OBJECT-ORIENTED IMPLEMENTATION OF THE MINIMALLY RESTRICTIVE LIVENESS ENFORCING SUPERVISORY POLICY IN A CLASS OF PETRI NETS

BY

SANGEETHA CHANDRASEKARAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Industrial Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Associate Professor Ramavarapu S. Sreenivas

Abstract

Livelock avoidance is an essential requirement in *Discrete-Event/Discrete-State* (DEDS) systems. Every event of a *live* DEDS system can be executed at some instant in the future, irrespective of its past activities. When a DEDS system is in a livelock-state, some events will enter into a state of suspended animation for perpetuity, while others proceed with no impediment. This report is about the automatic synthesis of *Liveness Enforcing Supervisory Policies* (LESPs) for *Petri net* models of DEDS systems.

Past research has shown that the existence of an LESP in DEDS systems modeled by a class of general *Free-Choice Petri Nets* (FCPNs) is decidable, and the *minimally restrictive* LESP is directly related to the presence of a *right-closed set* of states that are *control invariant* with respect to the system. A minimally restrictive LESP prevents the occurrence of events in a DEDS system only when it is absolutely necessary. This study describes an object-oriented implementation of the minimally restrictive supervisory policy for a class of Petri nets for which this policy is decidable.

To my husband, for his patience

To my mom and dad, for believing in me

Acknowledgements

I am extremely grateful to my adviser Professor R.S. Sreenivas without whose guidance and constant support, this work would not have been possible. I sincerely thank him for giving me the opportunity to work on this project. I am also thankful to the Illinois State Water Survey and my supervisor Dr.Elias G. Bekele for providing me with funding in the form of a Research Assistantship.

I would like to specially thank Nisha Somnath, a friend and colleague for her support when most needed. Without the continual encouragement, strength and selfless sacrifices of my parents, I could not have accomplished the little I did and I cannot thank them enough. Lastly but most dearly, I owe a great deal of gratitude to my husband for patiently tolerating me. His love and support have been my pillars of determination.

Table of Contents

List of Figures . . . . . **vi**

Chapter 1 Introduction **1**
1.1 Liveness in Discrete-Event/Discrete-State Systems 1
1.2 Notations, Definitions and Other Preliminaries 3

Chapter 2 Liveness Enforcing Supervisory Policy **6**
2.1 Supervisory Control of Petri Nets 6
2.2 Existence of an LESP 9
2.3 LESP Synthesis Algorithm 9

Chapter 3 Object-Oriented Implementation **12**
3.1 Class Diagram 12
3.2 Class *MarkingVector* 13
3.3 Class *NodeTable* 14
3.4 Class *MinimalElementsManager* 19
3.5 Class *PetriNet* 23

Chapter 4 Examples **29**
4.1 Input File Format 29
4.2 Illustrations 29

Chapter 5 Future Work **62**

References **64**

List of Figures

1.1 (a) An all-too-familiar instance of applications in a livelocked-state (Mozilla Firefox, in this case) in the Windows® operating system. (b) The reporting of the extant-state back to the code-developers for subsequent patches/updates through *LiveUpdate*. 3

2.1 The procedure for the construction of the *Karp and Miller tree* (KM-tree), $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$, for a partially controlled PN $N = (\Pi, T, \Phi)$ with an initial marking of $\mathbf{m}^0$ under the supervision of a monotone supervisory policy $\mathcal{P} : \mathcal{N}^n \times T \to \{0, 1\}$. 8
2.2 The coverability graph $G(N(\mathbf{m}^0), \mathcal{P})$ of a partially controlled FCPN $N(\mathbf{m}^0)$ under the supervision of a monotone policy $\mathcal{P}$ that makes sure the total number of tokens in $\{p_1, p_2, p_3, p_4\}$ is never zero. This policy can be shown to enforce liveness for this FCPN. 8

3.1 Class Diagram 13
3.2 Class structure of *MarkingVector* 14
3.3 Class structure of *NodeTable* 16
3.4 Hierarchical flowchart of *NodeTable* methods 16
3.5 Class structure of *MinimalElementsManager* 19
3.6 Class Structure of *PetriNet* 24
3.7 Hierarchical flowchart of *doesFullyControlledNetHaveLESP*( ) method 25
3.8 Hierarchical flowchart of *computeMinimalElementsOfFullyControlledNet*( ) method 26
3.9 Hierarchical flowchart of *computeMinimalElementsOfControlInvariantSet*( ) method 27

4.1 Petri net 1 30
4.2 Petri net 2 32
4.3 Petri net 3 34
4.4 Petri net 4 36
4.5 Petri net 5 38
4.6 Petri net 6 40
4.7 Petri net 7 42
4.8 Petri net 8 44
4.9 Petri net 9 46
4.10 Petri net 10 50
4.11 Petri net 11 52
4.12 Petri net 12 56
4.13 Petri net 13 58
4.14 Petri net 14 60

5.1 Petri net N1 63
5.2 Petri net N1 transformed to a free-choice Petri net N2 63

Chapter 1

Introduction

An incorrectly supervised manufacturing- or service-system can enter into a state where no activity progresses towards completion. An anthropomorphic analogy could be the well-intentioned directive of waiting for the other person to use the doorway when two persons arrive simultaneously at either end. While sensible, this directive creates a deadlock when the individuals on either side apply the same directive and wait indefinitely for the other to use the doorway. Analogous situations can occur with greater severity in service- and manufacturing-systems. A more insidious situation could be a livelock, where some events continue to be executed without impediment, while some others cannot be executed even once (i.e. these events enter into a state of suspended animation for perpetuity).

There are sound design principles that avoid deadlocks, but the same cannot be said for livelock-avoidance. Restarting a livelocked task after forcibly terminating it will not rectify the situation in most instances – as only terminating other extant jobs can reset the contentions that originally caused the livelocks. Identifying these jobs, and developing a fair termination policy is not easy even in the simplest of cases. Additionally, job-termination can have dire implications in service-systems. It is therefore of utmost importance to have supervisory policies that ensure none of the extant jobs ever enter into the state of suspended animation alluded to above. This study provides an implementation of a supervisory policy that is minimally restrictive for a certain class of Petri nets for the avoidance of livelocks.

1.1 Liveness in Discrete-Event/Discrete-State Systems

A *Discrete-Event/Discrete-State* (DEDS) system [1] is said to be *live* [2] if, irrespective of the events that occurred in the past, every event that the model describes, can occur at some point in the future. A live DEDS system does not experience deadlocks or livelocks. As any commuter waiting in gridlocked-traffic, or a Windows®-user who has had to forcibly terminate a livelocked application can testify, there are many instances where livelock-freedom is highly desirable. There are numerous instances of livelock and deadlocks in automated manufacturing systems, operations-management of multi-component organizations with event-

driven dynamics like shipyards, airports, hospitals, etc.

In this study, we concern ourselves with *Petri net* (PN) [3, 4] models of manufacturing- and service-systems, where *transitions* in the PN represent activities. The liveness property we seek guarantees that irrespective of the past transition firings, every transition is potentially fireable, although not necessary immediately, in the future. A concurrent system with this property does not experience livelocks, which is a property we desire.

We consider PN models that do not meet the aforementioned liveness specification, and we explore the existence of supervisory policies that enforce this property. At a given *marking* of the PN, the supervisory policy selectively disables a subset of *controllable transitions* to enforce the desired liveness property. A PN is *partially controlled* if the set of controllable transitions is a strict subset of the set of transitions.

The existence of a *liveness enforcing supervisory policy* (LESP) in an arbitrary, partially controlled PN is undecidable (cf. corollary 5.2 of [5]). Furthermore, neither the existence, nor the non-existence, of an LESP in an arbitrary, partially controllable PN is *semi-decidable* (cf. theorems 3.1 and 3.2 of [6]).

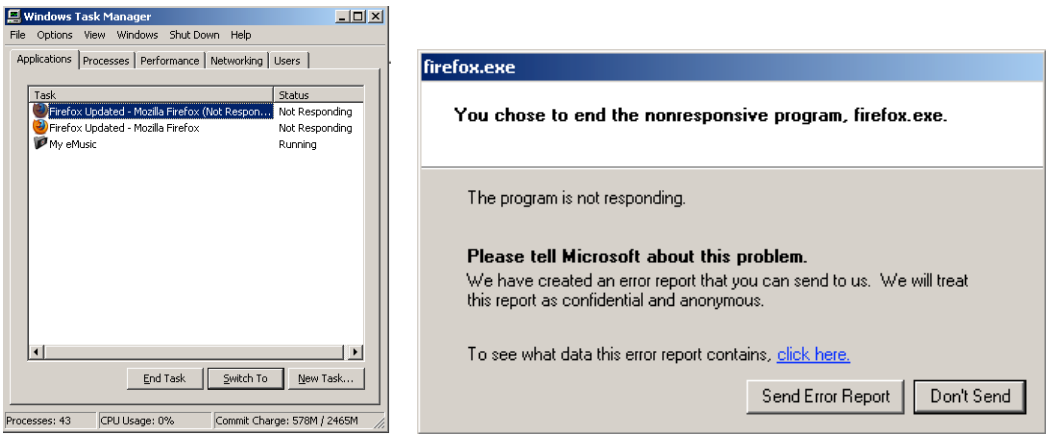
There is no point in attempting to develop algorithms for the synthesis of LESPs for the class of *ordinary* (or, *general*) PNs, as these problems are not even semi-decidable. We must therefore narrow our attention to a smaller class of PNs if we wish to automate the process of LESP-synthesis. Consequently, in this report we consider manufacturing- and service-systems that can be modeled by a class of general *Free-Choice Petri Nets* (FCPNs), $\mathcal{F}$, identified in subsequent text, where the existence of an LESP is decidable [7]. This class strictly contains the class of ordinary FCPNs, another class for which the existence of an LESP is decidable [6].

The class of ordinary FCPNs is capable of modeling the flow of control in manufacturing- and distributed-computation systems [8, 9], and the same is true of the larger class of general FCPNs $\mathcal{F}$. The implementation details of an algorithm that generates the minimally restrictive LESP for PN models that belong to this class of general PNs forms the primary focus of this report.

Every frequent user of the Windows® operating system is familiar with the screen-shot shown in figure 1.1(a), where some unresponsive program (in this case, it is Mozilla Firefox) remains in a state of suspended animation for perpetuity. The only recourse is to terminate the unresponsive program(s) by appropriate action using the *Windows Task Manager*. A user that has met with this fate should be familiar with the next-step, which involves the generation of a "report" that is sent back to the developers (cf. figure 1.1(b)), which is ostensibly used to generate the next-generation of patches/fixes through a service like *Windows Update*. Clearly, there are lacunae in our understanding of livelock-avoidance, otherwise instances like those shown in figure 1.1 would be a historic artifact by now.

Furthermore, forced-shutdown of tasks is not an option for safety critical software applications in the area of healthcare or avionics. A livelocked-application in these arenas would have disastrous consequences, and does not need elaboration. These systems are currently over-designed and scheduled inefficiently (to avoid any possibility of livelocks) – which leads to higher-costs and poor economies of scale.

Even if the paradigm permits the occasional livelock instance, which followed by a reset/reboot event, there is no clear understanding of the process of learning/improvement based on evidentiary data in the form of reports that are generated after each instance of livelock (cf. figure 1.1(b) for an illustration). If it were otherwise, the frequency of updates[1] should drop with time. Our experience does not bear this out. In fact, the (seemingly unending) update process is so commonplace that it has become standard IT-practice in organizations to insist that visitors have the latest patches installed before they are allowed to connect to their network.



(a) Livelocked *Mozilla Firefox* (b) Reporting the termination of Firefox

Figure 1.1: (a) An all-too-familiar instance of applications in a livelocked-state (Mozilla Firefox, in this case) in the Windows® operating system. (b) The reporting of the extant-state back to the code-developers for subsequent patches/updates through *LiveUpdate*.

1.2 Notations, Definitions and Other Preliminaries

An ordinary *Petri net structure* (PN structure) $N = (\Pi, T, \Phi)$ is an ordered 3-tuple, where $\Pi = \{p_1, \ldots, p_n\}$ is a set of n *places*, $T = \{t_1, \ldots, t_m\}$ is a collection of m *transitions*, and $\Phi \subseteq (\Pi \times T) \cup (T \times \Pi)$ is a set of *arcs*. The *initial marking function* (or the *initial marking*) of a PN structure N is a function $\mathbf{m}^0 : \Pi \to \mathcal{N}$, where $\mathcal{N}$ is the set of non-negative integers. We will use the term *Petri net* (PN) to denote a PN structure

[1] Assuming the updates/patches are generated in response to the reports of livelocks, which in turn result from a faulty policy regarding the scheduling of extant tasks.

along with its initial marking $\mathbf{m}^0$, and is denoted by the symbol $N(\mathbf{m}^0)$. We refer the reader to Peterson's text [3] for additional details.

The *state* of a PN $N(\mathbf{m}^0)$ is given by the *marking* $\mathbf{m}^i : \Pi \rightarrow \mathcal{N}$ which identifies the number of *tokens* in each place. A marking $\mathbf{m} : \Pi \rightarrow \mathcal{N}$ is sometimes represented by an integer-valued vector $\mathbf{m} \in \mathcal{N}^n$, where the i-th component $\mathbf{m}_i$ represents the token load ($\mathbf{m}(p_i)$) of the i-th place. Extending this notation to integer-valued vectors in general, the i-th component of any integer valued vector $\mathbf{x}$ is denoted by $\mathbf{x}_i$. The function- and vector-interpretation of the marking is used interchangeably in this report. The context should indicate the appropriate interpretation.

The unit vector whose i-th value is unity is represented as $\mathbf{1}_i$. The vector of all ones (zeros) is denoted as $\mathbf{1}$ ($\mathbf{0}$). Given two integer-valued vectors $\mathbf{x}, \mathbf{y} \in \mathcal{N}^k$, we use the notation $\mathbf{x} \geq \mathbf{y}$ if $\mathbf{x}_i \geq \mathbf{y}_i, \forall i \in \{1, 2, \ldots, k\}$. We use the term $\max\{\mathbf{x}, \mathbf{y}\}$ to denote the vector whose i-the entry is $\max\{\mathbf{x}_i, \mathbf{y}_i\}$. A set of integer-valued vectors $\Delta \subseteq \mathcal{N}^k$ is said to be *right-closed* if $((\mathbf{x} \in \Delta) \wedge (\mathbf{y} \geq \mathbf{x}) \Rightarrow (\mathbf{y} \in \Delta))$. Every right-closed set of vectors $\Delta \subseteq \mathcal{N}^k$ contains a finite set of minimal-elements $min(\Delta) \subset \Delta$ such that (i) $\forall \mathbf{x} \in \Delta, \exists \mathbf{y} \in min(\Delta)$, such that $\mathbf{x} \geq \mathbf{y}$, and (ii) if $\exists \mathbf{x} \in \Delta, \exists \mathbf{y} \in min(\Delta)$ such that $\mathbf{y} \geq \mathbf{x}$, then $\mathbf{x} = \mathbf{y}$. In general the (finite) set of minimal elements $min(\Delta)$ of a right-closed set Δ might not be effectively computable. Valk and Jantzen [10] present a necessary and sufficient condition that guarantees the effective computability of $min(\Delta)$ for an arbitrary right-closed set $\Delta \subseteq \mathcal{N}^k$. Specifically, $min(\Delta)$ is effectively computable if and only if the non-emptiness of $reg(\mathbf{z}) \cap \Delta$ is decidable for every $\mathbf{z} \in (\mathcal{N} \cup \omega)^k$, where $reg(\mathbf{z}) = \{\mathbf{x} \in \mathcal{N}^k \mid \mathbf{x} \leq \mathbf{z}\}$, and ω is a very large positive integer. A procedure for computing the size of $min(\Delta)$ can be found in reference [11].

For a given marking $\mathbf{m}^i$, a transition $t \in T$ is said to be *enabled* if $\forall p \in {}^\bullet t, \mathbf{m}^i(p) \geq 1$, where ${}^\bullet x := \{y \mid (y, x) \in \Phi\}$. The set of enabled transitions at marking $\mathbf{m}^i$ is denoted by the symbol $T_e(N, \mathbf{m}^i)$. An enabled transition $t \in T_e(N, \mathbf{m}^i)$ can *fire*, which changes the marking $\mathbf{m}^i$ to $\mathbf{m}^{i+1}$ according to the equation $\mathbf{m}^{i+1}(p) = \mathbf{m}^i(p) - card(p^\bullet \cap \{t\}) + card({}^\bullet p \cap \{t\})$, where $x^\bullet := \{y \mid (x, y) \in \Phi\}$ and the symbol $card(\bullet)$ is used to denote the cardinality of the set argument. In this study we do not consider simultaneous firing of multiple transitions.

We use the symbol T^* to denote the set of all possible strings that can be constructed from an alphabet T. A string of transitions $\sigma = t_1 t_2 \cdots t_k \in T^*$, where $t_j \in T (j \in \{1, 2, \ldots, k\})$ is said to be a *valid firing string* starting from the marking $\mathbf{m}^i$, if, (1) the transition $t_1 \in T_e(N, \mathbf{m}^i)$, and (2) for $j \in \{1, 2, \ldots, k-1\}$ the firing of the transition t_j produces a marking $\mathbf{m}^{i+j}$ and $t_{j+1} \in T_e(N, \mathbf{m}^{i+j})$ is enabled. If $\mathbf{m}^{i+k}$ results from the firing of $\sigma \in T^*$ starting from the initial marking $\mathbf{m}^i$, we represent it symbolically as $\mathbf{m}^i \rightarrow \sigma \rightarrow \mathbf{m}^{i+k}$. In those contexts where the marking is interpreted as a nonnegative integer-valued vector, it is useful to

define the *input matrix* **IN** and *output matrix* **OUT** as two $n \times m$ matrices, where

$$\mathbf{IN}_{i,j} = \begin{cases} 1 & \text{if } p_i \in {}^\bullet t_j, \\ 0 & \text{otherwise,} \end{cases} \quad \text{and } \mathbf{OUT}_{i,j} = \begin{cases} 1 & \text{if } p_i \in t_j^\bullet, \\ 0 & \text{otherwise.} \end{cases}$$

The *incidence matrix* $\mathbf{C}$ of the PN N is an $n \times m$ matrix, where $\mathbf{C} = \mathbf{OUT} - \mathbf{IN}$. If $\mathbf{x}(\sigma)$ is an m-dimensional vector whose i-th component corresponds to the number of occurrences of t_i in a valid string $\sigma \in T^*$, and if $\mathbf{m}^i \rightarrow \sigma \rightarrow \mathbf{m}^{i+j}$, then $\mathbf{m}^{i+j} = \mathbf{m}^i + \mathbf{C}\mathbf{x}(\sigma)$. Given an initial marking $\mathbf{m}^0$ the set of *reachable markings* for $\mathbf{m}^0$ denoted by $\Re(N, \mathbf{m}^0)$, is defined as the set of markings generated by all valid firing strings starting with marking $\mathbf{m}^0$ in the PN N. The *reachability problem* involves deciding if $\mathbf{m}^i \in \Re(N, \mathbf{m}^0)$, for an arbitrary $\mathbf{m}^i \in \mathcal{N}^n$. This problem is decidable [12, 13]. A PN $N(\mathbf{m}^0)$ is said to be *live* if $\forall t \in T, \forall \mathbf{m}^i \in \Re(N, \mathbf{m}^0), \exists \mathbf{m}^j \in \Re(N, \mathbf{m}^i)$ such that $t \in T_e(N, \mathbf{m}^j)$.

A collection of places $P \subseteq \Pi$ is said to be a *siphon* (*trap*) if ${}^\bullet P \subseteq P^\bullet$ ($P^\bullet \subseteq {}^\bullet P$). A trap (siphon) P, is said to be *minimal* if $\nexists \widetilde{P} \subset P$, such that $\widetilde{P}^\bullet \subseteq {}^\bullet\widetilde{P}$ (${}^\bullet\widetilde{P} \subseteq \widetilde{P}^\bullet$). A PN structure $N = (\Pi, T, \Phi)$ is *Free-Choice* (FC) if $\forall p \in \Pi, card(p^\bullet) > 1 \Rightarrow {}^\bullet(p^\bullet) = \{p\}$. In other words, a PN structure is FC if and only if an arc from a place to a transition is either the unique output arc from that place, or, is the unique input arc to the transition. The PN structure shown in figure 2.2 is FC. A PN $N(\mathbf{m}^0)$ where $N = (\Pi, T, \Phi)$ is FC, is said to be a *Free-Choice Petri net* (FCPN). *Commoner's Liveness Theorem* (cf. [8]) states an FCPN $N(\mathbf{m}^0)$ is live if and only if every minimal siphon in N contains a minimal trap that has a non-empty token load at the initial marking $\mathbf{m}^0$. Testing the liveness of an FCPN is NP-hard (cf. Problem MS3, [14]). The next chapter discusses about the supervisory policies that enforce liveness in PNs.

Chapter 2

Liveness Enforcing Supervisory Policy

2.1 Supervisory Control of Petri Nets

We assume a subset of transitions, called *controllable transitions*, $T_c \subseteq T$ can be prevented from firing by an external agent called the supervisor. The set of *uncontrollable transitions*, denoted by $T_u \subseteq T$, is given by $T_u = T - T_c$. If $T_c = T$, then we say we have a *fully-controlled* PN, otherwise we have a partially controlled PN. An FCPN is said to be *choice-controlled* if $\forall t \in T_u, ({}^\bullet t)^\bullet = \{t\}$. In the graphic representation of PNs controllable (uncontrollable) transitions will be represented by filled (unfilled) rectangles (cf. the PN shown in figure 2.2).

A *supervisory policy* $\mathcal{P} : \mathcal{N}^n \times T \rightarrow \{0, 1\}$, is a function that returns a 0 or 1 for each transition and each reachable marking. The supervisory policy $\mathcal{P}$ permits the firing of transition t_j at marking $\mathbf{m}^i$, only if $\mathcal{P}(\mathbf{m}^i, t_j) = 1$. If $t_j \in T_e(N, \mathbf{m}^i)$ for some marking $\mathbf{m}^i$, we say the transition t_j is *state-enabled* at $\mathbf{m}^i$. If $\mathcal{P}(\mathbf{m}^i, t_j) = 1$, we say the transition t_j is *control-enabled* at $\mathbf{m}^i$. A transition has to be state- and control-enabled before it can fire. The fact that uncontrollable transitions cannot be prevented from firing by the supervisory policy is captured by the requirement that $\forall \mathbf{m}^i \in \mathcal{N}^n, \mathcal{P}(\mathbf{m}^i, t_j) = 1$, if $t_j \in T_u$. This is implicitly assumed of any supervisory policy in this study.

A string of transitions $\sigma = t_1 t_2 \cdots t_k$, where $t_j \in T (j \in \{1, 2, \ldots, k\})$ is said to be a *valid firing string* starting from the marking $\mathbf{m}^i$ under the supervision of $\mathcal{P}$, if, (1) $t_1 \in T_e(N, \mathbf{m}^i)$, $\mathcal{P}(\mathbf{m}^i, t_1) = 1$, and (2) for $j \in \{1, 2, \ldots, k-1\}$ the firing of the transition t_j produces a marking $\mathbf{m}^{i+j}$ and $t_{j+1} \in T_e(N, \mathbf{m}^{i+j})$ and $\mathcal{P}(\mathbf{m}^{i+j}, t_{j+1}) = 1$. The set of reachable markings under the supervision of $\mathcal{P}$ in N from the initial marking $\mathbf{m}^0$ is denoted by $\Re(N, \mathbf{m}^0, \mathcal{P})$.

A supervisory policy $\mathcal{P} : \mathcal{N}^n \times T \rightarrow \{0, 1\}$ is said to be *marking monotone*, if $\forall t \in T, \forall \{\mathbf{m}^j, \mathbf{m}^i\} \subseteq \mathcal{N}^n, (\mathbf{m}^j \geq \mathbf{m}^i) \Rightarrow (\mathcal{P}(\mathbf{m}^j, t) \geq \mathcal{P}(\mathbf{m}^i, t))$. That is, if a transition is control-enabled at some marking by a marking monotone policy, it remains control-enabled for all larger markings. The *Karp and Miller tree* (KM-tree) of a PN $N(\mathbf{m}^0)$ under the supervision of a marking monotone policy $\mathcal{P}$ is a directed graph $\widehat{G}(N(\mathbf{m}^0), \mathcal{P}) = (\widehat{V}, \widehat{A}, \widehat{\Psi})$, where $\widehat{V}$ is the set of *vertices*, $\widehat{A}$ is the set of *directed edges*, and $\widehat{\Psi} : \widehat{A} \rightarrow \widehat{V} \times \widehat{V}$

is the *incidence function*. For each $\widehat{a} \in \widehat{A}$, if $\Psi(\widehat{a}) = (\widehat{v}_i, \widehat{v}_j)$, then the directed edge $\widehat{a}$ is said to *originate* (*terminate*) at $\widehat{v}_i$ ($\widehat{v}_j$). Borrowing from the notation used for PNs, we define ${}^\bullet\widehat{a} = \{\widehat{v}_i\}$ and $\widehat{a}^\bullet = \{\widehat{v}_j\}$. Each vertex $\widehat{v} \in \widehat{V}$ is associated with an *extended marking* $\mu(\widehat{v}) \in (\mathcal{N} \cup \omega)^n$, where ω can be interpreted as a very large positive integer. Each edge $\widehat{a} \in \widehat{A}$ is associated with a transition $\Gamma(\widehat{a}) \in T$. A marking monotone supervisory policy $\mathcal{P} : \mathcal{N}^n \times T \to \{0, 1\}$ can be naturally extended to $\mu(v) \in (\mathcal{N} \cup \omega)^n$ as

$$\mathcal{P}(\mu(v), t) = \begin{cases} 1 & \text{if } \exists \mathbf{m} \leq \mu(v) \text{ such that } \mathcal{P}(\mathbf{m}, t) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

Figure 2.1 contains the procedure for the construction of the KM-tree $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$. We note that if the (marking monotone) supervisory policy $\mathcal{P}$ control-enables all transitions in T for all markings, then this procedure will yield the conventional KM-tree in the literature (cf. section 4.2.1, [3]). Theorem 4.1 of reference [3] states that the KM-tree of an unsupervised PN is finite. The proof of this claim applies equally to $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$, which is finite too.

The *coverability graph* $G(N(\mathbf{m}^0), \mathcal{P}) = (V, A, \Psi)$ is essentially the KM-tree, where the duplicate nodes are merged as one. Figure 2.2 presents a partially controlled PN $N(\mathbf{m}^0)$ and its coverability graph under the supervision of a marking monotone policy $\mathcal{P}$ that disables $t_5 \in T_c$ only at markings in the set $\{(0\ 0\ 0\ 0)^T, (0\ 0\ 0\ 1)^T\}$. Each vertex in the coverability graph has at most one outgoing edge labeled by each transition in T. Therefore, directed paths in the coverability graph can be unambiguously identified by strings in T^*. If there is a path from $v_i \in V$ to $v_j \in V$ with label $\sigma^* \in T^*$ in $G(N(\mathbf{m}^0), \mathcal{P})$, we denote it as $v_i \to \sigma^* \to v_j$.

Theorem 4.2 of reference [15] states that when the KM-tree is constructed in the absence of supervision, $\forall v \in \widehat{V}, \forall k \in \mathcal{N}$, there exists a valid firing string σ starting from $\mathbf{m}^0$ such that $\mathbf{m}^0 \to \sigma \to \mathbf{m}^i$ and

$$\mathbf{m}^i_j = \begin{cases} \mu(v)_j & \text{if } \mu(v)_j \neq \omega, \\ \geq k & \text{otherwise.} \end{cases} \tag{2.1}$$

That is, if $\mu(v)$ for some vertex v in the KM-tree has a collection of ω-symbols, and if we replaced the ω-symbols with any integer k to obtain a marking $\widehat{\mathbf{m}}$, then there is a valid firing string $\sigma \in T^*$ such that $\mathbf{m}^0 \to \sigma \to \mathbf{m}^i$ such that $\mathbf{m}^i \geq \widehat{\mathbf{m}}$. This property is also true of $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$ and $G(N(\mathbf{m}^0), \mathcal{P})$ for a marking monotone $\mathcal{P}$.

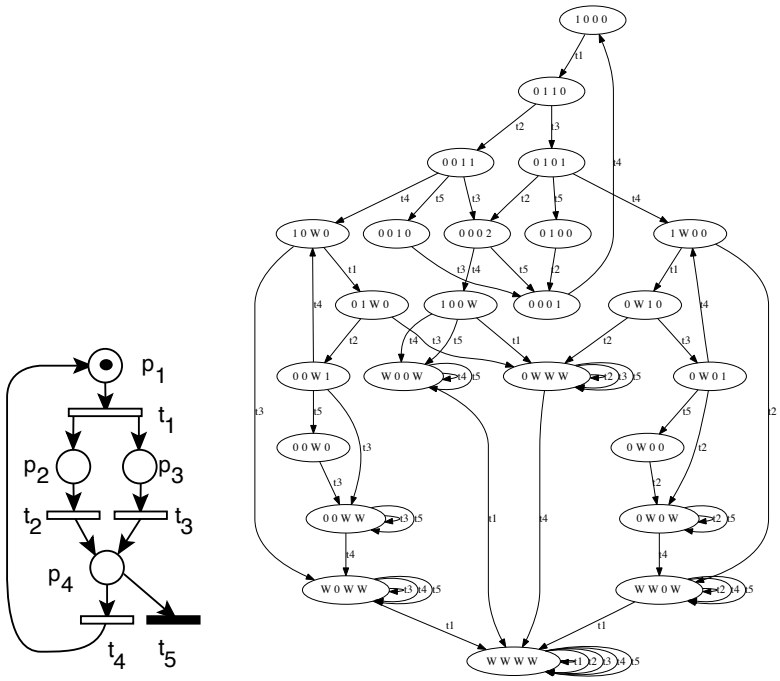
A transition t_k in a PN $N(\mathbf{m}^0)$ is *live* under the supervision of $\mathcal{P}$ if

$$\forall \mathbf{m}^i \in \Re(N, \mathbf{m}^0, \mathcal{P}), \exists \mathbf{m}^j \in \Re(N, \mathbf{m}^i, \mathcal{P}) \text{ such that } t_k \in T_e(N, \mathbf{m}^j) \text{ and } \mathcal{P}(\mathbf{m}^j, t_k) = 1.$$

KM − tree − with − supervision $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$, where $N = (\Pi, T, \Phi)$ is a partially controlled PN and $\mathcal{P}$ is a monotone policy.

The root vertex of $\widehat{G}(N)$ is v_0. $V \leftarrow \{v_0\}$, and $\mu(v_0) = \mathbf{m}^0$.
for $v_i \in V$ **do**
  if $\mu(v_i)$ is identical to $\mu(v_j)$ for some $v_j \in V$ **then**
    v_i has no children in $\widehat{G}(N)$ and is marked as the *duplicate* of v_j.
  end if
  if $T_e(N, \mu(v_i)) \cap \{t \in T \mid \mathcal{P}(\mu(v_i), t) = 1\} = \emptyset$ **then**
    v_i has no children in $\widehat{G}(N)$ and is marked as a *terminal* vertex.
  end if
  for $t_j \in T_e(N, \mu(v_i)) \cap \{t \in T \mid \mathcal{P}(\mu(v_i), t) = 1\}$ **do**
    Create a new vertex v_k. $V \leftarrow V \cup \{v_k\}$.
    Create a new directed edge $a_l, A \leftarrow A \cup \{a_l\}, {}^\bullet a_l = v_i, a_l^\bullet = v_k$ and $\Gamma(a_l) = t_j$.
    if $\mu(v_i)_p = \omega$ for some $p \in \{1, 2, \ldots, n\}$ **then**
      $\mu(v_k)_p = \omega$.
    end if
    if ($\exists v_q \in V$ on the directed path from v_0 to v_k in $\widehat{G}(N)$ such that $\mu(v_q) \leq \mu(v_i) + \mathbf{C}\mathbf{1}_j$, where $\mathbf{1}_j$ is the unit-vector that has a 1 at the j-th location) **and** ($\exists r \in \{1, 2, \ldots, n\}, \mu(v_q)_r < (\mu(v_i) + \mathbf{C}\mathbf{1}_j)_r$) **then**
      $\mu(v_k)_r = \omega$.
    else
      $\mu(v_q)_r = (\mu(v_i) + \mathbf{C}\mathbf{1}_j)_r, r \in \{1, 2, \ldots, n\}$.
    end if
  end for
end for=0

Figure 2.1: The procedure for the construction of the *Karp and Miller tree* (KM-tree), $\widehat{G}(N(\mathbf{m}^0), \mathcal{P})$, for a partially controlled PN $N = (\Pi, T, \Phi)$ with an initial marking of $\mathbf{m}^0$ under the supervision of a monotone supervisory policy $\mathcal{P} : \mathcal{N}^n \times T \rightarrow \{0, 1\}$.



(a) $N(\mathbf{m}^0)$ (b) $G(N(\mathbf{m}^0), \mathcal{P})$

Figure 2.2: The coverability graph $G(N(\mathbf{m}^0), \mathcal{P})$ of a partially controlled FCPN $N(\mathbf{m}^0)$ under the supervision of a monotone policy $\mathcal{P}$ that makes sure the total number of tokens in $\{p_1, p_2, p_3, p_4\}$ is never zero. This policy can be shown to enforce liveness for this FCPN.

A supervisory policy $\mathcal{P}$ enforces liveness if all transitions in N are live under $\mathcal{P}$. The policy $\mathcal{P}$ is said to be *minimally restrictive* if for every supervisory policy $\widehat{\mathcal{P}} : \mathcal{N}^n \times T \rightarrow \{0,1\}$ that enforces liveness in N, the following condition holds $\forall \mathbf{m}^i \in \mathcal{N}^n, \forall t \in T_e(N, \mathbf{m}^i), \mathcal{P}(\mathbf{m}^i, t) \geq \widehat{\mathcal{P}}(\mathbf{m}^i, t)$. Alternately, if a minimally restrictive supervisory policy $\mathcal{P}$ that enforces liveness in N prevents the occurrence of transition $t \in T_e(N, \mathbf{m}^i)$ at some marking $\mathbf{m}^i \in \mathcal{N}^n$, then every policy that enforces liveness in N should prevent the occurrence of $t \in T$ for the marking $\mathbf{m}^i$. There is a unique minimally restrictive policy that enforces liveness in every PN N that has some policy that enforces liveness (cf. theorem 6.1, [5]).

2.2 Existence of an LESP

The existence of a supervisory policy that enforces liveness in an arbitrary, partially controlled PN is undecidable (cf. theorem 5.3 and corollary 5.2, [5]). The existence of a supervisory policy that enforces liveness is decidable when all transitions in a PN are controllable (i.e. $T = T_c$), or if the PN is bounded or if the PN belongs to the family of general FCPNs $\mathcal{F}^2$, which strictly contains the class of ordinary FCPNs (cf. references [5, 6, 7]). The procedure of deciding the existence of an LESP in any of these decidable classes is NP-hard. The results in references [6, 7] forms the basis of the algorithm of LESP-synthesis and a survey of the results are given below.

If there is an LESP for some $N(\mathbf{m^0})$, then there is an unique minimally restrictive LESP for PN $N(\mathbf{m^0})$ (cf. theorem 6.1, [5]). The minimally restrictive LESP for a general FCPN that belongs to the class $\mathcal{F}$ is *marking monotone*. That is, if a controllable transition is control-enabled at a marking by the minimally restrictive LESP, then it is control-enabled at any larger marking. A transition $t \in T$ in an FCPN is said to be a *non-choice* transition, if $({}^\bullet t)^\bullet = t$. The minimally restrictive LESP for an ordinary FCPN does not control-disable any of the non-choice transitions [16].

2.3 LESP Synthesis Algorithm

The set of initial markings for which there is an LESP for a PN structure n , is defined as follows

$$\Delta(N) = \{\mathbf{m}^0 \mid \exists \text{ an LESP for } N(\mathrm{m}^0)\}$$

The set $\Delta(N)$ is em control invariant [17] with respect to the PN structure N. That is, if $\mathbf{m^1} \in \Delta(N)$, $t_u \in T_e(N, \mathbf{m^1}) \cap T_u$ and $\mathbf{m^1} \xrightarrow{t_u} \mathbf{m^2}$ in N, then $\mathbf{m^2} \in \Delta(N)$. In words, if an uncontrollable transition t_u can fire at a marking $\mathbf{m^1}$ in $\Delta(N)$, resulting in a marking $\mathbf{m^2}$, then $\mathbf{m^2}$ is also in $\Delta(N)$. Alternately, only the

firing of a controllable transition at an marking in $\Delta(N)$ can result in a new marking that is not in $\Delta(N)$. This observation follows directly from the definition of $\Delta(N)$.

Suppose $\mathbf{m}^0 \in \Delta(N)$, then the supervisory policy that control-disables any (controllable) transition at a marking in $\Delta(N)$ if its firing would result in a new marking that is not in $\Delta(N)$, is a minimally restrictive LESP for $N(\mathbf{m^0})$[6]. If the PN structure N is fully-controlled (i.e. $T_u = \emptyset$), or if N belongs to the class (F), then the set $\Delta(N)$ is right-closed, and is characterized by its minimal elements $\min(\Delta(N))$.

There is a procedure to test the control-invariance of a right-closed set of markings Ψ of a PN structure N. If Ψ does not pass this test, then it is possible to find the largest subset of Ψ that is control invariant with respect to N (cf. Lemma 5.10, [6]). If 1) Ψ is a right-closed set of markings that is control invariant with respect to N, 2) N is a PN structure where $\Delta(N)$ is known to be right-closed, 3) $\mathcal{P}_\Psi$ is a supervisory policy that control-disables any (controllable) transition at a marking in Ψ if its firing would result in a new marking that is not in Ψ, and 4) $\mathbf{m^0} \in \Psi$, we can construct the coverability graph, $G(N(\mathbf{m}^0), \mathcal{P}_\Psi)$, of $N(\mathbf{m^0})$ under the supervision of $\mathcal{P}_\Psi$, along the same lines as the coverability graph of a PN (cf. section 4.2.1, [3]).

The policy $\mathcal{P}_\Psi$ is an LESP for $N(\mathbf{m^0})$ if and only if

1. $\mathbf{m^0} \in \Psi$, and
2. there is a closed path $\nu \xrightarrow{\sigma} \nu$ in $G(N(\mathbf{m}^i), \mathcal{P}_\Psi)$, for each $\mathbf{m^i} \in \min(\Psi)$ where
   (a) all transitions appear at least once in σ (i.e. $x(\sigma) \geq 1$), and
   (b) the net-change in the token-load in each place after the firing of σ is non-negative (i.e. $Cx(\sigma) \geq 0$)

The LESP synthesis algorithm for a PN structure N that belongs to a class where $\Delta(N)$ is known to be right-closed essentially involves a search for a right-closed set of markings Ψ that is control invariant with respect to N, where each member of $\min(\Psi)$ meets the path-requirement on its coverability graph. This is done in an iterative manner starting with an initial set

$$\Psi_0 = \{\mathbf{m}^0 \mid \exists \text{ an LESP for } N(\mathbf{m}^0) \text{ if all transitions in } N \text{ are controllable } \}$$

which is known to be right-closed (cf. [5, 6]). The LESP synthesis procedure is described below.

- If $\mathbf{m}^0 \notin \Psi_i$, the procedure terminates with the conclusion that there is no LESP for $N(\mathbf{m}^0)$.
- If $\mathbf{m}^0 \in \Psi_i$, and Ψ_i is not control invariant with respect to the PN structure N, it is replaced with its largest control invariant subset, Ψ_{i+1} where $\Psi_{i+1} \subset \Psi_i$. Following this the process is repeated with $\Psi_i \leftarrow \Psi_{i+1}$.

- If $\mathbf{m}^0 \in \Psi_i$, and Ψ_i is control invariant with respect to the PN structure N, each minimal element of the control invariant, right-closed set Ψ_i is tested for the path-requirement on its coverability graph described earlier.

- If all the minimal elements satisfy this requirement, then the members of $\min(\Psi)$ are presented as a description of the LESP for $N(\mathbf{m}^0)$.

- If there are minimal elements that do-not meet the path-requirement, then each minimal element $\mathbf{m}^i$ that fails the requirement is "elevated" by $card(\Pi)$ many unit-vectors as follows

$$\mathbf{m}^i \leftarrow \{\mathbf{m}^i + 1_i \mid i \in 1, 2, \ldots card(\Pi)\}$$

  where 1_i is the i^{th} unit-vector, i.e., the above process replaces the minimal element $\mathbf{m}^i$ with $card(\Pi)$-many minimal elements, which in turn defines a right-closed set $\Psi_{i+1} \subset \Psi_i$. After this, the process is repeated with $\Psi_i \leftarrow \Psi_{i+1}$.

This procedure forms the corpus of the algorithm used to synthesize an LESP for $N(\mathbf{m}^0)$, when it exists, for a structure N for which it is known that $\Delta(N)$ is right-closed, and has been implemented as a command line application on Mac and Windows platforms. The details of this implementation are given in the next chapter.

Chapter 3

Object-Oriented Implementation

This chapter discusses the object-oriented implementation of the algorithms described in the chapters 1 and 2 to obtain the LESP for the class of PNs for which liveness is decidable. The implementation has been done in C++ using the Microsoft Visual C++ compiler as a command-line application. The implementation makes use of the object-oriented concepts like encapsulation and polymorphism. This implementation primarily uses STL Containers *viz. std::vector*, a sequence container for object collections. To enhance the performance and efficiency, the implementation also uses certain key features of *Boost* C++ Libraries (`http://www.boost.org/`). BOOST_FOREACH, one such feature, is a construct for C++ that iterates over sequences thereby freeing us from having to deal directly with iterators or write predicates. It also makes use of the *boost :: shared_ptr* and *boost :: unordered_map*, an associate container of the *Boost* library for object collections.

3.1 Class Diagram

Figure 3.1 shows the Object oriented representation of a minimally restrictive liveness enforcing supervisory policy. The implementation is done within four major classes called *PetriNet*, *NodeTable*, *MinimalElements-Manager* and *MarkingVector*.

PetriNet.h contains the declarations for Class *PetriNet* and its members. Pertrinet.cpp contains implementation for Class *PetriNet* and its members. Class *PetriNet* relates to Class *MinimalElementManager* using a 'Has-a' relationship i.e., PetriNet contains one or more objects of Class *MinimalElementManager*. MinimalElementManager.h contains declarations for Class *MinimalElementManager* and its members, MinimalElementManager.cpp implements all the members. Class *MinimalElementManager* holds a pointer to Class *PetriNet* and hence is marked by a 'Uses-a' relationship. NodeTable.h contains the declarations for the Class *NodeTable* and Class *MarkingVector* together with its members and NodeTable.cpp implements these two classes and its members. Class *NodeTable* and *PetriNet* are tightly coupled with each other and hence are declared as friends of each other, allowing both the classes to access each others private members. *PetriNet* contains objects of *NodeTable* and hence is marked by a 'Has-a' relationship while NoteTable holds

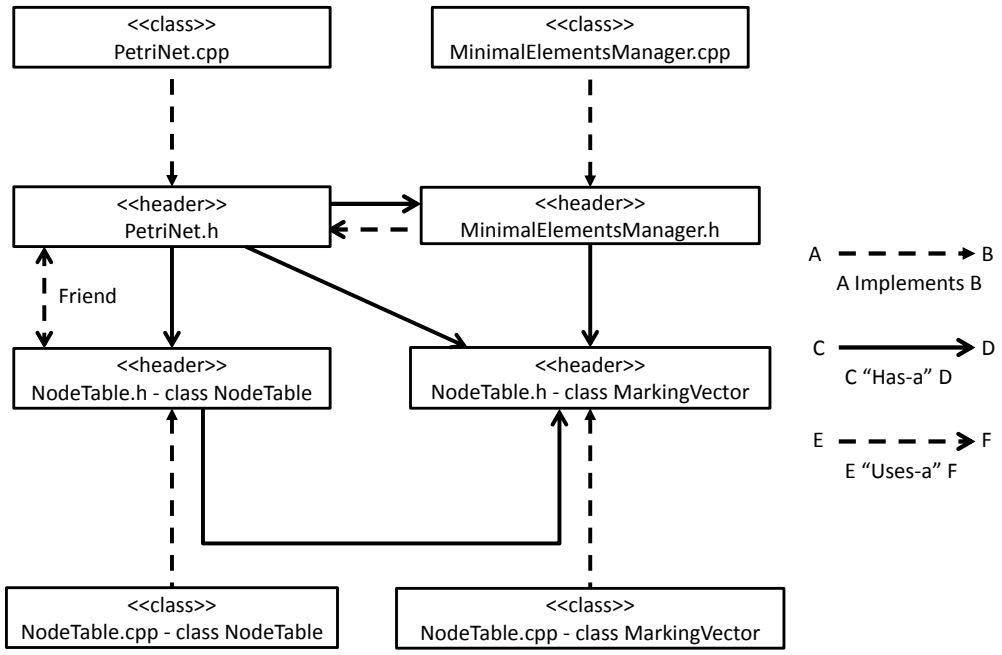


Figure 3.1: Class Diagram

a pointer to Class *PetriNet*. The three classes *PetriNet*, *NodeTable*, *MinimalElementManager* are marked by a 'Has-a' relationship with *MarkingVector* and hence contain one or more objects of Marking vector. The sections below give a detailed description and functionality of each of these classes.

3.2 Class *MarkingVector*

The marking vector $\mathbf{m}^i$ corresponds to a set containing the *number of tokens* in each *place* at any given state of the Petri net. The *MarkingVector* class is used to represent this set which forms the basic building block of the algorithms used to obtain the LESP for the net. The *public* members of this class are *place*, a vector (*STL container*) of integers that stores the token count. The class exposes overloaded methods for some basic arithmetic operations of addition (+), subtraction (-) and multiplication ($\times$) by a constant ω and set comparison operations such as $==$, $\prec$, $\preceq$ and $\succeq$. Every other class contains members which are objects of the *MarkingVector* class. The method *initialize*( ) is used to assign unit vector markings $\mathbf{1}_j$ corresponding to each place in the net. The figure 3.2 shows the structure of this class.

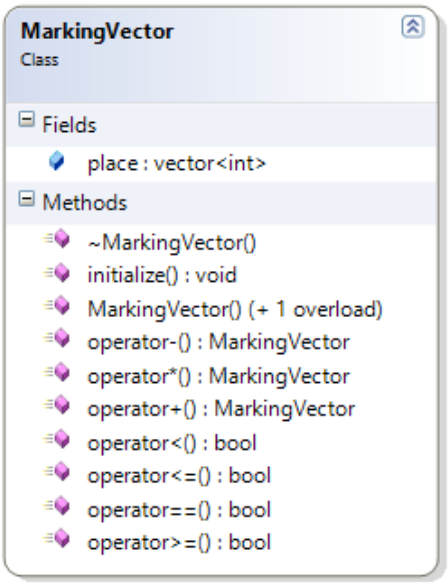


Figure 3.2: Class structure of *MarkingVector*

3.3 Class *NodeTable*

The *NodeTable* class implements the algorithm in figure 2.1 to obtain the coverability graph of the given PN structure with an initial marking, $\mathbf{m}^0$.

processNode( ), a recursive method is the primary method of this class which in turn invokes the other member functions to compute the vertex and edge parameters of the reachability graph. This method is initialized with the initial marking $\mathbf{m}^0$. Each vertex together with all its connecting edges forms a node in the *NodeTable* and is characterized by the members *fromNode*, *marking*, *byTransition*, *enabledTransitions*, *nodeType*, *concurrent*, *conflicting*, *duplicateNode* and *index*.

The function of each of the member variables in the figure 3.3 is given below

- *marking* is an object of the *MarkingVector* class and stores the marking corresponding to the node.
- The nodes are indexed by an integer value which is stored in *index*.
- Each vertex except for the root vertex has a corresponding parent vertex, the index of which is stored in the integer member variable, *fromNode*.
- All new vertices in the coverability graph are obtained by traversing an edge that corresponds to the firing of a transition. This edge corresponding to each node is represented in the node table by the member *byTransition*.

- Nodes are classified into four categories as root, internal, duplicate and terminal using integer constants. To obtain the coverability graph, it is important to identify the type of each node and *nodeType*, an integer, stores this information. The constant integral values associated with the node types are tabulated in Table 3.1.

- If a node is identified as a duplicate of another node, the index of the original node is stored in an integer member, *duplicateNode*.

- The list of the transitions enabled at any node are stored as a vector of integers in *enabledTransitions*. This depends on the state of the the net at that node which is defined by the *marking*, $\mathbf{m}^i$.

- Nodes can have concurrent/conflicting transitions which are identified by the integer flag *concurrent* and *conflicting*.

Node type	**Constant**
ROOT	0
INTERNAL	1
DUPLICATE	2
TERMINAL	3

Table 3.1: Node Classification

The nodes are stored as an *unordered map* of *shared pointers* to objects of class *NodeTable*. These are declared in class *PetriNet* as *nodeTableElements*. A pointer to the object of *PetriNet* class which defines the given net is used to access this map to insert an element as and when a new node is created. An *unordered map* of *shared pointers* with integer *keys* is *type-defined* as *NodeMap* to provide flexibility in modifying the data structure for further code development .

The first node in the coverability graph with the initial marking, $\mathbf{m}^0$ is initialized using the method *initialize*( ). Since this is the root node, its *fromNode* (parent node) and *byTransition* (traversed edge) are nil and hence assigned to integer constants NIL_NODE and NIL_TRANSITION with a value of -1.

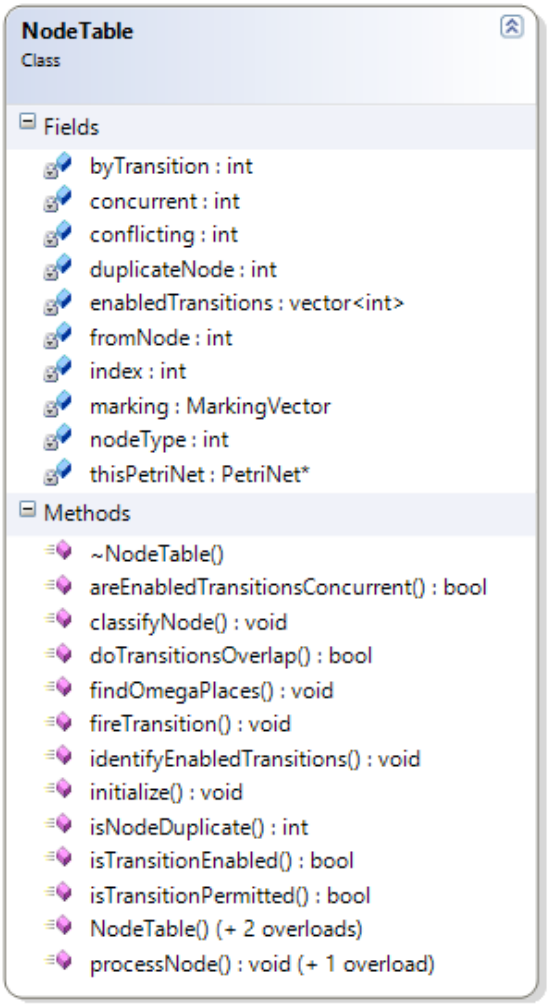


Figure 3.3: Class structure of *NodeTable*

The hierarchical flowchart of the member functions of *NodeTable* class is given in figure 3.4 and the Table 3.2 gives a short description, input parameters and the return type of all its methods.

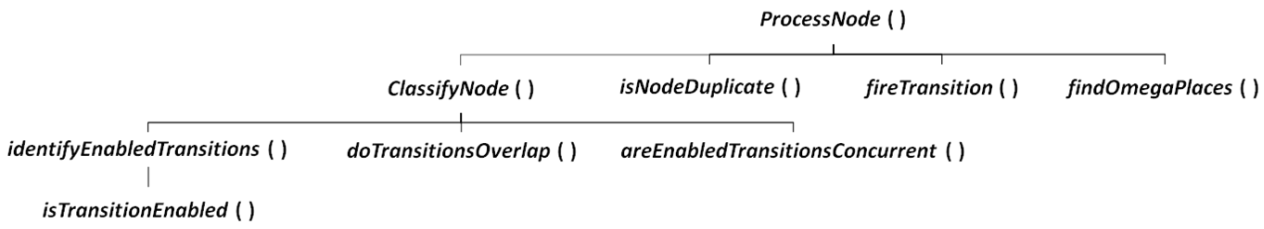


Figure 3.4: Hierarchical flowchart of *NodeTable* methods

The method *processNode*( ) is then called with this initial node as its parameter. *classifyNode*( ) is

the first step within the *processNode*( ) and it invokes the member functions *identifyEnabledTransitions*( ) to compute the list of enabled transitions. Any transition is considered *enabled* if the number of tokens corresponding to a place on the net is at least equal to or greater than the input arc weight from the place to that transition. The enabled transitions are then classified to check for overlap for concurrency using the functions *doTransitionsOverlap*( ) and *areEnabledTransitionsConcurrent*( ). On classification, the *nodeTableElements* is then pushed back with this current node. The node is then checked for duplicacy by comparison with all the other nodes on the that have been traversed thus far in the coverability graph within *isNodeDuplicate*( ). If the node is not duplicate, the recursive iteration of *processNode*( ) begins for each enabled transition for the node by firing the transition in *fireTransition*( ) and obtaining a new marking which corresponds to the new node. An object of *NodeTable* for this newly generated node is created and the recursive *processNode*( ) is called for this node. In this process, the generated marking could belong to the right-closed set of the parent node and in such case the place is marked as ω, a large integer constant value. This is done within *findOmegaPlaces*( ). Since the number of tokens in each place is limited by ω, any right-closed new marking that is generated from a parent node with ω tokens, with the increased token number in the same place is marked as a duplicate node. At some node, when there are no further transitions that can be fired, it is marked as a terminal node. When this process is recursively iterated over each node, the entire coverability graph is generated with its parameters stored in *nodeTableElements* of the *PetriNet*'s object. *isTransitionPermitted*( ) is invoked when the coverability graph is required to be drawn for a net which is partially controlled to obtain the minimal markings that form an LESP. This method is called within the *isTransitionEnabled*( ) method. For each transition, this method invokes the *fireTransition*( ) to obtain the marking that would result as a consequence of having traversed this edge. If this marking is within the right-closed set of the minimal markings of the fully controlled Petri net, it is *permitted* to fire, else the transition is not *enabled*.

Method Name	Description	Parameters	Return Type
initialize( )	initialization for ROOT node (first node)	*int* : parent Node *int* : edge *MarkingVector* : initial marking	*void*
processNode( )	obtain the elements of the coverability graph	*NodeTable* : current node	*void*
classifyNode( )	get the transitions that are enabled at any node, classify if they are concurrent and/or conflicting	*NodeTable* : current node	*void*
identifyenabledTransitions( )	identify the transitions enabled at a given node	*NodeTable* : current node	*void*
isTransitionEnabled( )	Check if the given transition is enabled	*int* : current transition *MarkingVector* : current marking	*bool*
isTransitionPermitted( )	In partially controlled Petri net, checks if a transition fire results in a right-closed marking of one of the minimal markings of a fully controlled Petri net	*int* : enabled transition *transition* : current marking	*bool*
doTransitionsOverlap( )	Check if the enabled transitions overlap	*vector<int>* : enabled transitions	*bool*
areEnabledTransitionsConcurrent( )	Check if enabled transitions can fire concurrently	*vector<int>* : enabled transitions	*bool*
isNodeDuplicate( )	Check if current node is a duplicate of any other node in the coverability graph	*NodeTable* : current node	*int*
fireTransition( )	Traverse through the give edge (transition) that is enabled to obtain the next vertex (node)	*int* : enabled transition *MarkingVector* : current marking *MarkingVector* : new Marking	*void*
findOmegaPlaces( )	Check if a new marking is in the right-closed set of any other node in the graph	*MarkingVector* : new marking *int* : current parent node	*void*

Table 3.2: Method definitions of Class *NodeTable*

3.4 Class *MinimalElementsManager*

The *MinimalElementsManager* is the class that implements the methods in the procedure explained in the chapter 2 that details the LESP synthesis to obtain the minimal markings of a Petri net for a fully or partially controlled Petri net. The class structure of *MinimalElementsManager* is given by figure 3.5 and Table 3.3 gives a list of all the methods, a description of its functionality, the input parameters and return type.

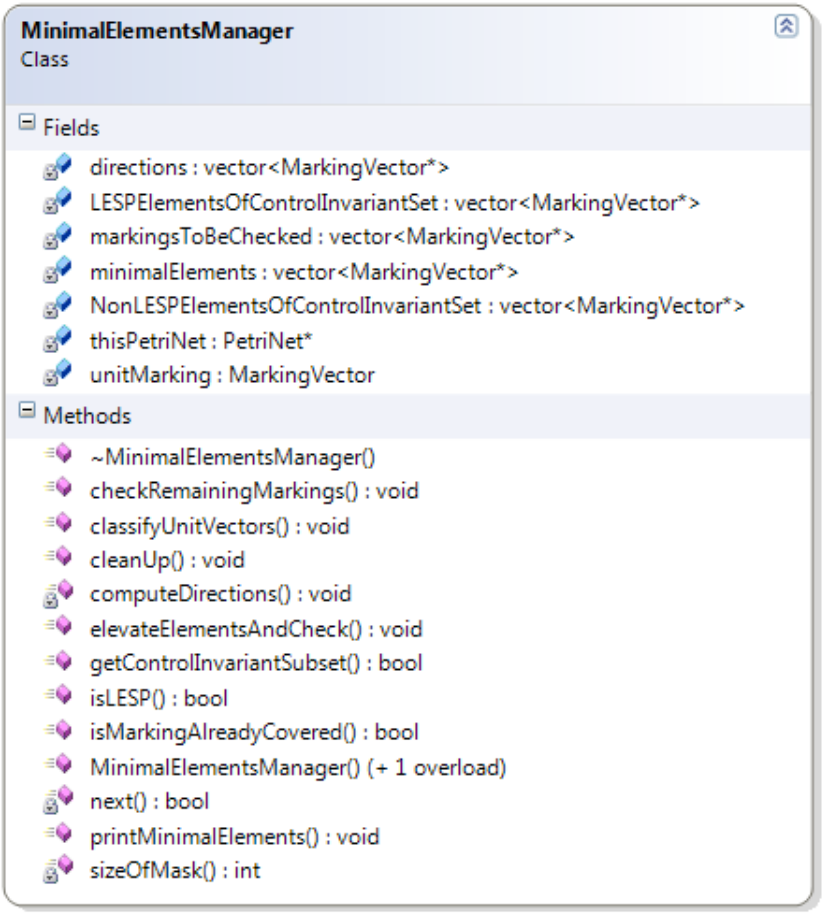


Figure 3.5: Class structure of *MinimalElementsManager*

A vector of pointer to objects of type *MarkingVector* called *minimalElements* is used to store the computed minimal markings for the net that characterize a LESP. The preliminary step in finding the minimal markings involves the test for all the unit markings, $\mathbf{1}_j$. For a given net that is fully controlled, the method *classifyUnitVectors*( ) is used to ascertain if the unit markings $\mathbf{1}_j$ are an LESP. The member of type*MarkingVector*, *unitMarking* is initialized inside a *for loop* construct for each *place* in the net. A object to *NodeTable* is created with *unitMarking* as the marking of the initial node and the *processNode*( ) method

is invoked to obtain the coverability graph of this net. The *doTheLoopTest*( ) method in *PetriNet* class is then called to test the existence of an LESP for each of unit marking. When this test satisfies, a pointer to a copy of *unitMarking* is pushed back into *minimalElements*. The copy of the unit markings for which the LESP test fails are pushed back in the member *markingsToBeChecked* as a vector of pointer to objects *MarkingVector* for further analysis.

The *checkRemainingMarkings*( ) method is used to identify the favourable directions along which there exists a possible minimal marking and compute the same. The number of directions that need to be checked is a power set of the total number of places in the net. The problem of obtaining the minimal markings is nothing but the problem of identifying the set containing number of tokens in each place for which the underlying net is live. Since any marking can be represented by a linear combination of the unit markings, the member *markingsToBeChecked* forms the basis for the power set of markings to be analyzed. This is done within the method called *computeDirections*( ). The power set is generated using the methods *next*( ) and *sizeOfmask*( ) called within *computeDirections*( ). For each marking within *markingsToBeChecked*, the favourable directions are identified by adding it with marking *mask* which corresponds to each element in the power set that is multiplied with a large constant ω (representing dumping infinite tokens) along the directions of evaluation and checking the condition of LESP feasibility by computing the coverability graph with the current marking as the initial node. The original basis directions corresponding to a positive test are then pushed back in the member *directions*. The marking in *markingsToBeChecked* is then elevated along each direction in *directions* and then checked for LESP condition again. If the test is satisfied then this elevated marking is a minimal marking and is pushed back into *minimalElements*, else the marking is added to the existing set of *markingToBeChecked* and this process is repeated until all the remaining markings have been checked for. *isMarkingAlreadyCovered*( ) is a method used to identify markings which are in the right-closet set of basis markings. This check is done every time before the new marking identified as *minimal* is pushed back and also after the method *checkRemainingMarkings*( ) completes. This completes the methods used in *MinimalElementsManager* to obtain the minimal elements for a fully controlled Petri net.

The *getControlInvariantSubset*( ) method is used to obtain the the largest control invariant subset of the set of markings that have been marking as minimal for the fully controlled net. The *minimalElements* member is updated with this set when this method is executed i.e., any marking which is not control invariant is deleted from the original set. The method *isLESP*( ) is then used to test for the LESP condition

for each marking in the control invariant set. The markings for which the test satisfies are stored in the member *LESPElementsofControlInvariantSet* and the ones that fail are stored in *NonLESPElementsOfControlInvariantSet*. Both these members are defined by a vector of pointers to *MarkingVector*. Each element in *NonLESPElementsOfControlInvariantSet* is then elevated by unit vectors along the favourable direction and the process is repeated until an LESP exists.

Method Name	Description	Parameters	Return Type
classifyUnitVectors( )	Check if unit markings are minimal markings for a fully controlled net	*PetriNet* * : current Petri net	*void*
checkRemainingMarkings( )	Check for the minimal markings for a fully controlled net in directions where the unit markings are not minimal	*void*	*void*
computeDirections( )	Compute feasible directions to obtain a minimal marking	*MarkingVector* : marking	*void*
sizeOfMask( )	Calculate the number of unit tokens in each computed direction	*MarkingVector* : direction	*int*
next( )	Successive marking denoting the direction to be checked next	*MarkingVector* : direction	*bool*
cleanUp( )	Obtain the smallest subset of the set of computed minimal elements (basis elements)	*void*	*void*
getControlInvariantSubset( )	Obtain the set of minimal markings that are control invariant	*void*	*bool*
isLESP( )	Check if LESP test is satisfied for the control invariant marking of the partially controlled net	*void*	*bool*
isMarkingAlreadyCovered( )	Check if a marking is covered by some minimal element already computed	*MarkingVector* : marking	*bool*
elevateElementsAndCheck( )	To elevate the number of tokens along a feasible direction, If LESP test fails in that direction	*void*	*void*
printMinimalElements( )	Write the set of minimal markings that characterize an LESP to the console	*void*	*void*

Table 3.3: Method definitions of Class *MinimalElementsManager*

3.5 Class *PetriNet*

The PetriNet class (figure 3.6) is the nerve centre of the entire code which encompasses objects of the *MarkingVector*, *NodeTable* and *MinimalElementsManager* and exposes methods to the user to solve the goal problem. The interface is written such that this forms the first layer of the code.

Inputs, Initialization & Validation

This section describes the member functions in this class that are used for obtaining the user inputs, initializations and input validation.

To solve for the LESP for any FCPN, the inputs to the code are n, m, *input matrix* (**IN**) and *output matrix* (**OUT**) and $\mathbf{m}^0$ are required to be provided by the user. These are stored in the members *noOfPlaces*, *noOfTransitions*, *inputWeightsToTransition*, *outputWeightsFromTransition* and *initialMarking* respectively. *noOfPlaces* and *noOfTransitions* are integer members and *initialMarking* is of type *MarkingVector*. For a *Controlled* PN, there is an additional input which corresponds to the transitions that are controllable. This input is given as a switch with 1 and 0 denoting controllable transitions and uncontrollable transitions respectively. A *global* function *getIOFile*( ) is used to read the name of the input file from the command line which initializes the *inputFileName*, *noOfPlaces* and *noOfTransitions* for the net with which an object to *PetriNet* is assigned.

The members *inputWeightsToTransition*, *outputWeightsFromTransition* are defined as *std::vector* of pointers to objects of type *MarkingVector*. The system model of any Petri net is characterized by it Incidence Matrix, **C** which is represented by the member *incidenceMatrix*. Also, to analyze a partially controlled Petri net, it is required to identify the net token load in each *place* corresponding to the firing of uncontrollable incidence and this is stored in the member *uncontrollableIncidence*. These two members have been defined as *std::vector* of pointers to objects of type *MarkingVector*.

The *loadInputData*( ) method initializes **IN**, **OUT**, m^0, T_u and computes the *incidence matrix* **C** and the *uncontrollableIncidence* of the net. Since these algorithms that obtain the minimally restrictive LESP markings are applicable only to a specific class of Petri nets that include free-choice nets, a method *isNetFreeChoice*( ) has been included to validate the Petri net input to the code. If the net is not free choice, the algorithm terminates with a message that there is no LESP for the net.

Two print methods *printInputsToConsole*( ) and *printControllableTransitions*( ) have been included with overloads for std:out for printing the inputs to the code.

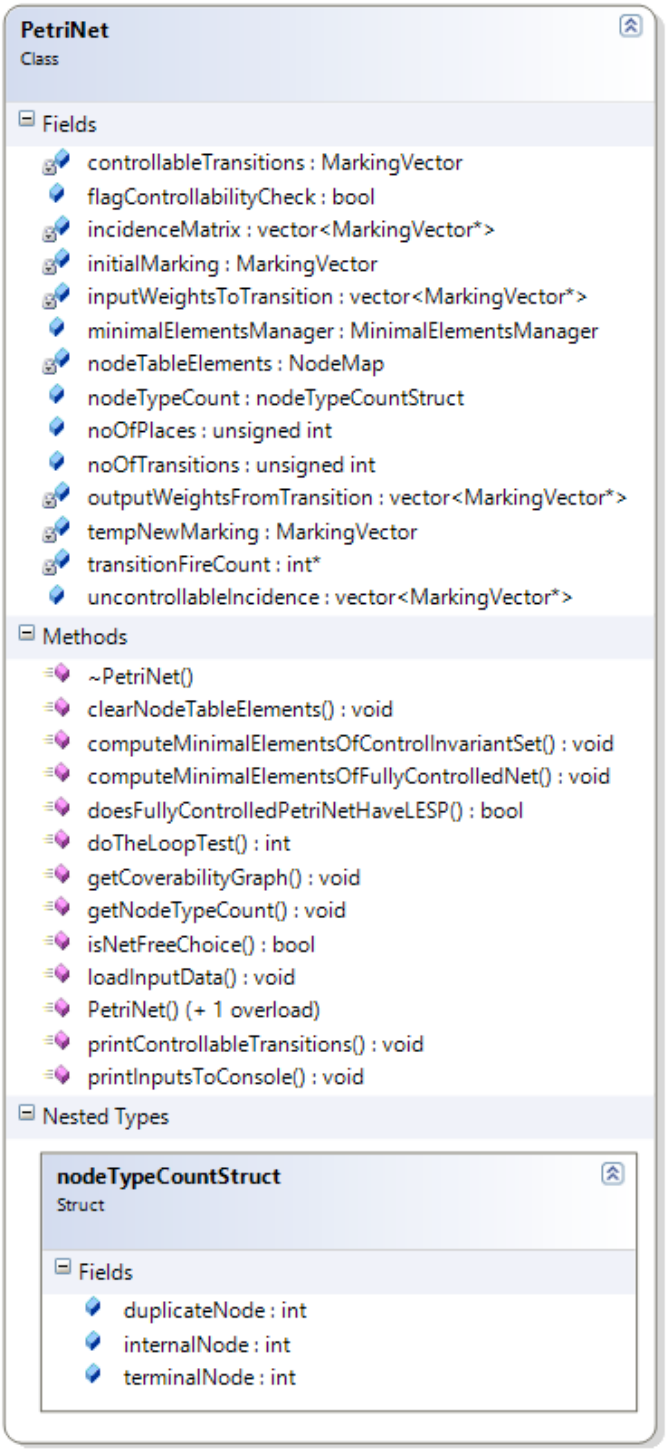


Figure 3.6: Class Structure of *PetriNet*

LESP Implementation

The LESP test for the class of Petri nets where $\Delta(N)$ is known to be right-closed are primarily implemented into three modules

- LESP test for a given initial state (initial marking, m_0) of the net
- Computing the minimal elements that characterize the LESP for a net where all the transitions are assumed controllable
- Computing the minimal elements that characterize the LESP for a net where the controllable transitions are specified

The member functions that are correspond to each of the above procedure are *doesFullyControlledNetHaveLESP*( ), *computeMinimalElementsOfFullyControlledNet*( ) and *computeMinimalElementsOfControlInvariantSet*( ).

The member function *doTheLoopTest* is used to test for the existence of the closed path $\upsilon \xrightarrow{\sigma} \upsilon$ in $G(N(\mathbf{m}^i), \mathcal{P}_\Psi)$ for any marking $\mathbf{m}^i$. This is implemented as a feasibility-test for an appropriately posed *Integer Linear Program*. lp_solve 5.5.2.0, a Mixed Integer Linear Programming Solver [18] based on revised simplex methods and Branch-and - bound methods for integers has been integrated in the algorithm for this purpose. Specifically, the ILP-instance is solved using a solve(lprec *lp) command where lp is a pointer to the ILP-model, within the code. The existence of the path with the desirable property in $G(N(\mathbf{m}^i), \mathcal{P}_\Psi)$ corresponds to solve returning a zero in the implementation.

In *doesFullyControlledNetHaveLESP*( ), the node table corresponding to the net is initialized with $\mathbf{m}^0$ and the *processNode*( ) method is invoked to obtain the coverability graph of the net. All the information corresponding to the coverability graph is stored within this member *nodeTableElements*. With the *nodeTableElements* populated with the entire graph, the feasibility test *doTheLoopTest*( ) is invoked. There is an LESP if the test returns a zero and no LESP if a one is returned. The code flow for this procedure is represented by figure 3.7.

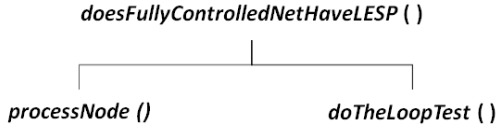


Figure 3.7: Hierarchical flowchart of *doesFullyControlledNetHaveLESP*( ) method

To obtain the minimal markings that characterize an LESP for both a fully Controlled PN and a partially controlled PN, the object *minimalElementsManager* to the class *MinimalElementsManager* declared in this class is used to invoke the appropriate methods. In *computeMinimalElementsOfFullyControlledNet*( ), the LESP test is checked for a sequence of markings evaluated iteratively first by checking for the individual token loads for each place of the net using *clasifyUnitVectors*( ) and then iterating over the remaining elements in *checkRemainingMarkings*( ) for which the first condition was not satisfied. These functions populate the *minimalElements* member of the class *minimalElementsManager*. The *cleanUp*( ) function class of is then invoked for the *minimalElementsManager* object to obtain the min(Ψ).

Figure 3.8 and figure 3.9 show the hierarchical flow chart for the methods *computeMinimalElementsOfFullyControlledNet*( ) and *computeMinimalElementsOfControlInvariantSet*( ) respectively.

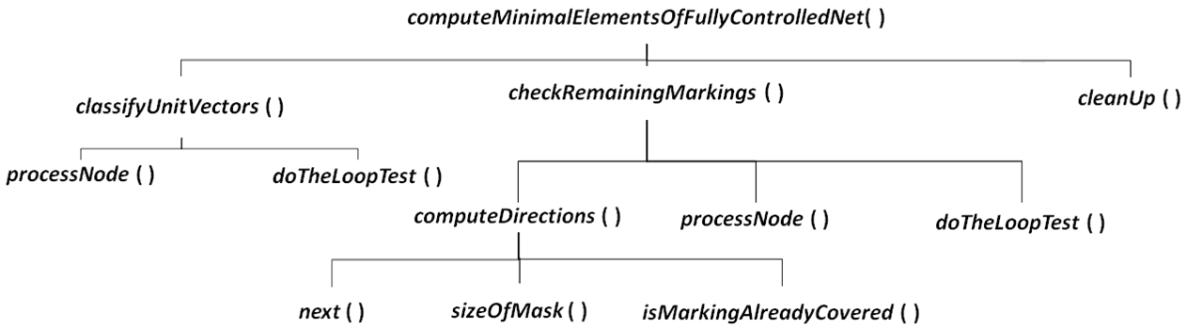


Figure 3.8: Hierarchical flowchart of *computeMinimalElementsOfFullyControlledNet*( ) method

For the partially controlled net, the algorithm requires the set of minimal markings with an LESP for the fully controlled net to have been already computed and given either as input or this method is invoked in sequence with *computeMinimalElementsOfFullyControlledNet*( ). The method *computeMinimalElementsOfControlInvariantSet*( ) invokes the *getControlInvariantSubset*( ) of the *minimalElementsManager* to first obtain the largest control invariant subset of the minimal markings. This has been implemented through a *while* control statement until no minimal marking violates the control invariance property. A constant, MAX_ITERATION limits it from getting into an infinite loop. To test for the LESP condition for each marking $\mathbf{m}^i \in \Psi_i$ is checked using the member function *isLESP* of *minimalElementsManager*. If an LESP exists for each of these markings,the list of markings is output with the message "There is an LESP". If for some marking, the LESP condition fails, that particular marking is elevated by the markings that correspond to the favourable directions and the function is recursed until all the elements have an LESP. If none of the minimal markings of the fully controlled net satisfy the control invariance property, there is no LESP for the net.

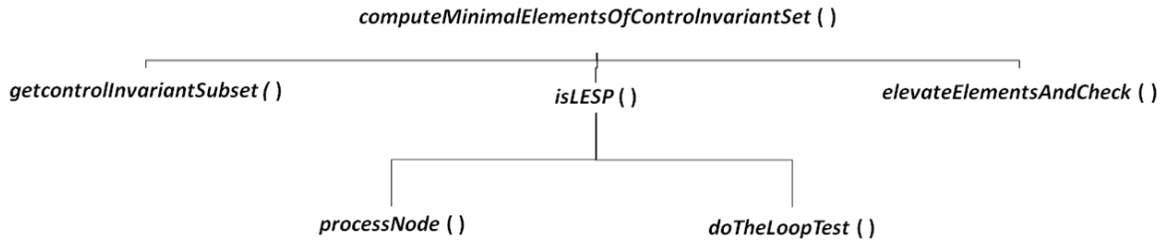


Figure 3.9: Hierarchical flowchart of *computeMinimalElementsOfControlInvariantSet*( ) method

The procedure to draw the coverability graph for a fully controlled net and the partially controlled net varies in this respect that the computation of enabled transitions involves a check to see if the transition if enabled is permitted through *isTransitionPermitted*( ) method in *NodeTable* class that has been described previously, i.e., this condition checks to see that the new marking stays within the right-closed control invariant subset for the partially controlled net and *flagControllabilityCheck* is used to switch between the two cases.

An object of type *MarkingVector*, *tempNewMarking* has been included as a member of this class for reusability purposes. This instance has been used whenever a new temporary object is required to be created. It gives an advantage by reducing the overhead of calling the *MarkingVector* constructor for temporary objects.

Since these algorithms involve obtaining the coverability graph each time the LESP condition is to be checked, the member function *clearNodeTableElements*( ) has been provided so as to be able to reuse the vector of objects *nodeTableElements*. This function calls the clear() method for vectors and deletes the existing graph.

A graph visualization software tool 'Graphviz' [19] is used to obtain the actual graphic of the reachability graph of the net. The *PetriNet* class also has a member functions *getCoveribilityGraph*( ) that writes the coverability graph information into a (*.viz) file in a format that is readable by Graphviz. Along with the *nodeTableElements*, this file also requires the count of the different node classifications and the struct *nodeTableCountStruct* is used to keep track of this count. It holds three integer variables *duplicateNode*, *internalNode* and *terminalNode* that represents the cardinality of each named type respectively. Table 3.4 gives a brief description of the methods discussed above together with their parameters and return type.

Method Name	Description	Parameters	Return Type
loadInputData( )	Assigns the Petri net's inputs to the corresponding members	*char ** : input file name	*void*
printInputsToConsole( )	Print method to write the inputs to the console	*void*	*void*
printControllableTransitions( )	Print method to write the set of controllable transitions to the console	*void*	*void*
isNetFreeChoice( )	Condition to check if the given Petri net is free-choice	*void*	*bool*
doTheLoopTest( )	Feasibility test for existence of LESP through an ILP implementation using *lp_solve*	*void*	*int*
doesFullyControlledNetHaveLESP( )	LESP test for the fully controlled net for a given initial marking	*void*	*bool*
computeMinimalElementsOfFullyControlledNet( )	Get the minimal markings that characterize the LESP for the fully controlled Petri net	*void*	*void*
computeMinimalElementsOfControlInvariantSet( )	Get the minimal marking that characterize the LESP for the partially controlled Petri net	*void*	*void*
clearNodeTableElements( )	Erase the previously computed elements of the coverability graph	*void*	*void*
getNodeTypeCount( )	Count the number nodes of each type namely, internal, duplicate and terminal	*NodeTypeCountStruct* : node type count	*void*
getCoverabilityGraph( )	Write an output file of the node table that is recognized by *Graphviz*	*char ** : output file name	*void*

Table 3.4: Method definitions of Class *PetriNet*

Chapter 4

Examples

This chapter gives some illustrations of the implemented algorithms on a few partially controlled Petri nets to obtain the minimal markings that characterize the liveness-enforcing supervisory policy (LESP).

4.1 Input File Format

The format of the input file to this code is as below

- - - - - - - - - - - - - - - -
INPUT FILE FORMAT
- - - - - - - - - - - - - - - -
n m

IN

OUT

$\mathbf{m}^0$

$\mathcal{C}_i$
- - - - - - - - - - - - - - - -

where ,

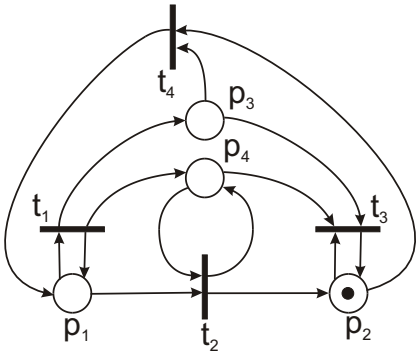
$$\mathcal{C}_i = \begin{cases} 1 & \text{if} \quad t_j \in T_c \\ 0 & \text{if} \quad t_j \in T_u \end{cases}$$

4.2 Illustrations

The figure 4.1a - figure 4.14a are some examples of PN structures both *free-choice* and *non free-choice* that belong to a class of PNs with a decidable LESP. The input files corresponding to these PNs are shown in figure 4.1b - figure 4.14b and the LESP policy determined from executing the algorithm are shown in figure 4.1c - figure 4.14c respectively. In the PN structure, the controllable transitions are shown as a filled rectangle and the uncontrollable transitions are denoted by an unfilled rectangle. All self loops are denoted by *S in the incidence matrix that is output. In these PNs, the basic assumption that non-choice transitions

are uncontrollable holds to obtain the minimally restrictive supervisory policy.

Petri net 9 in figure 4.9a has all the transitions except t_7 controllable. However, the non-choice transitions t_5, t_6, t_8, t_9 and t_{11} are also assumed uncontrollable in the input file in accordance to the minimally restrictive supervisory policy. Petri net 12 in figure 4.12a and Petri net 14 in figure 4.14a show the existence of a supervisory policy even on the account of having no controllable transitions.



(a) PN-1

```
pn1        Wed Dec 05 02:51:18 2012       1
4 4
1 1 0 0
0 0 1 1
0 0 1 1
0 1 1 0
1 0 0 1
0 1 1 0
1 0 0 0
1 1 0 0
0 1 0 0
1 1 1 1
```

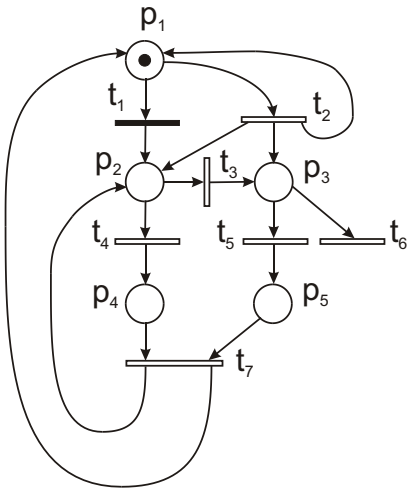
(b) Input file for PN-1

Figure 4.1: Petri net 1

```
pn1.res       Wed Dec 05 19:30:40 2012       1
 Input File = "pn1"

 Incidence Matrix :

    T  1  2  3  4
 P
   1  *S -1  .  1
   2   .  1 *S -1
   3   1  . -1 -1
   4   1 *S -1  .

 Initial Marking : ( 0 1 0 0 )

 There is no LESP for this PN
 ____________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 )
  2: ( 0 1 1 0 )


 List of Controllable Transitions
 --------------------------------
 t1 t2 t3 t4


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 )
  2: ( 0 1 1 0 )


 This is An LESP
```

(c) Output file with LESP for PN-1

Figure 4.1 (cont.): Petri net 1



(a) PN-2

```
pn2        Wed Dec 05 03:38:39 2012       1
5 7
1 1 0 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 1 0 0 0 0 1
1 1 0 0 0 0 1
0 1 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
1 0 0 0 0
1 0 0 0 0 0 0
```

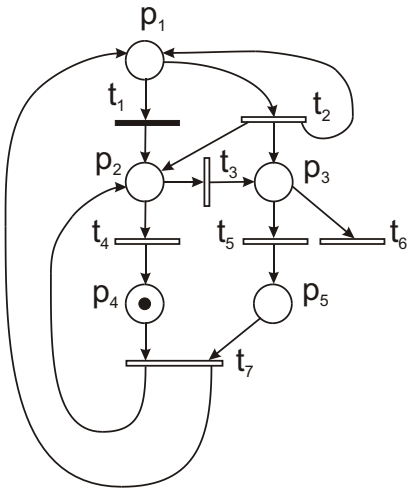
(b) Input file for PN-2

Figure 4.2: Petri net 2

```
pn2.res        Wed Dec 05 19:54:29 2012       1
 Input File = "pn2"

 Incidence Matrix :

    T  1  2  3  4  5  6  7
 P
   1  -1 *S  .  .  .  .  1
   2   1  1 -1 -1  .  .  1
   3   .  1  1  . -1 -1  .
   4   .  .  .  1  .  . -1
   5   .  .  .  .  1  . -1

 Initial Marking : ( 1 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 1 0 0 1 )
  3: ( 0 0 0 1 1 )
  4: ( 0 1 0 1 0 )
  5: ( 0 0 1 1 0 )
  6: ( 0 1 1 0 0 )
  7: ( 0 2 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 0 0 1 1 )


 This is An LESP
```

(c) Output file with LESP for PN-2

Figure 4.2 (cont.): Petri net 2



(a) PN-3

```
pn3        Thu Dec 06 21:44:48 2012       1
5 7
1 1 0 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 1 0 0 0 0 1
1 1 0 0 0 0 1
0 1 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 0
1 0 0 0 0 0 0
```

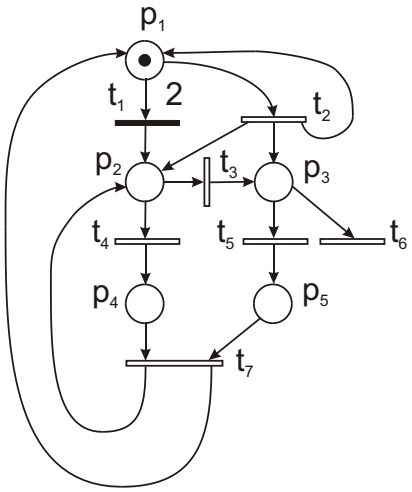
(b) Input file for PN-3

Figure 4.3: Petri net 3

```
pn3.res       Wed Dec 05 19:59:01 2012       1
 Input File = "pn3"

 Incidence Matrix :

    T  1  2  3  4  5  6  7
 P
   1  -1 *S  .  .  .  .  1
   2   1  1 -1 -1  .  .  1
   3   .  1  1  . -1 -1  .
   4   .  .  .  1  .  . -1
   5   .  .  .  .  1  . -1

 Initial Marking : ( 0 0 0 1 0 )

 There is no LESP for this PN
 ____________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 1 0 0 1 )
  3: ( 0 0 0 1 1 )
  4: ( 0 1 0 1 0 )
  5: ( 0 0 1 1 0 )
  6: ( 0 1 1 0 0 )
  7: ( 0 2 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 0 0 1 1 )


 This is An LESP
```

(c) Output file with LESP for PN-3

Figure 4.3 (cont.): Petri net 3



(a) PN-4

```
pn4        Wed Dec 05 20:05:50 2012       1
5 7
2 1 0 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 1 0 0 0 0 1
1 1 0 0 0 0 1
0 1 1 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 1 0 0
1 0 0 0 0
1 0 0 0 0 0 0
```

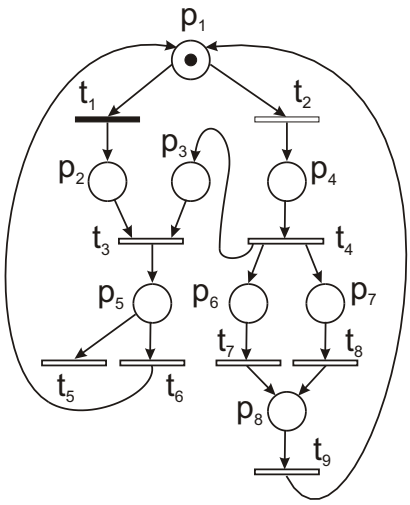
(b) Input file for PN-4

Figure 4.4: Petri net 4

```
pn4.res        Wed Dec 05 20:06:06 2012       1
 Input File = "pn4"

 Incidence Matrix :

    T  1  2  3  4  5  6  7
 P
   1  -2 *S  .  .  .  .  1
   2   1  1 -1 -1  .  .  1
   3   .  1  1  . -1 -1  .
   4   .  .  .  1  .  . -1
   5   .  .  .  .  1  . -1

 Initial Marking : ( 1 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 1 0 0 1 )
  3: ( 0 0 0 1 1 )
  4: ( 0 1 0 1 0 )
  5: ( 0 0 1 1 0 )
  6: ( 0 1 1 0 0 )
  7: ( 0 2 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 )
  2: ( 0 0 0 1 1 )


 This is An LESP
```

(c) Output file with LESP for PN-4

Figure 4.4 (cont.): Petri net 4



(a) PN-5

```
pn5        Sun Dec 02 14:28:52 2012       1
8 9
1 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1
1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0
1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

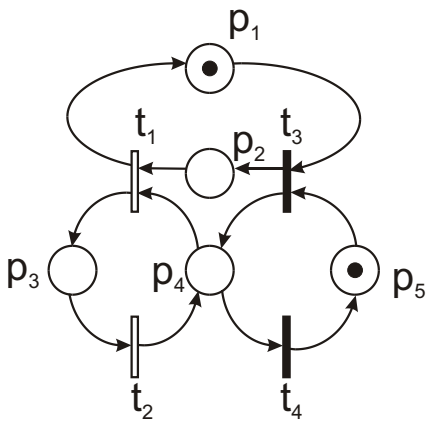
(b) Input file for PN-5

Figure 4.5: Petri net 5

```
pn5.res        Wed Dec 05 20:31:34 2012       1
 Input File = "pn5"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9
 P
   1  -1 -1  .  .  .  1  .  .  1
   2   1  . -1  .  .  .  .  .  .
   3   .  . -1  1  .  .  .  .  .
   4   .  1  . -1  .  .  .  .  .
   5   .  .  1  . -1 -1  .  .  .
   6   .  .  .  1  .  . -1  .  .
   7   .  .  .  1  .  .  . -1  .
   8   .  .  .  .  .  .  1  1 -1

 Initial Marking : ( 1 0 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 _____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 )
  2: ( 0 0 0 1 0 0 0 0 )
  3: ( 0 0 0 0 1 0 0 0 )
  4: ( 0 0 0 0 0 1 0 0 )
  5: ( 0 0 0 0 0 0 1 0 )
  6: ( 0 0 0 0 0 0 0 1 )
  7: ( 0 1 1 0 0 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 )
  2: ( 0 0 0 1 0 0 0 0 )
  3: ( 0 0 0 0 0 1 0 0 )
  4: ( 0 0 0 0 0 0 1 0 )
  5: ( 0 0 0 0 0 0 0 1 )


 This is An LESP
```

(c) Output file with LESP for PN-5

Figure 4.5 (cont.): Petri net 5



(a) PN-6

```
pn6        Wed Nov 21 22:51:55 2012       1
5 4
0 0 1 0
1 0 0 0
0 1 0 0
1 0 0 1
0 0 1 0
1 0 0 0
0 0 1 0
1 0 0 0
0 1 1 0
0 0 0 1
1 0 0 0 1
0 0 1 1
```

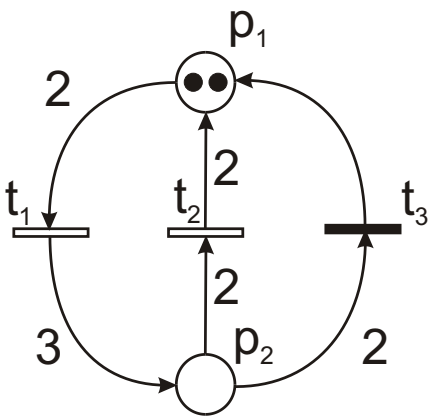
(b) Input file for PN-6

Figure 4.6: Petri net 6

```
pn6.res       Wed Dec 05 20:33:54 2012       1
 Input File = "pn6"

 Incidence Matrix :

    T  1  2  3  4
 P
   1   1  . -1  .
   2  -1  .  1  .
   3   1 -1  .  .
   4  -1  1  1 -1
   5   .  . -1  1

 Initial Marking : ( 1 0 0 0 1 )

 There is an LESP for this (fully controlled) PN

 _____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 1 )
  2: ( 1 0 0 1 0 )
  3: ( 0 1 0 1 0 )
  4: ( 1 0 1 0 0 )
  5: ( 0 1 1 0 0 )


 List of Controllable Transitions
 --------------------------------
 t3 t4


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 1 )
  2: ( 1 0 0 1 0 )
  3: ( 0 1 0 1 0 )
  4: ( 1 0 1 0 0 )
  5: ( 0 1 1 0 0 )


 This is An LESP
```

(c) Output file with LESP for PN-6

Figure 4.6 (cont.): Petri net 6



(a) PN-7

```
pn7        Wed Nov 21 23:51:12 2012       1
2 3
2 0 0
0 2 2
0 2 1
3 0 0
2 0
0 0 1
```

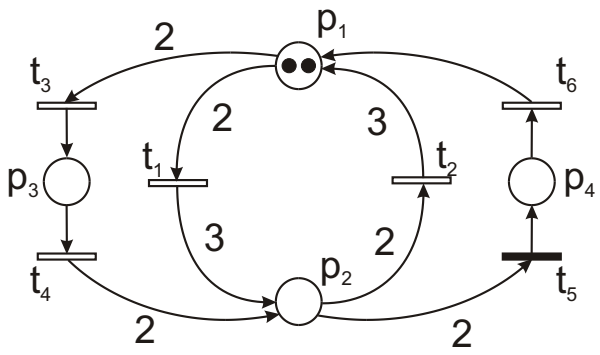
(b) Input file for PN-7

Figure 4.7: Petri net 7

```
pn7.res       Wed Dec 05 20:39:43 2012       1
 Input File = "pn7"

 Incidence Matrix :

    T  1  2  3
P
   1  -2  2  1
   2   3 -2 -2

 Initial Marking : ( 2 0 )

 There is an LESP for this (fully controlled) PN

_____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 0 2 )
  2: ( 2 0 )


 List of Controllable Transitions
 --------------------------------
 t3


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 2 )
  2: ( 2 0 )


 This is An LESP
```

(c) Output file with LESP for PN-7

Figure 4.7 (cont.): Petri net 7



(a) PN-8

```
pn8        Thu Dec 06 00:04:55 2012       1
4 6
2 0 2 0 0 0
0 2 0 0 2 0
0 0 0 1 0 0
0 0 0 0 0 1
0 3 0 0 0 1
3 0 0 2 0 0
0 0 1 0 0 0
0 0 0 0 1 0
2 0 0 0
0 0 0 0 1 0
```

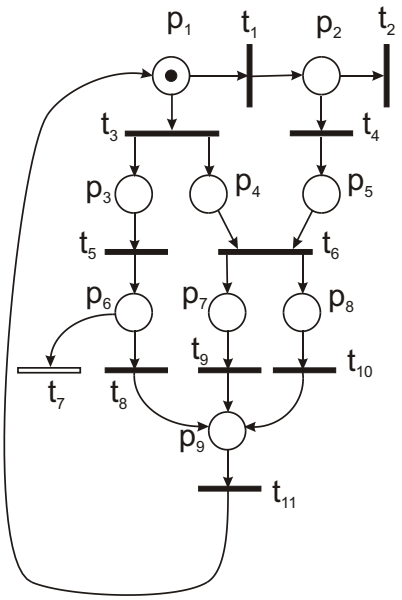
(b) Input file for PN-8

Figure 4.8: Petri net 8

```
pn8.res        Wed Dec 05 21:23:47 2012        1
 Input File = "pn8"

 Incidence Matrix :

    T  1  2  3  4  5  6
P
   1  -2  3 -2  .  .  1
   2   3 -2  .  2 -2  .
   3   .  .  1 -1  .  .
   4   .  .  .  .  1 -1

 Initial Marking : ( 2 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 0 0 1 0 )
  2: ( 1 0 0 1 )
  3: ( 0 0 0 2 )
  4: ( 0 2 0 0 )
  5: ( 2 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t5


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 1 0 )
  2: ( 1 0 0 1 )
  3: ( 0 0 0 2 )
  4: ( 0 2 0 0 )
  5: ( 2 0 0 0 )


 This is An LESP
```

(c) Output file with LESP for PN-8

Figure 4.8 (cont.): Petri net 8



(a) PN-9

```
pn9        Wed Dec 12 00:17:27 2012       1
9 11
1 0 1 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 0
1 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 1 0 0 0
```

(b) Input file for PN-9

Figure 4.9: Petri net 9

```
pn9.res        Thu Dec 06 11:07:17 2012        1
 Input File = "pn9"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9 10 11
 P
   1 -1  . -1  .  .  .  .  .  .  .  1
   2  1 -1  . -1  .  .  .  .  .  .  .
   3  .  .  1  . -1  .  .  .  .  .  .
   4  .  .  1  .  . -1  .  .  .  .  .
   5  .  .  .  1  . -1  .  .  .  .  .
   6  .  .  .  .  1  . -1 -1  .  .  .
   7  .  .  .  .  .  1  .  . -1  .  .
   8  .  .  .  .  .  1  .  .  . -1  .
   9  .  .  .  .  .  .  .  1  1  1 -1

 Initial Marking : ( 1 0 0 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 _____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 0 )
  2: ( 0 0 1 0 0 0 0 0 0 )
  3: ( 0 0 0 0 0 1 0 0 0 )
  4: ( 0 0 0 0 0 0 1 0 0 )
  5: ( 0 0 0 0 0 0 0 1 0 )
  6: ( 0 0 0 0 0 0 0 0 1 )
  7: ( 0 0 0 1 1 0 0 0 0 )
  8: ( 0 1 0 1 0 0 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1 t2 t3 t4 t8


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 0 )
  2: ( 0 0 0 0 0 0 1 0 0 )
  3: ( 0 0 0 0 0 0 0 1 0 )
  4: ( 0 0 0 0 0 0 0 0 1 )
  5: ( 0 0 0 1 1 0 0 0 0 )
  6: ( 0 1 0 1 0 0 0 0 0 )
```

(c) Output file with LESP for PN-9

Figure 4.9 (cont.): Petri net 9

```
pn9.res        Thu Dec 06 11:07:17 2012        2


The loop-test failed for the minimal_element: ( 1 0 0 0 0 0 0 0 0 )

The loop-test failed for the minimal_element: ( 0 0 0 0 0 0 1 0 0 )

The loop-test failed for the minimal_element: ( 0 0 0 0 0 0 0 1 0 )

The loop-test failed for the minimal_element: ( 0 0 0 0 0 0 0 0 1 )

 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 0 1 1 0 0 0 0 )
  2: ( 0 1 0 1 0 0 0 0 0 )
  3: ( 2 0 0 0 0 0 0 0 0 )
  4: ( 1 1 0 0 0 0 0 0 0 )
  5: ( 1 0 1 0 0 0 0 0 0 )
  6: ( 1 0 0 1 0 0 0 0 0 )
  7: ( 1 0 0 0 1 0 0 0 0 )
  8: ( 1 0 0 0 0 1 0 0 0 )
  9: ( 1 0 0 0 0 0 1 0 0 )
 10: ( 1 0 0 0 0 0 0 1 0 )
 11: ( 1 0 0 0 0 0 0 0 1 )
 12: ( 1 0 0 0 0 0 1 0 0 )
 13: ( 0 1 0 0 0 0 1 0 0 )
 14: ( 0 0 1 0 0 0 1 0 0 )
 15: ( 0 0 0 1 0 0 1 0 0 )
 16: ( 0 0 0 0 1 0 1 0 0 )
 17: ( 0 0 0 0 0 1 1 0 0 )
 18: ( 0 0 0 0 0 0 2 0 0 )
 19: ( 0 0 0 0 0 0 1 1 0 )
 20: ( 0 0 0 0 0 0 1 0 1 )
 21: ( 1 0 0 0 0 0 0 1 0 )
 22: ( 0 1 0 0 0 0 0 1 0 )
 23: ( 0 0 1 0 0 0 0 1 0 )
 24: ( 0 0 0 1 0 0 0 1 0 )
 25: ( 0 0 0 0 1 0 0 1 0 )
 26: ( 0 0 0 0 0 1 0 1 0 )
 27: ( 0 0 0 0 0 0 1 1 0 )
 28: ( 0 0 0 0 0 0 0 2 0 )
 29: ( 0 0 0 0 0 0 0 1 1 )
 30: ( 1 0 0 0 0 0 0 0 1 )
 31: ( 0 1 0 0 0 0 0 0 1 )
 32: ( 0 0 1 0 0 0 0 0 1 )
 33: ( 0 0 0 1 0 0 0 0 1 )
 34: ( 0 0 0 0 1 0 0 0 1 )
 35: ( 0 0 0 0 0 1 0 0 1 )
 36: ( 0 0 0 0 0 0 1 0 1 )
 37: ( 0 0 0 0 0 0 0 1 1 )
```

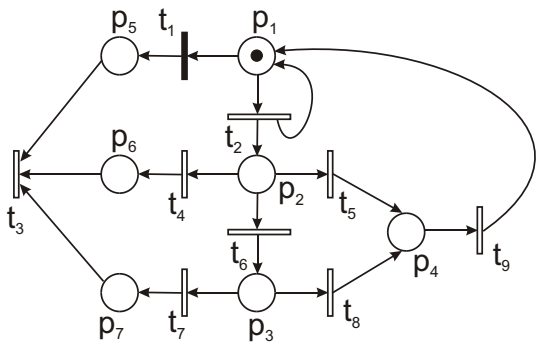
(d) Output file with LESP for PN-9 (cont.)

Figure 4.9 (cont.): Petri net 9

```
 38: ( 0 0 0 0 0 0 0 0 2 )


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 0 1 1 0 0 0 0 )
  2: ( 0 1 0 1 0 0 0 0 0 )
  3: ( 2 0 0 0 0 0 0 0 0 )
  4: ( 1 1 0 0 0 0 0 0 0 )
  5: ( 1 0 0 1 0 0 0 0 0 )
  6: ( 1 0 0 0 1 0 0 0 0 )
  7: ( 1 0 0 0 0 0 1 0 0 )
  8: ( 1 0 0 0 0 0 0 1 0 )
  9: ( 1 0 0 0 0 0 0 0 1 )
 10: ( 0 1 0 0 0 0 1 0 0 )
 11: ( 0 0 0 1 0 0 1 0 0 )
 12: ( 0 0 0 0 1 0 1 0 0 )
 13: ( 0 0 0 0 0 0 2 0 0 )
 14: ( 0 0 0 0 0 0 1 1 0 )
 15: ( 0 0 0 0 0 0 1 0 1 )
 16: ( 0 1 0 0 0 0 0 1 0 )
 17: ( 0 0 0 1 0 0 0 1 0 )
 18: ( 0 0 0 0 1 0 0 1 0 )
 19: ( 0 0 0 0 0 0 0 2 0 )
 20: ( 0 0 0 0 0 0 0 1 1 )
 21: ( 0 1 0 0 0 0 0 0 1 )
 22: ( 0 0 0 1 0 0 0 0 1 )
 23: ( 0 0 0 0 1 0 0 0 1 )
 24: ( 0 0 0 0 0 0 0 0 2 )


 This is An LESP
```

(e) Output file with LESP for PN-9 (cont.)

Figure 4.9 (cont.): Petri net 9



(a) PN-10

```
pn10        Wed Nov 21 23:09:39 2012       1
7 9
1 0 0 0 0 1 0 0 0
0 1 0 1 0 0 1 0 0
0 0 1 0 0 0 0 1 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
```

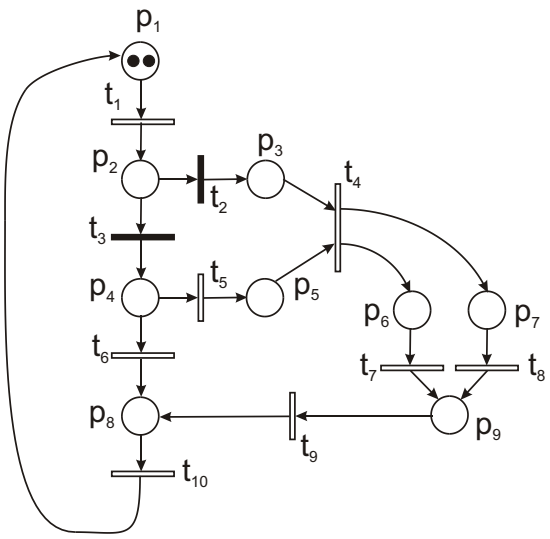
(b) Input file for PN-10

Figure 4.10: Petri net 10

```
pn10.res       Wed Dec 05 21:09:27 2012       1
 Input File = "pn10"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9
 P
   1  *S  .  .  .  1 -1  .  .  .
   2   1 -1  . -1  .  . -1  .  .
   3   .  1 -1  .  .  .  . -1  .
   4   .  .  1  1 -1  .  .  .  .
   5   .  .  .  .  .  1  .  . -1
   6   .  .  .  .  .  .  1  . -1
   7   .  .  .  .  .  .  .  1 -1

 Initial Marking : ( 1 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 0 0 )
  2: ( 0 1 0 0 0 0 0 )
  3: ( 0 0 1 0 0 0 0 )
  4: ( 0 0 0 1 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t1


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------


 The set of minimal markings is empty !
 Not an LESP !
```

(c) Output file with LESP for PN-10

Figure 4.10 (cont.): Petri net 10



(a) PN-11

```
pn11        Wed Dec 05 22:39:04 2012       1
9 10
1 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 1 1 0 0
2 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0
```

(b) Input file for PN-11

Figure 4.11: Petri net 11

```
pn11.res        Wed Dec 05 22:10:00 2012        1
 Input File = "pn11"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9 10
 P
   1 -1  .  .  .  .  .  .  .  .  1
   2  1 -1 -1  .  .  .  .  .  .  .
   3  .  1  . -1  .  .  .  .  .  .
   4  .  .  1  . -1 -1  .  .  .  .
   5  .  .  . -1  1  .  .  .  .  .
   6  .  .  .  1  .  . -1  .  .  .
   7  .  .  .  1  .  .  . -1  .  .
   8  .  .  .  .  .  1  .  .  1 -1
   9  .  .  .  .  .  .  1  1 -1  .

 Initial Marking : ( 2 0 0 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

_____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 1 )
  2: ( 0 1 0 0 0 0 0 0 1 )
  3: ( 0 0 1 0 0 0 0 0 1 )
  4: ( 0 0 0 1 0 0 0 0 1 )
  5: ( 0 0 0 0 1 0 0 0 1 )
  6: ( 0 0 0 0 0 1 0 0 1 )
  7: ( 0 0 0 0 0 0 1 0 1 )
  8: ( 0 0 0 0 0 0 0 1 1 )
  9: ( 0 0 0 0 0 0 0 0 2 )
 10: ( 1 0 0 0 0 0 0 1 0 )
 11: ( 0 1 0 0 0 0 0 1 0 )
 12: ( 0 0 1 0 0 0 0 1 0 )
 13: ( 0 0 0 1 0 0 0 1 0 )
 14: ( 0 0 0 0 1 0 0 1 0 )
 15: ( 0 0 0 0 0 1 0 1 0 )
 16: ( 0 0 0 0 0 0 1 1 0 )
 17: ( 0 0 0 0 0 0 0 2 0 )
 18: ( 1 0 0 0 0 0 1 0 0 )
 19: ( 0 1 0 0 0 0 1 0 0 )
 20: ( 0 0 1 0 0 0 1 0 0 )
 21: ( 0 0 0 1 0 0 1 0 0 )
 22: ( 0 0 0 0 1 0 1 0 0 )
 23: ( 0 0 0 0 0 1 1 0 0 )
 24: ( 0 0 0 0 0 0 2 0 0 )
```

(c) Output file with LESP for PN-11

Figure 4.11 (cont.): Petri net 11

```
 25: ( 1 0 0 0 0 1 0 0 0 )
 26: ( 0 1 0 0 0 1 0 0 0 )
 27: ( 0 0 1 0 0 1 0 0 0 )
 28: ( 0 0 0 1 0 1 0 0 0 )
 29: ( 0 0 0 0 1 1 0 0 0 )
 30: ( 0 0 0 0 0 2 0 0 0 )
 31: ( 1 0 0 0 1 0 0 0 0 )
 32: ( 0 1 0 0 1 0 0 0 0 )
 33: ( 0 0 1 0 1 0 0 0 0 )
 34: ( 0 0 0 1 1 0 0 0 0 )
 35: ( 1 0 0 1 0 0 0 0 0 )
 36: ( 0 1 0 1 0 0 0 0 0 )
 37: ( 0 0 1 1 0 0 0 0 0 )
 38: ( 0 0 0 2 0 0 0 0 0 )
 39: ( 1 0 1 0 0 0 0 0 0 )
 40: ( 0 1 1 0 0 0 0 0 0 )
 41: ( 1 1 0 0 0 0 0 0 0 )
 42: ( 0 2 0 0 0 0 0 0 0 )
 43: ( 2 0 0 0 0 0 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t2 t3


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 1 )
  2: ( 0 1 0 0 0 0 0 0 1 )
  3: ( 0 0 1 0 0 0 0 0 1 )
  4: ( 0 0 0 1 0 0 0 0 1 )
  5: ( 0 0 0 0 1 0 0 0 1 )
  6: ( 0 0 0 0 0 1 0 0 1 )
  7: ( 0 0 0 0 0 0 1 0 1 )
  8: ( 0 0 0 0 0 0 0 1 1 )
  9: ( 0 0 0 0 0 0 0 0 2 )
 10: ( 1 0 0 0 0 0 0 1 0 )
 11: ( 0 1 0 0 0 0 0 1 0 )
 12: ( 0 0 1 0 0 0 0 1 0 )
 13: ( 0 0 0 1 0 0 0 1 0 )
 14: ( 0 0 0 0 1 0 0 1 0 )
 15: ( 0 0 0 0 0 1 0 1 0 )
 16: ( 0 0 0 0 0 0 1 1 0 )
 17: ( 0 0 0 0 0 0 0 2 0 )
 18: ( 1 0 0 0 0 0 1 0 0 )
 19: ( 0 1 0 0 0 0 1 0 0 )
 20: ( 0 0 1 0 0 0 1 0 0 )
 21: ( 0 0 0 1 0 0 1 0 0 )
```

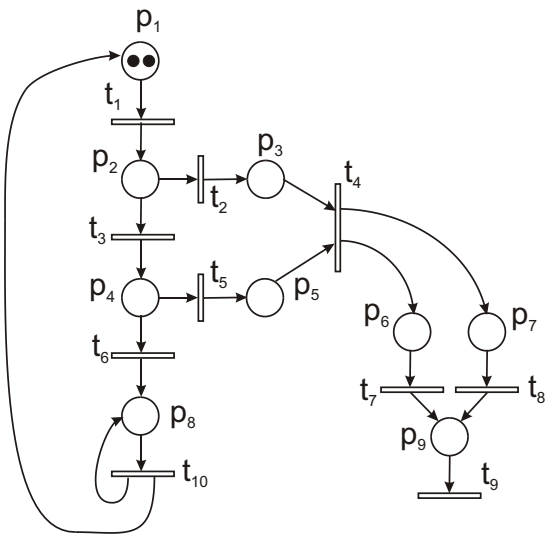
(d) Output file with LESP for PN-11 (cont.)

Figure 4.11 (cont.): Petri net 11

```
 22: ( 0 0 0 0 1 0 1 0 0 )
 23: ( 0 0 0 0 0 1 1 0 0 )
 24: ( 0 0 0 0 0 0 2 0 0 )
 25: ( 1 0 0 0 0 1 0 0 0 )
 26: ( 0 1 0 0 0 1 0 0 0 )
 27: ( 0 0 1 0 0 1 0 0 0 )
 28: ( 0 0 0 1 0 1 0 0 0 )
 29: ( 0 0 0 0 1 1 0 0 0 )
 30: ( 0 0 0 0 0 2 0 0 0 )
 31: ( 1 0 0 0 1 0 0 0 0 )
 32: ( 0 1 0 0 1 0 0 0 0 )
 33: ( 0 0 1 0 1 0 0 0 0 )
 34: ( 1 0 0 1 0 0 0 0 0 )
 35: ( 0 1 0 1 0 0 0 0 0 )
 36: ( 0 0 1 1 0 0 0 0 0 )
 37: ( 1 0 1 0 0 0 0 0 0 )
 38: ( 0 1 1 0 0 0 0 0 0 )
 39: ( 1 1 0 0 0 0 0 0 0 )
 40: ( 0 2 0 0 0 0 0 0 0 )
 41: ( 2 0 0 0 0 0 0 0 0 )


 This is An LESP
```

(e) Output file with LESP for PN-11 (cont.)

Figure 4.11 (cont.): Petri net 11



(a) PN-12

```
pn12        Wed Dec 05 22:39:35 2012       1
9 10
1 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 1 1 0 0
2 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

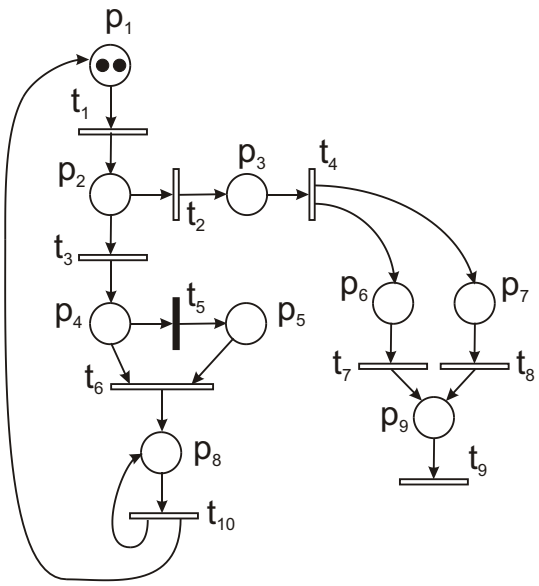
(b) Input file for PN-12

Figure 4.12: Petri net 12

```
pn12.res        Wed Dec 05 22:39:42 2012        1
 Input File = "pn12"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9 10
P
  1  -1  .  .  .  .  .  .  .  .  1
  2   1 -1 -1  .  .  .  .  .  .  .
  3   .  1  . -1  .  .  .  .  .  .
  4   .  .  1  . -1 -1  .  .  .  .
  5   .  .  . -1  1  .  .  .  .  .
  6   .  .  .  1  .  . -1  .  .  .
  7   .  .  .  1  .  .  . -1  .  .
  8   .  .  .  .  .  1  .  .  . *S
  9   .  .  .  .  .  .  1  1 -1  .

 Initial Marking : ( 2 0 0 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 1 0 0 0 0 0 0 0 0 )
  2: ( 0 1 0 0 0 0 0 0 0 )
  3: ( 0 0 0 1 0 0 0 0 0 )
  4: ( 0 0 0 0 0 0 0 1 0 )


 List of Controllable Transitions
 --------------------------------



 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 0 0 0 0 0 1 0 )


 This is An LESP
```

(c) Output file with LESP for PN-12

Figure 4.12 (cont.): Petri net 12



(a) PN-13

```
pn13        Wed Dec 05 22:37:19 2012       1
9 10
1 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 1
0 0 0 0 0 0 1 1 0 0
2 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
```

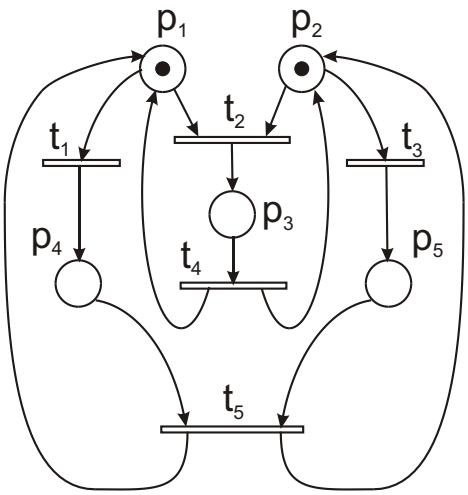
(b) Input file for PN-13

Figure 4.13: Petri net 13

```
pn13.res        Wed Dec 05 22:34:23 2012        1
 Input File = "pn13"

 Incidence Matrix :

    T  1  2  3  4  5  6  7  8  9 10
P
   1  -1  .  .  .  .  .  .  .  .  1
   2   1 -1 -1  .  .  .  .  .  .  .
   3   .  1  . -1  .  .  .  .  .  .
   4   .  .  1  . -1 -1  .  .  .  .
   5   .  .  .  .  1 -1  .  .  .  .
   6   .  .  .  1  .  . -1  .  .  .
   7   .  .  .  1  .  .  . -1  .  .
   8   .  .  .  .  .  1  .  .  . *S
   9   .  .  .  .  .  .  1  1 -1  .

 Initial Marking : ( 2 0 0 0 0 0 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 0 0 0 0 0 0 0 1 0 )
  2: ( 1 0 0 0 1 0 0 0 0 )
  3: ( 0 1 0 0 1 0 0 0 0 )
  4: ( 0 0 0 1 1 0 0 0 0 )
  5: ( 1 0 0 1 0 0 0 0 0 )
  6: ( 0 1 0 1 0 0 0 0 0 )
  7: ( 0 0 0 2 0 0 0 0 0 )
  8: ( 1 1 0 0 0 0 0 0 0 )
  9: ( 0 2 0 0 0 0 0 0 0 )
 10: ( 2 0 0 0 0 0 0 0 0 )


 List of Controllable Transitions
 --------------------------------
 t5


 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 0 0 0 0 0 1 0 )
  2: ( 0 0 0 1 1 0 0 0 0 )
  3: ( 0 0 0 2 0 0 0 0 0 )


 This is An LESP
```

(c) Output file with LESP for PN-13

Figure 4.13 (cont.): Petri net 13



(a) PN-14

```
pn14        Wed Dec 05 22:42:01 2012       1
5 5
1 1 0 0 0
0 1 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 1
0 0 0 1 1
0 0 0 1 1
0 1 0 0 0
1 0 0 0 0
0 0 1 0 0
1 1 0 0 0
0 0 0 0 0
```

(b) Input file for PN-14

Figure 4.14: Petri net 14

```
pn14.res        Wed Dec 05 22:42:10 2012        1
 Input File = "pn14"

 Incidence Matrix :

    T  1  2  3  4  5
 P
   1  -1 -1  .  1  1
   2   . -1 -1  1  1
   3   .  1  . -1  .
   4   1  .  .  . -1
   5   .  .  1  . -1

 Initial Marking : ( 1 1 0 0 0 )

 There is an LESP for this (fully controlled) PN

 ____________________________________________


 Minimal Elements of the fully controlled Net
 -------------------------------------------

  1: ( 0 0 1 0 0 )
  2: ( 1 0 0 0 1 )
  3: ( 0 0 0 1 1 )
  4: ( 0 1 0 1 0 )
  5: ( 1 1 0 0 0 )


 List of Controllable Transitions
 --------------------------------



 (Final) Minimal Elements of the control-invariant set
 -----------------------------------------------------

  1: ( 0 0 1 0 0 )
  2: ( 1 0 0 0 1 )
  3: ( 0 0 0 1 1 )
  4: ( 0 1 0 1 0 )
  5: ( 1 1 0 0 0 )


 This is An LESP
```

(c) Output file with LESP for PN-14

Figure 4.14 (cont.): Petri net 14

Chapter 5

Future Work

The algorithms discussed in this report are applicable only to a class of Petri nets for which the structure of the net conforms to a minimal marking set for the partially controlled net being right-closed and this property, in general is true for free-choice nets. The net discussed in [20] given in figure 5.1 is not free-choice as places $card(p_7) \geq 1$ and $card(p_8) \geq 1$ and $({}^\bullet(p_7^\bullet)_{N_1})_{N_1} = ({}^\bullet(p_8^\bullet)_{N_1})_{N_1} = \{p_7, p_8\}$. This net can be made free-choice using transformation/reduction techniques to the net shown in figure 5.2. Here, $({}^\bullet(p_7^\bullet)_{N_2})_{N_2} = \{p_7\}$ and $({}^\bullet(p_8^\bullet)_{N_2})_{N_2} = \{p_8\}$. On executing the algorithm to obtain the coverability graph for this net, millions of nodes are generated and depending on the resource constraint of the computing device, the algorithm can take very long to run. This is also attributed to number of concurrent transitions in this net. Further, with the increase in complexity of the system with a large number of places and transitions the computations become tedious.

Complex nets like this can be analyzed using techniques that can effectively reduce them into an equivalent combination of smaller nets thereby reducing the computation time. Future work will involve extending the algorithms for efficient computation of the LESP test for such complex nets. Furthermore, with the increasing complexity of the PN, the time taken by the algorithm to compute the minimal markings increase manifold. Currently the implementation is not optimized for performance. Future work will also involve the improvements in performance of the algorithm with appropriate modifications in the procedure constructs and data structures in the code.

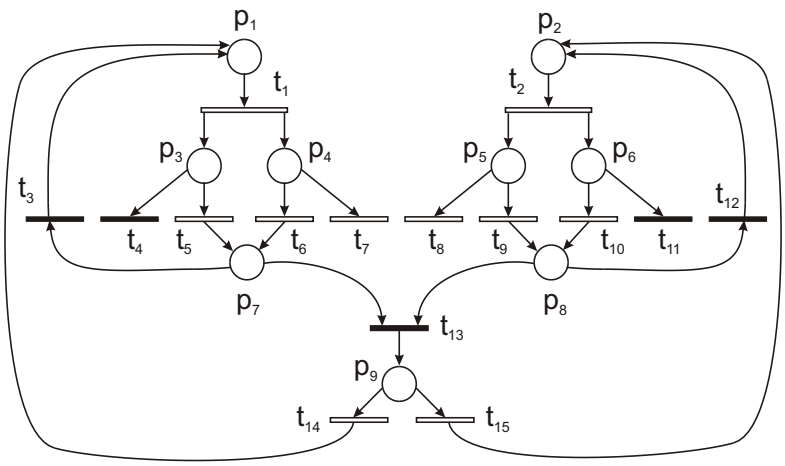


Figure 5.1: Petri net N1

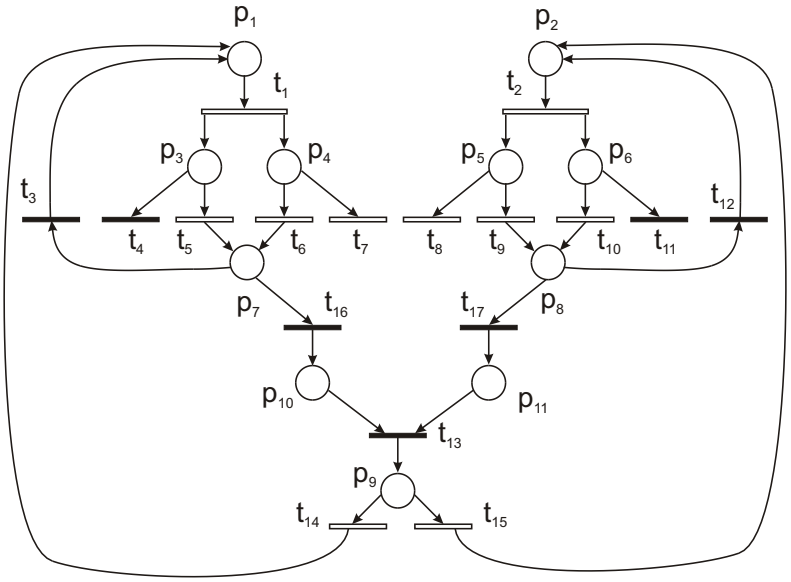


Figure 5.2: Petri net N1 transformed to a free-choice Petri net N2

References

[1] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2007.

[2] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181 – 185, 1985.

[3] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[4] T. Murata, "Petri Nets: Properties, Analysis and Applications," in *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[5] R. Sreenivas, "On the existence of supervisory policies that enforce liveness in discrete-event dynamic systems modeled by controlled petri nets," *Automatic Control, IEEE Transactions on*, vol. 42, no. 7, pp. 928–945, 1997.

[6] R. Sreenivas, "On the existence of supervisory policies that enforce liveness in partially controlled free-choice petri nets," *IEEE Transactions on Automatic Control*, vol. 57, no. 2, pp. 435–449, 2012.

[7] N. Somnath and R. Sreenivas, "On deciding the existence of a liveness enforcing supervisory policy in a class of partially-controlled general free-choice petri nets," *IEEE Transactions on Automation Science and Engineering*, 2012. Submitted.

[8] M. Hack, "Analysis of production schemata by petri nets," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1972.

[9] J. Desel and J. Esparza, *Free Choice Petri Nets*. Cambridge, UK: Cambridge Tracts in Theoretical Computer Science, Cambridge, UK, 1995.

[10] R. Valk and M. Jantzen, "The residue of vector sets with applications to decidability problems in petri nets," *Acta Informatica*, vol. 21, pp. 643–674, 1985.

[11] H.-C. Yen and C.-L. Chen, "On minimal elements of upward-closed sets," *Theor. Comput. Sci.*, vol. 410, pp. 2442–2452, May 2009.

[12] S. R. Kosaraju, "Decidability of reachability in vector addition systems (preliminary version," in *In STOC*, pp. 267–281, ACM, 1982.

[13] E. W. Mayr, "An algorithm for the general petri net reachability problem," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, (New York, NY, USA), pp. 238–246, ACM, 1981.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[15] C. Reutenauer, *The mathematics of Petri nets*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[16] R. Sreenivas, "Some observations on supervisory policies that enforce liveness in partially controlled free-choice petri nets," *Mathematics and Computers in Simulation*, vol. 70, no. 5, pp. 266–274, 2006.

[17] P. J. Ramadge and W. M. Wonham, "Modular feedback logic for discrete event systems," *SIAM J. Control Optim.*, vol. 25, pp. 1202–1218, Sept. 1987.

[18] M. Berkelaar, K. Eikland, and P. Notebaert, "lp_solve, open source (mixed-integer) linear programming system," Version 5.0.0.0 dated 1 May 2004.

[19] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software - Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[20] R. Sreenivas, "On a decidable class of partially controlled petri nets with liveness enforcing supervisory policies," *IEEE Transactions on Systems, Man and Cybernetics*, 2012. To appear.