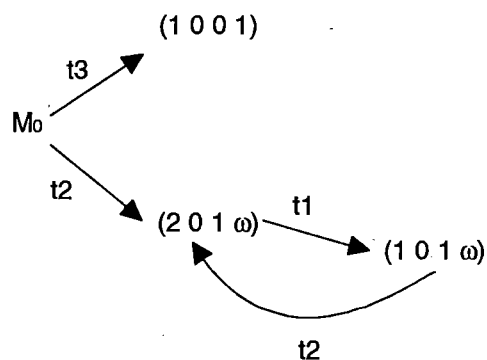


Figure 1.6 (the corresponding coverability graph)

Even if the coverability tree/graph allows us to verify some properties, it must be pointed out that when dealing with very large Petri nets, it is sometimes impossible to determine structural properties by simply examining the Petri net.

B. The linear algebra approach

Consider a Petri net $N=(P, T, Pre, Post)$ and a function $f: A \rightarrow \mathbb{N}: p \rightarrow f(p)$

Let C be the difference between the output and input matrix of N :

$$C = Post - Pre$$

Here are three theorems to verify the boundedness of places and/or of the net.

Theorem 1.1

A Petri net is bounded if there exists a positive function f such that $f^T \cdot C \leq 0$,

where f^T denotes the transposition of f :

$$N \text{ is bounded} \Leftrightarrow \exists f: (f > 0 \wedge f^T \cdot C \leq 0)$$

This theorem means that a possible way to verify the boundedness of a Petri net consists in solving a particular inequations system, such as for instance :

$$(f(p_1) \ f(p_2) \ f(p_3)) \cdot \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 1 & -1 \end{pmatrix} \leq 0 \text{ with } f > 0$$

$$\Leftrightarrow \begin{cases} f(p_1) > 0 \\ f(p_2) > 0 \\ f(p_3) > 0 \\ -f(p_2) - f(p_3) \leq 0 \\ f(p_2) + f(p_3) \leq 0 \\ -f(p_3) \leq 0 \end{cases}$$

Theorem 1.2

Let $\|f\| = \{p \in A : f(p) \neq 0\}$ and $\Delta(t, f) = f^T \cdot C(*, t)$. If $E = \{t \in T : \Delta(t, f) > 0\} = \emptyset$,

then - all places of $\|f\|$ are bounded (whatever the marking)

$$- \forall p \in \|f\|, \forall M \in R(N, M_0) : M(p) \leq \frac{f^T \cdot M_0}{f(p)}$$

Theorem 1.3

Let $F = \{f \in \mathfrak{N}^m : f^T \cdot C = 0\}$. If $f \in F$ then

- all places of $\|f\|$ are bounded (whatever the initial marking is)

$$- \text{if } p \in \|f\|, \text{ then the place } p \text{ is } k\text{-bounded with } k \leq \min_{f \in F} \left(\frac{f^T \cdot M_0}{f(p)} \right)$$

Illustrations of those theorems will be given in the second part of this work.

1.3 Predicate/transition nets

Predicate/transition nets [GENRICH86] can be defined as « *formal objects that can be interpreted and manipulated in a mathematical way that is comparable to working with logic formula or algebraic expressions* » (because their formalism is very close to the one of first-order predicate logic). Here, a dynamic system is viewed as a set of individuals that is structured by functions and relations. This structure is partially static and partially dynamic. The static part is the support of the dynamic system. The annotations of the net are interpreted in terms of a given static relational structure : the support. Operations (functions symbols) and predicates (relation symbols) form the vocabulary of the language in which the properties and relations of individuals will be described. The language used by Genrich is that of first-order predicate logic plus a class of simple algebraic expressions.

To illustrate this idea of simplifying the net representation of dynamic structures by merging conditions and events into transitions and places respectively, look at the following example (figure 1.7). It shows how a basic Petri net can be summarized :

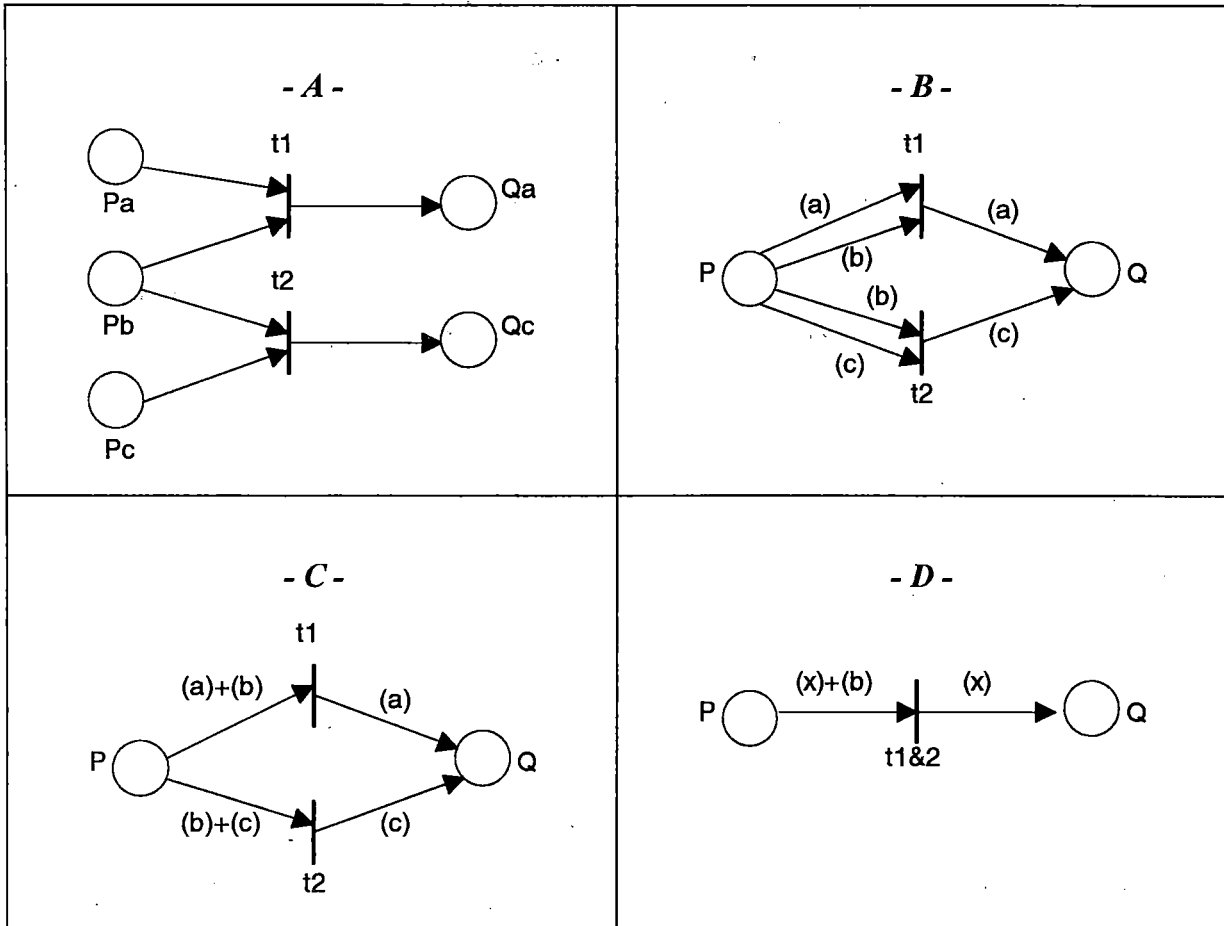


Figure 1.7 (from basic to predicate/transition nets)

Let us look at another example :

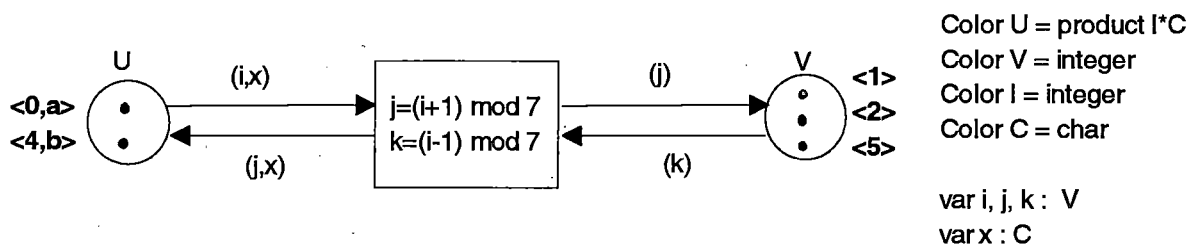


Figure 1.8 (another example)

The box in figure 1.8 denotes a set of events. The substitutions generating those events are determined by the relation existing between the individuals U and V. Here, this (static) relation is expressed by a first-order logical formula. For an event (a transition) to occur, the variables

x, i, j and k must be replaced by constants in such a way that the formula at the transition holds, and tuples generated by the substitution can be removed from/put on the corresponding places.

The notion of predicate/transition net will be explained in more depth in chapter 7.

1.4 Colored Petri nets

Colored Petri nets [JENSEN86, JENSEN90] take the main idea of predicate/transition nets, but this new net class is aimed at improving the method of invariant analysis. The main reason for which colored Petri nets have been developed is that « *they - without losing the possibility of formal analysis - allow the modeller to make much more succinct and manageable descriptions than can be produced by means of low level nets* ». This means that it facilitates the description of more complex systems. It is also possible to describe simple data manipulations. Such manipulations are implemented by expressions located on the arcs. In basic Petri nets, there is only one kind of token, and the state of a place can thus be represented by an integer³ denoting the marking of the place. In contrast, each token in a colored Petri net can carry complex information; each token is associated to a data structure. The data value attached to a given token is referred to the token color.

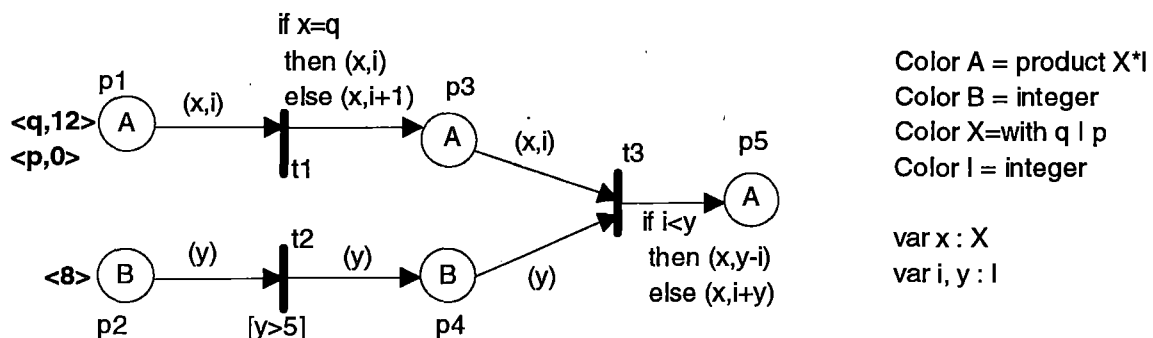


Figure 1.9 (a colored Petri net)

Let's analyze the net in figure 1.9. We can distinguish three different parts : the net structure, the declarations and the net inscriptions :

³ And even by a boolean in the case of binary (that is 1-bounded) places.

- The **net structure** is similar to the one of a basic Petri net. It is thus a directed graph with two types of nodes (places and transitions) interconnected by arcs.
- The **declarations** (in the right) tell us about the different color sets (A,B,X and I) and variables (x,i, and y). Each place is attached to a color set, and a token in that place must be an element of this set. These sets are defined in a particular syntax : “*product*” denotes the cartesian product of n sets, while “*with*” is the enumerated type constructor.
- A **net inscription** can be attached to a place, transition or arc. Places have three kinds of inscriptions : names, colour sets and initialisation expressions. For instance, the place named p2 is of color B and is initialized with a token whose value is 8. Transitions have two kinds of inscriptions : names and guards. A guard is a boolean expression which must be fulfilled⁴ for a transition to occur. For instance, the guard of the transition t2 is $y > 5$. Arcs have only one kind of inscriptions : the arc expressions. They may contain variables, constants, functions and operations (defined or implicit), and explain how the value of the output token(s) is derived from the one of the input token(s).

We now give a formal definition of colored Petri net, which can be found in [JENSEN90].

Definition 1.12 (Colored Petri net)

A colored Petri net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$

where Σ is a finite set of types, called color sets

P is a finite set of places

T is a finite set of transitions, with $P \cap T = \emptyset$

A is a finite set of arcs, with $A \cap T = A \cap P = \emptyset$

N is the node function mapping each arc into a pair (source-node, destination-node) :

$$N: A \rightarrow (T \times P) \cup (P \times T)$$

C is the colored function mapping each place into a color set :

$$C: P \rightarrow \Sigma$$

G is the guard function mapping each transition into a predicate, with

$$\forall t \in T: [Type(G(t)) = \text{boolean} \wedge Type(Var(G(t))) \subseteq \Sigma]$$

E is the arc expression function mapping each arc into an expression such that :

$$\forall a \in A: [Type(E(a)) = C(p(a)) \wedge Type(Var(E(a))) \subseteq \Sigma]$$

I is the initialisation function mapping each place into an expression such that :

$$\forall p \in P: [Type(I(p)) = C(p) \wedge Var(I(p)) = \emptyset]$$

⁴ By default, an empty guard is evaluated to true.

Notations :

$\text{Var}(t)$ denotes the set of variables of t ($t \in T$).

$E(x_1, x_2)$ denotes the expressions on the arc linking x_1 to x_2 ($x_i \in (P \cup T)$)

Next, we define a binding. Intuitively, a binding is a substitution that replaces each variable of t with a color. Each guard of t must be evaluated to true :

Definition 1.13 (Binding of a transition)

For a transition $t \in T$ with variables $\text{Var}(t) = \{v_1, v_2, \dots, v_n\}$, we define a binding type $BT(t)$ as follows: $BT(t) = \text{Type}(v_1) \times \text{Type}(v_2) \times \dots \times \text{Type}(v_n)$

The set of all bindings b of a transition t is defined as follows :

$$B(t) = \{(c_1, c_2, \dots, c_n) \in BT(t) \mid G(t) \langle v_1 = c_1, v_2 = c_2, \dots, v_n = c_n \rangle = \text{TRUE}\}$$

Definition 1.14 (Binding distribution)

A binding distribution is a function Y , defined on T , such that :

$$\forall t \in T: Y(t) \in B(t)$$

A step is a non-empty binding distribution.

Definition 1.15 (Enabled transition)

A step Y is enabled in a marking M if the following property is satisfied :

$$\forall p \in P: \sum_{(t,b) \in Y} E(p,t) \langle b_t \rangle \leq M(p)$$

When $(t,b) \in Y$, the transition t is enabled in M for the binding b .

For instance, in the net of figure 1.5, t_1 and t_2 are both enabled. But if the guard of t_2 was $y > 10$, then t_2 would not be enabled.

Definition 1.16 (Firing of a transition)

When a transition is enabled by b in a marking M , it may fire, changing the marking M into another marking \bar{M} defined by :

$$\forall p \in P: \bar{M}(p) = M(p) - \sum_{(t,b) \in Y} E(p,t) \langle b_t \rangle + \sum_{(t,b) \in Y} E(t,p) \langle b_t \rangle$$

where the first sum represents the removed tokens, whereas the second corresponds to the added tokens.

Example :

In the next figure, we show a possible state of the net⁵ after the firing of the sequence of transitions $s=t_1t_2t_3$ (we use the token $\langle p,0 \rangle$ in place p_1).

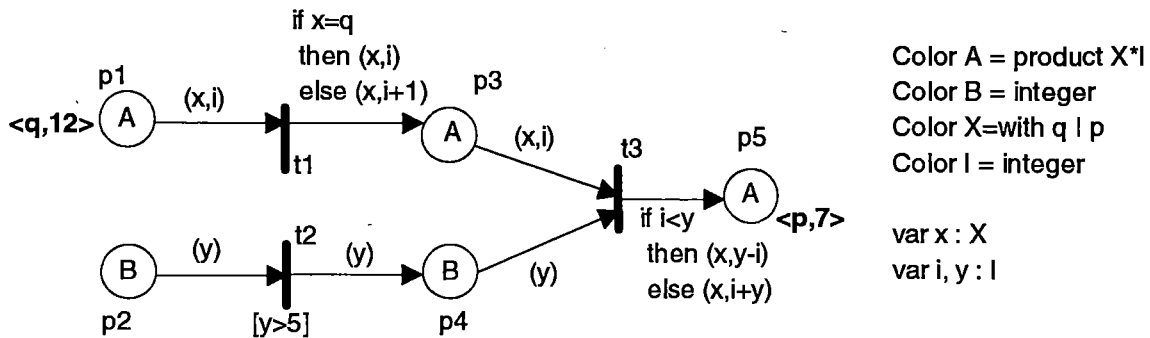


Figure 1.10 (the state of the net after firing of s)

Remark:

If its number of colors is finite, a colored Petri net can be unfolded into a regular Petri net [MURATA89] by unfolding each place p into a set of places - one for each color of tokens that p may hold, and by unfolding each transition into a set of transitions - one for each way that it may fire.

1.5 Timed Petri nets.

Time is an important aspect when modeling discrete dynamic systems. Time does not need to be quantified to reason about *qualitative* temporal properties (liveness, deadlock, fairness, ...) but it is then impossible to express *quantitative* temporal properties (deadlines, durations, response times, delays, etc...). The original Petri net model is not capable of handling quantitative time. However, in a coloured Petri net, one can use a special place, representing a global clock, connected to every transition and containing an unique token whose value represents the current time.

⁵ whose initial state is given in figure 1.9

The introduction of time has been proposed in several ways, depending on the location of the time delays and on the type of time delays [AALST92] :

- The **location** of the time delays. Some authors have attached a duration to each transition, i.e. the tokens are consumed and withheld for some time. In such models, the firing is said to be a two-phase firing. Other authors have proposed models in which each transition is associated with an enabling time : a transition must remain enabled for a specified time before it can fire.
- The **type** of time delays. A delay can be fixed, stochastic or specified by an interval. Time can also be discrete or continuous. Very often, fixed delays are inappropriate, simply because the duration of an activity depends on external factors. Stochastic delays are useful for the evaluation of performance, but their conditions of application are too restrictive : it is assumed that the delays of two activities are independent. Also delays should be allowed to depend on token values. Therefore, the solution of delays described by an upper and a lower bound seems to be the best and more realistic way to represent time delays. These bounds can be used to verify time constraints.

In this section, we expose a recent timed Petri net model proposed by Van der Aalst [AALST92] : the ITCPN (Interval Timed Coloured Petri Net) model. The main difference between ITCPN and the other timed models is that time is associated to the tokens (instead of the transitions) : every token bears a timestamp. This timestamp indicates the time the token becomes available. Concretely, a token in the ITCPN model has four attributes : an identity (i), a position (p), a value (v) and a timestamp (x). It is thus a four-tuple $\langle i, p, v, x \rangle$. As suggested by its name, a time interval is associated to each transition. For instance, the duration of t_1 in figure 1.11 vary from 1.5 to 3.0, whereas the one of t_2 is fixed (2.0).

Like in basic Petri nets, a transition is said to be enabled if there are “enough” tokens in each input place. An **enabled transition** can fire when all the input tokens are available; in other words it can fire at time x if all the tokens to be consumed have a timestamp greater than x . And so, the **enabling time** of a transition is the maximum timestamp of its input tokens. For instance, the enabling time of t_1 in figure 1.9 is 3.0. Transitions with the smallest enabling time will fire first, but when two transitions (or more) have the same enabling time, any of them may fire first. Note that the firing is still an atomic action. The difference between the firing time and the timestamp of the produced tokens is called the **firing delay**. So, the timestamp of an output token corresponds to the enabling time of the fired transition increased by a variable but

bounded delay. The effect of the firing of t_1 is given in figure 1.12, assuming that the firing delay of t_1 is 2.0.

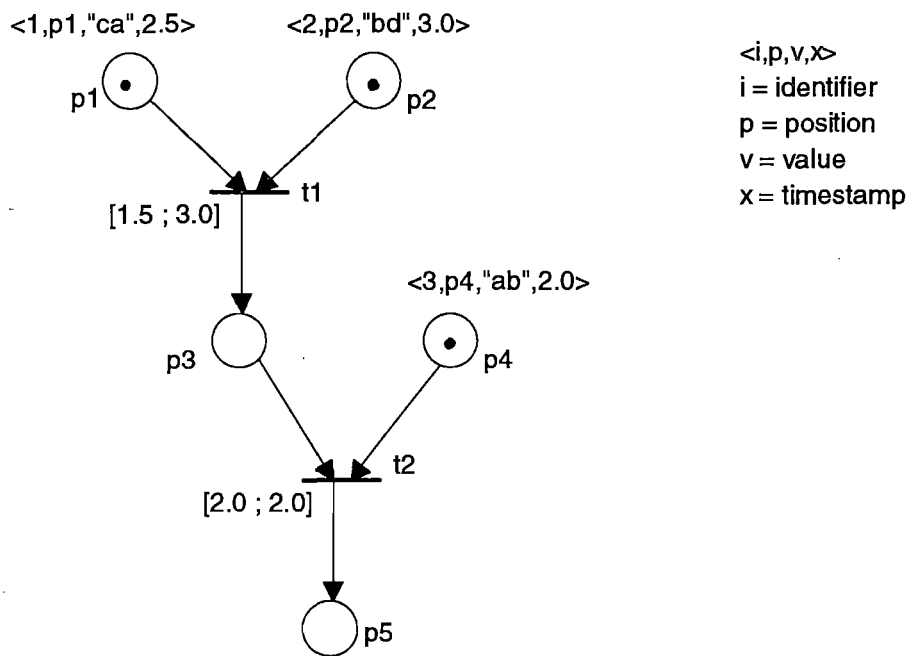


Figure 1.11 (example of net in the ITCPN model)

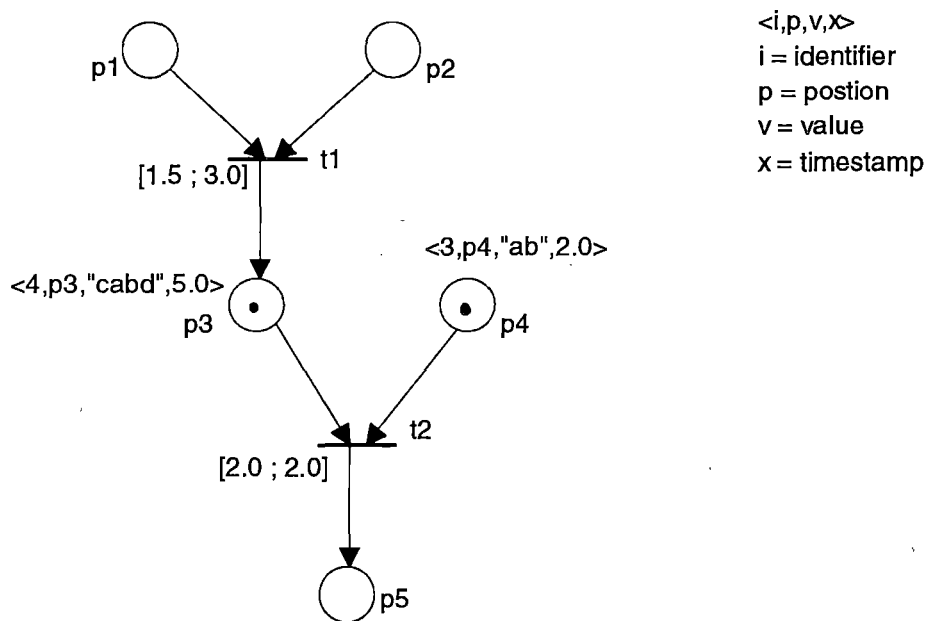


Figure 1.12 (the effect of the firing of t_2)

Finally, we give the definition of a net in the ITCPN model :

Definition 1.17 (ITCPNet)

An ITCPN is a seven-tuple (P, T, V, I, O, TS, F)

where P is a finite set of places

T is a finite set of transitions

V is the value/color function :

$$V: p \rightarrow V_p \quad (V_p \text{ being the value/color set of } p)$$

$I: T \rightarrow \text{Bag}(p)$ gives the input places of t

$O: T \rightarrow \text{Bag}(p)$ gives the output places of t

TS is the time set

INT is the set of all possible closed intervals :

$$INT = \{[a, b] \in TS \times TS : a \leq b \wedge b < \infty\}$$

CT is the set of all possible coloured tokens :

$$CT = \{ \langle p, v \rangle : p \in P \wedge v \in V_p \}$$

F is the transition function such that :

1. $\forall t \in T: \text{Dom}(F_t) = \left\{ c \in \text{Bag}(CT) : \forall p \in P: \sum_{v \in V_p} c(\langle p, v \rangle) = I_t(p) \right\}$
2. $\forall \langle p, v \rangle, x \in F_t(c): p \in O_t \wedge x \in INT$

1.6 Documentary Petri nets

We saw in the introduction (see section 1.1) that Petri nets are well-suited for the modeling of distributed systems because of their ability to represent concurrency and synchronization. As a consequence of the growing development of the *Electronic Data Interchange* (E.D.I.) and of the network infrastructure, more and more systems (e.g. electronic business and electronic contracting) include agents, components or entities that can be geographically very remote. This means that many distributed systems nowadays ensure the synchronization of their sub-processes by exchanging electronic documents or messages. Therefore, a new kind of Petri nets, the documentary Petri nets (DPN) have been recently proposed (see [BLWW95]), especially to support the design of trade procedures. CASE/EDI is a tool based on the DPN formalism, and has been applied to some international trade procedures such as exchange of bills of lading for letters of credit, custom clearances, etc... (see [LEE91, LEE92])

DPN's introduce a new kind of place, called **document place**, which have a specific graphical representation (see figure 1.13). Each document place is attached to a **document structure** described in a pseudo-language. The link between a document place and its structure is ensured by its label which also includes the name of the receiver (*to X*) in the case of an output document place, or the name of the sender (*from X*) in the case of an input document place.

Every token contained in a document place represents an instance of a document having a particular structure. Clearly, a DPN is a coloured Petri net where the color corresponds to the information that a document holds. But in contrast to colored Petri nets, a token can be used whatever it "contains" (that is whatever the value of the document components are) : there is no predicate to verify on the arcs, and no condition/guard associated to the transitions. Here, the fact that a message or document has the wanted structure is sufficient to use it, because CASE/EDI models **bureaucratic procedures**.

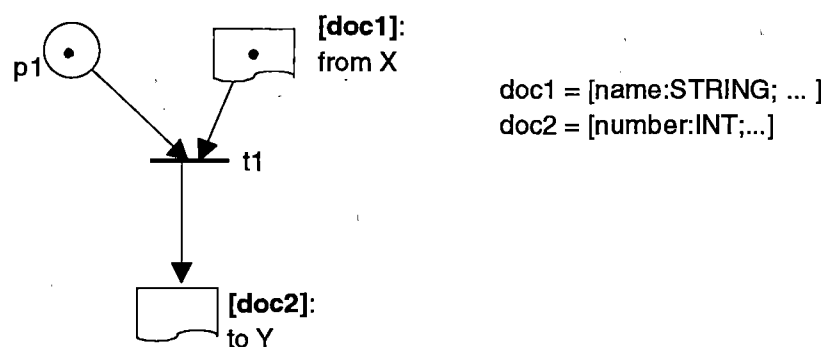


Figure 1.13 (two types of places)

In CASE/EDI, we model trade **scenarios** involving business participants. **Roles** are specifications of their allowable behaviour and are modeled by DPN's (one per role). An example of scenario is given in figure 1.14. Note that the notion of scenario denotes the **instance** of a (commercial) transaction.

The modeling of bureaucratic procedures is quite different from the modeling of logistic processes (like in the ITCPN model), because in bureaucratic procedures, deontic aspects such as obligations, prohibitions and permissions play an important role (see chapter 3 for more about deontic aspects). Performative communication in DPN's alters the state of commitment between the parties (the documents in a DPN are viewed as speech acts inducing obligations, permissions, etc...). So, if you accept or send a document (i.e. by using the token in the corresponding document places of a transition), you are supposed to accept at the same time

the obligations that are implicitly attached to this document. For instance, in the scenario of figure 1.14, the sending to the seller of the acceptance induces the obligation to pay for the buyer. Similarly, the sending of price quotations supposes that an acceptance from the buyer will result in the delivery of the goods by the seller.

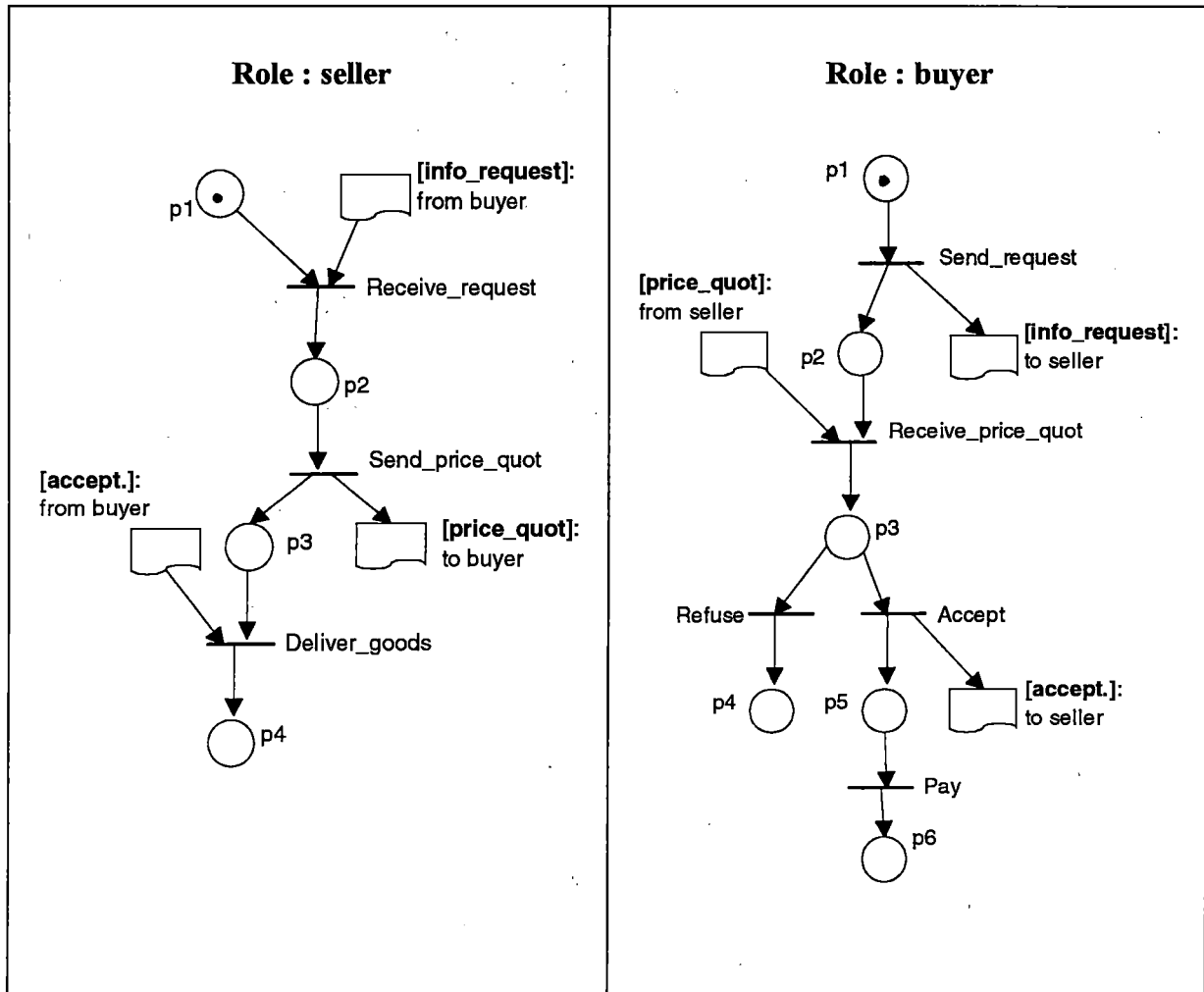


Figure 1.14 (a scenario in CASE/EDI)

Let us now (try to) give a formal definition of the concepts of CASE/EDI :

Definition 1.18 (Scenario)

A scenario is a 2-tuple (RS, RL)

where RS is a set of role specifications (i.e. of marked DPN) :

$$RS = \{(DPN_1, M_1), (DPN_2, M_2), \dots, (DPN_k, M_k)\}$$

RL is a set of role labels (e.g. seller, buyer, ...) :

$$RL = \{r_1, r_2, \dots, r_k\}$$

Elements of RS are marked documentary Petri nets, which in fact constitute a particular form of colored Petri nets :

Definition 1.19 (Role/DPN)

A role (a documentary Petri net) DPN_i is a 6-tuple $(\Sigma_i, P_i, T_i, Pre_i, Post_i, S_i)$

where Σ_i is a set of document types

P_i is a set of places

$$P_i = \bar{P}_i \cup \hat{P}_i \text{ (with } \bar{P}_i \cap \hat{P}_i = \emptyset)$$

\bar{P}_i is the sub-set of document places and \hat{P}_i the one of ordinary places

Pre and $Post$ are respectively the input and output function

S_i is a function mapping each document place into a document structure/type and a label indicating the source or destination role :

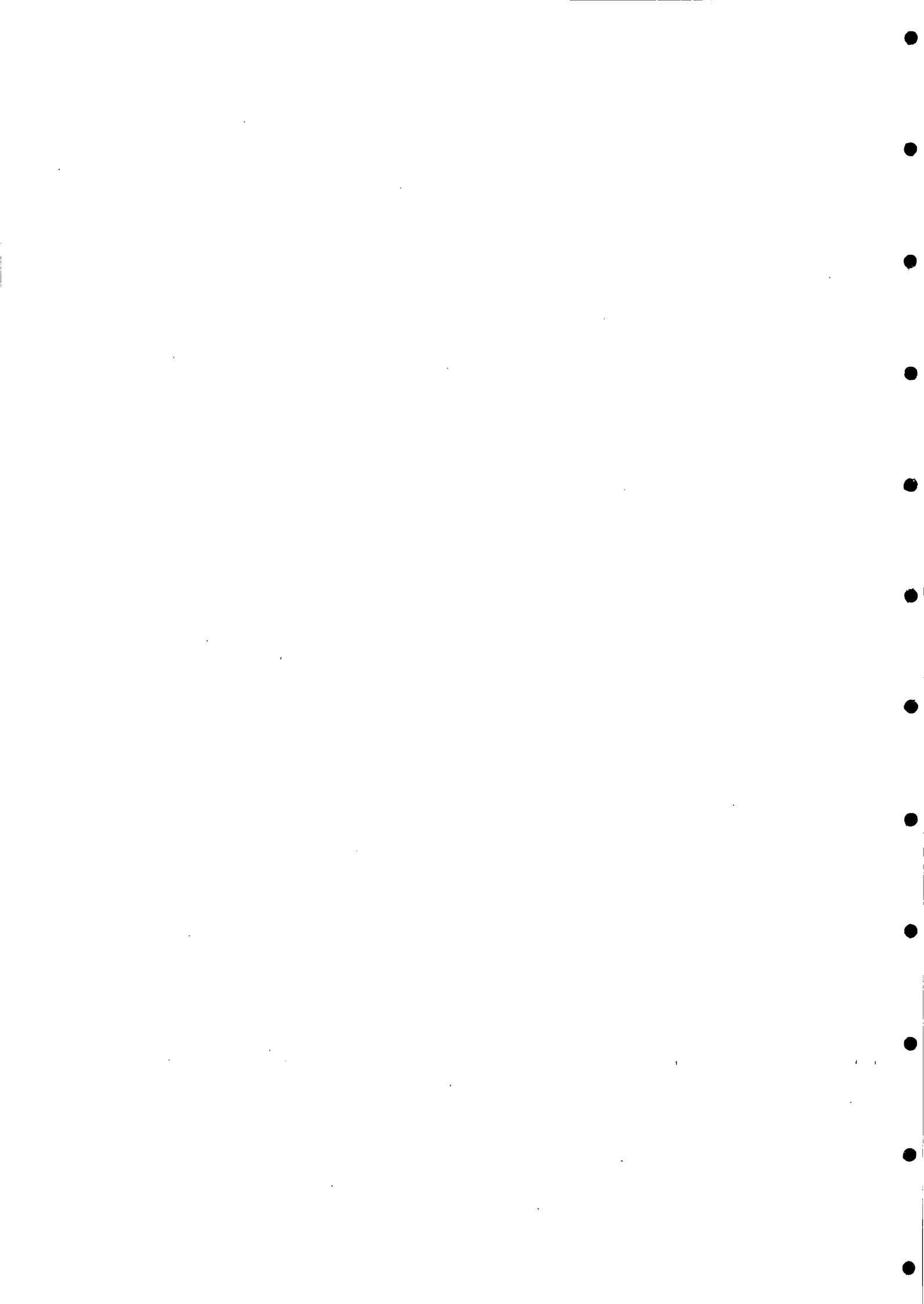
$$S_i: P_i \rightarrow \Sigma_i \times RL$$

Definition 1.20 (Marked DPN)

A marked DPN is a DPN at marking M with an additionnal constraint : all places of the net are 2-bounded places.

Note that an input document place in a role i must correspond to a similar output place in the role j (j being the source role labelling the input place), and vice-versa :

$$\forall p \in \bar{P}_i: (\exists t \in T_i: Pre_i(p, t) = 1) \Leftrightarrow ((p \in \bar{P}_j) \wedge (\exists t \in T_j: Post_j(p, t) = 1))$$



Chapter 2

The Temporal Logic formalism

2.1 Introduction

In this chapter, we introduce the temporal logic formalism. Temporal logic allows to reason over situations changing in time. Thus, this formalism is well suited for the description of distributed system.

Section 2.2 introduces the basic definitions of temporal logic : its semantics and the temporal operators. Section 2.3 briefly introduces the notion of real-time. Finally, section 2.4 presents a temporal based specification language : ALBERT [DDDP94a], [DDDP94b].

2.2 Temporal Logic

2.2.1 Modal Logic

Temporal logic is a special kind of modal logic. In this subsection, we introduce the basic concepts of a propositional modal language.

Definition 2.1 (Propositional language [AHO&ULLMAN93])

A propositional language contains a set of propositional letters P ($\dots, p, p_1, p_2, \dots, q, q_1, q_2, \dots$), two propositional constants T (true) and F (false), boolean connectives : \neg (not), \wedge (and), \vee (or), \rightarrow (if ... then ...), \leftrightarrow (if and only if), $I:P \rightarrow \{True, False\}$ an interpretation function that

assigns to each propositional letter the boolean value 'True' or 'False'. As a basis for the propositional language we take $\{\neg, \vee, \text{False}\}^1$:

- $\alpha \vee \beta$ is true iff α is true or β is true
- $\neg\alpha$ is true iff α is false

other operators are defined as follows :

- $\text{True} := \neg\text{False}$
- $\alpha \wedge \beta := \neg(\neg\alpha \vee \neg\beta)$
- $\alpha \rightarrow \beta := \neg\alpha \vee \beta$
- $\alpha \leftrightarrow \beta := \neg(\neg\alpha \vee \neg\beta) \vee \neg(\alpha \vee \beta)$

To the usual propositional language for modal logic [CHELLAS80], we add two unary operators : L for necessarily and M for possibly. These two operators are dual :

$$L(\varphi) \equiv \neg M(\neg\varphi)$$

The semantics of modal logic is based on frames and models. Let us define these two notions formally :

Definition 2.2 (Modal frame).

A modal frame is a 2-tuple $\langle W, R \rangle$ where W is a non-empty set of worlds and R a binary relation defined on $W \times W$: this relation is called the accessibility relation.

Definition 2.3 (Modal model).

A modal model is a 3-tuple $\langle W, R, V \rangle$ where $\langle W, R \rangle$ is a modal frame and V is a valuation on W , V maps proposition symbols to subsets of W (giving the set of worlds where this proposition holds).

Let us now define the truth value of a modal formula :

Definition 2.4 (Truth value of a modal formula).

The fact that a modal formula φ holds in a modal model $\Omega = (W, R, V)$ at world $w \in W$, noted $\Omega, w \models \varphi$, is defined by recursion on formulae :

1. $\Omega, w \models p$ iff $w \in V(p)$
2. $\Omega, w \models \neg p$ iff $w \notin V(p)$
3. $\Omega, w \models p \vee q$ iff $w \in V(p)$ or $w \in V(q)$
4. $\Omega, w \models L(\varphi)$ iff $\forall w' \in W [wRw' \rightarrow \Omega, w' \models \varphi]$
5. $\Omega, w \models M(\varphi)$ iff $\exists w' \in W [wRw' \rightarrow \Omega, w' \models \varphi]$

¹ In the following, α and β denotes propositional formula

2.2.2 Linear Temporal Frame²

Temporal logic [MACARTHUR76] is a special kind of modal logic where some restrictions on the 2-tuple $\langle W, R \rangle$ (the frame), are respected.

Definition 2.5 (Temporal frame).

A temporal frame is a 2-tuple $\langle \mathbb{N}_0, < \rangle$ where \mathbb{N}_0 is the set of positive integers and $<$ is the usual ordering relation over the positive integers. Each world, also often called state in temporal logic, is mapped to a positive integer number.

New modal operators, called temporal operators, are also introduced in temporal logic. Those operators are :

- for the future : X (Tomorrow, Next state), F (Eventually), G (Henceforth) and a binary operator U (Until)
- for the past : Y (Yesterday, previous state), P (Sometime in the past), H (Always in the past) and a binary operator S (Since).

We can give the semantics of these temporal operators :

1. $\Omega, i \models X(p)$ iff $\Omega, i+1 \models p$
2. $\Omega, i \models F(\varphi)$ iff $\exists j (i \leq j \rightarrow \Omega, j \models \varphi)$
3. $\Omega, i \models G(\varphi)$ iff $\forall j (i \leq j \rightarrow \Omega, j \models \varphi)$
4. $\Omega, i \models (\varphi)U(\psi)$ iff $\exists j (i \leq j \rightarrow \Omega, j \models \psi \wedge \forall k (i \leq k < j \rightarrow \Omega, k \models \varphi)$
5. if $i=1$ then $\Omega, i \models Y(\varphi)$ is always true
6. if $i>1$ then $\Omega, i \models Y(\varphi)$ iff $\Omega, i-1 \models \varphi$
7. $\Omega, i \models P(\varphi)$ iff $\exists j (j \leq i \rightarrow \Omega, j \models \varphi)$
8. $\Omega, i \models H(\varphi)$ iff $\forall j (j \leq i \rightarrow \Omega, j \models \varphi)$
9. $\Omega, i \models (\varphi)S(\psi)$ iff $\exists j (j \leq i \rightarrow \Omega, j \models \psi \wedge \forall k (j < k \leq i \rightarrow \Omega, k \models \varphi)$

2.2.3 Temporal Logic in Computer Science

Since the seminal paper of Pnueli 'The temporal logic of programs' [PNU77] appeared in 1977, the use of temporal logic to reason over programs and computer systems has been steadily increasing [KOYMANS92].

² The branching temporal logic will not be introduced here. For an introduction to branching linear logic, see [STI87].

Temporal logic is well suited for reasoning about situations changing in time. A computation can be seen as a sequence of states where each transition from one state to the next state can be thought as a tick of some computation clock. The behavior of a system can be viewed as the set of its possible computations. Therefore temporal logic can be used to describe the desired behavior of a system.

But concurrent systems, time-critical systems or distributed systems are also often characterized by quantitative timing properties. Therefore usual temporal logic is not sufficient since only qualitative time constraints, i.e. constraints on the ordering of states, can be specified. To overcome this drawback, the notion of distance in time between states must be introduced. Introducing this notion, we obtain a formalism called real-time temporal logic.

2.3 Real-Time Temporal logic

In real-time temporal logic, the notion of sequence of states is preserved. Besides that, each state is mapped to a point of the real-time. So it is now possible to compute a distance (in time) between two states and to express assertions on the distances.

The temporal operators are adapted syntactically and semantically. The following assertion

$$p \rightarrow F_{\leq 5min}(q)$$

expresses that if p is true in the state where the assertion is evaluated, then there must exist in the future a state distant of at most 5 minutes where q is true. In the sequel of this work, we investigate more formally the notions attached to real-time (see chapter 7 and chapter 10).

As an illustration of the expressive power of real-time temporal logic, let us consider the following constraints [KOYMANS92], which can be formulated in a distributed system specification, and their expression in real-time temporal logic :

- Maximal distance between an event and its reaction, for example, every A is followed by a B within 5 seconds (a typical promptness requirement) :

$$A \rightarrow F_{\leq 5sec}(B)$$

- Exact distance between events, for example, every A is followed by a B in exactly 2 seconds (as with the setting of a timer and its time-out) :

$$A \rightarrow F_{=2sec}(B)$$

- Minimal distance between events, for example, two consecutive A's are at least 3 seconds apart (assumption about the rate of input from the environment) :

$$A \rightarrow X(\neg F_{\leq 3sec.}(A))$$

- Periodicity, for example, event E occurs regularly with a period of 4 seconds :

$$E \rightarrow X(F_{=4sec.}(E) \wedge G_{<4sec.}(\neg E))$$

- Bounded response time, for example, there is a maximal number of time units so that each occurrence of an event E is responded to, by B, within this bound :

$$\exists t(E \rightarrow F_{\leq t}(B))^3$$

Remark. In chapter 1, we have presented the Petri net formalism. The figure 1.1 (page 1.2) shows that the Petri net formalism is used as a modeling language. On the other hand, temporal logic is a specification language. The style of Petri nets is operative while the one of temporal logic is declarative. Temporal logic based formalisms are usually used in an early stage of software engineering : the definition of the requirements of the system.

2.4 A Temporal Logic based language : ALBERT

2.4.1 The ALBERT language

ALBERT [DDDP94a], [DDDP94b] is a formal agent-oriented language for Requirements Engineering (RE). It is aimed at the specification of requirements for composite real-time systems. RE is the activity of obtaining from customers the initial requirements of a system to be implemented. Agent-oriented means that the language meets the main OO principles (namely encapsulation of data structures and actions on them). The word agent denotes a component or object having responsibilities and perceptions within the system. ALBERT is formal : it has a formal semantics giving a precise meaning to all specifications written in this language.

In the following subsections, we give an overview of this language.

³ The possibility of quantification over time is usually not permitted in real-time languages for purpose of completeness, but the formalism of Koymans [KOYMAN92] includes this possibility.

2.4.2 Main concepts

System

A system is a certain part of the world which is relevant to the customers. The word "system" is considered in a large sense, it can include software pieces, devices, humans, ... Goals are attached to the system.

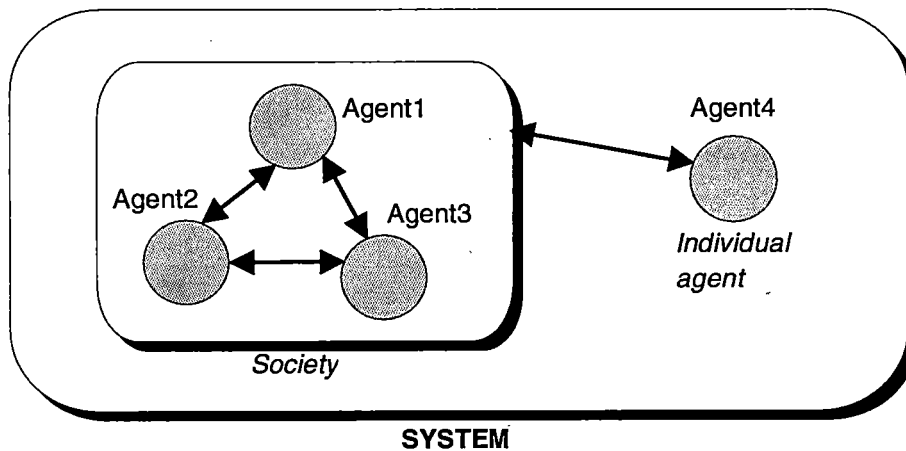


Figure 2.1 (An ALBERT system)

Agents

The agents are the basic blocks of the language, the units of specification. They are entities of different nature that have to communicate and cooperate together in order to achieve the systems goals. Therefore, we associate to each of them, a set of possible lives. Compound agents, called societies, are made of finer ones. The system is thus considered as an agent society. Individual agents are agents whose behavior is formally defined. Each agent has a structure (made of state components and action) and can lead certain lives represented as sequences of states (bearing a timestamp) and changes :

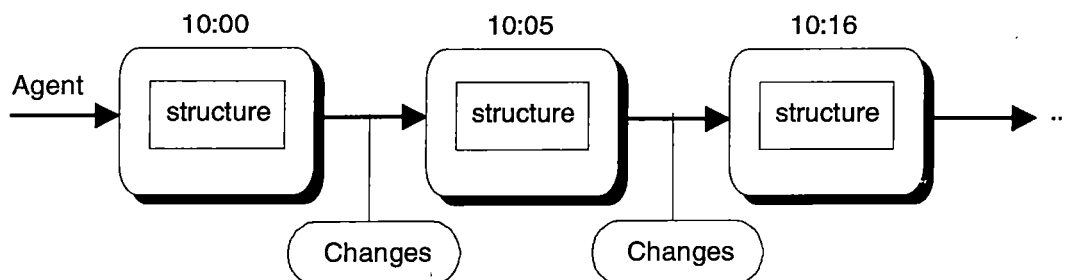


Figure 2.2 (A ALBERT agent life is a sequence of states)

Specifications

The system specification is obtained by combining the different agent specifications and the specification of the data and operations. In an individual agent specification, two parts are to be distinguished: (i) declarations describing its structure (that is its state components and its actions) and (ii) the constraints identifying possible lives and unwanted ones.

2.4.3 Agent declaration

State components

The state of an agent records its knowledge, each component is part of this knowledge and represents an agent attribute or information it handles. Each component has a type corresponding to the data type of its value. Those types can be predefined (like strings, integers, ...) or user-defined (like sets, sequences, tables, ...). State components declarations express visibility properties about state components: components outside the parallelogram belong to the state of other agents and are imported from them; inside the parallelogram, component with an outgoing arrow are exported to the indicated agent class; other components are private.

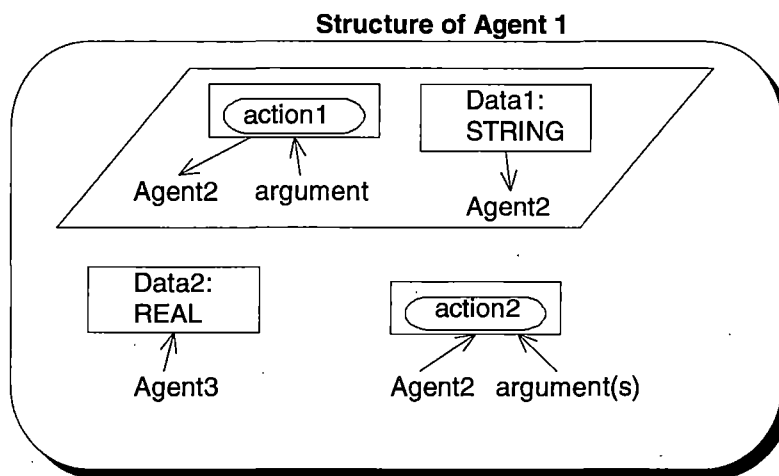


Figure 2.3 (An ALBERT agent structure)

Actions

Actions are phenomena or events which happen in the life of an agent and which change the value of the state components. Actions can be decided by an agent but it can also undergo actions. Actions may have arguments. The values of these arguments make intrinsically part of the occurrence of an action. Like for the state components, actions inside the parallelogram are under the control of the agent, while actions outside are performed by other agents. Arrows indicate which agent is importing/exporting the action.

The introduction of actions makes possible to overcome the well-know frame problem [BMR92], a typical problem resulting from the use of declarative specification languages.

2.4.4 ALBERT phrases

This section describes the different kinds of phrases that compose the ALBERT constraints.

Terms

ALBERT terms refer to state components. A term can be:

- a constant symbol (ex: TRUE, UNDEF)
- a variable identifier (ex: x, c)
- a state component reference (ex: Seller.name)
- an operation application (ex: remove(list, e))
- a term between brackets.

Logical expressions

Logical expressions (or formulae) are used to express assertions or conditions, like "Card(stored-cars)<Capacity" which means that the number of stored cars must always be smaller than the capacity. Simple logical expressions (or atomic formulae) are terms that yield a boolean value. Composed expressions use the logical connectives (like \wedge , \neg , \vee , ...) and quantifiers (\forall , \exists).

Temporal logical expressions

These are variants of logical expressions using special temporal connectives (to reason on the agent life). It is possible to refer to durations. In classical temporal logic, we can refer to a moment in the past; in ALBERT, we can also refer, for instance, to the three last hours which have passed. For example, " $\blacklozenge_{<5'} \beta$ " holds if β was true (at least once) in the last five minutes. Here are the main temporal connectives (that may be subscripted with a time period) :

- ◆ : sometimes in the past (including the present), this operator is the equivalent of P in usual temporal logic (see subsection 2.2.2).
- : always in the past (including the present), equivalent of H operator.
- ◇ : sometimes in the future (including the present), equivalent of F operator.
- : always in the future (including the present), equivalent of G operator.

2.4.5 Constraints

There are three kinds of constraints in ALBERT: basic constraints, local constraints and cooperation constraints.

Basic constraints

They are used to describe the initial state of an agent and to give derivation rules (when the value of a component depends on other ones).

Local constraints

They are related to the internal behavior of an agent.

State behavior

This kind of constraints must be satisfied by states. Static constraints are properties which must be true in all states, they are invariants with simple logical expressions. Dynamic constraints govern the evolution of the state along the agent life. These constraints are temporal logical expressions.

Effect of action

Here, we define how state components values change according to the occurrence of (internal or external) actions. The post-action state is specified by application of mathematical functions on the initial state. The effects of actions are valuations and not logical expressions.

Capability

Capability constraints are used to describe the responsibility of an agent with respect to its own actions, in terms of preventions (circumstances under which actions cannot occur) and obligations (circumstances under which actions must occur). They are expressed with the help of special connectives: *F* for forbidden, *O* for obligation and *XO* for exclusive obligation).

Action composition

ALBERT provides a set of connectors to express action sequences (;), *n* occurrences of *a* at the same time $\{a\}^n$, simultaneity (\otimes), parallelism (\parallel), or alternatives (\oplus).

Action duration

Under this heading, the specifier may put constraints on the length of internal action occurrences. The constraint may be an exact duration ($|<action>| = <duration>$), or, a lower ($|<action>| > <duration>$) or upper bound ($|<action>| < <duration>$) for the duration.

Cooperation constraints

In contrast with the local constraints (related to the internal behavior of an agent), cooperation constraints govern the information exchanges between the agent and the outside.

Action perception

Here, we describe how the agent is sensitive to actions occurring outside. This is done in terms of ignorance (*I*) and knowledge (*K*) and exclusive knowledge (*XK*).

State perception

Under this heading, we define how agents see parts of the state of others agents belonging to the same society. State perceptions are also specified using *K*, *I* and *XK*.

Action information

Information is the dual of perception and expresses how the agent let others about agents know actions it performs. Again, the ignorance and knowledge connectives are used.

State information

Describe how an agent shows parts of its state to others agents belonging to the same society (again with *I*, *K* and *XK*).

Futher readings. We refer the interested reader to [DUBOIS95a], [DUBOIS95b] and [DDZ95].

Chapter 3

Temporal logic versus Petri nets

3.1 Introduction

In the last few years, many languages have been proposed for the purpose of specifying or modeling distributed systems. We can divide such languages in two categories. On the one hand, we have the mathematics based languages, like Petri net based languages. They have progressively shifted from semi formal notations (box and arrows) to more formal notations (predicates). They offer many formal and informal analysis methods (namely liveness, boundedness, pattern recognition). On the other hand, some more recent languages are based on temporal logical grounds. Their formal semantics associated to the existence of rigorous rules of interpretation and of deductive inference provide to the analyst an interesting support in his/her modeling task.

Before proposing (in chapter 6) a new language combining these two different ways of modeling distributed systems, we first compare both approaches and outline some of their respective features (section 3.2), and then show that it is possible to transform a basic Petri net into a specification in ALBERT for the purpose of reverse engineering (section 3.3).

3.2 Comparison of both approaches

To perform this comparison, we are going to study two particular languages. The first one, ALBERT (see section 2.3) is based on the temporal logic formalism, while the second, CASE/EDI (see section 1.6) is a colored Petri net based language for the modeling of formal commitment procedures. In this section, we are particularly interested in what (i.e. which constraints) we can and cannot express, and in how it can be expressed.

3.2.1 Concurrency, choice and synchronization

As we study the modeling of distributed systems, we will emphasize on three fundamental mechanisms that are inherent to such systems : the concurrency, the choice and the synchronization.

Concurrency

It means that simultaneous events can occur. The rule in CASE/EDI is that all the transitions whose input places contain a token¹ *may* fire. Thus, when we want to allow transitions to occur simultaneously, we have to build a net, and thereafter to verify by a mathematical analysis (e.g. coverability graph) that this simultaneity is possible. Although the Petri nets are especially suitable for modeling concurrency, we must be careful about the concurrency possibilities of a Petri net, because it is not straightforward when examining its graphical representation, to see which actions will be allowed to occur at the same time. Look at figure 3.1. In this net, it is not always possible to fire *Action1* and *Action2* simultaneously (it depends on the history of the fired transitions).

In contrast, in ALBERT, the concurrency is *implicitly* permitted in ALBERT : actions *may* always occur at the same time except if it is *explicitly* forbidden (F) by a constraint (1), or if they alter the same information (2) :

- | | |
|--|-------------------------|
| (1) $F(\text{MacroAction} / \text{TRUE})$ | % capability constraint |
| $\text{MacroAction} \leftrightarrow (\text{Action2} \otimes \text{Action3})$ | % action composition |
| (2) $\text{Action1}: n=n+1$ | % effects of action |
| $\text{Action2}: n=n-2$ | % effects of action |

¹ Only binary places exist in CASE/EDI

Further more, one can precise when this simultaneity is obligated (O) :

$O(\text{MacroAction} / \beta)$ % capability constraint
 $\text{MacroAction} \leftrightarrow (\text{Action2} \otimes \text{Action3})$ % action composition

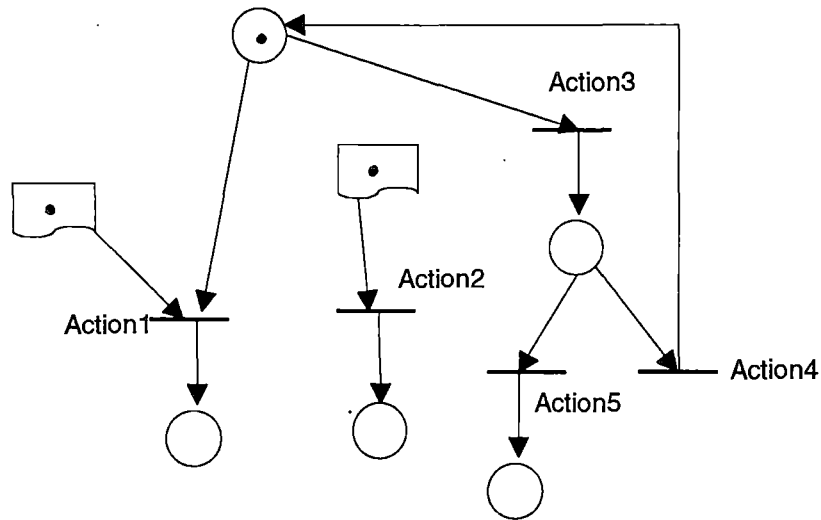


Figure 3.1 (concurrency)

Decision points

These situations are met when actions can be executed individually but not simultaneously or successively, often because they need the same resources or simply because they are antagonist. Concretely, in CASE/EDI, decision points are modeled by imposing a common input place to the transitions in question (see figure 3.2).

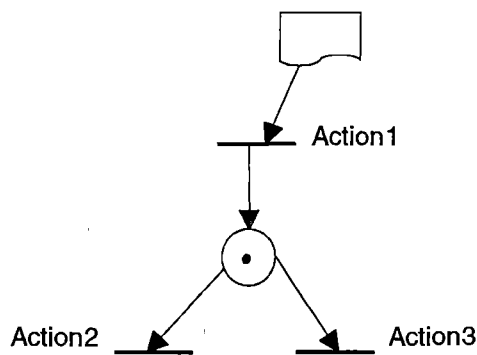


Figure 3.2 (choice)

Such choice situations do not exist in ALBERT but can be modeled by capability and causality constraints using temporal operators. Actually, these situations are rather forbidden sub-sequences of concurrent actions. For instance, the net of figure 3.2 can be expressed by the two following constraints :

- | | |
|---|--------------------------------|
| (1) $MacroAction \leftrightarrow (Action2 \otimes Action3)$ | <i>% action composition</i> |
| $F(MacroAction / TRUE)$ | <i>% capability constraint</i> |
| (2) $Flag: =FALSE$ | <i>% initial valuation</i> |
| $Action1: (Flag: =TRUE)$ | <i>% effect of action</i> |
| (3) $F(Action2 / \neg Flag)$ | <i>% capability constraint</i> |
| $F(Action3 / \neg Flag)$ | <i>% capability constraint</i> |

which state that (1) an *Action2* and *Action3* can never be performed simultaneously , and that (3) the first action performed can only be *Action1*.

Synchronization

Parallelism is useful only if the different processes (the agents in ALBERT, the roles in CASE/EDI) can cooperate. Such cooperation requires the sharing of information and resources. This sharing must be controlled via synchronization mechanisms, to ensure the correct behavior of the whole system. Synchronization points are used to wait for the termination of a specified number of cooperating processes before continuing. They can be easily implemented in a Petri net by linking, for each transition (*Action1* and *Action2*) to be synchronized, at least one output place to the transition (*Action3*) that must follow their termination (see figure 3.3).

Again, we do not find in ALBERT, the equivalent of synchronization points. The correctness of the overall system is ensured by a combination of cooperation, action composition and capability constraints (see below). This combination can be understood as synchronization directives. The cooperation constraints are used to warn an agent about the occurrence of an action "outside", the causality and capability constraints indicate the order of the actions occurrences (i.e. the possible sequences of actions in the time) :

- | | | |
|----------------|---|---------------------------|
| <u>Agent1:</u> | $K(Action1.Agent3 / TRUE)$ | <i>% cooperation</i> |
| <u>Agent2:</u> | $K(Action2.Agent3 / TRUE)$ | <i>% cooperation</i> |
| <u>Agent3:</u> | $MacroAction \leftrightarrow (Agent1. Action1 \parallel Agent2. Action2)$ | <i>% action compos.</i> |
| | $MacroAction: (Flag: =TRUE)$ | <i>% effect of action</i> |
| | $F(Action3 / \neg Flag)$ | <i>% capability</i> |

where $K(action.Agentx / \beta)$ states that agent x knows (K) under condition β when an action is performed.

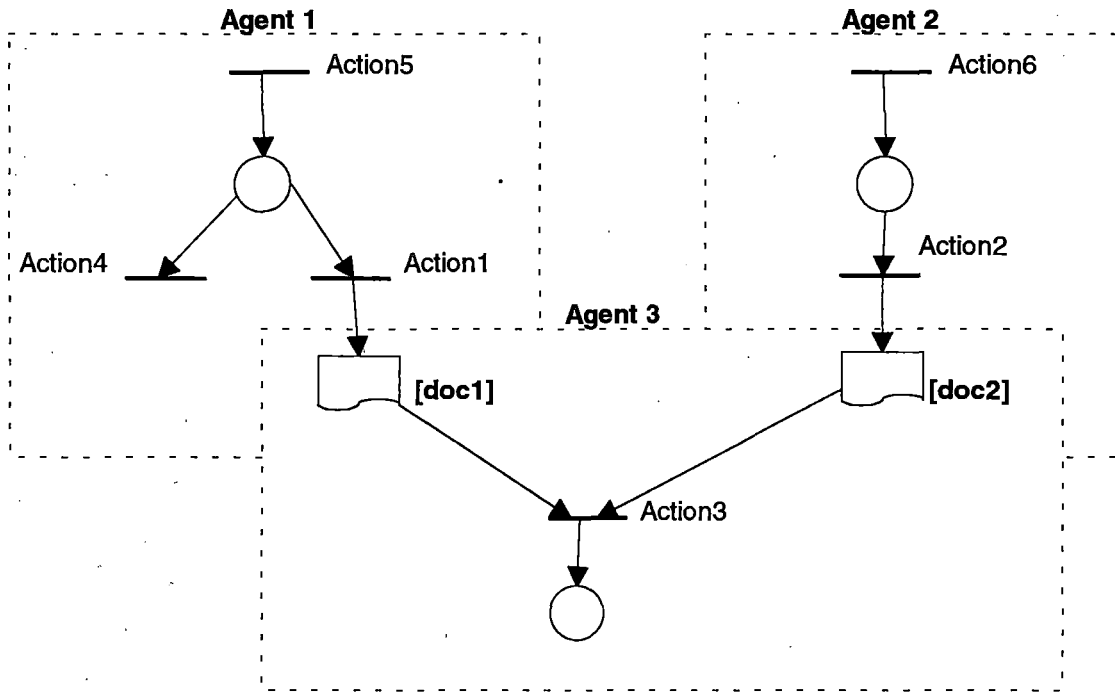


Figure 3.3 (synchronization)

3.2.2 Causality

Let us now examine how the causality is modeled in both languages. In temporal logic based languages like ALBERT, we don't need to describe when an action can be performed. We simply define the causality link between actions : instead of imposing the moment an action has to occur, we refer to the occurrence or distance of some other actions. The system is thus expressed in terms of sequences of events or process (in the sense of process algebra) by means of combinators in action composition constraints.

In the Petri net based languages such as CASE/EDI, actions are triggered through ECA (Event-Condition-Action). Actions can occur when pre-conditions are met, that is when all input places contain a token. The causality between two actions can be seen graphically by examining the places that are at the same time input place for the first transition, and output place for the other transition (see figure 3.4).

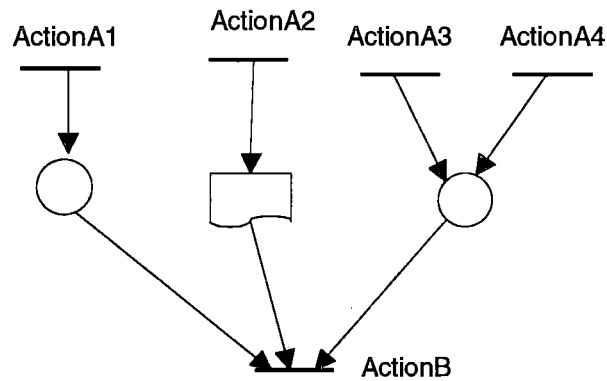


Figure 3.4 (causality between actions)

Note that the situation here above can be translated in the following action composition constraint in ALBERT :

$$ActionA1 || ActionA2 || (ActionA3 \oplus ActionA4); ActionB$$

This would suggest that a Petri net can be converted into constraints describing the causality relationships between the transitions of the net. However, the presence of self-loops, as in the next example, can rise some problems. Indeed, the constraint $(Action3; Action4)$ does not hold here, since one cannot be sure that $Action4$ will fire one day, after the firing of $Action3$ (i.e. a possible firing sequence could be $Action1, Action3, Action2, Action2, Action2, \dots, Action2, \dots$)

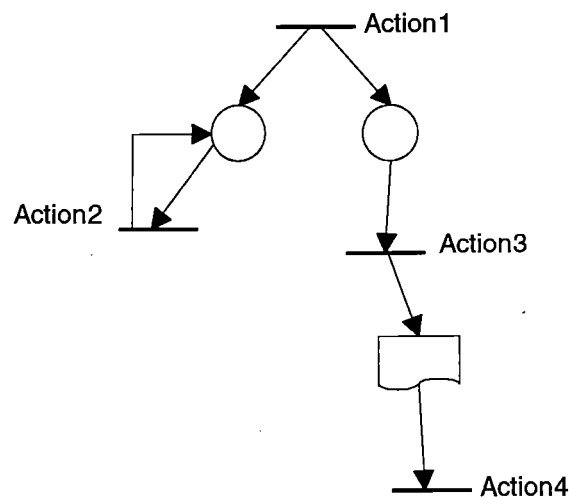


Figure 3.5 (a net with a self-loop)

3.2.3 Determinism and non-determinism

From the moment we want the specification language to also model human behaviors, like in a business procedure, it is necessary to take into account the uncertainty of actions. We have to distinguish deterministic events (things that *must* happen) and non-deterministic ones (things that *may* happen). Now we are interested in how we can model obligations, permissions and preventions in both languages.

ALBERT has a default rule: in the absence of constraints, all actions are permitted, whatever the situation. An action may occur, except if this action is forbidden in the current state of the system, or if an obligation is attached to it (in that case, it must occur). ALBERT provides a set of special connectives (*F* for forbidden, *O* for obligation and *XO* for exclusive obligation) in order to describe the responsibility of an agent with respect to its own actions. Since in ALBERT and more generally in temporal logic based languages, the system is viewed as a generator of possible lifes for each agent, deontic features can be modelled by defining the semantics of such connectives on those lifes.

In contrast, due to the non-deterministic nature of the Petri nets (when several actions are enabled, each of them may be the next to fire), one cannot explicitly represent obligations in CASE/EDI. A possible solution would be to map the obligations into a set of mathematical properties that the net has to respect. But there exists no method to derive a Petri net from such properties. These deontic aspects, especially the modeling of obligations and prohibitions will be discussed in the next chapters (chapter 4 and 5).

3.2.4 Conclusion

If we model with Petri nets, we can take advantage of the numerous analysis methods and simulation tools. Their graphical representation and their principles are very simple, even for a non-specialist. But on the other hand, it is sometimes difficult to represent complex preconditions or elaborated constraints (like performance constraints or permissions). Such constraints are easier to model by using the temporal logic formalism. Hence have we thought (in chapter 5) about a language based on mathematical and logical grounds. This will for instance allow the analyst to refer to a particular state of the system in the past or in the future. Such reference would have lead to the introduction of somewhat artificial places and transitions in the Petri net description.

3.3 Transformation of Petri nets into ALBERT

3.3.1 The mapping of the places and tokens

Remember that a basic Petri Net structure is characterized (see definition 1.1) by a set P of places, and that (see definition 1.2) the marking M of the net gives the distribution of the tokens in these places.

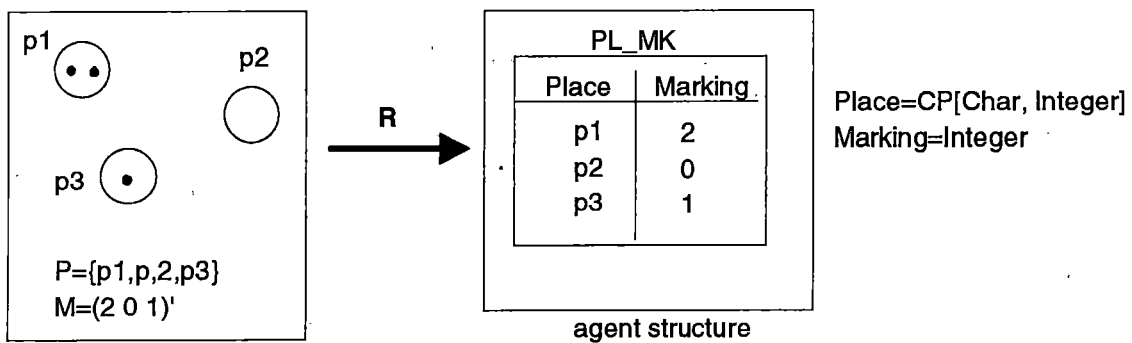
We propose to map the set P of places into a table² PL_MK - for places and marking - belonging to the agent structure and which contains as many "records" (lines) as places of P .

The two following rules have to be respected:

$$(R1) \quad \forall p \in P: p \in Dom(PL_MK)$$

$$(R2) \quad \forall p \in Dom(PL_MK): p \in P$$

For the mapping of the marking M , we can use the same table (PL_MK). We propose to replace the number of tokens of a place p by an integer equal to $M(p)$ that is to be inserted in the second column of the table, at the line related to the place p :



and the first two rules become :

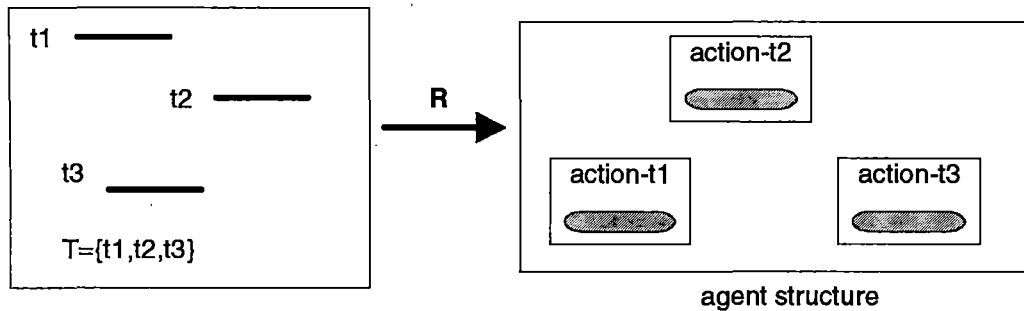
$$(R1)' \quad \forall p \in P: (p \in Dom(PL_MK) \wedge Marking[p] = M(p))$$

$$(R2)' \quad \forall \langle p, i \rangle \in PL_MK: (p \in P \wedge i = M(p))$$

² A table in ALBERT is a set of 2-tuples (ID, X), where ID is an identifier and X any information related to ID.

3.3.2 The mapping of the transitions

Another element of a Petri net structure is the set T of transitions (see definition 1.1). We can easily map the transitions of a Petri net into actions belonging to the agent structure of the ALBERT specification :



That way, we obtain the next transformation rule.

$$(R3) \quad \forall t \in T: t \xrightarrow{R} action_t \wedge \forall action_t: \exists t \in T$$

3.3.3 The mapping of the arcs

In a Petri net structure, the arcs are represented by two functions, *Pre* and *Post*, mapping each transition into a bag of input/output places (see definition 1.1).

Intuitively, we can consider the input arcs as preconditions that have to be satisfied to fire a transition. For example, $Pre(p1, t2) = 2$ means that the place $p1$ must contain at least two tokens to fire the transition $t2$. In other words, it is forbidden to fire the transition $t2$ if the place $p1$ contains less than two tokens. We saw (in chapter 2) that ALBERT provides a special connective (*F*) to express preventions in capability constraints. So, we propose the following mapping for the input arcs :

$$(R4) \quad \forall t \in T, \forall p \in P: (z_i = Pre(p, t) > 0 \xrightarrow{R} F(action_t / Marking[p] < z_i))$$

Thus, there are as many preventions as numbers, in the input matrix, greater than zero:

$$\begin{array}{ccc}
 \text{Pre} & t1 & t2 \\
 p1 & 1 & 0 \\
 p2 & 0 & 2 \\
 p3 & 1 & 0
 \end{array}
 \xrightarrow{R}
 \begin{array}{l}
 \text{Capability constraints:} \\
 F(\text{action_}t1 / \text{Marking}[p1] < 1) \\
 F(\text{action_}t1 / \text{Marking}[p3] < 1) \\
 F(\text{action_}t2 / \text{Marking}[p2] < 2)
 \end{array}$$

Moreover, the input arcs also inform us about the number of tokens consumed by the firing of a transition. Since we mapped the places P and the marking M into an ALBERT table, the input matrix Pre will be translated in "effects of actions" constraints that will change some values in the column "Marking" of this table :

$$(R5) \quad \forall t \in T, \forall p \in P: (z_i = \text{Pre}(p,t) > 0 \xrightarrow{R} \text{action_}t: \text{Dec}(\text{Marking}[p], z_i))$$

where $\text{Dec}(x,i)$ is a function decreasing x by i .

For instance,

$$\begin{array}{ccc}
 \text{Pre} & t1 & t2 \\
 p1 & 1 & 0 \\
 p2 & 0 & 2 \\
 p3 & 1 & 0
 \end{array}
 \xrightarrow{R}
 \begin{array}{l}
 \text{Effects of actions:} \\
 \text{action_}t1: \text{Dec}(\text{Marking}[p1], 1) \\
 \text{action_}t1: \text{Dec}(\text{Marking}[p3], 1) \\
 \text{action_}t2: \text{Dec}(\text{Marking}[p2], 2)
 \end{array}$$

On the other hand, the output arcs in a Petri net describe the effect of the firing of a transition on the token distribution. For example, $\text{Post}(p2,t2)=1$ means that the firing of the transition $t2$ adds one token to the place $p2$. Here again, the output matrix $Post$ will be translated in "effects of actions" constraints altering the table content :

$$(R6) \quad \forall t \in T, \forall p \in P: (z_o = \text{Post}(p,t) > 0 \xrightarrow{R} \text{action_}t: \text{Inc}(\text{Marking}[p], z_o))$$

where $\text{Inc}(x,o)$ is a function increasing x by o .

For instance,

<i>Post</i>	<i>t1</i>	<i>t2</i>		<u>Effects of actions:</u>
<i>p1</i>	0	0	\xrightarrow{R}	<i>action_t1:Inc(Marking[p2],2)</i>
<i>p2</i>	2	1		<i>action_t2:Inc(Marking[p2],1)</i>
<i>p3</i>	0	1		<i>action_t2:Inc(Marking[p3],1)</i>

3.3.4 The simultaneity

In a Petri net, when several transitions are enabled (see definition 1.3), we have to choose which one we want to fire first. It is not possible to fire two transitions at the same time³, because the firing of the first one might disable the second one, as in the following example:

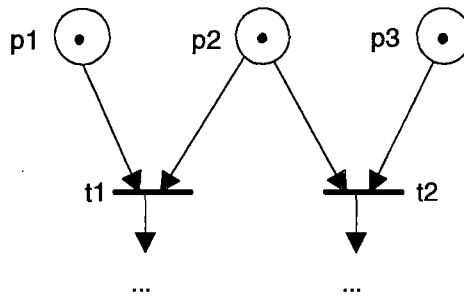


Figure 3.6 (two enabled transitions)

In contrast, in ALBERT, it is always permitted to execute several actions simultaneously except if it is forbidden in a constraint, or if their occurrence alter the same state component(s). We have thus to care for that our mapping does not allow sequences of actions that are not possible in the corresponding Petri net. Let us see if this mapping solves the problem of simultaneity. The preventions and the PL_MK table derived from the previous net (figure 3.6) would be :

³ Actually, if the successive firing of two transitions is possible, it is *asif* they fire simultaneously, since the firing is an instantaneous event taking zero-time.

PL_MK	
Place	Marking
p1	1
p2	1
p3	1

Capability constraints :

$F(action_t1 / Marking[p1] < 1)$

$F(action_t1 / Marking[p2] < 1)$

$F(action_t2 / Marking[p2] < 1)$

$F(action_t2 / Marking[p3] < 1)$

Effects of actions :

$action_t1: Dec(Marking[p1],1)$

$action_t1: Dec(Marking[p2],1)$

$action_t2: Dec(Marking[p2],1)$

$action_t2: Dec(Marking[p3],1)$

Figure 3.7 (the corresponding ALBERT constraints)

The four capability constraints (figure 3.7) allow us to execute the sequence ($action_t1 \otimes action_t2$) - where the symbol \otimes denotes the simultaneity - which is not permitted in the Petri Net. But since these two actions alter the same "line" of the table (i.e. the one related to p2) in their effects of actions constraints, it is not permitted to execute them simultaneously. More generally, the mapping R ensures that two transitions having at least one (input or output) place in common, cannot be performed at the same time in ALBERT; in all other cases, they can (because the two actions modify different "lines" of the table). However, if we want a semantics-preserving transformation, both systems must have the same sequences of states. For instance, the ALBERT sequence of states $s_0 \xrightarrow{t1} s_1 \xrightarrow{t2 \otimes t3} s_2 \xrightarrow{t4} s_3$ is mapped into the Petri net sequence $M_0 \xrightarrow{t1} M_1 \xrightarrow{t2} M_2 \xrightarrow{t3} M_3 \xrightarrow{t4} M_4$ which includes one more intermediary state. Therefore, we introduce in the ALBERT specification a dummy state component (dum_st) and a dummy effect (f_d) for *every* action, in order to forbid any simultaneity of actions in ALBERT :

$$(R7) \quad \forall t \in T \xrightarrow{R} action_t: dum_st = f_d(dum_st)$$

That way, all the actions do modify the component dum_st and thus, two actions cannot be executed at the same time since they alter the same component.

3.3.5 Illustration of the transformation rule R

Let's consider the following net. It models a critical section (CS) :

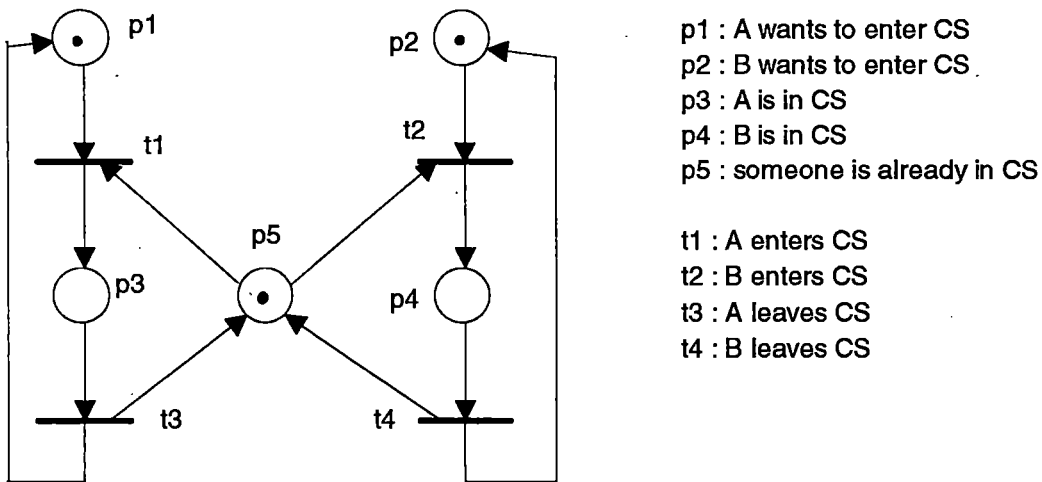


Figure 3.8 (the graphical description of a Petri net)

which can be formalized by the following structure $(P, T, Pre, Post)$ and the marking M :

	<i>Pre</i>	<i>t1</i>	<i>t2</i>	<i>t3</i>	<i>t4</i>	<i>Post</i>	<i>t1</i>	<i>t2</i>	<i>t3</i>	<i>t4</i>
$P = \{p1, p2, p3, p4, p5\}$	<i>p1</i>	1	0	0	0	<i>p1</i>	0	0	1	0
	<i>p2</i>	0	1	0	0	<i>p2</i>	0	0	0	1
$T = \{t1, t2, t3, t4\}$	<i>p3</i>	0	0	1	0	<i>p3</i>	1	0	0	0
	<i>p4</i>	0	0	0	1	<i>p4</i>	0	1	0	0
$M = (1\ 1\ 0\ 0\ 1)$	<i>p5</i>	1	1	0	0	<i>p5</i>	0	0	1	1

Figure 3.9 (the formal description of the same net)

and translated in an equivalent ALBERT specification:

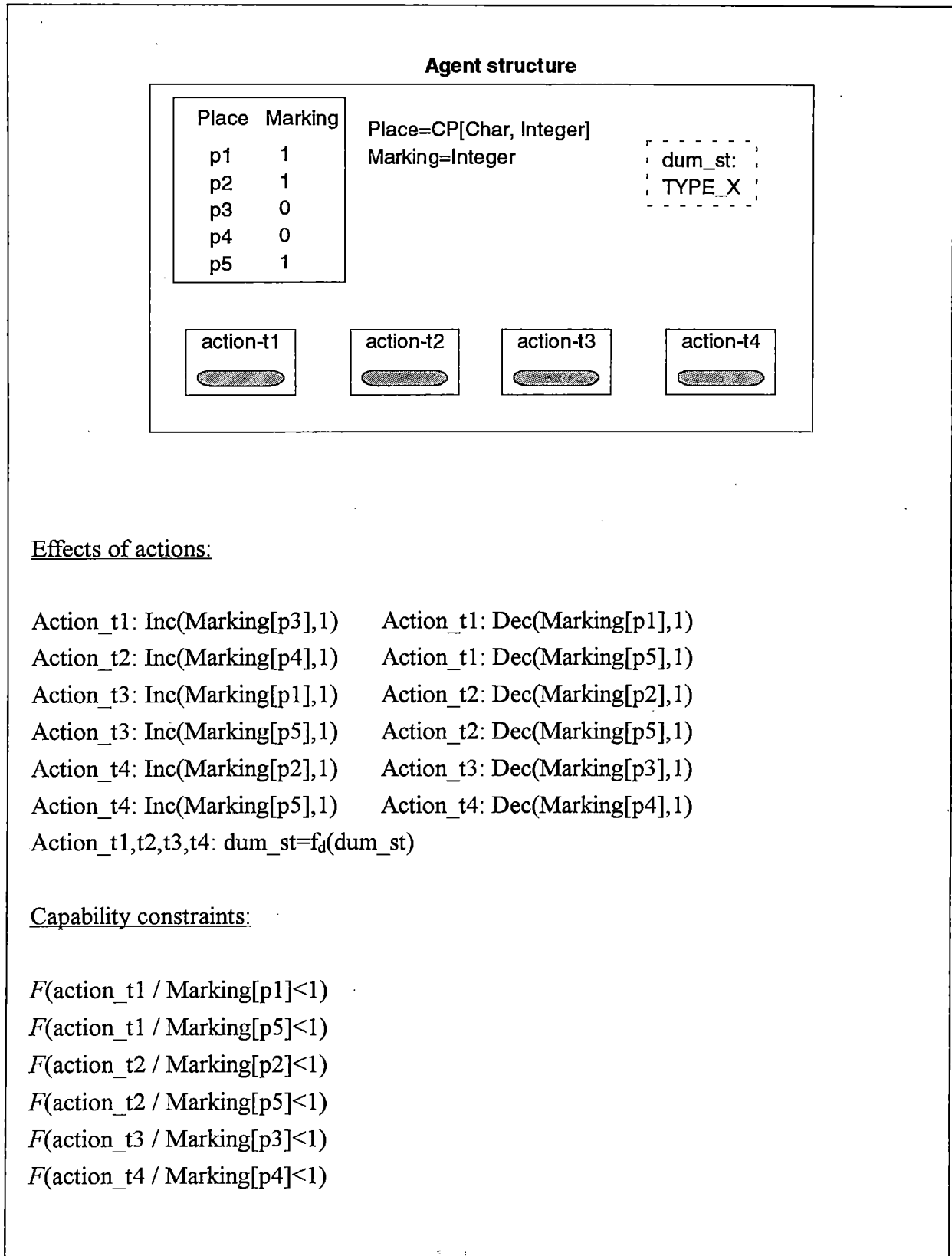
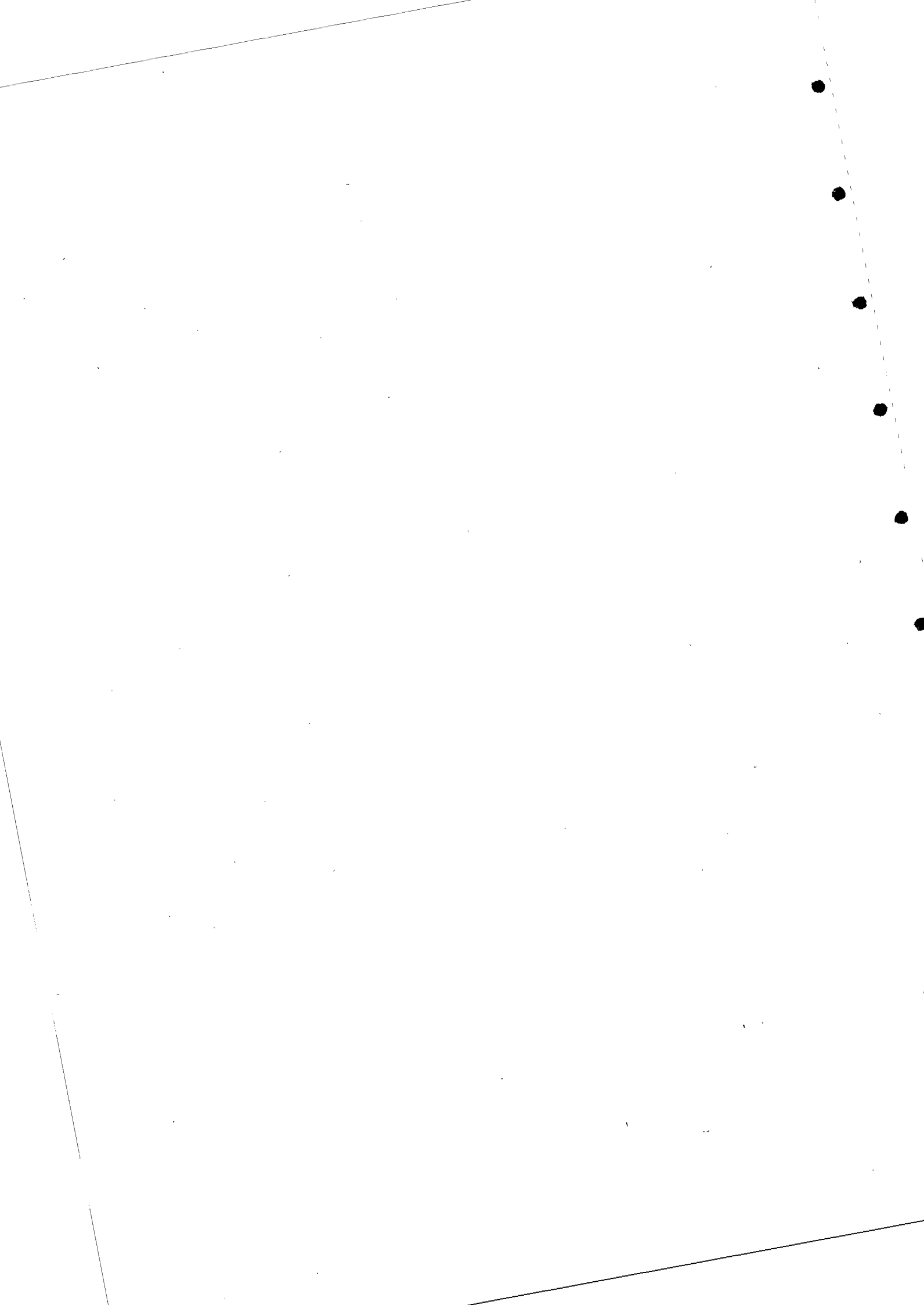


Figure 3.10 (the derived ALBERT specification)

3.3.6 Summary and conclusion

We have formally defined a semantics-preserving transformation rule translating Petri nets into the ALBERT language. The purpose of this section was to show that it is possible to translate an operational specification (a Petri net) into another declarative and more readable language (ALBERT). This can be very interesting in a perspective of retro-engineering. Since Petri nets (colored or not) and ALBERT are located at different levels in the specification phase, it can sometimes be useful to "go up" by one level, namely in order to check the correctness of the development process. However, it must be stressed that this transformation does not induce an equivalence between those two languages. Indeed, a lot of constraints in ALBERT are not "translatable"⁴ in Petri nets (even extended Petri nets).

⁴ Some of those constraints may be translated, but very often, it results in a too complicated (and thus unreadable) Petri net.



Chapter 4

Deontic Logic

4.1 Introduction

The word 'deontic' is derived from the Greek word 'δεοντως' which may be translated as 'as it should be' [HILPINEN71]. Deontic logic is thus characterized by the distinction between the actual and the ideal. Deontics deals also with normative use of language such as permission, obligation and prohibition.

Deontic aspects are met in many areas : contracts, human behavior, laws [TT95], ... [MEYER 91]. A lot of research is made in logic to capture deontic aspects, to obtain a logical system which permits to reason about deontic facts [DIG94], [TORRE94] These logical systems permit to specify, for example, the behavior of persons involved in contracting. We can specify the ideal behavior of these persons. For example, when the seller accepts the terms of a contract, he has the obligation to deliver the goods before a certain date. But we can also be interested in expressing the consequences of a violated obligation. If the seller doesn't deliver the goods before the specified date, he has to pay a fine. It's the same with prohibition, one may be interested in consequences which follow a violated prohibition.

In this chapter, we briefly present in section 4.2 the SDL deontic system and its limitations. In section 4.3, we underline the concepts of interest in deontic for the specification of distributed systems.

4.2 SDL : a modal logic for deontic reasoning

SDL, Standard Deontic Logic, is the more familiar deontic system. This logic system is based on the works of G. H. von Wright [WRIGHT51]. Von Wright's approach to deontic logic is based upon the observation that there exists a significant analogy between the **deontic** notions *obligation* and *permission* and the **modal** notions *necessity* and *possibility*. In fact, a proposition p is necessary if and only if its negation $\neg p$ is not possible, this expresses the definition $L(p) \equiv \neg M(\neg p)$ (see sub-section 2.2.1). Similarly, an act or a fact p is obligatory if and only if its negation $\neg p$ is not permitted. The notion of permission is the primitive of the von Wright's system. $P(p)$, where P is a modal operator, expresses that p is permitted. The notion of obligation is defined in terms of permission by : (def) $O(p) \equiv \neg P(\neg p)$, where $O(p)$ must be read : ' p is obligated'.

Standard deontic logic (SDL) respects the definition (def) and two other axioms (KD) :

- (K) $O(p \rightarrow q) \rightarrow (O(p) \rightarrow O(q))$ which states that modus ponens holds within the scope of the modal operator O .
- (D) $\neg(O(p) \wedge O(\neg p))$ which states that something cannot be obliged to be the case and obliged not to be the case at the same time.

Another operator is also often defined : $F(p) \equiv \neg P(p)$ which states that something is forbidden if and only if it is not permitted.

Until now we have considered the axiomatic definition of SDL, we give here the definition of a model for a deontic theory in SDL.

Definition 4.1 (World model of a deontic theory in SDL)

A possible world (Kripke) model for a deontic theory in SDL is a 4-tuple $M = \langle w_M, W, R, V \rangle$ where w_M is the actual world $w_M \in W$, W is a non-empty set of worlds, R is an accessibility relation between worlds, V is a valuation function that assigns in each world $w \in W$ a truth value to atomic propositions.

A formula $O(p)$ is true in a world w in a model M , written $M, w \models_{\text{SDL}} O(p)$, iff for all world w' with wRw' : $M, w' \models_{\text{SDL}} p$. A formula p is true in a model M , written $M \models_{\text{SDL}} p$ iff $M, w_M \models_{\text{SDL}} p$. A formula p entails q , written $p \models_{\text{SDL}} q$, iff $M \models_{\text{SDL}} p$ then $M \models_{\text{SDL}} q$.

The obligations $O(p)$ that can be derived from a SDL theory T can be classified in :

- *fulfilled obligations* if p is entailed by the theory T .
- *violated obligations* if $\neg p$ is entailed by the theory T .
- *moral cue* if neither p nor $\neg p$ is entailed by the theory T .

The violated obligations are an answer to the question 'what has been done wrong?' (what is the case but should not be the case) and moral cue are answers to the question 'what should be done now ?' (what is not yet the case or not the case but should be done), [TT94a]. More formally :

Definition 4.2 (Fulfilled, violated obligation, moral cue)

Let T be a theory of SDL, $O(p)$ is a *fulfilled obligation* of the theory T iff $T \vdash_{\text{SDL}} O(p)$ and $T \not\vdash_{\text{SDL}} \neg p$. $O(p)$ is a *violated obligation* of the theory T iff $M \vdash_{\text{SDL}} O(p)$ and $M \vdash_{\text{SDL}} \neg p$. $O(p)$ is a *moral cue* of the theory T iff $M \vdash_{\text{SDL}} O(p)$, $M \not\vdash_{\text{SDL}} p$ and $M \not\vdash_{\text{SDL}} \neg p$.

Unfortunately, SDL is plagued by a large number of paradoxes. For instance the formula $O(p) \rightarrow O(p \vee q)$ is a theorem of SDL. This theorem says that if a certain state of affairs p ought to be the case then $p \vee q$ ought to be the case. We can interpret this formula as follows [HILPINEN71] : 'If I ought to mail a letter, I also ought to mail or burn it. But if I in fact ought to mail a letter, then surely it is awkward to say that I ought to mail it or burn it'. Other notorious paradoxes are Forester and Chisholm paradoxes. Those paradoxes are consequences of an impossibility to model correctly *contrary-to-duty* (CTD) obligations in SDL.

Definition 4.3 (Contrary-to-duty obligation)

A *contrary-to-duty obligation* is an obligation conditional to a violation describing sub-ideal behavior. The conditional obligation $\alpha \rightarrow O(\beta)$ is a CTD obligation of the (primary obligation) $O(\delta)$ when α and δ are contradictory.

An example of CTD obligation (2) of a primary obligation (1) :

- (1) $O(p)$
- (2) $\neg p \rightarrow O(q)$

For Tan and Torre (see [TT94a, TT94b]), the fundamental problem underlying these paradoxes is that the type of possible world semantics of SDL is not flexible enough. In these semantics only two types of worlds are distinguished in a model; actual and ideal ones. The ideal worlds have to satisfy all obligations in a deontic theory T . Clearly, if these obligations contradict each other, then a problem arises. (...) in order to model these paradoxes properly, we need a notion of sub-ideal worlds, in which some but not all obligations are satisfied.

A further development would go beyond the scope of this work. Nevertheless, the idea of sub-ideal worlds will be adapted in chapter 5 for representing varying sub-ideality in Petri nets.

4.3 Deontic aspects in our work

In this work we use deontics because deontic aspects often play an important role in the specification of distributed systems [DUBOIS91].

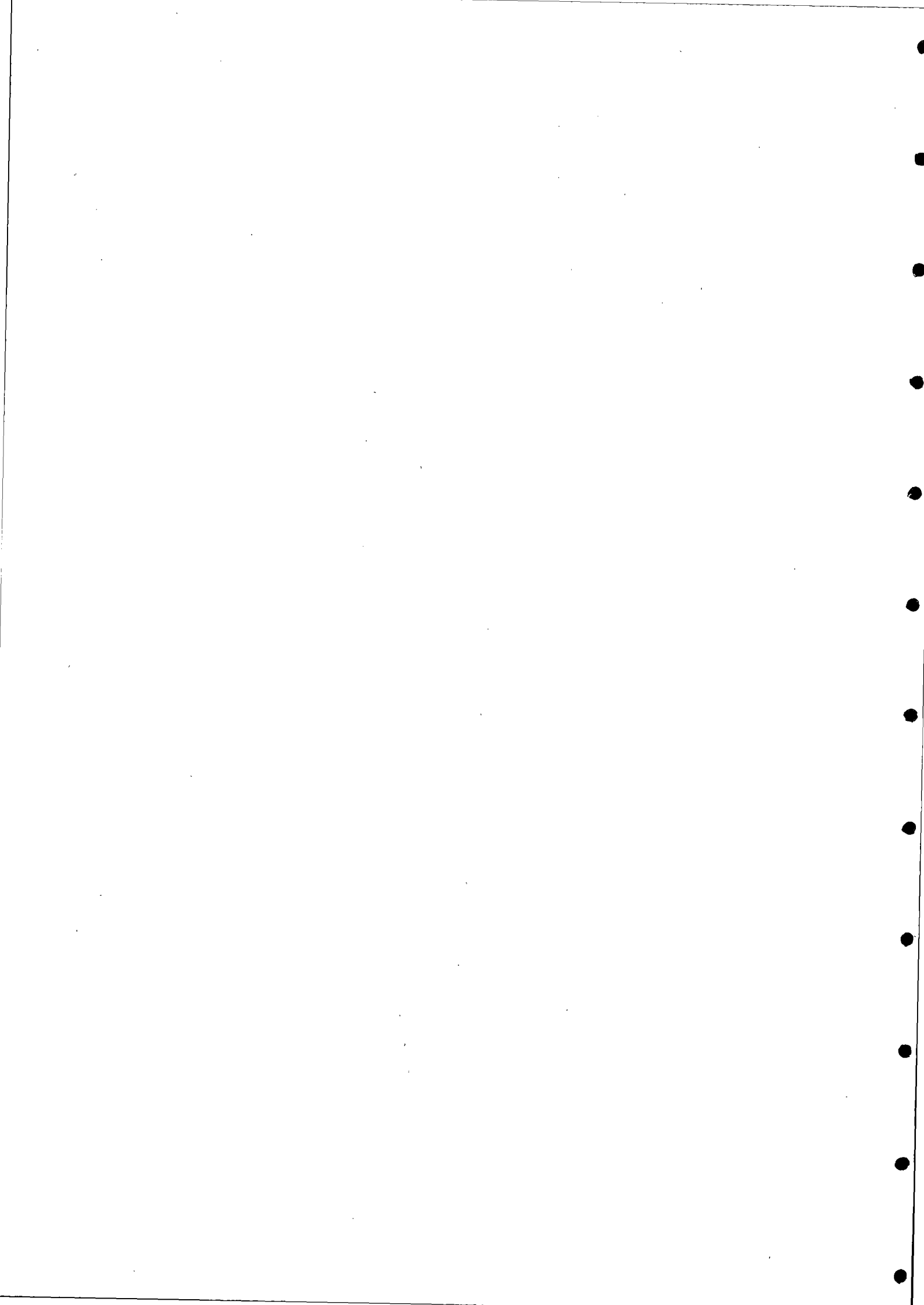
In distributed systems, components interact to achieve a common goal. In such systems, a component may send a service request to another component of the system. If a component receives a service request, it has the **obligation** to execute the asked service. We can say that there exists a kind of contract between the components of a distributed system. At this point, we may partition the deontic aspects in two categories : the strict obligations/prohibitions and the obligations/prohibitions which can be violated.

A strict obligation/prohibition is always respected. The situations where a strict obligation is violated are not considered. It must be noted that this notion of strict obligation, which can not be violated, is quite different from the deontic notion of obligation. In fact, a strict obligation is rather an usual constraint which must be respected by the system to be implemented. In the following chapter, we investigate how this kind of constraint can be modeled in Petri nets. We have to study such constraints because they are often expressed in declarative style and it may be problematic to represent them in an operative style.

Another important aspect in distributed systems, especially when human components play a role, is the obligations/prohibitions that can be violated. When specifying a human behavior, but sometimes also a hardware component, we may consider sub-ideal situations, situations resulting of the violation of an obligation/prohibition or of the execution of an action judged sub-ideal. In those cases, it should be possible to reflect in a formal way which behaviors are sub-ideal. We will investigate this problem in the Petri nets approach in section 5.2.

Remark. It should be noted that the operators F , O and XO introduced in ALBERT (see subsection 2.4.5 p. 2.9) support only strict obligations/prohibitions. The notion of sub-ideality is not covered by the ALBERT language.

PART TWO
A NEW INTEGRATED LANGUAGE



Chapter 5

Petri Nets and Deontic aspects

5.1 Introduction

In this chapter we see how deontic aspects can be represented in the Petri net formalism. In the introductory chapter to deontic logic, we have classified deontic aspects in two categories : strict deontic aspects and the deontic distinction between ideal and sub-ideal behavior. We keep here this latter approach. So, we see in a first section how strict deontic aspects can be represented in the original Petri net formalism. In a second section, we propose an extension of the Petri net formalism in order to make possible the formalization of ideal and varying sub-ideal behaviors.

5.2 Strict Obligations/Prohibitions

Recall that when modeling a system, one must limit the part of the system which is modeled. The model represents always a part of the reality. For example, if a contract is represented in a Petri net, the net should represent that if goods have been delivered by the seller, then subsequently the buyer is obliged to pay the bill for the goods. Also the net should be able of representing that if the buyer does not pay, the buyer is obliged to return the goods. But the modeler may make the hypothesis that the second obligation (returning the goods in situation of no payment) can not be violated and thus is always respected. Due to this hypothesis, the model does not represent the juridical procedures that can follow from a refusal of returning the goods. This kind of obligation, which can not be violated, is called in our

terminology a **strict obligation**. Strict obligations can be opposed to **obligations which can be violated**. In this section, we see how to represent strict obligations/prohibitions in the original Petri net formalism. More precisely, we give conditions that a net must satisfy to represent strict obligations/prohibitions. We also underline the limitations of the approach of keeping the original Petri net formalism for the representation of strict deontic aspects.

5.2.1 Stricts obligations

In this sub-section, we define conditions that a Petri net must satisfy to model a strict obligation. A strict obligation is anything **which ought to be done** (like returning the goods in case of no payment). If t_i is the transition that models, in a Petri net, a strict obligated action in a state D , t_i must be fired in the markings representing state D .

Recall that when two transitions t_1, t_2 are enabled in a marking M of a net N , the firing rule of the Petri net formalism says that either t_1 or t_2 fires in marking M . Thus if a transition ought to fire in a marking M , it has to be **the only one** enabled in M . We can write this condition in a more formal style.

Condition 5.1 (Strict obligation in a Petri net).

Let α denote an action of the system S , S_α the set of states of the system S where the action α ought to be done (strict obligation), MS_α the set of markings that model S_α , t_α the transition that models the action α , then :

$$\forall M \in MS_\alpha:$$

$$(1) \forall p \in P: M(p) \geq Pre(p, t_\alpha)$$

$$(2) \forall t \in T / \{t_\alpha\}: \exists p \in P: M(p) < Pre(p, t)$$

(1) guarantees that t_α is enabled in all markings M which belong to MS_α .

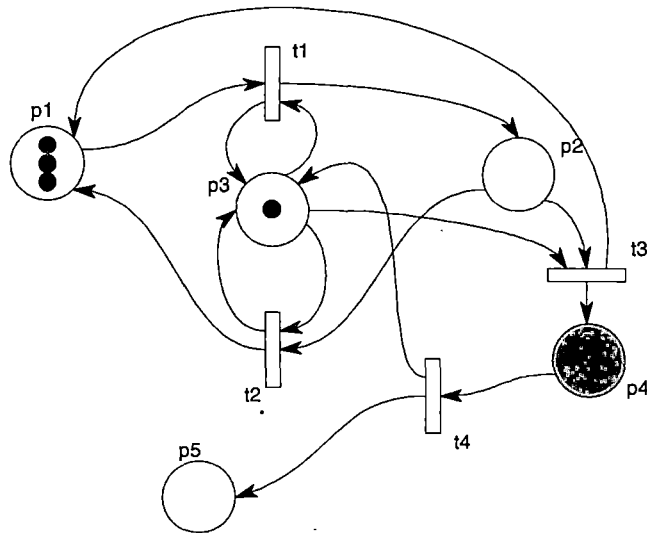
(2) guarantees that t_α is the only enabled transition in all markings which belong to MS_α .

To illustrate the condition 5.1, let us continue with the following example and see how the condition 5.1 can be applied.

Example 5.1 (Modeling a strict obligation).

In library rules, we may read that a borrower must have returned a book within a month and if it's not the case, he ought to pay a fine to be authorized to borrow again or to do anything else in the library. In this example, we can distinguish two kinds of obligation: the first one, returning the book within a defined period which can be violated, the second one, the

obligation to pay the fine, which can not be violated. The payment of the fine is thus, in our terminology, a strict obligation.



p_1 : Book_Free; t_1 : Borrow_Book; p_2 : Borrowed_Book; t_2 : Return_in_time; p_3 : Permission_to_borrow; t_3 : Return_Late; p_4 : Violation_State; t_4 : Pay_Fine; p_5 : Fines_Paid.

Figure 5.1 (Petri net model of the library example)

If we apply the condition 5.1 to our library example, we have :

- α : Pay a fine.
- S_α : the set of states reached when the borrower returns a book too late.
- MS_α : the set of markings where $m(p_4)=1$.
- t_α : t_4 (Pay_Fine).

To be sure that the net of figure 5.1 respects the condition 5.1, we must prove that when the place p_4 is marked with one token (marking where the payment of the fine is strictly obligated), the only enabled and thus obliged transition is t_4 (Pay_Fine).

Proof 5.1¹ (The model of Figure 5.1 respects the condition 5.1).

Here, the matrix representation of the library example in terms of *Pre*, *Post*, *C* and M_0 :

					$t1$	$t2$	$t3$	$t4$						
<i>Pre:</i>					$p1$	1	0	0	0					
					$p2$	0	1	1	0					
					$p3$	1	1	1	0					
					$p4$	0	0	0	1					
					$p5$	0	0	0	0					
					<i>Post:</i>									
					$p1$	0	1	1	0					
					$p2$	1	0	0	0					
					$p3$	1	1	0	1					
					$p4$	0	0	1	0					
					$p5$	0	0	0	1					

¹ We use here the algebraic invariant method, other analysis techniques such as reachability analysis are available (see chapter I, The Petri net formalism).

$$\begin{array}{ccccc}
 & t1 & t2 & t3 & t4 \\
 p1 & -1 & 1 & 1 & 0 \\
 C: p2 & 1 & -1 & -1 & 0 \\
 p3 & 0 & 0 & -1 & 1 \\
 p4 & 0 & 0 & 1 & -1 \\
 p5 & 0 & 0 & 0 & 1
 \end{array}$$

$$M_0: \begin{bmatrix} 3 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

With the linear fundamental equation for the analysis of Petri nets : $M' - M = C \times \bar{s}$,
 $M \xrightarrow{\bar{s}} M'$ and $C \times w = 0$, we obtain the following system :

$$\begin{cases}
 -w(p1) + w(p2) = 0 \\
 w(p1) - w(p2) = 0 \\
 w(p1) - w(p2) - w(p3) + w(p4) = 0 \\
 w(p3) - w(p4) + w(p5) = 0
 \end{cases}$$

which has two non negative integer linearly independent solutions :

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

These two solutions give two independent invariants :

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \Rightarrow M(p1) + M(p2) \equiv \text{Const.}(1)$$

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \Rightarrow M(p3) + M(p4) \equiv \text{Const.}(2)$$

From (1) and the initial marking M_0 we conclude that the sum of tokens in place $p1$ and $p2$ is always equal to 3. Which is obvious since the number of books must stay constant in our system (a book is free or borrowed).

From (2) and the initial marking M_0 we conclude that the sum of tokens in place $p3$ and place $p4$ is always equal to 1 in all reachable markings. So we can say that if $p3$ is marked then $p4$ is not and that one of the two is marked in each reachable marking. This allows us to say that when we are in the violation state, $p3$ is not marked since $p4$ is marked and the only fireable transition is thus $t4$ (Pay_Fine).

Condition 5.1 is often too strict. If we consider an agent/object oriented approach, one models a system into a number of separated sub-nets and then merges them into a single net. An obligation, but also a prohibition, is often related to an agent/object or to an agent/object class. Thus an obligation for an agent A must normally have no direct influence on the behavior of another agent B .

Let us give a new condition for a net to represent strict obligation, in an agent/object oriented approach :

If t_α represents a "strictly" obligated action for an agent/object A whose behavior is modeled by a sub-net N_A in a marking M then the next transition of the sub-net N_A to fire is t_α .

Condition 5.2 (Strict obligation in an agent/object approach).

Let S denote the modeled system, A an agent/object that is part of the system S , N the aggregated net that models S , N_A the sub-net that models the agent/object A , α an action of A , t_α the transition that models α , MS_α the set of markings that models the states of S where A has the strict obligation to do α , T_A the set of transitions that belongs to net N_A , $E(N,M)$ the set of fireable sequences of transitions in marking M of the net N , $Pref(l,t)$ the longest prefix of the sequence l of transitions that does not contain the transition t .

$$\forall M \in MS_\alpha, \forall l \in E(N, M): T_A \cap Pref(l, t_\alpha) = \emptyset.$$

This condition garantees that when the transition t_α is strict obligated for an agent/object A modeled by N_A , t_α is the first transition of N_A to fire.

5.2.2 Strict prohibitions

Throughout the previous sub-section we were concerned about the modeling of strict obligations. The present sub-section deals with strict prohibitions. Recall that a strict prohibition is anything which is not permitted. In a Petri net, if t_i is a transition that models a prohibited action in a state D , t_i may not be fired in the markings representing D .

Again notice that in the Petri net formalism, when a transition is enabled, it may fire. Thus if a transition is strictly prohibited (may not fire) in a marking M , it may not be enabled. Let us write this condition in a more formal manner.

Condition 5.3 (Strict prohibition in a Petri net).

Let α denote an action of the system S , S_α the set of states of the system S where α is strictly prohibited, MS_α the set of markings that model the states of S_α , t_α the transition that models the action α :

$$\forall M \in MS_\alpha : \exists p \in P : M(p) < Pre(p, t_\alpha)$$

This condition guarantees that t_α is never enabled in markings of MS_α .

This condition can, like condition 5.1, be extended for an agent/object oriented approach.

As pointed out in part one, Petri nets are often used for their neat graphical representation. But by applying condition 5.3, the resulting net could be complicated and not very readable. To permit a direct graphical representation of strict prohibition, we can easily extend the Petri net formalism by adding inhibitor arcs.

This is the graphical representation of an inhibitor arc :

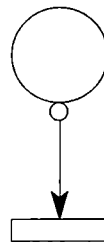


Figure 5.2 (An inhibitor arc)

To give the semantics of an inhibitor arc, we have to specify a new firing rule, different from the one of the original Petri net model.

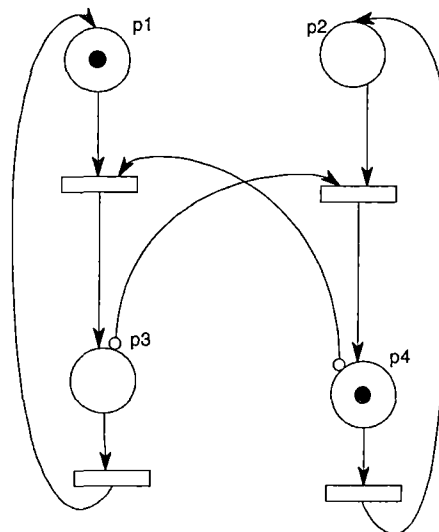
Definition 5.1 (Firing rule of Petri nets with inhibitor arcs).

A transition is enabled in a Petri net with inhibitor arcs when tokens are in all of its (normal) input places and zero tokens are in all of its inhibitor input places. The transition fires by removing tokens from its (normal) input places. A place is an inhibitor input place for a transition if this transition is linked to the place with a inhibitor arc.

Thanks to inhibitor arcs, a strict prohibition can be modeled in a more suitable way. To illustrate the ability of inhibitor arcs to model strict prohibitions, let us continue with the following example :

Example 5.2 (Critical section example).

Two processes pr_1 and pr_2 have a critical section, when one process executes its critical section the other one is not permitted to execute its one. Let's explicitly represent the strict prohibition to execute the critical section by inhibitor arcs.



p1 : Processus_pr1_normal_processing.
 p2 : Processus_pr2_normal_processing.
 p3 : Processus_pr1_critial_processing.
 p4 : Processus_pr2_critial_processing.

Figure 5.3 (Critical sections net)

It should however be noted that Petri nets with inhibitor arcs have not the same analytical possibilities as the original Petri nets. It can be shown (see [BRAMS 83b]) that Petri nets with inhibitor arcs have the modeling power of Turing Machines. For instance, boundedness is undecidable in Petri nets with inhibitor arcs. Fortunately, some Petri nets² with inhibitor arcs can be transformed, for analysis purposes, in an equivalent net without inhibitor arcs. The transformation consists in adding for each place linked with a inhibitor arc, a **complementary**

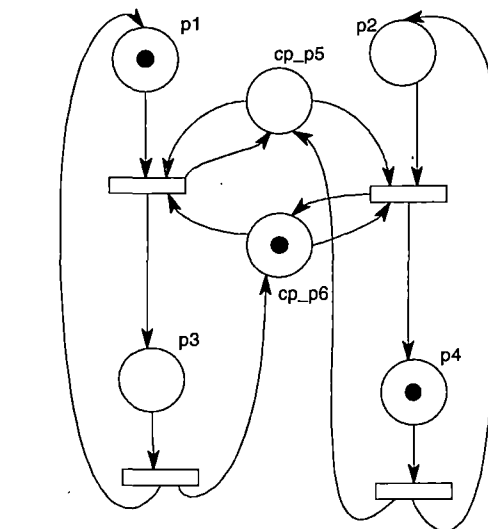
² Petri nets whose inhibitor places are bounded.

place and in linking this place with the transitions linked with the first place. This is precisely the kind of place we use in the library example (p_3 and p_4).

Definition 5.2 (Complementary place).

In a marked Petri net $\langle N, M \rangle$, two places $p_i, p_j \in P$ are **complementary places**, iff, $\forall M' : M' \in R(N, M), M'(p_i) + M'(p_j) = a$, where a is a constant. In other words, $p_i, p_j \in P$ are complementary places in a marked net if in all reachable markings the sum of tokens contained in p_i and p_j is constant.

If we apply the transformation introduced above on the net of figure 5.3, we obtain :



p1 : Processus_pr1_normal_processing.
 p2 : Processus_pr2_normal_processing.
 p3 : Processus_pr1_critical_processing.
 p4 : Processus_pr2_critical_processing.
 cp_p5 : Permission_processus_pr1_critical_processing.
 cp_p6 : Permission_processus_pr2_critical_processing.

Figure 5.4 (Critical sections net with complementary places)

The place cp_p5 is a complementary place of $p4$, and cp_p6 is a complementary place of $p3$. Note that the transformation is only possible if the inhibitor place is bounded, see [BRAMS83b] p. 47.

5.2.3 Limitations of the approach

In the sub-sections 5.2.1 and 5.2.2, we have only defined conditions that must be fulfilled for a correct modeling of strict deontic aspects in Petri nets. The modeler must find solutions

to represent strict obligations/prohibitions. These solutions must often be checked by analysis methods.

These difficulties are due to the over operational style of Petri nets. In fact, the Petri net formalism is a modeling language and not a specification language (see chapter 6). This characteristic (the operational style) may represent a serious hindrance for the use of Petri nets to model a system with a lot of strict deontic aspects. We think that the Petri net formalism should be extended. So we extend in the following chapter Petri nets with temporal logic. The main purpose of this extension is to make possible a declarative specification of non-operational constraints. In the end of the chapter 6, we show how easily strict deontic aspects can then be specified with temporal logic formulae.

The end of this chapter is concerned with another extension of the Petri net formalism expressing the deontic distinction between ideal and sub-ideal behaviors.

5.3 Modeling varying sub-ideality in Petri nets

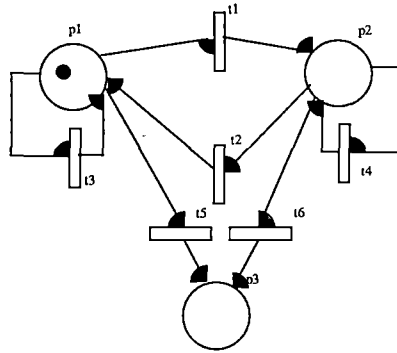
5.3.1 Introduction

Recall our contract example described in the previous section. If a contract is represented by a Petri net, the net should represent that if goods have been delivered by the seller, then subsequently the buyer must pay the bill for the goods. Also, the net should be capable of representing that if the buyer does not pay, then the buyer is obliged to return the goods. This second obligation, to return the goods, is conditional to the violation of the first to pay the delivered goods. Such an obligation, which is evoked when another obligation is violated, is called a *contrary-to-duty* (CTD) obligation (see definition 4.3). CTD obligations tell you what you should do, given that you already have violated an obligation. We say that a behavior by which no obligations are violated is *ideal*, and when there are obligations violated, the behavior is said *sub-ideal*. Clearly, CTD obligations apply to sub-ideal behavior only.

It should be possible to reflect, in a formal manner, that violating the first obligation (to pay the delivered goods) is not desired. In other words we should be able to reflect preferences between different possible scenarios. Scenarios where the goods are paid are preferred to scenarios where the goods are not paid and returned.

In this section we show how to represent the deontic notions of ideal and sub-ideal behavior in Petri nets. We extend standard Petri nets with a preference relation. This

preference ordering has common features³ with preference ordering that was introduced in DIODE [TT94a] and [TT94b]. It may be helpful to fix the notion of sub-ideality in Petri nets with an illustrative example before giving the formal definition of our preference ordering. To illustrate the distinction between ideal versus sub-ideal behavior, consider the following Petri net :



p_1 : borrowed book; p_2 : damaged book; p_3 : returned book; t_1 : To damage the book;
 t_2 : To repair the book; t_3 : 1 week too late; t_4 : 1 week too late; t_5 : To return the book;
 t_6 : To return the book;

Figure 5.5 (Book borrowing model)

Figure 5.5 models the possible behaviors of a borrower. A *marking* for this net is denoted by a tuple $\langle n_1, n_2, n_3 \rangle$ where n_i indicates the number of tokens at place p_i . In the initial marking $\langle 1, 0, 0 \rangle$ displayed in the figure 5.5, there is only a token at place p_1 , which represents that the borrower has a book. At this marking he has the choice between returning the book, returning it too late, or damaging the book. If he decides to be too late, he has again the same choices. If he damages the book he can choose to repair it before returning it, or to return it damaged, he may also be even later. In the rest of this section, we consider the executions of this Petri net which starts in marking $\langle 1, 0, 0 \rangle$ and ends in marking $\langle 0, 0, 1 \rangle$, which represents that the book is returned. When executing this Petri net, there is a choice between performing ideal behaviors or sub-ideal behaviors. For example, if the borrower returns the book in time and undamaged, we can say that he performs the ideal behavior. On the other hand, if he returns the book one week too late, he does not perform the ideal behavior. The distinction between ideal and sub-ideal cannot be represented in standard Petri nets as Figure 5.5. One can model the choice, but nothing in the Petri net formalism indicates that one execution is better than another one. In this section we show how the standard Petri net formalism can be extended with a preference relation such that it can represent this distinction. In the set of all possible executions of a

³ In the two approaches the possible models are ordered by order of sub-ideality. Nevertheless our ordering is linear when the ordering of DIODE is not.

system, intuitively, the preferred ones are those which contain a minimum of sub-ideal behaviors.

5.3.2 The extended Petri net formalism

In order to represent the distinction between ideal and sub-ideal states in a Petri net, we can partition the set of transitions of a Petri net into two subsets, that represent ideal and sub-ideal transitions respectively. Given this partition, we can define a preference ordering on executions of a net that compares the sub-ideal transitions of the executions. For example, we can define that an execution s is preferred to another execution s' iff s contains less sub-ideal transitions than s' .⁴ This preference ordering is denoted by the symbol \geq_n . As an example, consider the set $S = \{t_1, t_3, t_4\}$ that represents the sub-ideal transitions of our example in figure 5.5. Between marking $M_1 = \langle 1, 0, 0 \rangle$ and marking $M_2 = \langle 0, 0, 1 \rangle$, we have $\langle t_3, t_5 \rangle >_n \langle t_1, t_4, t_6 \rangle$ because the first execution contains less sub-ideal behaviors than the second one. This is intuitively correct, we prefer an execution in which one returns a book too late but undamaged to an execution in which one returns the book too late and damaged.

However, the execution $\langle t_3, t_5 \rangle$ and $\langle t_1, t_6 \rangle$ are equivalent for \geq_n . This is unintuitive, because one prefers a book returned late to a damaged book returned in time. The order relation \geq_n takes only into account the number of sub-ideal behaviors. However, the violations do not have the same seriousness. As a solution, the transitions can be partitioned in more than two subsets, which express the deontic notion of ideal and **varying** sub-ideal, see [TT94a], [TT94b]. This can be modeled by assigning a weight to each transition. This weight can, for example, be an integer which is large if the violation corresponding to the sub-ideal behavior is serious. Given this partition, in ideal and varying sub-ideal transitions, the new problem is how to compare executions. A simple solution is to say that a *sequence of transitions* s is preferred to a second *sequence of transitions* s' iff the *sum of weights* of the transitions of s is less than that of the transitions of s' . This preference relation is noted \geq_p . Intuitively, the weights represent fines for the violations and the preference relation prefers a minimal total sum of fines. For example, the weight function of our example can be given by $\langle w(t_1) = 10, w(t_3) = 1, w(t_4) = 1 \rangle$ which states that damaging a book is ten times worse than returning a book one week too late. We have $\langle t_3, t_5 \rangle >_p \langle t_1, t_6 \rangle$, which is intuitively correct.

However, even with this definition some problems subsist. For example, the two executions $\langle t_1, t_2, t_5 \rangle, \langle t_1, t_6 \rangle$ are equivalent for \geq_p . This is not intuitively correct, because, we want to prefer executions where one repairs a damaged book to executions where one does

⁴ Another often used solution is to define a subset ordering on the sub-ideal transition, see [TT94a].

not repair it. These preferences are related to deontic notion of *Contrary-To-Duty (CTD)* obligations (see chapter 4, definition 4.3). To deal with this kind of transitions, which we call *repairing transitions* of sub-ideal behavior, we define another subset of transitions, that contains the repairing transitions of sub-ideal behaviors and negative weights are assigned to the transitions of this set. Intuitively, the negative fines can be considered as rewards for good behavior. This preference ordering is presented formally in the following section.

The extended fine system

An extended Petri net is a Petri net with varying sub-ideal and repairing transitions.

Definition 5.3 (Extended Petri net).

Let $N = \langle P, T, Pre, Post \rangle$ be a Petri net and Z the set of integers. An extended Petri net $EN = \langle P, T, S, R, w, Pre, Post \rangle$ is the extension of N with two disjoint sets $S \subseteq T$ and $R \subseteq T$ that represent sub-ideal and repairing behavior respectively, and the fine function $w: T \rightarrow Z$, defined as follows :

$$\begin{cases} t \in T / \{S \cup R\}: w(t) = 0 \\ t \in S: w(t) > 0 \\ t \in R: w(t) < 0 \end{cases}$$

A repairing transition is a transition that has a negative weight in order to recover from a sub-ideal situation that was brought by sub-ideal behavior. We can now give a formal definition of the extended relation on executions.

Definition 5.4 (Preference ordering on executions).

Let $EN = \langle P, T, S, R, w, Pre, Post \rangle$ be an extended Petri net, and M_1, M_2 two markings, $\geq_p: E(EN, M_1, M_2) \times E(EN, M_1, M_2)$ a preference relation defined on the set $E(EN, M_1, M_2)$ of the possible executions of EN from the marking M_1 to the marking M_2 , $s_1, s_2 \in E(EN, M_1, M_2)$ two executions and the function $lg(s)$ the length of the tuple $s = \langle s_1, \dots, s_m \rangle$ (here equal to m). s_1 is preferred to s_2 , written $s_1 \geq_p s_2$, iff :

$$\sum_{i=1}^{lg(s_1)} w(s_{1,i}) \leq \sum_{j=1}^{lg(s_2)} w(s_{2,j})$$

Definition 5.5 (Equivalent executions for the ordering).


Let $EN = \langle P, T, S, R, w, Pre, Post \rangle$ be an extended Petri net, and M_1, M_2 two markings, two executions $t_1, t_2 \in E(EN, M_1, M_2)$ are equivalent for the relation \geq_p , iff $(t_1 \geq_p t_2) \wedge (t_2 \geq_p t_1)$.

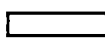
The preference ordering gives preferred executions.


Definition 5.6 (Preferred execution).

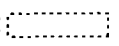
Let $EN = \langle P, T, S, R, w, Pre, Post \rangle$ be an extended Petri net, and M_1, M_2 two markings, $s \in E(EN, M_1, M_2)$ an execution from the marking M_1 to the marking M_2 . The execution s is a preferred execution from M_1 to M_2 iff : $\forall s' \in E(EN, M_1, M_2) : s \geq_p s'$

In the following example, we use some new graphical notations in addition to the usual notations of the Petri net formalism.

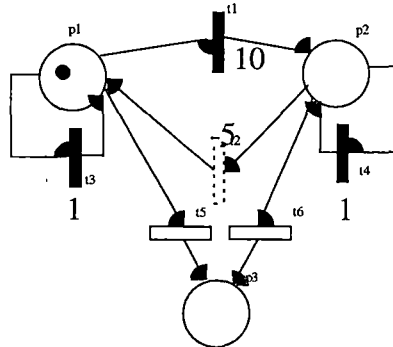
A place is represented in the usual way by :  p_i

If $t_i \in T \setminus \{S \cup R\}$, t_i is represented in the usual way by :  t_i
(the weight is not represented, because it is always equal to zero).

If $t_j \in S$, t_j corresponds to a sub-ideal behavior and is represented by :  t_j
with $w(t_j) \geq 0$.

If $t_k \in R$, t_k corresponds to a repairing transition and is represented by :  t_k
with $w(t_k) \leq 0$.

As a illustration of extended Petri nets, consider the extension of the example in figure 5.5 :



with $S = \{t_1, t_3, t_4\}, R = \{t_2\}$, and the definition of the fine system w :
 $\langle w(t_1) = 10, w(t_2) = -5, w(t_3) = 1, w(t_4) = 1 \rangle$
 p1: borrowed book; p2: damaged book; p3: returned book; t1: To damage the book;
 t2: To repair the book; t3: 1 week too late; t4: 1 week too late; t5: To return the book;
 t6: To return the book;

Figure 5.6 (Extended book borrowing model)

With our extended fine system w , we obtain the intuitive result : $\langle t_1, t_2, t_5 \rangle >_p \langle t_1, t_6 \rangle$. Furthermore, we have that $\langle t_5 \rangle >_p \langle t_1, t_2, t_5 \rangle$. This states that we prefer a borrower not to damage the book even if he repairs it.

The fine function given in definition 5.3 is rather basic, we could impose extra constraints on it in order to get certain desirable properties. For example, we could argue that the fine function should reflect the property that it is better not to do a sub-ideal behavior than doing it first and repairing it afterwards. A simple way to obtain this property is to impose on the fine function the following constraint : if transition t_1 represents a sub-ideal behavior and transition t_2 repairs the sub-ideal behavior t_1 , then $-w(t_2) < w(t_1)$.

Preferred reachable states

The preferences on transitions model what ought to be done. Besides these *ought-to-do obligations* also *ought-to-be obligations* can be defined, which are preferences on the markings (the states). Preferences on markings can be derived from preferences on transitions, or vice versa. In this section we show how preferences on markings can be derived in our extended Petri nets. We have decided to define the preference relation on the transitions because it is more expressive : two transitions between the same place can have different preferences. The preference relation on markings is defined on all reachable markings.

Definition 5.7 (Reachable markings).

Let $EN = \langle P, T, S, R, w, Pre, Post \rangle$ be an extended Petri net with marking M , and T^* the set of all sequences that can be composed of transitions of T . The set of reachable markings of the marked net $\langle EN, M \rangle$ is $R(EN, M) = \{ \bar{M} : \exists s \in T^*, M \xrightarrow{s} \bar{M} \}$.

A reachable marking M_1 is preferred to a second reachable marking M_2 , iff a preferred execution, which leads from M to M_1 , has a weight less than the weight of a preferred execution which leads from M to M_2 (where M is the initial marking).

Definition 5.8 (Preference ordering markings).

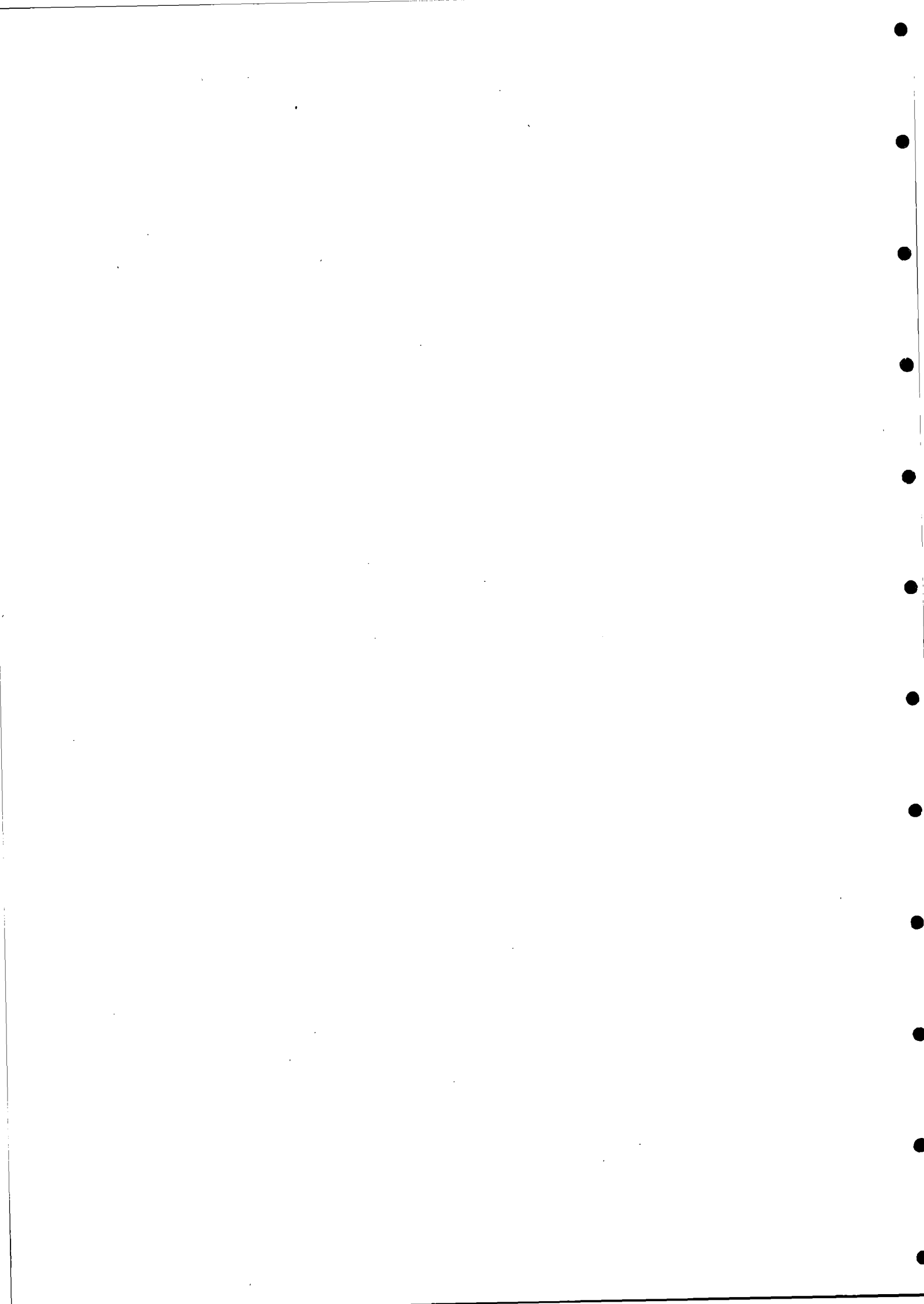
Let $EN = \langle P, T, S, R, w, Pre, Post \rangle$ be an extended Petri net with marking M , $s_1, s_2 \in E(EN, M_1, M_2)$ two executions, $\geq_M : R(EN, M) \times R(EN, M)$ a preference relation defined on the set $R(EN, M)$ of the reachable markings of EN , $M_1, M_2 \in R(EN, M)$ two markings and $lg(s)$ the length of the tuple s . M_1 is preferred to M_2 , written $M_1 \geq_M M_2$, iff :

$$\sum_{i=1}^{lg(s_1)} w(s_{1,i}) \leq \sum_{j=1}^{lg(s_2)} w(s_{2,j})$$

Where s_1 is a preferred execution of $\langle EN, M \rangle$ to $\langle EN, M_1 \rangle$ and s_2 is a preferred execution of $\langle EN, M \rangle$ to $\langle EN, M_2 \rangle$

5.4 Conclusion

In this chapter, we have defined in section 5.2. conditions that a Petri net must fulfil to model strict obligations and prohibitions. In this approach, the modeler must find a solution to respect those conditions. The limitations of this approach have been underlined and a better solution is proposed in chapter 6. In section 5.3., we have extended the Petri net formalism with a relation of preference on the possible executions of a net. This relation of preference and the definition of a fine system make possible the formalization of the distinction between ideal and varying sub-ideal behaviors.



Chapter 6

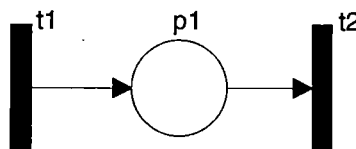
Petri Nets and Temporal Logic

6.1 Introduction

The extension of the Petri net formalism we propose to define here, is based on the following observation : it's easy to represent the arguments of a transition in the Petri net formalism, on the other hand, it is not easy to construct a Petri net that respects complex (temporal) non-operational constraints.

The idea to avoid this difficulty, is to give the possibility to the analyst to add temporal formulae to its Petri net models in order to limit their possible executions. As an illustration of this concept, consider the following example :

Example 6.1 : (Modeling a simple producer - consumer system with one buffer).



t_1 : the producer; t_2 : the consumer; p_1 : the buffer.

Figure 6.1 (a simple producer-buffer-consumer model)

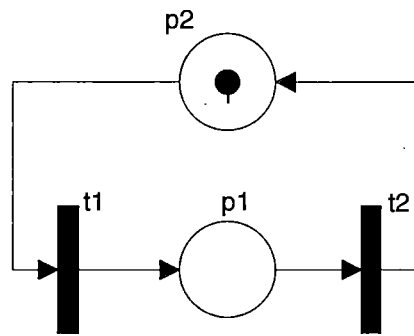
These are some possible executions of this model:

(e₁) $t_1 t_1 t_1 t_2 t_1 t_2 t_2 \dots$

(e₂) t₁t₂t₁t₂t₁t₂t₁t₂t₁t₂...

...

Now if we add as constraint that the buffer can only contain one token, the execution (e₁) is no longer valid. To model this constraint in the Petri net formalism, we may add a complementary place to place p₁. Then we obtain the following model :



t₁: the producer; t₂: the consumer; p₁: the buffer;
p₂ : complementary¹ place of p₁.

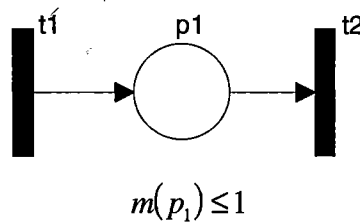
Figure 6.2 (a simple producer-binary buffer-consumer model)

Note that the set of possible executions of the Petri net of figure 6.2 is only the execution t₁t₂t₁t₂t₁t₂t₁t₂t₁t₂... Execution which verifies the additional constraint. But the place p₂ is an artificial element (over specification), a construction that implements the supplementary constraint. It does not represent a specification of the constraint ! It's a reason why we think that the original Petri net formalism is not a good specification formalism due to its over operational style [PETERSON81]. So, if we want to keep Petri nets as a specification language, because, for instance, some constraints are easily expressed in operational style, we have to extend the net formalism in order to make possible declarative specifications of non-operational constraints. Our solution is to associate Petri nets and temporal logic formulae. The logic formulae will be used to express constraints which can not be easily expressed in an operational style. Our choice is motivated by the great expressiveness of temporal logic and the easy link that can be made between the operational semantics of Petri nets expressed as infinite sequences of states (see definition 6.8) and the semantics of temporal logic specifications also expressed as infinite sequences of states (see definition 2.5).

This is the equivalent specification² of the model of figure 6.2, expressed in our language :

¹ Complementary because p₂ is marked when p₁ is not and inversely.

² We now use the term specification because we do not give a solution for the implementation of the supplementary constraint.



t_1 : the producer; t_2 : the consumer; p_1 : the buffer.

Figure 6.3 (a simple producer-binary buffer-consumer specification)

The formula $m(p_1) \leq 1$ restricts the set of possible executions of the net to executions in which the marking of place p_1 is one or zero.

As we can see in this trivial example, the semantics of the formulae can easily be given as the reduction of the set of possible executions of the net. To define in a formal manner the additional language, we first give the operational (behavioral) semantics of the original Petri net model.

6.2 Operational semantics of the Petri nets

In the literature three styles of semantics are distinguished : (1) operational semantics, (2) axiomatic semantics and (3) denotational semantics. Since the Petri net formalism is operative, the most natural way to define its semantics is to use operational semantics. The operational semantics of the Petri net model is given by means of a *transition system*. In the sequel we use usual definitions and notations of the Petri net formalism. These notions and definitions have been introduced in part one of this document.

Definition 6.1 (Transition system).

A transition system is a pair $\langle S, R \rangle$, where S is a set, called the state space and $R \subseteq S \times S$ is a relation called the transition relation.

A Petri net can be considered as a transition system where S , the state space, is the set of reachable markings $R(N, M)$ (see definition 1.5) and the transition relation is the following set :

$$\{ \langle M, M' \rangle : M \in R(N, M) \wedge M' \in R(N, M) \wedge \exists t \in T : M \xrightarrow{t} M' \}.$$

The process of a transition system starting at an initial state s (the initial marking for a Petri net) is described by the set of all execution paths starting at s . These execution paths

represent all possible "behaviors" of the transition system. An execution path is a maximal³ sequence of states (see definition 6.8) such that for any successive pair, their markings belong to the transition relation. An execution path starting in a marking M can also be seen as a sequence of transitions since the sequence of states can be computed from the sequence of transitions as firing a transition, in the Petri net formalism, is deterministic.

To formalize the operational semantics of a marked Petri net in terms of its possible executions (behaviors), we recall here some important definitions of the Petri net formalism⁴ and introduce some new definitions :

Definition 6.2 (Sequence of transitions).

Let us consider T as an alphabet composed of the symbols t of the transitions of Petri net N . We will write T^∞ the set of words that can be formed by concatenation of symbols of T . In that way, every sequence of transitions of the net N can be represented by a word of T^∞ .

Definition 6.3 (Firing a sequence of transitions).

A sequence of transitions s of T^∞ is fireable in the marked net (N, M) which will be written $M \xrightarrow{s}$ and the firing of the sequence s leads the net N to the marking M' , which will be written $M \xrightarrow{s} M'$, if and only if :

1. either $s = \lambda$ (the empty sequence), then $M = M'$.
2. or $s = s't$, with $s' \in T^\infty$ et $t \in T$, then : $\exists M' : M \xrightarrow{s} M' \wedge M' \xrightarrow{t} M'$.

The set of possible executions of a marked Petri net (its process) can thus be defined in terms of fireable sequences of transitions.

Definition 6.4 (The process of a Petri net).

The set of possible executions of a Petri net $N = \langle P, T, Pre, Post \rangle$ from a marking M is defined by the set : $E(N, M) = \{s \in T^\infty : M \xrightarrow{s}\}$

Definition 6.5 (Operational semantics of marked Petri net).

The semantics of a marked Petri net $\langle N, M \rangle$ is the set of all its possible executions $E(N, M)$.

As we mentioned in the introductory section, our idea is to use temporal logic formulae in order to reduce the set $E(N, M)$, representing the possible executions of the net, to a set of **desired executions**. In the next section, we define a language for the expression of properties

³Because when there is at least an enabled transition in a state, the transition must fire. Thus an execution is either infinite or finite and its last state is a terminal state (a state with a terminal marking see definition 6.7)

⁴Some of these definitions have already been introduced in part one but, for reasons of readability, we rewrite them here.

that desired executions must fulfill. For reasons of convenience⁵, we consider, in the following, an execution of a Petri net as an infinite sequence of states as in [MP92]. Each state is composed of a marking and an exiting transition.

Definition 6.6 (Execution State).

An execution state is a 2-tuple $\langle m, e \rangle$ where $m(S)$ is the marking of the state S , $m(S, p)$ represents the marking of the place p in state S and $e(S)$ is the transition which is fired in state S , it is the exiting event of the state S .

In the original Petri net model, executions are not necessarily infinite, some reachable markings may be *terminal markings*, i.e., have no enabled transition.

Definition 6.7 (Terminal marking).

A marking M is called a *terminal marking* for the Petri net $N = \langle P, T, Pre, Post \rangle$ iff $\neg \exists t \in T : \forall p \in P, m(p) \geq Pre(p, t)$. There is thus no enabled transition in a terminal marking.

To keep our definition of state (with an exiting transition), we introduce a new transition, the *null transition*. This *null transition can only and must* be fired in **terminal** states. The firing of the *null transition* leads in a state with the same marking and thus with the *null transition* as exiting transition... Let us give a more formal definition of the set of possible behaviors as a set of infinite sequences of states :

Definition 6.8 (Infinite sequences of states).

The set of possible executions of a marked Petri net $\langle N, M \rangle$ where $N = \langle P, T, Pre, Post \rangle$ is the set of infinite sequences of states $E_{inf}(N, M)$ such that : $\forall s \in E_{inf}(N, M) :$

- $s \in \langle R(N, M), T \cup \{null\} \rangle^\infty$. In other words, each sequence s of $E_{inf}(N, M)$ is an infinite sequence of execution states.
- $\exists \sigma \in E(N, M) : \forall i : 1 \leq i \leq lg(\sigma)$
 $e((s, i)) = (\sigma, i)$ ⁶. In other words, for each infinite sequence s there exists a firing sequence σ and the exit event of an execution state is the transition that is fired in this state.
 If σ is not infinite (end in a terminal state) then $\forall i : i > lg(\sigma) : e((s, i)) = null$.
 The sequence is artificially made infinite by the firing of the null transition.
- $m((s, 1)) = M$. The initial marking of each sequence s is the initial marking of the net.

⁵ In order to keep the semantics of the temporal operators introduced in chapter 2.

⁶ (s, i) denotes the i^{th} state of the sequence s .

$\forall i: 1 < i:$

if $e((s, i - 1)) = t \wedge t \neq \text{null}$
 then $\forall p: m((s, i), p) = m((s, i - 1), p) - \text{Pre}(p, t) + \text{Post}(p, t)$
 else $\{e((s, i - 1)) = \text{null}\} \forall p: m((s, i), p) = m((s, i - 1), p)$

In other words, the marking between two successive states is changed by firing a transition of the net or stay the same if the null transition is fired.

This representation of the possible behaviors of a Petri net is kept in the sequel of this document.

6.3 The logic formulae of the language

As mentioned in the introduction of this chapter, the Petri net formalism is more a modeling language than a specification⁷ language. To represent some (temporal) constraints, we have to find a solution, an implementation of them in the Petri net formalism. The Petri net language style is operative and not declarative. So, to use the Petri net formalism as a basis of a specification language, we have to introduce the possibility to add temporal formulae to a net in order to specify, in a declarative style, the (temporal) constraints that can not be specified easily in the net formalism. The semantics of the logic formulae that will accompany the nets, will be given as a reduction of the possible executions of the nets.

Considering a Petri net as a generator of a set of possible executions, we expect that the temporal logic formulae should provide an alternative characterization, more descriptive and less operational, of the desired set of executions of the net. The temporal logic formulae will express predicates over infinite sequences of states. Thus, each formula of temporal logic is satisfied by some sequences and falsified by some other sequences. We will restrict the executions of a net with temporal formulae to the executions of the net that satisfy all the temporal formulae. In the sequel we will first define the syntax and the semantics of logic formulae that do not contain temporal operators. Those formulae are interpreted in a state. The logic formulae that contain temporal operators and that will be introduced after, are interpreted in a particular state of a sequence.

6.3.1 State formulae

A state formula is evaluated at a certain position in a sequence and the formula expresses properties of the state occurring at this position. We will thus introduce a state language that

⁷ By *specification* we mean the description of the desired behavior of the system, while avoiding references to the method or details of its implementation.

permits us to express properties of Petri net states. As we have shown in the previous section, an execution of a Petri net can be represented by a sequence of states, composed of a marking and an exiting transition. Thus, we must be able to speak about the number of tokens in a place (the marking of the place) and over the transition which is fired in the state (the exiting transition). It can also be interesting to declare sub-sets of the set of transitions T . It can be convenient to declare a set which contains all the transitions of a part of the modeled system and to express properties over the transitions of this set. So we will give the possibility of such declarations and also the possibility of quantification on these sets. We will first give the syntax of the state language and thereafter its semantics.

Syntax of the state formulae

SYMBOLS :

The characters A, B, \dots, Z and symbols $=, \{, \}$, for the declaration of *sub-sets* of the set T of the transitions⁸.

The function m : this function $m: P \rightarrow \mathbb{N}$ is defined on the set of places of the Petri net and returns a natural number. $m(p_i)$ gives the number of tokens contained in the place p_i .

The predicate "*Fired*" : this predicate is defined on the set T of the transitions of the Petri net. $Fired(t_i)$ expresses the fact that the transition t_i is the exiting transition of the state under consideration. As two transitions can not fire in the same state, we have the following axiom :

$$\forall t_1, t_2: Fired(t_1) \wedge Fired(t_2) \Rightarrow t_1 = t_2$$

The predicate "*Enabled*" : this predicate is defined on the set T of the transitions of the Petri net. $Enabled(t_i)$ expresses the fact that the transition t_i is enabled in the state under consideration. As in the Petri net formalism a transition must be enabled to fire, we have the following axiom :

$$\forall t: Fired(t) \Rightarrow Enabled(t)$$

The usual boolean connectors : $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$.

The predicate symbols over positive integer numbers : $=, <, \leq, \geq, >, \neq$.

The usual addition operator over positive integer numbers : $+$

⁸The declaration of the *subsets* of the set T are not *formulae*. The utility of such subset declarations is illustrated at the end of this chapter in the case study.

The usual quantifications symbols \forall, \exists for quantification on the set T of transitions and declared sub-sets of T .

Two punctuation symbols : (,).

FORMATION RULES OF THE FORMULAE :

$$\begin{aligned}
 \langle \text{constant_place} \rangle &::= \langle p_1, p_2, \dots, p_n \rangle & 9 \\
 \langle \text{constant_transition} \rangle &::= \langle t_1, t_2, \dots, t_m, \text{null} \rangle \\
 \langle \text{variable_place} \rangle &::= \langle p_a, p_b, \dots, p_z \rangle \\
 \langle \text{variable_transition} \rangle &::= \langle t_a, t_b, \dots, t_z \rangle \\
 \langle \text{variable} \rangle &::= \langle \text{variable_place} \rangle \mid \langle \text{variable_transition} \rangle \\
 \langle \text{Subset_of_transitions} \rangle &::= \langle A..Z \rangle \\
 \langle \text{Subset_declaration} \rangle &::= \\
 &\quad \langle \text{Subset_of_transitions} \rangle' = \langle \{ \langle \text{Constant_transition} \rangle, * \} \rangle \\
 \langle \text{integer_term} \rangle &::= \\
 &\quad \langle \text{positive integer} \rangle \\
 &\quad \mid \langle m(\langle \text{variable_place} \rangle) \rangle \\
 &\quad \mid \langle m(\langle \text{constant_place} \rangle) \rangle \\
 &\quad \mid \langle \text{term} \rangle' + \langle \text{term} \rangle \\
 \langle \text{atomic formula} \rangle &::= \\
 &\quad \langle \text{Fired}(\langle \text{variable_transition} \rangle) \rangle \\
 &\quad \mid \langle \text{Fired}(\langle \text{constant_transition} \rangle) \rangle \\
 &\quad \mid \langle \text{Enabled}(\langle \text{constant_transition} \rangle) \rangle \\
 &\quad \mid \langle \text{Enabled}(\langle \text{variable_transition} \rangle) \rangle \\
 &\quad \mid \langle \text{term} \rangle' =, <, \leq, \geq, >, \neq \langle \text{term} \rangle
 \end{aligned}$$

⁹ The constants are thus the symbols that represent the name of the places and the transitions of the net.

$$\langle \text{state formula} \rangle ::=$$

$$\begin{array}{|l} \langle \text{atomic formula} \rangle \\ \langle \text{state formula} \rangle \wedge, \vee, \neg, \rightarrow, \leftrightarrow \langle \text{state formula} \rangle \\ \langle \text{logic formula} \rangle \\ \forall, \exists \langle \text{variable list} \rangle \in \langle \text{Subset_of_transitions} \mid T \rangle : \langle \text{logic formula} \rangle \end{array}$$

Semantics of states formulae

Next, we consider the semantics of the different constructions, showing how to evaluate them over states. We will note $[\varphi]_{S=\langle M,t \rangle}$ the truth value of the state formula φ in the state $S = \langle M, t \rangle$ where M is the marking of the state S and t the exiting transition of this state. We will note $I(t)$ the interpretation of the *term* t by the *interpretation function* I . We first define an interpretation function for our logic formulae.

Definition 6.9 (The interpretation function I).

- I assigns a value in the set T of transitions to all constant_transitions and all free variable_transitions. We restrict here to interpretation functions that map the constant_transitions in the following way: $I(t_i) = t_i$.
- I assigns a value in the set P of places to all constant_places and all free variable_places. We restrict ourselves here to interpretation that maps the constant_places in the following way: $I(p_i) = p_i$.
- I maps the functor m to the application M that represents the marking function.

Interpretation of a term :

- $I(n) = n$ where n is a positive integer number.
- $I(m(p)) = M(I(p))$ where p is a variable_place.
- $I(m(p)) = M(p)$ where p is a constant_place.
- $I(t_1 + t_2) = I(t_1) + I(t_2)$ where t_1 and t_2 are terms.

Semantics of the predicates over terms :

- $[t_1 = t_2]_{S=\langle M,t \rangle, I}$ is true iff $I(t_1) = I(t_2)$ (Where t_1 and t_2 are terms).
- $[t_1 \geq t_2]_{S=\langle M,t \rangle, I}$ is true iff $I(t_1) \geq I(t_2)$ (Where t_1 and t_2 are terms).
- $[t_1 > t_2]_{S=\langle M,t \rangle, I}$ is true iff $I(t_1) > I(t_2)$ (Where t_1 and t_2 are terms).

- $[t_1 < t_2]_{S=(M,t),I}$ is true iff $I(t_1) < I(t_2)$ (Where t_1 and t_2 are terms).
- $[t_1 \leq t_2]_{S=(M,t),I}$ is true iff $I(t_1) \leq I(t_2)$ (Where t_1 and t_2 are terms).
- $[t_1 \neq t_2]_{S=(M,t),I}$ is true iff $I(t_1) \neq I(t_2)$ (Where t_1 and t_2 are terms).

Semantics of the predicates "Fired" and "Enabled" :

- $[Fired(a)]_{S=(M,t),I}$ is true iff $I(a) = t$.
- $[Enabled(a)]_{S=(M,t),I}$ is true iff $\forall p \in P: Pre(p, I(a)) \leq M(p)$.

Semantics of the usual boolean connectors :

- $[\vartheta_1 \vee \vartheta_2]_{S=(M,t),I}$ is true iff $[\vartheta_1]_{S=(M,t),I}$ is true or $[\vartheta_2]_{S=(M,t),I}$ is true.
- $[\neg\vartheta]_{S=(M,t),I}$ is true iff $[\vartheta]_{S=(M,t),I}$ is false.
- The semantics of the other boolean connectors is deduced in the usual way, see definition 2.1.

Semantics of the quantifiers :

- $[\exists x \in Z: \vartheta]_{S=(M,t),I}$ is true iff $\exists t_i \in Z: [\vartheta: t_i]_{S=(M,t),I}$ is true where $\vartheta: t_i$ is the formula obtained by replacing the occurrences of variable x by the value t_i and Z is a declared subset of the set T of transitions or the set T .
- $[\forall x \in Z: \vartheta]_{S=(M,t),I}$ is true iff $\forall t_i \in Z: [\vartheta: t_i]_{S=(M,t),I}$ is true where $\vartheta: t_i$ is the formula obtained by replacing the occurrences of variable x by the value t_i and Z is a declared subset of the set T of transitions or the set T .

6.3.2 Temporal formulae

A temporal formula is constructed from state formulae to which we apply temporal operators and boolean connectives. We will use and adapt the syntax and the semantics of future and past operators of common linear temporal logic. These operators are very similar to those used in ALBERT [DDDP94b] and in OBLOG [SCS92],[SGS92]. As for the state formulae, we will first define the syntax of the temporal formulae and then their semantics.

Syntax of temporal formula

As usual in temporal logic, in the sequel, X is the "next" operator, F is the "eventually"¹⁰ operator, G is the "henceforth"¹¹ operator and U is the "until" operator. These operators

¹⁰ Also often called the "sometimes" operator.

represent the future operators of our language. The past operators are: Y the "previous state" operator, P the "sometimes in the past" operator, H the "always in the past" operator and S the "since" operator.

Syntactical formation rule :

$$\langle \text{Temporal_formula} \rangle ::=$$

$$\begin{aligned} & \langle \text{State_formula} \rangle \\ & 'X' \langle \text{Temporal_formula} \rangle \\ & 'F' \langle \text{Temporal_formula} \rangle \\ & 'G' \langle \text{Temporal_formula} \rangle \\ & \langle \text{Temporal_formula} \rangle 'U' \langle \text{Temporal_formula} \rangle \\ & 'Y' \langle \text{Temporal_formula} \rangle \\ & 'P' \langle \text{Temporal_formula} \rangle \\ & 'H' \langle \text{Temporal_formula} \rangle \\ & \langle \text{Temporal_formula} \rangle 'S' \langle \text{Temporal_formula} \rangle \\ & '¬' \langle \text{Temporal_formula} \rangle \\ & \langle \text{Temporal_formula} \rangle ' \wedge, \vee, \rightarrow, \leftrightarrow ' \langle \text{Temporal_formula} \rangle \\ & ' \forall, \exists ' \langle \text{variable_transition} \rangle ' \in ' \langle \text{Set_of_transitions} \rangle ' : ' \langle \text{Temporal_formula} \rangle \end{aligned}$$

Semantics of temporal formulae

As the truth value of a state formula is evaluated in a state, the truth value of a temporal formula is evaluated in a state which belongs to a sequence of states (see section 2.1. Temporal logic). Let us first define some notations :

Notations (The truth value of a temporal formula).

A temporal formula is evaluated at a *position* i of an infinite sequence of states (like those of set $E_{inf}(N,M)$), we will note *the state of position i in the infinite sequence s* as (s,i) . The future of the state (s,i) is the suffix of sequence s that starts in position i (thus including the present), we will note it $(s, i..+\infty)$. Symetrically, the past of the *state* i is constituted of the prefix of the sequence s that ends in *state* i (thus including the present), we will note it $(s, 1..i)$. The *truth value of temporal formula* ϑ in position (s,i) will be noted: $[\vartheta]_{(s,i)}$.

Let's now give the semantics of the temporal formulae.

¹¹ Also often called the "always" operator.

The future operators

X next operator: $[X\vartheta]_{(s,i)}$ is true iff $[\vartheta]_{(s,i+1)}$ is true.

F eventually operator: $[F\vartheta]_{(s,i)}$ is true iff $\exists j:(i \leq j), [\vartheta]_{(s,j)}$ is true.

G henceforth operator: $[G\vartheta]_{(s,i)}$ is true iff $\forall j:(i \leq j), [\vartheta]_{(s,j)}$ is true.

U until operator: $[\vartheta U \phi]_{(s,i)}$ is true iff

$$\left\{ \begin{array}{l} \exists j:i \leq j: (\forall k:i \leq k < j: [\vartheta]_{(s,k)} \text{ is true}) \\ \wedge ([\phi]_{(s,j)} \text{ is true}) \end{array} \right.$$

The past operators

Y previous state operator: $[Y\vartheta]_{(s,i)}$ is true for $i > 1$ iff $[\vartheta]_{(s,i-1)}$ is true; $[Y\vartheta]_{(s,1)}$ is always true.

F_p sometimes in the past operator: $[F_p \vartheta]_{(s,i)}$ is true iff $\exists j:(1 \leq j \leq i), [\vartheta]_{(s,j)}$ is true.

G_p always in the past operator: $[G_p \vartheta]_{(s,i)}$ is true iff $\forall j:(1 \leq j \leq i), [\vartheta]_{(s,j)}$ is true.

S since operator: $[\vartheta S \phi]_{(s,i)}$ is true iff

$$\left\{ \begin{array}{l} \exists j:1 \leq j \leq i: (\forall k:j < k \leq i: [\vartheta]_{(s,k)}) \\ \wedge ([\phi]_{(s,j)} \text{ is true}) \end{array} \right.$$

With this semantic of *Y* operator, we can define a new predicate : *init* which is only true in the initial state of the sequence, its formal definition : $init \equiv [Y \perp]$. This new predicate allows to write logic formulae which express properties over the initial marking of the Petri net.

6.3.3 The desired executions

We have defined the syntax and the semantics of the logic formulae that will accompany the Petri nets in our specification language. Let us now give a formal definition of the set of executions obtained by the conjunction of the Petri net and the logic formulae in a specification.

Definition 6.10 (Desired executions).

The desired executions of a marked Petri net $\langle N, M \rangle$ accompanied by a set of logic formulae Ψ is the set of possible executions of $\langle N, M \rangle$ that satisfy each formula of the set Ψ . We will note this set $E_r(N, M, \Psi)$:

$$E_r(N, M, \Psi) = \left\{ s: s \in E(N, M); \forall \vartheta \in \Psi, \forall i \in \mathbb{N}: \left[\vartheta \right]_{(s,i)} \text{ is true} \right\}$$

Definition 6.11 (Semantics of a PNTL specification).

The semantics of a PNTL specification constituted of a marked Petri net $\langle N, M \rangle$ and a set of logic fomulae Ψ , is the restricted executions that belongs to the set $E_r(N, M, \Psi)$.

Thanks to the new predicate *init*, it is possible to define properties that an initial marking must fulfil. So we can define the possible behaviors of a class of marked Petri nets whose initial marking fulfil initialisation properties. Let's note ξ the set of formulae of the form $(init \rightarrow \vartheta)$ that define the properties of the accepted initial markings ($\xi \subseteq \Psi$). Below we define the possible behaviors of the Petri net N with initial marking that fulfil ξ .

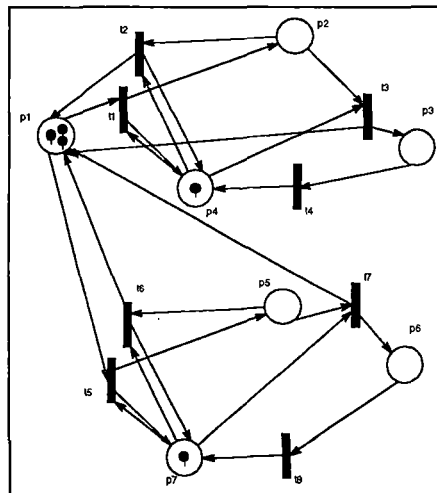
$$E_r(N, \xi, \Psi) = \left\{ \begin{array}{l} \forall M: [\xi]_{(s,1)=(M,t)} \text{ is true,} \\ s \in E_r(N, M, \Psi) \end{array} \right\}$$

Futher in this chapter we give some examples of specifications in PNTL. We will also show how logic formulae can be used to specify strict deontic aspects.

6.4 A specification in PNTL

Let us consider a small library that contains three books. We are here interested in the behavior of two borrowers b_1 and b_2 . Each of them may borrow books from the library but has to return the books in time or has to pay a fine before doing anything else in the library(c1).

6.4.1. A model of this case in the common Petri nets

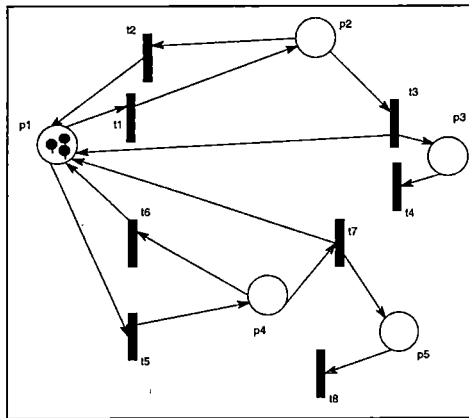


Legend: p1: Books free; p2: Books borrowed by b1; p3: Fines to pay; p4: implementation of constraint (c1); t1: b1 borrows a book; t2: b1 returns a book in time; t3: b1 returns a book to late; t4: b1 pay a fine; p5: Books borrowed by b2; p6: Fines to pay; p7: implementation of constraint (c1); t5: b2 borrows a book; t6: b2 returns a book in time; t7: b2 returns a book to late; t8: b2 pays a fine;

Figure 6.4 (a model of the library example)

The figure 6.4 and its legend is a model of our library case. It's not a specification because the place p_4 and p_7 are solutions (over specifications) for the realisation of the constraint c_1 . A proof of validity of the solution for the implementation of constraint c_1 is given in chapter 5.

6.4.2 The PNTL specification of the case



The logic formulae Ψ of the PNTL specification:

$$Fired(t_3) \rightarrow (\forall t \in \{t_1, t_2, t_3\}: \neg Fired(t)) U (Fired(t_4)) \quad (1)$$

$$Fired(t_7) \rightarrow (\forall t \in \{t_5, t_6, t_7\}: \neg Fired(t)) U (Fired(t_8)) \quad (2)$$

Legend: p1: Books free; p2: Books borrowed by b1; p3: Fines to pay; t1: b1 borrows a book; t2: b1 returns a book in time; t3: b1 returns a book to late; t4: b1 pay a fine; p4: Books borrowed by b2; p5: Fines to pay; t5: b2 borrows a book; t6: b2 returns a book in time; t7: b2 returns a book to late; t8: b2 pays a fine;

Figure 6.5 (a specification of the library example)

The first formula (1) says that the borrower b1 must pay a fine after returning a book late if he wants to do anything else afterwards, (2) says the same for b2. The formulae (1) and (2) are specifications of the constraint c_1 , they do not represent a solution in terms of places and transitions for the implementation of the constraint. Those constraints represent strict deontic aspects : the strict obligation to pay the fine in order to borrow again.

With the logic formulae, we can easily specify a large set of constraints. For example, the fact that a borrower can continue to return his books even if he has to pay a fine. In other

words, when he has to pay a fine, the only action he cannot undertake is borrowing a new book. This can be expressed by the following formula:

$$m(p_3) > 0 \rightarrow \neg \text{Fired}(t_1) \quad (3)$$

$$m(p_5) > 0 \rightarrow \neg \text{Fired}(t_5) \quad (4)$$

6.4.3 Evaluations

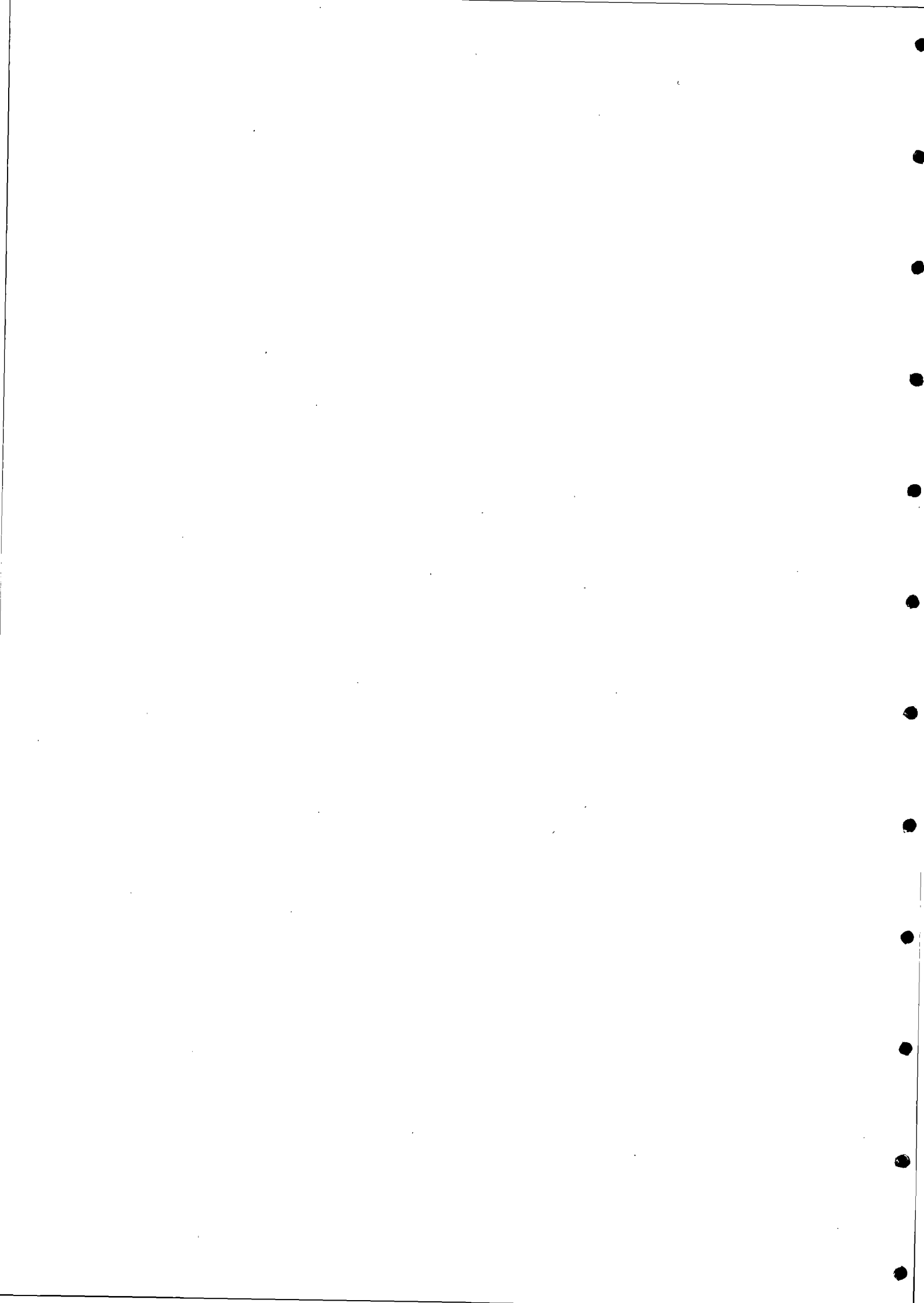
As we can see in the illustrative example, the addition of logic formulae to a Petri net model, allows to obtain :

- more readable specifications (compare figure 6.4 and figure 6.5)
- less operational specifications.

Futhermore, the strict obligations/prohibitions can now easily be specified, see assertion (1)-(4). Nevertheless, PNTL is rather basic :

- the tokens have no values
- the quantitative time properties cannot be expressed (real time)...

So in the following chapter we extend PNTL by defining a new formalism PNRTL. This new formalism will be based on real-time Predicate/Transition nets.



Chapter 7

The PNRTL language

7.1 Motivations

In the previous chapter, we have defined a language based on the formalisms of Petri nets and temporal logic. This language (PNTL) allows to specify in a declarative and also in an operational style. But its expressiveness should be extended. Recall that the nets used in PNTL are basic Petri nets, the tokens which appear in the nets have no value (no color). Again recall that the temporal operators used in the logic formulae are usual temporal operators, real-time aspects cannot be expressed.

Since our objective is to define an expressive language for the specification of distributed systems with real-time and deontic features, we extend here PNTL in four directions. First, we introduce a net formalism that allows valued tokens. Our formalism is based on Predicate/Transition nets [GENRICH86], see also section 1.4. Our choice is motivated by the close relation which exists between Pr/Tr nets and first order logic. With these Pr/Tr nets the link between the net concepts and the logic concepts can easily be defined. Secondly, we introduce the results of chapter 5 in our net definitions to allow the specification of the deontic distinction between ideal and sub-ideal behavior. Thirdly, we augment PNTL with real-time features. Finally, to allow more structured specifications, we introduce types and the possibility to specify a system in separate sub-nets.

To present these extensions of PNTL in a clear way, we structure this chapter in five other sections. Section 7.2 introduces new notions that are related to the notion of dynamic first

order structure. Section 7.3 introduces the new net formalism of the specification language including its operational semantics. This semantics is still expressed in terms of possible sequences of states and reflects now real-time features. Section 7.4 introduces the new logical constructs of the language. Section 7.5 presents how types are introduced in the language. Finally, section 7.6 shows how a specification can be structured in sub-nets.

7.2 New concepts

As pointed out in section 7.1, we want to extend the PNTL formalism to allow the association of values to tokens and in general to introduce the notion of individuals in our formalism. The work we accomplish here is very similar to the work necessary for going from propositional logic to first-order predicate logic.

Let us first introduce the notion of relational structure.

Definition 7.1 (Relational structure).

A relational structure is a tuple of objects $S = \langle D; f_1, \dots, f_k; R_1, \dots, R_n \rangle$ where D is a non-empty set of individuals called the domain of S , the f_i are functions in D and the R_i are relations in D .

A relational structure can describe situations where properties of individuals and relations between individuals are static. But we are interested in describing dynamic systems. In such systems, properties of individuals and relations between individuals may vary during the execution of the system. So we are interested in dynamic relational structure.

Definition 7.2 (Dynamic relational structure).

A dynamic structure will be characterised by the fact that some relations are variable in the sense that their extensions may vary from state to state due to the occurrence of processes (actions) in the modeled system.

We have chosen structured sets of individuals to support the modeling of dynamic systems. Operators (function symbols) and predicates (relation symbols) form the vocabulary of the language in which we will talk formally about structures, i.e. about properties and relations of individuals. The language we use is that of first order predicate logic :

Definition 7.3 (A language for structure).

Let for each $n \geq 0$, $\Omega^{(n)}$ be a set of n -ary operators and $\Pi^{(n)}$ a set of n -ary predicates. These operators and predicates form the vocabulary of the first order language L that consists of two kind of expressions, terms and formulae. In addition, there is a set of symbols, V , disjoint

from Ω (the set of operators) and from Π (the set of predicates), whose elements serve as (individual) variables. Terms and formulae are built in the following way :

1. Terms

- A variable is a term.
- If $f^{(n)}$ is a n -ary operator $\in \Omega^{(n)}$ and v_1, \dots, v_n are terms then $f(v_1, \dots, v_n)$ is a term. (Note that 0-ary operators are terms; they are used as proper names of distinct individuals).
- No other expressions is a term.

2. Formulae:

- If v_1, v_2 are terms then $v_1 = v_2$, is an atomic formula.
- If $P^{(n)}$ is a n -ary predicate $\in \Pi^{(n)}$ and v_1, \dots, v_n are terms then $P(v_1, \dots, v_n)$ is an atomic formula.
- If p_1, p_2 are formulae then $\neg p_1$ and $(p_1 \vee p_2)$ are formulae.
- If x is a variable and p a formula then $(\exists x)p$ is a formula.
- No other expression is a formula

Remark : The connectors $\wedge, \rightarrow, \leftrightarrow$ and \forall are derived from \neg, \vee and \exists in the usual way, see section 2.2.

We must still define some common notions of first order logic, those notions are constantly used in the sequel of this chapter.

Definition 7.4 (Free occurrence of a variable).

An occurrence of a variable x in a formula E is called free occurrence if it is not in the range of a $(\exists x)$ or $(\forall x)$. The occurrences of variables in a single term are free.

Definition 7.5 (Index of an expression).

The set of variables that occur freely in an expression (term or formula) is called the index of that expression.

Definition 7.6 (Closed expression).

An expression v is closed if its index is empty.

We can use a first-order language L for talking of a relational structure S if we associate with each operator and each predicate in the vocabulary of L a function respectively a relation of S .

Definition 7.7 (L-Structure).

Given a first-order language L , we call a structure S a structure for L , or L -structure, if every operator $f^{(m)}$ of L denotes a m -ary function of S designated by f_S and every predicate $P^{(n)}$ of

L denotes a n -ary relation of S designated by R_S . To ensure that each individual in the domain of S can be named in a sentence, we now add to the vocabulary of L a new set, U_S , of constants denoting the individuals of S in a one-to-one fashion¹. The individual denoted by a constant d is designated by d_S .

The structure S assigns to each closed term, v , of L_S an individual of S , designated by $S(v)$, and to each closed formula (proposition), p , of L_S , the truth value *true* or *false*, designated by $S(p)$.

Definition 7.8 (Substitution).

Let E be an expression (term or formula), x_1, \dots, x_n be variables, t_1, \dots, t_n be terms. Then $\gamma = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ is called a substitution, and $E:\gamma = E:\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ designates the result of substituting t_i for each free occurrence of x_i , for $1 \leq i \leq n$. $E:\gamma$ is called the γ -instance of E .

Definition 7.9 (Valuation).

A valuation α is a special kind of substitution $\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$ where all t_i ($1 \leq i \leq n$) are constants $\in U_S$.

Definition 7.10 (Interpretation function for a structure).

Let v be a closed term and p a closed formula of the language L_S . The $S(v)$ and $S(p)$ are defined recursively on their respective syntactic structure.

1. $S(v)$

- If v is a constant d , $S(v)$ is the individual denoted by d , d_S .
- If v is $f^{(n)}(v_1, \dots, v_n)$ then $S(v) = f_S(S(v_1), \dots, S(v_n))$.

2. $S(p)$

- If p is $v_1 = v_2$ then $S(p) = \text{true}$ iff $S(v_1)$ and $S(v_2)$ are the same individual.
- If p is $P^{(n)}(v_1, \dots, v_n)$ then $S(p) = \text{true}$ iff $\langle S(v_1), \dots, S(v_n) \rangle \in P_S$.
- If p is $(p_1 \vee p_2)$ then $S(p) = \text{true}$ iff $S(p_1) = \text{true}$ or $S(p_2)$ is true.
- If p is $\neg q$ then $S(p) = \text{true}$ iff $S(q) = \text{false}$.
- If p is $(\exists x)q$ then $S(p) = \text{true}$ iff there is a constant d such that $S(q:\{x \leftarrow d\}) = \text{true}$.

Remark : The semantics of the connectors $\wedge, \rightarrow, \leftrightarrow$ and \forall are derived from the semantics of \neg, \vee and \exists in the usual way, see section 2.1.

Recall that in an ordinary first-order structure S , all functions and relations are static, as opposed to dynamic structure where some relations are variable. The presentation of dynamic structures requires that we distinguish between predicates denoting static relations and

¹This is possible because we have decided that the domains of interpretation are fixed domains, see section 7.5.

predicates denoting variable relations. Hence we divide the set of predicates, Π , into a set of *static predicates* Π_s , and a set of *dynamic predicates* that will be designated by Π_d .

Definition 7.11 (Static formula).

A logic formula is a static formula iff it does not contain dynamic predicates.

In our specification language, we separate the static and the dynamic part from each other. The static part remains an ordinary relational structure. It is often called the **support of the system**. The dynamics are presented as **an annotated net and a set of real-time temporal formulae**. The variable relations (dynamic predicates) appear as the places of the net. (see chapter 1, section 1.4).

7.3 Real Time Pr/Tr Nets

7.3.1 Introduction

In this section we introduce the net formalism of our language. The net formalism is used to represent the **operative constraints** of the dynamic part of the system to be specified. As in Pr/Tr nets (see, section 1.4), the places are annotated by dynamic predicates and the tokens present in a place represent the current extension of the predicate that annotates the place. The transitions of the net model actions of the system.

The remainder of this section is organized in two sub-sections. Sub-section 7.3.2 considers the introduction of real-time features and the sub-section 7.3.3 formalizes the notion of PNRTL net and gives its firing rule and operational semantics.

7.3.2. Introduction of Real-Time

In this paragraph the main distinctions between Pr/Tr nets and real-time Pr/Tr nets are briefly pointed out :

- In usual Petri nets, a net execution can be seen as a sequence of states (see Definition 6.8, p 6.5). In real-time Pr/Tr nets, and this is new, every net state is mapped to two real instants of time : the time at which the net enters the state and the time the net quits the state.
- In usual Pr/Tr nets, firing a transition is atomic, i.e. all actions (represented by transitions) are instantaneous. In reality actions are either instantaneous or have a

duration of time. To model this property of actions, we consider that firing a transition is either instantaneous or has a duration. In the first case, firing remain an atomic action, in the second case, firing a transition is decomposed in two atomic events : the beginning and the end of the firing. Between those two events, the firing is said in progress.

- In real-time Pr/Tr nets, it seems necessary to refine the notion of transition (action). There is a differentiation between transitions and transition instances. The notion of *transition instance* can be used to express that a transition fires n times within a certain interval. As firing is no longer necessarily atomic and can have duration, many occurrences of a same transition may be in progress during the same period (this property is often called autoconcurrency).

Due to the introduction of real-time, it's necessary to give a new definition of a net state. This is a definition which supports the introduction of the real-time features :

Definition 7.12 (PNRTL state)

In order to include the real time features, a PNRTL state S will be composed of :

- **An entering time : $InTime(S)$.** *It represents the moment at which the system enters in state S .*
- **An exiting time : $OutTime(S)$.** *It represents the moment at which the system exits the state S . Let us note that the relation $InTime(S) \leq OutTime(S)$ is always verified.*
- **A marking : $M(S)$** *which represents the extension of each dynamic predicate that annotates the places of the net.*
- **A set of occurrences of actions : $H(S)$** *which are in progress during state S .*
- **An exiting event : $Exit(S)$** *is the event which makes change the state of the system. This event is either the beginning of an action (the firing of a transition) or the termination of an action.*

The fact that firing a transition is no longer necessarily atomic, causes a new problem. Let us consider the following part of a net as an illustration :

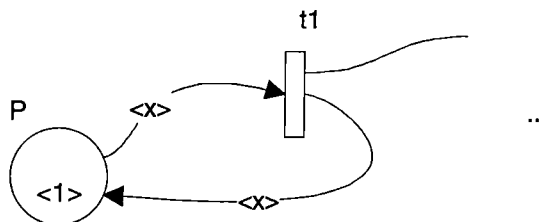


Figure 7.1 (A part of a Pr/Tr net)

The firing of t_i in an usual Pr/Tr net has no influence on the extension of P . In fact, the extension of the dynamic predicate P is not changed since the token $\langle 1 \rangle$ stays in P in the reached state since $\langle 1 \rangle$ is added by the output arc and firing is atomic. The construct of figure 7.1 (arc (1) and arc (2)) is often used to represent a precondition for the firing of a transition and this precondition is not modified by firing the transition. Unfortunately, this construct is no longer valid if the firing of t_i has a duration. In fact if we consider that when one fires a transition, at the beginning event tokens are removed from the input places and at the end event tokens are added to the output places, the firing of an occurrence of t_i must be ended before another occurrence may fire (if we make the supplementary assumption that tokens are only added to place P by t_i).

For that reason, we will introduce new types of arcs.

New types of arcs

The semantics of the new arcs is presented here in an informal way. A more formal presentation of their semantics (in terms of elements of a PNRTL state) is given in definitions 7.21, 7.23 and 7.24, commented examples of the use of the different arcs are given in chapter 8 (case study in PNRTL).

Input arcs :

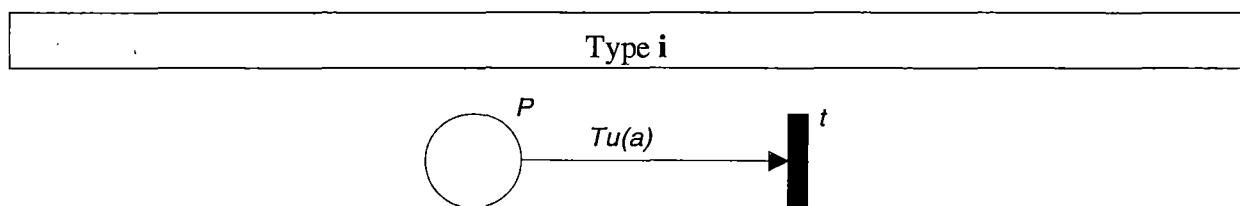


Figure 7.2 (Graphical representation of a *type i* arc)

This arc is enabled for an α -instance of t (where α is a valuation) if for all tuples of $Tu(a)^2$, the α -instance of the tuple belongs to the extension of the dynamic predicate P . These α -instances of the tuples are removed from P at the beginning of action t - α . This type of arc can be used to represent a positive precondition (ex: $P(x)$ must be true) for the beginning of an action and the precondition becomes false directly after the beginning of the action.

² $Tu(a)$ denotes the set of tuples which annotate the arc a.

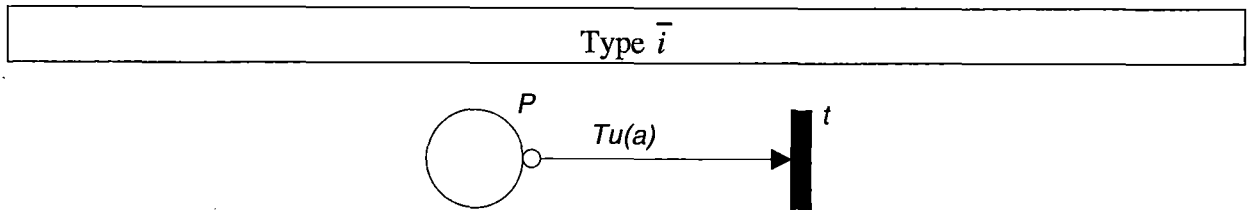


Figure 7.3 (Graphical representation of a type \bar{i} arc).

This arc is enabled for an α -instance of t (where α is a valuation) if for all tuples of $Tu(a)$, the α -instance of the tuple does not belong to the dynamic predicate P . These α -instances of the tuples are added to P at the beginning of action t - α . This type of arc can be used to represent a negative precondition (ex: $P(x)$ must be false) for the beginning of an action and the negative precondition becomes false, i.e. $P(x)$ is true, directly after the beginning of the action.

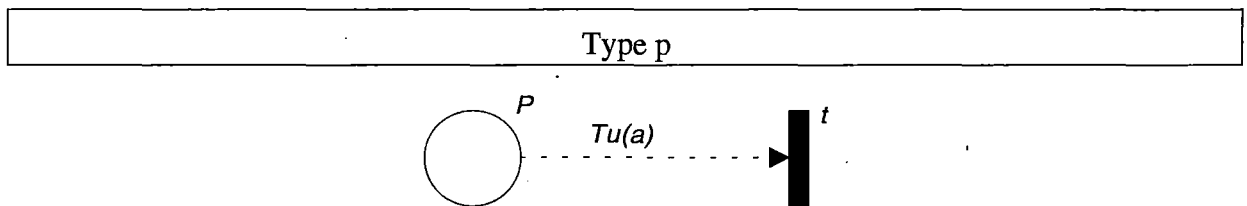


Figure 7.4 (Graphical representation of a type p arc)

This arc is enabled for an α -instance of t (where α is a valuation) if for all tuples of $Tu(a)$, the α -instance of the tuple belongs to the dynamic predicate P . These α -instance of the tuples are not removed from P at the beginning of action t - α . Here, the positive precondition is not modified at the beginning of the action.

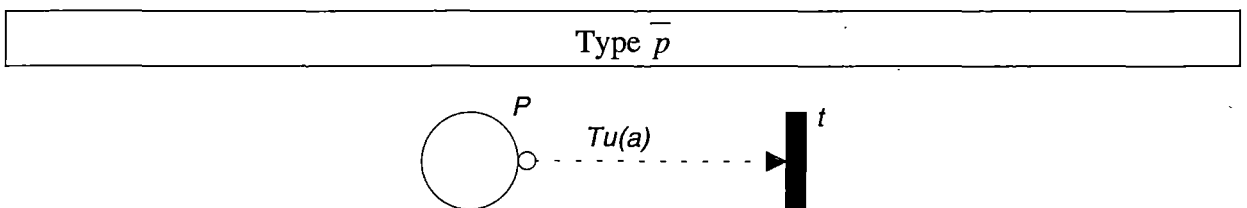


Figure 7.5 (Graphical representation of a type \bar{p} arc)

This arc is enabled for an α -instance of t (where α is a valuation) if for all tuples of $Tu(a)$, the α -instance of the tuple does not belong to the dynamic predicate P . These α -instances of tuples are not added to P at the beginning of action t - α . Here, the negative precondition is not modified at the beginning of the action.

Output arcs :

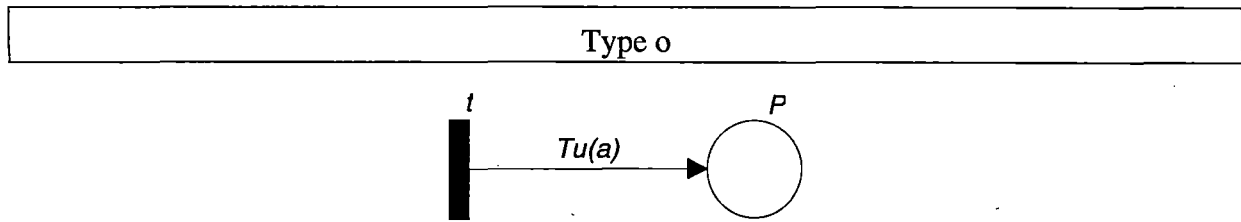


Figure 7.6 (Graphical representation of a *type o* arc)

At the termination of the firing of an α -instance of t (where α is a valuation) the α -instances of the tuples of $Tu(a)$ are added to the extension of the predicate P . This type of arc can be used to represent the fact that at the end of action $t(x)$, $P(x)$ becomes true.

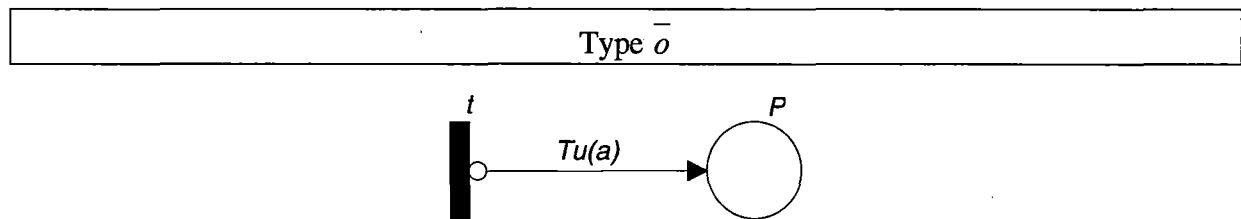


Figure 7.7 (Graphical representation of a *type o-bar* arc)

At the termination of the firing of an α -instance of t (where α is a valuation) the α -instances of the tuples of $Tu(a)$ are removed from the extension of the predicate P . This type of arc can be used to represent the fact that at the end of action $t(x)$, $P(x)$ becomes false.

With these new arcs we can now easily model the kind of constraint of figure 6.1 :

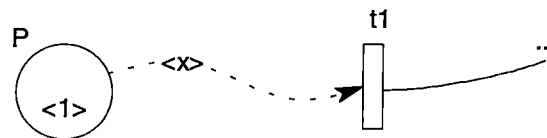


Figure 7.8 (Constraint of figure 7.1 in a PNRTL net)

7.3.3. Firing rule and operational semantics

So far we have only presented PNRTL nets in an informal manner. In this sub-section, we present formal definitions and formal semantics of a PNRTL net. The definitions of a PNRTL net include a fine function w . This function allows to specify the deontic distinction between ideal and sub-ideal behavior as defined in sub-section 5.2.2. We only give in definition 7.26 the

way to calculate the weight of a PNRTL execution, the reader can easily generalize the other definitions given in chapter 5 for the PNRTL nets.

Elements of a PNRTL net

Definition 7.13 (PNRTL net)

A *PNRTL net* N is a five-tuple $N = \langle P, T, A, L, w \rangle$ where P is a finite set of places which represent dynamic predicates, T is a set of transitions, A is a finite set of arcs which link places to transitions and vice versa, L is a language for structure which is used to annotate the net, w is a fine function defined on the set T that allows the deontic distinction between ideal, sub-ideal and repairing transitions.

Let us now detail the elements of a PNRTL net.

Definition 7.14 (PNRTL transition)

A *PNRTL transition*, in a net $N = \langle P, T, A, L, w \rangle$, is a five-tuple $t = \langle s, IA, OA, w, i \rangle$ where $t \in T$, $s(t)$ denotes a static formula called the selector of the transition, $IA(t)$ and $OA(t)$ denote respectively the set of input arcs and the set of output arcs of the transition, $w(t)$ is the weight of the transition t in the fine system, $i(t)$ is true if the transition is instantaneous³, false otherwise.

Definition 7.15 (PNRTL place)

A *PNRTL place*, in a net $N = \langle P, T, A, L, w \rangle$, is a couple $p = \langle Pr, Ext \rangle$ where $p \in P$, $Pr(p)$ denotes the dynamic predicate that annotates the place p , $Ext(p)$ is a set of tuples that belong to p and represent the extension of the predicate of place p ($Pr(p)$).

Definition 7.16 (PNRTL input arc)

A *PNRTL input arc*, in a net $N = \langle P, T, A, L, w \rangle$, is a four-tuple $a = \langle IP, Tr, Tu, Ty \rangle$ where $a \in A$, $IP(a) \in P$ denotes the input place of a , $Tr(a) \in T$ the transition linked to a , $Tu(a)$ the set of tuples, written in language L , that annotate the arc, $Ty(a) \in \{i, \bar{i}, p, \bar{p}\}$ the type of arc a .

Definition 7.17 (PNRTL output arc)

A *PNRTL output arc*, in a net $N = \langle P, T, A, L, w \rangle$, is a four-tuple $a = \langle OP, Tr, Tu, Ty \rangle$ where $a \in A$, $OP(a) \in P$ denotes the output place of a , $Tr(a) \in T$ the transition linked to a , $Tu(a)$ the set of tuples, written in language L , that annotate the arc, $Ty(a) \in \{o, \bar{o}\}$ the type of arc a .

³ We consider that a transition can be classified as instantaneous or not instantaneous. We consider that this characteristic is fixed regarding the action that the transition models and does not change during the process of the system. An instantaneous transition will be represented by a white rectangle and a non instantaneous by a black rectangle.

Definition 7.18 (Marking of a PNRTL net)

The marking M of a PNRTL net $N = \langle P, T, A, L, w \rangle$ where $P = \{p_1, p_2, \dots, p_n\}$, is a function returning the extension of the dynamic predicate that annotates the place given as argument :

$$M(p_i) = \text{Ext}(p_i), (1 \leq i \leq n), p_i \in P$$

Definition 7.19 (Index of a transition t)

The index of a transition t is the set of variables that appear freely in $s(t)$ the selector of the transition and in the tuples e of each arc connected to the transition :

$$\text{Index}(t) = \bigcup \{ \text{vars}(e) : e \in \text{Tu}(a) \wedge (a \in \text{IA}(t) \vee a \in \text{OA}(t)) \} \cup \text{vars}(s(t))$$

Graphical conventions. An instantaneous transition will be represented as a white rectangle and a non-instantaneous transition as a black rectangle.

Firing rule

To express the firing rule of the PNRTL nets, we define when a transition is enabled and also the effects of firing a transition instance. In the sequel, we constantly use the notion of PNRTL state. This notion has been introduced in the definition 7.12. Let us precise this definition :

Definition 7.20 (Elements of a PNRTL state).

A PNRTL state is a five tuple $S = \langle \text{InTime}, \text{OutTime}, M, H, \text{Exit} \rangle$ where :

- $\text{InTime}(S)$ and $\text{OutTime}(S)$ are state functions which return a positive real number. $\text{InTime}(S)$ returns the time at which the system has entered the state S , OutTime returns the time at which the system has quit the state S .
- $M(S)$ is a vector that gives the extension of each dynamic predicate of the system in the state S . The extension of the dynamic predicate which annotates place p in a state S will be noted $\text{Ext}_{M(S)}(p)$.
- $H(S)$ is a state function that returns a set of 2-tuples of the form $\langle t_n, \alpha \rangle$ where t_n is a transition occurrence and α a valuation for the index of t . $H(S)$ represents the set of transition occurrences, with their valuation, which are in progress during state S .
- $\text{Exit}(S)$ is a state function which returns a 3-tuple $\langle \text{BE}, t_n, \alpha \rangle$ where BE is either 'Begin', either 'End', or 'BeginEnd' if the exit event of the state is the beginning, respectively the end of the firing of the transition occurrence t_n , respectively the instantaneous firing of the transition occurrence t_n with a valuation α . This 3-tuple represents the event which makes quit the state S .

Definition 7.21 (Enabled α -instance transition)

A transition $t \in T$ is *enabled* for a valuation α in a marked net $\langle N, M \rangle$ which dynamizes a relational structure S iff :

1) α is a valuation for the variables of the index of t .

2) $S(s(t):\alpha) = \text{true}$, in other words the interpretation of the α instance of the formula $s(t)$ is true in the structure S .

3) $\forall a: a \in IA(t) \wedge (Ty(a) = i \vee Ty(a) = p):$
 $\forall e \in Tu(a): (e:\alpha) \in Ext_M(IP(a))$

In other words, an arc a of type i or p is enabled for a valuation α if for all tuples of the arc, each α -instance of the tuples belongs to the extension of the dynamic predicate of the place $IP(a)$.

4) $\forall a: a \in IA(t) \wedge (Ty(a) = \bar{i} \vee Ty(a) = \bar{p}):$
 $\forall e \in Tu(a): (e:\alpha) \notin Ext_M(IP(a))$

In other words, an arc a of type \bar{i} or \bar{p} is enabled for a valuation α if for all tuples of the arc, each α -instance of the tuples does not belong to the extension of the dynamic predicate of the place $IP(a)$.

5) $\neg \exists (e_1:\alpha), (e_2:\alpha):$
 $e_1 \in Tu(a) \wedge a \in OA(t) \wedge Ty(a) = o \wedge OP(a) = p$
 $\wedge e_2 \in Tu(b) \wedge b \in OA(t) \wedge Ty(b) = \bar{o} \wedge OP(a) = p$
 $\wedge (e_1:\alpha) = (e_2:\alpha)$

In other words, if a transition t is fireable for a valuation α then it does not exist two tuples whose α -instances are equal and must, at the terminaison of t , be added in and removed from the same place.

We have defined when a transition is enabled in a PNRTL net, let us now describe the effects of firing a transition. We describe those effects on the sequence of states (see definition 7.23) which represents an execution of a PNRTL net. In the sequel, S_i denotes the i^{th} state of the sequence S .

Definition 7.22 (Timing of the firing of a non-instantaneous transition).

The firing of a non-instantaneous transition occurrence is characterized by two events : its beginning and its end. $Begin_Time(t_n)$, $End_Time(t_n)$ denotes the time at which the n^{th} occurrence of the transition t begins, respectively ends to be fired. The difference

$End_Time(t_n)$ - $Begin_Time(t_n)$ represents the duration of the firing of t_n , let us note it $Duration(t_n)$.

Definition 7.23 (Firing rule of PNRTL)

A transition t is **fireable** for a valuation α whenever it is enabled for that valuation. We distinct here two cases :

A) Firing of a non-instantaneous transition :

The action of **firing** the n^{th} occurrence of transition t with a valuation α in a state S_i at time T for a duration d has two effects : one at the beginning of the firing and one at the end of the firing. Let us formally discribe those two effects :

at the **Beginning** of firing $(t:\alpha)$ in state S_i :

- $Exit(S_i) = (begin, t_n, \alpha)$, the exiting event of state S_i is the beginning of the α -instance of n^{th} occurrence of t .
- $S_i \xrightarrow{Exit(S_i)} S_{i+1}$, the reached state is S_{i+1} .
- $InTime(S_{i+1}) = T$, the time when the state S_{i+1} is reached is the time of the beginning of $(t_n:\alpha)$, $OutTime(S_i) = T = InTime(S_{i+1})$.
- The extensions of the dynamic predicates are changed in the following way :
 - 1) $\forall p \in \{pl: (IP(a) = pl \wedge a \in IA(t))\}$
 $Ext_{M(S_{i+1})}(p) \leftarrow Ext_{M(S_i)}(p)$
 $\cup \{(e:\alpha): \exists a \in IA(t) \wedge Ty(a) = i \wedge e \in Tu(a) \wedge IP(a) = p\}$
 $\cup \{(e:\alpha): \exists a \in IA(t) \wedge Ty(a) = \bar{i} \wedge e \in Tu(a) \wedge IP(a) = p\}$
 - 2) $\forall p \in P$ and $\notin \{pl: (IP(a) = pl \wedge a \in IA(t))\}$
 $Ext_{M(S_{i+1})}(p) \leftarrow Ext_{M(S_i)}(p)$
- The n^{th} occurrence of t is added to the set of transitions in progress in the reached state:
 $H(S_{i+1}) = H(S_i) \cup \langle t_n, \alpha \rangle$

at the **End** of firing $(t:\alpha)$ occuring in state S_j :

- $Exit(S_j) = (end, t_n, \alpha)$, the exiting event of state S_j is the end of α -instance of n^{th} occurrence of t .
- $S_j \xrightarrow{Exit(S_j)} S_{j+1}$, the reached state is S_{j+1} .
- $InTime(S_{j+1}) = T+d$, the time when the state S_{j+1} is reached is the time of the end of $(t_n:\alpha)$ ($T = Begin_time(t_n)$, $d = duration(t_n)$), $OutTime(S_j) = InTime(S_{j+1})$.
- The extensions of the dynamic predicates are changed in the following way :
 - 1) $\forall p \in \{pl: (OP(a) = pl \wedge a \in OA(t))\}$

$$\begin{aligned}
Ext_{M(S_{j+1})}(p) &\leftarrow Ext_{M(S_j)}(p) \\
&\cup \{(e:\alpha): \exists a \in OA(t) \wedge Ty(a) = o \wedge e \in Tu(a) \wedge OP(a) = p\} \\
&/ \{(e:\alpha): \exists a \in OA(t) \wedge Ty(a) = \bar{o} \wedge e \in Tu(a) \wedge OP(a) = p\}
\end{aligned}$$

$$2) \forall p \in P \text{ and } \notin \{pl: (OP(a) = pl \wedge a \in OA(t))\}$$

$$Ext_{M(S_{j+1})}(p) \leftarrow Ext_{M(S_j)}(p)$$

- The n^{th} occurrence of t is removed from the set of transitions in progress in the reached state:

$$H(S_{j+1}) = H(S_j) / \langle t_n, \alpha \rangle$$

B) Firing of a instantaneous transition :

The action of firing *instantaneously* the n^{th} occurrence of transition t with a valuation α in state S_i at time T has the following effects :

- $Exit(S_i) = (begin_end, t_n, \alpha)$.
- $S_i \xrightarrow{Exit(S_i)} S_{i+1}$
- $InTime(S_{i+1}) = OutTime(S_i) = T$.
- To describe the extension of the dynamic predicates in S_{i+1} , we decompose the instantaneous firing in two successive phases, the first one describes the effects of the input arcs and is followed by the second one which describes the effects of the output arcs. For a precise description of these two phases, we use a intermediary marking which we note $M_{(Int)}$.

- Description of $M_{(Int)}$:

$$1) \forall p \in \{pl: (IP(a) = pl \wedge a \in IA(t))\}$$

$$Ext_{M_{(Int)}}(p) \leftarrow Ext_{M(S_i)}(p)$$

$$/ \{(e:\alpha): \exists a \in IA(t) \wedge Ty(a) = i \wedge e \in Tu(a) \wedge IP(a) = p\}$$

$$\cup \{(e:\alpha): \exists a \in IA(t) \wedge Ty(a) = \bar{i} \wedge e \in Tu(a) \wedge IP(a) = p\}$$

$$2) \forall p \in P \text{ and } \notin \{pl: (IP(a) = pl \wedge a \in IA(t))\}$$

$$Ext_{M_{(Int)}}(p) \leftarrow Ext_{M(S_i)}(p)$$

- Description of $M(S_{i+1})$:

$$1) \forall p \in \{pl: (OP(a) = pl \wedge a \in OA(t))\}$$

$$Ext_{M(S_{j+1})}(p) \leftarrow Ext_{M_{(Int)}}(p)$$

$$\cup \{(e:\alpha): \exists a \in OA(t) \wedge Ty(a) = o \wedge e \in Tu(a) \wedge OP(a) = p\}$$

$$/ \{(e:\alpha): \exists a \in OA(t) \wedge Ty(a) = \bar{o} \wedge e \in Tu(a) \wedge OP(a) = p\}$$

$$2) \forall p \in P \text{ and } \notin \{pl: (OP(a) = pl \wedge a \in OA(t))\}$$

$$Ext_{M(S_{j+1})}(p) \leftarrow Ext_{M_{(Int)}}(p)$$

- $H(S_{i+1}) = H(S_i)$.

Definition 7.24 (Execution of a PNRTL net)

An *execution sequence* is a sequence of states obtained by applying the firing rule and that respects a couple of supplementary restrictions :

- if $Exit(S_i) = \langle begin, t_n, \alpha \rangle$ then $\exists j: (j > i) \wedge Exit(S_j) = \langle end, t_n, \alpha \rangle$. In other words, if a non-instantaneous action instance begins then it ends eventually. Thus we do not allow actions that last forever.
- if $Exit(S_i) = \langle end, t_n, \alpha \rangle$ then $\exists j: (1 \leq j < i) \wedge Exit(S_j) = \langle begin, t_n, \alpha \rangle$. Analogously, the occurrence of the end of a non-instantaneous action instance implies that that action has begun at an earlier state.
- if $Exit(S_i) = \langle begin \text{ or } begin_end, t_{n1}, \alpha \rangle$, $Exit(S_j) = \langle begin \text{ or } begin_end, t_{n2}, \beta \rangle$ and $i \neq j$ then $n1 \neq n2$. In other words, if an action instance happens at some state, it may not happen again even with a different valuation.
- if $Exit(S_i) = \langle begin \text{ or } begin_end, t_{n1}, \alpha \rangle$ and $1 \leq n2 < n1$ then $\exists j: 1 \leq j < i: Exit(S_j) = \langle begin \text{ or } begin_end, t_{n2}, \alpha \rangle$. In other words, the numbering of the action instances is continuous.
- $H(S_i) = \emptyset$. The set of transition instances in progress in the initial state S_1 is empty.
- The null transition, as in definition 6.8, must and can only be fired in terminal states. The firing of the null transition is always instantaneous. If there is no enabled transition in S_i : $Exit(S_i) = \langle Begin_end, null, - \rangle$, $S_i \xrightarrow{null} S_{i+1}$ and $M(S_{i+1}) = M(S_i)$.
- $InTime(S_i) \leq OutTime(S_i)$. The time at which a state is quit is always superior or equal to the time at which the state is reached.
- $InTime(S_{i+1}) = OutTime(S_i)$ and $InTime(S_0) = 0$. The *InTime* of a state is equal to the *OutTime* of the previous state and the *InTime* of the first state is 0.

Definition 7.25 (Semantics of a marked PNRTL net)

The operational semantics of a marked PNRTL $\langle N, M_0 \rangle$ is the set of all possible sequences of states S whose first state S_0 has the marking M_0 and other states are obtained by applying the firing rule defined in definitions 7.21, 7.23 and 7.24. The set of possible executions of a marked PNRTL net $\langle N, M_0 \rangle$ is noted $E(N, M_0)$.

We must still define how the deontic weight of an execution between two states can be computed in the PNRTL formalism. Recall that the weight of an execution between two markings in an extended Petri net (see section 5.2.2) is the sum of the weights of the fired transitions of the execution. In a PNRTL execution, the firing of a non-instantaneous transition

generates two events : a begin and an end event. So the weight of an execution can be defined as the sum of the weights of the transitions that begin during the execution.

Definition 7.26 (Deontic weight of PNRTL execution).

The deontic weight (DW) of an execution S between state S_i and state S_j of a PNRTL net $N = \langle P, T, A, L, w \rangle$ is calculated as follows :

$$DW(S_i, S_j) = \sum_{k=i}^j (w(Exit(S_k)) \times \delta_k)$$

Where $w(Exit(S_k))$ is the weight of the transition involved in the exiting event of state S_k and δ_k is equal to 1 if $Exit(S_k)$ is the begin event of a transition firing or the begin_end event of an instantaneous firing and is equal to 0 if $Exit(S_k)$ is the end event of a transition firing.

7.4 Logical formulae of PNRTL

As we see a PNRTL net as a generator of a set of possible executions $E(N, M)$, we expect that the temporal formulae should provide an alternative characterization, more descriptive and less operational of the desired set of executions of the PNRTL net.

7.4.1 State formulae

In the section 7.2, we have partitioned the set of predicates in a static part and in a dynamic part. The predicates of the static part can be used to speak about individuals properties which do not depend on the state of evaluation. The syntax and the semantics of a formula which contains just static predicates is given in definitions 7.3 and 7.10. In this subsection, we will only be interested in the logic constructions in relation with the notion of PNRTL state.

Recall that a PNRTL state is composed of five elements : a marking (the extension of each dynamic predicate of the system), an entering time (InTime), an exiting time (OutTime), an exiting event (the begin, the end or the begin_end of a transition firing), a set of firings that are in progress. So in the sequel we will define the syntax and the semantic of predicates and functions in order to speak about these five elements.

Syntax

New symbols :

The set of *dynamic predicates* which annotate the places of the PNRTL net of the specification is defined as follows : $\{pr: pr = Pr(p) \wedge p \in P\}$. Those predicates allow us to speak about their extension in a state.

The predicate *InProgress* whose arguments are a *transition occurrence* and a *n-tuple* of values or variables where *n* is the arity of the index of *t*. $InProgress(t_n(x,a,b))$ expresses the fact that the *nth occurrence* of the transition *t* which was fired with a valuation α which assigns the value of the variable *x* to the first element of the index of *t*, the value *a* to its second element, the value *b* to its last element, **is in progress** in the state under consideration.

The state function *InTime* : $\rightarrow Real$ which returns the input time of the state. Assertions about this function will be constructed with usual predicate symbols over real numbers : $=, <, \leq, \geq, >, \neq$.

The state function *OutTime* : $\rightarrow Real$ which returns the output time of the state. Assertions about this function will be constructed with common predicate symbols over real numbers : $=, <, \leq, \geq, >, \neq$.

The three predicates *Begin*, *BeginEnd* and *End* whose arguments are a transition occurrence and a n-tuple of value or variables. $Begin(t_n(x,a,b))$ expresses the fact that the begin of *nth occurrence* of the transition *t* with a valuation α which assigns the value of the variable *x* to the first element of the index of *t*, the value *a* to its second element, the value *b* to its last element, is **the exiting event** of the state under consideration. $End(t_n, \alpha)$ expresses the same but for the end of a firing, $Begin_End(t_n, \alpha)$ for the instantaneous firing of *t_n*.

Semantics

We will give here the **truth** value of the constructs previously introduced. Those truth values are evaluated in a state. For the other expressions that can be constructed from the structure language *L*, we refer the reader to Definition 7.4, page 7.10.

Let $[\delta]_S$ denote the truth value of the logic formula δ in the State *S* whose entering time is *InTime*, exiting time is *OutTime*, marking is *M*, set of progressing transitions occurrences is *H* and whose exiting event is *Exit*.

$[InProgress(t_n, \alpha)]_S$ is true iff $\langle t_n, \alpha \rangle \in H(S)$. Note that we can consider a generalization on the occurrence number : $[InProgress(t, \alpha)]_S$ is true iff there exists a occurrence number, n , such that $\langle t_n, \alpha \rangle \in H(S)$.

$[Begin(t_n, \alpha)]_S$ is true iff $Exit(S) = (begin, t_n, \alpha)$. Here we can also consider a generalization on the occurrence number : $[Begin(t, \alpha)]_S$ is true iff there exists an occurrence number, n , such that $Exit(S) = (begin, t_n, \alpha)$.

$[End(t_n, \alpha)]_S$ is true iff $Exit(S) = (end, t_n, \alpha)$. Here we can also consider a generalization on the occurrence number : $[End(t, \alpha)]_S$ is true iff there exists an occurrence number, n , such that $Exit(S) = (end, t_n, \alpha)$.

$[BeginEnd(t_n, \alpha)]_S$ is true iff $Exit(S) = (begin_end, t_n, \alpha)$. Here we can also consider a generalization on the occurrence number : $[BeginEnd(t, \alpha)]_S$ is true iff there exists an occurrence number, n , such that $Exit(S) = (begin_end, t_n, \alpha)$.

For the predicates constructed over real time, we define the interpretation of the two time state function : $I_S(InTime) = InTime(S)$ and $I_S(OutTime) = OutTime(S)$.

$[P(x)]_S$, where P is a dynamic predicate, is true iff the value of $x \in Ext_{M(S)}(p)$ where $Pr(p) = P$. In other words, $P(x)$ is true in a state S if the value of x belongs to the tuples contained in the place annotated by the dynamic predicate P .

7.4.2. Real Time Temporal formulae

A PNRTL net execution can be viewed as a sequence of states as defined in definition 7.25. Each state in an execution has two time stamps, the first one **InTime** represents the (real) time at which the net has entered in that state, and the second one **OutTime** represents the (real) time at which the net has exited that state. In the section 6.3.2 we have introduced temporal operators to express properties on the order of states in a sequence. In this subsection we will extend to real time the syntax and the semantics of those operators.

The temporal assertions constructed with the temporal operator of section 6.3.2 may constraint the order of possible states but those assertions cannot express constraints about the distance in time between states. However, this temporal information, of **quantitative** nature, is often very important (see chapter 2).

In the sequel, we will adapt the syntax and the semantics of real time temporal operator introduced by Koymans [KOYMANS89], [KOYMANS92]. Variants of those operators are also used in many other works : ALBERT [DDDP94b], ERAE [DLT91], RTOSL [SK93],...

Example 7.1 (A real-time temporal assertion).

$G_{\leq t \text{ sec}} \phi$ is true if ϕ is verified in all future states within $t \text{ sec}^2$.

Recall that in our language, a net execution is a sequence of states where each state is formed of, among others, a marking and an exiting event. The following figure represents a part of a possible PNRTL net execution :

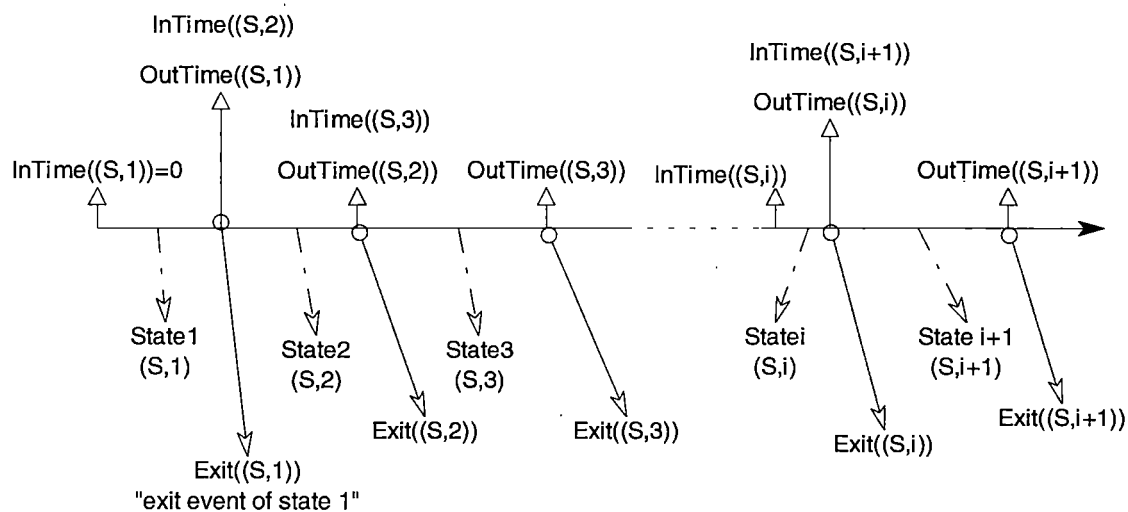


Figure 7.9 (A possible execution sequence)

If we want to give a precise semantics to the formula $G_{\leq t \text{ sec}} \phi$, we have to decide which timestamp of a state (*InTime* or *OutTime*) is taken into account to compute the distance between two states.

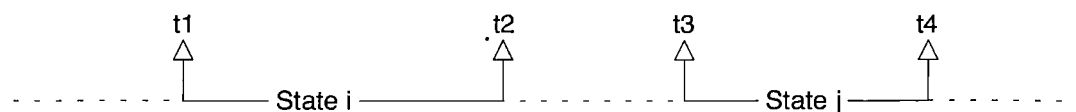
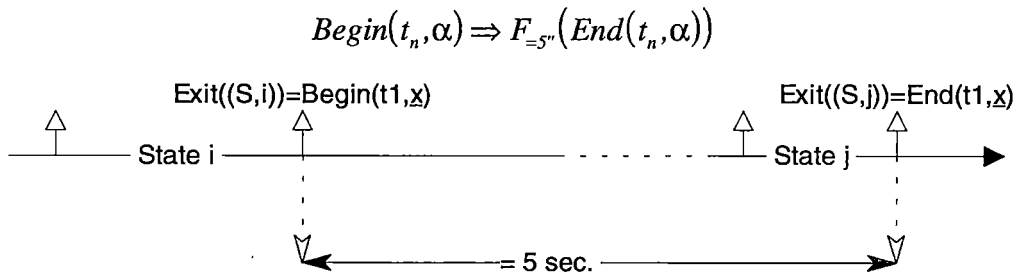


Figure 7.10 (The distance between two states)

In fact $(t_3 - t_1)$ may be different of $(t_4 - t_2)$. Constraints on the duration between two events will easily be defined by measuring the distance between two *OutTime* state points. For instance, if we consider the following constraint : "Each firing of t take exactly 5 seconds". We would like to write :



State j is distant of 5 seconds from the State i if we consider the two *OutTime* time stamps

Figure 7.11 (Distance Out-Out between two states)

Now if we want to express "P(a) may never stand 4 seconds continuously", we cannot write : $P(a) \Rightarrow F_{\leq 4}(\neg P(a))$ (7.1). In fact, consider the following figure :

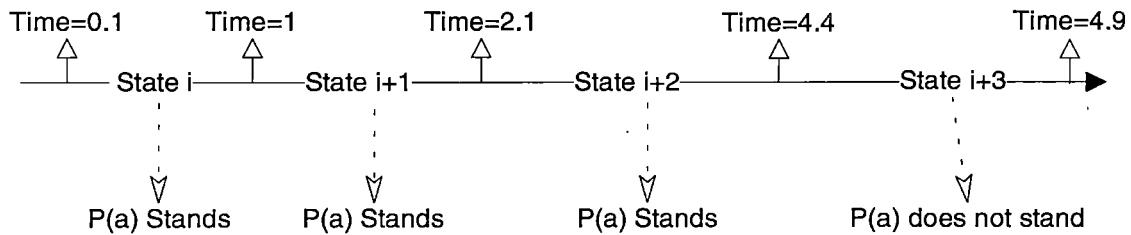


Figure 7.12 (A possible execution sequence)

If we interpret the formula (7.1) in *state i* with the semantics of intervals computed on *OutTime*, its truth value is **true** which is **counter-intuitive** ! In fact, $P(a)$ stands from time $t=0.1s$ to time $t=4.4s$, i.e., continuously during 4.3 s which is not allowed by our constraint. So, to solve this problem, we propose to add an exponent to the real-time temporal operators. This exponent will allow to determine how to compute the interval (from *InTime* to *OutTime*, from *OutTime* to *InTime*...). This exponent is a 2-tuple which elements belongs to the set $\{i, o\}$.

$$P(a) \Rightarrow F_{\leq 4}^{(i,i)}(\neg P(a)) \quad (7.2)$$

The formula (7.2) expresses the desired constraint since the meaning of the exponent (i,i) is : "the interval contains all states that are at *InTime* within 4 seconds from the *InTime* of the state where the formula is evaluated". Besides, the truth value of the formula (7.2) in state *i* of the figure 7.12 is false which is intuitive !

In the sequel, we will give a more formal syntactic and semantic definition of our real-time temporal operators.

Syntax

We will use the temporal operators : F (eventually), G (henceforth) and U (untill) for the future and P (Sometimes in the past), H (always in the past) and S (since) for the past. As we already said, the syntax of those operators will be extended in two ways : a subscribe will allow to restrict the meaning of the formula to a set of states which constitute an interval and an exposant to specify the precise way to compute intervals of states.

Let here extend the syntax of the temporal formulae given in section 5.3 :

$$\begin{aligned}
 \langle \text{Temporal_formula} \rangle ::= & \\
 & \langle \text{State_formula} \rangle \\
 & | X \langle \text{Temporal_formula} \rangle \\
 & | F_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | G_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | (\langle \text{Temporal_formula} \rangle) U_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | Y \langle \text{Temporal_formula} \rangle \\
 & | P_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | H_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | (\langle \text{Temporal_formula} \rangle) S_{\langle \text{Interval} \rangle}^{(\text{Bound})} (\langle \text{Temporal_formula} \rangle) \\
 & | \neg \langle \text{Temporal_formula} \rangle \\
 & | \langle \text{Temporal_formula} \rangle \wedge, \vee, \Rightarrow, \Leftrightarrow \langle \text{Temporal_formula} \rangle
 \end{aligned}$$

$$\begin{aligned}
 \langle \text{Bound} \rangle ::= & \varepsilon \\
 & | \langle (i,i) \rangle | \langle (i,o) \rangle | \langle (o,o) \rangle | \langle (o,i) \rangle
 \end{aligned}$$

$$\begin{aligned}
 \langle \text{Interval} \rangle ::= & \varepsilon \\
 & | = \langle \text{Duration symbol} \rangle \\
 & | < \langle \text{Duration symbol} \rangle \\
 & | \leq \langle \text{Duration symbol} \rangle \\
 & | \geq \langle \text{Duration symbol} \rangle \\
 & | > \langle \text{Duration symbol} \rangle
 \end{aligned}$$

Additional rule : if *Interval* is equal to ϵ (the empty interval) then *Bound* must also be equal to ϵ .

Semantics

As we already said the intervals are interpreted as sets of states. Let us first give a formal definitions of these sets.

To alleviate the definition of the interpretation of interval, we introduce a meta-function which returns either the function *InTime* or the function *OutTime*.

Notation 7.1 (Meta time function)

The meta-function *mt* is defined as follows : $mt(i) = \text{"InTime"}$, $mt(o) = \text{"OutTime"}$ and $mt(\epsilon) = \text{"OutTime"}$. The expression $mt(i)(s,j)$ is equivalent to $InTime(s,j)$.

The truth value of a real-time temporal formula is evaluated with regard to an interval of states. In the following definition, we will show how to compute intervals.

Notation 7.2 (Interval interpretation function).

The interpretation of the scope of a real-time formula is a set of states. Let I_{int} denote the interval interpretation function. The arguments of this function are :

1. *f* or *p* for interval in the futur or in the past.
2. a sequence and a state position, ex (S,i) , where the formula is evaluated.
3. $\begin{matrix} (Bounds) \\ (Interval) \end{matrix}$ are the subscribe and the exopant of the real-time temporal operator to evaluate.

Intervals in the future :

$$I_{int}(f, (S,i)_{(=t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, i..+\infty) : mt(\delta 2)(S, j) - mt(\delta 1)(S, i) = t\}$$

$$I_{int}(f, (S,i)_{(<t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, i..+\infty) : mt(\delta 2)(S, j) - mt(\delta 1)(S, i) < t\}$$

$$I_{int}(f, (S,i)_{(\leq t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, i..+\infty) : mt(\delta 2)(S, j) - mt(\delta 1)(S, i) \leq t\}$$

$$I_{int}(f, (S,i)_{(>t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, i..+\infty) : mt(\delta 2)(S, j) - mt(\delta 1)(S, i) > t\}$$

$$I_{int}(f, (S,i)_{(\geq t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, i..+\infty) : mt(\delta 2)(S, j) - mt(\delta 1)(S, i) \geq t\}$$

Intervals in the past :

$$I_{int}(p, (S,i)_{(=t)}^{(\delta 1, \delta 2)}) = \{(S, j) \in (S, I..i) : mt(\delta 1)(S, i) - mt(\delta 2)(S, j) = t\}$$

$$I_{Int}(p, (S, i)_{(\delta 1, \delta 2)}^{(< t)}) = \{(S, j) \in (S, I..i) : mt(\delta 1)(S, i) - mt(\delta 2)(S, j) < t\}$$

$$I_{Int}(p, (S, i)_{(\delta 1, \delta 2)}^{(\leq t)}) = \{(S, j) \in (S, I..i) : mt(\delta 1)(S, i) - mt(\delta 2)(S, j) \leq t\}$$

$$I_{Int}(p, (S, i)_{(\delta 1, \delta 2)}^{(> t)}) = \{(S, j) \in (S, I..i) : mt(\delta 1)(S, i) - mt(\delta 2)(S, j) > t\}$$

$$I_{Int}(p, (S, i)_{(\delta 1, \delta 2)}^{(\geq t)}) = \{(S, j) \in (S, I..i) : mt(\delta 1)(S, i) - mt(\delta 2)(S, j) \geq t\}$$

Semantics of the operators :

$$[F_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff } \exists(S, j) \in I(f, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true.}$$

$$[G_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff } \forall(S, j) \in I(f, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true.}$$

$$[(\phi)U_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff}^4 :$$

$$\exists(S, j) \in I_{Int}(f, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true and}$$

$$\forall k : i \leq k < j : [\phi]_{(S,k)} \text{ is true.}$$

$$[P_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff } \exists(S, j) \in I(p, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true.}$$

$$[H_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff } \forall(S, j) \in I(p, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true.}$$

$$[(\phi)S_{Int}^b(\vartheta)]_{(S,i)} \text{ is true iff :}$$

$$\exists(S, j) \in I_{Int}(p, (S, i)_{Int}^b) : [\vartheta]_{(S,j)} \text{ is true and}$$

$$\forall k : j < k \leq i : [\phi]_{(S,k)} \text{ is true.}$$

7.5 Many-sorted structures

The use of many-sorted structures is often very beneficial. It allows to identify categories of objects and to structure a specification in declarations which identify objects and assertions which express properties [DTL91].

If we distinguish for a structure different sorts of individuals, the signature has to assign as indices not just numbers but strings of sort symbols to predicates and strings paired with a single sort symbol to operators indicating the distribution of domains. If A, B, C, D are sort symbols, then $P^{(A,B,D)}$ denotes a relation in $(A \times B) \times D$ and $F^{(A,C:B)}$ denotes a function from $A \times C$ into B .

In PNRTL we provide a set of predefined data types : *Boolean*, *Integer*, *Real*, *Char* and *String*, these types are interpreted **over fixed domains**. We also give the possibility to the user

⁴ The chosen interpretation imposes that the state where ϑ is true belongs to the specified interval.

to construct new data types. Structured data types are built by the analyst using the following constructors⁵ :

- a *Cartesian product* groups in a same structure values of possible different types, the constructor of this type is : **CP**.
- A *Set* groups distinct values of a same type, the constructor is : **SET**.
- A *Bag* is a set where the multiple membership is allowed, the constructor for the bag is : **BAG**.
- A *Sequence* is a bag where elements are ranked, constructor : **SEQ**.
- An *Union* of several types include data of all types member of the union, constructor : **UNION**.
- An *Enumeration* is a static structure enumerating some particular value, constructor : **ENUM**.

A set of predefined operators to work on the structures which can be constructed, is also provided to the analyst. For instance, the operators *Card* applied to a set returns its *cardinality*. The complete list of operators is given in the [DDDP94b].

Let now define the syntax of *declarations* and *types constructions*.

$$\langle \text{Type_Construction} \rangle ::=$$

$$\text{Sort } \langle \text{Constructed_Type_Name} \rangle : \langle \text{Type_Name} \rangle$$

$$\left| \text{CP} \left[\langle \text{Type_Name} \rangle (\langle \text{Type_Name} \rangle)^* \right] \right|$$

$$\left| \text{SET} \left[\langle \text{Type_Name} \rangle \right] \right|$$

$$\left| \text{SEQ} \left[\langle \text{Type_Name} \rangle \right] \right|$$

$$\left| \text{BAG} \left[\langle \text{Type_Name} \rangle \right] \right|$$

$$\left| \text{UNION} \left[\langle \text{Type_Name} \rangle (\langle \text{Type_Name} \rangle)^* \right] \right|$$

$$\left| \text{ENUM} \left[\langle \text{Constant_Symbol} \rangle (\langle \text{Constant_Symbol} \rangle)^* \right] \right|$$

$$\langle \text{Type_Name} \rangle ::= \langle \text{Predefined_Type_Name} \rangle \mid \langle \text{Constructed_Type_Name} \rangle$$

$$\langle \text{Predefined_Type_Name} \rangle ::= \text{Real} \mid \text{Boolean} \mid \text{Integer} \mid \text{Char} \mid \text{String}$$

$$\langle \text{Constant_Declaration} \rangle$$

$$::= \text{Const } \langle \text{Constant} \rangle (\langle \text{Constant} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*$$

⁵These constructors are taken from the formal language ALBERT.

$$\langle \text{Static_Predicate_Decl} \rangle$$

$$::= \text{St_Pred} \langle \text{Predicate} \rangle (, \langle \text{Predicate} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*$$

$$\langle \text{Dynamic_Predicate_Decl} \rangle$$

$$::= \text{Dyn_Pred} \langle \text{Predicate} \rangle (, \langle \text{Predicate} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*$$

$$\langle \text{Transition_Declaration} \rangle$$

$$::= \text{Trans} \langle \text{Transition} \rangle (, \langle \text{Transition} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*{}^6$$

$$\langle \text{Function_Declaration} \rangle$$

$$::= \text{Funct} \langle \text{Function} \rangle (, \langle \text{Function} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*$$

$$\langle \text{Variable_declaration} \rangle$$

$$::= \text{Var} \langle \text{Variable} \rangle (, \langle \text{Variable} \rangle)^* : \langle \text{Type_Name} \rangle (\times \langle \text{Type_Name} \rangle)^*$$

The introduction of types in PNRTL imposes some additional constraints which must be fulfilled by a well-formed specification. Moreover the semantics of the language is affected by the introduction of types.

Definition 7.27 (Type well-formed specification)

A specification is said type well-formed if it respects the following rules :

- The constructions of types respect the BNF rule given above.
- All variables, constants⁷, predicates, functions and transitions which appear in a specification must be declared.
- For all function applications $f(t_1, t_2, \dots, t_n)$ for which the function declaration is $\text{Funct } f: \tau_1 \times \tau_2 \times \dots \times \tau_n \rightarrow \tau$, the term t_i must be declared of type τ_i , the result of the function f is of type τ .
- For all predicate application $P(t_1, t_2, \dots, t_n)$ whose predicate declaration is $\text{St_Pred, Dyn_Pred } P: \tau_1 \times \tau_2 \times \dots \times \tau_n$, the term t_i must be declared of type τ_i .
- For all equalities $t_i = t_j$, the type of t_i must be equal to the type of t_j .

⁶ This declaration schema allows to declare the types of the variables appearing in the index of the transition.

⁷ Some constants don't need to be declared, it is the case for the constant representing elements of the predefined types.

The semantic of the language is also affected by the introduction of types : the types introduced must be interpreted and the semantics of the quantifier is modified. The interpretation of the predefined types is done on fixed domain :

- $I_T(\text{'Integer'})^8$ is the set of integer numbers.
- $I_T(\text{'Real'})$ is the set of real numbers.
- $I_T(\text{'Boolean'})$ is the set $\{\text{True}, \text{False}\}$ of boolean values.
- $I_T(\text{'Char'})$ is the set $\{\text{'a'}, \text{'b'}, \dots, \text{'A'}, \text{'B'}, \dots, \text{' '}, \text{'.'}, \text{';'}, \dots\}$ of characters.
- $I_T(\text{'String'})$ is the set of words which can be constructed by concatenation of elements of the set of characters.

The semantics of the quantified assertions is modified in the following way :

- $[\forall xP]_S$ is true iff for all values $d \in \text{Type}(x): [P:\{x \leftarrow d\}]_S$ is true.
- $[\exists xP]_S$ is true iff there exists a value $d \in \text{Type}(x): [P:\{x \leftarrow d\}]_S$ is true.

The function *Type* returns the set of values of the same type as that of the term given as argument.

7.6. A specification in separate sub-nets

In the previous specification examples, a single net was used to represent the dynamic part of the modeled system. This was possible because the modeled systems were very simple. But if one specified a complex system with a single net, the result could be unreadable. So to allow more readable specifications, we allow the analyst to split his specification in separate sub-nets. This possibility is only a graphical and syntactical facility, the semantics of a specification is evaluated in aggregating sub-nets specifications.

In the sequel of this section, we will introduce the new concepts with an illustrative example.

Example 7.2 (An other producer - consumer system)

Let consider a system made of two sets of components. The first set contains 4 producer machines and the second 4 consumer machines. A consumer machine may send a request for a piece of 2 different types to the set of producer machines. Each of those requests must be served within 5 seconds and in order of arrival (FIFO). As supplementary constraints, we impose that a consumer machine may not order a piece if it waits for a previous one or if it works on a piece, a piece ready to be consumed must be handled within 1 second, the final work on this piece lasts exactly 1 second.

⁸ $I_T(\text{'Type_Name'})$ denotes the type interpretation function applied to 'Type_Name'.

An intuitive way to separate the net into two sub-nets in this case is to model the set of producer machines in one net and the set of consumer machines in one other. But if we model this problem into two separate sub-nets, we must find communication means to allow an interaction between the two sets of components. The communication can be realized by shared places, i.e., places which belong to both sub-nets or shared transitions.

A place which belongs to many separate sub-nets is annotated by the same dynamic predicate in each sub-net. Furthermore, for reasons of readability, it will be represented by a grayed circle.

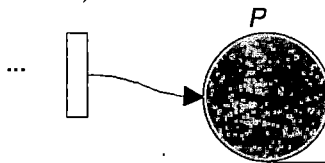


Figure 7.13 (A shared place representation)

Likewise a transition that belongs to many separate sub-nets is annotated by the same name, selector and have the same set of arcs.

Let us specify our illustrative example :

```
Sort ProdMachine=ENUM[p1,p2,p3,p4]
      % Identifier of the 4 producer machines
ConsMachine=ENUM[c1,c2,c3,c4]
      % Identifier of the 4 consumer machines
TypeIntPiece=ENUM[t1,t2]
      % The two types of intermediary pieces
TypeProdPiece=ENUM[pp1,pp2]
      % The two types of final pieces
```

```
St_Pred CanProduce : ProdMachine X TypePiece
      % The static predicate CanProduce(m,t) states that the machines
      m can produce a piece of type p.
```

```
Dyn_Pred Free_To_Demand : ConsMachine
Free_To_Produce : ProdMachine
Piece_Asked, Piece_To_Consume : ConsMachine X TypeIntPiece X Integer
Produced_Piece : TypeProdPiece X Integer
```

```
Trans Demand : ConsMachine X TypeIntPiece X Integer
Produce : ProdMachine X ConsMachine X TypeIntPiece X Integer
Consume : ConsMachine X TypeIntPiece X Integer
```

```
Function Manuf : TypeIntPiece → TypeProdPiece
```

Var md, md₁ : ConsMachine
 m : ProdMachine
 t, t₁ : TypeIntPiece
 n, n₁ : Integer
 tf : TypeProdPiece

Static Part

- (a1) $\forall m : \text{CanProduce}(m, t_1)$
 % All producer machines can produce the type t₁ piece.
- (a2) $\text{CanProduce}(p_1, t_2) \wedge \text{CanProduce}(p_2, t_2)$
 $\wedge \neg \text{CanProduce}(p_3, t_2) \wedge \neg \text{CanProduce}(p_4, t_2)$
 % Only the producer machines p₁ and p₂ can produce the
 intermediary piece of type t₂
- (a3) $\text{Manuf}(t_1) = pp_1 \wedge \text{Manuf}(t_2) = pp_2$
 % Definition of the Manuf function

Dynamic part

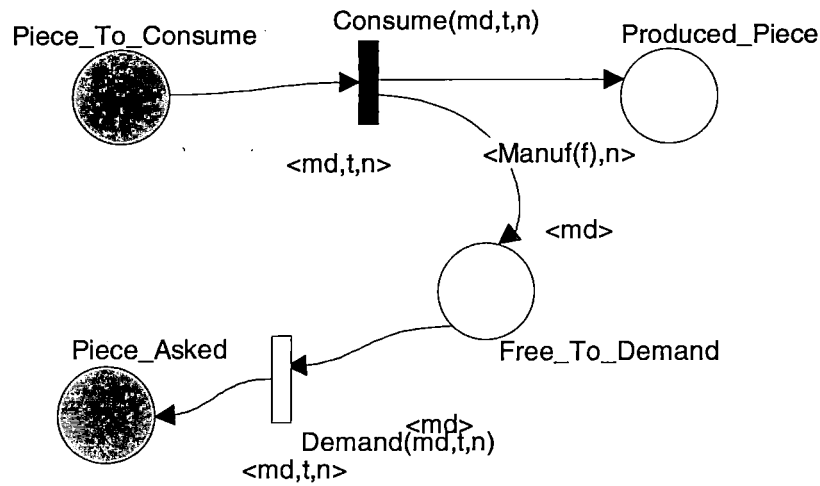


Figure 7.14 (The consumer machines)

- (a4) $\text{Begin}(\text{Demand}, (md, t, n)) \rightarrow X(G(\neg \text{BeginEnd}(\text{Demand}, (md, t, n))))$
 % Uniqueness of a demand
- (a5) $\text{Piece_To_Consume}(md, t, n) \rightarrow F_{\leq 1 \text{sec}}^{(t, o)}(\text{Begin}(\text{Consume}, (md, t, n)))$
 % A piece to consume does not stay more than one second before being
 handled
- (a6) $\text{Begin}(\text{Consume}, (md, t, n)) \rightarrow F_{=1 \text{sec}}(\text{End}(\text{Consume}, (md, t, n)))$
 % The action Consume takes exactly 1 second
- (a7) $\text{Init} \rightarrow \forall md : \text{Free_To_Demand}(md)$
 $\text{Init} \rightarrow \forall tf, n : \neg \text{Produced_Piece}(tf, n)$
 $\text{Init} \rightarrow \forall md, t, n : \neg \text{Piece_To_Consume}(md, t, n)$
 $\text{Init} \rightarrow \forall md, t, n : \neg \text{Piece_Asked}(md, t, n)$
 % Initializations

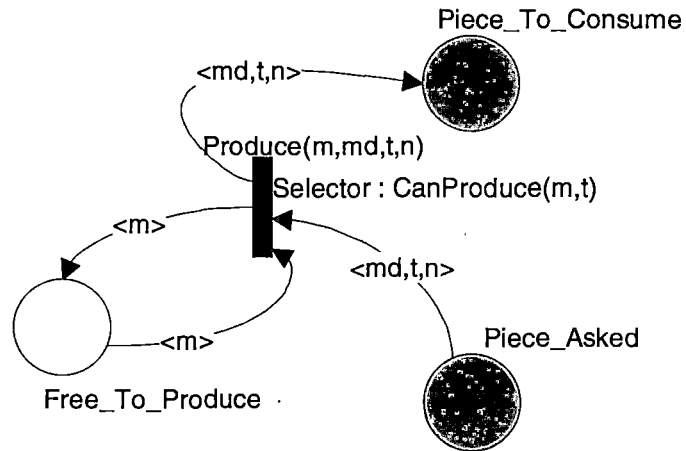
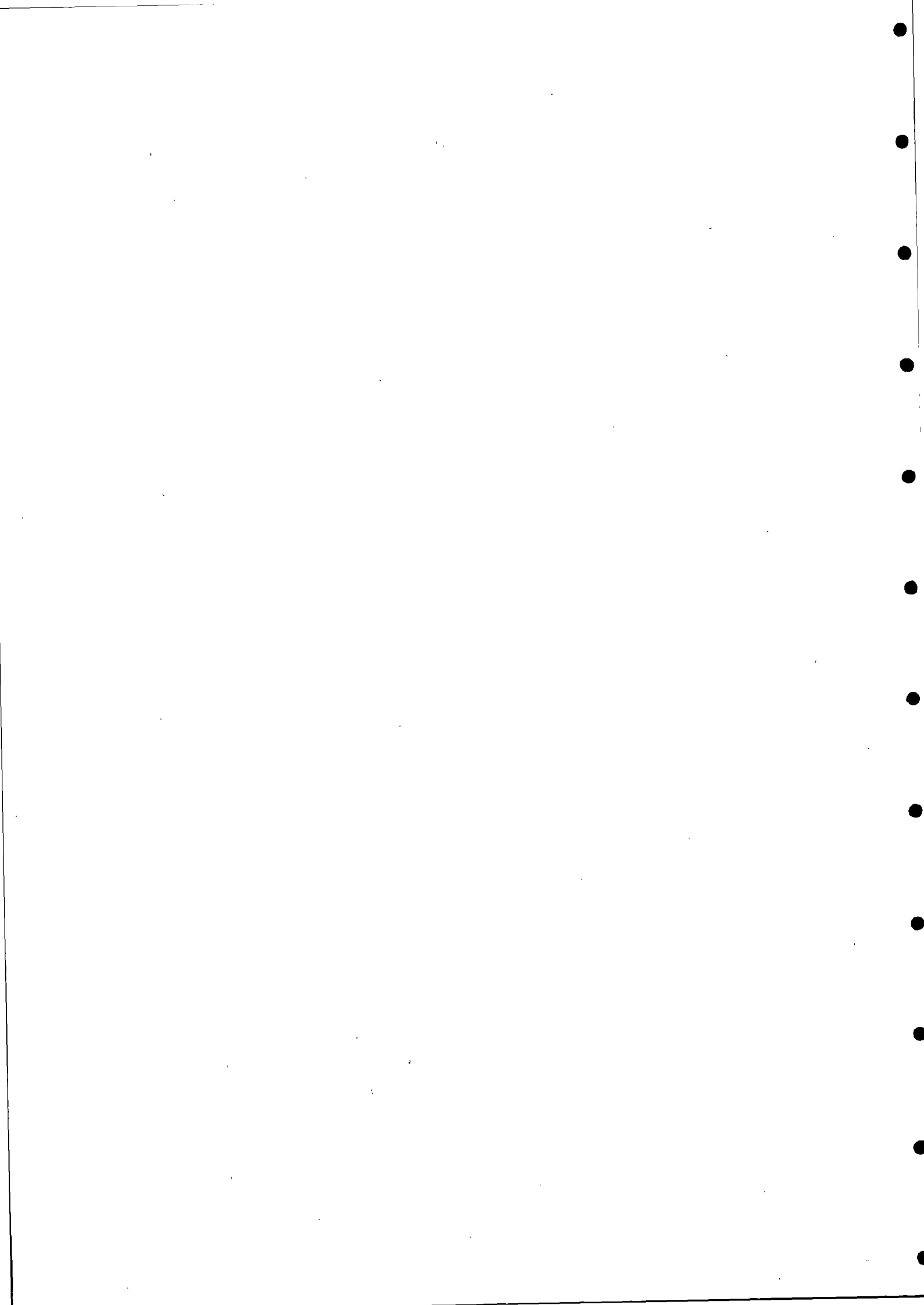
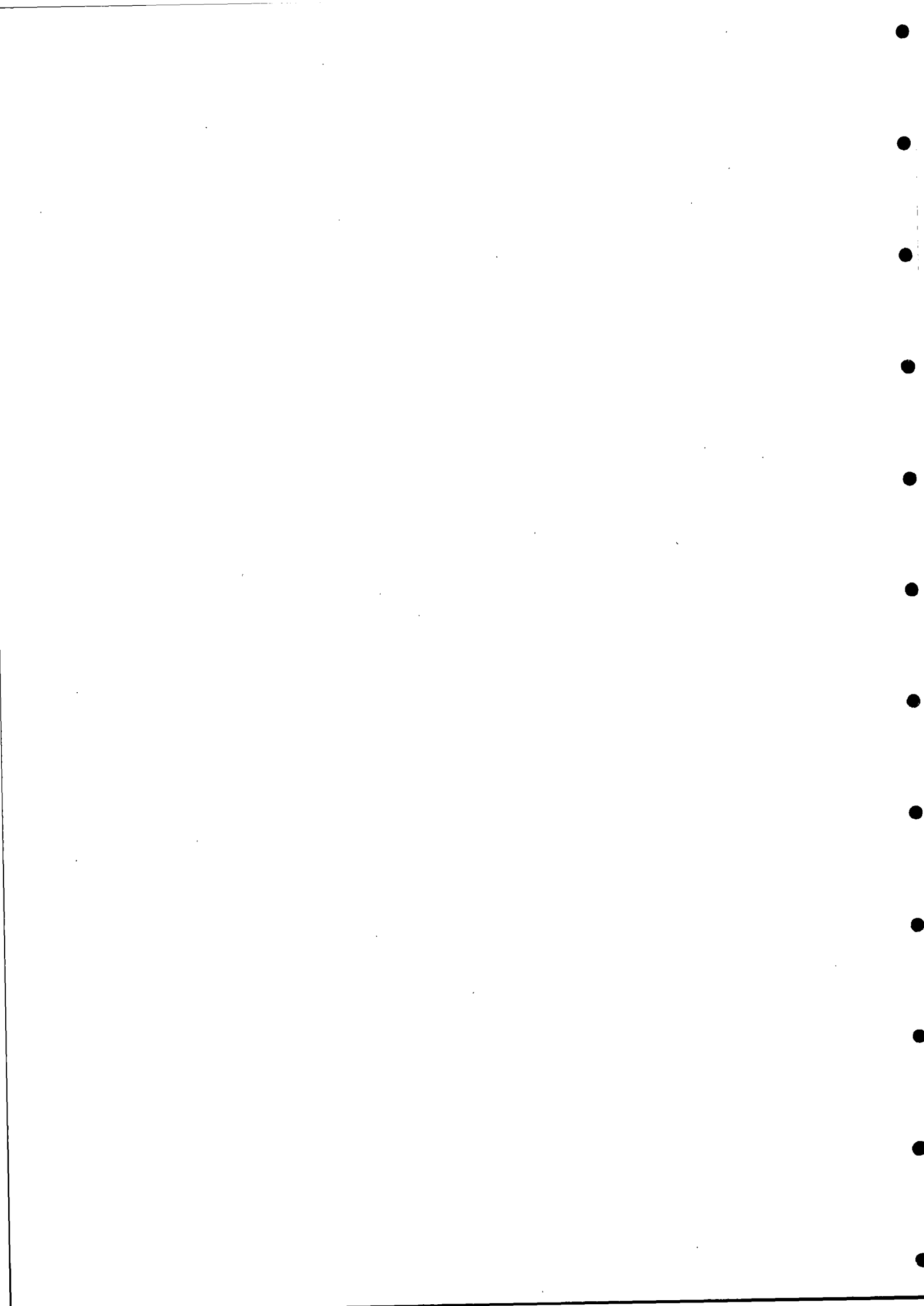


Figure 7.15 (The producer machines)

- (a8) $\text{Piece_Asked}(md,t,n) \wedge$
 $F(\neg \text{Piece_Asked}(md1,t1,n1) \wedge X(\text{Piece_Asked}(md1,t1,n1)))$
 $\rightarrow F(\text{Begin}(\text{Produce},(md,t,n)) \wedge F(\text{Begin}(\text{Produce},(md1,t1,n1))))$
% FIFO
- (a9) $\text{Piece_Asked}(md,t,n) \rightarrow F_{\leq 5\text{sec}}^{(i,i)}(\text{Piece_To_Consume}(md,t,n))$
% An required piece in produced within 5 seconds
- (a10) $\text{Init} \rightarrow \forall m : \text{Free_To_Produce}(m)$
% Initially, all machines are free to produce



PART THREE
APPLICATION AND TOOLS



Chapter 8

A case study in PNRTL

8.1 Description of the case

Three libraries decide to cooperate in order to offer their customers a maximum choice of readings. They intend to build a net which will allow someone to borrow or consult as well local books as books belonging to other libraries. The transfer of books between the different libraries is ensured by a van. Here are the rules that the borrowers and the libraries undertake to respect :

- A **book** belongs to a single library, but can also be ordered by people of other libraries.
- People are only allowed to go in the library in which they are registered.
- When an ordered book is arrived, it remains **48 hours** in (for the one who has ordered the book to come and take it). Over this period, it is resent to the owner-library. If the user who has ordered the book does not come, he receives a fine (this behavior is clearly subideal).
- A library can send a book to another library by the intermediary of the van system. The van service ensures that a book is transferred **within 1 day**.
- People can borrow **maximum 3 books at the same time** except for the students who are entitled to keep 4 readings.
- The duration of a borrowing is **30 days** for a book of a local library and **15 days** for a book of another library.

- Every reading that is returned late gives rise to a **fine** (returning a book too late is clearly a subideal behavior).
- A fine **must** be paid before any new borrowing and within **30 days** (from the moment it was administered).
- Damaging a book is a very subideal behavior. Unfortunately, a control can not be made at the library and thus damaging a book does not give rise to a fine.
- When an user makes a demand, the library must reply to its demand **within 2 minutes**.
- When a library l_1 receives a demand for a book from another library l_2 , the library l_2 must reply **within 2 seconds**.

In the following we present the specification of the case. The specification is commented.

8.2 Specification of the case

We have divided the specification of the system in three main components : the users, the libraries and the van system. The dynamic behavior of the users and the van system is presented by a single net. The behavior of the libraries is too complex to be presented in a single net. So we have divided the specification of their behavior in four subnets which represent 'macro'-actions of a library. These four views are :

- *Library-User-Demand* : this subnet specifies how a library must handel an user's demand for a book.
- *Library-User-Return* : this view shows what a library must do when an user return a book.
- *Library-Demand-Inter-Libraries* : this subnet specifies how a library must reply to a demand of another library for a book.
- *Library-Van* : this subnet defines the behavior of a library when it receives books of the van system.

Sort

```
TLib = ENUM[Library_1, Library_2, Library_3]
```

```
% There are three libraries in the system.
```

```
TBook = CP[Id:Integer, Title:String, Author:String, Owner:TLib]
```

```
% A book is identified by Id, has a Title, an Author and an Owner library.
```

```
TUser = CP[Id:Integer, Name:String, Lib:TLib, Student:Boolean]
```

```
% An user is identified by Id, has a Name, is member of a library Lib and is a student or not.
```

```
TReply = ENUM[OK,KO]
```

```
% OK is a positive reply while KO is a negative one.
```

```
TDate = Real;
```

```
% A date is represented by a real number.
```

Dyn_Pred

Returned_Book : TBook \times TLib \times TUser \times Integer \times TDate

% Returned_Book(b,l,u,n,d) means that the book b has been returned to the library l by the user u, d is the date b was borrowed and n serves in an identification mechanism

Book_To_Take : TBook \times TLib \times TUser \times Integer \times TDate

% Book_To_Take(b,l,u,n,d) means that the user u can take the book b in the library l, d represents the moment from which the book is available, n serves in an identification mechanism.

Demand : TBook \times TLib \times TUser \times Integer

% Demand(b,l,u,n) means that the user u wants to borrow the book b, the demand is issued to the library l, the 2-tuple (u,n) is an identifier for a user's demand.

Reply : TReply \times TBook \times TLib \times TUser \times Integer

% Reply(r,b,l,u,n) means that the library l has send the reply r (OK or KO) to the demand (u,n) of an user u for a book b.

Fine : Integer \times TUser \times Integer

% Fine(m,u,n) means that the user u must pay a fine of amount m, n serves in an identification mecanism.

Borrowed_Book : TBook \times TLib \times TUser \times Integer \times TDate

% Borrowed_Book(b,l,u,n,d) means that the book b is borrowed by user u since d and has been borrowed at the library l, n serves in an identifier mechanism.

Credit : TUser \times Integer

% Credit(u,n) means that the credit of user u is n.

Demand_Cr_OK : TBook \times TLib \times TUser \times Integer

% Demand(b,l,u,n) contains demands (u,n) whose user u has a sufficient credit

Demand_Int_Lib : TLib \times TBook \times TLib \times TUser \times Integer

% Demand_Int_Lib(l2,b,l1,u,n) means that the library l2 has received a demand for a book b by the intermediary of the library l1 for an user u, n serves in an identifier mechanism.

Reply_Int_Lib : TLib \times TReply \times TBook \times TLib \times TUser \times Integer

% Reply_Int_Lib(l2,r,b,l1,u,n) means that the library l2 has sent a reply r to the library l1 for a book b asked by user u.

Book_In_Shelf : TBook \times TLib

% Book_In_Shelf(b,l) means that the book b is in the shelves of the library l.

Book_To_Put_Away : TBook \times TLib

% Book_To_Put_Away(b,l) means that the book b must be put away in the shelves of the library l.

Book_To_Send : TBook \times TLib \times TLib \times TUser \times Integer

% Book_To_Exp(b,l1,l2,u,n). If owner(b)=l2 then it means that the library l1 wants to return the book b to its library l2. If owner(b)≠l2 then it means that the library l1 send the book b to the library l2 because the user u has asked to borrow the book b, n serves in an identifier mechanism.

Book_Arrived : TBook \times TLib \times TUser \times Integer

% Book_Arrived(b,l,u,n) means that a book b has been brought in library l by the van. If owner(b)=l then the book can be put away in the shelf. If owner(b)≠l then the book has been ordered by the user u (demand identified by (u,n)).

Trans

% Actions of the USERS

Return, Take, Damage : TBook \times TLib \times TUser \times Integer \times TDate

% Return : an user returns a book. Take : an user takes a book that he has demanded. Damage is a sub-ideal behavior.

Ask : TBook \times TLib \times TUser \times Integer

% Ask(b,l,u,n) means that an user u makes a demand for the borrowing of a book b at its library l. (u,n) identifies an user's demand.

Read : TReply \times TBook \times TLib \times TUser \times Integer

% Read(r,b,l,u,n) : means that the user u reads the reply r of the library l for the book b, the reply follows the demand (u,n).

Pay_Fine : Integer \times TUser \times Integer

% Pay_Fine(a,u,n) means that the user u pays a fine of amount a, n serves in an identification mechanism.

% Actions of the LIBRARIES

VerifC1 : TBook \times TLib \times TUser \times Integer \times Integer

% VerifC1(b,l,u,n,c) makes follow a demand of an user u (which has a credit $c>0$) for a book b.

VerifC2 : TReply \times TBook \times TLib \times TUser \times Integer \times Integer

% VerifC2(r,b,l,u,n,c) sends a negative reply (KO) to an user u which wants to borrow a book b but has no more credit ($c=0$).

Reply1 : TReply \times TBook \times TLib \times TUser \times Integer \times Integer

% Reply1(r,b,l,u,n,d) : if the book b asked by user u is present in the shelves of library l then reply $r=OK$ and the library l transfers the book from the shelf to the user.

Reply2 : TReply \times TBook \times TLib \times TUser \times Integer \times Integer

% Reply2(r,b,l,u,n) : if an asked book is not in the shelves reply $r=KO$ to the user which wants the book.

Decrease_Cr : TUser \times Integer

% Decreases the credit of an user when the library gives him a book.

Ask_Int_Lib : TLib \times TBook \times TLib \times TUser \times Integer

% Transfers a request to another library.

Give_Fine : Integer \times TUser \times Integer

% Give_Fine(a,u,n) : Gives a fine of amount a to the user u, n serves as identifier mechanism.

Take_RB1 : TBook \times TLib \times TLib \times TUser \times Integer \times Integer \times TDate

\times Integer

% Takes a returned book of another library and put it in the books to send.

Take_RB2 : TBook \times TLib \times TUser \times Integer \times Integer \times TDate \times Integer

% Takes a book of the library returned by an user.

Put_Away_Too_Late : TBook \times TLib \times TLib \times TUser \times Integer \times Integer

\times TDate

% Puts in the books to resend an ordered book whose user is not come over 48 hours.

Put_Away_Shelf : TBook \times TLib

% Put a book of the library l in its shelf.

Reply_Int1 : TReply \times TLib \times TBook \times TLib \times TUser \times Integer

% Positive reply (OK) to a demand of another library and sending the asked book via the Book_To_Exp place.

Reply_Int1 : TReply \times TLib \times TBook \times TLib \times TUser \times Integer
% Negative reply (KO) to the demand of another library.

Sort1, Sort2 : TBook \times TLib \times TUser \times Integer
% Sorting of the books arrived by the van.

% Action of the VAN

Transfer : TLib \times TBook \times TLib \times TUser \times Integer
% Transfer(l1,b,l2,u,n) : transfer of the book b from library l1 to l2.

Static part

Var b1,b2 : TBook
 u1,u2 : TUser

(s1) $\neg \exists b1,b2 (b1 \neq b2 \wedge \text{Id}(b1) = \text{Id}(b2))$
% Id is an identifier for a book

(s2) $\neg \exists u1,u2 (u1 \neq u2 \wedge \text{Id}(u1) = \text{Id}(u2))$
% Id is an identifier for an user

Dynamic part

% Initializations

Var b : TBook;
 u : TUser;
 l : TLib
 n : Integer
 d : TDate

Init $\rightarrow \forall b (\text{In_Shelf}(b, \text{owner}(b)))$
% At the initialization, all books are in the shelves of their library.

Init $\rightarrow \forall u (\text{Student}(u) \rightarrow \text{Credit}(u, 4))$

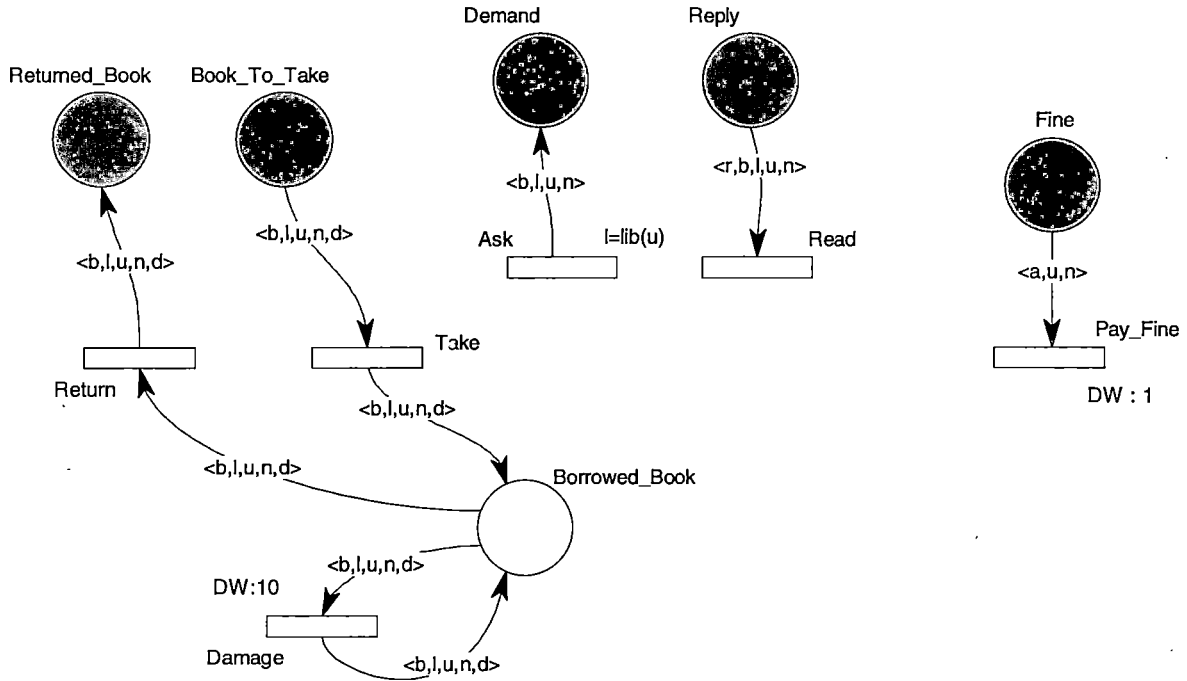
Init $\rightarrow \forall u (\neg \text{Student}(u) \rightarrow \text{Credit}(u, 3))$.
% At the initialization, the credit of an user which is student is 4, and 3 for a non student.

Init $\rightarrow \forall (b,l,u,n,d) (\neg \text{Returned_Book}(b,l,u,n,d)) \wedge \forall (b,l,u,n,d) (\neg \text{Book_To_Take}(b,l,u,n,d))$
 $\wedge \dots \wedge \forall (b,l,u,n) (\neg \text{Book_Arrived}(b,l,u,n))$
% At the initialization, apart Credit and In_Shelf, the extension of all dynamic predicates is empty.

% User behavior

Var b,b1 : TBook
 l,l1 : TLib
 u : TUser
 n,n1,n2,a : Integer

d : TDate



% Supplementary constraints

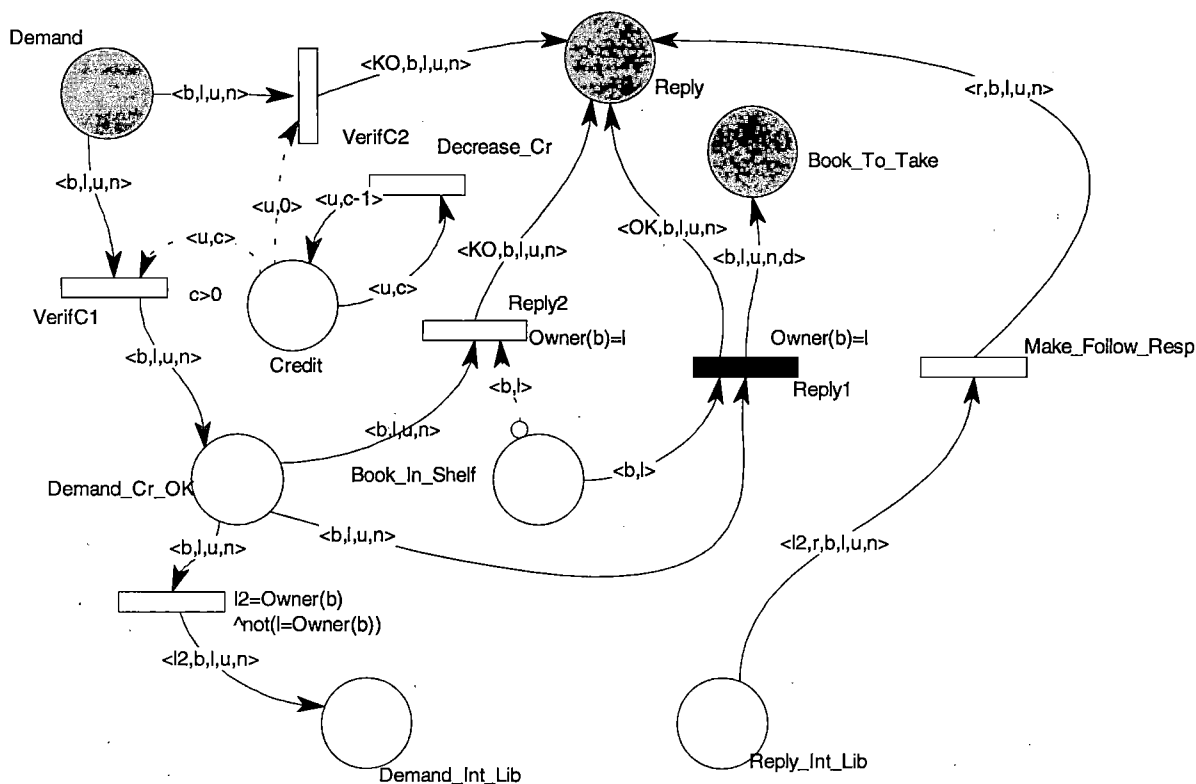
- (u1) $\text{BeginEnd}(\text{Ask},(b,l,u,n)) \rightarrow X(G(\neg\text{BeginEnd}(\text{Ask},(b,l,u,n))))$
% the 2-tuple (u,n) is identifier for a demand of an user.
- (u2) $\text{Book_To_Take}(b,l,u,n,d) \wedge \text{Owner}(b)=l$
 $\rightarrow F_{=0\text{sec}}^{(i,\phi)}(\text{BeginEnd}(\text{Take_Book},(b,l,u,n,d)))$
% When an user may borrow (his request has been accepted) a book of its library, he takes it immediately.
- (u3) $\text{Reply}(r,b,l,u,n) \rightarrow F_{=0\text{sec}}^{(i,\phi)}(\text{BeginEnd}(\text{Read},(r,b,l,u,n)))$
% An user reads immediately a reply sent to him by its library.
- (u4) $\text{Begin}(\text{Ask},(b,l,u,n)) \rightarrow X(\neg\text{BeginEnd}(\text{Ask},(b,l,u,n1))U(\text{Reply}(r,b,l,u,n)))$
% An user cannot do a demand if the previous one has not yet been replied.
- (u5) $\text{Fine}(a,u,n) \rightarrow F_{\leq 30d}^{(i,\phi)}(\text{BeginEnd}(\text{Pay_Fine},(a,u,n)))$
% A fine must always be paid within 30 days (strict obligation).
- (u6) $\text{Fine}(a,u,n1) \rightarrow \neg\text{BeginEnd}(\text{Ask},(b,l,u,n2))$
% An user which must pay a fine cannot demand to borrow a new book.

% General comments. The grayed places represent communications means between an user and its library. The transitions Pay_Fine and Damage have a positive deontic weight : 1 and 10. In fact, Pay_Fine is strictly obligated for an user that has received a fine. Thus preferred executions of an user are executions where he does not receive fines : executions where he does not return a book too late and where he always comes take an ordered book (these two actions are thus modeled as subideal).

% Library behavior

% View : Library-User-Demand

Var b : TBook
 l,l2 : TLib
 u : TUser
 n,c : Integer
 d : TDate



% Supplementary constraints

$$(10) \text{Begin}(\text{Reply1}, (r,l,b,u,n,d)) \vee \text{BeginEnd}(\text{Make_Follow_Reply}, (l1,OK,b,l2,u,n))$$

$$\leftrightarrow X(\text{BeginEnd}(\text{Decrease_Cr}, (u,c)))$$

% The credit of an user is decreased when he receives a positive reply to a demand of borrowing and only in this situation.

$$(11) \text{Demand}(b,l,u,n) \wedge Y(\neg\text{Demand}(b,l,u,n)) \rightarrow F_{\leq 2 \text{min}}^{(i,i)}(\text{Reply}(r,b,l,u,n))$$

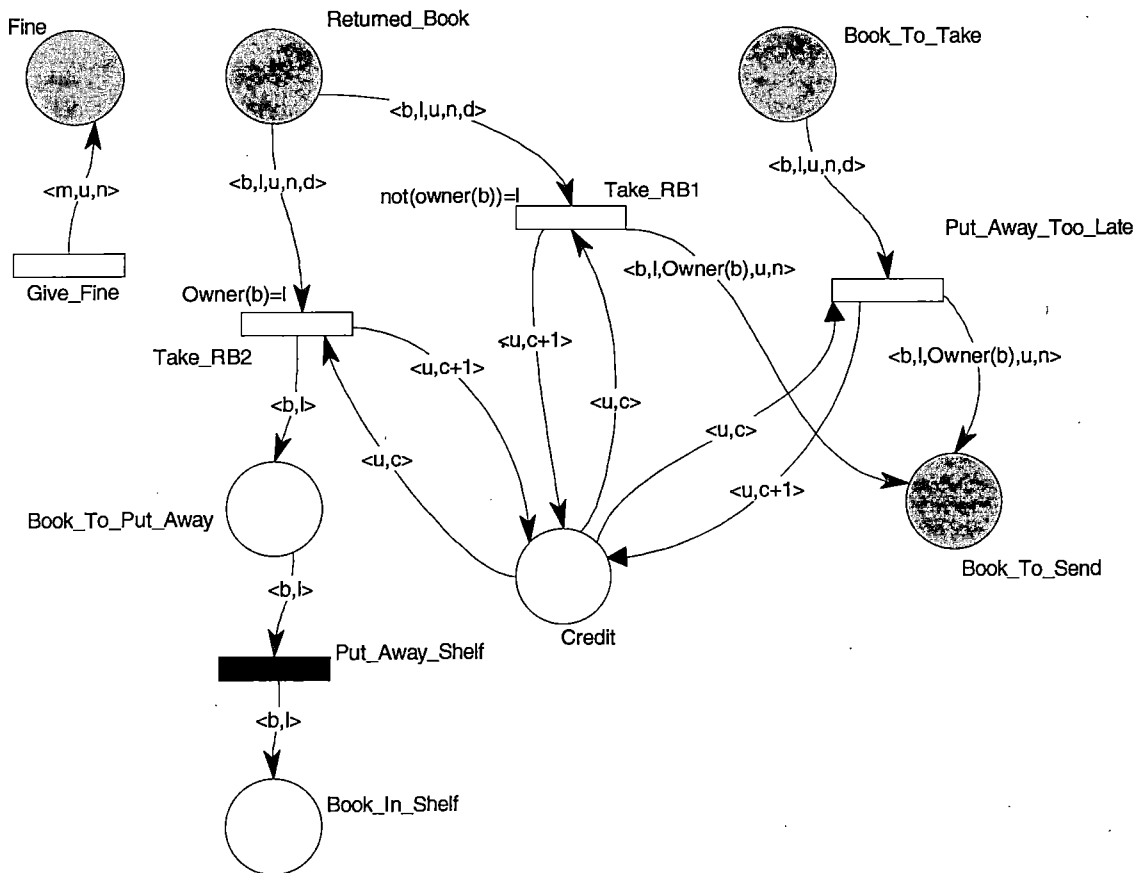
% A library must always reply to a demand of one of its users within 2 minutes.

% General comments. To simplify the subnet we have added the transition Decrease_Cr. In fact, this transition replaces arcs from transition Reply1 and a subdivision of Make_Follow_Reply in two transitions with one (for a positive reply) linked to the place credit. The firing of this transition is defined by the constraint (10).

The two transitions *Verif1* and *Verif2* are connected to place *Credit* by an arc to type *p* (see chapter7 for a formal definition), this also simplifies the net.
 Constraint (11) is a typical declarative constraint that is inexpressible in usual Petri net based languages.

% View : Library-User-Return

Var b : TBook
 l,l1,l2 : TLib
 u : TUser
 n,c : Integer
 d : TDate



% Supplementary constraints

$$(12) P_{\geq 48h}^{(o,i)}(\text{Book_To_Take}(b,l,u,n,d)) \wedge \text{Book_To_Take}(b,l,u,n,d)$$

$$\leftrightarrow \text{BeginEnd}(\text{Put_Away_Too_Late},(b,l,u,n,d))$$

% Over 48 hours, an ordered book when user is not come, is returned to its own library.

$$(13) \text{BeginEnd}(\text{Put_Away_Too_Late},(b,l,u,n,d)) \rightarrow F(\text{BeginEnd}(\text{Give_Fine},(100,u,n)))$$

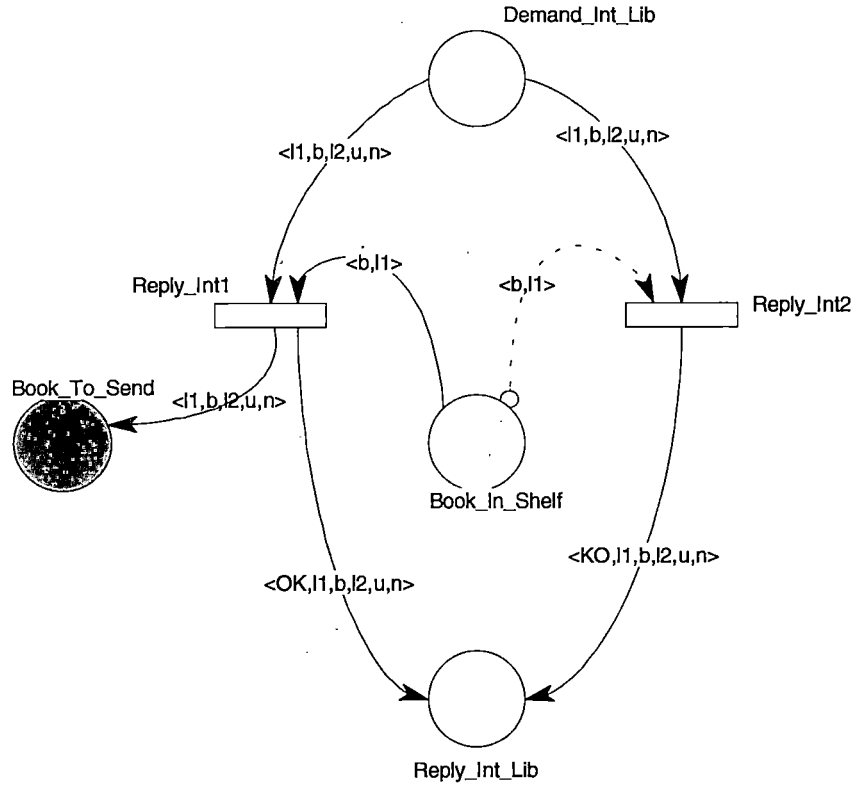
% If an user does not come and take the book he has ordered, the library administrates him a fine of 100.

- (14) $\text{BeginEnd}(\text{Take_RB1},(b,l,u,n,d)) \wedge \text{OutTime} = x \wedge (x-d > 15 d)$
 $\rightarrow F_{=0sec}(\text{BeginEnd}(\text{Give_Fine},(200,u,n))$
% If a borrower returns a book of another library too late (over 15 days), the library immediately administrates him a fine of 200.
- (15) $\text{BeginEnd}(\text{Take_RB2},(b,l,u,n,d)) \wedge \text{OutTime} = x \wedge (x-d > 30 d)$
 $\rightarrow F_{=0sec}(\text{BeginEnd}(\text{Give_Fine},(200,u,n))$
% If a borrower returns a book of the library too late (over 30 days), the library immediately gives him a fine of 200.
- (16) $\text{Begin}(\text{Put_Away_Shelf}_n,(b,l)) \rightarrow F_{\leq 5min}(\text{End}(\text{Put_Away_Shelf}_n,(b,l))$
% The action of putting away a book in its shelf lasts at most 5 minutes.
- (17) $Y(\neg \text{Book_To_Put_Away},(b,l)) \wedge \text{Book_To_Put_Away},(b,l)$
 $\rightarrow F_{\leq 1d}^{(i,o)}(\text{Begin}(\text{Put_Away_Shelf},(b,l))$
% A book stays at most one day before being put away.
- (18) $Y(\neg \text{Returned_Book}(b,l,u,n,d)) \wedge \text{Returned_Book}(b,l,u,n,d)$
 $\rightarrow F_{\leq 1min}^{(i,o)}(\text{BeginEnd}(\text{Take_RB1},(b,l,u,n,d)) \vee \text{BeginEnd}(\text{Take_RB2},(b,l,u,n,d)))$
% A returned book is treated within 1 minute.
- (19) $\text{BeginEnd}(\text{Give_Fine}(a,u,n))$
 $\rightarrow P ((\text{BeginEnd}(\text{Take_RB1},(b,l1,l2,u,n,d)) \wedge (\text{OutTime}-d > 15d))$
 $\vee (\text{BeginEnd}(\text{Take_RB2},(b,l1,l2,u,n,d)) \wedge (\text{OutTime}-d > 30d))$
 $\vee (\text{BeginEnd}(\text{Put_Away_Too_Late},(b,l1,l2,u,n,d))))$
% If a library gives a fine to an user u then u has returned a book too late or has not come to take the book he had ordered.

% General comments. In this subnet, we have taken the opposite approach for the increasing of the user credit to compare with the approach taken for the decreasing of the credit (previous subnet). This underlines a particular feature of our language : a lot of constraints can either be expressed by logic formulae or by graphical constructions.

% View : Library-Demand-Inter-library

Var b : TBook
 l1,l2 : TLib
 u : TUser
 n : Integer
 r : TReply



% Supplementary constraints

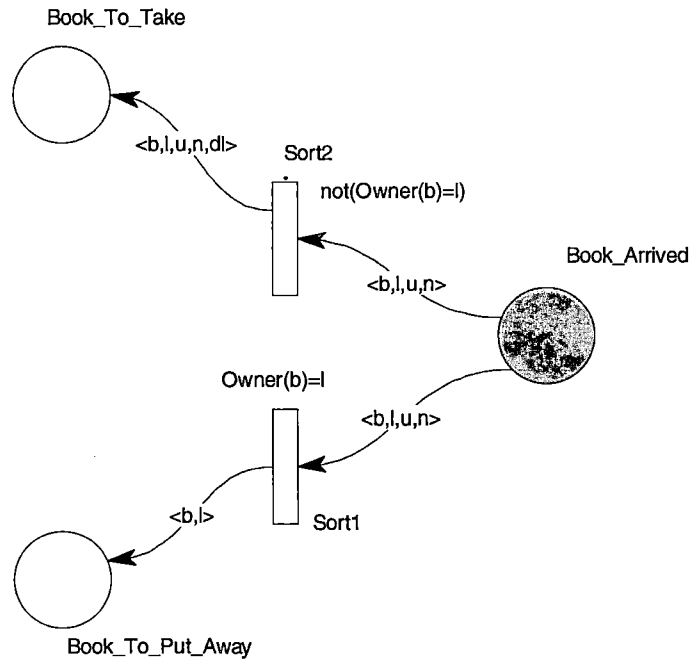
$$(110) \text{Demand_Int_Lib}(l1,b,l2,u,n) \wedge Y(\neg\text{Demand_Int_Lib}(l1,b,l2,u,n)) \rightarrow F_{\leq 2 \text{ sec}}^{(i,i)}(\text{Reply_Int_Lib}(r,l1,b,l2,u,n))$$

% An inter-library demand is always replied within two seconds (reponse time).

% General comments. The use of the arc of type \bar{p} (see chapter 7 for a formal definition) is very benefical. Without this type of arc we should add a complementary place to place *Book_In_Shelf*.

% View : Library - Van

Var b : TBook
 l : TLib
 u : TUser
 n : Integer
 d : TDate



% Supplementary constraints

$$(I11) \text{Book_Arrived}(b,l,u,n) \wedge Y(\neg \text{Book_Arrived}(b,l,u,n))$$

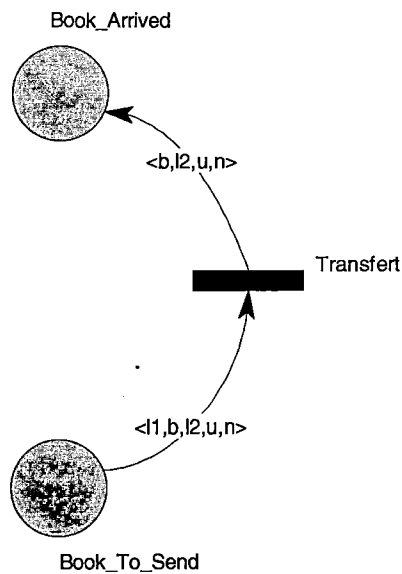
$$\rightarrow F_{\leq 1h}^{(t,\phi)} (\text{BeginEnd}(\text{Sort1},(b,l,u,n)) \vee \text{BeginEnd}(\text{Sort2},(b,l,u,n,d)))$$

% The books brought by the van are sorted within an hour.

$$(I12) \text{BeginEnd}(\text{Sort2},(b,l,u,n,d)) \rightarrow (d=\text{OutTime})$$

% The borrowing time of an ordered book is the time when it becomes available for the user.

% Van Behavior



% Supplementary constraints

(v1) $\text{Book_To_Exp}(11,b,l2,u,n) \wedge Y(\neg\text{Book_To_Exp}(11,b,l2,u,n))$
 $\rightarrow F_{\leq 1d}^{(i,i)}(\text{Book_Arrived}(b,l2,u,n))$
% A book is always transfered within a day.

8.3 Conclusion

This case study has demonstrated the ability of the PNRTL language for specifying composite concurrent systems. The main advantage of the language is the possibility given to the analyst to specify operational constraints in an operational style (net) and declarative constraints in a declarative style (logical formulae). The utility of the different types of arcs and of the real-time temporal operators has been underlined throughout the comments in the specification. Strict deontic aspects can easily be expressed by logic formulae and sub-ideal behaviors by deontic weights.

Chapter 9

Desired executions of a PNTL specification

9.1 Introduction

In the previous chapters, we explained that the modeling of complex constraints with Petri nets often results in a too operational specification making the net too complicated and thus hardly readable. Therefore, we proposed to attach to each Petri net a set of logic temporal formulae expressing such constraints. The semantics of those formulae was given as the reduction of the set of possible executions and of the possible markings of the Petri net. In other words, among the markings belonging to the reachability set of a Petri net, only some of them are accepted. But also the set of execution paths must be reduced into a subset which verify all the logic formulae which accompany the net.

Here, we present some techniques to reduce the set (finite or not) of *possible* executions paths into the set of the *desired* executions paths (these are the ones respecting *all* the logic formulae) for a specification in PNTL. Those techniques are based on the finite automata [TRAK&BARZ73] and the formal language [SALOMAA73] theory. Then, we expose other techniques for the testing of temporal properties on sets of infinite executions paths. And finally, we explain how we can extend the linear temporal logic by means of automata.

9.2 Infinite executions on bounded Petri nets

In the sequel, we always assume that the Petri nets studied are **bounded** - this hypothesis is necessary to use the automata theory. This means that the set of reachable markings is not infinite (but the set of possible firing sequences may be infinite). Remember that an execution of a Petri net is viewed as an **infinite** sequence of states. For Petri nets with terminal marking(s), we introduced a *null* transition that can and must only fire in terminal states and whose firing results in the same marking. For instance, two possible infinite sequences of the net in figure 9.1 are : $s_1 = t_1 t_1 t_1 \dots t_1 \dots$ and $s_2 = t_1 t_1 \dots t_1 t_2 t_3 t_3 \dots t_3$. The boundedness of the net guarantees that the mapping of a Petri net into an automaton is possible.

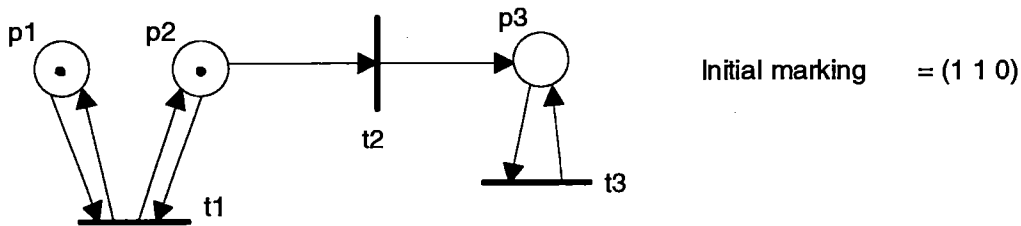


Figure 9.1 (a bounded Petri net)

9.3 Overview of the automata theory

9.3.1 Finite automata on finite words

A finite automaton [MAND&GHEZ87, EILENBERG74] is a machine that can reach a limited number of states, and which manipulates symbols received as input. Every handling of symbol affects the state of the automaton.

Definition 9.1 (Finite automaton)

A finite automaton is a 5-tuple $A = (\Sigma, Q, \delta, I, F)$

where Σ is an alphabet (that is a finite set of symbols).

Q is a finite set of states.

$\delta: Q \times \Sigma \rightarrow Q^*$ is the next state function (giving, for each state and each symbol, the possible resulting states of the automaton)¹.

¹ If the automaton is deterministic, the next-state function is : $\delta: Q \times \Sigma \rightarrow Q$

I is a finite set of initial states ($I \subset Q$).

F is a finite set of terminal states ($F \subset Q$).

An automaton A defines a set of **words** that are accepted by the automaton. This set of words is defined from the notion of **execution** of an automaton on a word.

Definition 9.2 (Execution of an automaton on a finite word)

An execution² of an automaton A on a finite word $w = a_0 a_1 \dots a_{n-1}$ is a finite sequence of states $\sigma(w) = s_0 s_1 \dots s_n$ such as:

- $\forall i \in [0, n]: s_i \in Q$
- $s_0 \in I$
- $\forall i \in [1, n]: s_i \in \delta(s_{i-1}, a_{i-1})$

Definition 9.3 (Admitted execution on a finite word)

An execution on a finite word w is admitted if the last state resulting from it, belongs to the set F :

$$\sigma(w) = s_0 s_1 \dots s_n \text{ is admitted} \Leftrightarrow s_n \in F$$

Definition 9.4 (Accepted word)

A word w is accepted if there exists on this word, an execution which is admitted:

$$w \text{ is accepted} \Leftrightarrow \exists \sigma \in Q^* : \sigma(w) \text{ is admitted}$$

Definition 9.5 (Language of an automaton)

The language $L(A)$ of an automaton A is the set of words accepted by A :

$$L(A) = \{w \in \Sigma^* : w \text{ is accepted}\}$$

9.3.2 Finite automata on infinite words

The Büchi automata [THAYSE89, DAVIS&WEYUKER83] - or finite automata on infinite words - are manipulating infinite sequences of symbols. They allow us to define in a formal way a set of infinite words. As the executions are infinite, terminal states don't exist anymore. They are replaced by accepting³ states, that is to say obligated states that must appear infinitely often in any execution of the automaton.

² It is not a function, except if the automaton is deterministic.

³ The word 'accepting' has been preferred to the one of 'final', although this last word can also be found in the literature.

Definition 9.6 (Büchi automaton)

A Büchi automaton is a 5-tuple $A = (\Sigma, Q, \delta, I, F)$

where Σ is an alphabet.

Q is a finite set of states.

$\delta: Q \times \Sigma \rightarrow Q^*$ is the next state function.

I is a finite set of initial states ($I \subset Q$).

F is a finite set of accepting states ($F \subset Q$).

Definition 9.7 (Execution of an automaton on an infinite word)

An execution of an automaton A on an infinite word $w = a_0 a_1 a_2 \dots$ is an infinite sequence of states $\sigma(w) = s_0 s_1 s_2 \dots$ such as:

- $\forall i \geq 0: s_i \in Q$
- $s_0 \in I$
- $\forall i > 0: s_i \in \delta(s_{i-1}, a_{i-1})$

Since an execution is infinite (thus without terminal state), we need to modify the notion of admitted execution :

Definition 9.8 (Admitted execution on an infinite word)

An execution on an infinite word w is admitted if it contains an infinite number of occurrences of the same accepting state:

$$\sigma(w) = s_0 s_1 s_2 \dots \text{ is admitted} \Leftrightarrow \exists q_f \in F, \exists \text{ infinity of } i \in \mathbb{N}: s_i = q_f$$

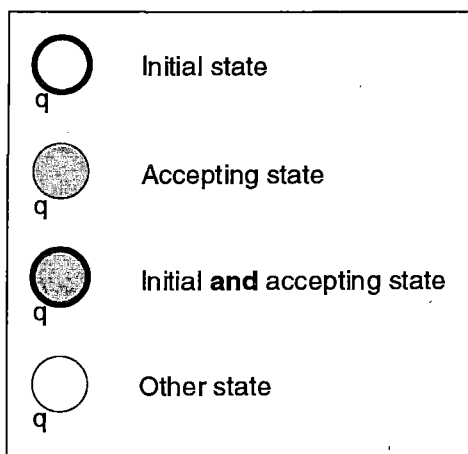
Notations and example :

Figure 9.2 (notations)

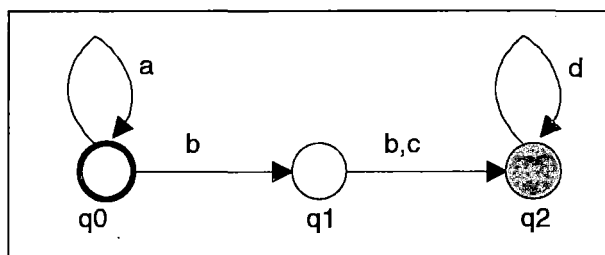


Figure 9.3 (example of a Büchi automaton)

Examples of words accepted by the automaton in figure 9.3 are :

w1 = aaabddddd...

w2 = bbcddddd...

w3 = abcddddd...

In practice, the language accepted by Büchi automata can always be defined by means of ω -regular expressions [AHO&ULLMAN93, GINZBURG68], using the symbol ' ω ' for infinite repetition. Other usual conventions are '*' for the finite repetition, '|' for the union and juxtaposition for the concatenation. For instance, the language of the automaton of figure 9.3 is :

$$L(A) = \{w = a^*b(b|c)d^\omega\}$$

9.3.3 Generalized Büchi automata

Generalized Büchi automata [THAYSE89] have a more complex definition, but on the other hand, it often allows an easier definition of certain languages.

Definition 9.9 (Generalized Büchi automaton - GBA)

A generalized Büchi automaton is a 5-tuple $\bar{A} = (\Sigma, Q, \delta, I, \bar{F})$

where the four first components are identical to the ones of a Büchi automaton

\bar{F} is a set of sets of accepting states : $\bar{F} = \{F_1, F_2, \dots, F_k\}$ with $F_j \subseteq Q$

Definition 9.10 (Admitted execution on a GBA)

For this type of automata, an execution is admitted if it contains infinitely often a state of each F_j :

$\sigma(w) = s_0 s_1 s_2 \dots$ is admitted $\Leftrightarrow \forall j \in [1, k] : \exists q_j \in F_j, \exists$ infinity of $i \in \mathbb{N} : s_i = q_j$

A GBA can always be brought to the form of a (normal) Büchi automaton which accepts the same language. The demonstration is given in [THAYSE89, pp. 192].

9.3.4 Closure properties of ω -regular languages

The languages accepted by Büchi automata form the class of ω -regular languages. This class of languages presents some interesting properties [DAVIS&WEYUKER83] : it is closed with respect to the operators of union, intersection and complementation :

If A_1 and A_2 are two automata accepting respectively the language $L(A_1)$ and $L(A_2)$, there always exists a Büchi automaton accepting

- $L(A_1) \cup L(A_2)$ (union)
- $L(A_1) \cap L(A_2)$ (intersection)
- $\Sigma^\omega / L(A_1)$ (complementation)

Such properties will be very useful for the reduction of the set of *possible* executions into the set of *desired* executions.

9.4 Büchi automata and bounded Petri nets

9.4.1 From Petri nets to automata

We have just seen that the language accepted by a Büchi automaton A is the set of words it accepts. Consider now a Petri net to which we attach a labelling function \mathcal{L} associating a distinct symbol⁴ to each transition ($\mathcal{L}: T \cup \{null\} \rightarrow \Sigma \cup \{\lambda\}$). Such Petri nets are called *free-labeled Petri nets* [PETERSON83]. So, any firing sequence $s = t_1 t_2 t_3 \dots$ corresponds⁵ to a word $w = \mathcal{L}(s)$. It is always possible for a bounded Petri net to build a deterministic automaton whose language contain the \mathcal{L} -projection of all firable sequences of the net.

Look at the net of figure 9.4. This Petri net can be formalized by a Büchi automaton A (figure 9.5), considering that the initial state is also the accepting state :

⁴ The empty symbol λ is always associated to the *null* transition.

⁵ The function $\mathcal{L}: T \cup \{null\} \rightarrow \Sigma$ is usually extended to $\mathcal{L}: (T \cup \{null\})^* \rightarrow \Sigma^*$

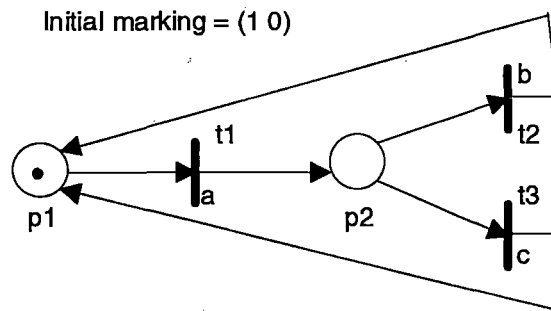


Figure 9.4 (a Petri net)

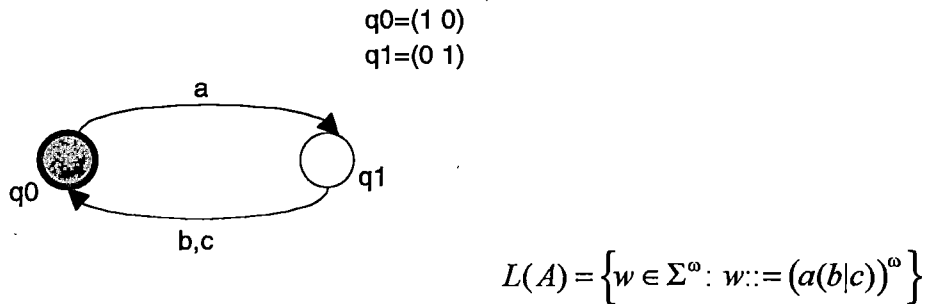


Figure 9.5 (the corresponding automaton)

As for the operational semantics of a Petri net (see chapter 6), we can consider an automata as a transition system with states (q_0 and q_1) and exiting transitions (the symbols on the arcs).

Since we model bounded Petri nets, the reachability sets $R(N, M_0)$ are finite (i.e. there is always a finite number of accessible markings from M_0). This means that every reachable marking is a possible state of the corresponding automaton, and that each arc t of the graph between two markings M_1 and M_2 is a transition of the automaton labeled with $\mathcal{L}(t)$ and linking q_1 and q_2 , where q_1 and q_2 are the states corresponding respectively to M_1 and M_2 :

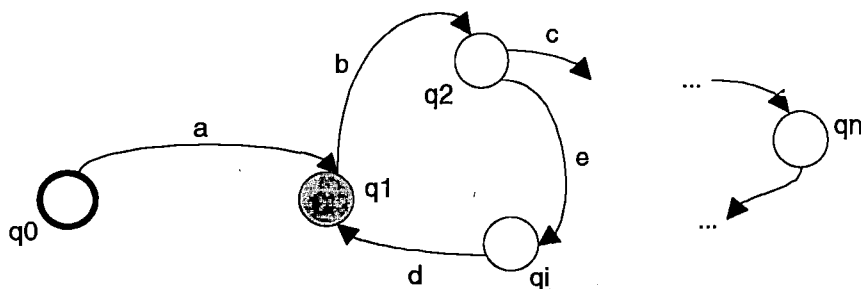


Figure 9.6 (an automaton modeling a Petri net)

And so, we can say, given a Petri net N and an initial marking M_0 , that:
 $\sigma(w) = s_0 s_1 s_2 \dots \Rightarrow \forall i \geq 0, s_i \in Q = \{q_0, q_1, \dots, q_n\}$ (with $Q \subseteq R(N, M_0)$)

9.4.2 The set of possible executions

A classical problem is the definition of the set $E(N, M_0)$ of possible executions of a Petri net starting with an initial marking M_0 . We saw that the solution of this problem can be found by the definition of the language of an automaton modeling the Petri Net (see below in section 9.4.4 for an algorithm). For the moment, we suppose there is only one accepting state ($\#F=1$).

Consider the graph of an automaton A modeling a bounded Petri net, like figure 9.6. An infinite word w is accepted (see definition 9.7 and 9.8) if the execution of A on w contains infinitely often the accepting state. Thus every word of $L(A)$ has two parts ($w ::= \alpha\beta$): the first part (α)⁶ is a **finite** word, while the second one (β) is an **infinite** word. The execution of α from the initial state (q_0) is a sequence whose last element and **only** last element is the accepting state; the execution of β from the accepting state is a sequence containing infinitely often this state. Since we have $\alpha ::= (\alpha_1|\alpha_2|\dots|\alpha_q)$ and $\beta ::= (\beta_1|\beta_2|\dots|\beta_p)^\omega$ - where α_i and β_j are words on Σ , the language can be defined as :

$$L(A) = \{w \in \Sigma^\omega : w ::= (\alpha_1|\alpha_2|\dots|\alpha_q)(\beta_1|\beta_2|\dots|\beta_p)^\omega\}$$

Example :

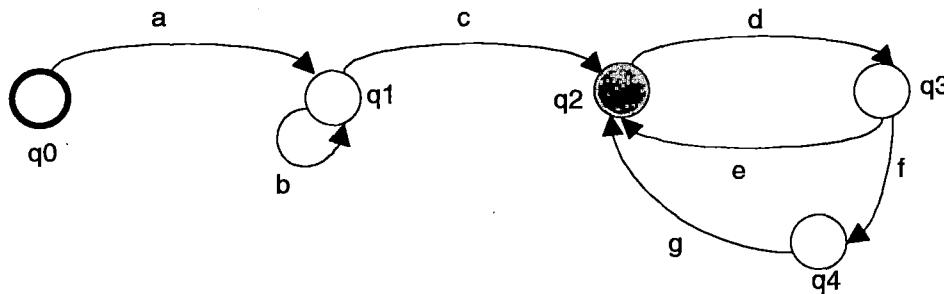


Figure 9.7 (a finite automaton)

Here, every accepted word w has the form $w ::= \alpha\beta$ where $\alpha ::= ab^*c$ and $\beta ::= (de|dfg)^\omega$

⁶ If the initial state is also the accepting state, the word α is empty.

As soon as we get the language of the automaton, we can then easily define the set of possible execution paths accepted by the Petri net N from the marking M_0 , by using the inverse of the labelling function: $E(N, M_0) = \mathcal{L}^{-1}(L(A))$

9.4.3 The library example

Let us take a simplified example of the case study. Here, we model a single user that can borrow maximum two books of a library which owns a stock of five books.

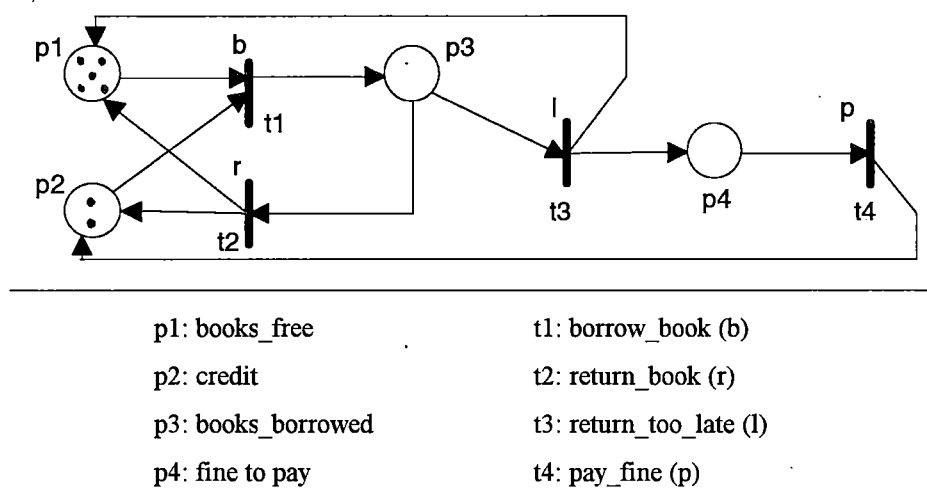


Figure 9.8 (the Petri net of a library)

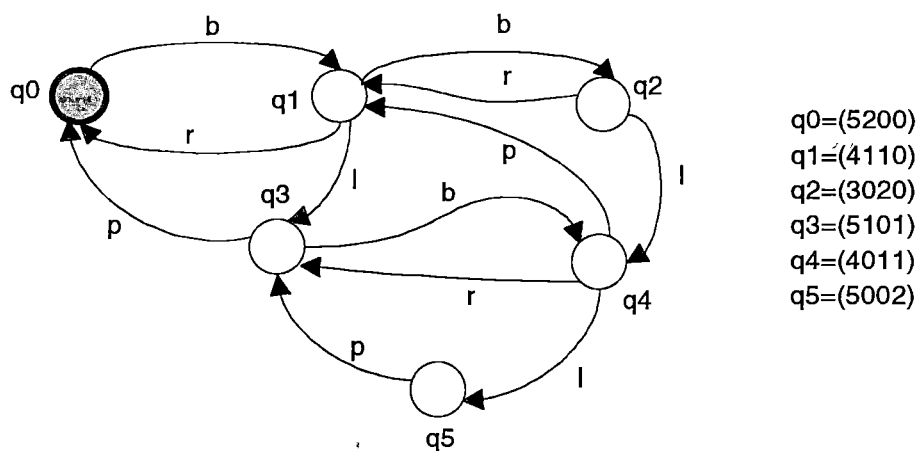


Figure 9.9 (the-corresponding automaton⁷)

⁷ Suppose that $M_0=(5\ 2\ 0\ 0)$ is also the accepting state.

The graph contains four finite cycles β from q_0 (to it-self) whose execution is $\sigma(\beta) = q_0 s_1 s_2 \dots s_{n-1} q_0$ with $\forall i \in [1, n-1], s_i \neq q_0$:

$$\beta_1 = b(br | bl(rb | lpb)^* p)^* r$$

$$\beta_2 = bl(br | blp)^* p$$

$$\beta_3 = bblr(br | blp)^* p$$

$$\beta_4 = bllp(br | blp)^* p$$

The language of the automaton is thus:

$$L(A) = \{w \in \Sigma^\omega : w ::= (\beta_1 | \beta_2 | \beta_3 | \beta_4)^\omega\}$$

And the set of possible executions of the Petri net is:

$$E(N, M_0) = \mathcal{L}^{-1}(L(A)) = \{s ::= (\mathcal{L}^{-1}(\beta_1) | \mathcal{L}^{-1}(\beta_2) | \mathcal{L}^{-1}(\beta_3) | \mathcal{L}^{-1}(\beta_4))^\omega\}$$

$$\text{with } \mathcal{L}^{-1}(\beta_1) = t_1(t_1t_2 | t_1t_3(t_2t_1 | t_3t_4t_1)^* t_4)^* t_2$$

$$\mathcal{L}^{-1}(\beta_2) = t_1t_3(t_1t_2 | t_1t_3t_4)^* t_4$$

$$\mathcal{L}^{-1}(\beta_3) = t_1t_1t_3t_2(t_1t_2 | t_1t_3t_4)^* t_4$$

$$\mathcal{L}^{-1}(\beta_4) = t_1t_1t_3t_3t_4(t_1t_2 | t_1t_3t_4)^* t_4$$

9.4.4 An algorithm

[AHO&ULLMAN93] propose an algorithm to catch the language of an automaton. The fact is that the language of the automata studied here can be formalized in a particular (much readable) form. Therefore we explain here another algorithm which gives the language $L(A)$ of an automaton A having an initial (q_0) and an accepting state (q_f), such as this language can be defined as :

$$L(A) = \{(\alpha_1 | \alpha_2 | \dots | \alpha_q)(\beta_1 | \beta_2 | \dots | \beta_p)^\omega\}$$

This algorithm consists in five successive steps :

Step 1: to identify, in the graph, the set C of elementary⁸ cycles which **never** pass through q_f .

⁸ A path (a cycle) is said elementary if no node (if no node except the last one) appear twice in the path (in the cycle).

Step 2: to identify the set P of elementary paths from q_0 to q_f

Step 3: to identify the set R of elementary cycles from q_f (to it-self).

Step 4: to build $(\alpha_1|\alpha_2|\dots|\alpha_q)$ by trying to insert the cycles of C in the paths of P.

Step 5: to build $(\beta_1|\beta_2|\dots|\beta_p)$ by trying to insert the cycles of C in the cycles of R.

Example :

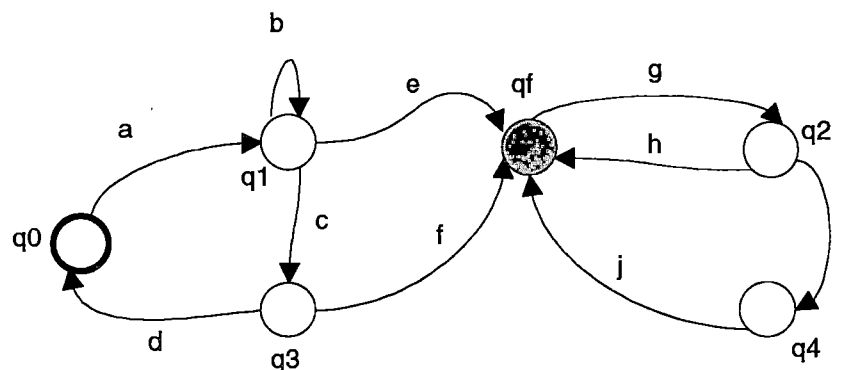


Figure 9.10 (an automaton)

Step1: The set C contains two elementary cycles : b (from q_1 to q_1) and acd (from q_0 to q_0)

Step 2: The set P contains two elementary paths : ae and acf

Step 3: The set R contains two elementary cycles : gh and gij

Step 4: - The path ae is combined with the two cycles of C :

$$ae \rightarrow (acd)^*ae \rightarrow (ab^*cd)^*ab^*e \quad \text{in order to form } \alpha_1 = (ab^*cd)^*ab^*e$$

- The path acf is also combined with the two cycles of C :

$$acf \rightarrow (acd)^*acf \rightarrow (ab^*cd)^*ab^*cf \quad \text{in order to form } \alpha_2 = (ab^*cd)^*ab^*cf$$

Step 5: No insertion of cycles of C is possible in the two cycles of R.

$$\text{Thus : } \beta_1 = gh \quad \text{and} \quad \beta_2 = gij$$

$$\text{The language is so : } L(A) = \left\{ w ::= ((ab^*cd)^*ab^*e \mid (ab^*cd)^*ab^*cf)(gh \mid gij)^\omega \right\}$$

Remark: It must be noticed that ,when inserting cycles in a path, we have to do it in a precise and intelligent order. Imagine that, in step 4, we would have first tried to insert the cycle b (instead of beginning with acd) in the path ae :

$$ae \rightarrow ab^*e \rightarrow (acd)^*ab^*e \quad \text{which is different than } \alpha_1 !$$

Here is now the complete algorithm:

Build-Language $(\Sigma, Q, \delta, q_0, q_f) \rightarrow L$

{ Return the language L of an automaton $A = (\Sigma, Q, \delta, q_0, q_f)$ }	
$C \leftarrow \text{Find_all_elementary_cycles}$	{Step 1}
Forall $w \in C$ do: if $q_f \in \sigma(w)$ then $C \leftarrow C / \{w\}$	
$P \leftarrow \text{Find_elementary_paths}(q_0, q_f)$	{Step 2}
$R \leftarrow \{ w \in C : \text{First}(\sigma(w)) = \text{Last}(\sigma(w)) = q_f \}$	{Step 3}
$\alpha \leftarrow \text{Insert_cycles}(P, C)$	{Step 4}
$\beta \leftarrow \text{Insert_cycles}(R, C)$	{Step 5}
$L \leftarrow (\alpha)(\beta)^{\omega}$	

Auxiliary procedures:

- The procedures *Find_all_elementary_cycles* and *Find_elementary_paths* are classical problems in the graph theory and therefore not explained here (the interested reader can refer to [GONDRAN&MINOUX79] for more about algorithms in graphs). The first procedure identifies all the elementary cycles of a graph; the second one returns a set of elementary paths between two nodes of a graph.

- The procedure *Insert_cycles* is defined as follows :

Insert-cycles $(W, C) \rightarrow \gamma$

{ (Try to) insert cycles of C in a each path of the set W to form γ }	
$q \leftarrow 0$	
For each $w \in W$ do:	
$C' \leftarrow \text{Sort}(C, w)$	
$q \leftarrow q+1$	
While $\text{Empty}(C') = \text{False}$ do :	
$c \leftarrow \text{First}(C')$	
if $\exists q: q \in \sigma(w)$ and $q \in \sigma(c)$ then $w \leftarrow \text{Merge}(w, c)$	
$C' \leftarrow \text{Sort}(C' / \{c\}, w)$	
if $q=1$ then $\gamma \leftarrow w$ else $\gamma \leftarrow \gamma + ']' + w$	

and uses two procedures (*Merge* and *Sort*) :

- The procedure *Sort*(C, w) consists in sorting a set C of elementary cycles in a precise manner (see remark p. 9.12) before trying to merge those cycles and the path w . The sorting is based on the number of different nodes that the cycles and the path w have in common :

$$c_i \leq c_j \ (c_i, c_j \in C) \Leftrightarrow \#\{q \in Q: q \in \sigma(w) \text{ and } q \in \sigma(c_i)\} \geq \#\{q \in Q: q \in \sigma(w) \text{ and } q \in \sigma(c_j)\}$$

- The procedure *Merge* is defined as follows :

Merge (w, c) $\rightarrow w'$

{ Merge a path w and an elementary cycle c into a new path w' }

$c = a_1 a_2 \dots a_n$

$\sigma(c) = s_0 s_1 s_2 \dots s_n$

$\bar{Q} \leftarrow \{q \in Q: q \in \sigma(w) \text{ and } q \in \sigma(c)\}$

For each $q \in \bar{Q}$

do:

$i \leftarrow k \in \mathbb{N}: s_k = q$

$Part1 \leftarrow SubPath(w, First(\sigma(w)), q)$

$Part2 \leftarrow SubPath(w, q, Last(\sigma(w)))$

$w \leftarrow Part1 + (a_{i+1} a_n a_1 \dots a_i)^* + Part2$

$w' \leftarrow w$

9.4.5 A more general algorithm

The assumption of an unique accepting state is too restrictive and not realistic. Indeed, since we model Petri nets in which choices are possible (like in figure 9.11), it is not always possible to identify a marking through which all executions are passing infinitely often.

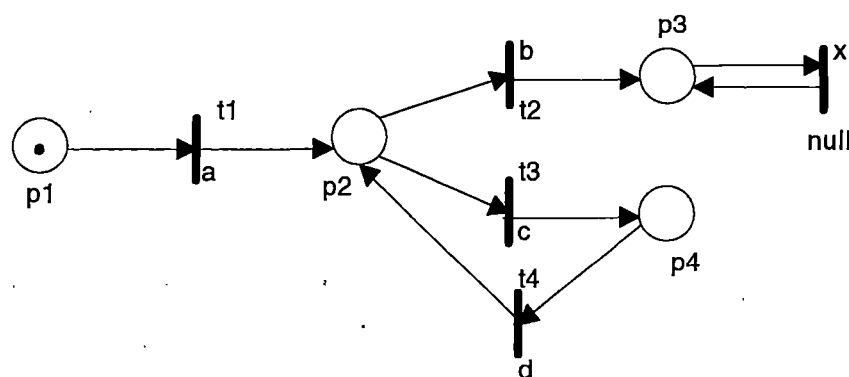


Figure 9.11 (another Petri net)

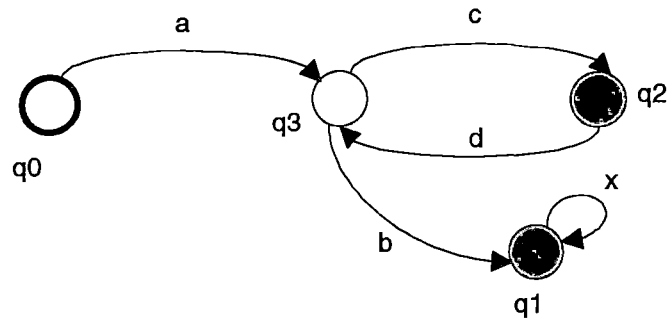


Figure 9.12 (its corresponding automaton)

It is thus necessary to refine the algorithm to take account of this. Simply, we can re-use the previous algorithm (*Build-Language*) by calculating the language of the automaton successively with the different accepting states :

Catch-Language $(\Sigma, Q, \delta, q_0, F) \rightarrow L$

{ Return the language L of an automaton $A = (\Sigma, Q, \delta, q_0, F)$ }

$L \leftarrow \emptyset$

Forall $q_i \in F$ do:

$L_i \leftarrow \text{Build-Language}(\Sigma, Q, \delta, q_0, q_i)$

$L \leftarrow L \cup L_i$

This gives for the automaton of figure 9.12 :

$$L = L_1 \cup L_2$$

$$L_1 = a(cd)^* bx^0$$

$$L_2 = a(cd)^0$$

9.4.6 Mapping Petri nets into Büchi automata

A difficult problem met when transforming a Petri net into a Büchi automaton is the choice of the accepting states : which markings are going to be considered as accepting states ? The easiest solution consists in considering *each* marking of the accessibility set as an accepting state (that is to say $Q=F$); this means that every execution of a Petri net must pass infinitely often through (at least) one of those markings. Don't forget that we want the net to be as general as possible, and to express the additional constraints separately.

But the problem is that the execution time of the previously explained algorithm is proportional to the number of accepting states of the automaton. Therefore, we should try to restrict the set of accepting states (F) to its minimum, without reducing the set of possible executions (a wrong choice of the accepting states may, in some cases, have the same effect as a constraint). Consider the following automaton:

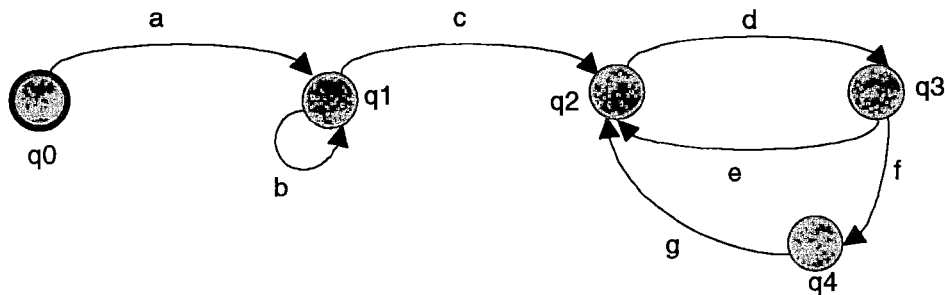


Figure 9.13 (considering all states as accepting states)

It is not necessary to consider q_3 and q_4 as accepting states, since considering q_2 as accepting state forces the executions to pass infinitely often through one of these two states. It is also not useful to include q_0 in the set of accepting states, because it does not belong to a cycle. More generally, we can say that a state q_i can be removed from the set of accepting states in two cases :

1. if there is no self-loop on this state, and if *all* elementary cycles from q_i (to itself) are passing through another accepting state.

ex: the state q_3

2. if there exists no cycle from q_i (to it-self).

ex: the state q_0

9.5 Reduction of the set of possible executions

Since we want to associate to each Petri net some constraints (expressed by a set of temporal logic formulae), it is necessary to reduce the set of *possible* executions into a set of *desired* executions respecting those formulae. Among the possible words accepted by the automaton, some of them have to be excluded, because their execution would result in a state forbidden by the temporal logic formulae. In this section, we won't study all kinds of constraints of PNTL; we only consider the constraints of the form " $\psi \rightarrow [X, F, G]\phi$ " where X, F and G are respectively the *next*, *sometimes (eventually)* and *always (henceforth)* operator. To simplify⁹, we assume that ψ and ϕ are atomic formulae.

9.5.1 Hypothesis

1. (N, M_0) is a marked Petri net (with M_0 as initial marking) and A is its corresponding automaton.

2. In the following, we suppose that the language of A is the union of n languages :

$$L(A) = \bigcup_{j=1}^n L_j$$

$$\text{with } L_j = \{w \in \Sigma^\omega : w ::= (\alpha_{j1}|\alpha_{j2}|\dots|\alpha_{jq})(\beta_{j1}|\beta_{j2}|\dots|\beta_{jp})^\omega\}$$

(where α_{ji} and β_{ji} are words on Σ)

and contains the set of *possible* executions of the Petri net :

$$E(N, M_0) = \mathcal{L}^{-1}(L(A))$$

3. This means that the execution of any word w of $L(A)$ belongs to

$$\sigma(A) = \bigcup_{j=1}^n \sigma_j$$

$$\sigma_j = \{\sigma \in \mathcal{Q}^\omega : \sigma ::= \sigma_0(\sigma_{j1}|\sigma_{j2}|\dots|\sigma_{jq})(\sigma_{j(q+1)}|\sigma_{j(q+2)}|\dots|\sigma_{j(q+p)})^\omega\}$$

where σ_0 is a sequence of one element: the initial state (q_0),

$\forall i: 1 \leq i \leq q, \sigma_{ji} = \sigma(\alpha_{ji})$ without its first element (q_0),

$\forall i: q < i \leq q + p, \sigma_{ji}$ is the execution of β_{ji} without its first element, with

accepting state as initial state of the automaton.

⁹ But it can be easily generalized for formulae. Indeed a formulae is a composition (conjunction, disjunction,...) of atomic formulae and we have seen that formal languages have closure properties (intersection, union, ...)

4. The set C contains the constraints attached to the Petri net:

$$C = \{c_1, c_2, \dots, c_r\}$$

We want to obtain a language $L'(A) \subseteq L(A)$ such as $E_a(N, M_0) = \mathcal{L}^{-1}(L'(A))$ contain all the *desired* executions of the net.

9.5.2 Constraints on states and constraints on words

According to the predicate or function involved in the atomic formulae ψ and ϕ , the truth value of those formulae is going to be defined by reasoning on the language $L(A)$ of the automaton, and/or on its possible sequences $\sigma(w)$ of states. We have thus two kinds of formulae: formulae on words for which we will reason on the language, and formulae on states for which we will reason on the sequences of execution of the automaton. For instance, "*Fired*(t_1)" deals with the words of the language, whereas " $m(p_2) > 0$ " is a condition on the states of the automaton.

9.5.3 Reduction when dealing with a constraint on words

To ensure the respect of this type of constraints, we are first going to define a language $L(c)$ (on the same alphabet as the language of A) whose all words are respecting the constraint $c \equiv \psi \rightarrow [X, F, G]\phi$. Then, thanks to the closure properties of the ω -regular languages (see section 9.3.4), we can calculate the intersection between this language and the one of the automaton A :

$$L'(A) = L(A) \cap L(c)$$

The intersection¹⁰ of two languages L_1 and L_2 is defined as follows:

$$L_1 \cap L_2 = \{w \in (\Sigma_1 \cap \Sigma_2)^\omega : (w \in L_1) \text{ and } (w \in L_2)\}$$

In the following, we suppose that $\psi \equiv \text{Fired}(t_i)$ and $\phi \equiv \text{Fired}(t_j)$. Let us now see the form of the language $L(c)$ according to the temporal operator of the right hand-side of the constraint:

¹⁰ See section 9.5.5 to see how we can calculate the intersection of two languages .

- $c \equiv \text{Fired}(t_i) \rightarrow \mathbf{X}(\text{Fired}(t_j))$

This means that each time the symbol $\mathcal{L}(t_i)$ is met in a word, it must always be *immediately* followed by the symbol $\mathcal{L}(t_j)$. Thus the language of the constraint can be defined as:

$$L(c) = \{w \in \Sigma^\omega : w ::= (ab \mid \$)^\omega\}$$

where $a = \mathcal{L}(t_i)$

$b = \mathcal{L}(t_j)$

$\$ =$ any symbol of Σ except the symbol a .

$= (x_1 \mid x_2 \mid \dots \mid x_n)$ with $x_i \in (\Sigma / \{a\})$

- $c \equiv \text{Fired}(t_i) \rightarrow \mathbf{F}(\text{Fired}(t_j))$

This means that each time the symbol $\mathcal{L}(t_i)$ is met in a word, the symbol $\mathcal{L}(t_j)$ must always be found "further" in this word. Thus the language of the constraint can be defined as:

$$L(c) = \{w \in \Sigma^\omega : w ::= (a\$^*b \mid \$)^\omega\}$$

where $a = \mathcal{L}(t_i)$

$b = \mathcal{L}(t_j)$

$\$ =$ any symbol of Σ except the symbol a .

$= (x_1 \mid x_2 \mid \dots \mid x_n)$ with $x_i \in (\Sigma / \{a\})$

- $c \equiv \text{Fired}(t_i) \rightarrow \mathbf{G}(\text{Fired}(t_j))$

This means that if the symbol $\mathcal{L}(t_i)$ is met in a word, *all* the following symbols must be $\mathcal{L}(t_j)$. Thus the language of the constraint can be defined as:

$$L(c) = \{w \in \Sigma^\omega : w ::= (\$^*ab^\omega \mid \$^\omega)\}$$

where $a = \mathcal{L}(t_i)$

$b = \mathcal{L}(t_j)$

$\$ =$ any symbol of Σ except the symbol a .

$= (x_1 \mid x_2 \mid \dots \mid x_n)$ with $x_i \in (\Sigma / \{a\})$

Examples :

Let's take again the example of the small library (section 9.4.3). We saw that the language of its corresponding automaton was:

$$L(A) = \{w \in \Sigma^\omega : w ::= (\beta_1 \mid \beta_2 \mid \beta_3 \mid \beta_4)^\omega\}$$

$$\beta_1 = b(br \mid bl(rb \mid lpb)^* p)^* r$$

with $\beta_2 = bl(br \mid blp)^* p$

$$\beta_3 = bblr(br \mid blp)^* p$$

$$\beta_4 = bllp(br \mid blp)^* p$$

1. Suppose that we have a constraint stating that if a book is returned late, the fine must always be paid :

$$Fired(t_3) \rightarrow F(Fired(t_4))$$

Since $\mathcal{L}(t_3)=l$ and $\mathcal{L}(t_4)=p$, we have :

$$L(c) = \{w \in \Sigma^\omega : w ::= (l(b|r|p)^* p \mid (b|r|p))^\omega\}$$

And it is easy to see that all the words of $L(A)$ have the wanted form :

$$L(A) = L(A) \cap L(c) \text{ since } L(A) \subset L(c)$$

2. Suppose now that a constraint forces the borrowers to immediately pay the fine when a book is returned late :

$$Fired(t_3) \rightarrow X(Fired(t_4))$$

Since $\mathcal{L}(t_3)=l$ and $\mathcal{L}(t_4)=p$, we have :

$$L(c) = \{w \in \Sigma^\omega : w ::= (lp \mid (b|r|p))^\omega\}$$

This time, we obtain a sub-language of $L(A)$:

$$L'(A) = L(A) \cap L(c) = \{w \in \Sigma^\omega : w ::= (b(br|blp)^* (r|lp))^\omega\}$$

9.5.4 Reduction when dealing with a constraint on states

We saw that the execution of a Büchi automaton is an infinite sequence of states. We can consider such sequence as an infinite word on a particular alphabet Q . Since we know the form of every sequence of execution of A (see hypothesis in section 9.5.1), we have already defined the language - that is $\sigma(A)$ - which contains all this words. Thus, like for constraints on words, we can translate a constraint $c \equiv \psi \rightarrow [X, F, G]\phi$ into a language $\sigma(c)$, and then calculate its intersection with the language $\sigma(A)$:

$$\sigma'(A) = \sigma(A) \cap \sigma(c)$$

When we get the set of desired executions¹¹ of the automaton, it still must be translated in terms of words accepted by the automaton. In other words, knowing all the desired sequences of states of the automaton, we have to restore a set of desired and accepted words. This translation¹² can be done thanks to the next-state function (δ) of the automaton : two successive states (q_i, q_j) can be mapped into a union of transitions linking the state q_i to q_j .

The first step consists in determining the states which verify the left and right hand-side of the constraint :

$$\bar{Q} = \{q \in Q : \psi(q) = TRUE\} = \{\bar{q}_1, \dots, \bar{q}_n\} \quad (n \geq 0)$$

$$\hat{Q} = \{q \in Q : \phi(q) = TRUE\} = \{\hat{q}_1, \dots, \hat{q}_m\} \quad (m \geq 0)$$

Some particular situations can already be identified : if $\bar{Q} = \emptyset$ or $\hat{Q} = Q$, the constraint is useless or redundant since it is always verified.

In the following, we suppose that $\bar{\sigma}$ and $\hat{\sigma}$ are respectively the union of states of \bar{Q} , and the union of states of \hat{Q} :

$$\bar{\sigma} = (\bar{q}_1 \mid \bar{q}_2 \mid \dots \mid \bar{q}_n)$$

$$\hat{\sigma} = (\hat{q}_1 \mid \hat{q}_2 \mid \dots \mid \hat{q}_m)$$

Let's now see the form of the language $\sigma(c)$ according to the temporal operator of the right hand-side of the constraint :

¹¹ Do not confuse the set of executions of the Petri net (which corresponds to a set of words on Σ) and the set of executions of the automaton (which corresponds to a set of words on Q).

¹² The translation is valid here because we cannot have, in the automaton, a state having two output arcs with the same label (indeed, the firing of a transition in a Petri net always leads to a unique marking).

- $c \equiv \psi \rightarrow \mathbf{X}(\varphi)$

The language of the constraint can be defined as:

$$\sigma(c) = \{ \sigma \in Q^\omega : \sigma ::= (\overline{\sigma} \hat{\sigma} \mid \$)^\omega \}$$

where $\$ = (x_1 \mid x_2 \mid \dots \mid x_r)$ with $x_i \in (Q / \overline{Q})$

- $c \equiv \psi \rightarrow \mathbf{F}(\varphi)$

The language of the constraint can be defined as:

$$\sigma(c) = \{ \sigma \in Q^\omega : \sigma ::= (\overline{\sigma} \$^* \hat{\sigma} \mid \$)^\omega \}$$

where $\$ = (x_1 \mid x_2 \mid \dots \mid x_r)$ with $x_i \in (Q / \overline{Q})$

- $c \equiv \psi \rightarrow \mathbf{G}(\varphi)$

The language of the constraint can be defined as:

$$\sigma(c) = \{ \sigma \in Q^\omega : \sigma ::= (\$^* \overline{\sigma} \hat{\sigma}^\omega \mid \$^\omega) \}$$

where $\$ = (x_1 \mid x_2 \mid \dots \mid x_r)$ with $x_i \in (Q / \overline{Q})$

Example :

We saw that any execution of the automaton modeling the small library belongs to the set:

$$\sigma(A) = \{ \sigma \in Q^\omega : \sigma ::= \sigma_0 (\sigma_1 \mid \sigma_2 \mid \sigma_3 \mid \sigma_4)^\omega \}$$

$$\sigma_0 = q_0$$

$$\sigma_1 = q_1 (q_2 q_1 \mid q_2 q_4 (q_3 q_4 \mid q_5 q_3 q_4)^* q_1)^* q_0$$

with $\sigma_2 = q_1 q_3 (q_4 q_3 \mid q_4 q_5 q_3)^* q_0$

$$\sigma_3 = q_1 q_2 q_4 q_3 (q_4 q_3 \mid q_4 q_5 q_3)^* q_0$$

$$\sigma_4 = q_1 q_2 q_4 q_5 q_3 (q_4 q_3 \mid q_4 q_5 q_3)^* q_0$$

Suppose that as soon as a borrower has returned a book too late, he can borrow, from this moment, only one book anymore (instead of two) :

$$m(p_4) = 1 \rightarrow \mathbf{G}(m(p_2) > 0)$$

We have :

$$\begin{aligned}\bar{Q} &= \{q_3, q_4\} \text{ and } \bar{\sigma} = (q_3 \mid q_4) \\ \hat{Q} &= \{q_0, q_1, q_3\} \text{ and } \hat{\sigma} = (q_0 \mid q_1 \mid q_3) \\ \$ &= (q_0 \mid q_1 \mid q_2 \mid q_5)\end{aligned}$$

And

$$\sigma(c) = \{\sigma \in Q^\omega : \sigma ::= (\$^* \bar{\sigma} \hat{\sigma}^\omega \mid \$^\omega)\}$$

And so :

$$\begin{aligned}\sigma'(A) &= \sigma(A) \cap \sigma(c) \\ &= \{\sigma' \in Q^\omega : \sigma' ::= (\sigma'_1 \mid \sigma'_2)\} \\ \sigma'_1 &= q_0(q_1(q_2q_1)^*q_0)^\omega \\ \sigma'_2 &= q_0(q_1(q_2q_1)^*q_0)^* \left((q_1q_3q_0 \mid q_1q_0)^\omega \mid q_1q_2q_4(q_3q_0 \mid q_1(q_0 \mid q_3q_0))(q_1q_0 \mid q_1q_3q_0)^\omega \right)\end{aligned}$$

Finally, this means that the desired language of the automaton is :

$$\begin{aligned}L'(A) &= \{w \in \Sigma^\omega : w ::= (\gamma_1 \mid \gamma_2)\} \\ \gamma_1 &= (b(br)^*r)^\omega \\ \gamma_2 &= (b(br)^*r)^* \left((blp \mid br)^\omega \mid bbl(rp \mid p(r \mid lp))(br \mid blp)^\omega \right)\end{aligned}$$

9.5.5 Calculating the intersection of two languages

We said (in section 9.3.4) that the class of ω -regular languages was closed in respect with (among others) the operator of intersection. To ensure the respect of a constraint associated to a Petri net, we have proposed to calculate the intersection of the language of this constraint and the language of the automaton modeling the Petri net. Let's now proof that the intersection is always calculable and see how we can calculate it :

Theorem.

If $A_1 = (\Sigma, Q_1, \delta_1, I_1, F_1)$ and $A_2 = (\Sigma, Q_2, \delta_2, I_2, F_2)$ are two Büchi automata, then it is always possible to build a GBA $= (\Sigma, Q, \delta, I, \bar{F})$ which accepts the language $L(A_1) \cap L(A_2)$

Proof.

We build a GBA whose states are 2-tuples of states of the two automata, and whose transitions are the ones that are possible in both automata. The set \bar{F} simply contains the two sets F_1 and F_2 . More formally :

$$Q = Q_1 \times Q_2$$

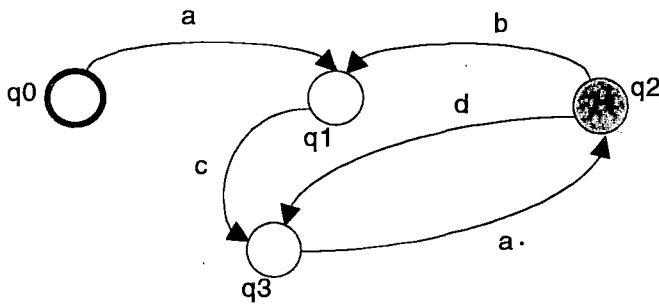
$$I = I_1 \times I_2$$

$$\bar{F} = \{F_1 \times Q_2, F_2 \times Q_1\}$$

$$q = \langle u, v \rangle \in \delta(\langle s, t \rangle, a) \text{ if } u \in \delta_1(s, a) \text{ and } v \in \delta_2(t, a)$$

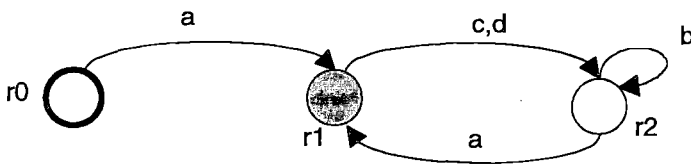
Example:

A1



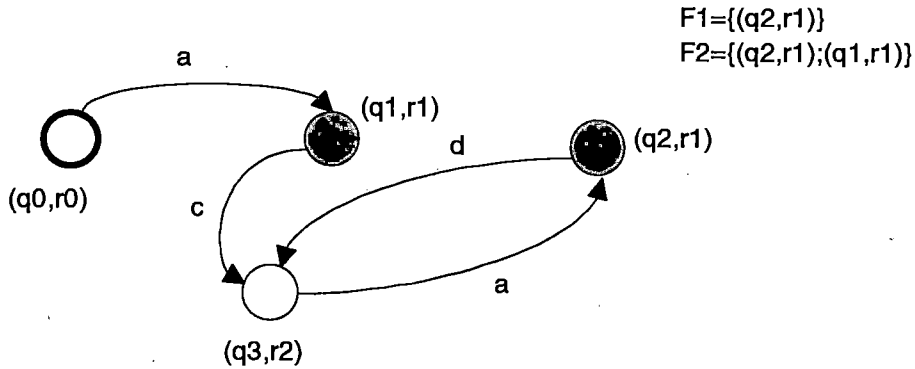
$$L(A_1) = \{w ::= aca(da|bca)^\omega\}$$

A2



$$L(A_2) = \{w ::= a(cb^*a|db^*a)^\omega\}$$

The intersection of the automata A_1 and A_2 is :



which accepts the language $L(A_1) \cap L(A_2) = \{w ::= ac(ad)^\omega\}$

9.6 Testing of properties on the set of desired executions

As explained in [MANNA&PNUELI84], we can partition most of temporal properties of programs into two classes. These properties can be characterized by the form of the temporal formulas expressing them :

- The class of *invariance* properties. These properties can be expressed by a formula of the form :

$$(a) \Box\psi \text{ or } (b) \phi \supset \Box\psi$$

The formula (a) says that the condition ψ is always true, while the other (b) states that whenever ϕ becomes true, ψ is immediately true and remains true in the future.

- The class of *liveness* properties. These properties can be expressed by a formula of the form:

$$(c) \Diamond\psi \text{ or } (d) \phi \supset \Diamond\psi$$

In both cases, the formulas guarantee the occurrence of a condition ψ . The only difference is that the first one (c) guarantees it unconditionally, whereas the second one (d) is conditional on an earlier occurrence of event ψ .

Although we do not reason on programs but on sequences of symbols and on sequences of words, it is obvious that most of the properties of the specification we want to test (namely for reasons of completeness), are also of this form - or can be brought to this form. But in order to test a property on the language of an automaton, we first have to transform it into a constraint whose syntax is the one of PNTL :

- Considering that any Petri net contains at least one place and that the number of tokens is a positive integer, the formula (a) can be translated in an implication - whose left hand-side is always true:

$$m(p1) \geq 0 \rightarrow G(\psi)$$

- For the same reason, the formula (c) gives:

$$m(p1) \geq 0 \rightarrow F(\psi)$$

- The formula (b) becomes :

$$\varphi \rightarrow G(\psi)$$

- The formula (d) becomes :

$$\varphi \rightarrow F(\psi)$$

We can take advantage of the algorithms that reduce the set of possible executions to the set of desired executions (see section 9.4). Indeed, there is an easy way to test a property. It simply consists in adding the property in the list of constraints attached to the net (we do as if it was a constraint), then in recalculating the language of the automaton to finally compare this language with the one obtained without the property. If they are equal¹³, we can say that the property is always verified (see figure 9.15).

Example :

Consider that the set C of constraints contains only one constraint :

$$Fired(t3) \rightarrow X(Fired(t4))$$

We saw (see section 9.5.3) that the set of desired executions of the library was :

$$L'(A) = \left\{ w \in \Sigma^\omega : w ::= (b(br|blp)^*(r|lp))^\omega \right\}$$

¹³ It must be pointed out that a same language can be expressed by several different ω -regular expressions. The question of knowing if two languages are equal is not as obvious as it seems to be !

and $\sigma'(A) = \left\{ \sigma \in Q^\omega : \sigma ::= q_0(q_1(q_2q_1|q_2q_4q_1)^*(q_0|q_3q_0))^\omega \right\}$

Suppose we want to test the following property : $p \equiv \square(m(p_4) \leq 1)$

Thus $\sigma(p) = \left\{ \sigma \in Q^\omega : \sigma ::= (q_0 | q_1 | q_2 | q_3 | q_4)^\omega \right\}$

Since $\sigma'(A) \cap \sigma(p) = \sigma'(A)$, the language $L'(A)$ is unchanged. Therefore, the property p is verified.

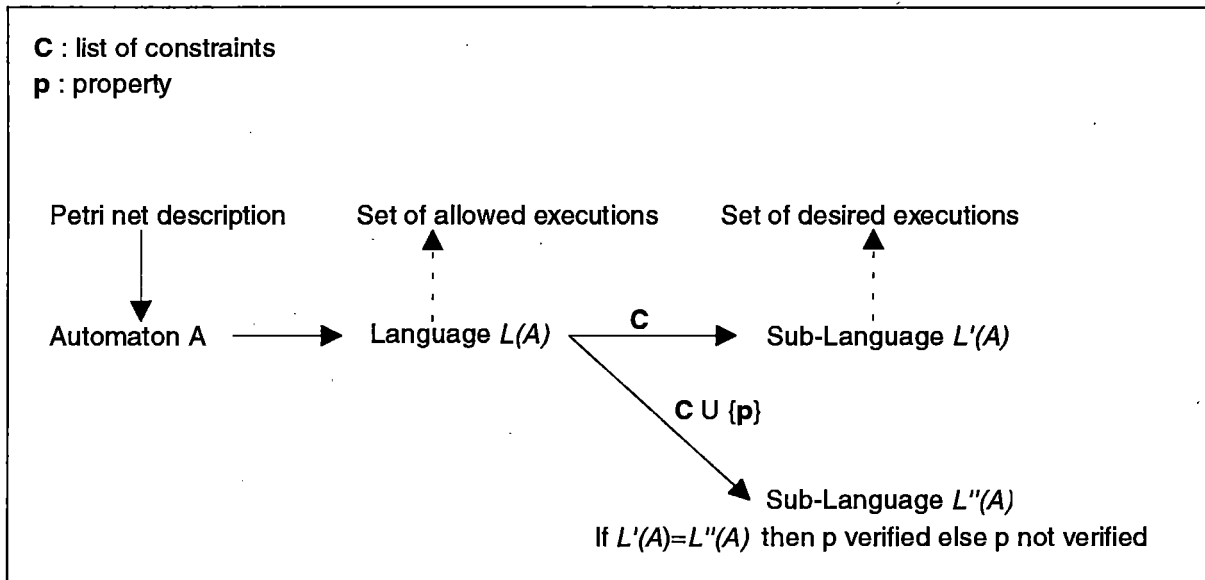


Figure 9.14 (the testing of a property)

9.7 Extension to the linear temporal logic

9.7.1 About the expressiveness of the linear temporal logic

Consider the set of sequences for which the proposition p is true in all even states (nothing is said about the odd states). This set contains among others the two following sequences :

$$\sigma_1 = p \neg p p \neg p p \neg p p \neg p p \neg p \dots$$

$$\sigma_2 = p p p \neg p p \neg p p \neg p p \neg p \dots$$

It seems quite surprising that the property which defines this set of sequences - and which we will denote $Even(p)$ - cannot be characterized by any formula of the temporal logic [THAYSE89]. One could think that the two following formulae are appropriate :

$$f_1 \equiv p \wedge \Box(p \supset \bigcirc \neg p) \wedge \Box(\neg p \supset \bigcirc p)$$

$$f_2 \equiv p \wedge \Box(p \supset \bigcirc \bigcirc p)$$

But it is not the case, since they both are false for the sequence σ_2 . Note that this property can be expressed in PNTL :

$$Init \rightarrow Even$$

$$Even \rightarrow X(X(Even))$$

$$Init \rightarrow X(\neg Even)$$

$$\neg Even \rightarrow X(X(\neg Even))$$

$$Even \rightarrow p$$

We now give a more general way to express such properties in what is called *extended temporal logic*.

9.7.2 Extended temporal logic

It is possible to extend the expressiveness of the linear temporal logic in order to express properties like $Even(p)$. We can add to the temporal logic all the operators and properties that are expressible by means of a finite automaton.

As in [THAYSE89], we will associate to each temporal operator a Büchi automaton. Since an interpretation in linear temporal logic is an infinite sequence, we can check if this sequence verify a property by answering the question "is the execution of this sequence an admitted execution on the automaton associated to this property ?"

Let $B = (\Sigma, Q, \delta, I, F)$ be a Büchi automaton. To define an operator, we associate a formula to *each* element of Σ . Those formulae will be the arguments of the operator. Thus if $\Sigma = \{a_1, a_2, \dots, a_n\}$, the operator will have n arguments. The application of the operator on the formula f_1, f_2, \dots, f_n will be noted $B(f_1, f_2, \dots, f_n)$. The semantics of $B(f_1, f_2, \dots, f_n)$ is defined in terms of accepted words of B . A sequence σ verify $B(f_1, f_2, \dots, f_n)$ if there exists a word accepted by B , such as if the symbol a_i is met in position j , the the formula f_i is true in this position. More formally :

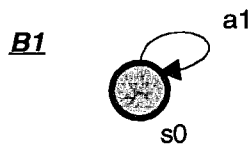
$$I \models_{\sigma} B(f_1, f_2, \dots, f_n) \text{ if } \exists w = a_{i_0}, a_{i_1}, a_{i_2}, \dots \text{ accepted by } B$$

$$\text{such as } \forall j \geq 0, I \models_{R(\sigma)}^j f_{i_j}$$

Let's see, for instance, how we can (re)define the temporal operator (**X**, **F**, **G**) and the property *Even(p)* :

Always operator (G):

$$G(f) = B_1(f)$$

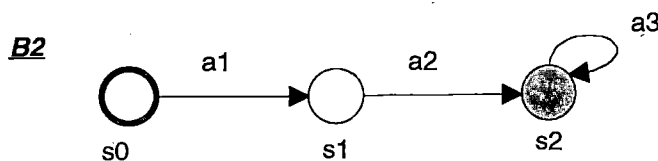


$$L(B_1) = \{w ::= a_1^{\omega}\}$$

The words accepted by B_1 are the ones which contain only the symbol a_1 . Thus the only accepted interpretations are the ones in which $I(f)$ is always true.

Next operator (X):

$$X(f) = B_2(T, f, T)$$

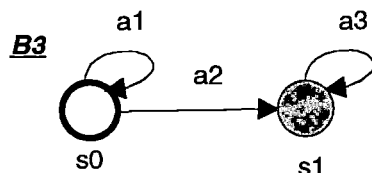


$$L(B_2) = \{w ::= a_1 a_2 a_3^{\omega}\}$$

The words accepted by B_2 are the ones whose second symbol is a_2 . Thus the only accepted interpretations are the ones in which $I(f)=TRUE$ in the next state.

Sometimes operator (F):

$$F(f)=B_3(T, f, T)$$

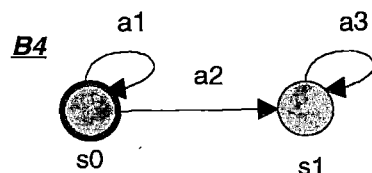


$$L(B_3) = \{w ::= a_1^* a_2 a_3^{\omega}\}$$

The words accepted by B_3 are the ones which contains at least once the symbol is a_2 . Thus the only accepted interpretations are the ones in which $I(f)=TRUE$ at least once.

Until operator (U):

$$f_1 U f_2 = B_4(f_1, f_2, T)$$

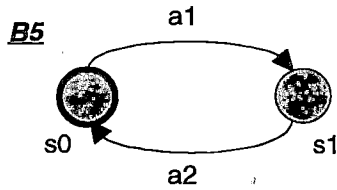


$$L(B_4) = \{w ::= (a_1^{\omega} \mid a_1^* a_2 a_3^{\omega})\}$$

The words accepted by B_4 are either words that contains exclusively the symbol is a_1 , or words in which a finite repetition of a_1 is immediately followed by a_2 . Thus the only accepted interpretations are either the ones where $I(f_1)$ is always true, or the ones in which f_1 holds till the moment $I(f_2)=TRUE$.

Even(f):

Even(f)=B₅(f, T)



$$L(B_5) = \{w ::= (a_1 a_2)^\omega\}$$

The words accepted by B₅ are the infinite repetitions of the finite (sub)word a₁a₂. Thus the only accepted interpretations are the ones in which I(f₁)=TRUE in all even states, and I(T)=TRUE in all odd states.

Chapter 10

Logical Proofs in PNTL and PNRTL

10.1 Introduction

In chapter 9, we have seen how the theory of Büchi automata can be used to make proofs of properties of a PNTL specification. Those proofs were conducted by verifying that the formal language associated with the PNTL specification did or did not verify the properties. Here, we define a logical framework to allow usual syntactic logical proofs in PNTL and PNRTL. To make possible those syntactic logical proofs, we must define a way to transform the net information (its structure and the firing rule) into logical formulae and provide a proof system. The logical formulae of a PNTL specification will be used as axioms in the logical proof system. The same schemata will be conducted for the PNRTL language.

In this chapter, section 10.2 concentrates on the PNTL language while section 10.3 concentrates on the real-time features of the PNRTL language. Section 10.4 concludes the chapter.

10.2 Logical proofs in PNTL

In this section we define a temporal proof system for our PNTL language. As suggested in [MP81], [MP83a], [MP83b], we distinguish three parts in our proof system :

- **Part A (the uninterpreted logic part)** : this part defines an axiomatic system for linear temporal logic. Theorems proved by part A are valid for all interpretations given to the predicate and function symbols appearing in the theorems.
- **Part B (the domain part)** : this part contains a set of axioms which depend on the domain, here the Petri nets. In this part, we formalize the firing rule and give axioms over the positive integer numbers since the function symbol m is interpreted as the marking function. Therefore, part B restricts the class of considered models to those in which all predicate and function symbols have a fixed interpretation and the individual variables range over fixed domains.
- **Part C (the net part)** : this part is constituted of a serie of axioms which translate the net structure in logic axioms. This part further reduces the class of models into the ones which reflect the set of possible executions of some Petri net, the Petri net of the PNTL specification on which we want to make a proof.

Let us now define the content of each part :

10.2.1 Part A : the pure logic part

This part contains :

- the axioms of propositional logic. In the proofs, references to those axioms will be denoted by **ProL** [AHO&ULLMAN93].
- the axioms of first order predicate logic with equality. In the proofs, references to those axioms are denoted by **PreL**. [AHO&ULLMAN93]
- the modus ponens inference rule **MP** : if $A \rightarrow B$ and A then infer B .
- the axioms of temporal linear logic [MP83a] :

axioms for the futur operators

- TLL1. $X(A \rightarrow B) \rightarrow (X(A) \rightarrow X(B))$
- TLL2. $\neg X(A) \leftrightarrow X(\neg A)$
- TLL3. $F(A) \leftrightarrow \neg G(\neg A)$
- TLL4. $G(A \rightarrow B) \rightarrow (G(A) \rightarrow G(B))$
- TLL5. $G(A) \rightarrow A$
- TLL6. $G(A) \rightarrow X(A)$
- TLL7. $G(A) \rightarrow X(G(A))$
- TLL8. $G(A \rightarrow X(A)) \rightarrow (A \rightarrow G(A))$
- TLL9. $(A)U(B) \rightarrow (B \vee (A \wedge X((A)U(B))))$
- TLL10. $(A)U(B) \rightarrow F(B)$
- TLL11. $[C \wedge G(C \rightarrow (B \vee (A \wedge X(C))))] \rightarrow (A)U(B)$

axioms for the past operators

- TLL12. $Y(A \rightarrow B) \rightarrow (Y(A) \rightarrow Y(B))$
- TLL13. $Init \leftrightarrow Y(\perp)$
- TLL14. $(\neg Init \wedge Y(\neg A)) \leftrightarrow \neg Y(A)$
- TLL 15. $P(A) \leftrightarrow \neg H(\neg A)$
- TLL 16. $H(A \rightarrow B) \rightarrow (H(A) \rightarrow H(B))$
- TLL 17. $H(A) \rightarrow A$
- TLL 18. $H(A) \rightarrow Y(A)$
- TLL 19. $H(A) \rightarrow Y(H(A))$
- TLL 20. $H(A \rightarrow Y(A)) \rightarrow (A \rightarrow H(A))$

relations over future and past operators

- TLL 21. $X(Y(A)) \leftrightarrow A$
 - TLL 22. $A \rightarrow F(A)$
 - TLL 23. $X(A) \rightarrow F(A)$
 - TLL 24. $A \rightarrow P(A)$
 - TLL 25. $\neg Init \wedge Y(A) \rightarrow P(A)$
 - TLL 26. $(Init \rightarrow A) \rightarrow (Y(X(A)) \leftrightarrow A)$
- The necessity rule **NR** : if A then infer $G(A)$ and $H(A)$.
 - The invariance rule **INV** : if $A \rightarrow X(A)$ then infer $A \rightarrow G(A)$.
 - The past invariance rule **INV-P**: if $A \rightarrow Y(A)$ then infer $A \rightarrow H(A)$
 - The initialized invariance rule **I-INV** : if $Init \rightarrow A$ and $A \rightarrow X(A)$ then infer $G(A)$.

10.2.2 Part B : the domain part

This part contains the translation of the firing rule and the axioms for $(\mathbb{N}, +, <)$

- Translation of the firing rule :
 - FR1. $Fired(t) \rightarrow Enabled(t)$. In other words, a transition is enabled when it fires.
 - FR2. $Fired(t_1) \wedge (t_1 \neq t_2) \rightarrow \neg Fired(t_2)$. There is only one transition that fires in a state.
 - FR3. $\exists t \in T : Fired(t) \vee Fired(null)$. In each state, there is a transition which is fired.
 - FR4. $Fired(null) \leftrightarrow \forall t \in T : \neg Enabled(t)$. The *null* transition is fired iff there is no transition of T which is enabled.
 - FR5. $Enabled(t) \leftrightarrow \forall p \in P : m(p) \geq pre(p, t)$. Definition of an enabled transition.
 - FR6. $Fired(t) \wedge m(p_i) = k_i \rightarrow X(m(p_i) + pre(p_i, t) = k_i + post(p_i, t))$. Where $pre, post$ and k_i stay constant in the next state, only $m(p_i)$ is variable. We must also consider the case of the *null* transition :

$$Fired(null) \wedge m(p_i) = k_i \rightarrow X(m(p_i) = k_i)$$

- FR7. The frame axiom :

$$\begin{aligned} & \text{Fired}(t) \wedge m(p_i) = k_i \wedge (\text{pre}(p_i, t) = 0 \wedge \text{post}(p_i, t) = 0) \\ & \rightarrow X (m(p_i) = k_i) \end{aligned}$$

It should be noted that FR7 is a logical consequence of FR6.

- The use of axioms over the addition in the positive integers :

- +COM. $a+b=b+a$. Commutativity of +.
- +ASS. $(a+b)+c=a+(b+c)$. Associativity of +.
- +0. $(a+0)=a$. 0 neutral for +.
- +<R. $(b \neq 0) \leftrightarrow a < a+b$. Relation between + and < in the positive integers.
- +Def : other theorems of the addition over positive integer. ex :

$$a+b = 1 \rightarrow (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1).$$

- Axioms over < :

- <IRR. $\neg(a < a)$. Irreflexibility of <.
- <TRANS. $(a < b) \wedge (b < c) \rightarrow (a < c)$. Transitivity of <.
- <CONN. $(a < b) \vee (a = b) \vee (b < a)$. Connection of <.
- <MIN. $(a = 0) \vee (0 < a)$. 0 is minimum for <.

- Definition of $\leq, \neq, >, \geq$ with $\neg, =, <$:

- RDEF1. $(a \leq b) \leftrightarrow (a = b) \vee (a < b)$.
- RDEF2. $(a \neq b) \leftrightarrow \neg(a = b)$.
- RDEF3. $(a > b) \leftrightarrow \neg(a < b) \wedge \neg(a = b)$.
- RDEF4. $(a \geq b) \leftrightarrow \neg(a < b)$.

10.2.3 Part C : the net part

This part translates the net information of the particular specification for which we want to make a proof. It consists in two types of axioms :

- If the net is marked : axioms over the initial marking M_0 :

$$\text{INM. } \forall p \in P: \text{Init} \rightarrow m(p) = M_0(p)$$

- Translation of the net structure by the definition of the functions *pre* and *post* :

$$\text{NS. } \forall p \in P, \forall t \in T: \text{pre}(p, t) = \text{Pre}(p, t) \wedge \text{post}(p, t) = \text{Post}(p, t).$$

Those axioms are given for a Petri net $N = \langle P, T, \text{Pre}, \text{Post} \rangle$.

10.2.4 Two proof examples in PNTL

As an illustration of the use of the proof system, let us consider the following PNTL specification :

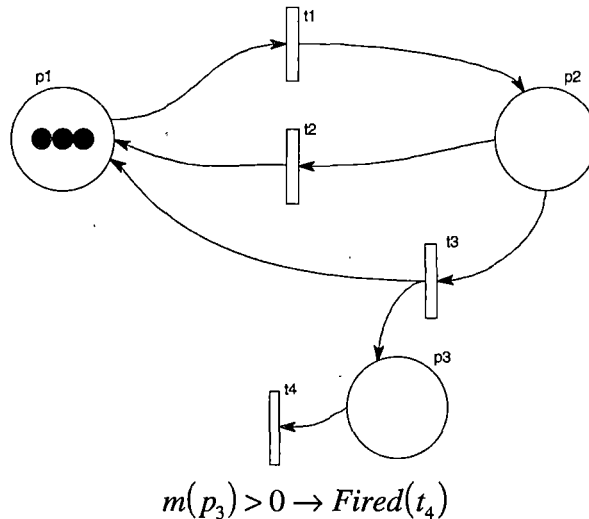


Figure 10.1 (A PNTL specification)

Before proving two theorems of this specification, we will translate the net information (part C of the proof system) of this PNTL specification :

(i) Initial marking :

$$\text{Init} \rightarrow (m(p_1) = 3 \wedge m(p_2) = 0 \wedge m(p_3) = 0)$$

(ii) Pre and Post functions :

$$\begin{aligned} \text{pre}(p_1, t_1) &= 1 \wedge \text{pre}(p_1, t_2) = 0 \wedge \text{pre}(p_1, t_3) = 0 \wedge \text{pre}(p_1, t_4) = 0 \\ \text{pre}(p_2, t_1) &= 0 \wedge \text{pre}(p_2, t_2) = 1 \wedge \text{pre}(p_2, t_3) = 1 \wedge \text{pre}(p_2, t_4) = 0 \\ \text{pre}(p_3, t_1) &= 0 \wedge \text{pre}(p_3, t_2) = 0 \wedge \text{pre}(p_3, t_3) = 0 \wedge \text{pre}(p_3, t_4) = 1 \\ \text{post}(p_1, t_1) &= 0 \wedge \text{post}(p_1, t_2) = 1 \wedge \text{post}(p_1, t_3) = 1 \wedge \text{post}(p_1, t_4) = 0 \\ \text{post}(p_2, t_1) &= 1 \wedge \text{post}(p_2, t_2) = 0 \wedge \text{post}(p_2, t_3) = 0 \wedge \text{post}(p_2, t_4) = 0 \\ \text{post}(p_3, t_1) &= 0 \wedge \text{post}(p_3, t_2) = 0 \wedge \text{post}(p_3, t_3) = 1 \wedge \text{post}(p_3, t_4) = 0 \end{aligned}$$

Theorem 10.1 : $G(m(p_1) + m(p_2) = 3)$

Proof 10.1 (Proof of the theorem 10.1)

(a1) $\text{Init} \rightarrow (m(p_1) = 3 \wedge m(p_2) = 0)$ (INM)

(a2) $\text{Init} \rightarrow (m(p_1) + m(p_2) = 3)$ (+def)

(a3)

- $$m(p_1) + m(p_2) = 3 \leftrightarrow (m(p_1) = 0 \wedge m(p_2) = 3) \\ \vee (m(p_1) = 1 \wedge m(p_2) = 2) \\ \vee (m(p_1) = 2 \wedge m(p_2) = 1) \\ \vee (m(p_1) = 3 \wedge m(p_2) = 0) \quad (+\text{def})$$
- (a4) $Fired(t_1) \vee Fired(t_2) \vee Fired(t_3) \vee Fired(t_4) \vee Fired(\text{null})$ (FR3)
- (a5) $(m(p_1) = 3 \wedge m(p_2) = 0) \\ \rightarrow Enabled(t_1) \wedge \neg Enabled(t_2) \wedge \neg Enabled(t_3)$ (FR5)
- (a6) $(m(p_1) = 3 \wedge m(p_2) = 0) \wedge Fired(t_1) \rightarrow X(m(p_1) = 2 \wedge m(p_2) = 1)$ (FR6)
- (a7) $(m(p_1) = 3 \wedge m(p_2) = 0) \wedge Fired(t_4) \rightarrow X(m(p_1) = 3 \wedge m(p_2) = 0)$ (FR6)
- (a8) $(m(p_1) = 3 \wedge m(p_2) = 0) \rightarrow X(m(p_1) + m(p_2) = 3)$ (a3,a5,a6,a7)
- (a9) $(m(p_1) = 2 \wedge m(p_2) = 1) \\ \rightarrow Enabled(t_1) \wedge Enabled(t_2) \wedge Enabled(t_3)$ (FR5,NS)
- (a10) $(m(p_1) = 2 \wedge m(p_2) = 1) \wedge Fired(t_1) \rightarrow X(m(p_1) = 1 \wedge m(p_2) = 2)$ (FR6,NS)
- (a11) $(m(p_1) = 2 \wedge m(p_2) = 1) \wedge Fired(t_2) \rightarrow X(m(p_1) = 3 \wedge m(p_2) = 0)$ (FR6,NS)
- (a12) $(m(p_1) = 2 \wedge m(p_2) = 1) \wedge Fired(t_3) \rightarrow X(m(p_1) = 3 \wedge m(p_2) = 0)$ (FR6,NS)
- (a13) $(m(p_1) = 2 \wedge m(p_2) = 1) \wedge Fired(t_4) \rightarrow X(m(p_1) = 2 \wedge m(p_2) = 1)$ (FR6,NS)
- (a14) $(m(p_1) = 2 \wedge m(p_2) = 1) \rightarrow X(m(p_1) + m(p_2) = 3)$ (a3,a4,a9-13,ProL)
- (a15) $(m(p_1) = 1 \wedge m(p_2) = 2) \\ \rightarrow Enabled(t_1) \wedge Enabled(t_2) \wedge Enabled(t_3)$ (FR5,NS)
- (a16) $(m(p_1) = 1 \wedge m(p_2) = 2) \wedge Fired(t_1) \rightarrow X(m(p_1) = 0 \wedge m(p_2) = 3)$ (FR6,NS)
- (a17) $(m(p_1) = 1 \wedge m(p_2) = 2) \wedge Fired(t_2) \rightarrow X(m(p_1) = 2 \wedge m(p_2) = 1)$ (FR6,NS)
- (a18) $(m(p_1) = 1 \wedge m(p_2) = 2) \wedge Fired(t_3) \rightarrow X(m(p_1) = 2 \wedge m(p_2) = 1)$ (FR6,NS)
- (a19) $(m(p_1) = 1 \wedge m(p_2) = 2) \wedge Fired(t_4) \rightarrow X(m(p_1) = 1 \wedge m(p_2) = 2)$ (FR6,NS)
- (a20) $(m(p_1) = 1 \wedge m(p_2) = 2) \rightarrow X(m(p_1) + m(p_2) = 3)$ (a3,a4,a15-19,ProL)
- (a21) $(m(p_1) = 0 \wedge m(p_2) = 3) \\ \rightarrow \neg Enabled(t_1) \wedge Enabled(t_2) \wedge Enabled(t_3)$ (FR5,NS)
- (a22) $(m(p_1) = 0 \wedge m(p_2) = 3) \wedge Fired(t_2) \rightarrow X(m(p_1) = 1 \wedge m(p_2) = 2)$ (FR6,NS)
- (a23) $(m(p_1) = 0 \wedge m(p_2) = 3) \wedge Fired(t_3) \rightarrow X(m(p_1) = 1 \wedge m(p_2) = 2)$ (FR6,NS)
- (a24) $(m(p_1) = 0 \wedge m(p_2) = 3) \wedge Fired(t_4) \rightarrow X(m(p_1) = 0 \wedge m(p_2) = 3)$ (FR6,NS)
- (a25) $(m(p_1) = 0 \wedge m(p_2) = 3) \rightarrow X(m(p_1) + m(p_2) = 3)$ (a3,a4,a21-24,ProL)
- (a26) $m(p_1) + m(p_2) = 3 \rightarrow X(m(p_1) + m(p_2) = 3)$ (a3,a8,a14,a20,a25,ProL)
- (a27) $G(m(p_1) + m(p_2) = 3)$ (I-INV,a2,a26)

Theorem 10.2 : (b0) $G(\text{Fired}(t_3) \rightarrow X(\text{Fired}(t_4)))$

Proof 10.2 (Proof of the theorem 10.2) .

(b1) $m(p_3) > 0 \rightarrow \text{Fired}(t_4)$	(Spec)
(b2) $\text{Fired}(t_4) \rightarrow \neg \text{Fired}(t_3)$	(FR2)
(b3) $m(p_3) > 0 \rightarrow \neg \text{Fired}(t_3)$	(b1,b2,ProL)
(b4) $\text{Fired}(t_3) \rightarrow m(p_3) = 0$	(b3,ProL,RDEF)
(b5) $\text{Fired}(t_3) \wedge m(p_3) = 0 \rightarrow X(m(p_3) = 1)$	(FR6,NS)
(b6) $\text{Fired}(t_3) \rightarrow X(m(p_3) = 1)$	(b4,b5,ProL)
(b7) $\text{Fired}(t_3) \rightarrow X(m(p_3) > 0)$	(b6,+<R)
(b8) $\text{Fired}(t_3) \rightarrow X(\text{Fired}(t_4))$	(b7,Spec)
(b9) $G(\text{Fired}(t_3) \rightarrow X(\text{Fired}(t_4)))$	(b8,NR)

10.3 A proof system for PNRTL

In this section, we present a logical proof system for PNRTL. First, we formally characterize the distance function d which gives the distance in time between two states. Then we present an axiomatization of the definitions 7.21, 7.23 and 7.24 and axioms over the real-time temporal operators of PNRTL. Finally we illustrate the use of the logical system by proving a theorem containing real-time aspects.

10.3.1 Formal characterization of the distance function d

To handle quantitative temporal properties, we have added a distance function between states. This function d has three arguments :

- a couple $(\delta 1, \delta 2)$ which indicates which timestamps must be considered (InTime/OutTime)
- a first and a second state

and returns a positive real number : the distance in real-time which separates the two states.

The function d is defined as follows :

$$d((\delta 1, \delta 2), (S, i), (S, j)) =$$

- if $(i \leq j)$: $mt(\delta 2)((S, j)) - mt(\delta 1)((S, i))$ ¹
- if $(i > j)$: $mt(\delta 1)((S, i)) - mt(\delta 2)((S, j))$

The function d satisfies the two following properties :

¹ $mt(\delta)$ returns the function InTime if $\delta=i$, the function OutTime if $\delta=o$, see notation 7.1.

$$(d1) \ d((\delta 1, \delta 2), (S, i), (S, j)) = d((\delta 2, \delta 1), (S, j), (S, i))$$

$$(d2) \ \text{if } (S, i) < (S, j) < (S, k) \ (i < j < k) \\ d((\delta 1, \delta 3), (S, i), (S, k)) = d((\delta 1, \delta 2), (S, i), (S, j)) + d((\delta 2, \delta 3), (S, j), (S, k))$$

These two properties of the function d are used in the following subsections to justify axioms over the real-time temporal operators.

10.3.2 Axiomatization of the PNRTL nets

In this subsection, we define axiom schemata for the translation of a PNRTL net into logical formulae. These axioms schemata translate the definitions 7.21, 7.23 and 7.24.

(1) *Axiomatization of an enabled transition t for a valuation α :*

$$(FR1). \quad \begin{aligned} Enabled(t, \alpha) \leftrightarrow & (S : \alpha) \wedge (P_1(\bar{v}_1) : \alpha) \wedge \dots \wedge (P_{n1}(\bar{v}_{n1}) : \alpha) \\ & \wedge \neg(Q_1(\bar{w}_1) : \alpha) \wedge \dots \wedge \neg(Q_{n2}(\bar{w}_{n2}) : \alpha) \\ & \wedge (\bar{x}_1 : \alpha) \neq (\bar{y}_1 : \alpha) \wedge \dots \wedge (\bar{x}_{n3} : \alpha) \neq (\bar{y}_{n3} : \alpha) \end{aligned}$$

- S is the selector of the transition t .
- For each tuple e that annotates an input arc of type i or type p of transition t , there exists $f : 1 \leq f \leq n1$, $e = \bar{v}_f$ and P_f is the dynamic predicate of the input place of the arc annotated by e .
- For each tuple e that annotates an input arc of type \bar{i} or type \bar{p} of transition t , there exists $f : 1 \leq f \leq n2$, $e = \bar{w}_f$ and Q_f is the dynamic predicate of the input place of the arc annotated by e .
- For each tuples e_1, e_2 where e_1 annotates an arc of type o of t and e_2 an arc of type \bar{o} of t and the output place of the two arcs is the same place, then there exists $f : 1 \leq f \leq n3$, $\bar{x}_f = e_1$ and $\bar{y}_f = e_2$.

(2) *Effects of firing an instantaneous transition t with a valuation α in time T :*

$$\begin{aligned}
\text{BeginEnd}(t_n, \alpha) \rightarrow X(\text{InTime} = T & \\
& \wedge (P_1(\bar{u}_1): \alpha) \wedge \dots \wedge (P_{n1}(\bar{u}_{n1}): \alpha) \\
\text{(FR2)} \quad & \wedge \neg(Q_1(\bar{v}_1): \alpha) \wedge \dots \wedge \neg(Q_{n2}(\bar{v}_{n2}): \alpha) \\
& \wedge (R_1(\bar{w}_1): \alpha) \wedge \dots \wedge (R_{n3}(\bar{w}_{n3}): \alpha) \\
& \wedge \neg(S_1(\bar{x}_1): \alpha) \wedge \dots \wedge \neg(S_{n4}(\bar{x}_{n4}): \alpha))
\end{aligned}$$

- For each tuple e that annotates a *type* o arc of t , there exists $f: 1 \leq f \leq n1$, $e = \bar{u}_f$ and P_f is the dynamic predicate of the output place of the arc annotated by e .
- For each tuple e that annotates a *type* \bar{o} arc of t , there exists $f: 1 \leq f \leq n2$, $e = \bar{v}_f$ and Q_f is the dynamic predicate of the output place of the arc annotated by e .
- For all e_1 :

$$e_1 \in Tu(a_1) \wedge a_1 \in IA(t) \wedge Type(a_1) = \bar{i} \wedge IP(a_1) = p$$

$$\wedge \neg \exists e_2 (e_2 \in Tu(a_2) \wedge a_2 \in OA(t) \wedge Type(a_2) = \bar{o} \wedge OP(a_2) = p \wedge (e_1: \alpha = e_2: \alpha))$$
 there exists $f: 1 \leq f \leq n3$, $R_f = Pr(p)$ and $\bar{x}_f = e_1$.
- For all e_1 :

$$e_1 \in Tu(a_1) \wedge a_1 \in IA(t) \wedge Type(a_1) = i \wedge IP(a_1) = p$$

$$\wedge \neg \exists e_2 (e_2 \in Tu(a_2) \wedge a_2 \in OA(t) \wedge Type(a_2) = o \wedge OP(a_2) = p \wedge (e_1: \alpha = e_2: \alpha))$$
 there exists $f: 1 \leq f \leq n4$, $S_f = Pr(p)$ and $\bar{y}_f = e_1$.

(3) *Effect of firing an non-instantaneous transition t with a valuation α in time T with a duration D :*

$$\begin{aligned}
\text{Begin}(t_n, \alpha) \rightarrow X(\text{InTime} = T \wedge \neg(P_1(\bar{u}_1): \alpha) \wedge \dots \wedge \neg(P_{n1}(\bar{u}_{n1}): \alpha) \\
\text{(FR3)} \quad \wedge (Q_1(\bar{v}_1): \alpha) \wedge \dots \wedge (Q_{n2}(\bar{v}_{n2}): \alpha) \wedge \text{InProgress}(t_n, \alpha))
\end{aligned}$$

- For each tuple e that annotates a *type* i arc of transition t , there exists $f: 1 \leq f \leq n1$, $e = \bar{u}_f$ and P_f is the dynamic predicate of the input place of the arc annotated by e .
- For each tuple e that annotates a *type* \bar{i} arc of transition t , there exists $f: 1 \leq f \leq n2$, $e = \bar{v}_f$ and Q_f is the dynamic predicate of the input place of the arc annotated by e .
- $\text{InProgress}(t_n, \alpha)$, in the reached state, the transition occurrence t_n is in progress.

$$\begin{aligned}
\text{End}(t_n, \alpha) \rightarrow X(\text{InTime} = T + D \wedge (R_1(\bar{x}_1): \alpha) \wedge \dots \wedge (R_{n3}(\bar{x}_{n3}): \alpha) \\
\text{(FR4)} \quad \wedge \neg(S_1(\bar{y}_1): \alpha) \wedge \dots \wedge \neg(S_{n4}(\bar{y}_{n4}): \alpha) \wedge \neg \text{InProgress}(t_n, \alpha))
\end{aligned}$$

- For each tuple e that annotates a *type* o arc of transition t , there exists $f: 1 \leq f \leq n3$, $e = \bar{x}_f$ and R_f is the dynamic predicate of the output place of the arc annotated by e .

- For each tuple e that annotates a *type* \bar{o} arc of transition t , there exists $f: 1 \leq f \leq nA$, $e = \bar{y}_f$ and S_f is the dynamic predicate of the output place of the arc annotated by e .
- $\neg InProgress(t_n, \alpha)$, in the reached state, the transition occurrence t_n is no more in progress.

To complete the axiomatization of the firing rule, we must add that *a transition that fires must be enabled* :

$$(FR5) \quad BeginEnd(t, \alpha) \rightarrow Enabled(t, \alpha)$$

$$(FR6) \quad Begin(t, \beta) \rightarrow Enabled(t, \beta)$$

and axiomatize the *supplementary constraints* (SC) of definition 7.24 :

1. $Begin(t_n, \alpha) \rightarrow X (F (End(t_n, \alpha)))$
2. $End(t_n, \alpha) \rightarrow Y (P (Begin(t_n, \alpha)))$
3. $Begin(t_n, \alpha) \rightarrow X (G (\neg Begin(t_n, \beta)))$
4. $BeginEnd(t_n, \alpha) \rightarrow X (G (\neg BeginEnd(t_n, \beta)))$
5. $Begin(t_n, \alpha) \wedge (1 \leq n_2 \leq n) \rightarrow Y (P (Begin(t_{n_2}, \beta)))$
6. $BeginEnd(t_n, \alpha) \wedge (1 \leq n_2 \leq n) \rightarrow Y (P (BeginEnd(t_{n_2}, \beta)))$
7. $Init \rightarrow \forall t: \neg InProgress(t, \alpha)$
8. $p \wedge BeginEnd(null, -) \rightarrow X(p)$
9. $Enable(null, -) \rightarrow \forall t, \forall \beta: \neg Enabled(t, \beta)$
10. $InTime \leq OutTime$
11. $OutTime = x \rightarrow X(InTime = x)$
12. $Init \rightarrow InTime = 0$

10.3.3 Axioms for real-time temporal operators

In this subsection, we present axiom schemata for the real-time temporal operators of PNRTL. In the following \mathfrak{S} , (δ_1, δ_2) represents an interval, respectively a pair of bounds, as defined in chapter 7.

Definitions :

$$(RT1). \quad G_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\varphi) \leftrightarrow \neg F_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\neg\varphi)$$

$$(RT2). \quad H_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\varphi) \leftrightarrow \neg P_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\neg\varphi)$$

$$(RT3). \quad F_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\varphi) \rightarrow F(\varphi)$$

$$(RT4). \quad P_{\mathfrak{S}}^{(\delta_1, \delta_2)}(\varphi) \rightarrow P(\varphi)$$

$$(RT5). \quad (\varphi)U_{=t}^{(\delta_1, \delta_2)}(\vartheta) \leftrightarrow G_{<t}^{(\delta_1, \delta_2)}(\varphi) \wedge F_{=t}^{(\delta_1, \delta_2)}(\vartheta)$$

$$(RT6). \quad (\varphi)S_{=t}^{(\delta_1, \delta_2)}(\vartheta) \leftrightarrow H_{<t}^{(\delta_1, \delta_2)}(\varphi) \wedge P_{=t}^{(\delta_1, \delta_2)}(\vartheta)$$

Distribution schema :

$$(RT7). G_3^{(\delta_1, \delta_2)}(p \rightarrow q) \rightarrow (G_3^{(\delta_1, \delta_2)}(p) \rightarrow G_3^{(\delta_1, \delta_2)}(q))$$

$$(RT8). H_3^{(\delta_1, \delta_2)}(p \rightarrow q) \rightarrow (H_3^{(\delta_1, \delta_2)}(p) \rightarrow H_3^{(\delta_1, \delta_2)}(q))$$

Characterizations of the properties of the metric point structure :

$$a) d((\delta 1, \delta 2), (S, i), (S, j)) = d((\delta 2, \delta 1), (S, j), (S, i))$$

$$(RT9). (p \wedge F_3^{(\delta_1, \delta_2)}(q)) \rightarrow F_3^{(\delta_1, \delta_2)}(q \wedge P_3^{(\delta_2, \delta_1)}(p))$$

$$(RT10). (p \wedge P_3^{(\delta_1, \delta_2)}(q)) \rightarrow P_3^{(\delta_1, \delta_2)}(q \wedge F_3^{(\delta_2, \delta_1)}(p))$$

b) if $(S, i) < (S, j) < (S, k)$ ($i < j < k$)

$$d((\delta 1, \delta 3), (S, i), (S, k)) = d((\delta 1, \delta 2), (S, i), (S, j)) + d((\delta 2, \delta 3), (S, j), (S, k))$$

Let us consider the following table :

+	$y \leq b$	$y < b$	$y = b$	$y > b$	$y \geq b$
$x \leq a$	$x + y \leq a + b$	$x + y < a + b$	$b \leq x + y \leq a + b$	$x + y > b$	$x + y \geq b$
$x < a$	$x + y < a + b$	$x + y < a + b$	$b \leq x + y < a + b$	$x + y > b$	$x + y \geq b$
$x = a$	$a \leq x + y \leq a + b$	$a \leq x + y < a + b$	$x + y = a + b$	$x + y > a + b$	$x + y \geq a + b$
$x > a$	$x + y > a$	$x + y > a$	$x + y > a + b$	$x + y > a + b$	$x + y > a + b$
$x \geq a$	$x + y \geq a$	$x + y \geq a$	$x + y \geq a + b$	$x + y > a + b$	$x + y \geq a + b$

Table 10.1 (Addition and order relations in the positive real numbers)

From the table 10.1, we can deduce, for example the following axioms :

$$F_{<a}^{(\delta_1, \delta_2)}(F_{\geq b}^{(\delta_2, \delta_3)}(p)) \rightarrow F_{\geq b}^{(\delta_1, \delta_3)}(p)$$

$$P_{>a}^{(\delta_1, \delta_2)}(P_{>b}^{(\delta_2, \delta_3)}(p)) \rightarrow P_{>a+b}^{(\delta_1, \delta_3)}(p)$$

References to this axiom schema is noted RT11.

Axioms schemas relating to arithmetic over the metric operators :

$$(RT12). F_{=t_1}^{(\delta_1, \delta_2)}(P_{=t_1+t_2}^{(\delta_2, \delta_3)}(p)) \rightarrow P_{=t_2}^{(\delta_1, \delta_3)}(p)$$

$$(RT13). P_{=t_1}^{(\delta_1, \delta_2)}(F_{=t_1+t_2}^{(\delta_2, \delta_3)}(p)) \rightarrow F_{=t_2}^{(\delta_1, \delta_3)}(p)$$

$$(RT14). F_{=t_1+t_2}^{(\delta_1, \delta_2)}(P_{=t_1}^{(\delta_2, \delta_3)}(p)) \rightarrow F_{=t_2}^{(\delta_1, \delta_3)}(p)$$

$$(RT15). P_{=t_1+t_2}^{(\delta_1, \delta_2)}(F_{=t_1}^{(\delta_2, \delta_3)}(p)) \rightarrow P_{=t_2}^{(\delta_1, \delta_3)}(p)$$

Axioms over X and F :

$$(RT16). X(F_3^{(i, \delta_2)}(p)) \rightarrow F_3^{(o, \delta_2)}(p)$$

$$(RT17). F_3^{(\delta_1, o)}(X(p)) \rightarrow F_3^{(\delta_1, i)}(p)$$

These two axiom shemata are justified by : $OutTime=x \rightarrow X(InTime=x)$ (SC11).

10.3.4 A proof example in PNRTL

We present here the proof of a theorem of the example 7.2 of chapter 7 (page 7.26). Let us first translate the net information of example 7.2 into logical formulae :

$$(i1) Enabled(Consume, (md, t, n)) \leftrightarrow Piece_To_Consume(md, t, n) \quad (FR1)$$

$$(i2) Enabled(Demand, (md, t, n)) \leftrightarrow Free_To_Demand(md) \quad (FR1)$$

$$(i3) Enabled(Produce, (m, md, t, n)) \leftrightarrow CanProduce(m, t) \wedge Piece_Asked(md, t, n) \\ \wedge Free_To_Produce(m) \quad (FR1)$$

$$(i4) Begin(Consume, (md, t, n)) \rightarrow X(\neg Piece_To_Consume(md, t, n)) \quad (FR3)$$

$$(i5) End(Consume, (md, t, n)) \rightarrow X(Produced_Piece(Manuf(t), n) \\ \wedge Free_To_Demand(md)) \quad (FR4)$$

$$(i6) BeginEnd(Demand, (md, t, n)) \rightarrow X(\neg Free_To_Demand(md) \\ \wedge Piece_Asked(md, t, n)) \quad (FR2)$$

$$(i7) Begin(Produce, (m, md, t, n)) \rightarrow X(\neg Piece_Asked(md, t, n) \\ \wedge \neg Free_to_Produce(m)) \quad (FR3)$$

$$(i8) End(Produce, (m, md, t, n)) \rightarrow X(Free_To_Produce(m) \\ \wedge Piece_To_Consume(md, t, n)) \quad (FR4)$$

References to axioms of the specification of example 7.2 will be noted **a1..a10**.

Theorem 10.3 : $BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 7sec}^{(o, i)}(Produced_Piece(Manuf(t), n))$ (c0)

$$(c1) BeginEnd(Demand, (md, t, n)) \rightarrow X(Piece_Asked(md, t, n)) \quad (i6)$$

$$(c2) BeginEnd(Demand, (md, t, n)) \rightarrow X(F_{\leq 5sec}^{(i, i)}(Piece_To_Consume(md, t, n))) \quad (c1, a9)$$

$$(c3) BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 5sec}^{(o, i)}((Piece_To_Consume(md, t, n))) \quad (c2, RT16)$$

$$(c4) BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 5sec}^{(o, i)}(F_{\leq 1sec}^{(i, o)}(Begin(Consume, (md, t, n)))) \quad (c3, a5)$$

$$(c5) BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 6sec}^{(o, o)}(Begin(Consume, (md, t, n))) \quad (c4, RT11)$$

$$(c6) BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 6sec}^{(o, o)}(F_{=1sec}^{(o, o)}(End(Consume, (md, t, n)))) \quad (c5, a6)$$

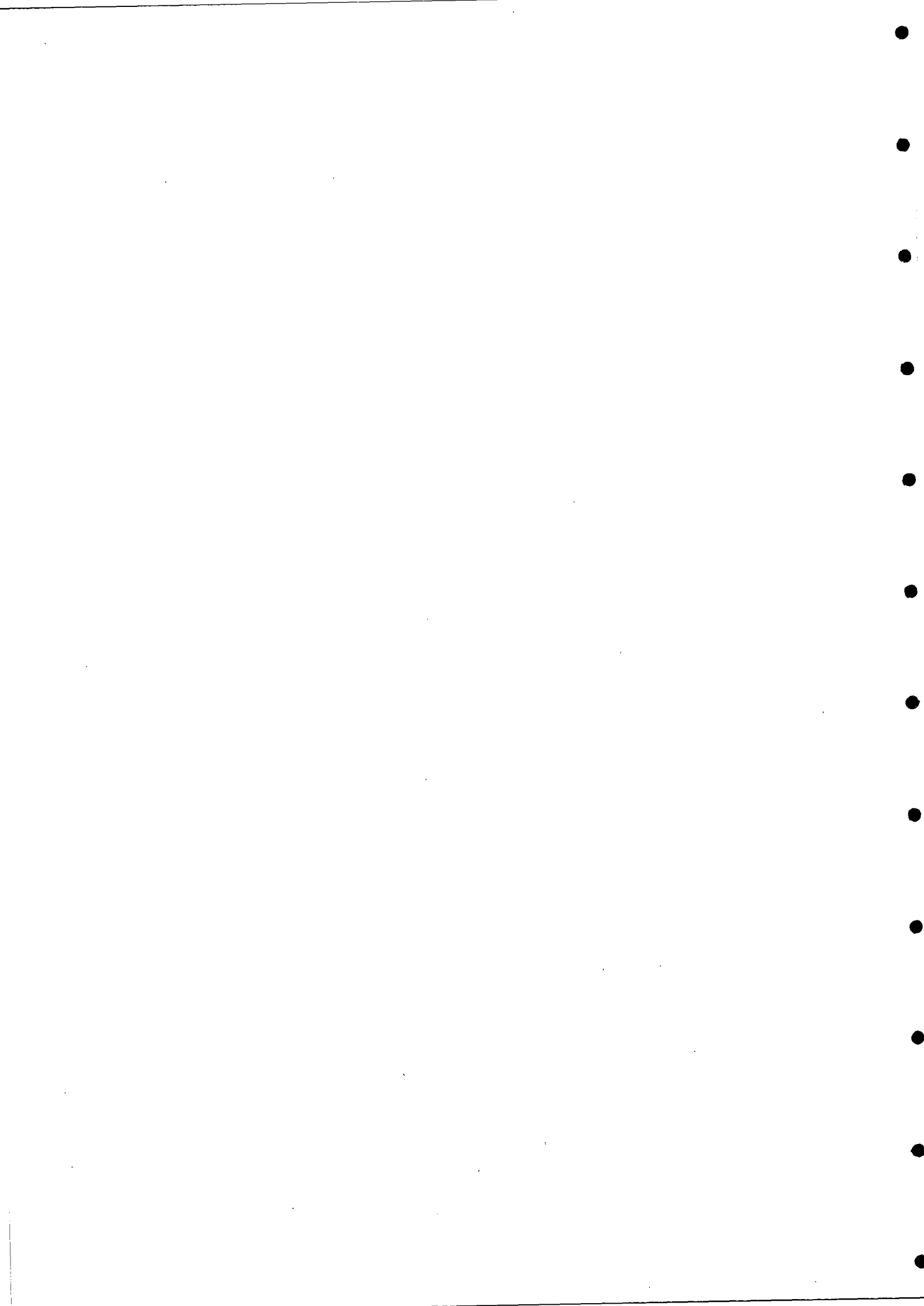
$$(c7) BeginEnd(Demand, (md, t, n)) \rightarrow F_{\leq 7sec}^{(o, o)}(End(Consume, (md, t, n))) \quad (c6, RT11)$$

$$(c8) \text{BeginEnd}(\text{Demand},(md,t,n)) \rightarrow F_{\leq 1 \text{ sec}}^{(o,o)}(X(\text{Produced_Piece}(\text{Manuf}(t),n))) \quad (c7,i5)$$

$$(c9) \text{BeginEnd}(\text{Demand},(md,t,n)) \rightarrow F_{\leq 1 \text{ sec}}^{(o,i)}(\text{Produced_Piece}(\text{Manuf}(t),n))) \quad (c8,RT17)$$

10.4 Conclusion

We have presented a way to translate the net information of a PNTL and a PNRTL specification into logical formulae. We have also defined a logical proof system for the two languages. Proof examples even if obvious, have demonstrated the power of the proof systems. Usual syntactical proof can be conducted in a natural way.



Conclusion

Petri net and temporal logic

The combination of two different styles - declarative and operational - is to our knowledge an unusual and rather original approach for modeling distributed systems. The benefits of such combination are twofold. In relation to Petri nets, it becomes now possible to express various kinds of constraints (e.g. deontic aspects, real-time features); also specifications are easier to read. In relation to temporal logic, it allows us to include constraints that are by nature operational (e.g. the presence of a certain resource). Another original point is the introduction of a preference order on the set of executions. The case study has illustrated the potentialities of this combination.

The PNRTL language

The language we have proposed in this work provides several advantages. First, the integration of logical formulae allows to model declarative constraints. Next, we have developed two proof techniques that can support the modeling of distributed systems with PNTL and PNRTL. The first one maps the Petri net and the constraints that accompany it into an automaton whose language can then be calculated and for which the testing of properties is possible. The second one translates the information held in a Petri net into an axiomatic

systems. It is important to see that they both rest on well-known formalisms and are therefore amenable to automation.

From specification to modeling

The addition of temporal logic formulae to colored Petri nets probably makes proof possibilities more arduous, but it certainly provides the analyst with a more flexible method in his/her modeling task. Indeed, if we start from the point of view that temporal logic is aimed at specifying systems while Petri nets are rather operational and are therefore modeling tools, it is important to see that the combination of both approaches eases the transition from specification to modeling and reduces the gap between those two phases in the development process.

When dealing with any constraint, the analyst often has the choice to "translate" it either into additional places and transitions, or into temporal formulae attached to the net. He/she should first opt for the second possibility, in order to keep the Petri nets as simple (i.e. easy to read) as possible. Once the completeness of the specification has been shown, one can then progressively make it more operational by translating, one by one, the logic formulae into semantically equivalent sub-nets. At last, one would obtain a pure Petri net model which could be tested thanks to some of the numerous simulation tools existing for Petri nets.

Perspectives

It would be particularly interesting to (semi-)automatize such transformations - or at least some of them - to accelerate the development process and thereby to decrease its cost. Furthermore, such (semi)automation would decrease the risk of errors when going from specification to modeling. One can even think of a CASE tool aimed at helping the analyst in making the specification operational, for instance by suggesting him/her to replace a logical formula by an addition of some places and/or transitions. We could also think of improving the semantics of our language in order to avoid the - rather annoying - exponents decorating the real-time operators.

Bibliographical References

[AALST92]

Van der Aalst W.M.P., "*Timed coloured Petri nets and their application to logistic*", ch1-2, PhD thesis, Eindhoven University of Technology, 1992.

[AHO&ULLMAN93]

Aho Alfred and Ullman Jeffrey, "*Concepts fondamentaux de l'informatique*", Dunod, Paris, 1993.

[BAC94]

Haféda Bachatène, "*Une approche Modulaire Intégrant les Réseaux de Petri Colorés pour la Spécification de Systèmes Distribués*", Thèse de doctorat, Université de Paris VI, Septembre 1994.

[BLWW94]

Roger W.H. Bons, Ronald M. Lee, René W. Wagenaar and Clive D. Wrigley, "*Computer Aided Design of Inter-organizational Trade Scenarios: A CASE for Open-edi*", Report No. WP 94.03.01, Euridis, Erasmus University Rotterdam, The Netherlands, 1994.

[BLWW95]

Bons, Lee, Wagenaar and Wrigley, "*Modelling inter-organizational trade procedures using documentary Petri nets*", Proceedings of the Hawaii international conference on system Science, Hawaii, 1995.

[BMR92]

A. Borgida, J. Mylopoulos, and R. Reiter, ... *and nothing else changes : the frame problem in procedure specification*. Technical Report DCS-TR-281, Dept of Computer Science, Rutgers University, 1992.

[BRAMS83a]

G.W. Brams, "*Réseaux de Petri : théorie et pratique, tome 1 : théorie et analyse*", MASSON, 1993.

[BRAMS83b]

G.W. Brams, "*Réseaux de Petri : théorie et pratique, tome 2 : modélisation et application*", MASSON, 1993.

[CHELLAS80]

B.F. Chellas, "*Modal logic : An Introduction*". Cambridge University Press, 1980.

[DAVIS&WEYUKER83]

Davis Martin D. and Weyuker Elaine J., "*Computability, complexity and languages*", chapter 6, *Fundamental of Theoretical Computer Science*, New-York, 1983.

[DDDP94a]

Dubois, Du Bois, Dubru and Petit, "*Agent-oriented requirements engineering : a case study using the ALBERT language*", Proceedings of the fourth international working conference on dynamic modelling and information systems - DYNMOD IV, Noordwijkerhoud, The Netherlands, 1994.

[DDDP94b]

Dubois, Du Bois, Dubru and Petit, "*The ALBERT course, vol. 1 : the language*", University of Namur (Belgium), 1994.

[DDP94c]

Eric Dubois, Philippe Du Bois and Michaël Petit. *Albert: an Agent-oriented Language for Building and Eliciting Requirements for real-Time systems*. In Proc. of the 27th Hawaii International Conference on System Sciences - HICSS-27, Maui (Hawaii), January 1994. IEEE.

[DDZ95]

Eric Dubois, Philippe Du Bois, Jean-Marc Zeippen, *A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems*, in Proceedings of RTS'95, Paris (France), January 11-13, 1995.

[DIG94]

Frank Dignum (Eindhoven University of Technology, Netherlands) and Hans Weigand (Tilburg University, Netherlands), *Communication and Deontic Logic*, Report Paper, 01-08-1994.

[DLT91]

Eeric Dubois, A. van Lamsweerde, A. Thayse, *Approche logique de l'intelligence artificielle 4 : De l'apprentissage artificiel aux frontières de l'IA*. Dunod Paris, 1991.

[DUBOIS91]

Eric Dubois, *Use of Deontic Logic in the Requirements Engineering of Composite Systems*, in "*Deontic Logic in Computer Science, Normative System Specification*", John Wiley & Sons Ltd, 1991.

[DUBOIS94]

Dubois Eric, "*ALBERT at the age of two*", University of Namur (Belgium), 1994.

[DUBOIS95a]

Philippe Du Bois, *Intuitive Definition of the Albert II language*, Research Paper RP-95-007, Computer Science Institute, Namur University, Belgium, 1995.

[DUBOIS95b]

Philippe Du Bois, *Semantic Definition of the Albert II Language*, Research Paper RP-95-008, Computer Science Institute, Namur University, Belgium, 1995.

[EILENBERG74]

Eilenberg Samuel, *"Languages and Machines"*, Volume A, Columbia University, New-York 1974.

[FICHEFET88]

Fichefet Jean, *"Introduction aux réseaux de Petri"*, Notes de cours, University of Namur (Belgium), 1988.

[GENRICH86]

Genrich H.J., *"Predicate transition nets"*, Lectures notes in Computer Science, vol 254, pp 207-247, 1986.

[GINZBURG68]

Ginzburg Abraham, *"Algebraic theory of automata"*, chapter 4, ACM Monograph Series, Academic Press, New-York, 1969.

[GONDRAN&MINOUX79]

Gondran M. and Minoux M. , *"Graphes et algorithmes"*, Eyrolles, Paris, 1979.

[HILPINEN71]

R. Hilpinen, *Deontic Logic : Introductory and Systematic Readings*, Reidel Publishing Compagny, 1971.

[JENSEN86]

Jensen K., *"Coloured Petri nets"*, Lectures notes in Computer Science, vol 254, pp 248-300, 1986.

[JENSEN90]

Jensen K., *"Coloured Petri nets : a high level language for system design and analysis"*, Lectures notes in Computer Science, vol 486, pp 342-416, 1990.

[KOYMANS89]

R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, Ph.D. Thesis, Eindhoven University of Technology, 1989.

[KOYMANS92]

R. Koymans, *Specifying Message Passing and Time-Critical Systems*, Springer Verlag Berlin, 1992.

[LEE91]

Lee R.M., *"CASE/EDI : EDI modelling - User documentation"*, Technical report, Euridis, University of Rotterdam, 1991.

[LEE92]

Lee R.M., "*Dynamic modelling of documentary procedures : a case for EDI*", Technical report, Euridis, University of Rotterdam, 1992.

[MACARTHUR76]

R. MacArthur, "*Tense Logic*", D. Reidel Publishing Compagny. 1976.

[MAND&GHEZ87]

Mandrioli Dino and Ghezzi Carlo, "*Theoretical foundations of computer science*", chapter 1, John Wiley&Sons, New-York, 1987.

[MANNA&PNUELI84]

Manna Z. and Pnueli A., "*Adequate proof principles for invariance and liveness properties of concurrent programs*", Science of Computer Programming, vol 4, pp 257-289, 1984.

[MEYER91]

John-Jules Ch. Meyer and Roel J. Wieriga, "*Deontic Logic in Computer Science, Normative System Specification, part I : Tutorial introduction*", John Wiley & Sons Ltd, 1991.

[MP81]

Zohar Manna and Amir Pnueli, "*Verification of concurrent programs: the temporal framework*", in "*The Correctness Problem in Computer Science*", edited by Robert S Boyer and J Strother Moore, Internaional Lecture Series in Computer Science, 1981.

[MP83a]

Zohar Manna and Amir Pnueli, "*How to Cook a Temporal Proof System four your Pet Language*", in "*Proceedings of the Tenth ACM symposium on the Principles of Programming Languages*", pp. 141-154, 1983.

[MP83b]

Z. Manna, A. Pnueli. "*Verification of Concurrent Programs : A Temporal Proof System*", pp. 163-255 in J. Bakker, J. Van Leeuwen (eds.) "*Foundations of Computer Science IV*", Mathematical center Tracts Vol 159, CWI, Amsterdam, 1983.

[MP92]

Zohar Manna and Amir Pnueli, "*The Temporal Logic of Reactive and Concurrent Systems, Specification*", Springer-Verlag, 1992.

[MURATA89]

Murata T., "*Petri nets : properties, analysis and applications*", Proceedings of the IEEE, vol 77, No 4, 1989.

[PETERSON81]

James L. Peterson, "*Petri net theory and the modeling of systems*", Prentice-Hall, 1981.

[PETRI62]

Petri C. A., "*Kommunikation mit automaten*", PhD dissertation, University of Bonn, West-Germany, 1962.

[PNU 77]

A. Pnueli. *The Temporal Logic of Programs*, in *Proceedings of the Eighteenth Symposium on the Foundations of Computer Science*, pp. 46-57, 1977.

[SALOMAA73]

Salomaa Arto, "*Formal languages*", ACM Monograph Series, Academic Press, New-York, 1973.

[SCS92]

Cristina Sernadas, José Félix Costa and Amilcar Sernadas, *Object Specification Logic*, Departamento de Matematica, Instituto Superior Tecnico, INESC Lisbon Portugal, June 1992.

[SGS92]

Cristina Sernadas, Paula Gouveia and Amilcar Sernadas, *OBLOG: Object-oriented, Logic-based Conceptual Modeling*, Departamento de Matematica, Instituto Superior Tecnico, INESC Lisbon Portugal, November 1992.

[SK93]

Amilcar Sernadas and Klemens Böhm, *Real-Time Object Specification Logic*, Departamento de Matematica, Instituto Superior Tecnico, INESC Lisbon Portugal, 1993.

[THAYSE89]

Thayse A.&co-auteurs, "*Approche logique de l'I.A. - 2. De la logique modale à la logique des bases de données*", chapitre 4, Dunod Informatique, 1989.

[TORRE94]

Leendert WN van der Torre, *Violated obligations in a defeasible deontic logic*, Report paper No RP 94.08.03, Euridis Erasmus University Rotterdam, 1994.

[TRAK&BARZ73]

Traktenbrot B.H. and Barzdin Y.M., "*Finite automata: behavior and synthesis*", Fundamental Studies in Computer Science, North Holland, 1973.

[TT94a]

Yao-Hua Tan and Leendert W.N. van der Torre, *DIODE: Deontic Logic Founded on Diagnosis From First Principles*. RP 94.08.04. Euridis, Erasmus University Rotterdam, 1994.

[TT94b]

Yao-Hua Tan and Leendert W.N. van der Torre. *Representing Deontic Reasoning in a Diagnostic Framework*. RP 94.08.05. Euridis, Erasmus University Rotterdam, 1994.

[TT94c]

Yao-Hua Tan and Leendert W.N. van der Torre. *Multi Preference Semantics for a Defeasible Deontic Logic*, in *Proceedings of the Seventh International Conference on Legal-Knowledge-Based Systems (JURIX'94)*. Amsterdam, the Netherlands, 1994.

[TT95]

Yao-Hua Tan and Leendert W.N. van der Torre, *Representing Legal Knowledge in a Diagnostic Framework*, Euridis Erasmus University Rotterdam, 1995.

[WRIGHT51]

von Wright, Georg Henrik, '*Deontic Logic*', *Mind* 60 (1951) 1-15. Reprinted in *Logical Studies* (by G. H. von Wright), Routledge and Kegan Paul, London, 1957, p. 58-74.

*