TECHNICAL REPORT 159

PETRI NET LANGUAGES

Michel Hack

March 1976

This report was prepared with the support of the National Science Foundation research grant DCR 74-21822.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

Petri Net Languages

Abstract

In a Labelled Petri Net we assign symbols from an alphabet to some or all the transitions of a Petri Net. To each firing sequence of such a Labelled Petri Net corresponds a string over the alphabet. We study the languages obtained in this way by all firing sequences of a Petri Net, or by all firing sequences which reach a given final marking. We consider the closure properties of these languages, their characterization, their relation to other language families, and the decidability of various problems concerning these languages. The last chapter relates Petri Nets to Counter Automata and Weak Counter Automata, introduces Inhibitor Nets and Priority Nets, and considers extensions and limitations of the Petri Net Languages.

Petri Net Languages

Abstract

In a labelled Petri Net we assign symbols from an alphabet to some or all the transitions of a Petri Net. To each firing sequence of such a labelled Petri Net corresponds a string over the alphabet. We study the languages obtained in this way by all firing sequences of a Petri Net, or by all firing sequences which reach a given final marking. We consider the closure properties of these languages, their characterisation, their relation to other language families, and the decidability of various problems concerning these languages. The last chapter relates Petri Nets to Counter Automata and Weak Counter Automata, introduces Inhibitor Nets and Priority Nets, and considers extensions and limitations of the Petri Net Languages.

Table of Contents

0. Introduction 1

1. Basic Definitions 3
 1.1 Generalized Petri Nets 3
 1.2 Restricted Petri Nets 6
 1.3 Labelled Petri Nets 6

2. Standard Forms for Labelled Petri Nets 9
 2.1 Some Language-preserving Transformations using λ-transitions 9
  2.1.1 The "Run" Place 9
  2.1.2 The "Start" Place 10
  2.1.3 The "Stop" Transition 10
  2.1.4 The "Clear" Place, for $\mathcal{L}^{\lambda}$ 11
  2.1.5 Elimination of Multiple Labels 12
 2.2 Some Language-preserving Transformations without λ-transitions 12
  2.2.1 The "Run" Place 12
  2.2.2 The "Start" Place 12
  2.2.3 The "Stop" Transitions 16
 2.3 The Standard Form Theorem 18
 2.4 The Relation of $\mathcal{L}_0$ to Peterson's Computation Sequence Sets 19

3. Simple Closure Properties 23
 3.1 Closure under Concurrent Composition 23
 3.2 Closure under Intersection 26
 3.3 Closure under Union 30
 3.4 Closure under Concatenation 32

4. Regular Languages and Related Closure Properties 35
 4.1 Petri Net Generation of Regular Languages 35
  4.1.1 $\mathcal{L}_0$ and λ-free Regular Languages 35
  4.1.2 Prefix Regular Languages 36
 4.2 Other Petri Net Codings for Finite State Machines 37
 4.3 Clean Standard Forms 37
 4.4 Closure under Clean Substitution 40
  4.4.1 Clean Substitution 40
  4.4.2 Regular Substitution 43
  4.4.3 Homomorphisms 44
 4.5 Inverse Homomorphism for $\mathcal{L}$ and $\mathcal{L}_0$ 44
 4.6 Closure under FST-Mappings 47
 4.7 Inverse Regular Substitution and $\mathcal{L}_0^{\lambda}, \mathcal{L}^{\lambda}$ 48
 4.8 Other Closures: Regular Control, Limited Erasing, Promptness 49

5. A Characterization of the Petri Net Language Families 53
 5.1 Free Petri Net Languages 53
 5.2 Simple Parenthesis Languages 54
 5.3 Closed Subnets 55
 5.4 The Restriction Operation on Languages 59
 5.5 A Characterization using Inverse Homomorphism 60
 5.6 Bounded Subnets 63
 5.7 A Simple Characterization of $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$ 64
 5.8 A Characterization using Finite Substitution 67

6. Decidable Properties of Petri Net Languages 73
6.1 Membership for $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$-languages 73
6.2 Emptiness and Finiteness for $\mathcal{L}$ and $\mathcal{L}^\lambda$-languages 74

7. Problems Equivalent to the Reachability Problem 77
7.1 Emptiness for $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$-languages 77
7.2 Membership for $\mathcal{L}_0^\lambda$-languages 78
7.3 Equivalence Problems for Free Petri Net Languages 79

8. Undecidable Equivalence Problems 85
8.1 The Polynomial Graph Inclusion Problem 85
8.2 Encodings of Polynomial Graphs as $\mathcal{L}$ or $\mathcal{L}_0$-languages 85
8.2.1 The Elementary Multiplier A(x) 86
8.2.2 The n-input Multiplier $B_n(x_1,\ldots,x_n)$ 88
8.2.3 Computing Polynomials 89
8.2.4 Completion of the Encoding Petri Net 91
8.3 The Undecidability Results 94

9. Limitations and Extensions of the Power of Petri Net Languages 99
9.1 Counter Automata 99
9.1.1 Program Machines 99
9.1.2 Language-generating Counter Automata 101
9.2 Inhibitor Arcs and Priority Firing Rules 107
9.2.1 Inhibitor Nets 107
9.2.2 Priority Nets 109
9.2.3 Converting Inhibitor Nets into Priority Nets 110
9.2.4 Converting Priority Nets into Inhibitor Nets 111
9.2.5 Characterization of the Languages generated by λ-free Inhibitor Nets and λ-free Priority Nets 113
9.3 Limits to the Power of Petri Net Languages 116
9.3.1 The Context-Free Language Q_0 116
9.3.2 Non-Closure under Kleene Star 117
9.3.3 The Languages of λ-free Inhibitor Nets or Priority Nets 117
9.4 Weak Counter Automata 118

References 127

List of Definitions

D 1.1:	Generalized Petri Net	3
D 1.2:	Labelled Petri Net	6
D 1.3:	Petri Net Languages	7
D 3.1:	Concurrent Composition of Languages	25
D 4.1:	Prefix Regular Languages	36
D 4.2:	Clean Standard Form; Clean Petri Net Language	38
D 4.3:	$\mathcal{F}$-Substitution	41
D 4.4:	k-Promptness	50
D 4.5:	k-Limited Erasing	50
D 5.1:	Free-Labelled Petri Nets and Free Petri Net Languages	53
D 5.2:	Simple Parenthesis Languages P and P_0	54
D 5.3:	Closed Subnets	55
D 5.4:	The Restriction Operation	60
D 9.1:	Counter Automaton (CA)	102
D 9.2:	Deterministic Counter Automaton	103
D 9.3:	Inhibitor Arc and Inhibitor Net	107
D 9.4:	Priority Net	109
D 9.5:	Weak Counter Automaton (WCA)	119
D 9.6:	Language of a WCA	119
D 9.7:	Prompt WCA	122

List of Theorems

T 2.1:	Inclusion Relations among Petri Net Languages	11
T 2.2:	Elimination of Multiple Labels	12
T 2.3:	Standard Form Theorem	19
T 3.1:	Closure under Concurrent Composition	26
T 3.2:	" " Intersection	30
T 3.3:	" " Union	30
T 3.4:	" " Concatenation	33
T 4.1:	Generation of Regular Languages as $\mathcal{L}_0$, $\mathcal{L}_0^\lambda$	36
T 4.2:	Generation of Prefix Regular Languages as $\mathcal{L}$, $\mathcal{L}^\lambda$	36
T 4.3:	Generation of Regular Languages by 2-place Petri Nets	37
T 4.4:	" " " " " 4-dimensional V.A.S.	37
T 4.5:	Clean is equivalent to Regular	40
T 4.6:	Closure under Clean Substitution	43
T 4.7:	" " Regular Substitution	43
T 4.8:	" " Homomorphism	44
T 4.9:	" " Inverse Homomorphism for $\mathcal{L}_0$,	46
corollary:	Closure under Finite Substitution	46
T 4.10:	Closure under FST-Mapping	47
corollary:	Closure under GSM-Mapping	47
T 4.11:	Inverse Regular Substitution and Inverse Homomorphism for $\mathcal{L}_0^\lambda$, $\mathcal{L}^\lambda$	49
T 4.12:	Closure under k-limited Erasing	51

T 5.1	Generating Petri Net Languages from Free Petri Net Languages	53
T 5.2	Closure of Free Petri Net Languages under Intersection and Inverse Homomorphism	54
L 5.1	Language of the Union of Closed Subnets	56
T 5.3	Characterization of the Free Petri Net Languages	62
T 5.4	$\mathcal{L}_0$, $\mathcal{L}$ are Context-Sensitive	63
T 5.5	Promptness-preserving Reduction of Labelled GPN to RPN	67
T 5.6	Characterization of $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$-languages via restriction	67
L 5.2	Language of Free RPN's	68
T 5.7	Characterization of $\mathcal{L}$ and $\mathcal{L}_0$ via Finite Substitution	72
T 5.8	Generation of Petri Net Languages by RPN's or FSM's with Buffers	72
L 6.1	Decidability of Boundedness and Coverability	73
T 6.1	Decidability of Membership for $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$	74
T 6.2	Decidability of Emptiness and Finiteness for $\mathcal{L}$, $\mathcal{L}^\lambda$	75
T 7.1	Emptiness for $\mathcal{L}_0$, $\mathcal{L}_0^\lambda$ equivalent to Reachability	77
T 7.2	Membership for $\mathcal{L}_0^\lambda$ equivalent to Reachability	78
T 7.3	Inclusion and Equivalence for $\mathcal{L}^f$ reducible to Reachability	80
T 7.4	Inclusion and Equivalence for $\mathcal{L}_0^f$ equivalent to Reachability	82
T 7.5	Complementation Closure of $\mathcal{L}^f$ is included in $\mathcal{L}_0$	82
L 8.1	Language of the n-input Multiplier B_n	88
T 8.1	Petri Net Languages can encode Polynomial Graphs	93
T 8.2	Inclusion and Equivalence for Petri Net Languages are Undecidable	94
T 8.3	Complementation Closure implies Undecidability of Reachability	95
T 8.4	Undecidability of Change in Language due to removing a Transition	95
T 8.5	" " " " " " " " Token or Place	96
T 8.6	" " whether Terminal Language same as Nonterminal	96
T 9.1	Generation of Recursively Enumerable Languages by Counter Automata	106
T 9.2	" " " " " " Inhibitor Nets	109
T 9.3	" " " " " " Priority Nets	110
T 9.4	Undecidability of Boundedness and Reachability for Inhibitor Nets and for Priority Nets	110
T 9.5	λ-free Transformation between Inhibitor Nets and Priority Nets	113
T 9.6	" " into Restricted Inhibitor or Priority Nets	114
T 9.7	Characterization of λ-free Inhibitor Net Languages	115
T 9.8	The Context-Free Language Q_0 cannot be generated by a Petri Net	117
T 9.9	Non-Closure under Kleene Star of $\mathcal{L}_0$, $\mathcal{L}_0^\lambda$	117
T 9.10	Palindromes cannot be recognized by λ-free Inhibitor Nets	118
T 9.11	The WCA-languages are the class $\mathcal{L}_0^\lambda$	120
T 9.12	The Prompt WCA-languages are the class $\mathcal{L}_0$, up to λ	122

Petri Net Languages.

0. Introduction.

In many applications of Petri nets it is the set of firing sequences generated by the net that is of prime importance. Usually, certain transitions represent actions, such as synchronization, operator execution, etc., and we would like to know which sequences of synchronizations or operations are compatible with the constraints imposed by the system being modelled and expressed by the structure of the Petri net.

For this purpose, it is useful to treat a Petri net like an automaton whose states are the markings of the net, and whose state-transition function expresses how and when transitions of the Petri net fire. This approach was already implicit in the work of Keller [12] and Baker [2].

We shall consider mainly Generalized Petri Nets. For other definitions, and for the relationship of Generalized Petri Nets to Ordinary Petri Nets and to Vector Addition Systems, see Hack [7].

The basic approach consists in assigning a name or label (i.e. a symbol drawn from some alphabet; not necessarily a different symbol for each transition) to some or all the transitions in the Petri net, and studying the strings obtained from firing sequences by replacing each transition occurrence by the corresponding transition label, or erasing it if it has no label (λ-transition).

Given a Petri net N with labelled transitions, we distinguish two basic kinds of Petri net languages:

a) The language $L(N,M,M')$ obtained from the set of firing sequences starting at marking M and leading to marking M'. The class of languages that can be obtained in this manner by some Generalized Petri Net (GPN) is designated $\mathcal{L}_0$ if every transition has a label, and $\mathcal{L}_0^\lambda$ if there are λ-transitions, i.e. unlabelled or "invisible" transitions in the net. If the initial and final markings are understood we often write $\mathcal{L}_0^\lambda(N)$ for this language, called the terminal language of the net.

b) The language $L(N,M,*)$ obtained from all firing sequences of N starting at the initial marking M. The class of languages generated in this manner is designated $\mathcal{L}$, or $\mathcal{L}^\lambda$ if there are λ-transitions. If M is understood, we may also write $\mathcal{L}(N)$, for example.

We shall study these languages from the point of view of their closure properties,

their relation to other language families, and their generation from a set of basis languages by certain of their closure properties. This will permit us to give several characterizations of the language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$.

We shall also consider the decidability of membership, emptiness, and equivalence for these language families. Furthermore, we shall show how the introduction of a priority firing rule, or of a zero-testing capability, extends the family $\mathcal{L}_0^\lambda$ to all type 0 (recursively enumerable) languages. This latter fact is of interest because many practical synchronization problems involve priority rules of the type "if both A and B are enabled, only A shall be allowed to proceed".

Similar results have recently been obtained by T. Agerwala [1], and J. Peterson has studied a family of languages practically identical to our family $\mathcal{L}_0$ [17].

1. Basic Definitions

1.1. Generalized Petri Nets.

Definition 1.1: A Generalized Petri Net (GPN) $N = \langle \Pi, \Sigma, F, B, M_0 \rangle$ consists of the following:

1. a finite set of places, $\Pi = \{p_1, \ldots, p_r\}$
2. a finite set of transitions, $\Sigma = \{t_1, \ldots, t_s\}$ disjoint from Π
3. a forwards incidence function $F: \Pi \times \Sigma \to \mathbb{N}$ ($\mathbb{N}$ is the set of non-negative integers)
4. a backwards incidence function $B: \Pi \times \Sigma \to \mathbb{N}$
5. an initial marking $M_0: \Pi \to \mathbb{N}$

It is represented graphically as follows:

1. places are represented by circles
2. transitions are represented by bars
3. circles and bars are connected by bundles of arcs: if p is a place and t is a transition, and F(p, t) = 3, we have a bundle of 3 arcs going from p to t; 3 is the size of the arc bundle.
4. a marking is represented by drawing a number of tokens into a place, or writing the number.

$\Pi = \{p_1, p_2, p_3\}$

$M_0 = \langle 5, 1, 0 \rangle$

$\Sigma = \{t_1, t_2, t_3, t_4\}$

$F(p_i, t_j) =$

i \ j	1	2	3	4
1	1	3	0	0
2	0	1	0	0
3	0	0	1	5

$B(p_i, t_j) =$

i \ j	1	2	3	4
1	0	0	1	0
2	2	1	0	0
3	0	2	0	0

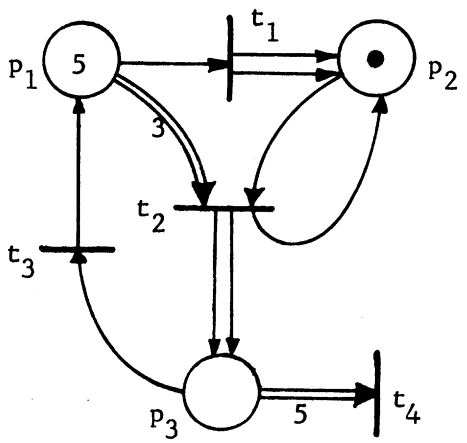


Figure 1.1

When we draw a bundle of arcs we expect each fibre to carry along one token when a transition fires. The firability of a transition is thus defined as follows:

> A transition t is said to be firable iff for every place $p \in \Pi$ we have $M(p) \geq F(p, t)$. Since this is always true when $F(p, t) = 0$ we need to inspect only the input places of transition t, i.e. those for which $F(p, t) > 0$.

If a firable transition fires, it changes the marking by removing $F(p, t)$ tokens if p is an input place and by adding $B(p, t)$ tokens if p is an output place ($B(p, t) > 0$). The new marking M' is now such that: $\forall p$: $M'(p) = M(p) - F(p, t) + B(p, t)$, which can be abbreviated as: $M' = M + (B(t) - F(t))$. By ordering the set of places Π we can interpret markings and the effect of firing on a marking as vectors of r-coordinates 1...r corresponding to the places $p_1 \ldots p_r$.

In this context we may also view F(t) and B(t) as vectors in $\mathbb{N}^r$, where $F(t)(i) = F(p_i, t)$ for $1 \leq i \leq r$. Then we can define the relation $M[t\rangle M'$ which says: "Transition t is firable at marking M and leads to marking M' " as follows:

$$M[t\rangle M' \quad \Leftrightarrow \quad M \geq F(t) \;\&\; M' = M - F(t) + B(t)$$

A firing sequence can now be defined as a sequence of transition names (or a string σ in Σ^*), such that each prefix leads to a marking at which the following transition is firable. Thus, Figure 1.2 shows the result of firing t_2 in the Petri net of Figure 1.1. Since t_3 is firable at that new marking, $t_2 t_3$ is a firing sequence. Note that $t_3 t_2$ is not a firing sequence, since t_3 is not firable at the initial marking.

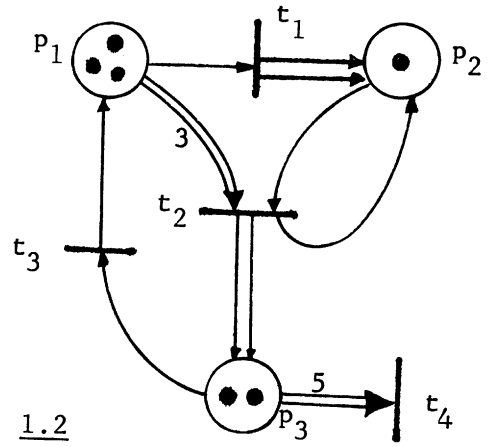


Figure 1.2

The dynamic aspects of the Petri Net N can now be described by the set of firing sequences $S_N(M_0)$ starting at the initial marking M_0, and by the set of reachable markings $R_N(M_0)$, i.e. the markings M' such that some firing sequence $\sigma \in S_N(M_0)$ leads from M_0 to M'. This we write as $M_0[\sigma\rangle M'$, where the relation $[\sigma\rangle$ is defined as the composition of the relations $[t_i\rangle$ for the transitions t_i as they occur in the string σ; composition for the relations corresponds to concatenation for the strings of transition names, so that:

$$M[\sigma t\rangle M' \quad \Leftrightarrow \quad \exists M'' \in \mathbb{N}^r \quad : \quad M[\sigma\rangle M'' \ \& \ M''[t\rangle M'$$

Thus we have:

$$S_N(M_0) = \{\sigma \in \Sigma^* \mid \exists M' \in \mathbb{N}^r : M_0[\sigma\rangle M'\}$$

$$R_N(M_0) = \{M \in \mathbb{N}^r \mid \exists \sigma \in \Sigma^* : M_0[\sigma\rangle M\}$$

Given a "final" marking M_f, we also define the set of terminal firing sequences $T_N(M_0, M_f)$ which contains all those firing sequences which lead from M_0 to M_f :

$$T_N(M_0, M_f) = \{\sigma \in \Sigma^* \mid M_0[\sigma\rangle M_f\}$$

Clearly, $T_N(M_0, M_f) \subseteq S_N(M_0)$ and $M_f \notin R_N(M_0) \Leftrightarrow T_N(M_0, M_f) = \emptyset$.
We notice that:

$$M_0[\sigma\rangle M \ \Rightarrow \ \sigma \in S_N(M_0) \ \& \ M \in R_N(M_0)$$

Finally, we observe the following effect of increasing the initial marking. Let $M_0' \geq M_0$. Then we have:

$$M_0[\sigma\rangle M_1 \ \Rightarrow \ (M_0'[\sigma\rangle M_1' \quad \& \quad M_1' - M_1 = M_0' - M_0)$$

$$S_N(M_0) \subseteq S_N(M_0') \ \dagger$$

$$\{M \in \mathbb{N}^r \mid \exists M_1 \in R_N(M_0) \ \& \ M = M_1 + M_0' - M_0\} \subseteq R_N(M_0')$$

$$T_N(M_0, M_f) \subseteq T_N(M_0', M_f + M_0' - M_0)$$

---

† This is also called the containment property.

1.2. Restricted Petri Nets

In some cases it is useful to restrict the definition of Petri Nets. Ordinary Petri Nets are GPN's where the size of arc bundles is restricted to one. This corresponds to Holt's original definition [5,10]. Selfloop-free Petri Nets have no pairs p,t that are both forwards and backwards connected, i.e. $B(p, t) \cdot F(p, t) = 0$ for all places p and transitions t. Restricted Petri Nets† (RPN) are Selfloop-free ordinary Petri Nets: any place-transition pair is connected by at most one arc.

The relations between these various restrictions and Vector Addition Systems are discussed in a more detailed manner in [7]. See also Miller [14].

1.3. Labelled Petri Nets and Petri Net Languages

Definition 1.2: A Labelled Petri Net $A = \langle N, \mathcal{A}, \Lambda \rangle$ over an alphabet $\mathcal{A}$ is a GPN $N = \langle \Pi, \Sigma, F, B, M_0 \rangle$ together with a labelling function $\Lambda: \Sigma \to \mathcal{A}$. If the labelling function is only partial, the net is said to contain λ-transitions, namely those transitions that have no label assigned by Λ.

The labelling function is extended to firing sequences in the natural way. If $\sigma t \in \Sigma^*$ is a firing sequence, we have:

$$\begin{aligned} \Lambda(\sigma t) &= \Lambda(\sigma) \cdot \Lambda(t) && \text{iff t has a label} \\ &= \Lambda(\sigma) && \text{otherwise.} \end{aligned}$$

Finally, $\Lambda(\lambda) = \lambda$. (the empty string)

Thus, λ-transitions in firing sequences transform as if their label were the empty string λ.

If σ is a firing sequence, then $\Lambda(\sigma)$ is called a label sequence. Note that a given label sequence may correspond to several firing sequences, and that if there are λ-transitions, the firing sequences can be longer than the corresponding label sequences -- in fact, a given label sequence may correspond to arbitrarily long firing sequences. This is why we shall distinguish between λ-free Labelled Petri Nets (no λ-transitions) and unrestricted Labelled Petri Nets.

In the context of system modelling, λ-transitions may correspond to "internal," "invisible," or otherwise "uninteresting" events, whereas similarly

† C.A. Petri calls these nets "Pure Petri Nets".

labelled transitions may be used to model actions which can be enabled by several distinct sets of conditions. It should be pointed out, however, that describing the behavior of a system by means of label sequences may completely hide inherent concurrency, since each label sequence corresponds to a totally ordered execution sequence.

The languages generated by Petri Nets are the sets of label sequences corresponding to all firing sequences, or just all terminal firing sequences, of the Petri Net.

We, therefore, distinguish four classes of Petri net languages.

Definition 1.3:

(a) $\mathcal{L}$ is the class of all languages obtained as the set of all label sequences of some λ-free Labelled Petri Net:

$$L \in \mathcal{L} \overset{\Delta}{\Leftrightarrow} \exists A = \langle N, \mathcal{A}, \Lambda \rangle;\ N = \langle \mathbb{N}, \Sigma, F, B, M_0 \rangle:$$
$$L = \{x \in \mathcal{A}^* \mid \exists \sigma \in S_N(M_0)\ \&\ x = \Lambda(\sigma)\}$$

and Λ is a total function on Σ. We write this as $L = S_A(M_0)$ (or $\mathcal{L}(A)$, if M_0 is understood), where A is the labelled net.

(b) $\mathcal{L}^\lambda$ is defined like $\mathcal{L}$, except that λ-transitions are permitted: Λ may be partial on Σ.

(c) $\mathcal{L}_0$ is the class of all λ-free languages obtained as the set of all terminal label sequences of a λ-free Labelled Petri Net:

$$L \in \mathcal{L}_0 \overset{\Delta}{\Leftrightarrow} \begin{cases} \exists A \text{ Labelled Petri Net as above} \\ \exists M_f \text{ final marking for the Petri Net, } M_f \neq M_0: \end{cases}$$
$$L = \{x \in \mathcal{A}^* \mid \exists \sigma \in T_N(M_0, M_f)\ \&\ x = \Lambda(\sigma)\}$$
and Λ is total on Σ.

We write this as: $L = T_A(M_0, M_f)$
Or, if the markings are understood: $\mathcal{L}_0(A)$

(d) $\mathcal{L}_0^\lambda$ is defined like $\mathcal{L}_0$, except that λ-transitions are allowed, and that the language may contain the empty string λ.

Figure 1.3 illustrates this definition.

	no λ-transitions (λ-free)	λ-transitions allowed
all firing sequences	$\mathcal{L}$	$\mathcal{L}^{\lambda}$
only terminal firing sequences	$\mathcal{L}_0$	$\mathcal{L}_0^{\lambda}$

Figure 1.3

Remark: The exclusion from $\mathcal{L}_0$ of languages containing the empty string may seem arbitrary. It is motivated by reasons similar to those excluding λ from context-sensitive languages according to the strict definition. In particular, allowing the initial and final markings of a Labelled Petri net to be the same (the only way to include λ in the $\mathcal{L}_0$-language) causes unwanted side-effects: it demands special consideration in various proofs, and it also implies that if an $\mathcal{L}_0$-language L contains λ, then it is closed under concatenation and Kleene-star: $L = L \cdot L = L^*$. As a result, the class $\mathcal{L}_0$ would not be closed under union, since both {λ} and {a} would be in $\mathcal{L}_0$, but {λ, a} would not be in $\mathcal{L}_0$. The only way to avoid this peculiarity would be to allow for two possible final markings,* one of which is the initial marking. Since this is not needed for the other families, we choose the λ-free alternative as the most consistent one in the general context of this report.

We notice that languages in $\mathcal{L}_0^{\lambda}$ may or may not contain the empty string, whereas languages in $\mathcal{L}_0$ cannot contain the empty string, and languages in $\mathcal{L}$ or $\mathcal{L}^{\lambda}$ must contain the empty string, since the initial marking is reachable by the empty firing sequence.

For certain purposes it may be useful to define the class of cyclic (terminal) languages, which are generated by nets whose final marking is the same as their initial marking. (The language family $\mathcal{L}_c$; we may also define $\mathcal{L}_c^{\lambda}$.)

---

*This is Peterson's [17] approach: He allows any number of final markings including the initial marking. His Computation Sequence Sets are thus slightly more general than the class $\mathcal{L}_0$. We will see in Section 2.4 that a CSS which does not contain λ is in $\mathcal{L}_0$, and if it contains λ, it is equal to the union of {λ} and some language in $\mathcal{L}_0$. Conversely, $\mathcal{L}_0$ is precisely the class of λ-free CSS.

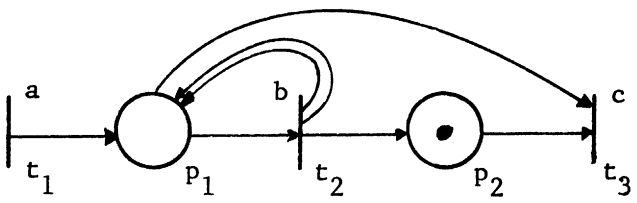
2. Standard Forms for Labelled Petri Nets

2.1. Some Language-Preserving Transformations Using λ-Transitions

For many constructions, such as will be used in later proofs, it is useful to impose certain constraints on the Petri nets used to generate various languages, as long as the additional constraints do not change the generating power of the class of nets considered. Since many of the desired properties can easily be achieved by using additional λ-transitions, we shall first discuss Petri nets generating languages in $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$.

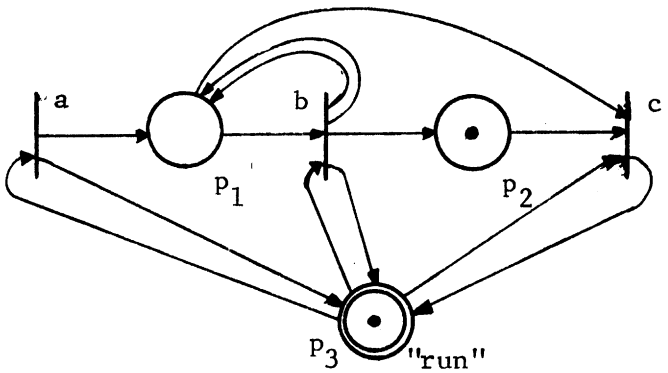
2.1.1. The "Run" Place

A useful property of a Petri net is the ability to be "switched on" or "off." This can be done by adding one place which self-loops on every transition in the Petri net. If this place contains a token, then the net behaves exactly as without the new place, but if we remove this token, then all transitions are disabled. If we apply the construction to the example of Figure 2.1, we get the net in Figure 2.2.



initial marking: ⟨ 0, 1⟩
final marking : ⟨ 1, 0⟩

Figure 2.1



initial marking: ⟨ 0, 1, 1⟩
final marking : ⟨ 1, 0, 1⟩

Figure 2.2

2.1.2. The "Start" Place

Another useful modification is the standardization of the initial and final markings. Unless we wish to study how the language generated by a Petri net depends on the initial (and/or final) marking, we shall consider the standard initial marking to be one token in a designated "start" place, and zero tokens everywhere else. This can be combined with the "on-off" control place (which we call the "run" place) introduced above: The first transition firing in the modified net removes the token in the "start" place, deposits the proper initial marking, and drops a token into the "run" place. In the case of $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$, this "first" transition can be a λ-transition, and the construction is trivial.

2.1.3. The "Stop" Transition

Similarly, we choose the zero marking (zero tokens in all places) as the standard final marking. Used in conjunction with the "run" place, this convention implies that no transition is firable at the final marking. Basically, all we need is that the last transition firing removes all tokens, including the "run" token, from the net. In the case of $\mathcal{L}_0^{\lambda}$, this is again very simply achieved with an output-less λ-transition, called the "stop" transition, which removes the final marking and the "run" token. The result of adding the "start" and "stop" capabilities to the net of Figure 2.2 is shown in Figure 2.3.

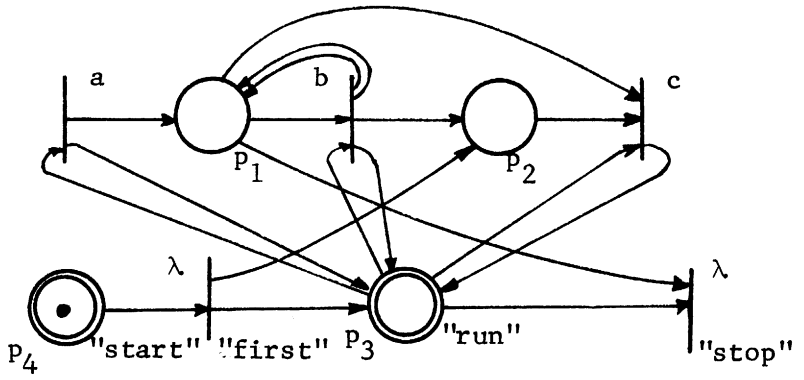


Figure 2.3

2.1.4. The "Clear" Place, for $\mathcal{L}^{\lambda}$.

In the case of a Petri net used to generate an $\mathcal{L}^{\lambda}$-language, no final marking is involved, but we can still stop the net by removing the token from the "run" place with a "stop" transition. In fact, we can go further and transfer the token from the "run" place into a "clear" place before discarding it. The "clear" place self-loops on a series of new λ-transitions, one per "ordinary" place, which remove all tokens from the net. See the construction in Figure 2.4.

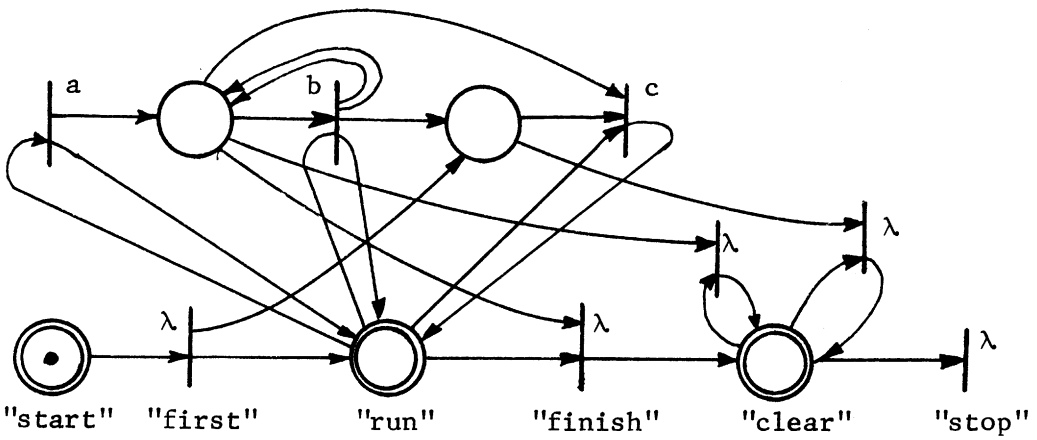


Figure 2.4

This last construction also shows that $\mathcal{L}^{\lambda} \subseteq \mathcal{L}_0^{\lambda}$, since the terminal language ($\mathcal{L}_0^{\lambda}$) of the modified net is precisely the same as the firing language ($\mathcal{L}^{\lambda}$) of the old net. The containment is proper since a language in $\mathcal{L}_0^{\lambda}$ does not have to contain the empty string, as is the case for $\mathcal{L}^{\lambda}$. We note this, together with other obvious inclusions, in:

Theorem 2.1: $\mathcal{L}^{\lambda} \subseteq \mathcal{L}_0^{\lambda}$ & $\mathcal{L}^{\lambda} \neq \mathcal{L}_0^{\lambda}$

$\mathcal{L} \subseteq \mathcal{L}^{\lambda}$

$\mathcal{L}_0 \subseteq \mathcal{L}_0^{\lambda}$ & $\mathcal{L}_0 \neq \mathcal{L}_0^{\lambda}$

Thus $\mathcal{L}_0^{\lambda}$ is the most general family, as expected; $\mathcal{L}$ is the least general. We will show later (Theorem 5.9) that we also have $\mathcal{L} \subseteq \mathcal{L}_0$ (up to λ). An important open question is whether $\mathcal{L}_0 = \mathcal{L}_0^{\lambda} - \{\lambda\}$.

Remark: The preceding constructions are of the same kind as those used in the reducibility proofs for the Reachability Problem in Hack [7, 8].

2.1.5. Elimination of Multiple Labels

In some cases it may be useful to have only one transition labelled with a given symbol -- for example, when trying to interconnect two nets by sharing similarly-labelled transitions, as will be done in some of the constructions of Chapters 3 and 4. If the net has a "run" place, this is easy to accomplish: We add one place per symbol in the alphabet, and use these places to break the self-loops of the "run" place, as shown on an example in Figure 2.5, which transforms into the net of Figure 2.6. Each new place remembers the symbol of the last firing, and the symbol itself is generated when the "run" token is returned to the "run" place. All old transitions become λ-transitions. Thus:

Theorem 2.2: All $\mathcal{L}_0^\lambda$ and $\mathcal{L}^\lambda$-languages can be generated without using multiply labelled transitions.

2.2. Some Language-Preserving Transformations Without λ-Transitions

If we are studying languages in $\mathcal{L}_0$ or $\mathcal{L}$ and we want to modify a λ-free Labelled Petri net generating such a language, we would like to be able to keep the net λ-free. The constructions of the preceding paragraph must thus be modified.

2.2.1. The "Run" Place

Adding a "run" place to a λ-free net, as in 2.1.1, does not by itself introduce any new transitions. But if we want to switch the net "on" or "off", we will have to do so in a λ-free manner.

2.2.2. The "Start" Place: Switching the Net "On"

As before, the objective is to have a standard initial marking of exactly one token in a designated "start" place, and zero tokens everywhere else. The first transition firing will then put a token on the "run" place. So let us consider all those transitions which might be the first to fire, at the initial marking. For each transition t_i firable at the initial marking M_0, we add a

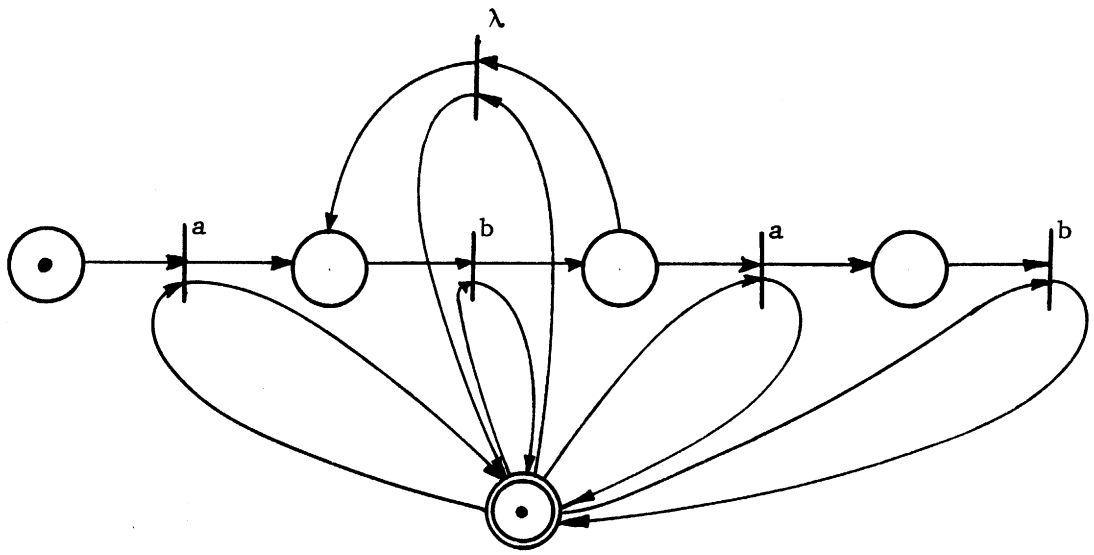


Figure 2.5

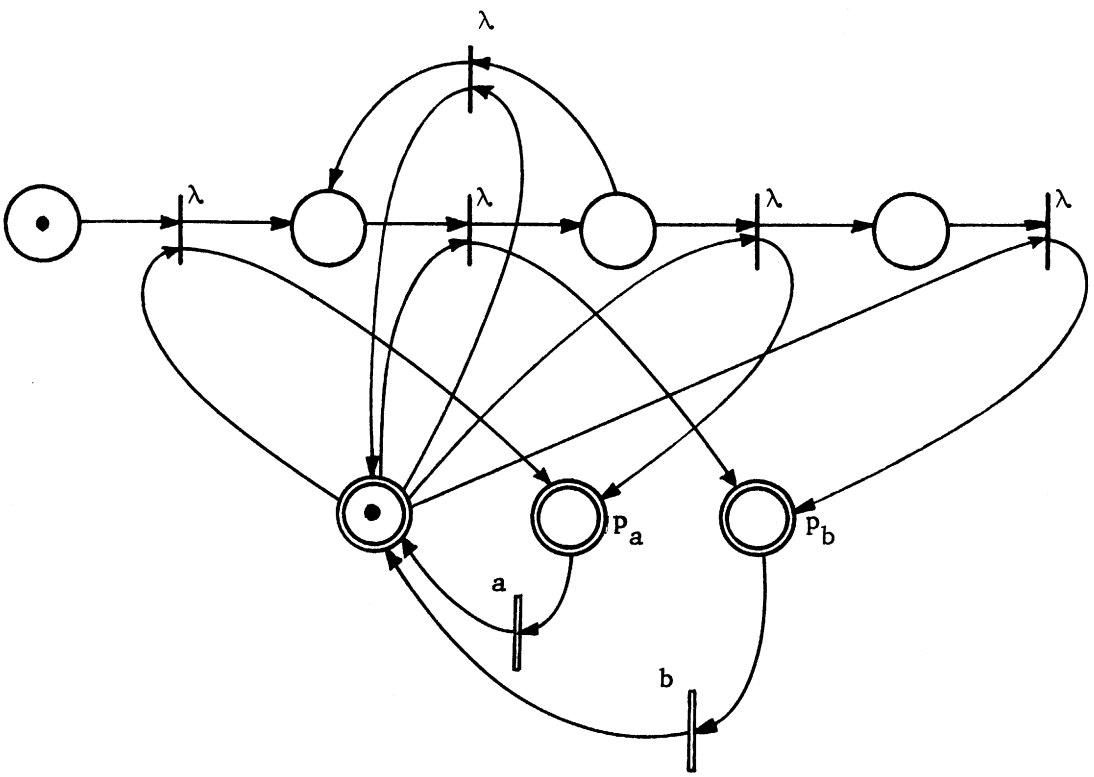
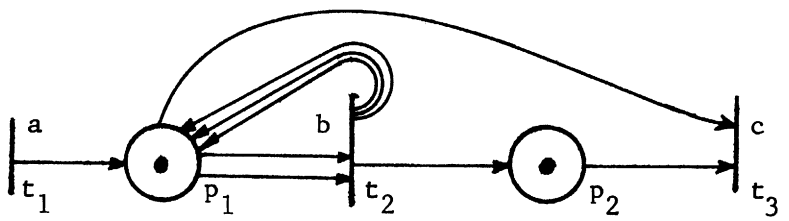


Figure 2.6

new transition t_i', labelled like t_i, which removes the "start" token, deposits the "run" token, and deposits the marking $M_0[t_i\rangle$ which would have resulted from a firing of t_i at M_0. This does not change the set of label sequences generated by the net. The set of primed transitions is called "first."

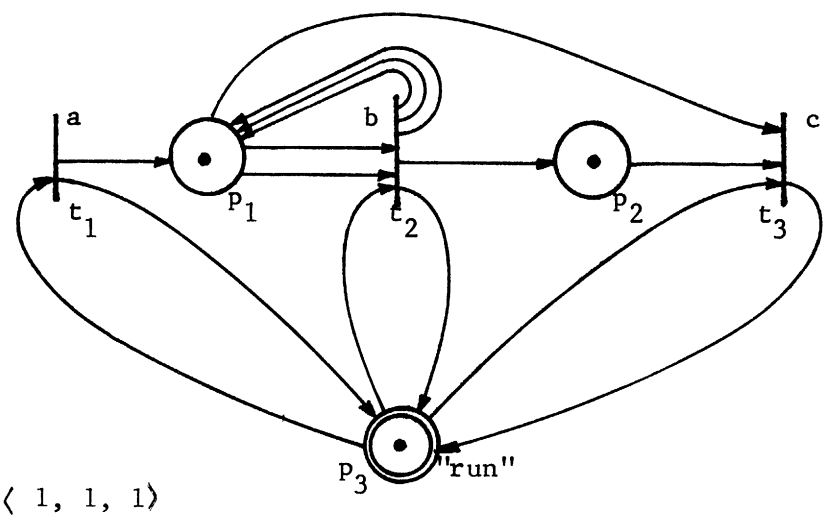
Example: Figure 2.7.



$M_0 = \langle 1, 1\rangle$

Figure 2.7

Adding a "run" place, we get Figure 2.8:



$M_0 = \langle 1, 1, 1\rangle$

Figure 2.8

The only transitions firable at M_0 at t_1 and t_3, We thus add a "start" place and new transitions t_1' and t_3' to get Figure 2.9.

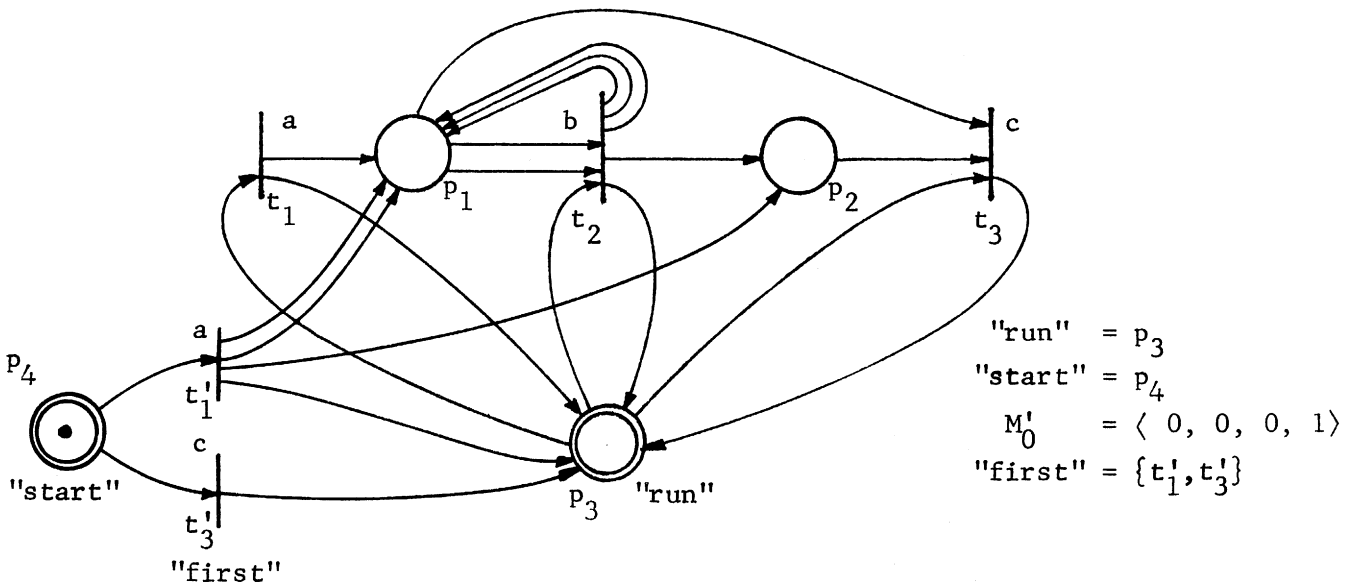


Figure 2.9

The output arcs of t_1' and t_3' are determined by the fact that $M_0[t_1\rangle = \langle 2, 1\rangle$ and $M_0[t_3\rangle = \langle 0, 0\rangle$, so that:

$$M_0'[t_1'\rangle = \langle 2, 1, 1, 0\rangle$$

$$M_0'[t_3'\rangle = \langle 0, 0, 1, 0\rangle$$

Clearly, the set of label sequences has not been changed, because t_i and t_i' have the same label, and t_i' can fire only once, replacing precisely the first occurrence of t_i.

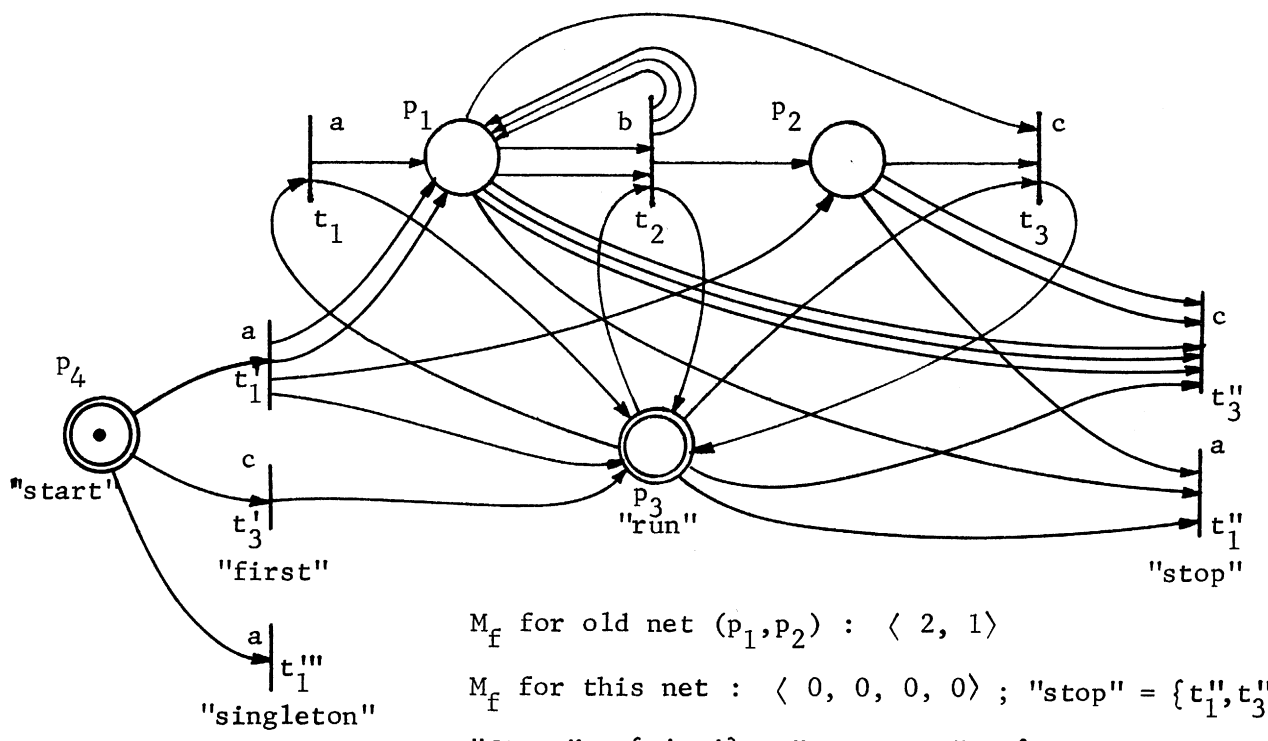
2.2.3. The "Stop" Transitions

Now we shall try to switch a Labelled Petri net "off" in a λ-free manner, by removing a token from the "run" place. If we are interested only in terminal firing sequences (which reach a final marking M_f), we can at the same time introduce a standard final marking, namely the zero marking.

To do this, we first construct a new set of "stop" transitions t_i'' corresponding to those transitions which are likely to fire last, i.e. those t_i such that there exists a penultimate marking M: $M[t_i\rangle = M_f$. These transitions t_i'' will then be used to remove precisely the corresponding penultimate marking (which may be denoted by $[t_i\rangle M_f$ or $M_f[-t_i\rangle$), as well as the "run" token.

We must also be able to take care of terminal firing sequences of a single transition, i.e. the case where $M_0[t_i\rangle M_f$. For each such transition, we introduce a new transition t_i''' with the same label which simply removes the "start" token. Such transitions are called "singleton." Since we assume $M_f \neq M_0$, there is no other case to be considered.

As an example, let us use the Labelled Petri net of Figure 2.7 with a final marking $\langle 2, 1\rangle$, or the partly modified net of Figure 2.9 with a final marking $\langle 2, 1, 1, 0\rangle$, as shown in Figure 2.10.



M_f for old net (p_1, p_2) : $\langle 2, 1\rangle$

M_f for this net : $\langle 0, 0, 0, 0\rangle$; "stop" = $\{t_1'', t_3''\}$

"first" = $\{t_1', t_3'\}$; "singleton" = $\{t_1'''\}$

Figure 2.10

It is clear that the new net will reach the zero marking iff the old net reached M_f from M_0; the new firing sequence is obtained from the corresponding old firing sequence by priming the first transition and double-priming the last one (or triple-priming a single firing); the terminal label sequence will be unchanged.

It turns out that the set of non-terminal firing sequences is also unchanged by this construction. To show this, we have to prove that the "stop" transition t" corresponding to t can only fire if t could also fire. In other words, we must show that $F(t") \geq F(t)$. Indeed, a stop transition t" is introduced only if M_f can be reached from a penultimate marking by a firing of t, which implies $M_f \geq B(t)$. But in this case, we have, by construction: $F(t") = M_f - (B(t) - F(t)) \geq F(t)$.

If we are generating a language in $\mathcal{L}$, and consider all label sequences, the Petri net can be stopped at any time. This can be done by introducing a "stop" transition $t_i"$ for each transition t_i (with the same label), where $t_i"$ removes the same tokens as t_i, plus the "run" token; in addition, we introduce a "singleton" transition t_i''' for each "first" transition t_i', where t_i''' simply removes the "start" token. This does not change the set of label sequences, but we must remember that the net can only be stopped after at least one firing. Figure 2.11 shows the result of the construction applied to the net of Figure 2.9:

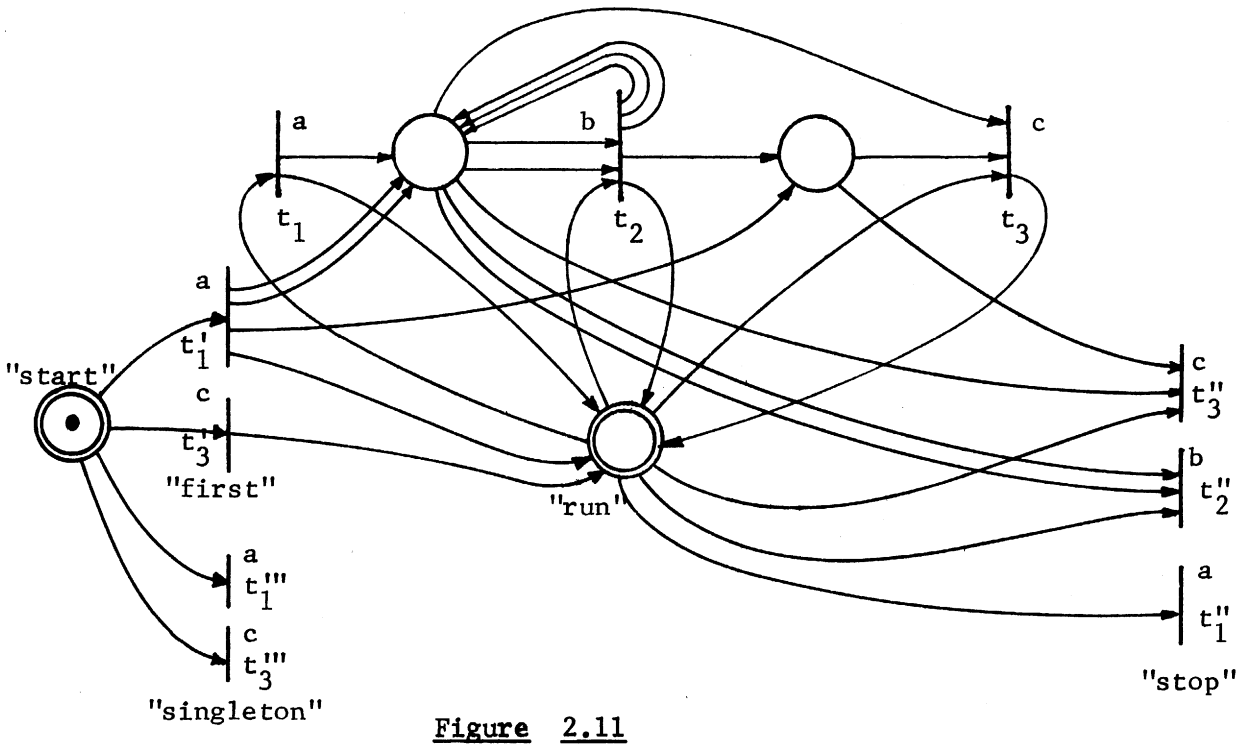


Figure 2.11

If we wanted to save the marking reached after stopping the net, we could of course have the "singleton" and "stop" transitions return the correct number of tokens, with the exception of the "run" or "start" token, of course.

2.3. The Standard Form Theorem

By applying the transformations described in the previous paragraphs to a Labelled Petri net, we can obtain a new net generating the same language and satisfying the following additional properties:

SF1: There is a distinguished place, called the "start" place, which is the output of no transition. The initial marking consists of exactly one token in the "start" place, and zero tokens in all other places.

SF2: For the purpose of terminal sequences ($\mathcal{L}_0$, $\mathcal{L}_0^\lambda$), the standard final marking is the zero marking, at which no transition is firable. (Each transition has at least one input place.)

SF3: The transitions are partitioned into four groups (some of which may be empty):

- "singleton": These transitions have as only input the "start" place, and no output places. They are only used in λ-free nets, where they generate the one-symbol strings of the language ($\mathcal{L}_0$ or $\mathcal{L}$).
- "first": These transitions have as only input the "start" place, but they have at least one output place. In λ-free nets, these transitions represent the first symbol generated in any string. If there are λ-transitions ($\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$) this group consists exactly of one λ-transition.
- "stop": These transitions have no output places, but they are not connected to the "start" place. Only one "stop" firing may ever occur, and its occurrence leaves every transition disabled so that no further firings can occur. In the case of $\mathcal{L}_0$ this firing represents the last symbol in a terminal string

(of length >1) if the resulting marking is the zero marking. If there are λ-transitions, this group consists of a single λ-transition.

- "internal": All other transitions.

If a Labelled Petri net satisfies these three Standard Form conditions, it is said to be in Standard Form. The constructions of paragraphs 2.1.3. and 2.2.3 generate nets in Standard Form, but a net need not look like the result of these constructions; in particular, nothing is said about a single "run" place. If it is desirable to have a central "run" place whose token can be removed to disable all transitions, such a place can be added and connected in a self-loop on every "internal" transition; it gets a token from every "first" transition and has its token removed by every "stop" transition.

We can thus assert the following Standard Form Theorem:

Theorem 2.3: Every $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$-language can be generated by a net in Standard Form, and every string in the language (except the empty string in the case of $\mathcal{L}$) can be generated in such a way that the last transition firing was a "stop" or "singleton" transition (and thus leaves every transition disabled).

2.4. The Relation of $\mathcal{L}_0$ to Peterson's Computation Sequence Sets

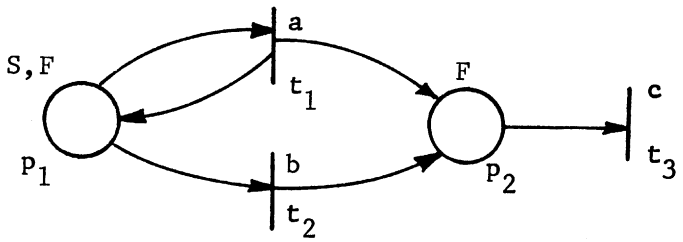
Peterson has studied the languages generated by Labelled Petri nets (which Peterson calls "p-nets") having a standard initial marking of exactly one token in a designated "start" place, and a set of final markings consisting in exactly one token in exactly one of several possible "final" places, possibly including the "start" place. The generated language consists of those label sequences (called "Computation Sequences," or CS) which lead from the initial marking to one of the final markings. The language is called a "Computation Sequence Set," or CSS [17b].

It is clear that every $\mathcal{L}_0$-language is a CSS: All we have to do is take a λ-free Labelled Petri net in standard form and then supply a "final" place which gets a token from every "stop" or "singleton" transition. The corresponding CSS is the original $\mathcal{L}_0$-language. If we list the "start" place as another "final" place, we see that augmenting an $\mathcal{L}_0$-language with the empty string also yields a CSS.

We will now show that the converse is true, namely that every λ-free CSS is in $\mathcal{L}_0$, and that if a CSS contains the empty string, we get a language in $\mathcal{L}_0$ by restricting the CSS to its non-empty strings.

Assume we are given a Labelled Petri net with an initial marking and several possible final markings, one of which may be the initial marking. We add to this net a new "start" place, a "run" place, and the appropriate set of "first" transitions, as indicated sub 2.2.2. Then we add a set of "singleton" and "stop" transitions, as indicated sub 2.2.3, for <u>each</u> final marking (possibly including the initial marking). Now it is clear that the zero marking in this new net will be reached by some firing sequence iff the corresponding firing sequence in the original "p-net" has reached one of the final markings, except for the empty firing sequence.

An example of a "p-net" is shown in Figure 2.12.

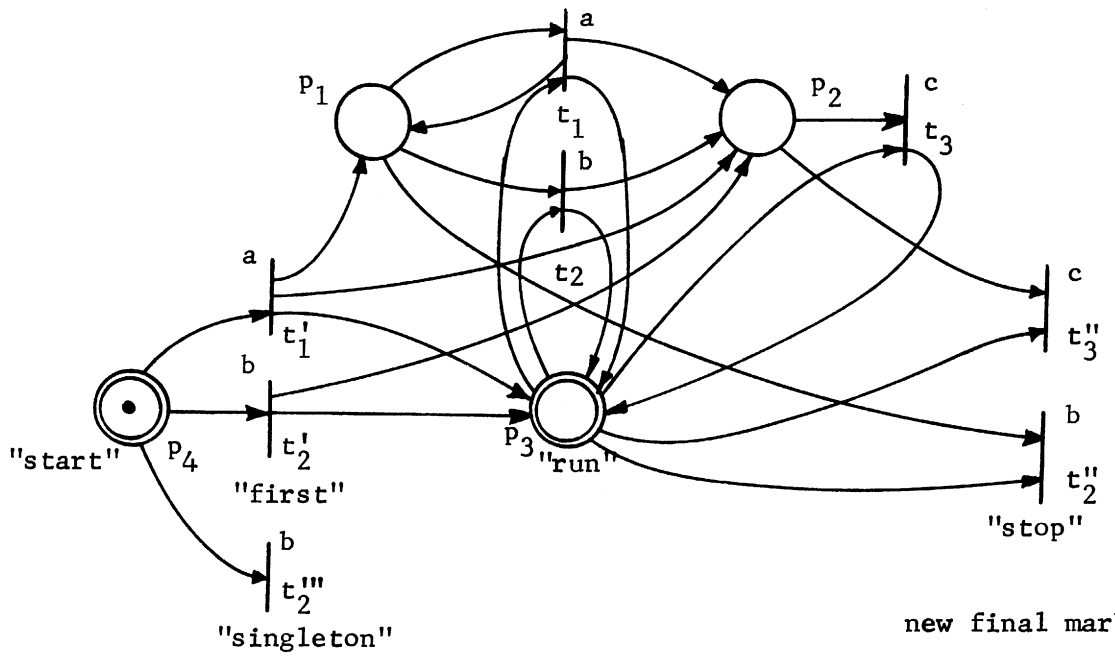


P-net:
initial (S) : ⟨ 1, 0⟩
final (F) : ⟨ 1, 0⟩
(2 possibilities) ⟨ 0, 1⟩

CSS-language: $a^n(\lambda + b)c^n$, for all $n \geq 0$.

Figure 2.12

The result of the transformation applied to this "p-net" yields the Labelled Petri net of Figure 2.13.



$\mathcal{L}_0$-language: $a^n(\lambda + b)b^n + b$, for all $n \geq 1$.

Figure 2.13

For example, the p-net in Figure 12 is transformed into the Labelled Petri net of Figure 13, whose terminal language is the CSS defined by the p-net, minus the empty string.

Note: In his thesis [17a] Peterson used p-nets with a single final place, and was mainly interested in the case where the start and final places were different, so that he was, in fact, describing the subset of CSS which exactly corresponds to $\mathcal{L}_0$. His notion of "well behaved" is also very closely related to our "Standard Form".

Figure 2.13

for example, the p-net in Figure 12 is transformed into the labelled Petri net of Figure 13, whose terminal language is the CSS defined by the p-net, minus the empty string.

Note: In his thesis [12a] Peterson used p-nets with a single final place, and was mainly interested in the case where the start and final places were different, so that he was, in fact, describing the subset of CSS which exactly corresponds to $\mathcal{L}_0$. His notion of "well-behaved" is also very closely related to our "standard form".

3. Simple Closure Properties

In this section we investigate the closure properties of the language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ under the operations of concurrency, union, intersection and concatenation. A family of languages is said to be closed under an operation if that operation applied to two languages in the family yields a language in the family.

Remark: The results for $\mathcal{L}_0$ are essentially the same as those obtained by Peterson [17], in view of Section 2.4.

3.1. Closure Under Concurrent Composition

The expression of concurrency or parallelism is a natural property of Petri nets; indeed, Petri nets were originally developed precisely to permit a clear and easy representation of concurrency and parallelism.

Consider the Labelled Petri net of Figure 3.1.

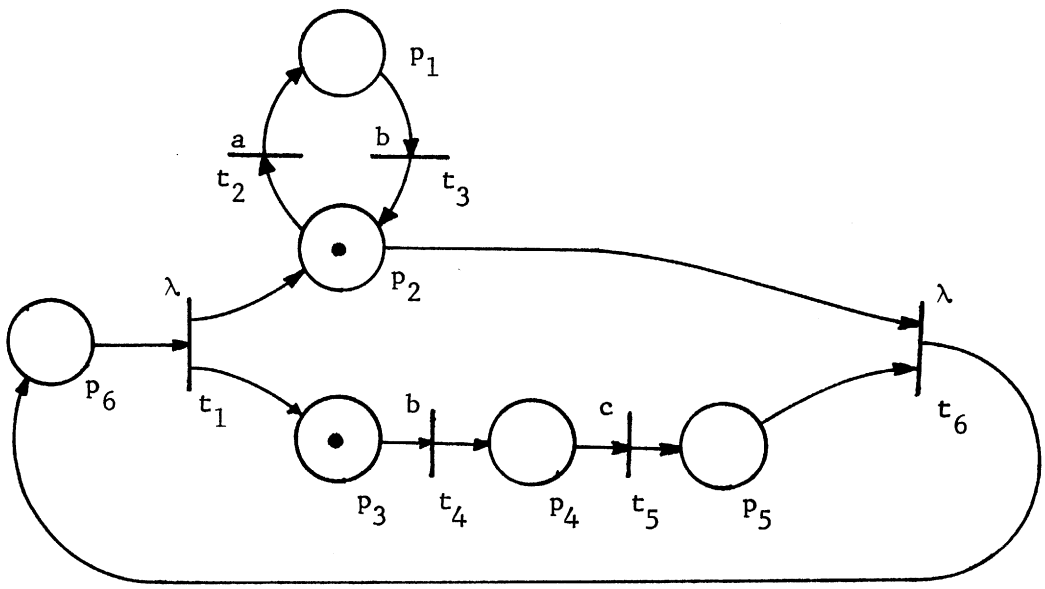
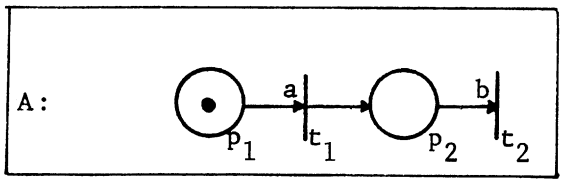
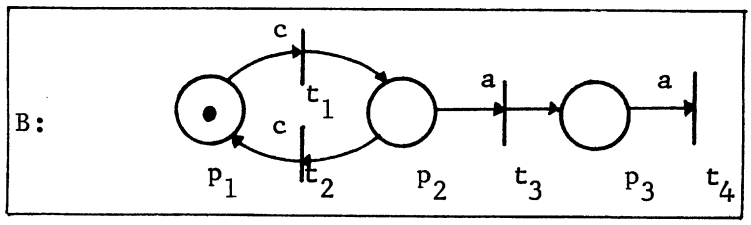


Figure 3.1



initial marking: ⟨ 1, 0⟩
final marking: ⟨ 0, 1⟩

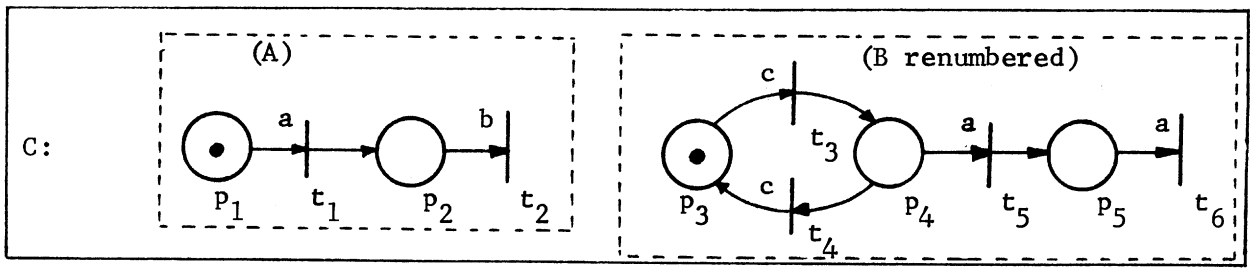
$\mathcal{L}(A) = \{\lambda, a, ab\}$
$\mathcal{L}_0(A) = \{a\}$



initial marking: ⟨ 1, 0, 0⟩
final marking: ⟨ 0, 1, 0⟩

$\mathcal{L}(B) = (cc)^*(\lambda + c + ca + caa)$
$\mathcal{L}_0(B) = (cc)^*$

Figure 3.2



initial: ⟨1,0,1,0,0⟩
final : ⟨0,1,0,1,0⟩

$\mathcal{L}(C) = \mathcal{L}(A) \parallel \mathcal{L}(B)$
$\mathcal{L}_0(C) = \mathcal{L}_0(A) \parallel \mathcal{L}_0(B) = (cc)^*(ac + ca)(cc)^*$

Figure 3.3

This net exhibits a simple cyclic behavior, which is most easily described by observing that between firings of t_1 and t_6, one portion generates the regular language $(ab)^*$, the other the string bc. A sample firing sequence would be $t_2t_3t_2t_4t_3t_2t_5t_3$, generating ababbacb. This can be expressed quite conveniently by using the concurrency operator, or parallel composition operator, denoted by $\|$; as in $(ab)^*\|bc$. If we had to write a regular expression for this, it would be rather complicated, like $(ab)^*(b(ab)^*(c + acb) + abcb)(ab)^*$.

The concurrent composition of two strings x and y can thus be defined as the set of strings obtained by merging the symbols from x and y into a new string z such that both x and y appear in z as a scattered substring; the length of z is the sum $|z| = |x| + |y|$. One can imagine two automata generating x and y respectively, in parallel and asynchronously, and writing the symbols on a shared output tape. Formally, we have:

Definition 3.1: The concurrent composition $x\|y$ of two strings $x,y \in \mathcal{A}^*$ is defined recursively as follows:

$$\forall a \in \mathcal{A}: \quad a\|\lambda = \lambda\|a = \{a\}$$

$$\left.\begin{array}{l}\forall a,b \in \mathcal{A} \\ \forall x,y \in \mathcal{A}^*\end{array}\right\} \quad a\cdot x\|b\cdot y = \{a\}\cdot(x\|(b\cdot y)) \cup \{b\}\cdot((a\cdot x)\|y)$$

The dot stands for concatenation, and the operators are extended to sets of strings in the natural way. Thus, for example:

$$ab\|c = \{abc, acb, cab\}$$

$$\{ab, c\}\|\{a, \lambda\} = \{aba, aab, ca, ac, ab, c\}$$

As we mentioned before, concurrency is a natural property of Petri nets, and indeed, closure under concurrency can be trivially established for Petri net languages.

Let A and B be two labelled Petri nets generating L_A and L_B, respectively (in one of the families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$). Let C be the juxtaposition of A and B, i.e. C is a new labelled Petri net obtained by regarding A and B as parts of one net, after renumbering the places and transitions of one component, say B. Thus the nets A and B of Figure 3.2 become the net of Figure 3.3. The

markings of C (including the specified initial and final markings) are likewise the "vector-concatenation" of the markings of A and B, as indicated by the renumbering. This simple juxtaposition does not introduce any new transitions, and neither adds nor removes constraints from the concurrently operating parts A and B of C. This permits us to assert:

Theorem 3.1: $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$ are closed under concurrent composition (concurrency, parallelism).

Remark: When standard forms are used, it is useful to share the "run" place. To get a single "start C" place, we duplicate the "firstA" transitions and have them deposit a token in "start B," and vice versa.

3.2. Closure Under Intersection

Let L_A and L_B be two languages over the same alphabet: $L_A \subseteq \alpha^*$ & $L_B \subseteq \alpha^*$. Then the intersection $L_C = L_A \cap L_B$ is:

$$L_C = \{x \in \alpha^* | x \in L_A \ \& \ x \in L_B\}.$$

Suppose we are given two Labelled Petri nets A and B. Let us first consider the case of $\mathcal{L}^\lambda$-languages. We shall construct a Labelled Petri net C such that its firing sequences correspond precisely to label sequences common to A and B. As a first step, we shall combine A and B in a way which forces them to generate the same strings. To do this, we juxtapose A and B (each with its initial marking). We add a new place π_o and, for each symbol $a \in \alpha$ (the alphabet α is common to A and B), a new place π_a. Initially, π_o has one token, all other π-places are blank.

As shown in Figure 3.4, we connect π_o as an input to each labelled $t \in \Sigma_A$, and as an output to each labelled $t \in \Sigma_B$. For each symbol $a \in \alpha$, we connect π_a as an output to each a-labelled $t \in \Sigma_A$, and as an input to each a-labelled $t \in \Sigma_B$. λ-transitions in Σ_A or Σ_B are not connected to the π-places.

This arrangement enforces a strict alternation between labelled firings in A and in B; λ-firings are not restricted. Each labelled firing in A is furthermore necessarily followed by a similarly labelled firing in B. In a sense, the π-places "remember" which symbol was last generated in A and enforce the repetition of this symbol in B before returning a token to π_o. As a result, the

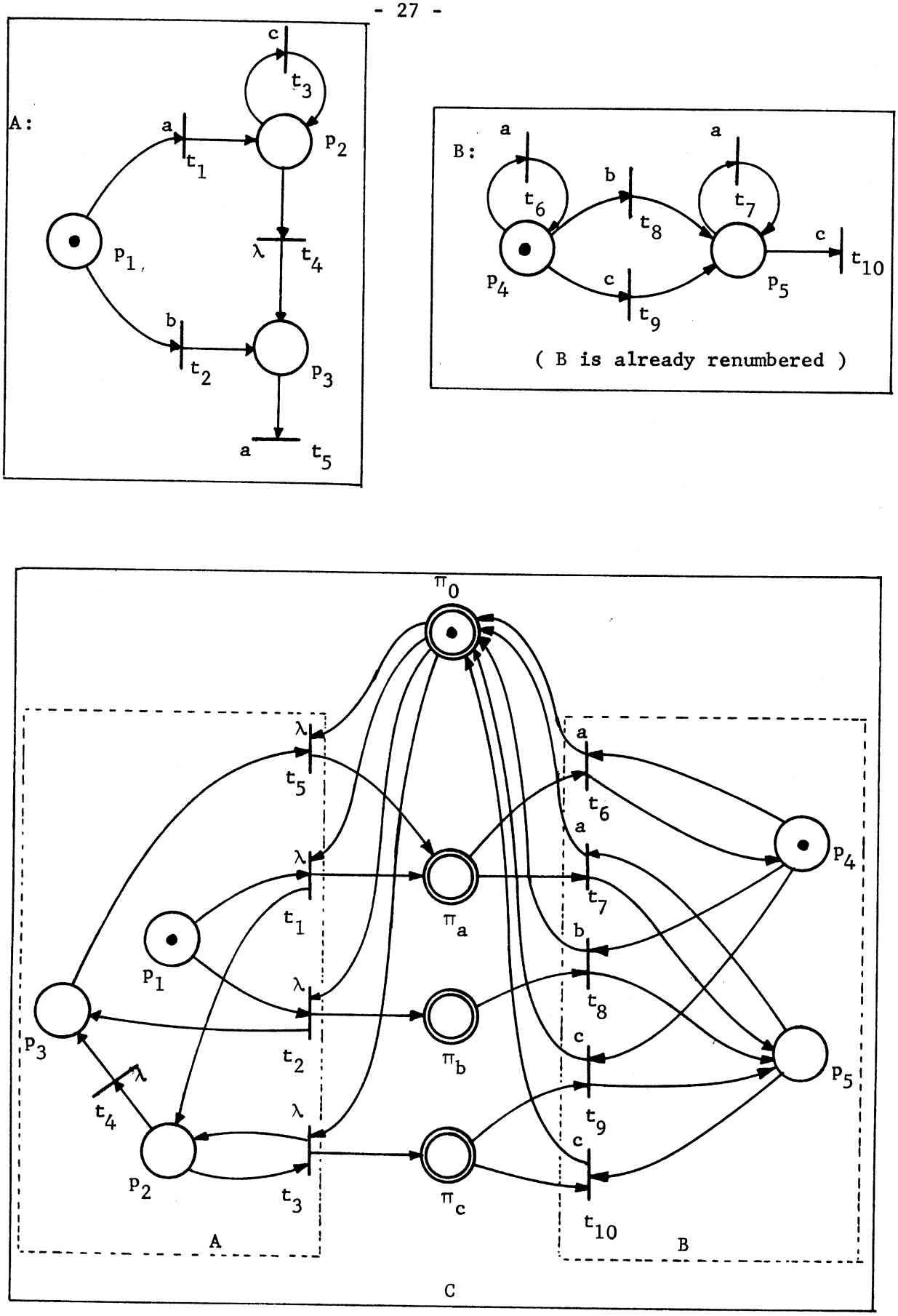

A:
a
t1
c
t3
p2
p1
λ
t4
b
t2
p3
a
t5
B:
a
t6
b
t8
a
t7
c
t10
p4
c
t9
p5
(B is already renumbered)
π0
λ
t5
a
t6
λ
t1
πa
a
t7
p4
p1
b
t8
λ
t2
πb
c
t9
p5
p3
λ
t4
λ
t3
πc
c
t10
p2
A
B
C

Figure 3.4

even-length label sequences of C are precisely those obtained by repeating twice each symbol from a label sequence that could be generated by both A and B. If we now remove the labels from all transitions in Σ_A, we will in effect erase the first symbol in each such repetition.

Our construction for the intersection of two $\mathcal{L}^\lambda$-languages consists thus of a Labelled Petri net C, as described above, where all transitions in Σ_A have become λ-transitions.

Then we have: $\mathcal{L}^\lambda(C) = \mathcal{L}^\lambda(A) \cap \mathcal{L}^\lambda(B)$. In the case of two $\mathcal{L}_0^\lambda$-languages, both nets A and B are to reach a final marking. Let the final marking of the net C, constructed as above, be the juxtaposition of the two final markings, and one token in π_o and zero tokens in the other π-places. Then it is clear that:

$$\mathcal{L}_0^\lambda(C) = \mathcal{L}_0^\lambda(A) \cap \mathcal{L}_0^\lambda(B)$$

The situation is more complicated in the case of $\mathcal{L}_0$ and $\mathcal{L}$-languages. If the original nets A and B don't have λ-transitions, the net C resulting from the previous construction will have λ-transitions, namely all the Σ_A-transitions. However, each λ-firing will be immediately followed by a labelled firing. We will show how to combine these two firings into a single labelled firing. A more general result will be proved in Theorem 4.12.

Figure 3.5 shows the portion of the Labelled Petri net C of Figure 3.4 that is connected to π_a.

We see that any a-labelled firing (t_6 or t_7) is always preceded by a firing of t_5 or t_1. There are four (2 x 2) possible combinations: t_5t_6, t_5t_7, t_1t_6, t_1t_7, each generating the symbol $\underline{a}$. Thus, we can eliminate the λ-transitions by replacing t_5, t_1, t_6, t_7 with four new a-labelled transitions which have the same effect as the combined firings t_5t_6, t_5t_7 ... ; this eliminates place π_a.

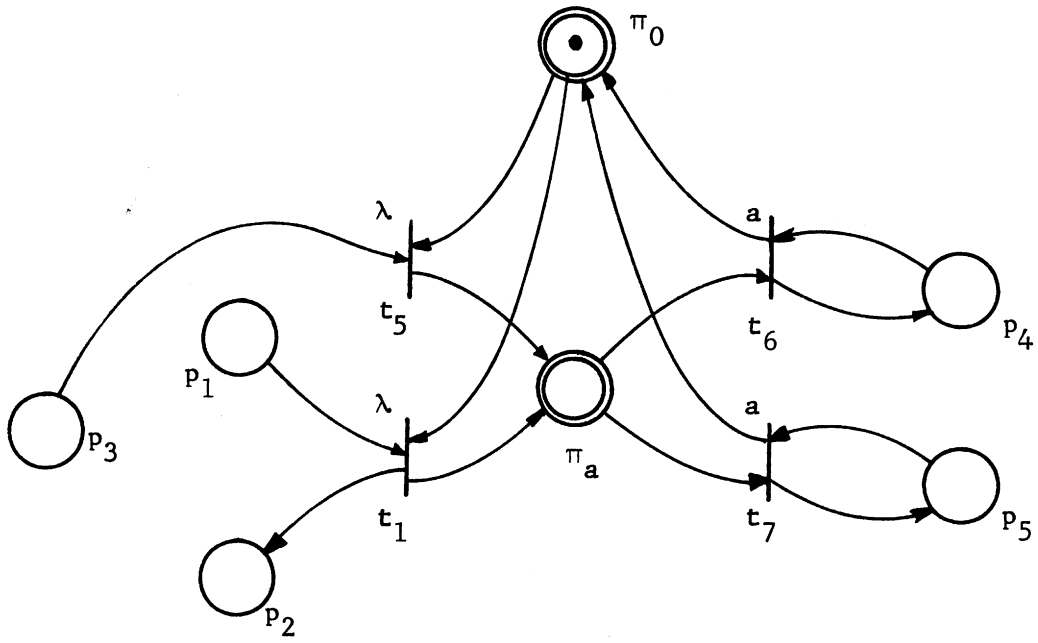


Figure 3.5

This reduction can be applied to all other π-places, except π_0 which remains as a marked self-loop on all new (combined) transitions, like a "run" place.

Figure 3.6 shows the result of eliminating place π_a from the partial net of Figure 3.5.

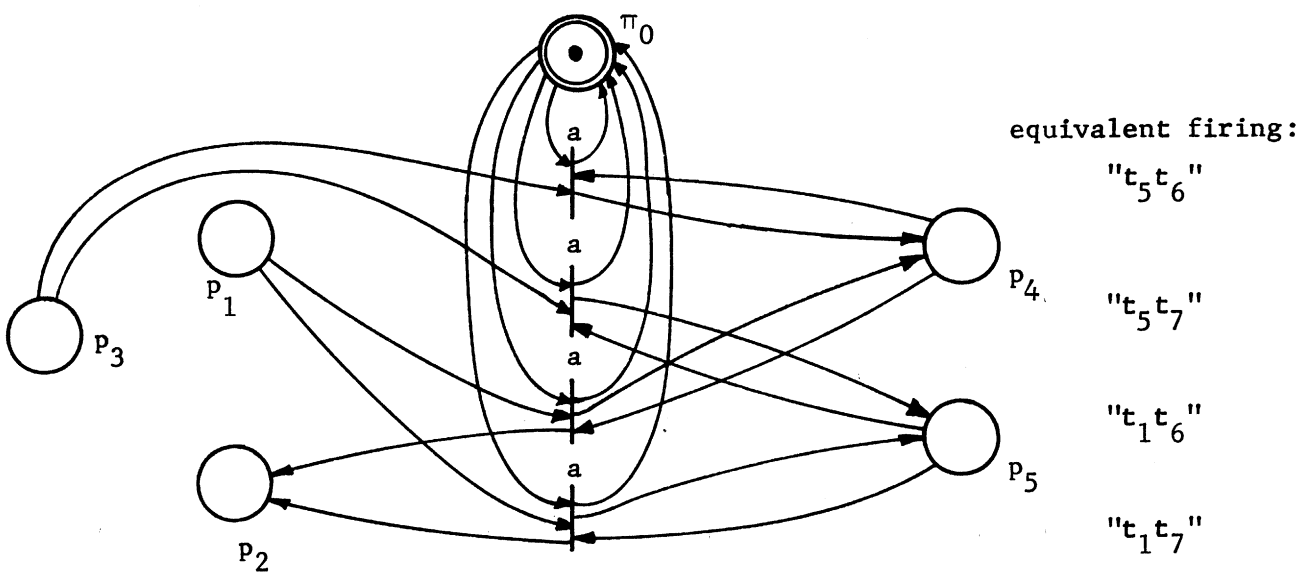


Figure 3.6

This construction shows that, if both A and B were λ-free, we can transform C into a λ-free Labelled Petri net whose $\mathcal{L}$ or $\mathcal{L}_0$-language is the intersection of the corresponding languages for A and B.

From this we may conclude:

Theorem 3.2: The families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ are closed under intersection.

3.3. Closure Under Union

To establish the closure of $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ under union it is advantageous to use Labelled Petri nets in Standard Form.

We recall that a net in Standard Form has a "start" place, which is the only place marked initially, and a standard final marking, the zero marking. Suppose we are given two nets A and B, generating L_A and L_B, respectively, as label sequences ($\mathcal{L}$, $\mathcal{L}^\lambda$) or terminal label sequences ($\mathcal{L}_0$, $\mathcal{L}_0^\lambda$). We then construct a new net C by juxtaposing the two nets A and B, and by identifying the two "start" places; the resulting net has thus one "start" place and two "run" places. We note that if A and B are λ-free, then so is C. An example is shown in Figure 3.7. The resulting net can easily be seen to satisfy the Standard Form conditions, and its label sequences are either those of A or those of B, depending on the first transition firing. The same applies to terminal sequences, since one portion of the net (corresponding to the language not simulated) retains its zero initial marking, and reaching the zero marking is thus the same as reaching the zero marking in the "active" portion of the net alone.

Thus: $\mathcal{L}(C) = \mathcal{L}(A) \cup \mathcal{L}(B)$

$\mathcal{L}_0(C) = \mathcal{L}_0(A) \cup \mathcal{L}_0(B)$

This permits us to claim:

Theorem 3.3: The language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ are closed under union.

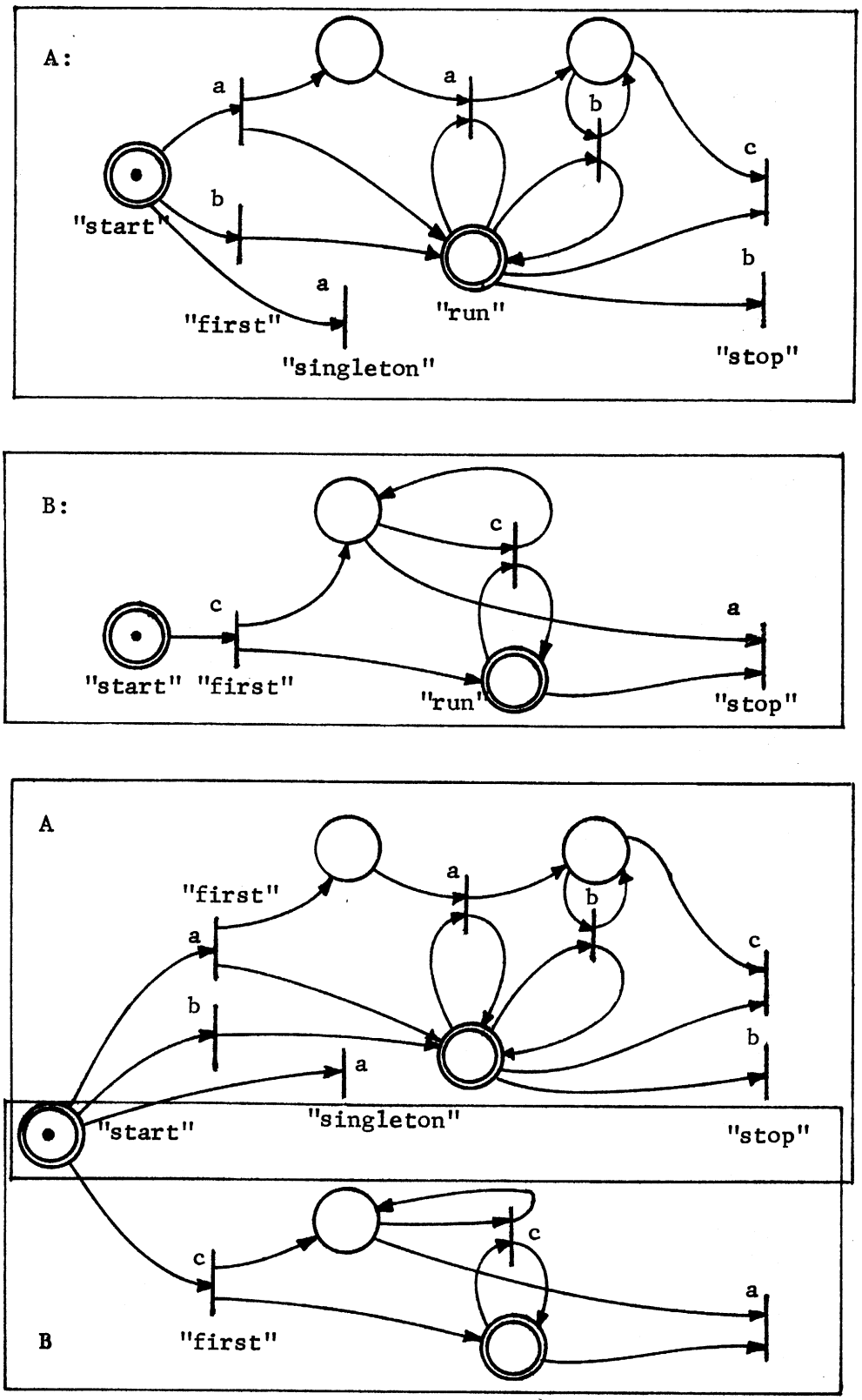

A:
a
a
b
c
b
"start"
b
a
"first"
"run"
b
"singleton"
"stop"
B:
c
c
a
"start"
"first"
"run"
"stop"
A
"first"
a
a
b
c
b
b
a
"start"
"singleton"
"stop"
c
c
a
B
"first"

Figure 3.7

3.4. Closure Under Concatenation

The concatenation of two languages consists of all strings that can be obtained by generating a string from the first language and following it by a string from the second language. Again, we will use nets in Standard Form; this time, the "stop" (and "singleton") transitions will be useful.

Suppose we are given two Labelled Petri nets A and B in Standard form. Let C be obtained by juxtaposing A and B, by removing the token in B's "start" place, and by making this place the output place of every "stop" or "singleton" transition of A, as shown in Figure 3.8 for the same example as used in the previous paragraph (Figure 3.7).

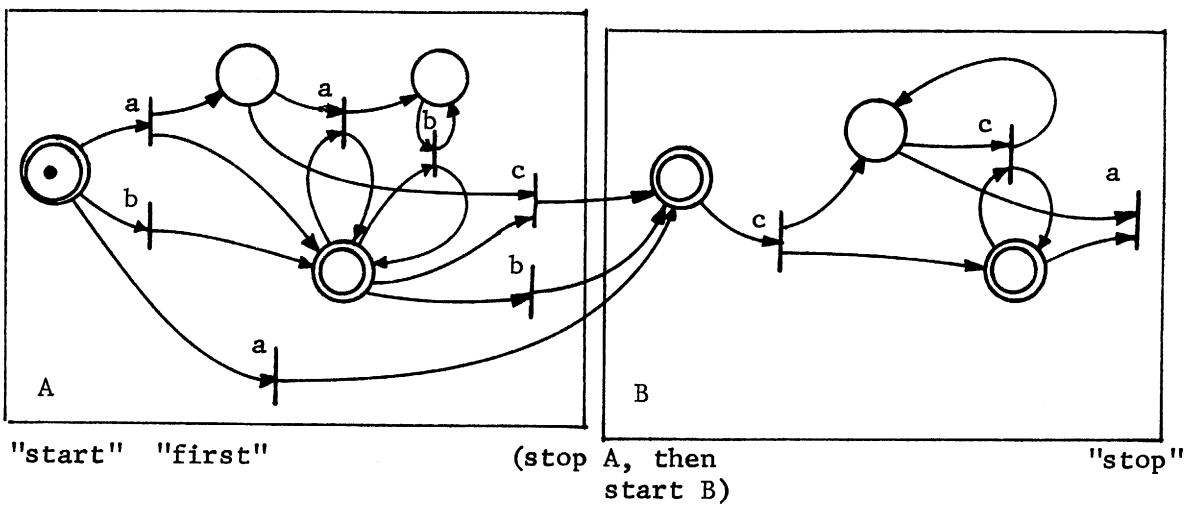


Figure 3.8

From the Standard Form Theorem it follows that, if L_A and L_B are the $\mathcal{L}_0$ (or $\mathcal{L}_0^\lambda$ or $\mathcal{L}^\lambda$)-languages of A and B, then each string in L_A can be generated in such a manner as to leave one token in "start B," and to leave every A-transition disabled. This string can then naturally be followed by a string in L_B. Conversely, every firing sequence of C is either a firing sequence of A or a firing sequence of A ending in a "stop A"-firing, followed by a firing sequence of B (since after the "stop A"-firing, no A-transition can be fired anymore, by virtue of the Standard Form conditions). As for terminal sequences, it appears that if the A-string was not terminal, i.e. that tokens were left

behind in A, then no firings in B will be able to reach the zero marking of C (A and B!). Hence, a terminal sequence of C must be a terminal sequence of A followed by a terminal sequence of B.

Thus: $\mathcal{L}_0(C) = \mathcal{L}_0(A) \cdot \mathcal{L}_0(B)$

$\mathcal{L}^\lambda(C) = \mathcal{L}^\lambda(A) \cdot \mathcal{L}^\lambda(B)$

$\mathcal{L}_0^\lambda(C) = \mathcal{L}_0^\lambda(A) \cdot \mathcal{L}_0^\lambda(B)$

As a matter of fact, the preceding reasoning also shows that:

$$\mathcal{L}(C) = (\mathcal{L}(A) - \{\lambda\}) \cdot \quad (B) \quad \cup \quad \{\lambda\}$$

To prove the closure of $\mathcal{L}$ under concatenation, it is sufficient to point out that $\mathcal{L}$ is closed under union and that:

$$\mathcal{L}(A) \cdot \mathcal{L}(B) = \mathcal{L}(C) \cup \mathcal{L}(B)$$

In fact, C can be augmented by adding an extra set of "start B" and "singleton B"-transitions which have "start A" as input, instead of "start B." To satisfy the Standard Form conditions for this construction, we must also add an extra set of "singleton A" and "stop A" transitions which have no output place, and thus become "singleton" and "stop"-transitions for the new net. Hence:

Theorem 3.4: The language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ are closed under concatenation.

behind in A, then no strings in B will be able to reach the zero marking of C (A and B!). Hence, a terminal sequence of C must be a terminal sequence of A followed by a terminal sequence of B.

Thus: $\mathcal{L}_0(C) = \mathcal{L}_0(A) \cdot \mathcal{L}_0(B)$

$\mathcal{L}_\gamma(C) = \mathcal{L}_\gamma(A) \cdot \mathcal{L}_\gamma(B)$

$\mathcal{L}_0^\gamma(C) = \mathcal{L}_0^\gamma(A) \cdot \mathcal{L}_0^\gamma(B)$

As a matter of fact, the preceding reasoning also shows that:

$$\mathcal{L}(C) = (\mathcal{L}(A) \cdot (\gamma)) \cdot (B) \cup (A)$$

To prove the closure of $\mathcal{L}$ under concatenation, it is sufficient to point out that $\mathcal{L}$ is closed under union and that:

$$\mathcal{L}(A) \cdot \mathcal{L}(B) = \mathcal{L}(C) \cup \mathcal{L}(B)$$

In fact, C can be augmented by adding an extra set of "start B" and "singleton B"-transitions which have "start A" as input, instead of "start B." To satisfy the standard form conditions for this construction, we must also add an extra set of "singleton A" and "stop A" transitions which have no output place, and thus become "singleton" and "stop"-transitions for the new net. Hence:

Theorem 3.4: The language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}_\gamma$, $\mathcal{L}_0^\gamma$ are closed under concatenation.

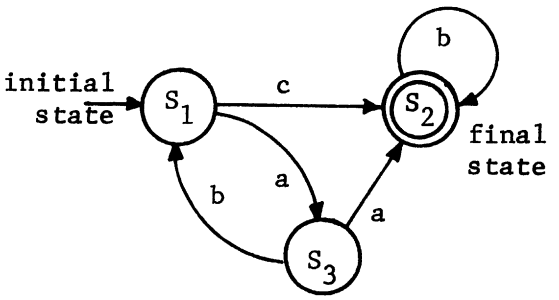
4. Regular Languages and Related Closure Properties

In this section we will show that Labelled Petri nets can generate Regular Languages by emulating Finite-State Machines.

4.1. Petri Net Generation of Regular Languages

4.1.1. $\mathcal{L}_0$ and λ-Free Regular Languages

A λ-free Regular Language can be generated by a non-deterministic Finite State Machine with n states S_1 ... S_n, an initial state S_1, a final state S_2, and a state-transition diagram where a symbol-labelled arc joins two state-labelled vertices to express a state-transition, as illustrated in Figure 4.1

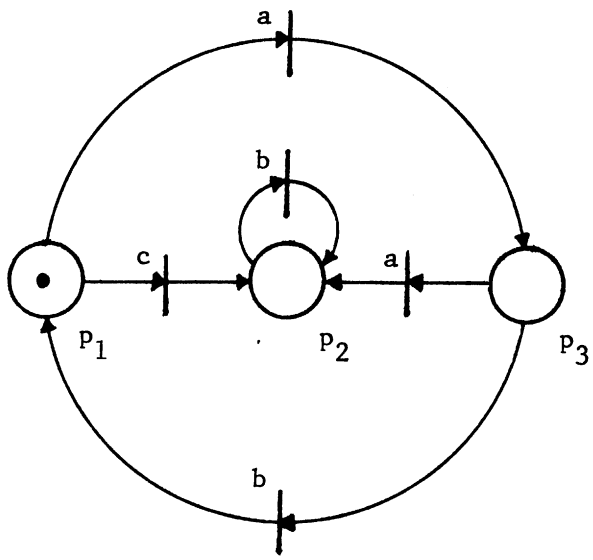


Finite-State Machine

Regular language: $(ab)^*(c + aa)b^*$

Figure 4.1

This graphical representation of a Finite-State Machine can directly be transformed into a λ-free Labelled Petri net by interpreting the state vertices as places, and by drawing a transition bar across each arc. The initial marking will consist of one token in the place corresponding to the initial state and zero tokens in all other places; the final marking corresponds to the final state in a similar way, as shown in Figure 4.2.



initial marking: ⟨1, 0, 0⟩
final marking: ⟨0, 1, 0⟩

$\mathcal{L}_0$-language: $(ab)^*(c + aa)\,b^*$
as in Figure 4.1.

Figure 4.2.

Such one-token State Machines are important building blocks for large classes of Petri nets [5 , 7].

Thus we may assert: (The extension to $\mathcal{L}_0^\lambda$ is trivial.)

Theorem 4.1: The class $\mathcal{L}_0$ contains all λ-free Regular Languages.
The class $\mathcal{L}_0^\lambda$ contains all Regular Languages.

4.1.2. Prefix Regular Languages

Definition 4.1. A Regular Language is said to be a Prefix Regular Language if, for every string in the language, all prefixes (including λ) are in the language.

In other words, in the Finite State Machine generating a Prefix Regular Language, every state is a final state. By using the same construction as in 4.1.1 it is clear that:

Theorem 4.2: The class of Prefix Regular Languages is contained in $\mathcal{L}$ and $\mathcal{L}^\lambda$.

4.2. Other Petri Net Codings for Finite State Machines

In the preceding section the states were encoded into markings in a straightforward way: For n states $S_1 \ldots S_n$ there are n places $p_1 \ldots p_n$, and state S_i is encoded by the marking consisting of one token in p_i and zero tokens in all other places. Another coding requires only two places, p_1 and p_2. If there are n states, $S_1 \ldots S_n$, we encode S_i by the marking $\langle i-1, n-i \rangle$. A transition t between S_i and S_j would be such that $F(t) = \langle i-1, n-i \rangle$ and $B(t) = \langle j-1, n-j \rangle$, so that it is enabled only in state S_i (all markings are incomparable), and its firing leads to the marking encoding S_j. It follows that:

Theorem 4.3: All (λ-free) regular languages can be generated as $(\mathcal{L}_0)\mathcal{L}_0^{\lambda}$ by two-place Labelled Petri nets.

If we prohibit Self-loops, we can still get by with four places, by encoding S_i as $\langle i-1, n-i, 0, 0 \rangle$ or $\langle 0, 0, i-1, n-i \rangle$. The transitions are now of the form $F(t) = \langle i-1, n-i, 0, 0 \rangle$ & $B(t) = \langle 0, 0, j-1, n-j \rangle$ or $F(t) = \langle 0, 0, i-1, n-i \rangle$ & $B(t) = \langle j-1, n-j, 0, 0 \rangle$. If λ-transitions are not allowed, we may have to encode some states by both markings indicated above, since tokens are always moved from one place-pair to the other. This permits us to announce:

Theorem 4.4: Four-dimensional Vector Addition Systems can simulate all Regular Languages, by way of 4-place Labelled Self-loop-free Petri nets.

4.3. Clean Standard Forms

A Labelled Petri net in Standard Form has the property that every string in the language can be generated by a firing sequence whose last firing is that of a "stop" transition, which leaves every transition disabled. If we are considering terminal strings $(\mathcal{L}_0, \mathcal{L}_0^{\lambda})$, this last firing leaves the net at the zero marking.

If this net is used as a component in a larger net, as in the constructions of the previous chapter, the occurrence of a "stop" firing is used as a signal that a certain string is complete, but usually it does not guarantee that the string is terminal or that the zero marking has been reached; a non-terminal

firing sequence may fire a "stop" transition and leave the net in an a priori indeterminate marking. This is a serious liability in a case where one wishes to re-utilize this portion of the Petri net, as one would in attempting to construct an iterative closure, such as Kleene star. This is where the notion "clean" comes in.

Definition 4.2: A Labelled Petri net in Standard Form is said to be clean iff every firing sequence which fires a "stop" transition leaves the net at the zero marking.

A Petri net language (in $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$) is said to be clean iff it can be generated by a clean net.

In a clean Labelled Petri net it is not only the case that terminal strings may always end with a "stop"*-firing, but that, conversely, every string which ends in a "stop"*-firing is terminal. In particular, the $\mathcal{L}$ (or $\mathcal{L}^\lambda$)-language of a clean net is also its $\mathcal{L}_0$ (or $\mathcal{L}_0^\lambda$)-language, up to λ in the case of $\mathcal{L}$.

We shall now show that a clean Petri net can generate only Regular Languages, and that Regular Languages can always be generated by clean nets.

The second fact can be proved trivially: For the constructions shown in the previous paragraphs (4.1 and 4.2), corresponding clean Standard Forms can easily be constructed: every transition whose firing would lead to a final marking (in the case of $\mathcal{L}$ and $\mathcal{L}^\lambda$ all reachable markings -- a finite number -- are considered final) gives rise to a "stop" transition which removes all the tokens in the present state. This works because every transition is firable at only one marking; no other covering markings are reachable, since all markings have the same number of tokens (one in 4.1; n-1 in 4.2). For example, a clean Standard Form for the net of Figure 4.2 is shown in Figure 4.3, for the $\mathcal{L}_0$-language of the net.

---

*For the purpose of "clean"-ness, "singleton" transitions are considered as "stop" transitions, since "singleton" firings are always "clean."

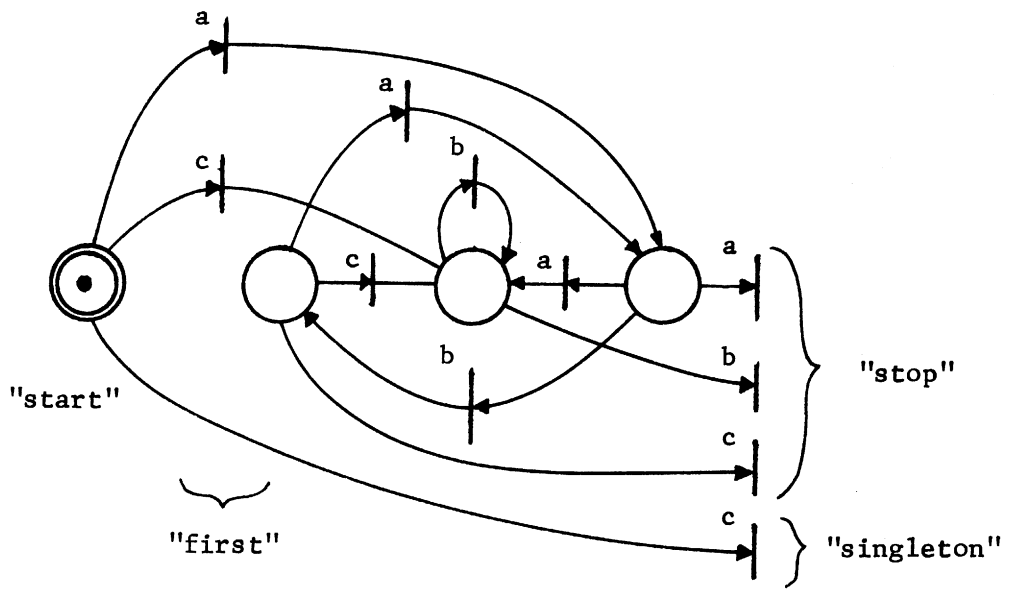


Figure 4.3

It is also easy to show that clean $\mathcal{L}$ or $\mathcal{L}^{\lambda}$-languages must be regular. Indeed, some "stop"-transition must be firable at every reachable marking, but only if the resulting marking is zero, i.e. there must be precisely one "stop"-transition for each reachable marking. Since the number of transitions in a Petri net is finite, by definition, there can only be a finite number of reachable markings, i.e. the net behaves like a Finite State Machine.

For $\mathcal{L}_0$ or $\mathcal{L}_0^{\lambda}$-languages, the situation is not so easy. It may be possible to reach arbitrarily large markings in a clean net, as shown in Figure 4.4.

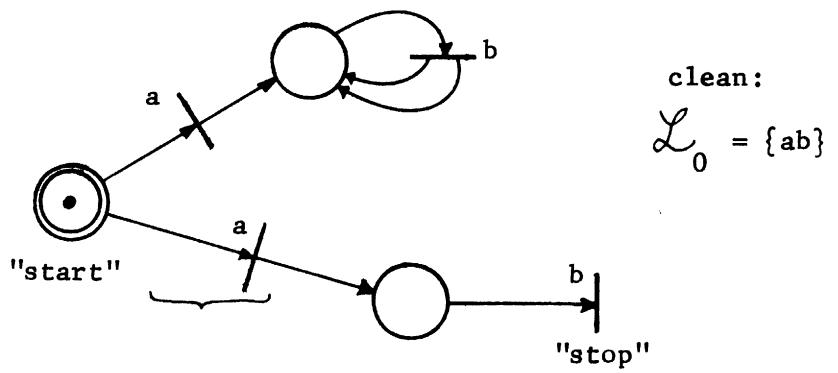


Figure 4.4

But we can show that the final marking (zero) is reachable only from a finite subset of the reachable markings, so that the number of states required to simulate only the terminal firing sequences is finite, which implies a Regular Language.

Assume there exist arbitrarily large reachable markings from which the zero marking can be reached. Then at least two such markings must be comparable and unequal: There exist M_1 and M_2 such that:*

$$M_0[\sigma_1\rangle M_1[\tau_1\rangle 0$$
$$M_0[\sigma_2\rangle M_2[\tau_2\rangle 0 \qquad M_2 \geq M_1 \;\&\; M_2 \neq M_1$$

Since the net is in standard form we know that both τ_1 and τ_2 end in a "stop" firing.

But $M_2 \geq M_1$ implies that τ_1 is also firable at M_2, thus:

$$M_0[\sigma_2\rangle M_2[\tau_1\rangle M_3$$

Now, $M_2 \geq M_1$ & $M_2 \neq M_1 \Rightarrow M_3 \neq 0$, and thus the net cannot be clean.

This permits us to affirm:

Theorem 4.5: The class of clean $\mathcal{L}_0^\lambda$-languages is precisely the class of all Regular languages; the class of all clean $\mathcal{L}_0$-languages is precisely the class of all λ-free Regular Languages, and the class of all clean $\mathcal{L}$ or $\mathcal{L}^\lambda$-languages is precisely the class of all Prefix Regular Languages.

4.4. Closure Under Clean Substitution

4.4.1. Clean Substitution

Substitution is an operation in which all occurrences of a given symbol a in a string from language L_1 are replaced by some string from language L_2. The result of this operation is a new language L_3. We write this as:

$$L_3 = [a \rightarrow L_2 \text{ in } L_1]$$

Thus, for example:

---

*There can be no infinite number of incomparable markings; see Hack [7].

$$[a \to (b + a^*) \quad \text{in} \quad acab^*] = (b + a^*)c(b + a^*)b^*$$

We see that if both L_1 and L_2 are regular, then L_3 is regular; this is expressed by saying that Regular Languages are closed under substitution. Sometimes there are restrictions on the language L_2. In that case we say that a family $\mathcal{F}_1$ is closed under $\mathcal{F}_2$-substitution: if $L_1 \in \mathcal{F}_1$ and $L_2 \in \mathcal{F}_2$, then $[a \to L_2 \text{ in } L_1] \in \mathcal{F}_1$.

Usually, the operation of substitution is defined in a more general way:

Definition 4.3: An $\mathcal{F}$-substitution is a mapping

$$S: \mathcal{A} \to \mathcal{F} \text{ (where } \mathcal{F} \subseteq 2^{\mathcal{B}^*})$$

which to every symbol of an alphabet $\mathcal{A}$ assigns a language over an alphabet $\mathcal{B}$ from a given family $\mathcal{F}$; this mapping is extended to strings in $\mathcal{A}^*$ and languages in $2^{\mathcal{A}^*}$.

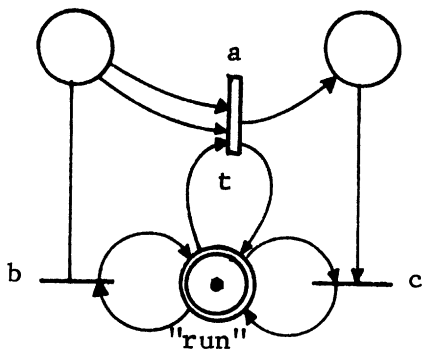
The one symbol case mentioned previously corresponds to the mapping:

$$S(a) = L_2$$

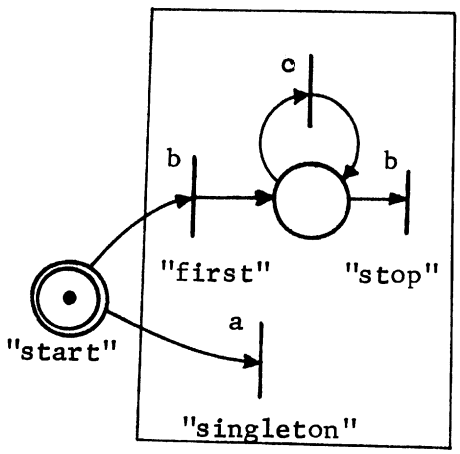
$$S(a') = \{a'\}, \quad \forall a' \in \mathcal{A} : a' \neq a$$

In the sequel, we will prove that Petri net languages are closed under clean substitutions; this means that we restrict $\mathcal{F}$ to be the family of clean Petri net languages of the same family as L_1. Let us first examine the case where a single symbol a is mapped into a clean language L_2; all others are unchanged.

Let A be a Labelled Petri net generating L_1 (as $\mathcal{L}_0$ or $\mathcal{L}$), and assume that A has a "run" place which self-loops on every transition. Let B be a clean Standard Form Petri net generating L_2, and assume, for the moment, that A contains a single transition t labelled a, as shown in Figure 4.5. We will replace transition t by the Petri net B, where the "start" place has been replaced by the set of input places to t, and where each "stop" or "singleton" transition is connected to all output places of t in the same way as t was connected. In other words, a firing sequence of B which ends in a "stop" or "singleton" firing has exactly the same effect in A as a firing of t. Moreover, since the subnet B swallows A's "run" token, no other transitions in A can fire while B is substituting a string for the firing of t. This construction is shown in Figure 4.6 for the example of Figure 4.5.



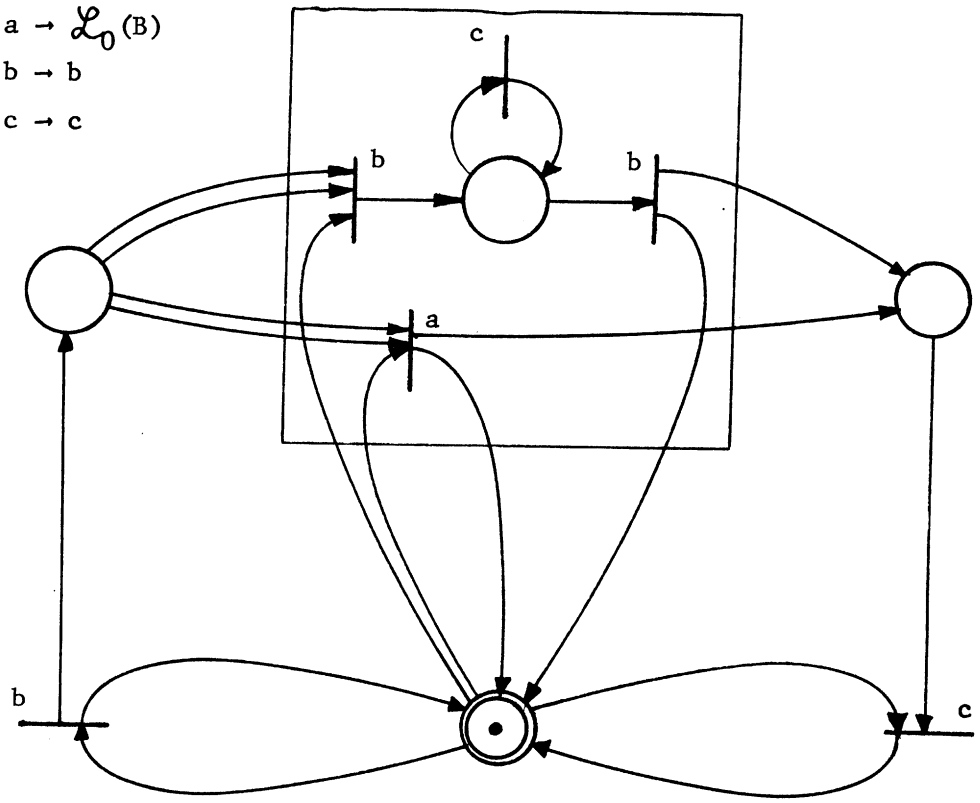
Petri net A



Petri net B

Figure 4.5

S: $\begin{cases} a \rightarrow \mathcal{L}_0(B) \\ b \rightarrow b \\ c \rightarrow c \end{cases}$



Petri net C

$$\mathcal{L}_0(C) = S(\mathcal{L}_0(A))$$

Figure 4.6

Since the net B is clean, its marking before and after generating a string is zero. In other words, at all times it is the case that either the "run" place is empty or the places of B are empty. B can be started repeatedly, and each time it will generate a string in L_2, because each time it starts with the proper initial marking. If B were not clean, this could not be guaranteed.

We also note that if L_1 and L_2 are in $\mathcal{L}$, the empty string (which is in $\mathcal{L}(B)$) cannot be substituted for a firing of t; this restriction has to be borne in mind.

Finally, if there are several transitions labelled "a" in A, we can, of course, replace each one by a copy of B. In practice, it would be sufficient to have several copies of B's "first," "singleton," and "stop" transitions, and extra places to remember which set of "stop" transitions is to be paired with a given set of "first" transitions.

The same construction, repeated for every symbol, yields general substitution. Hence:

Theorem 4.6: The families $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ are closed under clean substitution, and $\mathcal{L}$ is closed under λ-free clean substitution.

(where λ-free substitution means that we may substitute any legal string except λ).

4.4.2. Regular Substitution

A direct consequence of Theorems 4.5 and 4.6 is:

Theorem 4.7: $\mathcal{L}_0$ is closed under λ-free Regular Substitution.
$\mathcal{L}$ is closed under λ-free Prefix Regular Substitution.
$\mathcal{L}^\lambda$ is closed under Prefix Regular Substitution.
$\mathcal{L}_0^\lambda$ is closed under Regular Substitution.

We will show later (9.3.2) that Petri net languages are not closed under unrestricted substitution.

4.4.3. Closure Under Homomorphism

The homomorphic image of a Language L_1 is the result of replacing each symbol a in a string of L_1 by a string h(a), where h: $\mathcal{A} \rightarrow \mathcal{B}^*$ is a function from the alphabet of L_1 into the set of strings from another (or the same) alphabet $\mathcal{B}$. It is a special case of regular substitution. Thus:

Theorem 4.8: $\mathcal{L}_0$ is closed under λ-free homomorphism.

$\mathcal{L}_0^\lambda$ is closed under unrestricted homomorphism.

In the case of $\mathcal{L}$ and $\mathcal{L}^\lambda$, only single symbols can be substituted, since {abc} is not a Prefix Language; we could only substitute {λ, a, ab, abc} or {a, ab, abc}.

4.5. Inverse Homomorphism for $\mathcal{L}$ and $\mathcal{L}_0$

Given a language L_1 over alphabet $\mathcal{A}$ and a homomorphism h: $\mathcal{B} \rightarrow \mathcal{A}^*$, we want to generate the language $L_2 = h^{-1}(L_1)$ over alphabet $\mathcal{B}$, which consists of all those strings x such that $h(x) \in L_1$. (The functions h and h^{-1} are freely extended from alphabets to strings to sets of strings). Warning: $h(L_2) \subseteq L_1$ but not necessarily $h(L_2) = L_1$; some strings in L_1 may not have a homomorphic inverse!

In a λ-free Petri net generating L_1, this can be accomplished by removing all transitions and replacing them with new transitions labelled with symbols $b \in \mathcal{B}$ and having the same effect as a firing sequence of the old transitions which would have spelled out the string h(b). For a given string h(b) there may be several possible firing sequences of the old transitions, and correspondingly there will be several new transitions labelled b; in λ-free nets, this number is always finite. The "effect" of a firing sequence can be described by its hurdle, which is the smallest marking required to completely fire the sequence, and its marking change. The same effect will be produced by a new transition t such that F(t) equals the hurdle of the sequence, and B(t) - F(t) equals the required marking change. In particular, if for some symbol b we have $h(b) = \lambda$, then the corresponding new transition, labelled b, would have no input and no output place, and could of course fire anytime without changing the marking. In practice, it would be made to self-loop on some "run" place; in a Standard Form Petri net it would also appear as "first," "singleton" and "stop" transitions.

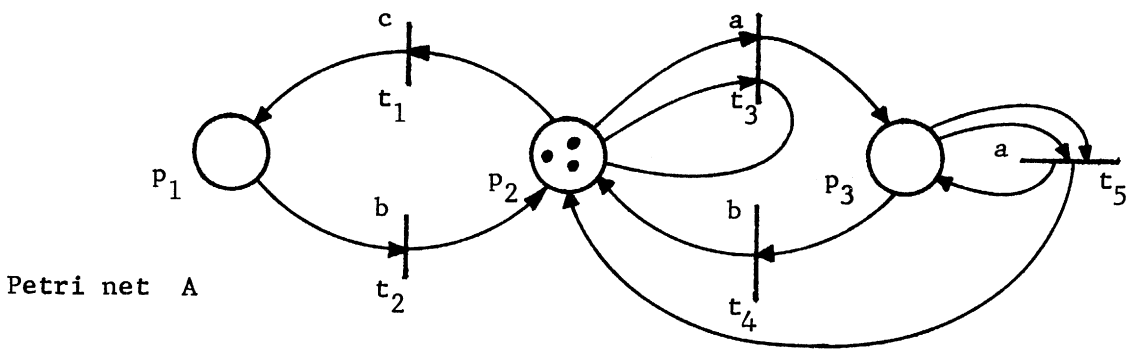
The construction works for both the $\mathcal{L}$ and the $\mathcal{L}_0$-language of a net.

An example is shown in Figure 4.7. The homomorphism is as follows :

(we assume $\mathcal{A} = \{a,b,c\}$ and $\mathcal{B} = \{a',b',c'\}$) $\quad h: \mathcal{B} \to \mathcal{A}^*$

$h(a') = bc \qquad h(b') = \lambda \qquad h(c') = aa$

The firing sequences in the old net and the corresponding new transitions are listed in the table of Figure 4.8. The resulting new net (not in standard form) is shown in Figure 4.9.

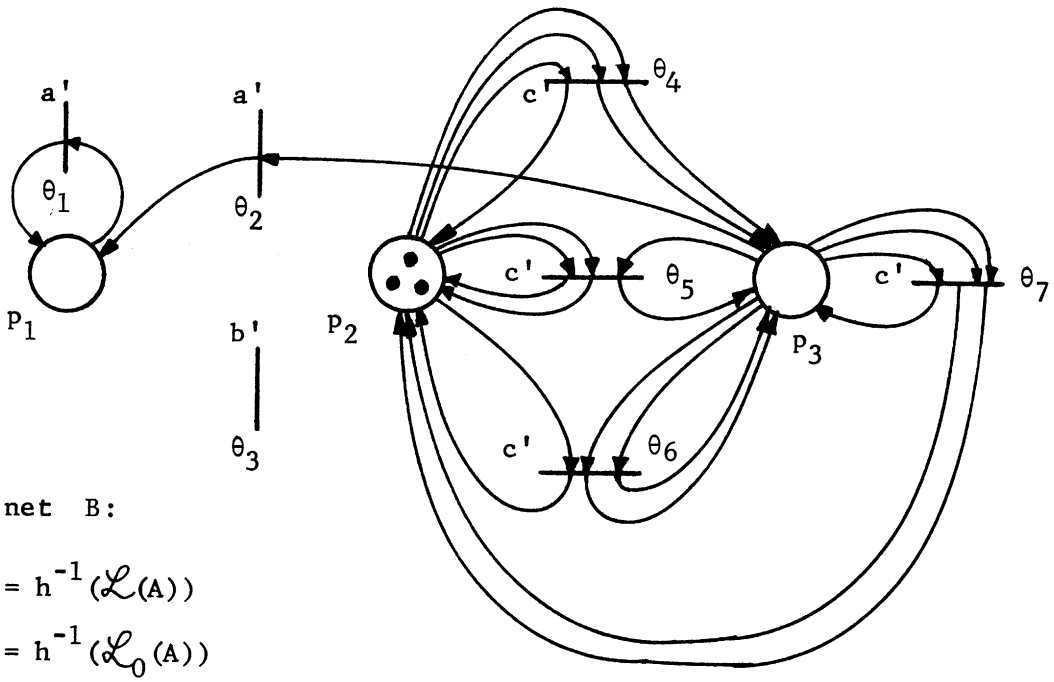


Petri net A

Figure 4.7

string to be replaced	old firing sequence	new transition	corresponding symbol
$h(a') = bc$	$t_2\ t_1$	θ_1	a'
	$t_4\ t_1$	θ_2	
$h(b') = \lambda$		θ_3	b'
$h(c') = aa$	$t_3\ t_3$	θ_4	c'
	$t_3\ t_5$	θ_5	
	$t_5\ t_3$	θ_6	
	$t_5\ t_5$	θ_7	

Figure 4.8



Petri net B:

$\mathcal{L}(B) = h^{-1}(\mathcal{L}(A))$

$\mathcal{L}_0(B) = h^{-1}(\mathcal{L}_0(A))$

(If the same final marking is specified for A and B.)

Figure 4.9

It should be clear that to every firing sequence in B there corresponds a firing sequence in A, and that no other firing sequences are possible in B. We notice that in this example, A has many firing sequences which are outside of the homomorphic image of $\mathcal{L}(B)$. (An example of such a sequence is ccbb).

Hence:

Theorem 4.9: $\mathcal{L}_0$ and $\mathcal{L}$ are closed under unrestricted inverse homomorphism.

In fact, we can strengthen this result to include inverse finite substitution, where a given new symbol $b \in \mathcal{B}$ can replace one of several strings $x_1, \ldots, x_n \in \mathcal{A}^*$. A first step would use inverse homomorphism to replace x_i by $b_i' \in \mathcal{B}'$, and then rename the b_i' to b using λ-free homomorphism:

Corollary: $\mathcal{L}_0$ and $\mathcal{L}$ are closed under inverse finite substitution.

For the closure under inverse homomorphism of $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$, see section 4.7 and Theorem 4.11.

4.6. Closure Under FST-Mapping

A Finite-State Transducer is a Finite State Machine as described in 4.1, with the additional feature that each state-transition carries two symbols: an input symbol and an output symbol. The interpretation is that the FST is started in its initial state and "reads" an input string, i.e. the state-transition input symbols spell out the input string. The FST produces an output string by spelling out the output symbols, provided the machine ends up in a final state. We may define a Prefix FST to have every state be final. To each input string the FST associates a set of output strings (empty if it doesn't accept the input string, singleton if it is accepting and deterministic). An FST can thus map a language L_1 into a language L_2: Those strings that can be obtained as output strings when the FST accepts an input string from L_1.

It is easy to show how to generate the FST-image of a given Petri net Language: We use a Petri net coding of the FST as indicated in 4.1 (or 4.2) and use the construction for the intersection of Petri net Languages described in 3.2 using the input labels of the FST transitions. In the end, however, we relabel the transitions with their FST output labels: The resulting Labelled Petri net clearly generates the FST-image of the original Petri net Language, given the proper restrictions: λ-free for $\mathcal{L}_0$, Prefix for $\mathcal{L}$ and $\mathcal{L}^\lambda$.

Thus:

Theorem 4.10: The Petri net languages are closed under Finite-State Transducer mappings, provided the FST is λ-free for $\mathcal{L}_0$, and Prefix FST for $\mathcal{L}$, $\mathcal{L}^\lambda$.

In conjunction with Theorem 4.8 this proves:

Corollary: $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$ are closed under GSM-mappings (λ-free for $\mathcal{L}_0$)

(A Generalized State Machine is like an FST, except that the output labels can be strings, not just single symbols).

4.7. Inverse Regular Substitution and $\mathcal{L}_0^\lambda$, $\mathcal{L}^\lambda$.

Recall the definition of substitution given in 4.4.1, in the case of regular substitution: ($\mathcal{R}$ = Regular Languages)

$$S: \mathcal{A} \to \mathcal{R}$$

Each symbol a can be replaced by some string from the associated regular language S(a); the image of a string x is the Regular Language S(x) which is the concatenation of the Regular Languages assigned to the symbols occurring in x.

The inverse image of a language L_1 under this substitution is the language L_2 defined by:

$$L_2 = \{x \mid S(x) \cap L_1 \neq \emptyset\} = S^{-1}(L_1)$$

It is the largest language whose image under S is contained in L_1.

Let us first note that Inverse Regular Substitution includes inverse homomorphism as a special case. In the case of $\mathcal{L}_0^\lambda$, however, closure under inverse homomorphism follows from closure under regular substitution, homomorphism, and union and intersection with regular languages, by a theorem of Salomaa [18]. The closure under inverse homomorphism for $\mathcal{L}_0^\lambda$ and $\mathcal{L}^\lambda$ cannot be established by the same method as used for λ-free Petri nets in 4.5, because a given label sequence may correspond to an infinite set of firing sequences. On the other hand, the availability of λ-transitions can be used to prove a stronger result.

As it turns out, Salomaa's proof can be extended to cover Inverse Regular Substitution. Let $L_2 = S^{-1}(L_1)$, where L_2 is over the alphabet $\mathcal{B}$ and L_1 over the alphabet $\mathcal{A}$; by renaming we can insure $\mathcal{A} \cap \mathcal{B} = \emptyset$. Substitution S assigns to each $a \in \mathcal{B}$ a regular language $S(a) \subseteq \mathcal{A}^*$. Let L_3 be defined over $\mathcal{A} \cup \mathcal{B}$ as:

$$L_3 = \left(\bigcup_{a \in \mathcal{B}} (S(a) \cdot a) \right)^*$$

In other words, L_3 consists of a succession of strings which are S-images of symbols from $\mathcal{B}$, followed by the corresponding source symbol.

Let $L_4 = L_1 \| \mathcal{B}^*$, in other words, L_1 with the symbols of $\mathcal{B}$ sprinkled through the strings in all possible ways.

Let $L_5 = L_3 \cap L_4$. Since L_4 has no constraints at all on symbols from $\mathcal{B}$, L_5 selects from L_3 precisely those strings whose $\mathcal{A}$-symbols spell out strings in L_1. To get L_2, all we have to do is erase from L_5 all $\mathcal{A}$-symbols, only leaving the $\mathcal{B}$-symbols which record which symbols in the L_2-string have been substituted by the $\mathcal{A}^*$-string which matches the L_1-string. This is done by using a homomorphism $h: \mathcal{A} \cup \mathcal{B} \rightarrow \mathcal{B} \cup \{\lambda\}$, defined by:

$$a \in \mathcal{A} \Rightarrow h(a) = \lambda$$

$$a \in \mathcal{B} \Rightarrow h(a) = a$$

Then $$L_2 = h(L_5) = S^{-1}(L_1)$$

Notice that, in a Petri net construction, this homomorphism simply amounts to removing the labels from all $\mathcal{A}$-labelled transitions.

Since $\mathcal{L}_0^\lambda$ is closed under all the operations carried out above, we conclude that $\mathcal{L}_0^\lambda$ is closed under inverse regular substitution (and inverse homomorphism).

Although the proof above does not apply to $\mathcal{L}^\lambda$, because the intermediate languages are not necessarily prefix languages, it can be seen that the $\mathcal{L}^\lambda$-language of the resulting Petri net is indeed the result of the inverse regular substitution. Hence:

Theorem 4.11: $\mathcal{L}_0^\lambda$ and $\mathcal{L}^\lambda$ are closed under inverse regular substitution (and inverse homomorphism).

4.8. Other Closures: Regular Control, Limited Erasing, Promptness.

The various constructions we have used so far demonstrate the flexibility of the Petri net model, especially when λ-transitions are allowed.

The constructions for the various mappings (such as substitution, and its inverse) can be extended to several arguments, and in a general way it can be said that any†) mapping $f(L_1, L_2, \ldots)$ which is defined by means of finite-state machines (regular control) transforms $\mathcal{L}_0^\lambda$ languages into $\mathcal{L}_0^\lambda$ languages, and that many constructions can be carried out without λ-transitions, or be made to respect the prefix property, so as to extend closure to $\mathcal{L}$, $\mathcal{L}_0$ and/or $\mathcal{L}^\lambda$. For example, it is easy in this way to prove closure under "perfect shuffle," where a string from one language is interleaved, symbol by symbol, with a string from another.

†) But beware of the implications of non-closure under Kleene Star.

Another important application concerns the elimination of λ-transitions.

Definition 4.4: A Labelled Petri net is said to be k-prompt iff the number of consecutive λ-firings between labelled firings is bounded by k.

This definition closely corresponds to that of Patil [16], who wishes to study systems in which only a bounded number of "internal" firings may occur between interactions with the environment, so that the response to a stimulus occurs "promptly," i.e. cannot be delayed arbitrarily long.

It turns out that the languages generated by prompt nets can also be generated by λ-free nets, and are thus $\mathcal{L}_0$ or $\mathcal{L}$.

Definition 4.5: A K-limited erasing is a homomorphism which either maps into λ (erases) or maps into itself each symbol, but never erases more than K consecutive symbols.

A K-limited erasing can be obtained by first performing inverse homomorphism, then λ-free homomorphism (renaming). For example, consider a 3-limited erasing of the symbol c in a language L over the alphabet {a, b, c}. A first step would be to effectuate the following inverse homomorphism h^{-1}:

$$\begin{aligned} h:\ & a_0 \to a & & b_0 \to b \\ & a_1 \to ac & & b_1 \to bc \\ & a_2 \to acc & & b_2 \to bcc \\ & a_3 \to accc & & b_3 \to bccc \end{aligned}$$

This works if no string in L starts with c, cc, or ccc; otherwise, we could use $a_{i,j} \to c^i a c^j$ for all $i,j \in \{0,1,2,3\}$, for example.

Next we perform the renaming homomorphism g:

$$g(a_i) = a; \qquad g(b_i) = b \qquad (i = 0,1,2,3)$$

The result of the 3-limited erasing of c in L is $g(h^{-1}(L))$, assuming that L contains no strings with more than three consecutive c's, which are not in the domain of the erasing (or the inverse homomorphism h^{-1}, for that matter).

By virtue of Theorems 4.7, 4.9 and 4.11, we can thus state:

Theorem 4.12: $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$ are closed under K-limited erasing.

Theorem 4.12 can now be used to show that λ-transitions can be eliminated from prompt nets: If K is the bound on λ-firings implied by promptness,† we can assign an unused label to all λ-transitions to get an $\mathcal{L}_0$-language (or $\mathcal{L}_0^\lambda$), and then use K-limited erasing on this language to show that the original prompt $\mathcal{L}_0^\lambda$ ($\mathcal{L}^\lambda$) language is in fact $\mathcal{L}_0$ ($\mathcal{L}$). The Petri net construction associated with the mappings effectively removes the λ-transitions, at the price of usually a much larger number of new labelled transitions. This, by the way, indicates that many $\mathcal{L}_0$-languages can be more economically generated by nets having λ-transitions.

Finally, let us mention that Petri net languages are not closed under unlimited (basically, unclean) substitution, nor are they closed under Kleene star (iteration). This will be proved later (section 9.3.2). Closure under complement is considered unlikely, because it would imply the undecidability of the Reachability Problem; but this question is still open.

---

† Warning: There exist nets that are prompt, but not k-prompt for any k, in the sense that, at any reachable marking, only boundedly many λ-firings may occur before a labelled firing, but the bound may depend on the marking reached previously, so there is no a-priori bound. An example is shown in figure 4.10.

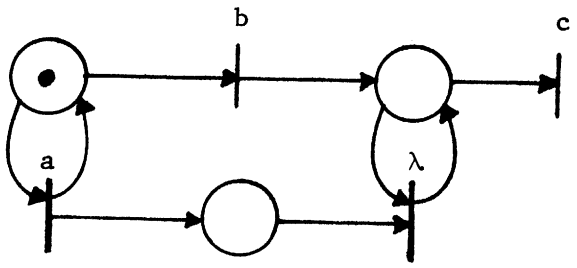


Figure 4.10

Theorem 4.12: $\mathcal{X}$, $\mathcal{X}_0$, $\mathcal{X}_\gamma$ and $\mathcal{X}_\gamma^0$ are closed under X-limited erasing.

Theorem 4.12 can now be used to show that γ-transitions can be eliminated from prompt nets: If K is the bound on γ-firings implied by promptness[†], we can assign an unused label to all γ-transitions to get an $\mathcal{X}_0^\gamma$-language (or $\mathcal{X}_\gamma^0$), and then use X-limited erasing on this language to show that the original prompt $\mathcal{X}_\gamma^0$ ($\mathcal{X}_\gamma$) language is in fact $\mathcal{X}_0$ ($\mathcal{X}$). The Petri net construction associated with the mapping effectively removes the γ-transitions, at the price of usually a much larger number of new labelled transitions. This, by the way, indicates that many $\mathcal{X}_0$-languages can be more economically generated by nets having γ-transitions.

Finally, let us mention that Petri net languages are not closed under unlimited (basically, useless) substitution, nor are they closed under Kleene star (iteration). This will be proved later (section 9.3.3). Closure under complement is considered unlikely, because it would imply the undecidability of the Reachability Problem; but this question is still open.

[†] Marking: There exist nets that are prompt, but not k-prompt for any k, in the sense that, at any reachable marking, only boundedly many γ-firings may occur before a labelled firing, but the bound may depend on the marking reached previously, so there is no γ-priori bound. An example is shown in Figure 4.10.

Figure 4.10

5. A Characterization of $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$

A characterization of a family of languages usually involves defining the family as the smallest set of languages containing a given set of languages (basis) and closed under various operations (induction), thus providing at the same time a constructive approach. Thus, for example, we can characterize regular languages as being the smallest class containing the languages {a}, where a is a symbol from an alphabet, and closed under concatenation, union, and Kleene star (iteration). We shall see that Petri net languages can be defined by their closure properties, using as a basis the regular languages and the simple parenthesis languages.

5.1. Free Petri Net Languages

The various Petri net languages are obtained by applying a labelling function $\Lambda: \Sigma \to \mathcal{A}$ to the firing sequences of a net. This labelling function can also be viewed as a homomorphism which renames the transitions from a net whose transitions are all distinctly labelled. Thus we define:

Definition 5.1: A free-labelled Petri net is a labelled Petri net where all transitions are labelled distinctly. The non-terminal and the terminal firing sequences of a free-labelled Petri net define the families $\mathcal{L}^f \subseteq \mathcal{L}$ and $\mathcal{L}_0^f \subseteq \mathcal{L}_0$ of free Petri net languages respectively.

Let a homomorphism whose range contains only one-symbol strings (and λ, if unrestricted) be called a renaming. Then it follows from the closure properties under finite substitution and homomorphism (4.4.3) that:

Theorem 5.1: $\mathcal{L}_0$ is the closure of $\mathcal{L}_0^f$ under λ-free homomorphism
$\mathcal{L}_0^\lambda$ is the closure of $\mathcal{L}_0^f$ under unrestricted homomorphism
$\mathcal{L}$ is the closure of $\mathcal{L}^f$ under λ-free renaming
$\mathcal{L}^\lambda$ is the closure of $\mathcal{L}^f$ under unrestricted renaming.

It is easy to see that $\mathcal{L}_0^f$ is properly contained in $\mathcal{L}_0$, and that $\mathcal{L}_0^f$ is not closed under union, since {a, aaa} $\notin \mathcal{L}_0^f$, for example, but {a} and {aaa} are in $\mathcal{L}_0^f$.

On the other hand, it can be seen that the constructions for closure under intersection (3.2) and inverse homomorphism (4.5), when used with free-labelled nets, produce new free-labelled nets. Thus:

Theorem 5.2: The free Petri net language families $\mathcal{L}^f$ and $\mathcal{L}_0^f$ are closed under intersection and inverse homomorphism.

5.2. Simple Parenthesis Languages

A simple parenthesis language is a context-free language over a 2-symbol alphabet {(,)} whose strings are well-nested parenthesis strings. We will use the symbols + and - instead of (and) because parentheses are more useful in the metalanguage, and we will also consider prefixes of well-nested strings.

Definition 5.2: The complete simple parenthesis language $P_0 \subseteq \{+, -\}^*$ is defined by the context-free grammar:

$$\begin{cases} S \to SS \\ S \to +S- \\ S \to \lambda \end{cases}$$

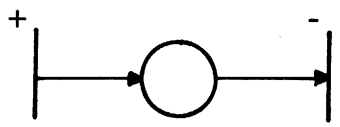
The incomplete simple parenthesis language $P \subseteq \{+, -\}^*$ consists of the prefixes of P_0 and is generated by the context-free grammar:

$$\begin{cases} S \to SS \\ S \to +S- \\ S \to +S \\ S \to \lambda \end{cases}$$

(S is the sentence symbol of the grammar).

Another characterization of P is the set of all strings such that, for every prefix, the number of "-" symbols does not exceed the number of "+" symbols. P_0 consists of those strings in P which have an equal number of "+" and "-". In other words, we have: $P = P_0 \| (+)^*$. The language P_0 will indeed turn out to express an essential characteristic of Petri nets.

It is easy to see that P and P_0 are the firing language and the cyclic terminal firing language , respectively, of the simple buffer shown in figure 5.1: (A cyclic terminal language is generated by a net whose final marking is the same as the initial marking. It is in CSS, but not strictly in $\mathcal{L}_0$; see the remark at the end of section 1.3; for Peterson's CSS see section 2.4).



Petri net A

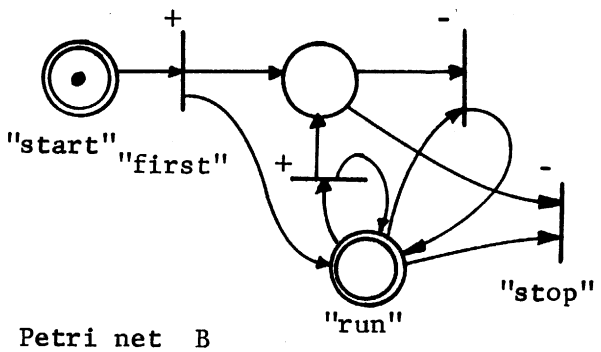
initial and final marking: $\langle 0 \rangle$

$\mathcal{L}(A) = P$

$\mathcal{L}_c(A) = P_0$

Figure 5.1

In this case it is a little annoying to exclude λ from $\mathcal{L}_0$-languages, because, strictly speaking, P_0 is not an $\mathcal{L}_0$-language; $P_0 - \{\lambda\}$ is, however, as can be seen from Figure 5.2, which shows a net in standard form (for $\mathcal{L}_0$):



Petri net B

$\mathcal{L}(B) = P$

$\mathcal{L}_0(B) = P_0 - \{\lambda\}$

Figure 5.2

5.3. Closed Subnets

The decomposition of Petri nets into subnets is a useful analytical tool, and has been used as such in Hack [5, 6].

Definition 5.3: A closed subnet of a Petri net $\mathbb{N} = \langle \Pi, \Sigma, F, B, M_0 \rangle$ is a Petri net $N' = \langle \Pi', \Sigma', F', B', M_0' \rangle$ such that:

$\Pi' \subseteq \Pi$ (defined by a subset of the places)

$\Sigma' = \{t \in \Sigma \mid \exists p \in \Pi' : F(p, t) > 0 \text{ or } B(p, t) > 0\}$

(all transitions connected to places in Π')

F' and B' are the restrictions of F and B to $\Pi' \times \Sigma'$.

M_0' is the restriction of M_0 to Π'.

In other words, a closed subnet of a Petri net consists of a subset of the places and all transitions connected to these places.

It is clear that if we look at the subnet N' within the original net N at some marking M reachable from M_0 by a firing sequence $\sigma \in \Sigma^*$, then the corresponding marking of N', written $M' = M/\Pi'$, could be obtained in N' from M'_0 by the firing sequence $\sigma' = \sigma/\Sigma'$ obtained from σ by erasing all transitions of $\Sigma - \Sigma'$. This is the characteristic property of a closed subnet: Its marking within the original net depends only on the firings of the subnet. All the rest of the net does is restrict the set of possible firing sequences.

The usefulness of the concept arises from the following Lemma:

Lemma 5.1: Let $A = \langle \Pi_A, \Sigma_A \ \ldots \ \rangle$ and $B = \langle \Pi_B, \Sigma_B \ \ldots \ \rangle$ be two closed subnets of a free-labelled Petri net $C = \langle \Pi_C, \Sigma_C \ \ldots \ \rangle$ such that $\Pi_C = \Pi_A \cup \Pi_B$. Then we have:

$$L(C) = (L(A) \| (\Sigma_C - \Sigma_A)^*) \cap (L(B) \| (\Sigma_C - \Sigma_B)^*)$$

where L is either the $\mathcal{L}$ or $\mathcal{L}_0$-language.

Proof: (Illustrated by means of the example of Figure 5.3). Let A' be the Petri net obtained by adding a non-connected transition for each transition in $\Sigma_C - \Sigma_A$, with the same label. Similarly, let B' be obtained by adding $\Sigma_C - \Sigma_B$ to B. (Figure 5.4). Now both A' and B' are free-labelled Petri nets over the alphabet Σ_C, and their respective languages are:

$$L(A') = L(A) | (\Sigma_C - \Sigma_A)^*$$

$$L(B') = L(B) | (\Sigma_C - \Sigma_B)^*$$

Let C' be the net obtained from C by duplicating the places in $\pi_A \cap \pi_B$. (Figure 5.5). This clearly does not change the language, since the two copies of a given place will at all times have the same marking and the same effect on firings as the single original place. Thus:

$$L(C') = L(C)$$

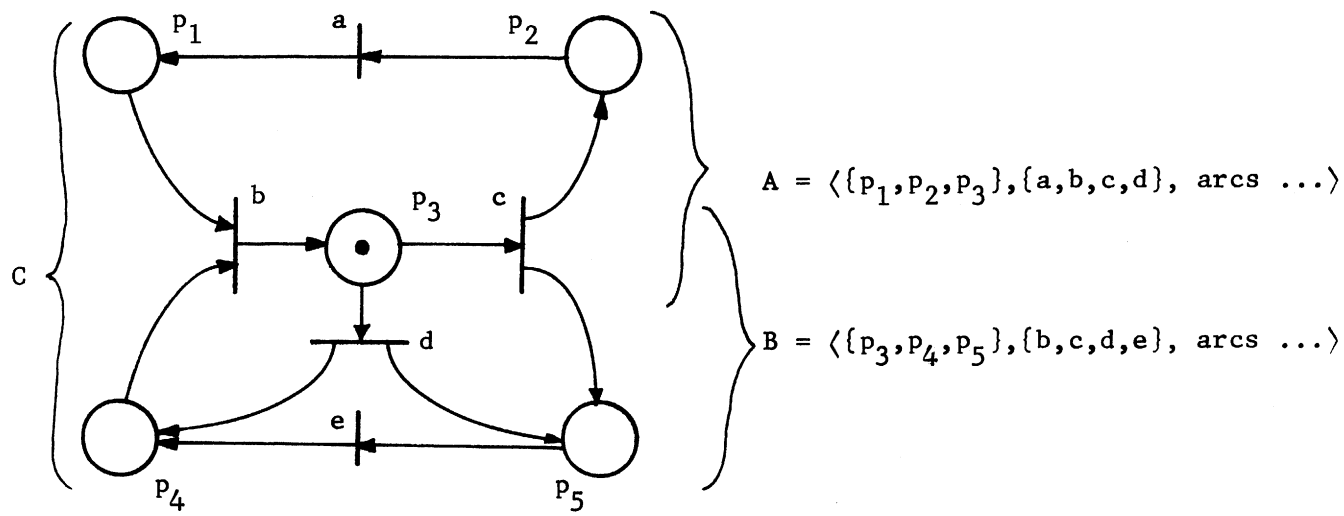
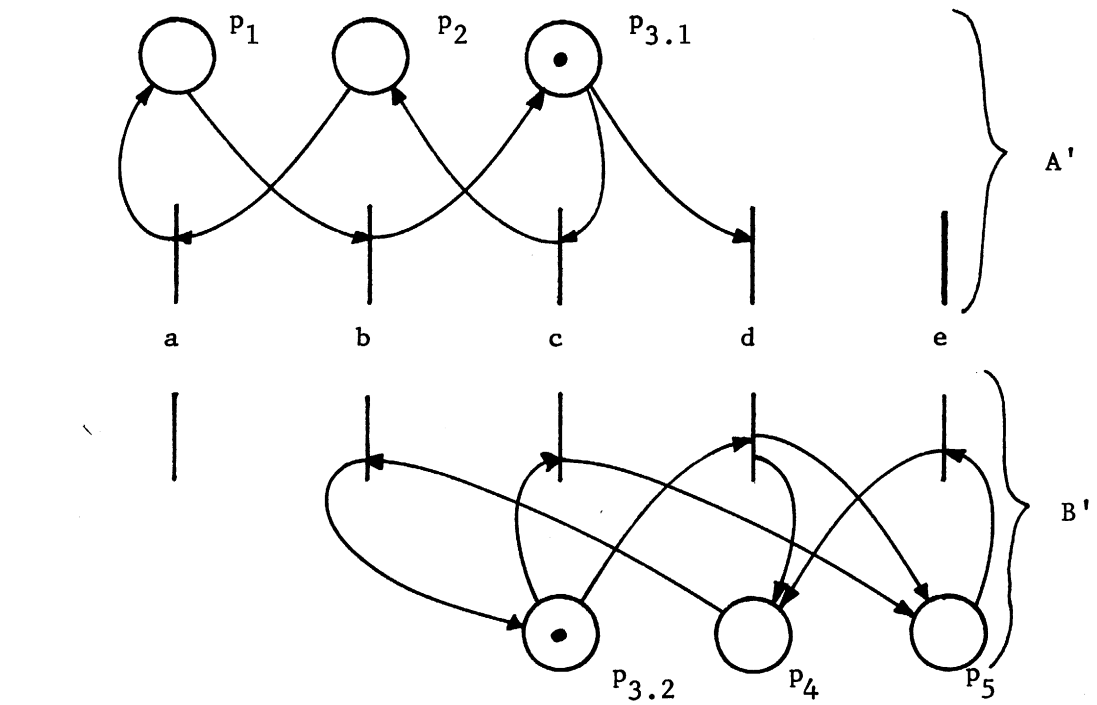


Figure 5.3



($p_{3.1}$ and $p_{3.2}$ are two copies of place p_3)

Figure 5.4

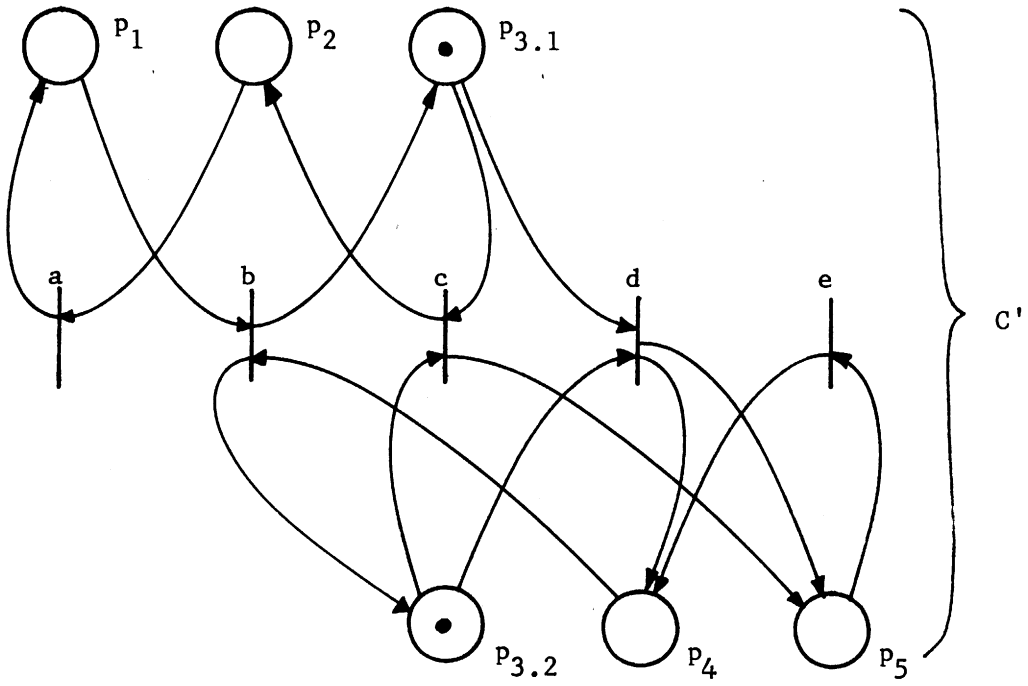


Figure 5.5

If we now refer to the construction for the intersection of two Petri net languages (Section 3.2), we notice that C' is precisely the result of that construction applied to A' and B', except that the "run" place π_0 is absent. Thus:

$$L(C) = L(C') = L(A') \cap L(B')$$

QED

As an example, consider the net in Figure 5.6. Its language can be determined by examining the whole reachability set, since it is finite and actually quite small; the firing sequences follow the pattern $(acbc)^*$.

But a more structure-oriented approach consists in first observing that the closed subnet $\langle\{p_2, p_3\}, \{a, b\}, \cdots\rangle$ imposes the alternation $(ab)^*$, whereas $\langle\{p_1, p_4\}, \{a, b, c\}, \cdots\rangle$ inserts a "c" after each "a" or "b": $((a + b)c)^*$. The result is $((ab)^* \| c^*) \cap ((a + b)c)^* = (acbc)^*$.

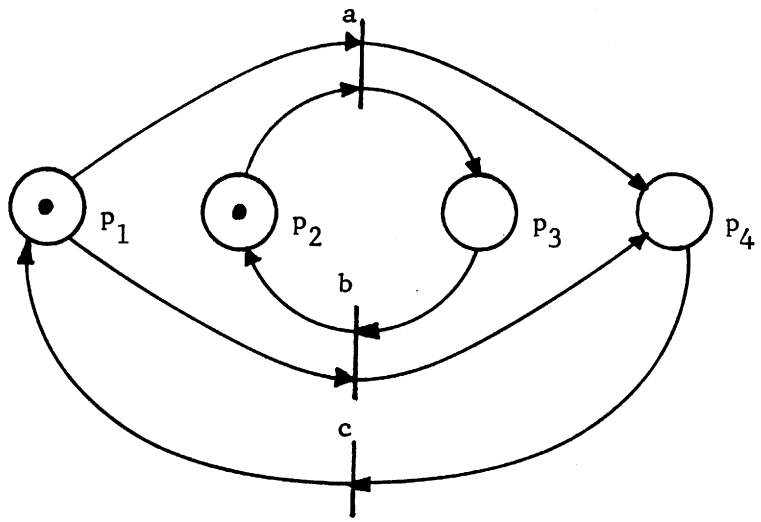


Figure 5.6

5.4. The Restriction Operation on Languages

In the previous section (5.3) we have seen that the language of a Petri net can be obtained from the languages of component subnets by an operation involving concurrency and intersection. Let us recall the formula from Lemma 5.1:

$$L(C) = (L(A) \| (\Sigma_C - \Sigma_A)^*) \cap (L(B) \| (\Sigma_C - \Sigma_B)^*)$$

Assuming for the moment that $\Sigma_C = \Sigma_A \cup \Sigma_B$, i.e. that the Petri net C did not have isolated (unconnected) transitions, we may rewrite this as: (replacing L(A), L(B), $\cdots$ by L_A, L_B $\cdots$)

$$L_C = (L_A \| (\Sigma_B - \Sigma_A)^*) \cap (L_B \| (\Sigma_A - \Sigma_B)^*)$$

This means that L_C consists of strings such that when all symbols from $(\Sigma_B - \Sigma_A)$ are erased, we get a string from L_A, and when all symbols from $(\Sigma_A - \Sigma_B)$ are erased, we get a string from L_B. In other words, the strings of L_C are constrained only by L_A over $(\Sigma_A - \Sigma_B)$, only by L_B over $(\Sigma_B - \Sigma_A)$, and by both L_A and L_B over $\Sigma_A \cap \Sigma_B$. If $\Sigma_B \subseteq \Sigma_A$, then L_C consists of those strings of L_A which satisfy the constraints imposed by L_B; the interpretation in the general case is a mutual restriction over the common alphabet, hence the name restriction operation:

Definition 5.4: The restriction of two languages L_A and L_B over alphabets Σ_A and Σ_B, respectively, is the language $L_A \textcircled{R} L_B$ over the alphabet $\Sigma_A \cup \Sigma_B$ defined by:

$$L_A \textcircled{R} L_B = (L_A \| (\Sigma_B - \Sigma_A)^*) \cap (L_B \| (\Sigma_A - \Sigma_B)^*)$$

Note: Since Petri net languages are closed under concurrency and intersection, and contain languages of the form $(\Sigma_B - \Sigma_A)^*$, they are also closed under restriction; in the case of $\mathcal{L}_0$ we observe that $L \| \Sigma^* = (L \| (\Sigma^* - \{\lambda\})) \cup L$.

5.5. A Characterization Using Inverse Homomorphism

From the two previous sections we know that the language of a free-labelled Petri net can be obtained from those of its closed subnets by the operation of restriction.

Consider a Petri net N in standard form. Let the places be $p_1 \ldots p_n$, where p_1 is the "start" place. Then we can construct a succession of closed subnets N_i obtained from $N_1 = \langle \{p_1\}$, "first" $\cup$ "singleton", $\ldots \rangle$ by successively adding more places: $N_{i+1} = N_i \cup \langle \{p_{i+1}\}$, {transitions connected to p_{i+1}}, $\cdots \rangle$. At each step, the language of N_{i+1} is obtained by the operation of restriction from the language of N_i and the additional constraints imposed by the new place p_{i+1}. Thus, only two primitive language forms are needed: The language of N_1, the "start" place, which is precisely the set "first" $\cup$ "singleton" for $\mathcal{L}_0^f$ or "first" $\cup$ "singleton" $\cup \{\lambda\}$ for $\mathcal{L}^f$, and the language of a place with input and output transitions and zero initial marking.

Consider a typical closed subnet defined by a single place, as in Figure 5.7.

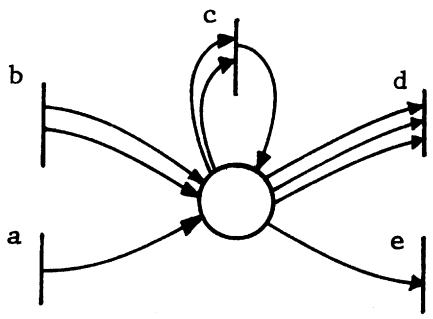


Figure 5.7

If a, b, c ... also stand for the number of occurrences of the corresponding transitions in a firing sequence, then it must be the case that, for every firing sequence (and every prefix thereof):

$$2a + b \geq c + 3d + e$$

Also, c is subject to the additional constraint that it may not fire unless there are at least two tokens, i.e. that $2a + b - c - 3d - e \geq 2$ for the firings that have already occurred.

This can be transformed into a simple parenthesis language by the following homormophism

$h: \{a, b, c, d, e\} \rightarrow \{+, -\}$ defined by:

$$\left|\begin{array}{l} h(a) = ++ \\ h(b) = + \\ h(c) = --+ \\ h(d) = --- \\ h(e) = - \end{array}\right.$$

a "-" for every arc from the place, followed by a "+" for every arc to the place.

Now the constraints described before can be simply expressed as:

x is a firing sequence $\Leftrightarrow$ $h(x) \in P$

x is a terminal sequence $\Leftrightarrow$ $h(x) \in P_0$

In other words, the $\mathcal{L}^f$ or $\mathcal{L}_0^f$ language of the one-place closed subnet with zero initial marking is obtained from P or P_0 by the inverse homomorphism h^{-1}.

Thus, every free Petri net language can be generated from a finite language consisting of one-symbol strings (and λ, for $\mathcal{L}^f$) and the inverse homomorphic images of P_0 or P by the repeated application of restriction. Note that this finite language can also be obtained from the language {-} containing the single string "-", or from $\{-, \lambda\}$, by inverse homomorphism. Conversely, since free Petri net languages are closed under restriction, every language generated in the above manner is a free Petri net language. In fact, a corresponding net can very easily be constructed directly from the various homomorphisms, one per place. The resulting net is not necessarily in standard form; the only properties of standard form we have used are the restriction on the initial marking: zero in all places that have input transitions, one in places that have no input transitions. Thus:

Theorem 5.3: The family $\mathcal{L}^f(\mathcal{L}_0^f)$ is precisely the class of languages obtained by the operation of restriction (or equivalently, the operations of intersection and concurrency) and inverse homomorphism from the (complete) simple parenthesis language P respectively P_0 and the "start" language $\{-\}$.

The table in Figure 5.8 shows the characterizations of the various families as it follows from Theorems 5.1, 5.2 and 5.3.

Family	Basis	Closure Operations
$\mathcal{L}_0^f$	$\{-\}$, P_o	inverse homomorphism restriction
$\mathcal{L}^f$	$\{\lambda, -\}$, P	inverse homomorphism restriction
$\mathcal{L}_0$ plus λ i.e. CSS (see 2.4) $\mathcal{L}_o$ consists of all λ-free languages in this class	$\{-\}$, P_0	inverse homomorphism restriction, or (concurrency †) (intersection †) λ-free renaming, or λ-free homomorphism
$\mathcal{L}$	$\{\lambda, -\}$, P	inverse homomorphism restriction, or (concurrency †) (intersection †) λ-free renaming
$\mathcal{L}^\lambda$	$\{-\}$, P	inverse homomorphism restriction, or (concurrency †) (intersection †) unrestricted renaming
$\mathcal{L}_0^\lambda$	$\{-\}$, P	inverse homomorphism restriction, or (concurrency †) (intersection †) unrestricted renaming, or unrestricted homomorphism

Figure 5.8

†) If we use concurrency and intersection instead of restriction, we also need the Regular language $\{-\}^*$ in the basis.

Thus, the general form of a Petri net language is:

- terminal: ($\mathcal{L}_0$,...) $\quad L = h_0(h_1^{-1}(\{-\}) \textcircled{R} h_2^{-1}(P_0) \textcircled{R} \cdots \textcircled{R} h_n^{-1}(P_0))$ †)

- non-terminal: ($\mathcal{L}$,...) $\quad L = h_0(h_1^{-1}(\{\lambda, -\}) \textcircled{R} h_2^{-1}(P) \textcircled{R} \cdots \textcircled{R} h_n^{-1}(P))$

where h_0 is a renaming homomorphism which is identity for $\mathcal{L}^f$ and $\mathcal{L}_0^f$, λ-free for $\mathcal{L}$ and $\mathcal{L}_0$, unrestricted for $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$. The other homomorphisms h_i, $1 \leq i \leq n$, are unrestricted.

From this we may also conclude that $\mathcal{L}$ and $\mathcal{L}_0$-languages are context-sensitive: P and P_0 are context-free, therefore $h^{-1}(P)$ or $h^{-1}(P_0)$ are context-free, and thus context sensitive. But context-sensitive languages are closed under concurrency, intersection, and λ-free homomorphism. Thus:

Theorem 5.4: The families $\mathcal{L}$ and $\mathcal{L}_0$ are contained in the family of context-sensitive languages (except for the empty string in $\mathcal{L}$-languages).

5.6. Bounded Subnets

In the previous section we have generated the free Petri net languages by applying the restriction operation to closed subnets consisting of a single place. But often it is advantageous to consider larger subnets whose language is easy to determine, thus reducing the number of restriction operations required.

A logical choice for such larger subnets are State Machines, as described in [5 , 6], and bounded closed subnets in general, because the associated languages are necessarily regular and can usually be determined without difficulty (for a qualification of this assertion, see section 9.4).

This approach generates the free Petri net languages by applying restrictions to one or several regular languages and the homomorphic inverses of P or P_0 for the remaining unbounded places.

---

†) It can be verified that the operation (R) is associative and commutative.

5.7. A Simple Characterization of $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$

In the case of $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$ we may use the flexibility offered by λ-transitions to obtain a simpler generation of $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$, without recourse to inverse homomorphism. As a bonus, this approach will also provide us with a technique for reducing multiple arcs in a promptness preserving manner (see Definition 4.4 in Section 4.8).

In Hack [7] we have shown that a GPN (Generalized, i.e. Multiple-arc Petri net) can be transformed into an equivalent -- up to λ-firings -- RPN (Restricted, i.e. Self-Loop free Ordinary Petri net). Therefore, the classes $\mathcal{L}^{\lambda}$ and $\mathcal{L}_0^{\lambda}$ can be generated by Restricted Petri nets. But the transformation does not preserve promptness, which may be a drawback in some cases.

We may take advantage of the observation in 5.6 and transform the bounded subnets into State Machines generating the same regular language by the method of Section 4.1. This is not necessary, however, and the following transformation can be carried out place by place in any GPN.

The principle behind this method lies in separating the buffering functions of a place from its "role" in distributing tokens to firable transitions. Our previous reduction did not separate these aspects.

The reduction still operates locally, on a place and its surrounding transitions (i.e. on a one-place closed subnet). Figure 5.9 shall be our example.

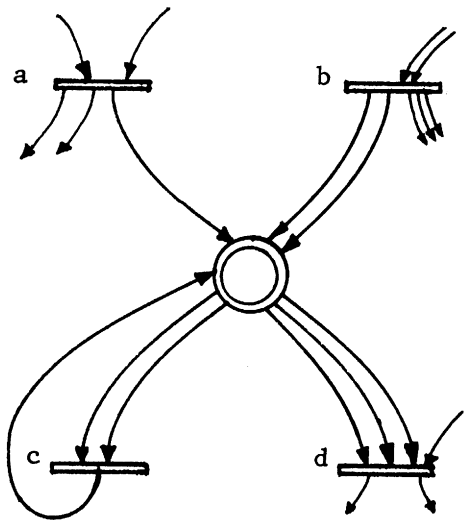


Figure 5.9

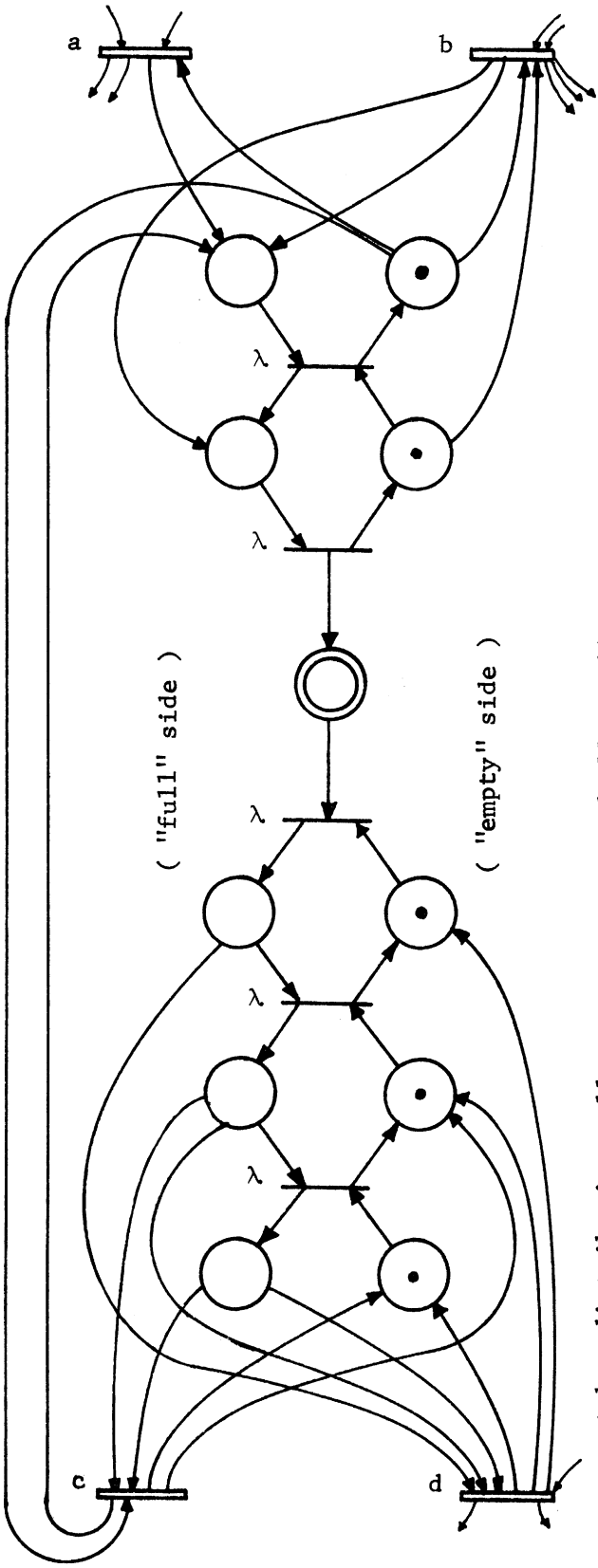


Figure 5.10a

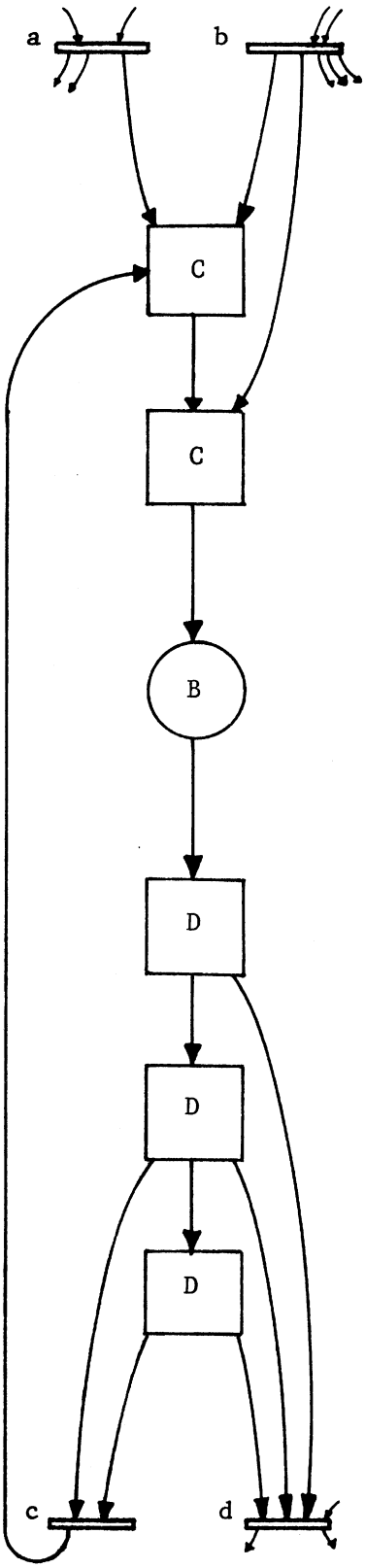


Figure 5.10b

The transitions may be shared with other places, and the corresponding arcs would be reduced together with these other places.

We shall first consider the token-distribution function of the place. Transitions c and d can fire only if there are at least 2 respectively 3 tokens available, and no transition requires more than 3 tokens. Any number of tokens above 3 should be handled by the buffering function of the place.

The distribution of 0 to 3 tokens can be accomplished by means of a 3-token pipeline, which may alternatively be looked at as a 4-state counter, indicating whether the pipeline contains zero, one, two, or three tokens on the "full" side (and the complement to three on the "empty" side).

This pipeline is shown at the bottom of the transformed net, shown in Figure 5.10a. It consists of three distribution stages (as shown schematically in Figure 5.10b) each of which is a 2-place one-token closed subnet. The token is either on the "full" or on the "empty" side. The number of stages which have their token on the "full" side determines how many tokens are available for firing an output transition (c or d). A similar "collecting" pipeline counts the tokens provided by the input transitions (a, b, or c). Adding n counts to such a pipeline means switching n tokens from the "empty" to the "full" side, and removing n counts does the reverse. Any overflow in the collecting pipeline is directed into the buffer place, which in turn fills the distribution pipeline, where the counts always travel down to the bottom of the pipeline. The output transitions draw their arcs from the bottommost stages of the pipeline; this prevents deadlock.

The construction preserves promptness, because between any firings of a, b, c and d there can be only a bounded number of firings of the newly introduced λ-transitions -- in this case, seven (if the two pipelines have n and m stages respectively, the maximum number of consecutive λ-firings is $(n^2+n)/2 + (m^2+m)/2$, as can easily be established). The number of stages required for the two pipelines is the maximum number of input arcs and output arcs, respectively.

The construction preserves firing sequences both ways, up to λ-firings, because equivalent firing sequences produce equivalent markings, and equivalent markings generate the same constraints on allowable firing sequences. An "old" marking of n tokens is equivalent to a "new" marking of $n = C + B + D$ tokens, where C and D are the number of "full" stages in the collecting and distributing pipelines, and B is the number of tokens in the buffer.

It may be verified that the transformation also preserves liveness.

Thus we have:

Theorem 5.5: For each prompt labelled GPN there exists a language-equivalent prompt RPN, possibly with more λ-transitions.

After having transformed all places (or at least all unbounded places) of a Petri net, the transformed net only contains one-input-one-output buffers and a collection of bounded closed subnets. There is no need to apply inverse homomorphisms to P or P_0 to obtain the language of these buffers -- a simple renaming will do. After transforming the net, we assign new, distinct names to all transitions to get a free-labelled net whose language can now be obtained by restrictions, then we use unrestricted renaming to get our original language. Thus:

Theorem 5.6: $\mathcal{L}^{\lambda}$ is the closure of P and the prefix regular languages under restriction (concurrency and intersection) and unrestricted renaming.

$\mathcal{L}_0^{\lambda}$ is the closure of P_0 and the regular languages under restriction (concurrency and intersection) and unrestricted renaming or homomorphism.

In both cases, two-state regular languages (Flip-Flops) are sufficient.

5.8. A Characterization Using Finite Substitution

Even in the case of $\mathcal{L}_0$ and $\mathcal{L}$, where we may not take advantage of λ-transitions, we may generate the languages without recourse to inverse homomorphism.

Inverse homomorphism was needed to obtain the language of one-place closed subnets such as the one in Figure 5.7, where there are multiple arcs and self-loops. But consider the one-place closed subnet of Figure 5.11, which is a Restricted Petri Net (RPN):

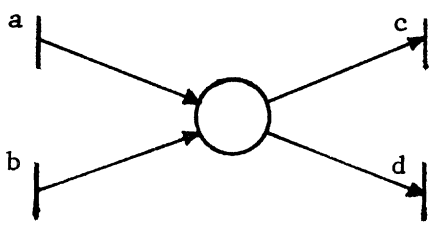


Figure 5.11

Its language is clearly the result of the finite substitution:
$\sigma(+) = \{a, b\}$
$\sigma(-) = \{c, d\}$ applied to P or P_0. From this follows:

Lemma 5.2: The languages of free RPN's can be obtained from the regular languages and P or P_0 by closure under finite substitution and restriction.

In fact, it is sufficient that unbounded places not have multiple arcs or self-loops.

Now we will show that we may replace an unbounded place that has self-loops and/or multiple arcs by a closed subnet consisting of two bounded places and one unbounded place without self-loops or multiple arcs, without using λ-transitions.

Consider a place p and the transitions connected to it, say t_1 and t_2 (Figure 5.12). Let F(t) and B(t) be the size of the arc bundle of transition t (t_1 or t_2) from or to place p.

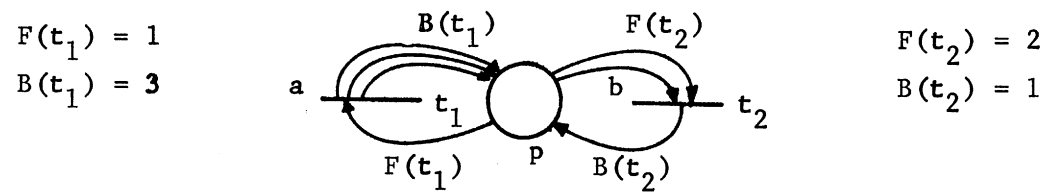


Figure 5.12

We will replace p with a bounded counter, consisting of new places π and $\bar{\pi}$, and an (a priori) unbounded buffer, consisting of a third new place β. The bounded counter has m+1 states representing a count of 0 to m tokens in the counter, where a count of x is represented by x tokens in π and m-x tokens in the complementary place $\bar{\pi}$; m is the capacity of the counter. The buffer is an unbounded counter, where each count represents k tokens. A given marking M(p) is represented by the combination $M(\pi) + k \cdot M(\beta)$, where $0 \leq M(\pi) \leq m$. If a transition fires, and the corresponding change of $M(p) = M(\pi) + k \cdot M(\beta)$ is within the bounds of the counter, only $M(\pi,\bar{\pi})$ changes; otherwise, k tokens are deposited or withdrawn from $k \cdot M(\beta)$, i.e. β gets or loses one token, and $M(\pi)$ changes so as to express the proper new marking.

Accordingly, each transition t will be replaced by three transitions t^0, t^+ and t^-, labelled like t, which change only $M(\pi)$, or change $M(\pi)$ and deposit or withdraw k counts from the buffer. This can be summarized in the following table (Figure 5.13); the full construction is shown in Figure 5.14. F(t) and B(t) stand for F(p, t) and B(p, t), i.e. the forwards ($p \rightarrow t$) and backwards ($p \leftarrow t$) arc size between p and t.

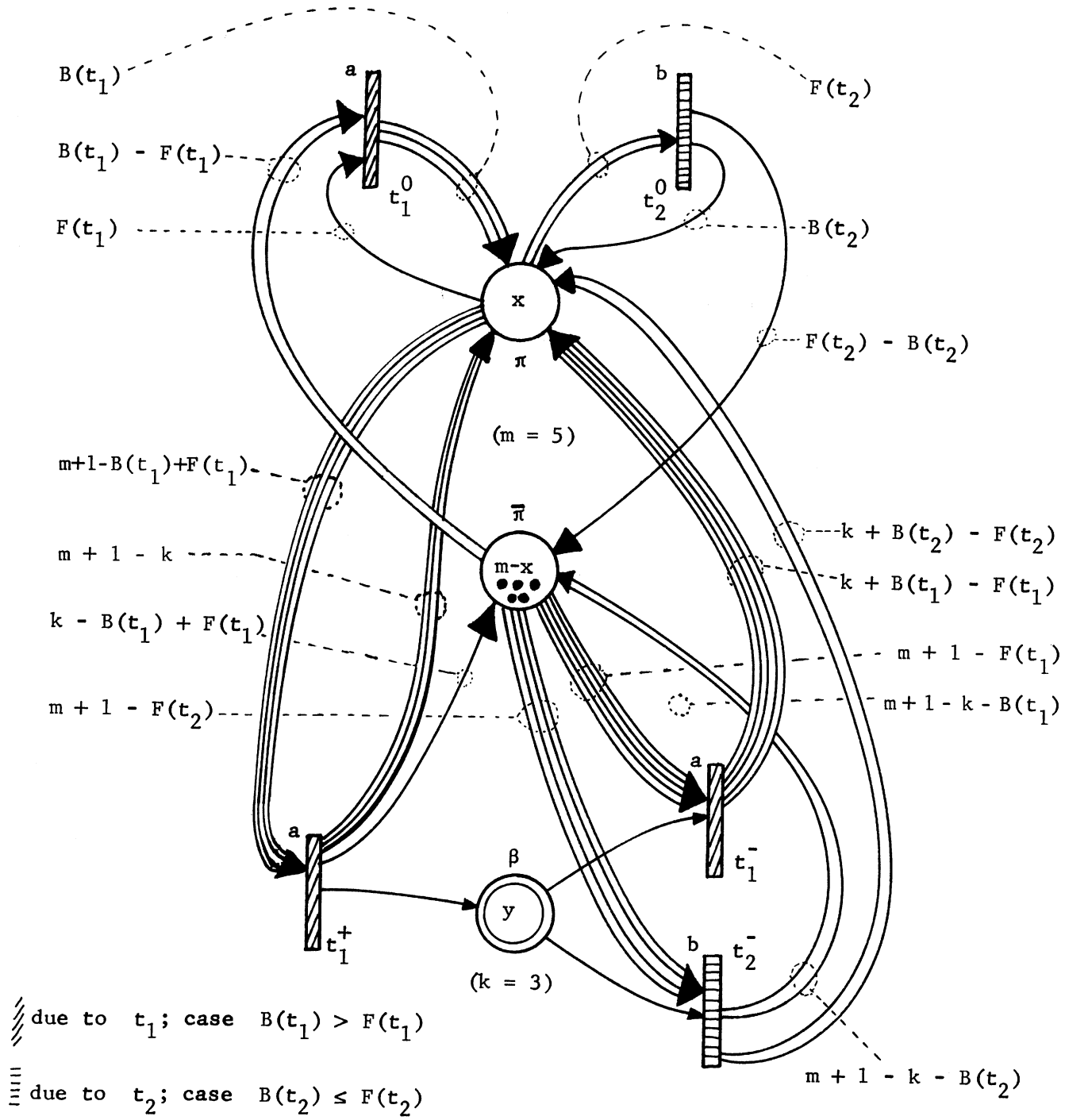
Marking Before the Firing:	Firing of:	Marking Change to:		
		$M(\pi)$	$M(\beta)$	$M(\pi) + k \cdot M(\beta)$
$M(p) = M(\pi) + k \cdot M(\beta)$	t			B(t) - F(t)
	t^o	B(t) - F(t)	0	B(t) - F(t)
	t^+	B(t) - F(t) - k	+1	B(t) - F(t)
	t^-	B(t) - F(t) + k	-1	B(t) - F(t)

Figure 5.13

Now we must choose m and k such that, for all possible markings at which t is firable, at least one of t^o, t^- or t^+ is firable such that $M(\pi)$ stays within bounds, and that when $M(p) < F(t)$, then none of t^o, t^- or t^+ are firable. The table of Figure 5.15 shows a suitable connection matrix, and the table in Figure 5.16 shows under which circumstances the various transitions may fire. It is advantageous to distinguish the cases $B(t) > F(t)$ and $B(t) \leq F(t)$, because in the latter case no t^+-transition is needed. It will be seen that, if and only if t is firable at M(p), then exactly one of t^o, t^-, or t^+ will be firable at $M(\pi, \bar{\pi}, \beta)$, and the resulting marking change is such that the equation $M(p) = M(\pi) + k \cdot M(\beta)$ remains valid.

To obtain values for m and k, we must make sure that $M(\pi)$ always stays within the bounds $0 \leq M(\pi) \leq m$. From the table of figure 5.16 it can be verified that this will be the case if we choose:

$$k \geq \max_{\text{(for all t connected to p)}}(|F(t) - B(t)|) \qquad \text{and} \qquad m \geq k - 1 + \max_{\text{(for all t connected to p)}}(B(t)).$$



equivalent marking: $x + k \cdot y$

In this example (cf. Fig. 5.12), we may choose: $k = 3$; $m = 5$.

Figure 5.14

transition θ =	case $B(t) > F(t)$			case $B(t) \leq F(t)$	
	t^0	t^-	t^+	t^0	t^-
$F(\pi,\theta)$	$F(t)$	0	$m+1-B(t)+F(t)$	$F(t)$	0
$F(\overline{\pi},\theta)$	$B(t)-F(t)$	$m+1-F(t)$	0	0	$m+1-F(t)$
$F(\beta,\theta)$	0	1	0	0	1
$B(\pi,\theta)$	$B(t)$	$k+B(t)-F(t)$	$m+1-k$	$B(t)$	$k+B(t)-F(t)$
$B(\overline{\pi},\theta)$	0	$m+1-k-B(t)$	$k-B(t)+F(t)$	$F(t)-B(t)$	$m+1-k-B(t)$
$B(\beta,\theta)$	0	0	1	0	0

Figure 5.15

case	marking of π before firing		firable transition	marking change		marking of π after firing		
	from	to		π	β	from	to	
$M(\beta) = 0$	0	F-1	none					N.A. if F=0
$M(\beta) > 0$	0	F-1	t^-	B-F+k	-1	k+B-F	k+B-1	
$B > F$	F	m+F-B	t^0	B-F	0	B	m	
	m+F-B+1	m	t^+	B-F-k	+1	m-k+1	m-k+B-F	
$B \leq F$	F	m	t^0	B-F	0	B	m+B-F	

(note: F = F(t), B = B(t))

Figure 5.16

The initial marking of the new places should be such that $M(\pi) + k \cdot M(\beta) = M(p)$ and $M(\overline{\pi}) = m - M(\pi)$. in particular, if $M(p) = 0$, we start out with $M(\pi) = 0$, $M(\overline{\pi}) = m$, and $M(\beta) = 0$. It is also always possible to choose m large enough in order to have zero tokens in the buffer β at the initial marking.

It appears that this transformation does not preserve free-labelling, since each transition may be replaced by two or three similarly-labelled transitions at each place-replacement step. But the $\mathcal{L}_0$ or $\mathcal{L}$-language is not changed. By performing the transformation until all (or at least all unbounded) places with multiple arcs have been eliminated, we arrive at a net which can be decomposed into closed subnets of two kinds: bounded, or single-arc buffers such as in figure 5.1. Thus:

Theorem 5.7: The λ-free Petri net languages (of families $\mathcal{L}_0$ augmented by λ, and $\mathcal{L}$) are obtained by closure under restriction (i.e. concurrency and intersection) and λ-free renaming from the Regular languages (prefix in the case of $\mathcal{L}$) and the images under finite substitution of P_0 (or P, in the case of $\mathcal{L}$).

Since the various bounded closed subnets, which may still contain multiple arcs, can be replaced by Restricted Petri nets by the methods of section 4.1, we also have:

Theorem 5.8: a) All Petri net languages ($\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$) can be generated using only Restricted Petri nets.

b) Alternate means for generating Petri net languages include Finite State Machines constrained by, or interconnected by, token buffers.

Finally, we remember that $P = P_0 \parallel \{+\}^*$. Now, if S is a substitution (finite or not), we have: $S(P_0 \parallel \{+\}^*) = S(P_0) \parallel (S(+))^*$. Therefore, $\mathcal{L}$-languages can in fact be obtained in the same way as $\mathcal{L}_0$-languages. In effect, each time we perform the construction described in figure 5.14, we add transitions $t_i^\dagger$ which are connected to places π and $\bar{\pi}$, and are labelled like t_i^+ , but do not deposit a token into buffer place β. It can be seen that no new label sequences are introduced, but for each firing sequence σ leading up to submarking M of the buffer places, and for each submarking $M' \leq M$, including zero, there exists a firing sequence σ' generating the same label sequence as σ and reaching M' instead of M. No new label sequences are possible, because $M' \leq M$ implies that every label sequence which may follow the label sequence generated by σ' could also have followed that generated by σ. If we now make every state of the Finite State Machine representing the bounded subnets a final state, it appears that every non-empty string of the $\mathcal{L}$-language of the original net can be generated by the $\mathcal{L}_0$-language of the modified net. Thus:

Theorem 5.9: a) $\{ L-\{\lambda\} \mid L \in \mathcal{L} \} \subseteq \mathcal{L}_0$

b) $\mathcal{L} \subseteq$ CSS (cf. section 2.4)

6. Decidable Properties of Petri Net Languages

Most of the decidable properties can be reduced to the decidability of boundedness of a place. This was first proved (for Vector Addition Systems) by Karp and Miller [11].

Lemma 6.1: It is decidable whether a given place is bounded in a given GPN with a given initial marking; and also whether a given place can ever receive at least one token.

Proof: See Hack [7].

6.1. Membership, for $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$-languages.

It is clearly decidable whether a given firing sequence is possible from the initial marking: Just start at the initial marking and try to fire it, transition occurence by transition occurrence. Since in the case of a λ-free Labelled Petri net, every label sequence corresponds to only a finite number of firing sequences, we conclude that membership in $\mathcal{L}$ or $\mathcal{L}_0$-languages is decidable. In fact, since we know that $\mathcal{L}$ and $\mathcal{L}_0$-languages are context-sensitive, and hence recursive,† we have already proved in the previous chapter that membership in $\mathcal{L}$ and $\mathcal{L}_0$-languages is decidable.

The case of $\mathcal{L}^\lambda$-languages is more interesting. Suppose we wish to decide whether a string, say "abac", is in the $\mathcal{L}^\lambda$-language of some labelled Petri net A. Let us construct a Petri net B which spells out the string "abac", as shown in Figure 6.1; it is a trivial Finite-State Machine. Place p_5 will receive a token if and only if the string "abac" is actually fired.

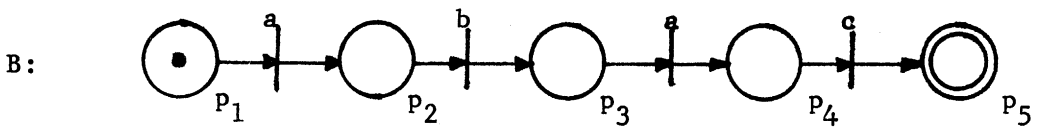


Figure 6.1

† and effectively so.

Now let us perform the intersection construction of Section 3.2 for the two nets A and B, as is indicated schematically in Figure 6.2.

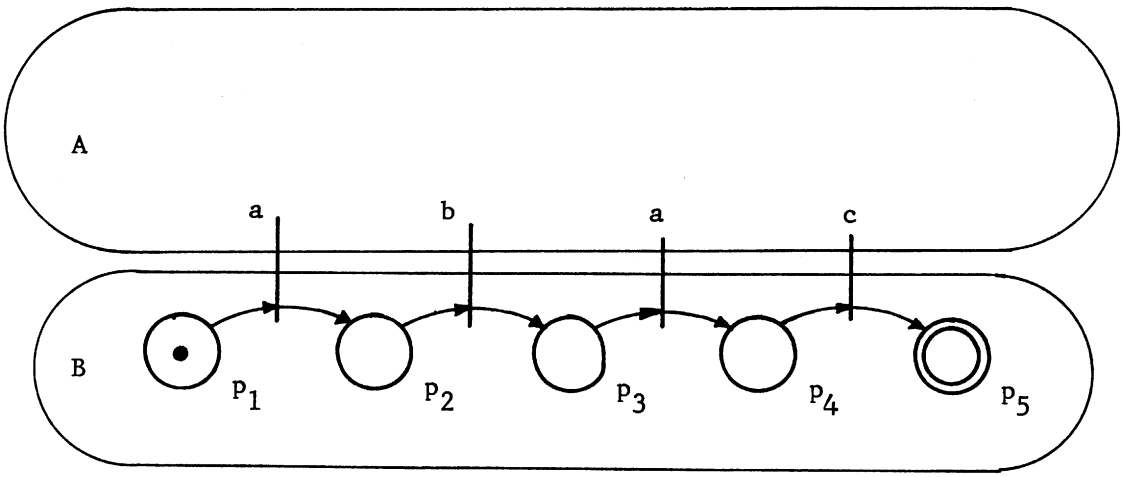


Figure 6.2

Now the test place p_5 of B may eventually receive a token if and only if abac $\in \mathcal{L}^{\lambda}$(A). Therefore, membership in $\mathcal{L}^{\lambda}$(A) is decidable.

Theorem 6.1: Membership is decidable for Petri net languages of the families $\mathcal{L}$, $\mathcal{L}_0$ and $\mathcal{L}^{\lambda}$; these families are thus included in the class of recursive languages.

We do not know at the present time whether $\mathcal{L}^{\lambda}$-languages are context-sensitive. The decidability of the membership problem for $\mathcal{L}_0^{\lambda}$ is also open; we shall return to this problem in chapter 7.

6.2. Emptiness and Finiteness for $\mathcal{L}$ and $\mathcal{L}^{\lambda}$-languages.

Emptiness of $\mathcal{L}$ and $\mathcal{L}^{\lambda}$-languages is an "empty" problem, since these languages always contain at least the empty string λ. If we ask whether the language contains other strings besides λ, we may just consider one-symbol strings because of the prefix property; this involves a finite number of membership tests. The emptiness problem for $\mathcal{L}_0$ and $\mathcal{L}_0^{\lambda}$ will be considered in the next chapter.

To decide whether an $\mathcal{L}$ or $\mathcal{L}^{\lambda}$-language is infinite, all we have to do is count the number of firings of labelled transitions, by adding a "count" place which gets a token from each labelled transition. Since a language is infinite if and only if it contains unboundedly long strings, the "count" place will be bounded if and only if the language is finite. Thus:

Theorem 6.2: Emptiness and Finiteness are decidable for $\mathcal{L}$ and $\mathcal{L}^{\lambda}$-languages.

So far, nothing is known about the finiteness problem for $\mathcal{L}_0$ or $\mathcal{L}_0^{\lambda}$-languages.

To decide whether an $\mathcal{L}$ or $\mathcal{L}_\gamma$-language is infinite, all we have to do is count the number of strings of labelled transitions, by adding a "count" place which gets a token from each labelled transition. Since a language is infinite if and only if it contains unboundedly long strings, the "count" place will be bounded if and only if the language is finite. Thus:

Theorem 6.2: Boundedness and Finiteness are decidable for $\mathcal{L}$ and $\mathcal{L}_\gamma$-languages.

So far, nothing is known about the finiteness problem for $\mathcal{L}_0$ or $\mathcal{L}_{0\gamma}$-languages.

7. Problems Equivalent to the Reachability Problem

The Reachability Problem is the problem to decide whether a given marking is reachable from the initial marking in a given Petri net. It is equivalent to the Zero Reachability Problem (trying to reach the Zero marking). A detailed account of this and related problems can be found in Hack [7 , 8]; the decidability of this problem is still an open question.

7.1. Emptiness of $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$-languages

We ask whether there exists, in a given labelled Petri net, any terminal label sequence. If the net is in Standard Form, this is precisely the Zero Reachability Problem -- it is actually totally irrelevant whether the transitions are labelled or not.

Thus:

Theorem 7.1: The emptiness problem for $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$-languages is equivalent to the Reachability Problem.

7.2. Membership in $\mathcal{L}_0^\lambda$-languages

Since $\mathcal{L}_0^\lambda$ is closed under intersection with regular languages, membership in an $\mathcal{L}_0^\lambda$-language can be reduced to emptiness of the intersection of that language and a one-string language, and thus membership in an $\mathcal{L}_0^\lambda$-language is reducible to the Reachability Problem.

Now we will show that the converse also holds. To do this, we will show that $\mathcal{L}_0^\lambda$-languages can suitably encode Reachability sets.

Let A be a GPN with places $p_1 \ldots p_n$ whose Reachability set is to be encoded. Let B be the labelled GPN obtained by leaving all of A's transitions unlabelled (λ-transitions) and by adding a "run" place π_0 which self-loops on every transition in A, a set of n places $\pi_1 \ldots \pi_n$, a set of new λ-transitions $\theta_1 \ldots \theta_n$, a set of n labelled transitions with labels $a_1 \ldots a_n$, and a "stop" λ-transition. See Figure 7.1.

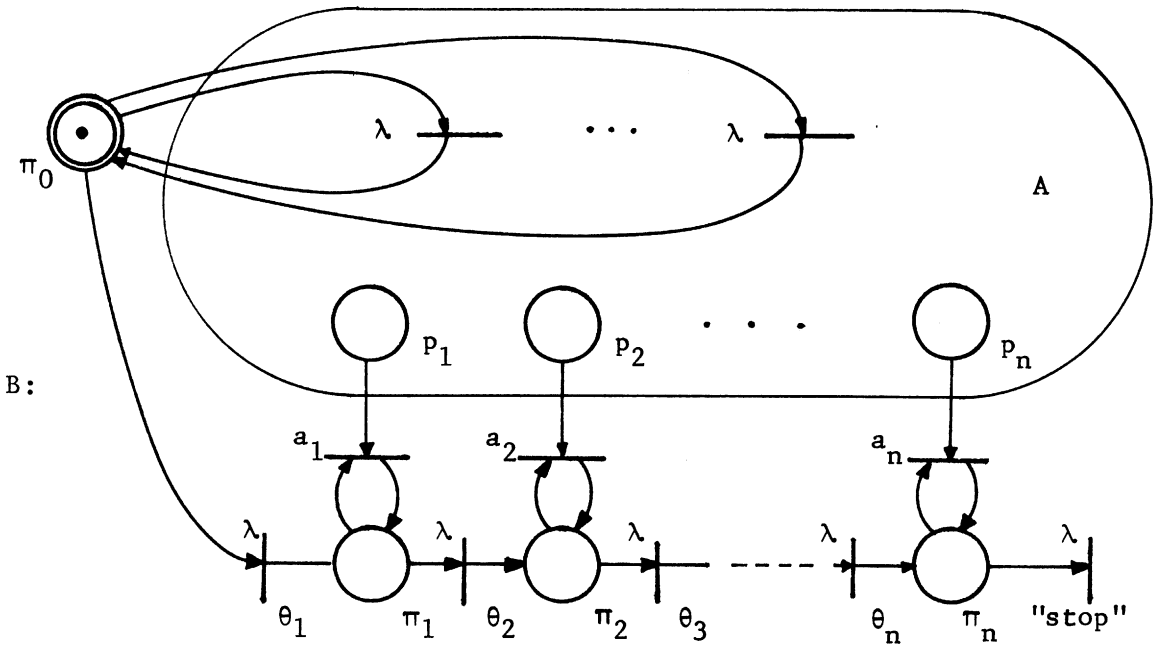


Figure 7.1

The initial marking consists of the initial marking of A for the old places $p_1 \cdots p_n$, one token in π_0 and zero tokens in $\pi_1 \cdots \pi_n$. The new λ-transitions θ_i transfer a token from π_{i-1} to π_i; "stop" removes a token from π_n. Each a_i-transition self-loops on π_i and removes one token from p_i.

While π_0 has its token, A fires as it did before being modified, and reaches some marking $M \in R(A)$ before θ_1 fires. Now the only way to reach the zero marking in the modified net B is to fire the firing sequence $\theta_1 a_1^{M(p_1)}\ \theta_2 a_2^{M(p_2)} \cdots \theta_n a_n^{M(p_n)}$ "stop". Therefore, the $\mathcal{L}_0^\lambda$-language of B encodes the reachability set of A as follows:

$$\mathcal{L}_0^\lambda(B) = \left\{ a_1^{x_1} a_2^{x_2} \cdots a_n^{x_n} \;\middle|\; \langle a_1, \ldots, x_n \rangle \in R(A) \right\}$$

We may now use this encoding to test whether a marking is reachable in A: We test whether the corresponding string is in $\mathcal{L}_0^\lambda(B)$. Thus:

Theorem 7.2: The membership problem for $\mathcal{L}_0^\lambda$-languages is equivalent to the Reachability Problem.

7.3 Equivalence Problems for Free Petri Net Languages.

We may recall that in a free-labelled Petri net every transition has a unique label. The corresponding families $\mathcal{L}^f$ and $\mathcal{L}_0^f$ of free Petri net languages are quite restricted; in fact, they enjoy practically none of the various closure properties. They are only closed under intersection (as can be stablished by transition sharing) and under suitably modified operations, such as disjoint concurrency, dijoint concatenation, etc. . Also see Theorem 5.2 .

In this section we show that the equivalence problems for free Petri net languages are reducible to the Reachability Problem, and may thus one day be shown to be decidable.

Consider the construction of figure 7.2 . It consists of two free-labelled Petri nets A and B , with transitions $t_1^A \cdots t_n^A$ and $t_1^B \cdots t_n^B$ respectively, where t_i^A and t_i^B have identical labels a_i . We wish to test whether $\mathcal{L}^f(A) \subseteq \mathcal{L}^f(B)$.

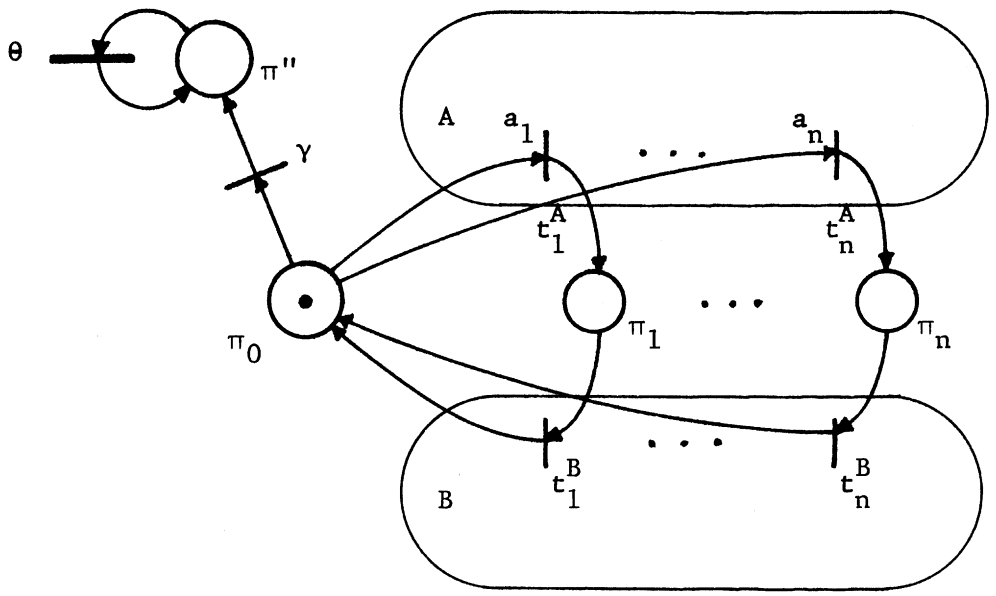


Figure 7.2

The two nets are connected as in the construction for for the intersection of $\mathcal{L}(A)$ and $\mathcal{L}(B)$, with a central place π_0 and symbol-remembering places π_i (cf. figure 3.4, section 3.2). In addition, there is a transition labelled γ which takes a token from π_0 to a place π'' (and thus freezes both A and B), and a transition θ which self-loops on π''. This transition is said to be live iff every firing sequence of the net can be continued to include or repeat a firing of θ. The problem

of deciding whether a transition θ is live or not is an instance of the Sub-Liveness Problem , which is recursively equivalent to the Reachability Problem (Hack, [7,8].)

It can be seen that the only way θ can fail to be live is if there exists a firing sequence σ which cannot be completed to include θ ; this means that the token originally in π_0 must get stuck in some π_i . That, in turn, can only happen if the label sequence w spelled out by A's transitions before the last firing of t_i^A was also spelled out by B's transitions, but $wa_i \in \mathcal{L}^f(A)$ and $wa_i \notin \mathcal{L}^f(B)$. This is because in a free Petri net, for a given label sequence there is exactly one firing sequence which generates it.

Conversely, if $\mathcal{L}^f(A) \not\subseteq \mathcal{L}^f(B)$, then there exists a shortest $w \in \mathcal{L}^f(A) \cap \mathcal{L}^f(B)$ such that for some symbol a_i, we have $wa_i \in \mathcal{L}^f(A)$, but also $wa_i \notin \mathcal{L}^f(B)$. Thus:

Theorem 7.3: The inclusion (and hence also the equivalence) problem for the free Petri net language family $\mathcal{L}^f$ is reducible to the Reachability Problem.

Now let us augment the construction of figure 7.2 as shown in figure 7.3 . We have added a place π' which receives a token from each A-transition and yields one token to each B-transition. So far, π' records the fact that some label sequence $w \in \mathcal{L}^f(A)$ has been has been generated by A, but the last firing has not yet been echoed by B.

We also add a number of transitions labelled α which self-loop on the places of A, and a number of transitions labelled β which self-loop on the places of B. If A (or B) has reached the zero marking, then all α-transitions (or all β-transitions) are disabled. Each α-transition carries a token from π' to π''. Each β-transition carries a token from π_0 to π'; this is possible only if all π_i-places ($1 \le i \le n$) are empty.

As before, θ is not live if and only if a token may get stuck -- in this case stuck in π'. This can only happen in one of two ways:

1) The label sequence w spelled out by A cannot be echoed by B (i.e. $w \in \mathcal{L}^f(A)$ & $w \notin \mathcal{L}^f(B)$) and no α-transition is enabled (i.e. $w \in \mathcal{L}_0^f(A)$). Since $\mathcal{L}_0^f(B) \subseteq \mathcal{L}^f(B)$ we have: $w \in \mathcal{L}_0^f(A) - \mathcal{L}_0^f(B)$.

2) The label sequence w spelled out by A has been fully echoed by B (i.e. $w \in \mathcal{L}^f(A)$ & $w \in \mathcal{L}^f(B)$), but a β-transition has fired (i.e. $w \notin \mathcal{L}_0^f(B)$) and no α-transition is firable (i.e. $w \in \mathcal{L}_0^f(A)$). Again, we have: $w \in \mathcal{L}_0^f(A) - \mathcal{L}_0^f(B)$.

Conversely, if there exists a $w \in \mathcal{L}_0^f(A) - \mathcal{L}_0^f(B)$, then either 2) or 1) may

occur, depending on whether $w \in \mathcal{L}^f(B)$ or not.

As before, a test for the liveness of θ establishes whether $\mathcal{L}_0^f(A) \subseteq \mathcal{L}_0^f(B)$ or not.

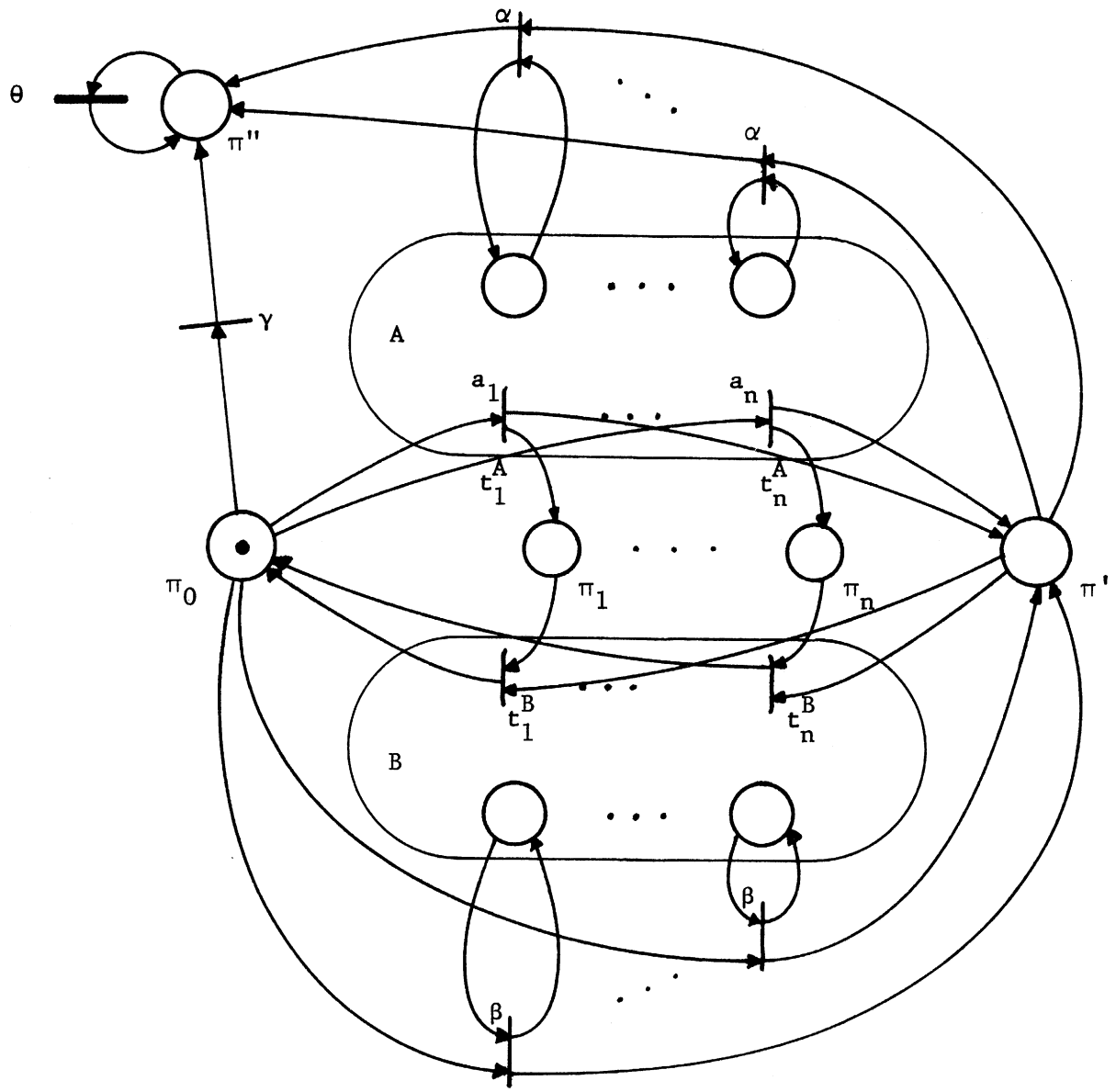


Figure 7.3

But we also know that the Reachability Problem is equivalent to the emptiness problem for terminal Petri net languages, since labelling does not matter for the purpose of Theorem 7.1, reachability of the zero marking in a Petri net A is clearly an instance of the equivalence problem $\mathcal{L}^f(A) = \emptyset$. Thus:

Theorem 7.4: The equivalence and inclusion Problems for the family $\mathcal{L}_0^f$ of free terminal Petri net languages are recursively equivalent to the Reachability Problem.

In all fairness, it must be said that we only considered the zero marking as a terminal marking. But the construction of figure 7.3 can be modified, by methods such as are used in Hack [7,8], to handle arbitrary final markings. The distinction has to be made because free Petri nets usually do not have a free Standard Form. *)

Finally, let us mention the fact that the family $\mathcal{L}^f$ is weak enough, and the family $\mathcal{L}_0^\lambda$ powerful enough, so that the complementation closure of $\mathcal{L}^f$ is actually a subset of $\mathcal{L}_0^\lambda$. Given a free Petri net A , we can add λ-transitions and labelled transitions to construct a new net B such that $\mathcal{L}_0^\lambda(B) = \alpha^* - \mathcal{L}^f(A)$, where α is the alphabet of A . Without going into details, the construction works as follows: Petri net A can be stopped after having generated some string $w \in \mathcal{L}^f(A)$. Now control is transferred to a construction which "chooses" an arbitrary transition t in A . The label of t is generated by some other, new, transitions, and an arbitrary marking strictly less than F(t) is removed from the input places of t ; the other places of A are "cleared", as in paragraph 2.1.4. If t was not firable, this could clear all places of A , but if t was firable (and the label sequence generated so far was still in $\mathcal{L}^f(A)$), then some tokens must remain in the input places of t , and no zero marking can be reached. If we could reach the zero marking as described above, we reached it by generating a label sequence $wa \notin \mathcal{L}^f(A)$, which can then be followed by an arbitrary string from α^*, since $\mathcal{L}^f(A)$ has the prefix property. Thus:

Theorem 7.5: The complementation closure of $\mathcal{L}^f$ is in $\mathcal{L}_0^\lambda$.

---

*) A construction for the general case can be found in Hack (1976). (Added reference)

In the next chapter we shall see that in the general case, equivalence problems are undecidable, and that the complementation closure of the non-free Petri Net Languages would imply the undecidability of the Reachability Problem.

Note: In Hack [7, 8] we conjectured that the Reachability Problem is decidable, because it seems that if a marking M is reachable, then there exists a firing sequence shorter than $K \cdot |M|$ which reaches it, where K is a computable constant. If this is true, it is likely that workspace arguments could be used to show that $\mathcal{L}_0^{\lambda}$-languages are actually context-sensitive.

In the next chapter we shall see that in the general case, equivalence problems are undecidable, and that the complementation closure of the non-free Petri Net languages would imply the undecidability of the Reachability Problem.

Note: In Hack [7, 8] we conjectured that the Reachability Problem is decidable, because it seems that if a marking M is reachable, then there exists a firing sequence shorter than $K \cdot |M|$ which reaches it, where K is a computable constant. If this is true, it is likely that workspace arguments could be used to show that $\mathcal{L}_0^{\gamma}$-languages are actually context-sensitive.

8. Undecidable Equivalence Problems.

The first undecidability result related to Petri nets was obtained by M. Rabin in 1967 about Vector Addition Systems. In Petri net language, it concerns the problem of deciding whether, of two reachability sets, one is a subset of the other. This is called the Inclusion Problem for reachability sets (IP). We have recently shown that the Equality Problem for reachability sets (EP) is also undecidable (Hack [9]); these undecidability proofs are based on a reduction of Hilbert's Tenth Problem to the IP and the EP for reachability sets.

In this chapter*we show that a similar reduction can be carried out for the Equivalence Problems of the Petri net language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$.

8.1 The Polynomial Graph Inclusion Problem. (PGIP)

Let $P(x_1,\cdots,x_n)$ be a polynomial with non-negative integer coefficients. The corresponding Polynomial Graph $G(P)$ is defined to be: (Hack [9])

$$G(P) = \{ \langle x_1,\cdots,x_n,y\rangle \in \mathbb{N}^{n+1} \mid y \le P(x_1,\cdots,x_n) \}$$

The PGIP is the problem of deciding whether $G(P) \subseteq G(Q)$, where P and Q are two given polynomials with non-negative integer coefficients. This problem is undecidable, a proof of this fact appears in Hack [7, 9].

The undecidability of the IP and the EP for reachability sets is established by showing how reachability sets can encode polynomial graphs. In the next section we shall propose a suitable encoding in terms of Petri net languages of type $\mathcal{L}$ or $\mathcal{L}_0$. We already know that $\mathcal{L}_0^\lambda$-languages can encode reachability sets (see the proof of Theorem 7.2), so that the undecidability of the equivalence problem for $\mathcal{L}_0^\lambda$-languages follows from the undecidability of the EP for reachability sets.

8.2 Encoding of Polynomial Graphs as $\mathcal{L}$ or $\mathcal{L}_0$-languages.

A suitable language for encoding the graph $G(P)$ of a polynomial $P(x_1,\cdots,x_n)$ is:

$$L(P) = \{ a_1^{x_1}a_2^{x_2}\cdots a_n^{x_n}b^y \mid y \le P(x_1,\cdots,x_n) \} \quad .$$

This language has the prefix property and is thus a suitable candidate for an $\mathcal{L}$-language. Since $\mathcal{L}_0$-languages don't contain the empty string, a suitable $\mathcal{L}_0$-language would be $L(P)-\{\lambda\}$ or $c\cdot L(P)$. We shall show that Petri nets can be constructed to generate these languages.

*) Hack (1976) contains a much simpler construction than this section. Also see the remark on page 93.

The construction is fairly complex, and will be presented in stages. We first build an elementary multiplier (8.2.1) from which we construct a multiple-input multiplier (8.2.2). Then we assemble these multipliers so as to weakly compute a polynomial (8.2.3), and finally we complete the design of a Petri net capable of generating the language L(P) described above (8.2.4).

8.2.1 The Elementary Multiplier A(x).

Our first objective is to build a Petri net A(x) whose initial marking contains x tokens in some "input" place, and whose language over an alphabet {a,b} is such that the length of the longest label sequence is proportional to the number of occurrences of the symbol b , the variable ratio being x .

Consider the labelled Petri net A(x) of figure 8.1 (disregard for the moment the dotted lines). It has an initial marking of $x \geq 1$ tokens in the "input" place p_1 (hence the name A(x)) and one token in p_3. There are six transitions labelled a $(a_1,\cdots,a_6)$ and six transitions labelled b . Inspection of the net shows that all firing sequences are prefixes of strings from the following regular language R; all terminal firing sequences (i.e. reaching the zero marking) are complete strings of R:

$$R = (b_1a_1{}^*b_2a_2{}^*)((b_3 + b_5a_3{}^*a_5) + (b_1a_1{}^*)(b_4 + b_6a_4{}^*a_6))$$

In fact, the a-transitions either shuttle tokens from p_1 to p_2 and back (a_1,a_2) or they empty p_1 or p_2 $(a_3,\cdots,a_6)$. There can be no more than x-1 consecutive a-firings between b-firings. Therefore, the $\mathcal{L}_0$ and $\mathcal{L}$-languages of A(x) can be characterized by the regular language R (or its prefixes) and the restrictions desribed in the table of figure 8.2 .

So it appears that:

$$\mathcal{L}_0(A(x)) \subseteq \mathcal{L}(A(x)) \subseteq (b + ba + \cdots + ba^{x-1})^*$$

If we limit the number of b-firings, the length of the longest firing sequence or the longest terminal firing sequence is x times the number of b-firings; the length of the shortest terminal firing sequence is x-1 plus the number of b-firings. This could be enforced by adding a new place p_7 (drawn in dotted lines in figure 8.1) initially marked with $y \geq 1$ tokens. Now the $\mathcal{L}$-language of the modified net contains strings up to a length of $x \cdot y$; the $\mathcal{L}_0$-language contains strings of any length between x+y-1 and $x \cdot y$. In this manner we shall be able to weakly compute products, polynomials, and finally encode polynomial graphs.

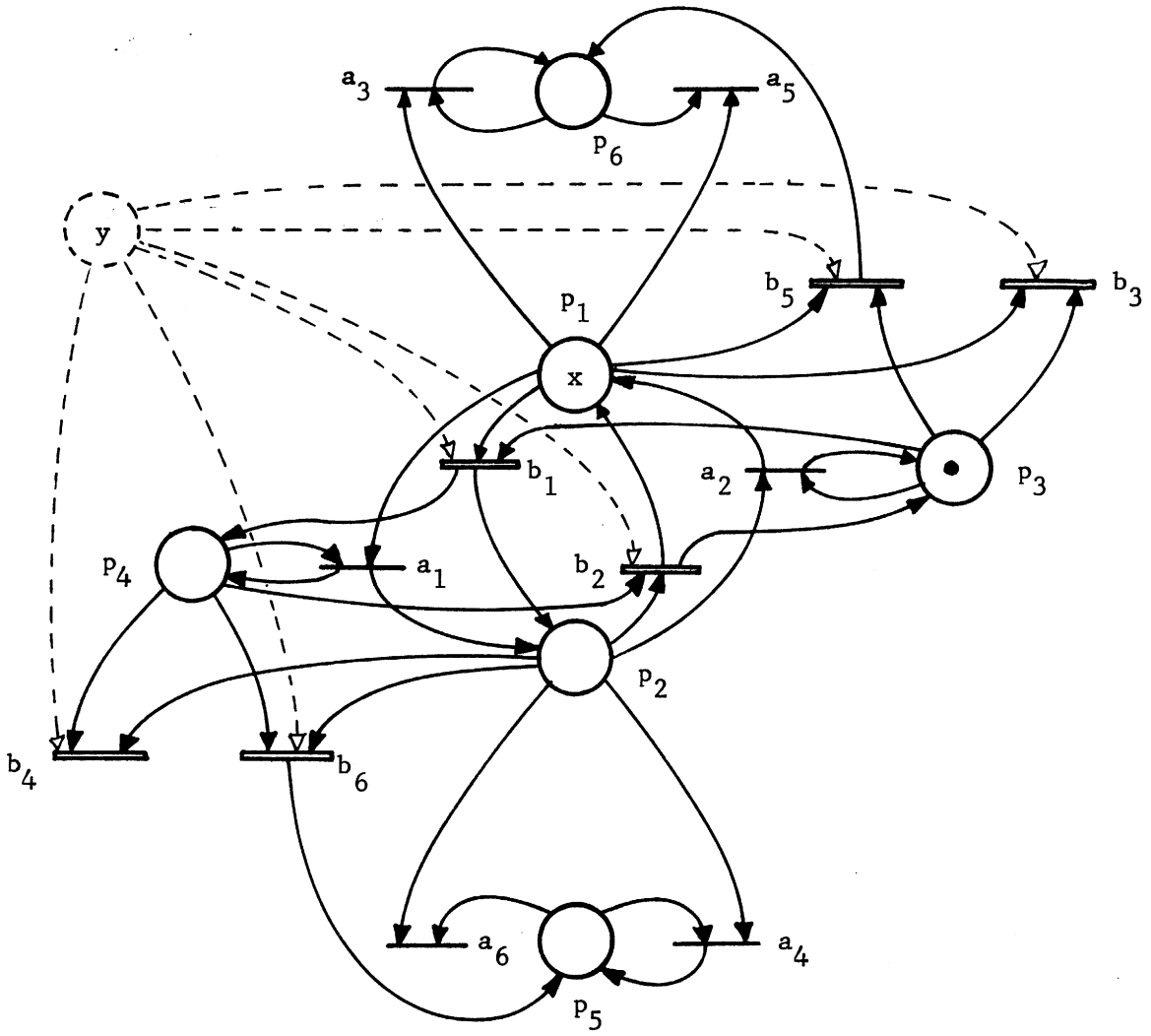


Figure 8.1

Language:	$\mathcal{L}(A(x))$	$\mathcal{L}_0(A(x))$
Strings selected from:	all prefixes of R	R
by the following restrictions:		
Number of successive firings of a_1 or a_2	$\leq x - 1$	$= x - 1$
Number of successive firings of a_3 or a_4	$\leq x - 2$	$= x \dot{-} 2$
Total excess of a_{odd}-firings over a_{even}-firings	$\leq x - 1$	$= x - 1$

Figure 8.2

8.2.2 The n-input Multiplier $B_n(x_1,\cdots,x_n)$.

Lemma 8.1: For each n, there exists a labelled Petri net $B_n(x_1,\cdots,x_n)$ with n distinguished input places initially marked with $x_1,\cdots,x_n$ tokens respectively, such that, for all $x_i \geq 1$:

$$\mathcal{L}(B_n(x_1,\cdots,x_n)) = \{ b^y \mid 0 \leq y \leq \prod_{i=1}^{n}(x_i) \}$$

$$\mathcal{L}_0(B_n(x_1,\cdots,x_n)) = \{ b^y \mid 1-n+\sum_{i=1}^{n}(x_i) \leq y \leq \prod_{i=1}^{n}(x_i) \}$$

Proof: by induction on n.

If $n=1$, the net $B_1(x_1)$ is the simple one-place net of figure 8.3:

$$\mathcal{L}(B_1(x_1)) = \{ b^y \mid 0 \leq y \leq x_1 \}$$
$$\mathcal{L}_0(B_1(x_1)) = \{ b^y \mid y = x_1 \}$$

Figure 8.3

If $n > 1$, assume that there exists a net $B_1(x_1,\cdots,x_n)$ which satisfies the conditions of the Lemma.

Consider the $\mathcal{L}$ or $\mathcal{L}_0$-language over the alphabet {a,b} defined by:

$$(a^* \parallel \mathcal{L}_{(0)}(B_n(x_1,\cdots,x_n))) \cap \mathcal{L}_{(0)}(A(x_{n+1}))$$

The left side of the intersection limits the number of b-firings in $\mathcal{L}_{(0)}(A(x_{n+1}))$ to y, where $y \leq \prod_{i=1}^{n}(x_i)$ and, in the case of $\mathcal{L}_0$, also $y \geq 1-n+\sum_{i=1}^{n}(x_i)$. Therefore, the lengths of the firing sequences in the above language are any number no greater than $x_{n+1} \cdot \prod_{i=1}^{n}(x_i)$, and in the case of $\mathcal{L}_0$, also no less than $x_{n+1}-1+(1-n+\sum_{i=1}^{n}(x_i))$.

If we now apply the relabelling function (h: b→b, a→b), we observe that:

$$h((a^* \parallel \mathcal{L}_{(0)}(B_n(x_1,\cdots,x_n))) \cap \mathcal{L}_{(0)}(A(x_{n+1}))) = \mathcal{L}_{(0)}(B_{n+1}(x_1,\cdots,x_{n+1}))$$

Since $\mathcal{L}$ and $\mathcal{L}_0$-languages are closed under concurrency, intersection, and relabelling, the corresponding constructions applied to $B_n(x_1,\cdots,x_n)$ and $A(x_{n+1})$ yield an appropriate labelled Petri net $B_{n+1}(x_1,\cdots,x_{n+1})$.

QED

We may note that the two-input weak multiplier described at the end of 8.2.1 is precisely the net $B_2(x,y)$.

8.2.3 Computing Polynomials.

Having shown how to compute multiple products (monomials), we proceed to show how to compute polynomials.

A polynomial is a sum of monomials: $P(x_1,\cdots,x_n) = \sum_{j=1}^{k} (M_j(x_1,\cdots,x_n))$.

Each monomial is of the form: $M_j(x_1,\cdots,x_n) = \alpha_j \cdot \prod_{i=1}^{n} (x_i^{\beta_{i,j}})$, $\alpha_j > 0$. $\beta_{i,j} \geq 0$

Some examples of such monomials are $2x_1x_2^2x_3$, or $2x_3$, or 5.

Now let us construct a labelled Petri net $C(P)$, as shown in figure 8.4 for the example $P(x_1,x_2,x_3) = 2x_1x_2^2x_3 + 3x_1x_2x_3 + 2x_3 + 5$. For each monomial M_j we include one copy of labelled Petri net B_ℓ , where $\ell = 1 + \sum_{i=1}^{n} \beta_{i,j}$, i.e. one input per factor, including the coefficient α_j .

Then we add n places (call them $p_1 \cdots p_n$) and $2n$ transitions (call them $t_1 \cdots t_{2n}$), where t_{2i-1} and t_{2i} ($1 \leq i \leq n$) are labelled a_i (not to be confused with the set of transitions of figure 8.1). These elements are connected as shown in figure 8.4. Each a_i-labelled transition deposits one token in each x_i-input place of the monomial subnets. The last transition, t_{2n}, also deposits all the coefficients (and also the constant, or zero-degree monomial), as well as the initial markings for the B-nets. This guarantees that no B-net can start to fire before t_{2n} has fired.† Therefore, all firing sequences of $C(P)$ are of the form:

$$t_1^* t_2 t_3^* t_4 \cdots t_{2n-1}^* t_{2n} X$$

where X is the concurrent composition of firing sequences from the B-nets . Since all B-nets have a language over the single letter b , the language of $C(P)$ consists of strings of the form:

$$a_1^{x_1} a_2^{x_2} \cdots a_n^{x_n} b^y \quad , \quad \text{where} \quad y \leq P(x_1,\cdots,x_n)$$

In particular, we have:

$$\mathcal{L}(C(P)) = \{ \text{ prefixes of } a_1 a_1^* \cdots a_n a_n^* \} \cup \{ a_1^{x_1} \cdots a_n^{x_n} b^y \mid x_i \geq 1 \ \& \ 0 \leq y \leq P(x_1,\cdots,x_n) \}$$

$$\mathcal{L}_0(C(P)) = \{ a_1^{x_1} \cdots a_n^{x_n} b^y \mid x_i \geq 1 \ \& \ Q(x_1,\cdots,x_n) \geq y \geq P(x_1,\cdots,x_n) \}$$

where $Q(x_1,\cdots,x_n)$ is a linear (first degree) polynomial expressing the minimum number of firings necessary to get rid of all the tokens accumulated by the a-firings.

† Actually, the internal initial tokens of the B-nets can be present at the initial marking, since as long as at least one input place is unmarked, no b-transition can fire. In this case, depositing the coefficients by t_{2n} switches the B-nets on.

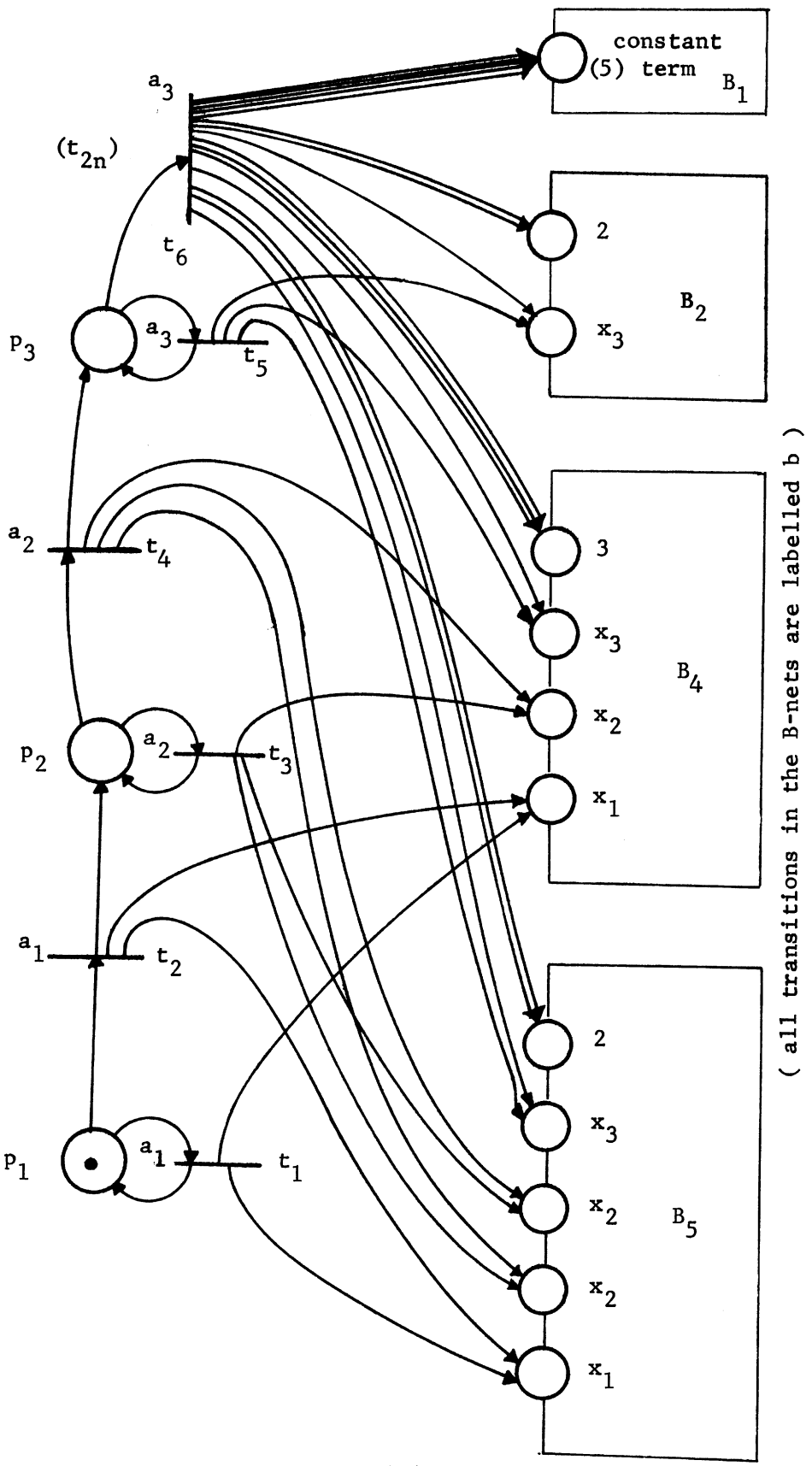

constant
(5) term
B_1
a_3
(t_{2n})
t_6
2
x_3
B_2
p_3
a_3
t_5
a_2
t_4
3
x_3
x_2
x_1
B_4
p_2
a_2
t_3
a_1
t_2
2
x_3
x_2
x_2
x_1
B_5
p_1
a_1
t_1
(all transitions in the B-nets are labelled b)

Figure 8.4

For a monomial M_j of the form described above we have, in fact (from the $\mathcal{L}_0$-language of the corresponding B-net) :

$$Q_j = \alpha_j + \sum_{i=1}^{n} \beta_{i,j}(x_i - 1)$$

Thus:

$$Q(x_1,\cdots,x_n) = \sum_{j=1}^{k}(Q_j)$$

The polynomial illustrated in figure 8.4 is

$$P(x_1,x_2,x_3) = 2x_1x_2^2x_3 + 3x_1x_2x_3 + 2x_3 + 5$$

which gives us: $$Q(x_1,x_2,x_3) = 2x_1 + 3x_2 + 3x_3 + 4$$

So far, it appears that we have encoded the graph of polynomial P with the exception of certain regions close to the origin: The hyperplanes where one or several variables are equal to zero, and, in the case of $\mathcal{L}_0$, the region below the hyperplane described by the first degree equation $y = Q(x_1,\cdots,x_n)$. We will produce these missing regions in the next paragraph.

8.2.4 Completion of the Encoding Petri Net.

Let us call positive polynomial graph $G^+(P)$ of a polynomial P the subset of G(P) where all variables are positive:

$$G^+(P) = \{ \langle x_1,\cdots,x_n,y\rangle \mid \forall i: x_i \geq 1 \ \& \ y \leq P(x_1,\cdots,x_n) \}$$

Then the languages generated by the Petri net C(P) constructed so far can be rewritten as:

$$\mathcal{L}(C(P)) = \{ \text{prefixes of } a_1a_1^*\cdots a_na_n^*\} \cup G^+(P)$$

$$\mathcal{L}_0(C(P)) = G^+(P) - G^+(Q - 1) \text{ , where } Q \text{ is of first degree.}$$

In the case of $\mathcal{L}$, it is sufficient take the union of the encodings of positive polynomial graphs of the polynomials resulting from the substitution of zero for one or more variables in P , retaining the alphabet $\{a_1,\cdots,a_n\}$ and the pairing $a_i^{x_i}$ as used in the $\mathcal{L}$-encoding of $G^+(P)$. Since $\{$ prefixes of $a_1a_1^*\cdots a_na_n^*\} \subseteq \{$ prefixes of $a_1^*\cdots a_n^*\} \subseteq G(P)$, the union of all these encodings is precisely G(P) .

In the case of $\mathcal{L}_0$, we shall first show that we can encode $G^+(P)$. Then the technique used above for $\mathcal{L}$ can be used to encode $G(P) - \langle 0,\cdots,0\rangle$, by means of the language $L(P) - \lambda$, which is thus shown to be $\mathcal{L}_0$. It is then easy to show that all of G(P) can be encoded by $c \cdot L(P)$, for example.

So it remains to be shown how to encode $G^+(P)$, where Q is of degree one.

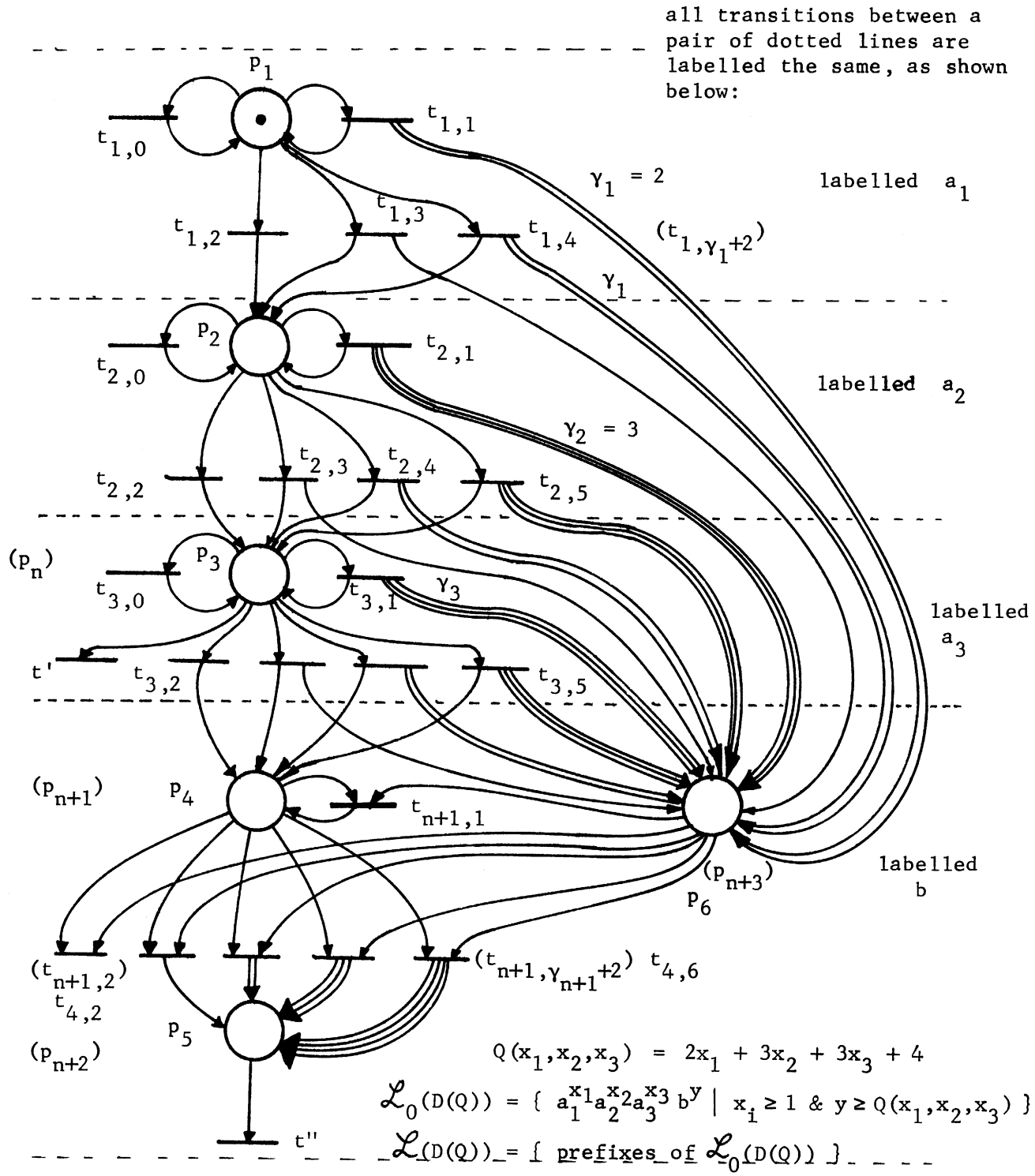

all transitions between a pair of dotted lines are labelled the same, as shown below:
p_1
$t_{1,0}$
$t_{1,1}$
$t_{1,2}$
$t_{1,3}$
$t_{1,4}$
$\gamma_1 = 2$
(t_{1,γ_1+2})
γ_1
labelled a_1
p_2
$t_{2,0}$
$t_{2,1}$
$t_{2,2}$
$t_{2,3}$
$t_{2,4}$
$t_{2,5}$
$\gamma_2 = 3$
labelled a_2
(p_n)
p_3
$t_{3,0}$
$t_{3,1}$
γ_3
t'
$t_{3,2}$
$t_{3,5}$
labelled a_3
(p_{n+1})
p_4
$t_{n+1,1}$
(p_{n+3})
p_6
labelled b
$(t_{n+1,2})$
$t_{4,2}$
$(t_{n+1,\gamma_{n+1}+2})$
$t_{4,6}$
p_5
(p_{n+2})
t''
$Q(x_1,x_2,x_3) = 2x_1 + 3x_2 + 3x_3 + 4$
$\mathcal{L}_0(D(Q)) = \{ a_1^{x_1}a_2^{x_2}a_3^{x_3} b^y \mid x_i \geq 1 \ \& \ y \geq Q(x_1,x_2,x_3) \}$
$\mathcal{L}(D(Q)) = \{ \text{prefixes of } \mathcal{L}_0(D(Q)) \}$

Figure 8.5

We shall construct a Petri net $D(Q)$ such that:

$$\mathcal{L}_0(D(Q)) = \{ a_1^{x_1} \cdots a_n^{x_n} b^y \mid (\forall i: x_i \geq 1) \ \& \ y \leq Q(x_1,\ldots,x_n) \}$$

Let $Q(x_1,\cdots,x_n) = \sum_{i=1}^{n} (\gamma_i \cdot x_i) + \gamma_{n+1}$. The net (figure 8.5) contains $n+1$ one-place stages (places $p_1 \cdots p_{n+1}$) and two counting places p_{n+2} and p_{n+3}. Initially, only p_1 is marked -- with one token. Each stage i, $1 \leq i \leq n$, contains γ_i+3 transitions $t_{i,j}$, where $0 \leq j \leq \gamma_i+2$. All these transitions are labelled a_i. Transitions $t_{i,0}$ and $t_{i,1}$ self-loop on place p_i; transitions $t_{i,2} \cdots t_{i,\gamma_i+2}$ transfer one token from p_i to p_{i+1}. In addition, $t_{i,1}$ deposits γ_i tokens into p_{n+3}, and transitions $t_{i,j}$ $(2 \leq j \leq \gamma_i+2)$ deposit $j-2$ tokens into p_{n+3}. Stage n has one extra a_n-labelled transition t' which simply removes the token from p_n. The firing sequences permitted so far are of the form $(t_{1,0} + t_{1,1})^*(t_{1,2} + \cdots + t_{1,\gamma_1+2}) \cdot (t_{2,0} + t_{2,1})^*(t_{2,2} + \cdots + t_{2,\gamma_2+2}) \cdot \ \ldots \ \cdot (t_{n,0} + t_{n,1})^*(t_{n,2} + \cdots + t_{n,\gamma_n+2} + t')$, corresponding to a label sequence $a_1^{x_1} \cdots a_n^{x_n}$. The marking resulting from such a firing sequence consists of z tokens in p_{n+3} and one or zero tokens in p_{n+1}, where z can have any value between 0 and $Q(x_1,\cdots,x_n) - \gamma_{n+1}$.

Stage $n+1$ has $\gamma_{n+1}+2$ transitions $t_{n+1,j}$ $(1 \leq j \leq \gamma_{n+1}+2)$, all labelled b. All these transitions take one token from each of p_{n+1} and p_{n+3}, although $t_{n+1,1}$ returns a token to p_{n+1} in a self-loop. In addition, $t_{n+1,j}$ $(2 \leq j \leq \gamma_{n+1}+2)$ also deposits $j-2$ (i.e. anywhere between 0 and γ_{n+1}) tokens into counter p_{n+2}. Finally, there is a transition t'', also labelled b, which removes tokens one by one from p_{n+2}. This portion of the net becomes active after the firing of the a_i-labelled transitions. In order to reach the zero marking, transition $t_{n+1,1}$ must remove all z tokens except one from p_{n+3}; then $t_{n+1,j}$ $(2 \leq j \leq \gamma_{n+1}+2)$ removes the last token from p_{n+3} and must be followed by $j-2$ additional b-firings of t'', which takes care of the constant term γ_{n+1} of the polynomial Q. The total number of b-firings is thus y, where $z \leq y \leq z+\gamma_{n+1}$, so that $0 \leq y \leq Q(x_1,\cdots,x_n)$, as intended.

We have thus proved (remember that $\mathcal{L} \subseteq \mathcal{L}^\lambda$ and $\mathcal{L}_0 \subseteq \mathcal{L}_0^\lambda$) :

Theorem 8.1: The Petri net language families $\mathcal{L}, \mathcal{L}_0, \mathcal{L}^\lambda, \mathcal{L}_0^\lambda$ can all encode polynomial graphs, by means of the language $L(P)$ defined at the beginning of this section.

Remark 1: The constructions we have used in this section are quite large and complicated; the number of transitions required by this method to generate $L(P)$ grows roughly like an exponential function of the degree or the number of variables of the polynomial P. It is possible to use a different encoding of incomplete polynomial graphs (with a linear subset missing) which is sufficient to establish the undecidability results, and which requires much smaller constructions.

Remark 2: For generating L(P) as an $\mathcal{L}$-language, the additional constructions of 8.2.4 are not required. But it may be observed that the $\mathcal{L}$-language of the net resulting from the complete construction is also L(P), so that the same net can be used to generate L(P) as an $\mathcal{L}$-language or as an $\mathcal{L}_0$-language, up to λ of course.

8.3 The Undecidability Results.

In section 8.1 we indicated that it is undecidable whether $G(P) \subseteq G(Q)$ for given polynomials P and Q with non-negative integer coefficients. From section 8.4 it follows that for a given polynomial P there exists a Petri net N(P) such that:

$$\mathcal{L}(N(P)) = L(P) = \{ a_1^{x_1} \cdots a_n^{x_n} b^y \mid y \leq P(x_1, \cdots, x_n) \}$$

$$\mathcal{L}_0(N(P)) = L(P) - \{\lambda\}$$

The following propositions are clearly equivalent:

$$G(P) \subseteq G(Q)$$

$$L(P) \subseteq L(Q)$$

$$L(P) - \lambda \subseteq L(Q) - \lambda$$

$$\mathcal{L}(N(P)) \subseteq \mathcal{L}(N(Q))$$

$$\mathcal{L}_0(N(P)) \subseteq \mathcal{L}_0(N(Q))$$

Therefore, the inclusion problems for $\mathcal{L}$ and $\mathcal{L}_0$, and hence $\mathcal{L}^\lambda$ and $\mathcal{L}_0^\lambda$, -languages are undecidable. Since all four families are closed under union, and since $L(P) \subseteq L(Q) \Leftrightarrow L(P) \cup L(Q) = L(Q)$, it follows that the corresponding equivalence problems are also undecidable:

Theorem 8.2: The inclusion and equivalence problems for the Petri net language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ are undecidable.

An interesting consequence of this theorem concerns the closure of Petri net language families under complementation. It is clear that neither $\mathcal{L}$ nor $\mathcal{L}^\lambda$ are closed under complement, because the complement of either contains no prefix languages. In the strict sense, $\mathcal{L}_0$ is not closed under complementation either, since each language whose complement is in $\mathcal{L}_0$ contains λ. So let us use the convention that $\overline{\mathcal{L}_0} = \{ L - \{\lambda\} \mid (\alpha^* - L) \in \mathcal{L}_0 \}$, where α is the alphabet of L. Now, since $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$ are closed under intersection, we observe that closure under complement ($\overline{\mathcal{L}_0} \subseteq \mathcal{L}_0$) implies, for any two $\mathcal{L}_0$ (or $\mathcal{L}_0^\lambda$) -languages L_A and L_B, the

existence of an $\mathcal{L}_0$ (or $\mathcal{L}_0^\lambda$) -language $L_C = L_A \cap (\mathcal{A}^* - L_B)$, and that $L_C = \emptyset \Leftrightarrow L_A \subseteq L_B$, so that the inclusion problem would be reducible to the emptiness problem (assuming of course that the complement of a language can be effectively determined), and hence reducible to the Reachability Problem (Theorem 7.1). In fact, it is sufficient if the complementation closure of any of the four families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ is a subset of $\mathcal{L}_0^\lambda$. Thus:

Theorem 8.3: If the complementation closure of any of the four Petri net language families $\mathcal{L}$, $\mathcal{L}_0$, $\mathcal{L}^\lambda$, $\mathcal{L}_0^\lambda$ is a subset of $\mathcal{L}_0^\lambda$, then the Reachability Problem is undecidable (assuming effective complementation).

Now we shall investigate to what degree the language generated by a Petri net depends on the structure of the net. We shall see that the dependency is quite sensitive to minor changes, which may have unpredictable effects.

Consider the labelled Petri net of figure 8.6. It contains two components A and B which are standard form generators for $\mathcal{L}(A)$ and $\mathcal{L}(B)$, or for $\mathcal{L}_0(A)$ and $\mathcal{L}_0(B)$. Let p_1 and p_2 be the respective "start" places for A and B. These places are connected to a one-token (initially) place p_3 by t_1 and t_2, both labelled c, where c is a new symbol, not in the alphabet of A or B. We have clearly:

$$\mathcal{L}(C) = c \cdot (\mathcal{L}(A) \cup \mathcal{L}(B)) \cup \{\lambda\}$$

$$\mathcal{L}_0(C) = c \cdot (\mathcal{L}_0(A) \cup \mathcal{L}_0(B))$$

Let $(C - t_2)$ be the designation of the Petri net obtained from C by removing transition t_2. Then we have:

$$\mathcal{L}(C - t_2) = c \cdot \mathcal{L}(A) \cup \{\lambda\}$$

$$\mathcal{L}_0(C - t_2) = c \cdot \mathcal{L}_0(A)$$

Hence:

$$\mathcal{L}(C) = \mathcal{L}(C - t_2) \Leftrightarrow \mathcal{L}(B) \subseteq \mathcal{L}(A)$$

$$\mathcal{L}_0(C) = \mathcal{L}_0(C - t_2) \Leftrightarrow \mathcal{L}_0(B) \subseteq \mathcal{L}_0(A)$$

From the undecidability of the inclusion problem it follows that:

Theorem 8.4: It is undecidable whether the addition or removal of a transition changes the language generated by a Petri net.

Now consider figure 8.7. Again, we have two components A and B in standard form, connected by c-labelled transitions t_1 and t_2 to a one-token place p_3 (The

new "start" place). In addition, we have a place p_4 from which t_1 and t_2 require one token, but p_1 may also receive a token from p_3 alone via transition t_3, which is also labelled c. Call this net D_0 if p_4 is not marked, and D_1 if p_4 is initially marked with one token; in both nets, p_3 has one token initially. So D_1 differs from D_0 only in the additional token of p_4. Thus:

$$\mathcal{L}(D_0) = c \cdot \mathcal{L}(A) \cup \{\lambda\} \qquad \mathcal{L}(D_1) = c \cdot (\mathcal{L}(A) \cup \mathcal{L}(B)) \cup \{\lambda\}$$

$$\mathcal{L}_0(D_0) = c \cdot \mathcal{L}_0(A) \qquad \mathcal{L}_0(D1) = c \cdot (\mathcal{L}_0(A) \cup \mathcal{L}_0(B))$$

As in the previous case, if we could test whether D_0 and D_1 generate the same language, we could decide whether the A-language contains the B-language. Thus:

Theorem 8.5: It is undecidable whether adding or removing one token from the initial marking will change the language generated by a labelled Petri net. (The same goes for adding or removing a place).

Now consider the Petri net of figure 8.8. It is the result of the union construction of $\mathcal{L}_{(0)}(A)$ and $\mathcal{L}_{(0)}(B)$, as described in section 3.3, plus an extra place called "hold" which receives a token from every "first" or "singleton" transition of B; this token cannot disappear. Let us call this net E. We have:

$$\mathcal{L}(E) = \mathcal{L}(A) \cup \mathcal{L}(B)$$

$$\mathcal{L}_0(E) = \mathcal{L}_0(A)$$

From remark 2 at the end of paragraph 8.2.4, we know that we may choose a Petri net A such that, for a given polynomial P, we have:

$$\mathcal{L}(A) = \mathcal{L}_0(A) \cup \{\lambda\} = L(P)$$

Similarly, let $\mathcal{L}(B) = L(Q)$ for another given plynomial Q. We observe that:

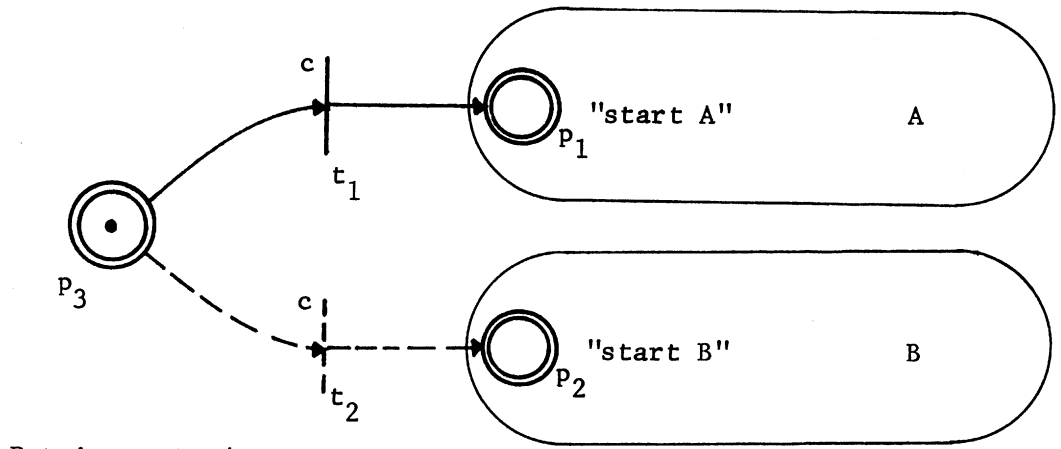
$$\mathcal{L}(E) = L(P) \cup L(Q)$$

$$\mathcal{L}_0(E) \cup \{\lambda\} = L(P)$$

$$\mathcal{L}(E) = \mathcal{L}_0(E) \cup \{\lambda\} \Leftrightarrow L(P) \cup L(Q) = L(P)$$

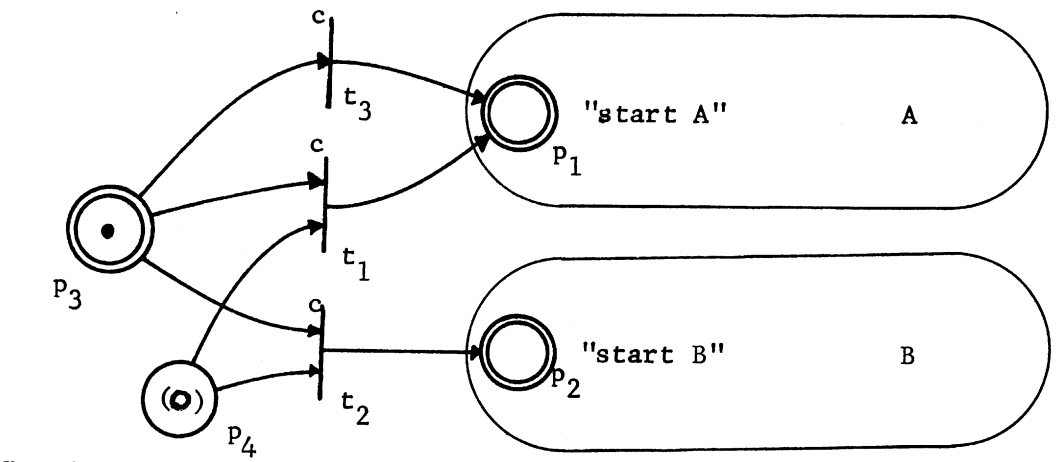
It follows that:

Theorem 8.6: It is undecidable whether every non-empty label sequence generated by a given labelled Petri net is also a terminal label sequence of that Petri net.



Petri net C (or $C - t_2$).

Figure 8.6



Petri net D_0 or D_1, depending on token in p_4.

Figure 8.7

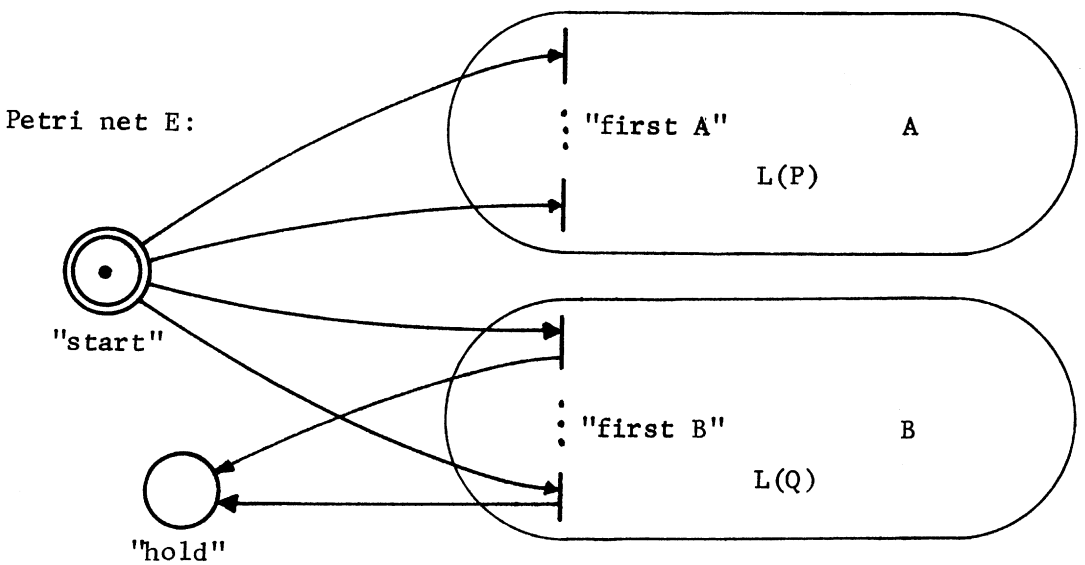
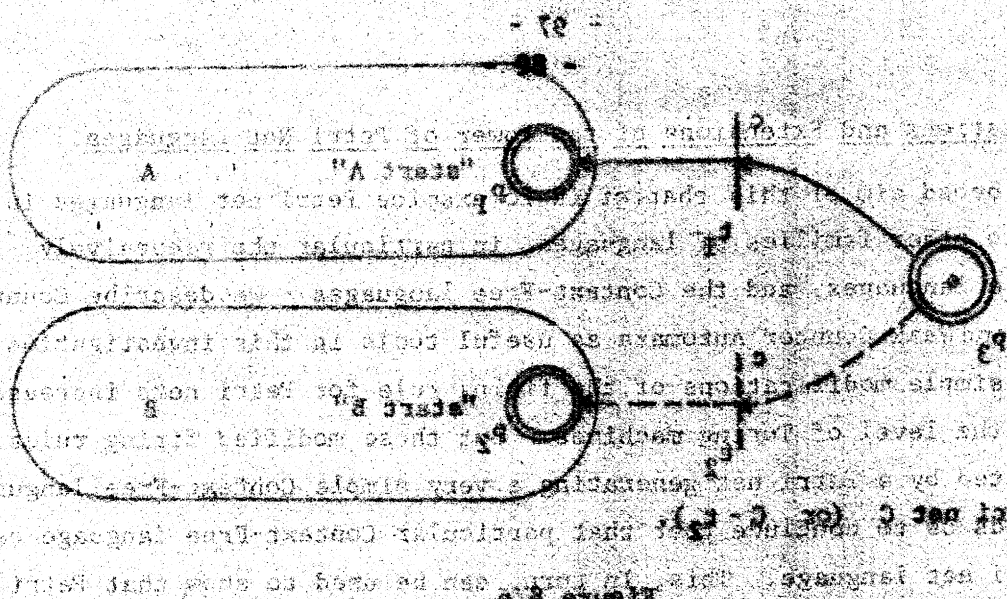
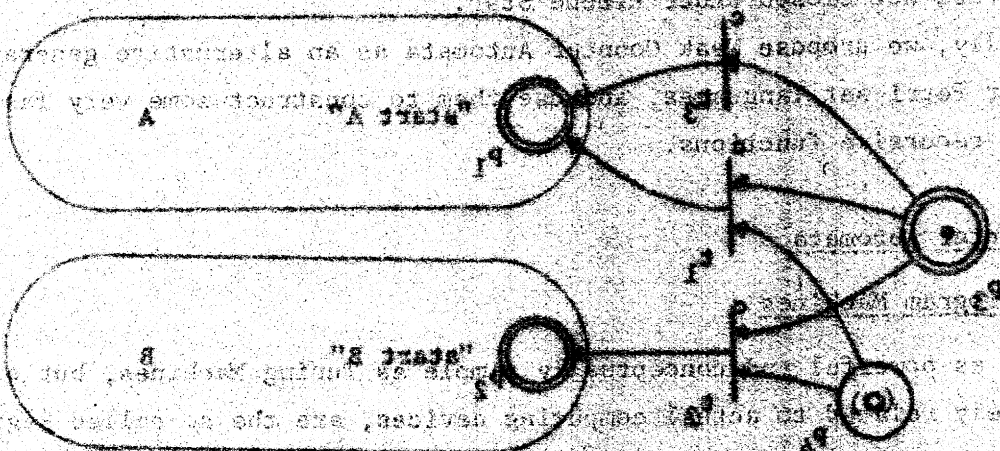


Figure 8.8



Petri net C (or C_2)

Figure 8.6



Petri net D or D_2, depending on token in p_4.

Figure 8.7

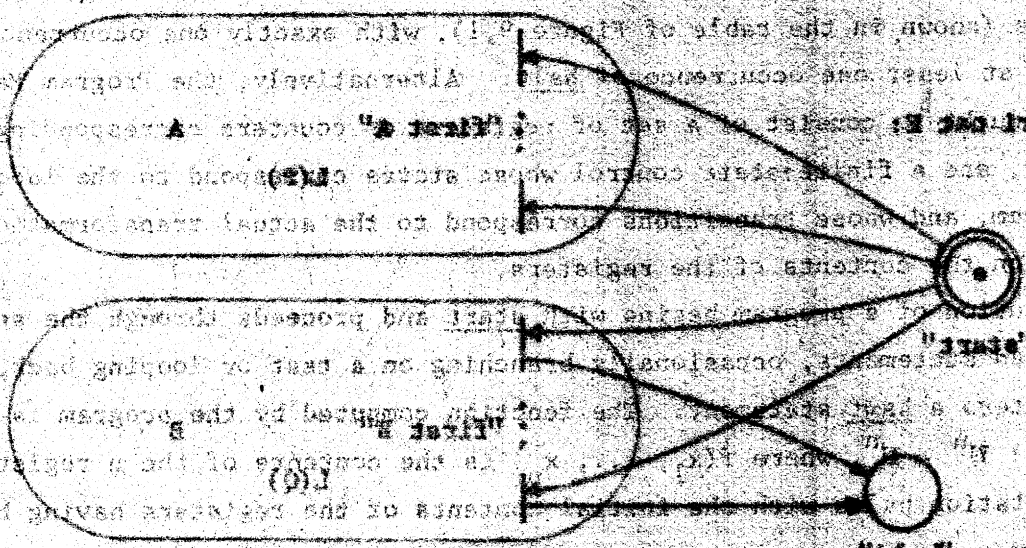


Figure 8.8

9. Limitations and Extensions of the Power of Petri Net Languages.

The broad aim of this chapter is to examine Petri net languages in comparison to other families of languages, in particular the recursively enumerable languages, and the Context-Free languages. We describe Counter Automata and Weak Counter Automata as useful tools in this investigation.

Two simple modifications of the firing rule for Petri nets increase their power to the level of Turing machines. But these modified firing rules could be simulated by a Petri net generating a very simple Context-Free language, which leads us to conclude that that particular Context-Free language cannot be a Petri net language. This, in turn, can be used to show that Petri net languages are not closed under Kleene Star.

Finally, we propose Weak Counter Automata as an alternative generating device for Petri net languages, and use them to construct some very fast growing primitive recursive functions.

9.1. Counter Automata

9.1.1. Program Machines

Just as powerful and conceptually simple as Turing Machines, but structurally more closely related to actual computing devices, are the so-called Register Machines (Shephardson and Sturgis [20]) or Program Machines (Minsky [15]).

Briefly, a Program Machine consists of a finite set of variables $\{x_i \mid 1 \leq i \leq n\}$ and a program, which is a finite list of labelled statements of four types (shown in the table of Figure 9,1), with exactly one occurrence of start and at least one occurrence of halt. Alternatively, the Program Machine may be thought to consist of a set of registers or counters corresponding to the variables, and a finite-state control whose states correspond to the labels in the program, and whose transitions correspond to the actual transformation carried out on the contents of the registers.

Execution of a program begins with start and proceeds through the sequence of labelled statements, occasionally branching on a test or looping back, until it encounters a halt statement. The function computed by the program is a mapping $f: \mathbb{N}^n \rightarrow \mathbb{N}^n$, where $f(x_1, \ldots, x_n)$ is the contents of the n registers if the computation halts with the initial contents of the registers having been $x_1, \ldots, x_n$.

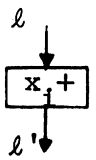
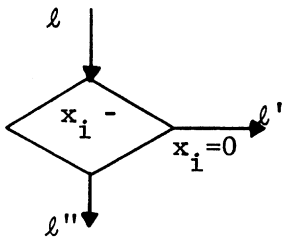
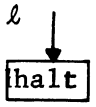
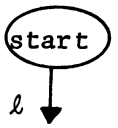
Statement	Symbol	Name
ℓ: $x_i \leftarrow x_i + 1$; goto ℓ'		increment
ℓ: if $x_i = 0$ then goto ℓ' else $x_i \leftarrow x_i - 1$; goto ℓ''		test and decrement
ℓ: halt		halt
: start ℓ		start

Figure 9.1

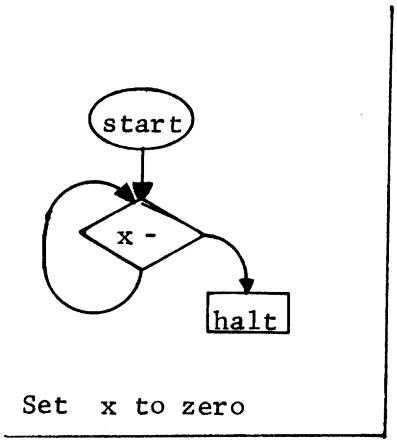
Minsky has shown [15] that for every partial recursive function $\psi: \mathbb{N} \rightarrow \mathbb{N} \cup \{\text{undefined}\}$ there exists a two-variable program machine such that $f(2^x,0) = \langle 2^{\psi(x)},0\rangle$ if ψ is defined for x, and such that the program machine never halts if $\psi(x)$ is undefined†. In particular, it is undecidable whether a given Program Machine will halt when started in some given initial configuration.

Figure 9.2 shows some simple examples which show how more complex operations can be built up from the limited repertoire of Program Machine statements. Such sub-programs can of course be chained (by merging start of one and halt of another sub-program).

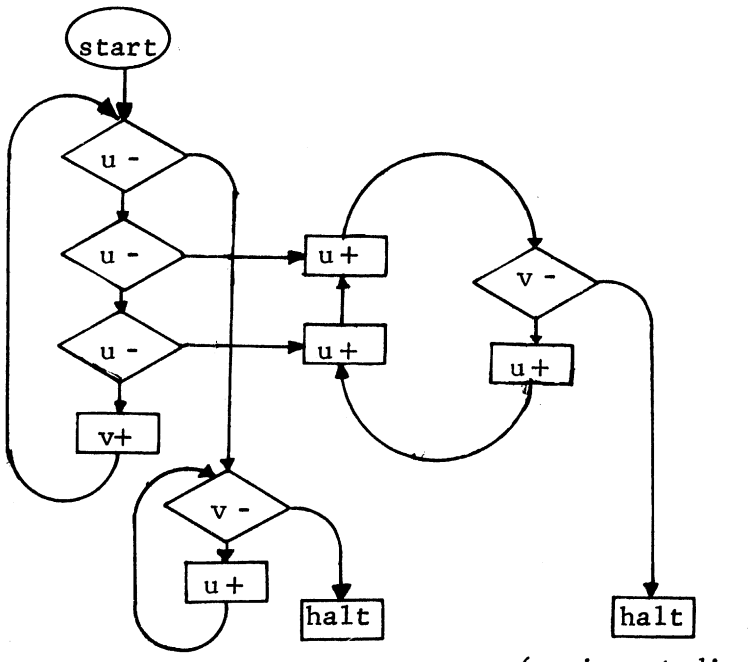
---

†To compute $\psi(x)$ directly, three variables are required (Schroeppel [19]); these can be encoded in the following way, which yields the result above:

Three-variable machine: (x,y,z)	Two-variable machine: (u,v)	
$f(x,0,0) = \langle \psi(x),0,0\rangle$	$u = 2^x 3^y 5^z$	$f(u,0) = \langle 2^{\psi(\log_2(u))},0\rangle$
	v : scratch	provided u is a power of 2.



Set x to zero



(u ← u/3)

(u is not divisible by 3)

At right:

Conditional divide-by-three, useful when encoding into a two-counter machine. Scratch variable v is assumed to be zero at the beginning of the subprogram, and is reset to zero at the end. If u is not divisible by 3 it is returned unchanged. The two halt statements are used for the branching of control.

Figure 9.2

9.1.2. Language-Generating Counter Automata

The Program Machines described so far are purely arithmetic; they can recognize or generate languages over an arbitrary alphabet only via some encoding. In this section we shall add symbol-generating (or recognizing) instructions to the instruction repertoire of Program Machines.

A deterministic recognizer could be obtained by adding a statement such as

$$\ell\text{: } \underline{\text{read}}\ a;\ \underline{\text{goto}}\ \ell' \quad \underline{\text{else}}\ \ \underline{\text{goto}}\ \ell''$$

whose intended meaning is: read the symbol on the input tape; if it is a, advance the head and go to ℓ'; otherwise go to ℓ''. The program would be started with read head at the beginning of an input tape and all registers at zero; acceptance could be defined as halting with all registers at zero.

But we are mainly interested in a language generator, i.e. a non-deterministic device which can print any string from a language-to-be-generated, and no other strings. (Such a generator can usually be trivially transformed into a non-deterministic acceptor or recognizer.)

We propose two new statement types to do the job. These statements are shown in the table of Figure 9.3:

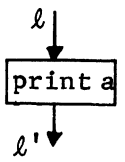
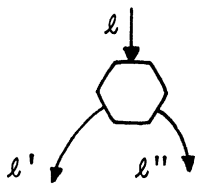
Statement	Symbol	Name
ℓ: print a; goto ℓ'		print
ℓ: goto ℓ' or ℓ''		choice

Figure 9.3

There is a print statement for each symbol from some alphabet. The choice statement introduces non-determinism. Now we define a Counter Automaton:

Definition 9.1: (a) A Counter Automaton consists of a finite set of counters $\{x_i \mid 1 \leq i \leq n\}$ all containing zero initially, a finite alphabet α, and a program consisting of labelled statements of the following types:

- increment (one per counter)
- test & decrement (one per counter)
- start
- halt
- print (one per symbol in α)
- choice

(b) The language generated by the Counter Automaton is the set of all strings printed by some halting computation (there may be several due to non-determinism) from the initial all-zero counter configuration.

Definition 9.2: A Deterministic Counter Automaton is like a Counter Automaton, except that it contains no choice statements, and that there is no restriction on its initial counter configuration. It expresses a partial function from $\mathbb{N}^n$ to $\mathcal{A}^*$ associating the string generated by a halting computation to the argument, which consists of the initial counter contents.

For a given initial configuration, a Deterministic Counter Automaton produces a definite output string, if it halts. The language of a Deterministic Counter Automaton could be defined as the range of the function it expresses.

Let us now show that every type 0 language (recursively enumerable) can be generated by some Counter Automaton. We know that every recursively enumerable language can be encoded into a recursively enumerable set. There are many ways to encode a string over an alphabet $\mathcal{A} = \{a_1, \ldots, a_m\}$ into an integer. Let us choose (m + 1)-ary representation, defined recursively as follows:

$$e: \mathcal{A}^* \to \mathbb{N} : \begin{cases} e(\lambda) = 0 \\ \left.\begin{array}{l} \forall w \in \mathcal{A}^* \\ \forall a_i \in \mathcal{A} \end{array}\right) e(a_i \cdot w) = i + (m+1) \cdot e(w) \\ (1 \le i \le m) \end{cases}$$

For example, if $\mathcal{A}$ contains the nine numerals 1 ... 9 in that order, then the code of a string of numerals is the decimal number spelled out by the string read from right to left, i.e. e(string 123) = number 321. We could of course have used the more "natural" order, but it is easier to decode least-significant-digit first.

This encoding is not onto. Non-zero numbers containing the numeral 0 in m+1-representation are not codes, but this eliminates the worry about leading zeros. But the encoding function is clearly one-one (different strings map into different integers) and thus has an inverse, the decoding function $e^{-1}: \mathbb{N} \to \mathcal{A}^*$, which is partial recursive.

As a matter of fact, the Deterministic Counter Automaton of Figure 9.4 precisely expresses the decoding function. If started with the initial configuration $\langle x, y \rangle$, it halts after printing the string $e^{-1}(x)$ if x is a code number, otherwise it does not halt.

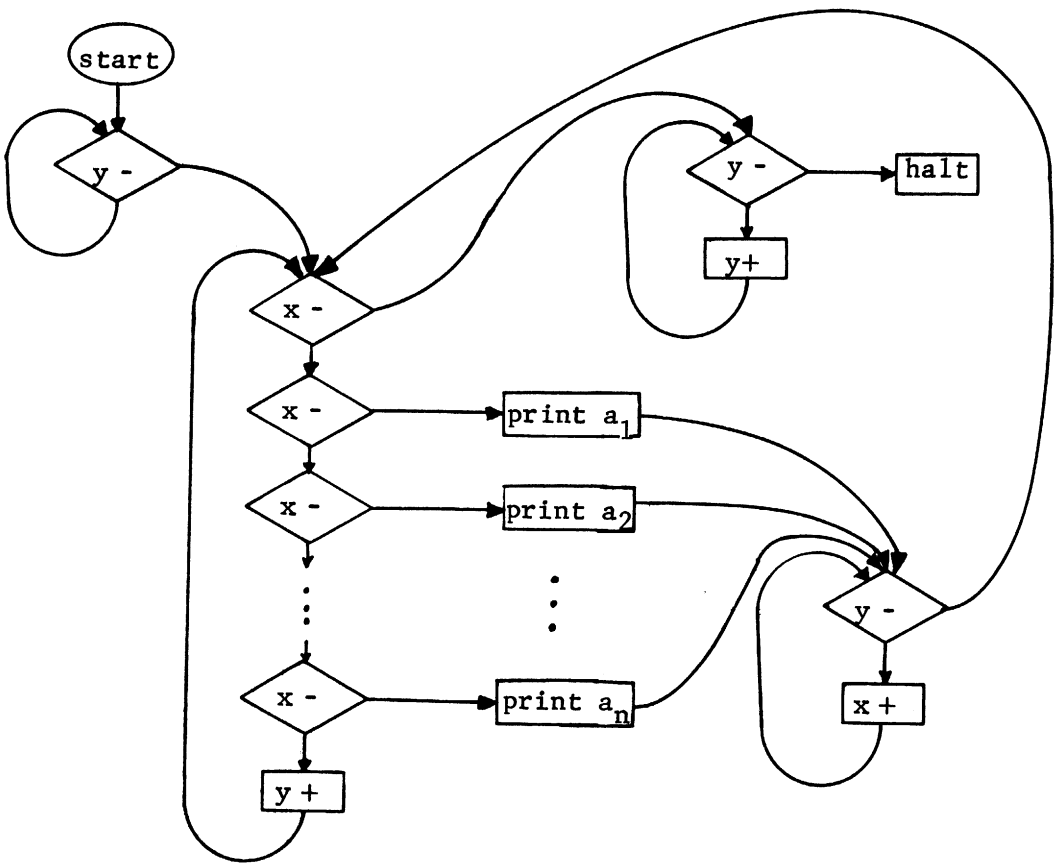


Figure 9.4

Since every recursively enumerable language is the image of some recursively enumerable set under the decoding function e^{-1}, and since every recursively enumerable set is the range of some partial recursive function ψ, which in turn can be computed by a 3-register Program Machine, it appears that every recursively enumerable language is the range of some Deterministic Counter Automaton as shown in Figure 9.5. This construction uses three registers, but we may reduce this to two registers by using the coding $u = 2^x 3^y 5^z$ and replacing increment and decrement on x,y,z by multiply and conditional divide by 2,3,5 on the new variable u, which can easily be done using the second new variable v as a scratch variable. The flow of control between these subroutines implementing the original statements will be the same as far as the print statements are concerned, so that the generated language will not be affected. We should also insure that arguments which are not exact powers of 2 will cause the program not to halt; this construction is also not difficult.

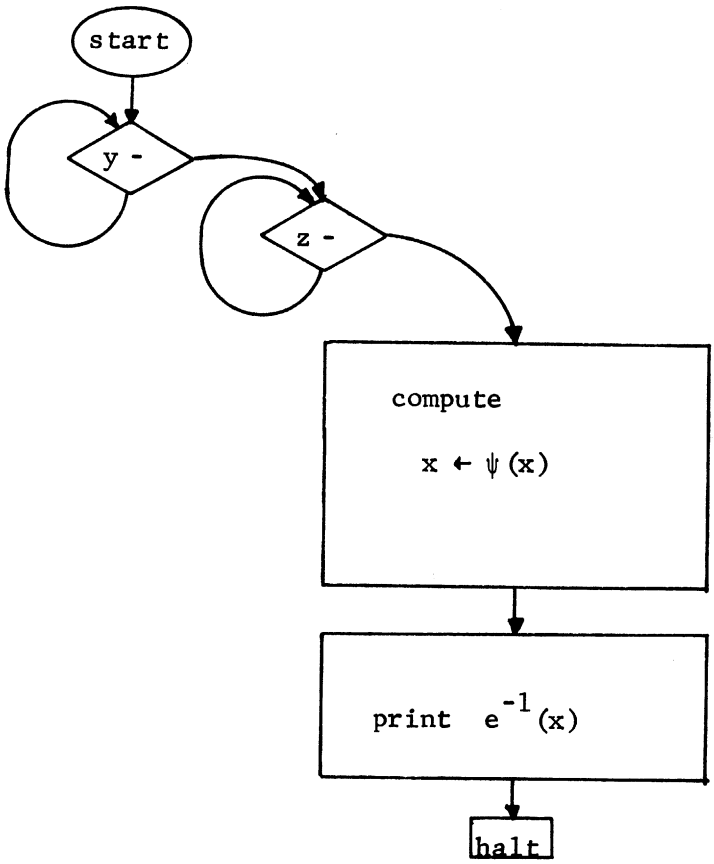


Figure 9.5

Finally, a simple non-deterministic Counter Automaton can be used to generate an arbitrary value for x. Together with the automaton of Figure 9.5, this yields the Counter Automaton of Figure 9.6 whose language is range $(\psi \circ e^{-1})$, as desired. If we use a two-register machine, we could of course directly generate an arbitrary power of 2 for u; this is easy, but not necessary if arguments not a power of two are rejected anyway.

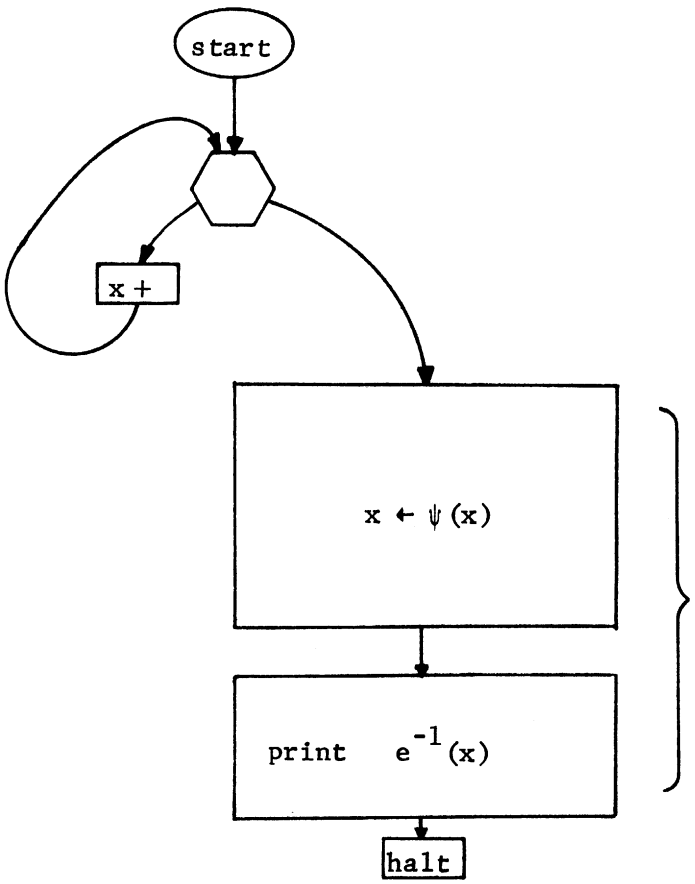


Figure 9.6

Thus:

Theorem 9.1: (a) Every type 0 (recursively enumerable) language is the range of some two-register Deterministic Counter Automaton.

(b) Every type 0 language can be generated by some (non-deterministic) two-register Counter Automaton.

9.2. Inhibitor Arcs and Priority Firing Rules

9.2.1. Inhibitor Nets

It is apparent that all operations carried out by Counter Automata can be simulated by Petri nets (having a finite-state component corresponding to the program and a number of buffer places corresponding to the counters) except for the crucial zero-testing ability. Many people including this author have felt for a long time that if Petri nets could in some way react to the absence of a token (in an unbounded place, otherwise there is no problem), then they could simulate Turing machines, via Program Machines. However, R. M. Keller [12] and S. Kosaraju [13] showed that this cannot be done. M. Flynn and T. Agerwala [4] introduced zero testing (inhibiting) arcs in a modified Petri net model to solve certain synchronization problems beyond the power of Petri nets; they observed that this modification was "quite powerful". Agerwala [1] later proved that they had indeed reached the power of Turing machines. In this section we will show how Petri nets augmented by inhibitor arcs can directly represent counter automata.

Definition 9.3: An inhibitor arc is a special kind of arc. It has no size (i.e. carries no tokens), and is only directed from a place to a transition. Graphically, it has a small circle instead of an arrowhead. The firing rule for transitions having incident inhibitor arcs is modified as follows: The transition is firable if and only if it was firable in the old sense and all inhibiting places (having inhibitor arcs directed at the transition) have zero tokens. Petri nets with inhibitor arcs are called Inhibitor Nets.

Figure 9.7 shows how the various statements of a Counter Automaton can be replaced by corresponding portions of an Inhibitor net. Each portion will have an input place p_ℓ corresponding to the label ℓ of the statement, and (except for halt), one or two output places $p_{\ell'}$, $p_{\ell''}$. In addition, there is a place π_i for each variable (counter) x_i. When interconnecting the portions, places bearing the same name are identified. For the purpose of generating a language, all transitions except those for print statements are unlabelled.

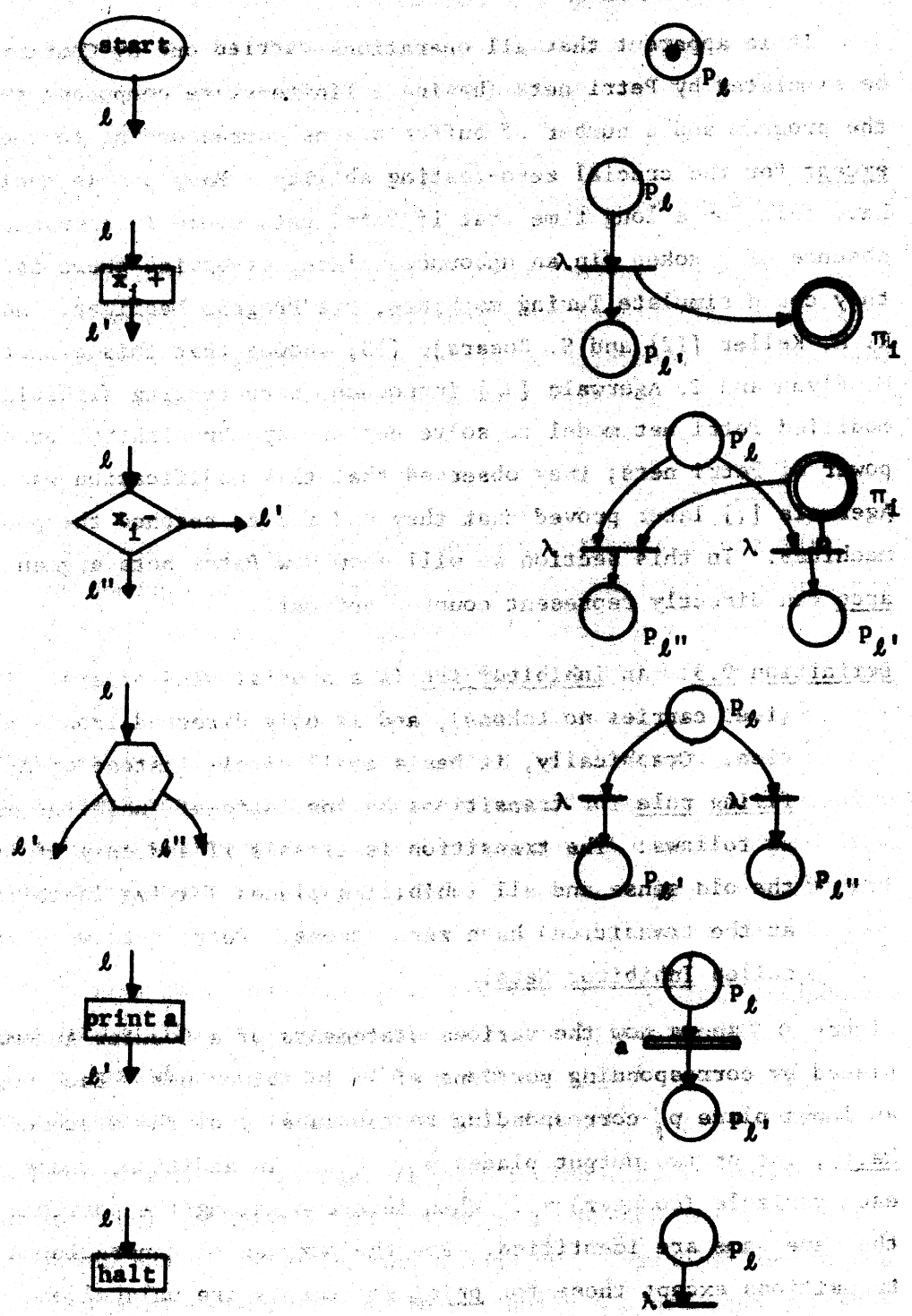


Figure 9.7

The correspondence is trivial; and Inhibitor Nets are at least as powerful as Counter Automata and Turing Machines, and can at most be more convenient. Let us use the convention that a Counter Automaton only halts when all counters contain zero. This can always be arranged by preceding every halt statement with a counter-clearing routine. Then we can say:

Theorem 9.2: Labelled Inhibitor Nets can effectively generate all recursively enumerable languages as $\mathcal{L}_0^{\lambda}$-languages.

Remark: The $\mathcal{L}^{\lambda}$-languages generated by labelled Inhibitor Nets are simply the prefixes of all recursively enumerable languages. Some thought will show that the languages generated by λ-free labelled Inhibitor Nets must still be recursive. We shall see that they must in fact be context-sensitive.

9.2.2 Priority Nets

The synchronization problems which Petri nets are unable to solve usually contain priority considerations such as: "if both queues are non-empty, queue 1 is to be served first". This notion of priority turns out to be just as strong as the notion of inhibitor arcs. Consider the Inhibitor Net replacement for the test and decrement statement (Figure 9.7). If we remove the inhibitor arc and assign priorities to the transitions such that the transition removing a token from π_i has a higher priority than the other transition, the effect will be the same: If both are firable, decrement takes place and control is transferred to $p_{\ell'}$; if π_i is empty, control is transferred to $p_{\ell''}$.

So let us define:

Definition 9.4: A Priority Net is a Petri net with a priority partial ordering of the transitions. The firing rule is modified to the extent that if several transitions are enabled at a given marking, only those whose priority is no less than any other enabled transition can fire.

Because of this definition, it is useful to distinguish between enabledness (which does not depend on priority) and firability (which requires enabledness and proper priority). We used a partial order in the definition, since this actually represents a situation often encountered in modelling parallel systems. If the order is total, the firing sequence will be monogenic, and no non-deterministic system (such as Counter Automata) could be modelled.

Since the translation of a Counter Automaton into a labelled Priority Net is essentially identical to that for Inhibitor Nets, we have:

Theorem 9.3: Labelled Priority Nets can effectively generate all recursively enumerable languages as $\mathcal{L}_0^{\lambda}$.

It is easy to establish the following consequences of the undecidability of the halting problem for Program Machines:

Theorem 9.4: The boundedness Problem and the Reachability Problem for Inhibitor Nets and for Priority Nets are undecidable.

9.2.3. Converting Inhibitor Nets Into Priority Nets

Both Inhibitor Nets and Priority Nets can, of course, be translated into Counter Automata. Any of the methods used to simulate Petri nets (with slightly modified firing rules) on a computer will do. Actually, the structure of Counter Automata is quite appropriate for this job. For example, a simple (non-prompt) approach is to first select an arbitrary transition (using choice), then test whether it is firable (by simulating the firing rule, taking whatever priorities into account), and accordingly either fire it or choose again.

But converting Inhibitor Nets into Priority Nets via Counter Automata is clumsy, introduces λ-transitions, and usually does not preserve structural transparency, which is important when we are modelling parallel systems, for example.

Now let us describe a λ-free method for transforming an Inhibitor Net into a Priority Net. Suppose the Inhibitor Net has n places $p_1 \ldots p_n$ and m transitions $t_1 \ldots t_m$. As a first step, let us assign incomparable priorities to all transitions. A priority will be a vector of positive integers, and the partial order will be the $\leq$ relation for vectors. The transformation will be progressive, place by place, and the priority partial order will be progressively refined by adding new coordinates (vector concatenation). Thus $\langle 1, 2\rangle$ has higher priority than $\langle 2, 2\rangle$ because $\langle 1, 2\rangle \leq \langle 2, 2\rangle$, whereas $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$ are incomparable. Vectors with different dimensions will also be considered to be incomparable. The initial priority assignment will be $Pr(t_j) = \langle j, m+1-j\rangle$, for $1 \leq j \leq m$.

At each step i we replace place p_i by two places p_i^+ and p_i^0, and we consider all transitions connected to p_i. The transitions are transformed as shown in Figure 9.8. Each transition becomes one or two transitions, labelled like the original transition and with the same priority, except in one case, where the priority is refined.

The transformation is such that place p_i^+ always has the same marking as p_i in the original net, and p_i^0 contains one token if and only if p_i^+ is empty; the initial marking is translated accordingly. We note that the number of transitions may increase significantly, as some are doubled, quadrupled, or multiplied even more. This can be moderated by only transforming places from which no inhibitor arcs originate.

We leave it to the reader to verify that the translation produces the desired effect: A Priority Net that has the same $\mathcal{L}$ or $\mathcal{L}_0$-language as the given Inhibitor Net, except that the standard final marking is one token in every p_i^0-place (this can be further transformed by the methods of Section 2.2).

9.2.4. Converting Priority Nets Into Inhibitor Nets

This conversion can also be carried out without introducing λ-transitions. As a first step we transform the net by the methods of Sections 5.8 and 4.1 into a Restricted Petri Net. (See Theorem 5.8.) This may create several new transitions corresponding to each original transition; each such new transition will have the same priority (and the same label) as the corresponding original transition. It can be verified that at any marking at most one of the new transitions corresponding to a given original transition will be firable. In other words, everything a Priority Net can do can also be done by a Priority RPN. We shall thus only transform these simpler Priority Nets (recall: no multiple arcs, no self-loops) into Inhibitor Nets.

Our objective is to enforce the firing constraints imposed by the priority rule. Let us define:

$$hpr(t) = \{t' \mid \text{transition } t' \text{ has strictly higher priority than } t\}$$

$$I(t') = \{p \mid p \text{ is an input place of } t', \text{ i.e. } F(p, t') = 1\} \quad \dagger)$$

Now we have:

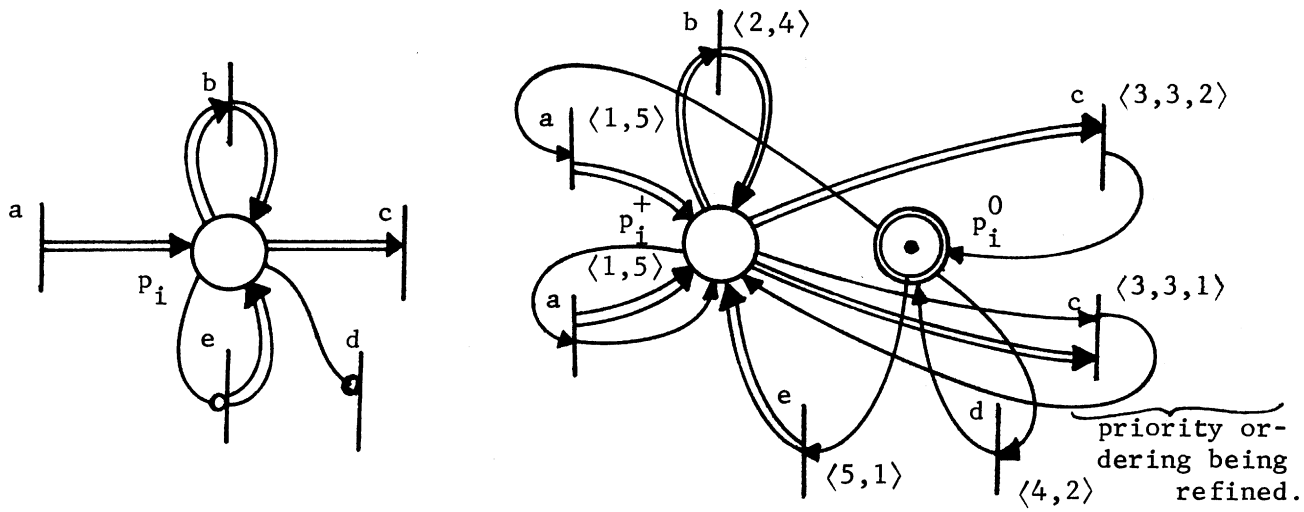
(t is firable) $\Leftrightarrow$ (t is enabled) & (no t' with higher priority is enabled)

In other words, at a given marking M:

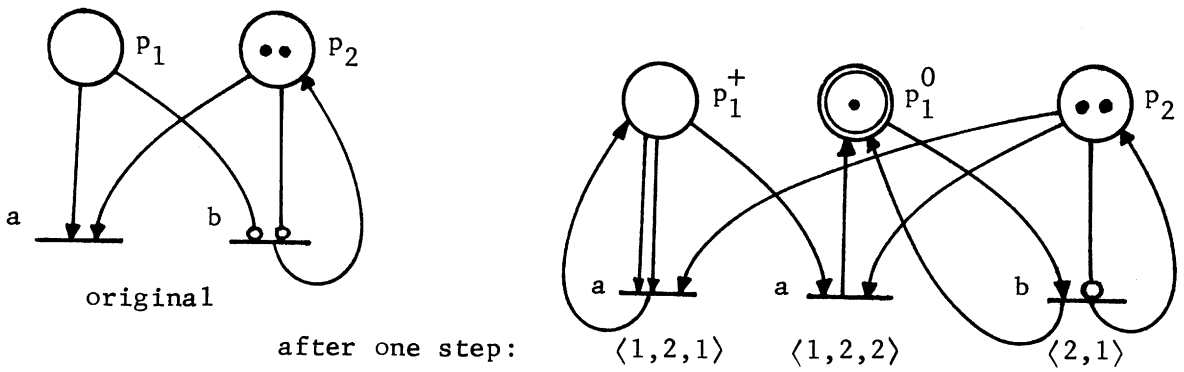
$\dagger)$ In Hack [5,6] the set of input places I(t) is denoted by $\cdot t$.

Transformation of Inhibitor Nets into Priority Nets:

1) An example involving the five possible transition types.



2) An example of the successive transformation of two places.



after two steps:

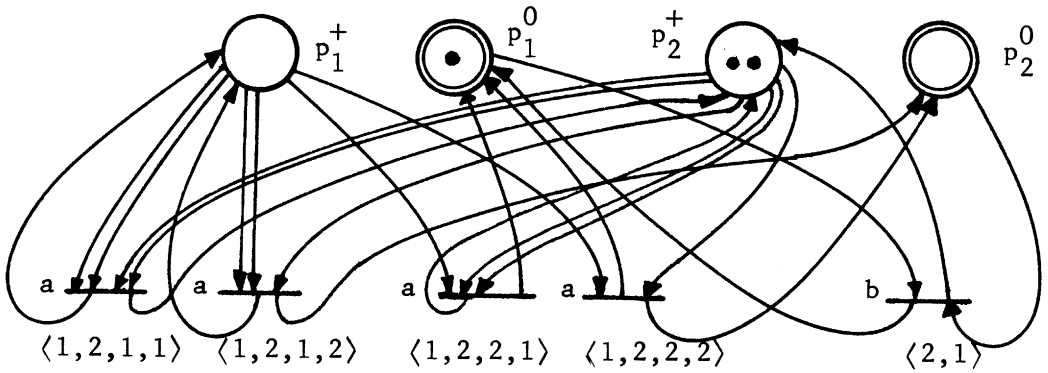


Figure 9.8

(t is firable) $\Leftrightarrow$ (t is enabled) & (t has priority)

where:

$$(t \text{ has priority}) \Leftrightarrow \underset{\forall t' \in hpr(t)}{AND} \left(\underset{\forall p \in I(t')}{OR} (M(p)=0) \right)$$

This condition can be rewritten in disjunctive normal form. Each disjunct expresses one possibility where t has priority. So we create as many copies of transition t as there are disjuncts (minterms). Each disjunct is of the form:

$$(M(p) = 0 \quad \& \quad M(p') = 0 \ldots \& \ M(p'') = 0)$$

This firability condition is expressed by drawing an inhibitor arc from each place p . . . p" in this disjunct to the copy of t corresponding to this disjunct.

After carrying out this operation for each transition, all priority rules will be enforced by the inhibitor arcs.

From this (9.2.4), and the previous section (9.2.3) we may conclude:

Theorem 9.5: Inhibitor Nets and Priority Nets can be transformed into each other in a λ-free, language-preserving manner.

9.2.5. Characterization of the Languages Generated by λ-Free Inhibitor Nets or Priority Nets

In the previous section we have seen that Restricted Petri nets are sufficient in the case of Priority Nets. Transforming a Priority RPN into an Inhibitor Net may at most introduce self-loops of the kind shown in Figure 9.9, where part of it is an inhibitor arc:

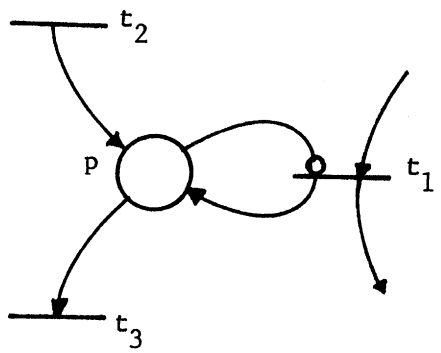


Figure 9.9

But such self-loops can be eliminated in a λ-free, language preserving manner. Place p is replaced by four places $\hat{p}$, p^+, p' and p^0. The last three are indicator places sharing one token; they are used to describe whether p holds zero (token in p^0), one (p') or more than one (p^+) tokens in the original net (Figure 9.9). The marking of $\hat{p}$ in the transformed net is one less than that of p in the original net. The transformed net appears in Figure 9.10. Each input or output transition (such as t_2) of p (except those in a self-loop on p) is transformed into three copies (t_2^+, t_2', t_2^0). A firing of such a transition is rendered in the transformed net by a firing of the appropriate copy. A transition in a self-loop on p can fire only if p has no token and puts a token on p. This is precisely what is simulated in the transformed net.

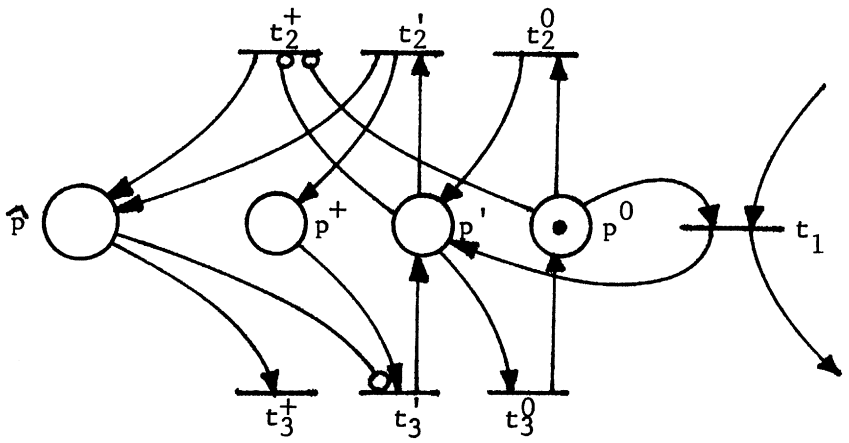


Figure 9.10

Thus:

Theorem 9.6: Inhibitor Nets and Priority Nets can be transformed in a λ-free, language preserving manner into Restricted (i.e. RPN) Inhibitor Nets or Restricted Priority Nets (Priority RPN's).

Now, Restricted Inhibitor Nets consist of an interconnection of closed subnets of the form shown in Figure 9.11, whose language can be obtained by finite substitution from that of the net of Figure 9.12 (recall Section 5.8).

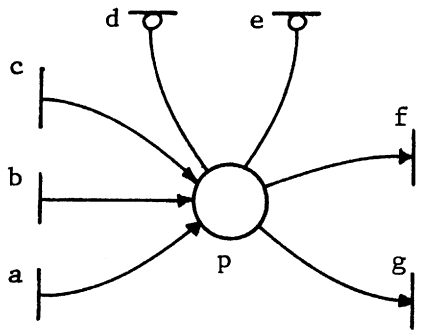


Figure 9.11

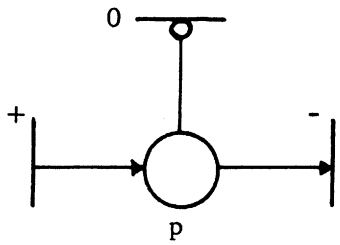


Figure 9.12

The $\mathcal{L}_0$-language[†] Q_0 of the net in Figure 9.12 over the alphabet {+, -, 0} is seen to be:

$$Q_0 = (P_0 \cdot 0)^* \cdot P_0$$

The $\mathcal{L}$-language Q of this net are all the prefixes of Q_0:

$$Q = (P_0 \cdot 0)^* \cdot P$$

The complete and incomplete parenthesis languages P_0 and P were defined in Section 5.2. We call the languages Q_0 and Q the complete and incomplete separated parenthesis languages. Q_0 and Q have very simple Context-Free Grammars:

Q_0:
- $S \to SOS$
- $S \to A$
- $A \to AA$
- $A \to +A-$
- $A \to \lambda$

Q:
- $S \to BC$
- $B \to BOB$
- $B \to A$
- $A \to AA$
- $A \to +A-$
- $A \to \lambda$
- $C \to CC$
- $C \to +C$
- $C \to +C-$
- $C \to \lambda$

By using the methods of Chapter 5, we get:

Theorem 9.7: (a) The $\mathcal{L}$ and $\mathcal{L}_0$-languages generated by λ-free Priority Nets or Inhibitor Nets are the closure under finite substitution, restriction (i.e. concurrency and intersection) and λ-free renaming of the language {-} and Q respectively Q_0.

(b) These languages are context-sensitive.

(c) These $\mathcal{L}_0$-languages yield all recursively enumerable (type 0) languages through erasure.

[†] actually, the cyclic language $\mathcal{L}_c$, as in section 5.8 .

Part (c) of this theorem is a consequence of Theorem 9.3 and is a novel characterization of the recursively enumerable languages. The closest classical characterization is as closure under homomorphism and intersection of Regular languages and Dyck-languages. The language Q_0 is related to a simple Dyck-language: Let h be the homomorphism:

$$h(+) = "("$$
$$h(-) = ")"$$
$$h(0) = "]["$$

Then the language $[h(Q_0)]$ is the subset of a Dyck language of two kinds of parentheses () and [], where the nesting depth of [] is limited to one (no nesting). A sample string is [(()())()][ ][(((()()))]].

9.3. Limits to the Power of Petri Net Languages

9.3.1. The Context-Free Language Q_0

In the previous section we have seen how a Priority Net or an Inhibitor net can be transformed into an interconnection of one-place Inhibitor subnets whose language is $S(Q_0)$ for some finite substitution S. If there existed a labelled Petri net whose $\mathcal{L}_0^\lambda$-language were Q_0, we could construct a net A whose $\mathcal{L}_0^\lambda$-language is $S(Q_0)$ (Section 4.5). Such a net could be coupled to the one-place closed subnet of Figure 9.11 as illustrated in Figure 9.13:

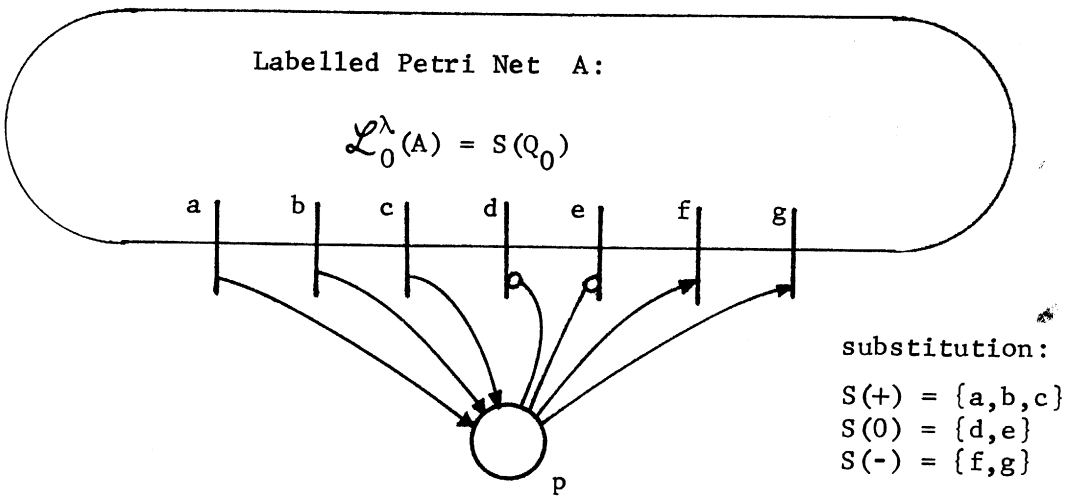


Figure 9.13

But now, Petri net A takes over the firing constraints of the inhibitor arcs from p to transitions d and e. After removing these inhibitor arcs, the Inhibitor Net of Figure 9.13 becomes just a plain labelled Petri net, and the interconnection of such closed subnets will be the original Inhibitor Net without the inhibitor arcs, but with added places and λ-transitions provided by nets such as A.

This Petri net simulates the original Inhibitor Net in the sense that the markings reachable in the original places are unchanged, and are reachable by the same firing sequences, disregarding the new λ-transitions. In particular, boundedness in the original places would now be decidable in the resulting Petri net, contradicting Theorem 9.4. Thus:

Theorem 9.8: The context-free language Q_0 is not in $\mathcal{L}_0^\lambda$, and can thus not be generated by any labelled Petri net.

9.3.2. Non-closure under Kleene Star.

We recall that the language Q_0 can be defined from P_0 by the Kleene Star: $Q_0 = (P_0 \cdot 0)^* \cdot P_0$. Since P_0 is in $\mathcal{L}_0$ (up to λ) and Q_0 is not in $\mathcal{L}_0^\lambda$, we conclude that neither $\mathcal{L}_0$ nor $\mathcal{L}_0^\lambda$ can be closed under Kleene Star. (Recall that both are closed under concatenation and that $\mathcal{L}_0 \subseteq \mathcal{L}_0^\lambda$.) Also remember that $a^*b \in \mathcal{L}_0$, which excludes closure under substitution by $\mathcal{L}_0$ or $\mathcal{L}_0^\lambda$. Thus:

Theorem 9.9: The Petri net language families $\mathcal{L}_0$ (completed by λ) and $\mathcal{L}_0^\lambda$ are not closed under Kleene Star, nor are they closed under arbitrary substitution.

9.3.3. The Languages of λ-Free Priority Nets and Inhibitor Nets.

Returning now to Priority Nets and Inhibitor Nets, we know that they can generate -- even without λ-transitions -- non-Petri net languages such as Q_0. A careful analysis of the constructions in Chapter 8 reveals that they can also generate the "surface" of polynomial graphs, i.e. languages of the form $\{a_i^{x_1} \ldots a_n^{x_n} b^y \mid y = P(x_1, \ldots, x_n)\}$, as opposed to the full graph $(y \leq P(x_1, \ldots, x_n))$ generated in $\mathcal{L}_0$. It is not known whether such languages are in $\mathcal{L}_0$ or $\mathcal{L}_0^\lambda$; if they are, then the Reachability Problem could easily be shown to be undecidable, using the methods of Chapter 8.

But we also know that λ-free Inhibitor Net languages must be context-

sensitive (Theorem 9.7b). Do we get all context-sensitive languages?

The answer is no. We still don't get even all context-free languages.

Peterson [17] has shown that CSS (and hence $\mathcal{L}_0$) cannot generate all palindromes over an alphabet with two or more symbols. His argument is that the number of distinct intermediate markings needed to generate palindromes of length n grows like an exponential function of n, whereas the number of distinct intermediate markings reachable via firing sequences of length n (remember, no λ-transitions!) grows like a power of n.

But Peterson's argument applies equally well to λ-free Priority Nets or λ-free Inhibitor nets. A characteristic feature of languages generated by λ-free nets, whether Priority or not, is that the number of distinct markings reachable during the generation of a string of length n grows like a polynomial function of n.

Thus:

Theorem 9.10: λ-free Priority Nets (or Inhibitor Nets) cannot generate the context-free language of palindromes over a two-symbol alphabet.

Note 1: These remarks seem to indicate that the languages generated by λ-free Priority Nets are closely related, if not identical, to log-space recognizable languages for Turing machines with a work tape and a one-way read-only input tape.

Note 2: It is interesting to observe the relative power of $\mathcal{L}_0$ and the Context-Free languages. Both are a subfamily of the Context-Sensitive languages, and both contain the regular languages. (Peterson has also shown that CSS, and hence $\mathcal{L}_0$ up to λ, contains all Bounded Context-Free languages [17].) They differ essentially in the fact that Context-Free language generators have one pushdown stack with several symbol types, whereas Petri nets have several "stacks", each with only one symbol type: indistinguishable tokens. The combination of the two properties immediately gives us the power of Turing machines.

9.4 Weak Counter Automata.

The only feature of a Counter Automaton which Petri nets cannot simulate without modifying the firing rule is the zero-testing capability. So let us drop this feature to get a new kind of automaton, which we call a Weak Counter Automaton.

Definition 9.5: A Weak Counter Automaton (WCA) is a Counter Automaton where the branching implied by the test&decrement is rendered non-deterministic by prefixing each test&decrement statement with a choice statement, as illustrated in figure 9.14.

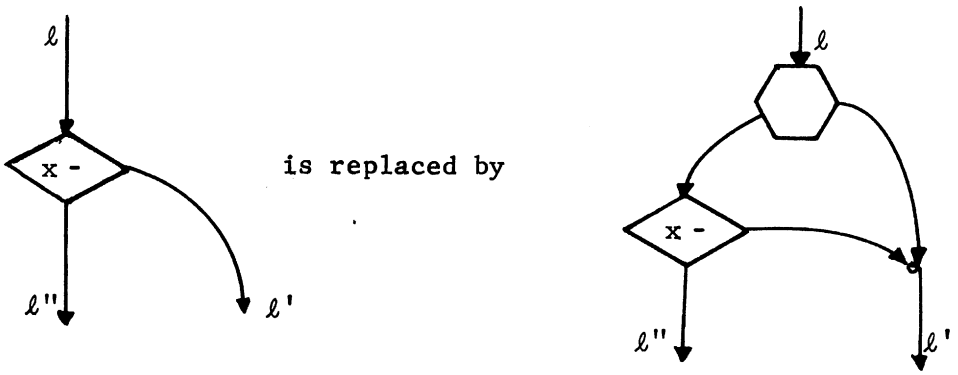


Figure 9.14

Unlike Counter Automata, Weak Counter Automata cannot usually be made to halt only when all counters contain zero. On the other hand, we can always modify a WCA such that, if there exists a halting computation, then there exists a zero-reaching halting computation. We take this into account in:

Definition 9.6: The language generated by a WCA consists of the strings printed by halting zero-reaching computations.

Now we observe that the translation of a WCA into an Inhibitor Net according to the rules illustrated in figure 9.7 always introduces inhibitor arcs in the context shown in figure 9.15a, which yields the Inhibitor Net shown in figure 9.15b. The Petri net of figure 9.15c, which has no inhibitor arcs, clearly performs the same coordination as the Inhibitor Net of figure 9.15b:

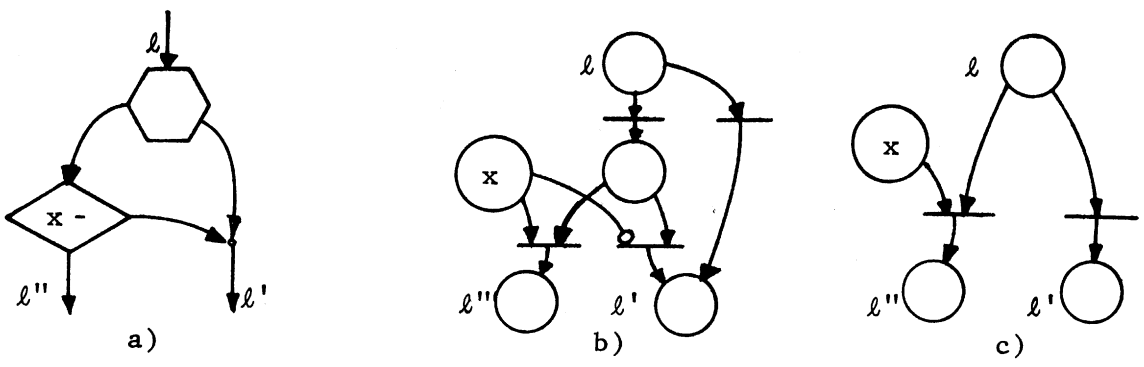


Figure 9.15

This implies that every language generated by a WCA is in $\mathcal{L}_0^\lambda$.

Conversely, the results of chapter 5 imply that every $\mathcal{L}_0^\lambda$-language can be generated by a Finite-State machine whose transitions are restricted by a number of token buffers. One method for doing this is the following (we may wish to first reduce the net to an RPN, but this is not necessary): We replace the closed subnet formed by all bounded places by a state machine (sections 4.1 and 5.6), some of whose transitions may add or remove tokens from collections of buffers. Then we decompose such multiple-buffer operations. It is not known whether this can always be done both in a promptness-preserving and in a hang-up-free manner, but we can do it one way or the other. Figure 9.16 shows a portion of the Finite State control acting on a few buffers. Figure 9.17 shows a (locally) hang-up-free, non-prompt WCA realization, and figure 9.18 shows a prompt WCA realization which may however halt prematurely. This does not of course affect the language of the WCA, as defined in definition 9.6, because not all counters are at zero.

This permits us to conclude:

Theorem 9.11: The languages generated by Weak Counter Automata are exactly the Petri net languages of family $\mathcal{L}_0^\lambda$.

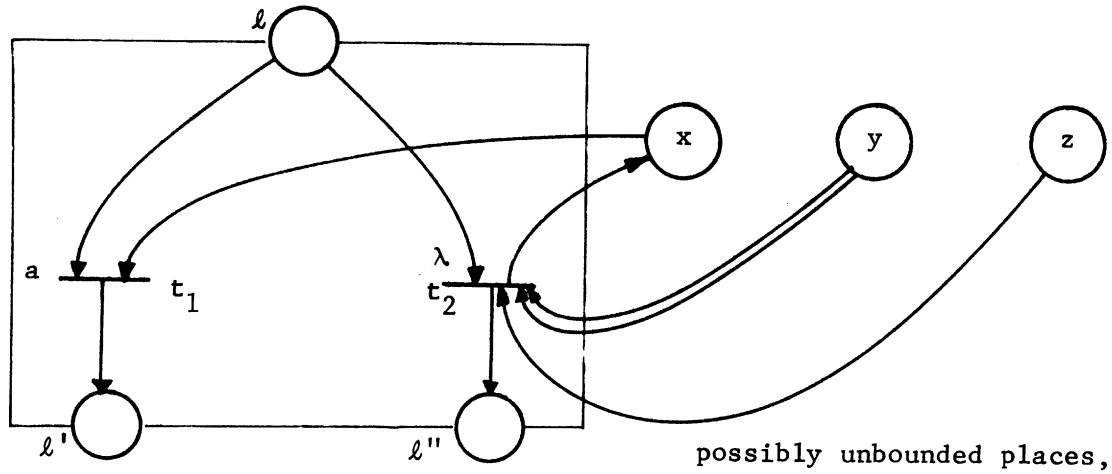
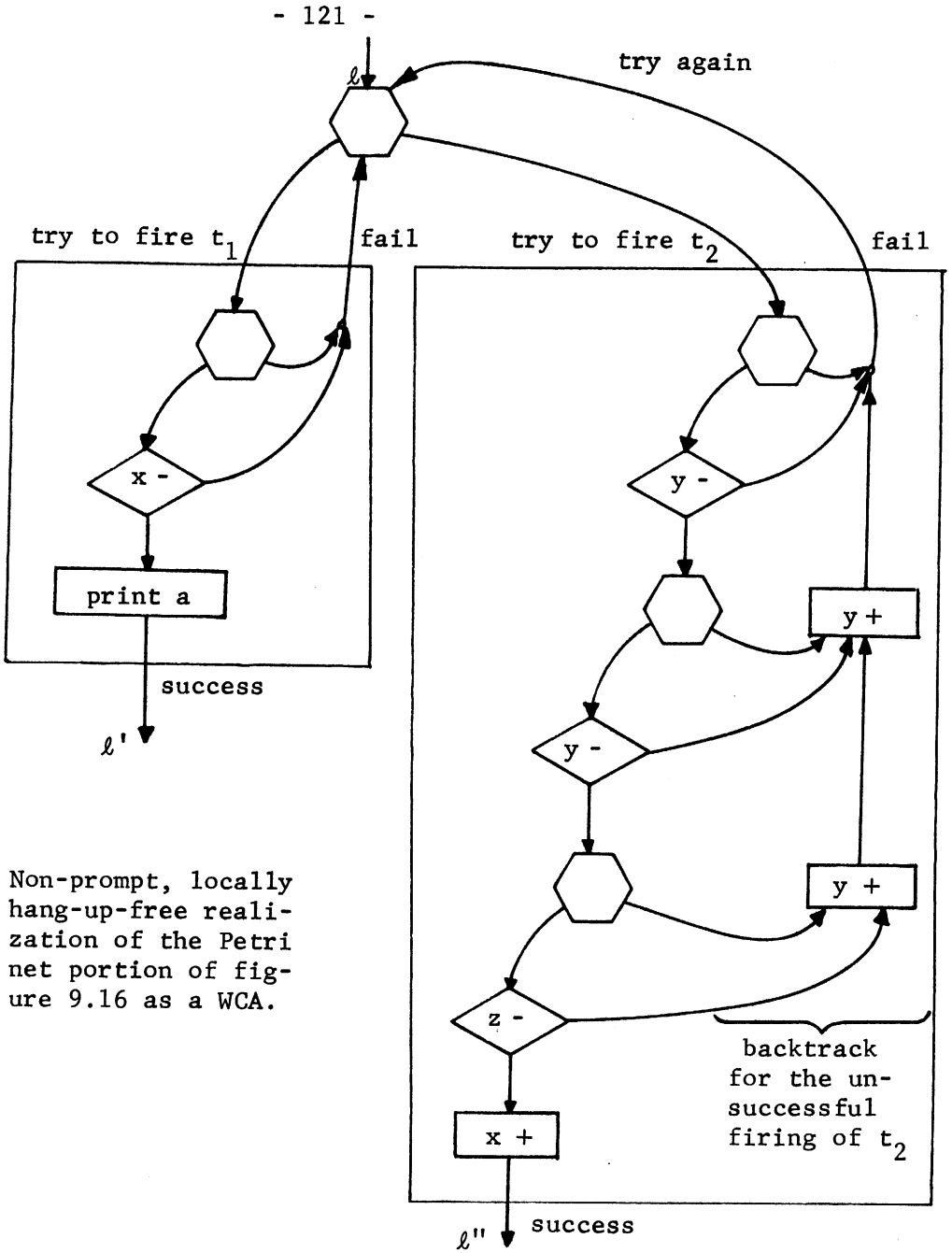


Figure 9.16

We may also define a restriction on WCA's which parallels the restriction of λ-free Petri nets: We want WCA's where print statements are always separated by an a priori bounded number of other statements. This is easily seen to be precisely the case if every loop contains a print statement:



Non-prompt, locally hang-up-free realization of the Petri net portion of figure 9.16 as a WCA.

Figure 9.17

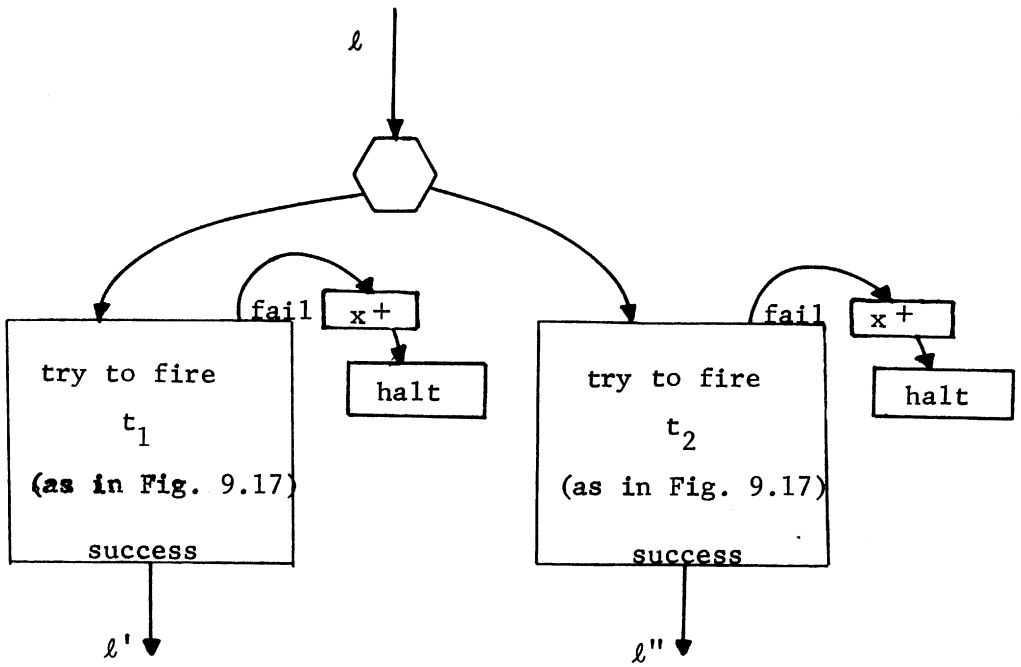


Figure 9.18

Definition 9.7: A Prompt WCA is a WCA where each directed simple circuit in the program contains at least one print statement.

A consequence of Theorem 4.12 (closure under k-limited erasing of $\mathcal{L}_0$) is that the language of a prompt WCA is in $\mathcal{L}_0$. Conversely, we may use the methods of chapter 5 and the construction illustrated in figure 9.18 to transform a λ-free labelled Petri net into a prompt WCA generating the same language. Hence:

Theorem 9.12: The languages generated by prompt WCA's are the family $\mathcal{L}_0$ completed by λ; in fact, the family CSS.

Note 3: In case the reader is wondering what languages would be generated by WCA's if all halting computations were accepted (as opposed to Definition 9.6), let us point out that these languages form the closure of $\mathcal{L}^{\lambda}$ (or $\mathcal{L}$, for prompt WCA's) under intersection with all Regular languages and unrestricted (or k-restricted) renaming.

Note 4: As an application of WCA's, let us construct a WCA that generates a language which encodes the graph of a very fast growing function, as a matter of fact, an arbitrarily fast growing primitive recursive function.

We first define a series of sub-programs A_i, recursively. The sub-program A_i has one entry and one exit, and it operates on variables $x_1 \cdots x_i$. It has the property that (a) no matter what the initial values of the variables are, the final values are bounded, and (b) there exists at least one computation such that $y \geq \alpha_i(x)$ where x and y are the value of x_i before and after the computation. The function α_i is defined by the double recursion:

$$\alpha_i(x) = \alpha_{i-1}(1 + \alpha_i(x-1))$$
$$\alpha_i(0) = 0$$
$$\alpha_1(x) = x + 1$$

The recursive construction of the sub-programs is illustrated in figure 9.19.

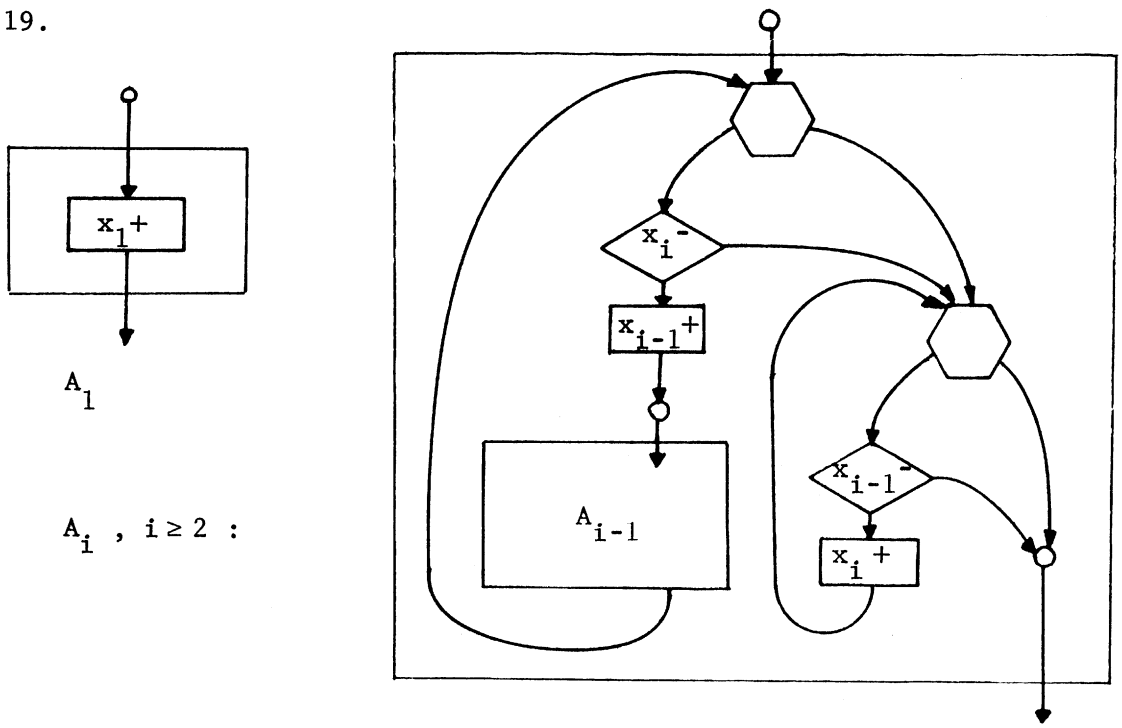


Figure 9.19

The boundedness property (a) can be established by observing that every loop decrements at least one variable that is not incremented in that loop. (This also establishes primitive-recursiveness). The achievability of large numbers (b) is due to the fact that if "scratch" variables $x_1 \cdots x_{i-2}$ are zero at the beginning of each iteration, then the next iteration may achieve

$x_{i-1} := \alpha_{i-1}(x_{i-1})$; if the "scratch" variables are initially positive, we can execute exactly the same statement sequence as before (this is a distinctive property of Weak Counter Automata, and corresponds to the covering property of markings in a Petri net; it has also been called the 'containment' property); but we might be able to do better.

Now, for any n, we may construct the WCA described in figure 9.20, which has n variables and 6n + 3 statements:

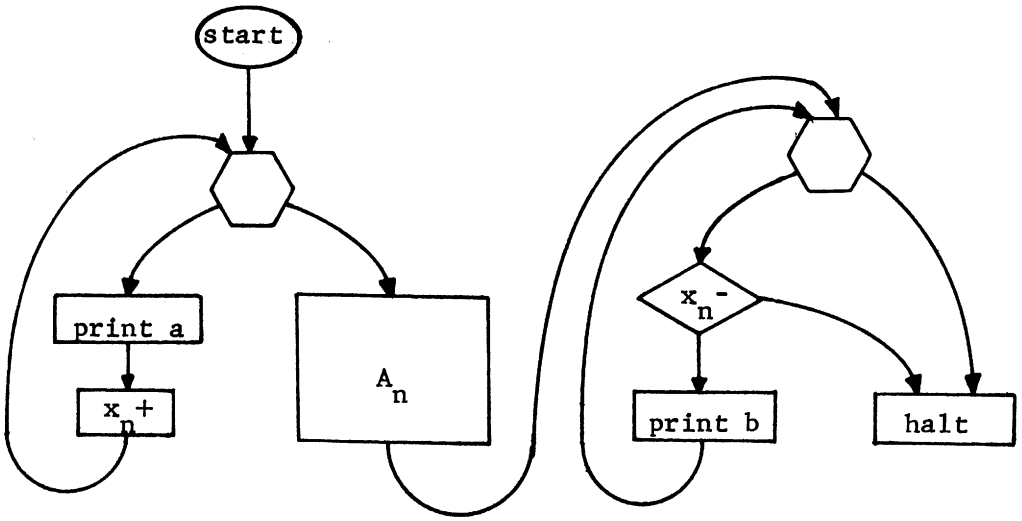


Figure 9.20

The language generated is $\{a^x b^y \mid y \leq \beta(x)\}$, where $\beta(x) \geq \alpha_n(x)$. By choosing n, β can be made to grow faster than any given primitive recursive function. It can be seen that:

$$\alpha_2(x) = 2x \qquad\qquad \alpha_4(x) \geq 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \text{ (x times)}$$

$$\alpha_3(x) = 2\cdot(2^x - 1)$$

For example, $\alpha_5(3)$ is a number so large it cannot really be comprehended, and yet it can be generated by a small bounded WCA, namely A_5 with an initial configuration of $\langle 0,0,0,0,3\rangle$. There corresponds to it a bounded Petri Net with less than a dozen places and transitions whose largest reachable marking is that big.

The main application of WCA's is, in fact, as a tool to understand the computational power of Petri nets. It appears that arbitrarily fast growing primitive recursive functions (though not all primitive recursive functions)

can be weakly computed by WCA's and by Petri nets, in the sense defined in Hack [7,9]. Some recent, unpublished results by Da-Lun Wang, of MIT [3], in combinatorial geometry can be used to show that only primitive recursive functions can be weakly computed by Petri nets, but it is not clear yet what impact this has on Petri net languages and their complexity.

Another consequence, in view of the remark about small Petri nets with very large bounded markings, is that the transformation of a bounded Petri net into a State Machine, as suggested in section 5.6, has at least the complexity of Ackermann's function A(n), which is similar to $\alpha_n(n)$.

- 126 -

can be weakly computed by UCA's and by Petri nets, in the sense defined in Hack [7,9]. Some recent, unpublished results by Da-Lun Wang, of MIT [3], in combinatorial geometry can be used to show that only primitive recursive functions can be weakly computed by Petri nets, but it is not clear yet what impact this has on Petri net languages and their complexity.

Another consequence, in view of the remark about small Petri nets with very large bounded markings, is that the transformation of a bounded Petri net into a State Machine, as suggested in section 5.6, has at least the complexity of Ackermann's function $A(n)$, which is similar to $\sigma_n(n)$.

References.

1. Agerwala, T. A complete model for representing the coordination of asynchronous processes. Hopkins Computer Research Report 32, Johns Hopkins University, Baltimore, Maryland, July 1974

2. Baker, H. G. Petri nets and languages. Computation Structures Group Memo 68, Project MAC, M.I.T., Cambridge, Massachusetts, May 1972

3. Da-Lun Wang, personal communication, M.I.T., March 1975

4. Flynn, M. and Agerwala, T. "Comments on capabilities, limitations and 'correctness' of Petri nets". Computer Architecture News, Vol. 2 No. 4, December 1973

5. Hack, M. Analysis of Production Schemata by Petri nets. MAC-TR 94, Project MAC, M.I.T., Cambridge, Mass., February 1972
   Corrections to 'Analysis of Production Schemata by Petri nets'. Computation Structures Note 17, Project MAC, M.I.T., June 1974

6. Hack, M. Extended State-Machine Allocatable Nets (ESMA), an extension of Free Choice Petri net results. Computation Structures Group Memo 78, Project MAC, M.I.T., May 1973

7. Hack, M. Decision problems for Petri nets and Vector Addition Systems. MAC-TM 59, Project MAC, M.I.T., March 1975. Previously published as Computation Structures Group Memo 95, Project MAC, March 1974

8. Hack, M. The recursive equivalence of the liveness problem and the reachability problem for Petri nets and Vector Addition Systems. Computation Structures Group Memo 107, Project MAC, M.I.T., August 1974. Also in Proceedings of the 15th Annual Symposium on Switching and Automata Theory, New Orleans, La., October 1974

9. Hack, M. The equality problem for Vector Addition Systems is undecidable. Computation Structures Memo 121, Project MAC, M.I.T., April 1975. Also to be published in the journal of Theoretical Computer Science.

10. Holt, A.W. and Commoner, F.C. "Events and Conditions". Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. ACM, New York, 1970, pp.3-52

11. Karp, R. and Miller, R.E. "Parallel Program Schemata: a mathematical model for parallel computation." Proceedings of the 8th Annual Symposium on Switching and Automata Theory. IEEE, October 1967, pp. 55-61

12. Keller, R.M. Vector Replacement Systems: A formalism for modelling asynchronous systems. TR117, Computer Science Laboratory, Princeton University, Princeton, N.J., December 1972.

13. Kosaraju, S.R. Limitations of Dijkstra's Semaphore primitives and Petri nets. Hopkins Computer Research Report 25, Johns Hopkins University, Baltimore, Md., May 1973

14. Miller, R.E. Some relationships between various models of parallelism and synchronization. IBM Research Report RC5074, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., October 1974

15a. Minsky, M. Computation: Finite and infinite machines. Prentice Hall, Englewoods Cliffs, N.J., 1967.

15b. An earlier account of the relevant theorems can be found in:"Recursive Unsolvability of Post's Problem." Annals of Mathematics, 74, pp. 437-454 (1961)

16. Patil, S. Proposed Research on Concurrent Systems. Computation Structures Group Memo 98, Project MAC, MIT, February 1974

17a. Peterson, J.L. Modelling of parallel Systems. Ph. D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, Calif., December 1973

17b. A condensed version of 17a, "Computation Sequence Sets", is to be published in the Journal of Computer and Systems Sciences.

18. Salomaa, A. Formal Languages. Academic Press, New York, 1973

19. Schroeppel, R. A two-counter machine cannot calculate 2^n. Artificial Intelligence Meme 257, A.I. Laboratory, M.I.T., May 1973 †

20. Shepherdson, J.C. and Sturgis, H.E. "Computability of recursive functions". JACM, Vol. 10 No. 2, pp. 217-255, April 1963

Hack (1976): Hack, M. Decidability Questions for Petri Nets. Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., Dec. 1975. To be published as an LCS (formerly Project MAC) Technical Report.

† Essentially the same results were obtained earlier by Ya. Bardzin: "On a class of Turing machines (Minsky machines)" Algebra i Logika, Tom I, vypusk 6, 1963. (in Russian)

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-159

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 136(141-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): 11,22,34,52,76,84,98,126

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP :	(1-136)
UNNUMBERED TITLE & SUPPORT PAGES	i-vi, 1-128
SCANCONTROL, PRINTER'S NOTES, TARGETS	(137-141)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 1/17/96 Date Returned: 1/18/96

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/17/1996

M.I.T. Libraries
Document Services