New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Electronic Theses and Dissertations

1-31-1993

Computer implementation of Mason's rule and software development of stochastic petri nets

Xiaoyong Zhao
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Databases and Information Systems Commons, and the Management Information Systems Commons

Recommended Citation
Zhao, Xiaoyong, "Computer implementation of Mason's rule and software development of stochastic petri nets" (1993). *Theses*. 2217.
https://digitalcommons.njit.edu/theses/2217

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

Computer Implementation of Mason's Rule and Software Development of Stochastic Petri Nets

by
Xiaoyong Zhao

A symbolic performance analysis approach for discrete event systems can be formulated based on the integration of Petri nets and Moment Generating Function concepts [1-3]. The key steps in the method include modeling a system with arbitrary stochastic Petri nets (ASPN), generation of state machine Petri nets with transfer functions, derivation of equivalent transfer functions, and symbolic derivation of transfer functions to obtain the performance measures. Since Mason's rule can be used to effectively derive the closed-form transfer function, its computer implementation plays a very important role in automating the above procedure. This thesis develops the computer implementation of Mason's rule (CIMR). The algorithms and their complexity analysis are also given. Examples are used to illustrate CIMR method's application for performance evaluation of ASPN and linear control systems. Finally, suggestions for future software development of ASPN are made.

COMPUTER IMPLEMENTATION OF MASON'S RULE AND SOFTWARE DEVELOPMENT OF STOCHASTIC PETRI NETS

by
Xiaoyong Zhao

A Thesis
Submitted to the Faulty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Department of Computer and Information Science

January 1993

APPROVAL PAGE

Computer Implementation of
Mason's Rule and
Software Development of Stochastic Petri Nets

Xiaoyong Zhao

12/15/92

Dr. MengChu Zhou, Thesis Adviser
Assistant Professor of Department of Electrical and Computer Engineering, and Center for Manufacturing Systems, NJIT

12/15/92

Dr. Daniel Chao, Committee Member
Assistant Professor of Department of Computer and Information Science, NJIT

12/15/92

Dr. David Wang, Committee Member
Assistant Professor of Department of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Xiaoyong Zhao

Degree: Master of Science in Computer and Information Science

Date: January 1993

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer and Information Science,
  New Jersey Institute of Technology, Newark, NJ, 1993
- Bachelor of Science in Electrical and Computer Engineering,
  University of Electronic Science and Technology of China, Chengdu, 1982

Major: Computer and Information Science

This thesis is dedicated
to
My parents, my sister, my wife
and
my son

ACKNOWLEDGMENT

I wish to express my sincere gratitude to my supervisor, Dr. MengChu Zhou, for his guidance, friendship, and moral support throughout this research. He taught me in discrete event dynamic system theory and methodology, took care of the research project and saw it through to its successful completion. I could not have finished this thesis without him.

I would like to thank the members of the thesis committee, Dr. Daniel Chao and Dr. David Wang for reviewing and commenting on my thesis report.

Finally, I would like to express here my deepest gratitude for my parents for their continuous support and encouragement throughout my studies. My greatest thanks go to my wife, Qing, for her patience and understanding in my whole work because I spent the many hours with my computer terminal instead of her and my son, and to my sister, Anna X. Zhao, for her special support and care during my academic years.

TABLE OF CONTENTS

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

OBJECTIVES AND MOTIVATION

This research work is motivated by the need in automating the moment generating function and Petri net based procedure for performance analysis of discrete event systems. Given a system in which an operation may take an arbitrarily distributed processing time, we can model this system as an arbitrary stochastic Petri net (ASPN). Then, the reachability graph is generated and transformed into a state machine Petri net with moment generating function included. The equivalent transfer functions are derived and performance measures are analyzed [1-3]. The method can result in a closed-form result for some classes of ASPNs. Since the transfer functions retain all the information of performance measures and thus often become very complex when the system state number grows, the human manipulation of this process becomes very difficult. The need arises to automate this process. One of the key steps is to use Mason's rule for derivation of an equivalent transfer function between the given nodes. Although the reduction methods can be used for some large and complex graphs, a computerized implementation of Mason's rule (CIMR) is more efficient and convenient.

Mason's rule was invented in 50s for signal flow graphs. It has been used for analysis of circuits and control systems. The computer manipulation of the Mason's rule recently receives attention and similar work is reported in [10] in order to determine the symbolic transfer function of a linear system with a SPICE-like system description language. The work presented in this thesis differs from the previous work in the following aspects:

1. Different motivations result in different system description environments;
2. The algorithms are improved in this work and the applications are enhanced;
3. The complexity analysis of the algorithms is conducted;

4. CIMR is applied to develop new software for ASPN.

The objectives of this thesis are to:

1. Present an efficient method to implement a computerized solution of the Mason's rule including forward path search, loop search, and non-touching loop check, etc.;

2. Provide the complexity analysis of the developed algorithms;

3. Propose a Stochastic Petri Net Language (SPNL) that describes a State Machine Petri Net;

4. Design and code a utility program (**cimr**) using C language to derive automatically an equivalent transfer function, which runs under UNIX;

5. Illustrate the application of **cimr** for the performance evaluation of discrete event system;

6. Illustrate the application of **cimr** for a complex net system in which it is very difficult to derive the transfer functions;

7. Propose a scheme to develop a synthesis software tool for performance analysis and evaluation of ASPN via moment generating function.

CHAPTER 2

INTRODUCTION TO PETRI NET AND METHODOLOGY

Carl A. Petri developed a net-theoretic approach to model and analyze communication systems [6]. Petri nets have been proven to be useful tool for the modeling, performance evaluation and analysis of discrete event dynamic system [12-13]. Specifically, they are useful for modeling systems with the following characteristics:

- *Concurrency or parallelism:* There are some systems, in which many operationstake place simultaneously.
- *Asynchronous operations:* Machines complete their operations in variable amountsof time and so the model must maintain the ordering of the occurrence of events.
- *Deadlock:* In this case, a state can be reached where none of the processes can continue. This can happen when two processes share two resources. The order by which these resources are used and released could produce a deadlock.
- *Conflict:* This may occur when two or more processes require a common resource at the same time. For Example, two workstations might share a common transport system or might want access to the same database.
- *Event driven:* The manufacturing system can be viewed as a sequence of discrete events. Since operations occur concurrently, the order of occurrence of events is not necessarily unique; it is one of many allowed by the system structure.

These types of systems have been difficult to accurately model with differential equations and queueing theory. Petri nets can provide accurate models for the following reasons:

- Petri nets capture the precedence relations and structural interactions of concurrent and asynchronous events.
- They are logical model derived from the knowledge of how the system works. As a result, they are easy to understand and their graphical nature is a good visual aid.
- Deadlock, conflicts, and buffer sizes can be modeled easily and concisely.
- Petri net models have a well developed mathematical foundation that allows a qualitative and quantitative analysis of the system.
- Petri net models can also be used to implement real-time control systems for a automated manufacturing system. They can sequence and coordinate the subsystems as a programming logic controller does.

2.1 Petri Net Structure and Graph

A petri net is composed of a set of **place** *P*, a set of **transition** *T*, an **input function** *I*, an **output function** *O*, and an **initial marking** m_0. A graph structure is often used for illustration of Petri nets where a circle "O" represents a place and a bar "|" represents a transition. An **arc** with an arrow from a place to a transition defines the place to be an input to the transition. Similarly, an output place is indicated by an arc from a transition to the place.

A formal definition used follows [3]:

An ordinary Petri Net is a five-tuple (P, T, I, O, m).

P = {p1, p2, ... p_n}, n>0, and is a finite set of places;

T = {t_1, t_2, ..., t_s}, s>0, is a finite set of transitions, $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$;

I: $P \times T \rightarrow N$ and is an input function that defines the set of directed arcs from P to T, where N={0, 1, 2, ...};

O: $P \times T \rightarrow N$ is an output function that defines the set of directed arcs from T to P;

m: P $\rightarrow$ N and is a marking whose i^{th} component represents the number of tokens in the i^{th} place. An initial marking is denoted by m_0;

The dynamic aspects of Petri net models are denoted by markings which are assignments of **tokens** to places of a Petri net. The execution of a Petri net is controlled by the number and distribution of tokens in the Petri Net. A transition is enabled if and only if each of its input places contains at least as many tokens as arcs exist from that place to the transition. When a transition is enabled, it may **fire**. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

The state of the Petri nets is defined by the marking. The change in state caused by firing a transition is defined by the next-state function. Given an initial state, the reachability set for the Petri net is the set of states that result from executing the Petri net. Both tree and graph have been used to represent the graph labeled with the present marking (i.e., the state) and the arcs represent transitions between states.

Figure 2.1 shows a simple Petri net. Here, tokens reside in places, travel along arcs, and their flow through the net is regulated by the transitions.

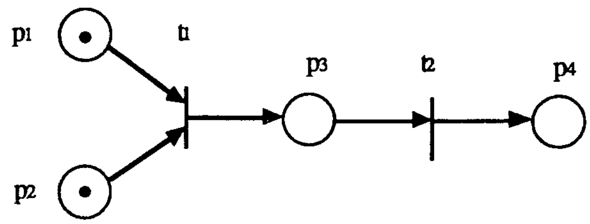
2.2 Behavioral Properties of Petri Nets

2.2.1 Liveness

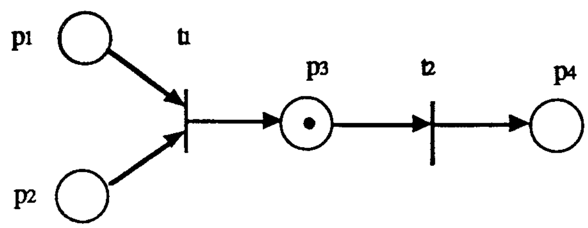
A Petri net is live with respect to a marking, if for any marking in $R\ (m_0)$, it is possible to fire any transition in the net. Liveness guarantees the absence of deadlock. Thus if a transition is live, it is always possible to maneuver the Petri net from its current marking to a marking which would allow the transition to fire.

2.2.2 Boundedness

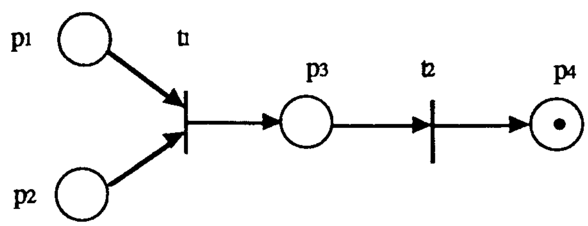
Boundedness is a generalization of safeness of a net with the situation that places can at



(a) A Petri net with initial marking



(b) Making after t 1 fires



(c) Making after t 2 fires

Figure 2.1: A Simple Petri Net

most hold a particular number of tokens. A place is k-bounded, if the number of tokens in that place cannot exceed an integer k, e.g., p is k-bounded if $m(p) \leq k$, $\forall\ m \in R$ (Z, m_0). If $\forall\ p \in P$, p is k-bounded, the Z is k-bounded. Since there are only a finite number of places in Petri net, we can find the k as the maximum of the bounds of each place and define a Petri net to be k-bounded if every place is k-bounded. In a manufacturing system, a bounded net implies that resource constraints have been met .

2.2.3 Conservativeness

A Petri net is conservative if, for any initial marking and a reachable marking $m \in R(m_0)$, there exists an n × vector x, each of whose component is non-zero such that

$$x^T m = x^T m_0$$

This says that the sum of the tokens weighted by x is constant.

2.2.4 Reversibility

A Petri net is resversible if for every $m \in R(m_0)$ then $m_0 \in R(m)$. Reversibility means that the initial mark is reachable from all reachable markings. This is important in a manufacturing system where failures occur and the system is able to be reinitialized.

2.3 Stochastic Petri Nets (SPN)

Stochastic Petri nets, evolved in late 1970's as Petri nets with exponential delay distributions. Important research work is referred to [18], [25-26], [27] and [28]. The researcher has contributed to theory, structural improvement, implementation and application of SPN in various fields as below.

- Communication Systems

  Communication systems have a main feature of synchronization. Florin and Natkin [27] by using the existing isomorphism of Markov Process and SPN, has modeled synchronous network queues by SPN. They derived ergodic criterion and steady-state solutions for the model.

- Local Area Networks

E. Gressier [29] has shown modeling of Ethernet protocol by SPN. The results are based upon simulation. M. A. Marsan [26] has computed a performance model for Carrier Sense Multiple Access with Collision Detection protocol (CSMA) of a bus LAN.

• Concurrent / Multiprocessor Systems

Much work has been done recently in these areas. SPN modeling for multiprocessor systems [25], interprocess communication, distributed file systems and concurrent task synchronization has been shown [30].

• Manufacturing Systems

SPN has also been used for modeling, design and control of manufacturing systems [1, 31, 32].

2.3.1 Timed Petri Nets (TPN)

Time can be included in a Petri net model by associating time with the transitions, to form a timed transition Petri net (TTPN), or with the places, resulting in a timed place Petri net (TPPN). Both representations are equivalent [15].

In a TTPN, the firing of a transition takes a certain amount of time. Note that this time is fixed, which makes TTPNs deterministic.

In a TPPN, a token enters a place and is unavailable for time d_i, after which it becomes available. In this net, only available tokens in a marking can enable a transition. During the unavailable period, another token may arrive in the place. A TPPN with all delays set zero reduces to an ordinary Petri net.

The properties of Petri nets analyzed in the previous section can be applied to timed Petri nets by using the incidence matrix. An alternate approach attempts to cast these nets in a system-theoretic framework [16, 17]. A minimal algebra is applied to the Petri net models and concepts analogous to transfer function, input-output models, feedback, etc., are developed. The potential for this approach lies in its ability to build on analogies with traditional control theory concepts. Its is still to be proven that useful analogies exist and can be extended to systems that include shared resources.

2.3.2 Definition of Stochastic Petri Net (SPN)

Molloy has defined SPN as a Petri nets in which each transition firing delay is associated with an exponentially distributed random variable [18].

Stochastic Petri Nets (SPN) is defined as a six-tuple (P, T, I, O, m, F)

Where,

$P = \{p_1, p_2, ..., p_n\}$, n>0, is a finite set of place;

$T = \{t_1, t_2, ..., t_s\}$, s>0, is a finite set of transitions, $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.;

I: $P \times T \rightarrow N$ and is an input function that defines the set of directed arcs from P to T, where, $N = \{0, 1, 2, ...\}$;

O: $P \times T \rightarrow N$ is an output function that defines the set of directed arcs from P to T;

m: $P \rightarrow N$ is a marking whose i^{th} component represents the number of tokens in the i^{th} place. An initial marking is denoted by m_0;

F : $T \rightarrow R$ is a firing time delay function with an stochastic distribution function.

2.3.3 Extensions to SPN

Generalized stochastic Petri nets (GSPNs) [35-36] incorporate both timed transitions and immediate transitions. GSPNs permit the use of inhibitor arcs, priority functions, and random switches. These additional modeling capabilities follow the equivalence with Markov chains. The steady-state probabilities obtained from the Markov chain are used to compute the expected number of tokens in a place. Thus exact solutions can be derived by solving the equivalent Markov models; thereby deriving the performance measures.

Marsan et al.[25] defined stochastic Petri nets where arbitrary distributed random firing delays can be associated to transitions. Zhou et al.[3] call the nets arbitrary stochastic Petri nets (ASPN). An ASPN is a six-tuple Petri net (P, T, I, O, m, f) where $f:T \rightarrow R$ is a firing delay function of arbitrary distributions [3]. ASPN extends above various kinds of Petri nets and allows various mechanisms such as inhibitor arcs, probability arcs, and priority firing.

Moment generating function (MGF) [1, 3] based methods approach the performance analysis of SPN in a different way from the above methods which use Markov models. Instead of solving the resulting Markov models, MGF-based methods derive the MGF of interesting performance measures. For arbitrary stochastic Petri nets, where arbitrary distributions are incorporated into stochastic Petri nets, the method can be used to find a lower and an upper bound. Exact solutions can be obtained for those Petri nets where transitions with non-exponential distributions are converted into subnets in which each transition has a firing delay of exponential distribution by using the existing techniques [22, 34].

Other methods have been used for modeling and performance analysis of various DEDS, form example, Markov analysis, queuing networks, perturbation analysis, and discrete event simulation [24, 41, 39]. Compared with these methods, Petri nets have their unique features. The advantages of Petri nets include their ease of modeling, duo to their graphical representation, and their ability to model various event-driven system characteristics: concurrency, conflicts, non-determinism, and mutual exclusion. In addition, they are more compact models than Markov models. At the design stage, the use of Petri nets avoids the need to enumerate all states, which is often impossible in modern manufacturing systems.

2.3.4 Transfer Function Analysis Method

The evaluation and analysis of Stochastic Petri Nets is proposed to be done by using a methodology which is based on the concepts of Markov theory, control systems and symbolic computation methods. It is called the Moment Generating Function (MGF) or the Transfer Function approach. This technique has been recently formulated by Guo, DiCesare and Zhou [1]. The implementation of this technique involves five main steps:

a) *Reachability Graph Generation and Transformation to State Machine Petri Net*

It has been shown by Molloy and others [18], that the reachability graph of a bounded SPN is isomorphic to a finite Markov Chain. Using this theorem, each marking in the reachability graph of the underlying PN is considered as a place of a state machine Petri net.

b) *Computation of MGF and Transfer functions*

Each transitions of the transformed state machine Petri net is assigned a transfer function, which is the product of branch probability and moment generating function. The transfer function depends upon the firing distribution of the transition and number of markings directly reachable from the marking under consideration.

c) *Computation of Equivalent Transfer function*

The application of Mason's rule or net reduction techniques leads to computation of equivalent transfer functions of the net. The equivalent function is useful for study and evaluation of the net.

d) *Computation of Performance Parameters*

Finally, computation of various performance parameters of the SPN model is done by computing derivative of equivalent transfer functions. The implementation of this technique can provide important analytical parameters of the modeled system such as fault rate, conflict rate, deadlocks, production rate, cycle time and system throughput.

The main advantages of this technique are:

1) It does not require simulation of transition firing delays for generation of reachability graph. This technique utilizes the reachability graph of the underlying Petri and imparts the timing information while analysis of the graph.

2) It identifies all possible system states by SPN execution and also indicates system parameters.

3) It provides detection of conflicts and deadlocks in the modeled system, for example, resource allocation problem and buffer overflow problem.

4) It implements net reduction techniques to reduce complexity of the net and ease its analysis.

5) Computation of performance indices of the modeled system. For example, steady-state probabilities, system throughput, fault rate, etc.

- **Definition of Moment Generating Function (MGF)**

For a random variable t with probability density function f(t), its Moment generating function (MGF) is defined [38] as

Discrete case

$$M(s) = \sum_{-\infty}^{\infty} e^{st} f(t)dt$$

Continuous case

$$M(s) = \int_{-\infty}^{\infty} e^{st} f(t)dt$$

where s is an arbitrary parameter and *f(t)* is a probability density function of random variable t. The n-th derivatives of MGF generates n-th moments of the function.

- **Properties of MGF**

a) The k-th moments are computed as

$$E(t^k) = \frac{\partial k}{\partial s} M(s) \,|_{s=0}$$

b) According to the definition of the pdf as the summation of probabilities, the value of MGF at s=0 equals to unity.

$$M(0) = \int_{-\infty}^{\infty} f(t)dt = 1$$

MGF for Exponential Distributions

The exponential probability density function is given as,

$$f(t) = \lambda e^{-\lambda t}, \qquad t \geq 0$$

The MGF is computed as

$$M(s) = \int_{-\infty}^{\infty} \lambda e^{(s-\lambda)t}\, dt = \int_{-0}^{\infty} \lambda e^{(s-\lambda)t}\, dt$$

$$M(s) = \frac{\lambda}{\lambda - s}$$

The moments are,

$$E(t) = \frac{1}{\lambda^2} \qquad \text{and} \qquad E(t^2) = \frac{2}{\lambda^2}$$

- **Transfer Functions**

The concept of transfer functions from control theory is applied in this analytical method. The procedure is that after obtaining the reachability graph, we transform it to a State-Machine Petri net (SMPN) with single input-single output transitions. We define a transfer function for each transition in the transformed SMPN as the product of MGF and the branch probability of firing P(t) of a transition. Thus a transfer function W(s) can be written as

$$W(s) = P(t)\, M(s)$$

Transfer function depends upon the marking and distribution of concurrent transitions. If the state i leads only to state j, by firing t_j and no other concurrent transition exists at that state, then branch probability is 1. In case when two transitions t_1 and t_2 with exponential distributions l_1 and l_2 are enabled concurrently at a marking, then the transfer functions are

$$W_1 = \frac{\lambda_1}{\lambda_2 + \lambda_1 - s}$$

$$W_2 = \frac{\lambda_2}{\lambda_2 + \lambda_1 - s}$$

2.4 Procedure for System Performance Evaluation

The moment generating function based Petri net performance evaluation methodology for ASPN consists of five stages: ASPN modeling, generation of reachability graph, generation of state machine Petri net, derivation of the transfer functions, and evaluation of performance measures.

a) *ASPN Modeling*

Using Petri net design methodologies such as bottom-up [19], top-down [20-21], and hybrid approaches [44], we can synthesize an ordinary Petri net model for a system based on its operations and relationship among these operations. After we get such an ordinary Petri net, time requirements for various operations result in an ASPN model where every transition is associated with an appropriate time delay that is either constant or random. The execution policy should be built up into such ASPN models to reflect the operations of practical systems.

b) *Generation of Reachability Graph*

Using conventional approaches [14], we can automatically generate a reachability graph of a Petri net. Such a graph represents all reachable states and their relationship among these states. Firing of a transition often implies a change from a state to another.

c) *Generation of State Machine Petri Net*

A state machine Petri net is generated based on the derived reachability graph and information on firing delays of transitions. In fact this state machine Petri net is an ASPN with a particular structure, i.e., each transition has exactly one input place and one output place. A place in the net can have multiple input and output transitions. The place with more than one output transition is called a choice place. It should be noted that in this net, a transition, which may differ from the original transition, is attached with a time delay variable computed based on that of the original transition in the ASPN and its relationship with other transitions in the net. The MGF of the firing delay of each transition in this state machine Petri net is computed. For a choice place, its branch probability is also calculated. Then the transfer function of each transition is derived, which also depends on different execution policies.

d) *Derivation of Transfer Functions*

For the above state machine Petri net, the transfer functions of interesting indices can be derived based on stepwise reductions It is noted that Mason's formula can be directly used in such a reduction process. A sequence, choice, or loop structure can be found to be equivalent of a transfer function.

e) *Evaluation of Performance Measures*

To obtain the i-th moments, we simply take the i-th ($i \geq 1$) derivative of a transfer function of a performance index. Means and derivations of certain measure can be obtained. The analytical results may be obtained by inverting their transfer functions. The mean time to a deadlock state is found. For the system with a deadlock resources such as passage time, reoccurrence time, and cycle time can also be derived for discrete event dynamic systems [2].

CHAPTER 3

MASON'S RULE

A linear system can be represented as a signal-flow graph in which each node represent a variable. The linear dependence T_{ij} between the independent variable x_i and a dependent variable x_j is given by Mason's *loop rule* [7]:

$$T_{ij} = \frac{\Sigma_k P_{ijk} \Delta_{ijk}}{\Delta}$$

where P_{ijk} = k th path from variable x_i to variable x_j,

Δ = determinant of the graph,

Δ_{ijk} = cofactor of the path P_{ijk},

and the summation is taken over all possible k paths from x_i to x_j. The cofactor Δ_{ijk} is the determinant with the loops touching the path removed. The determinant Δ is

$$\Delta = 1 - \sum_{n=1}^{N} L_n + \sum_{m=1,q=1}^{M,Q} L_m L_q - \sum L_r L_s L_t + \dots ,$$

where L_q equals the value of the qth loop transmittance. In other words,

$\Delta = 1 -$ (sum of all different loop gains)

$+$ (sum of the gain products of all combinations of 2 non-touching loops)

$-$ (sum of the gain products of all combinations of 3 non-touching loops)

$+ \dots$.

Two loops are non-touching if they do not have any common nodes.

Consider the following system shown in Figure 3.1 It can be difficult to reduce by block diagram techniques [7]. The forward paths are

$P_1 = G_1G_2G_3G_4G_5G_6$

$P_2 = G_1G_2G_7G_6$

$P_3 = G_1G_2G_3G_4G_8$

The feedback loops are

$L_1 = - G_1G_2G_7G_6H_3,$

$L_2 = - G_1G_2G_3G_4G_8H_3.$

$L_3 = - G_1G_2G_3G_4G_5G_6H_3,$

$L_4 = - G_7H_2G_2,$

$L_5 = - G_2G_3G_4G_5G_2,$

$L_6 = - G_4H_4,$

$L_7 = - G_8H_1,$

$L_8 = - G_5G_6H_1,$

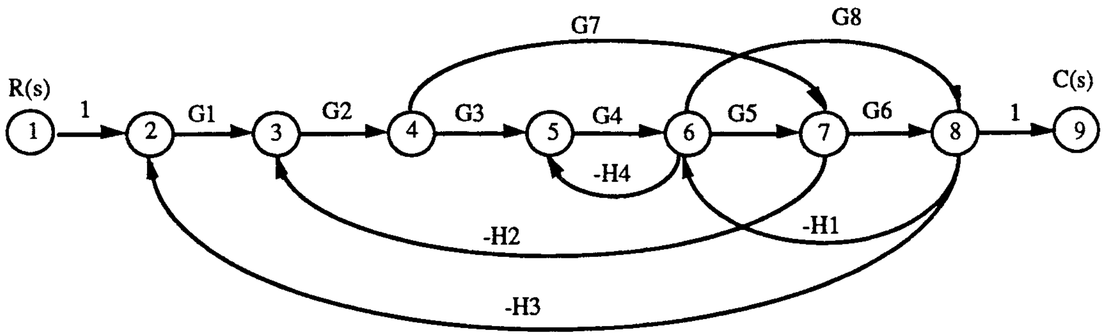


Figure 3.1. Multiple-loop system

Loop L_5 does not touch loop L_4 and loop L_7 ; loop L_3 dose not touch loop L_4; and all other loops are touched with each other. Therefore the determinant is

$$\Delta = 1 - (L_1 + L_2 + L_3 + L_4 + L_5 + L_6 + L_7 + L_8)+(L_6L_1 + L_6L_4 + L_7L_4)$$

The cofactors are

$\Delta_1 = \Delta_3 = 1$ and $\Delta_2 = 1 - L_6 = 1 + G_4H_4$.

Finally, the transfer function is then

$$T = \frac{C(s)}{R(s)} = \frac{P_1 + P_2\Delta_2 + P_3}{\Delta}$$

From this example, one can conclude usefulness of the Mason's rule. On the other hand, one may recognize the complexity of paths and loops search and their non-touching loop check when the system becomes complicated. This partially motivates the computer manipulation of the Mason's rule.

CHAPTER 4

DATA STRUCTURES AND ALGORITHMS

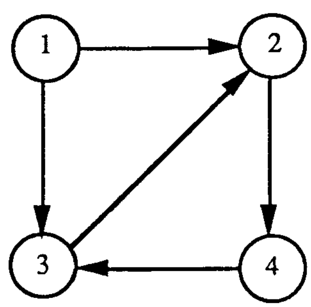
This chapter discusses two representations of a directed graph and two travel algorithms, ***Depth-First Searching*** **(DFS)** and ***Breadth-First Searching*** **(BSF)**[9]. The combination of DFS and BSF in the directed graph are used to search all the loops and forward paths. Then algorithms for check of k-non-touching loops and the loops touching a forward path are derived.

4.1 Representation for Directed Graphs

A directed graph G consists of a set of vertices V and a set of arcs E. The vertices are also called nodes; the arcs could be called directed edges. One common representation for a digraph G=(V, E) is the adjacency matrix (**Figure 4.1**), where V={1, 2, ... n} and E={1, 2, ... e}. Its storage space is $\Omega(n^2)$. Its search time is $O(n^2)$ [9].

$$
M = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}
$$

(a) Representation of adjacency matrix



(b) A directed graph

Figure 4.1 A directed graph

Another common representation for a directed graph G = (E, V) is called the adjacency list (**Figure 4.2**). Its storage space is $\Omega(n+e)$, where e is the number of edges. Its search time is O(n+e) [9]. This representation will be used in our implementation.

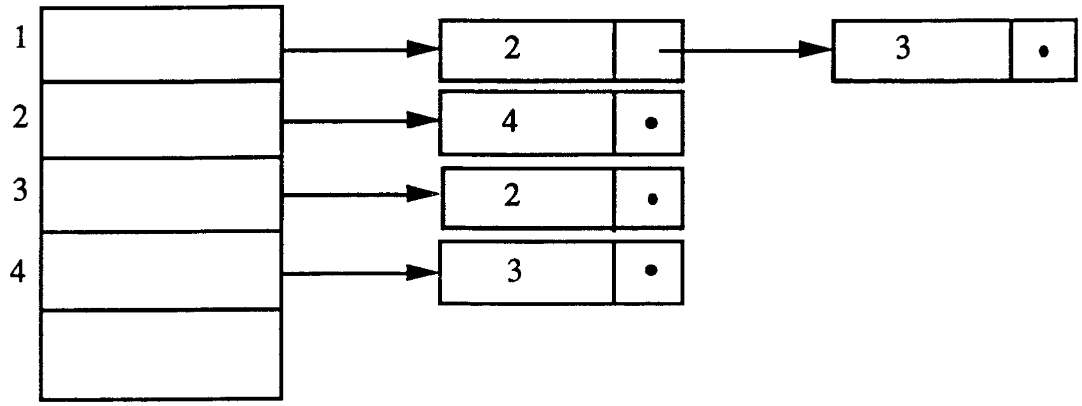


Figure 4.2 Adjacent list for a directed graph

4.2 DFS and BFS Algorithms

4.2.1 Depth-First Searching (DFS)

Breadth-First Searching is to visit all nodes of a graph. Suppose we have a directed graph G in which all nodes are initially marked unvisited. Depth-First Searching works by selecting one node v of G as a start node; v is marked visited. Then each unvisited node adjacent to v is searched in turn, using depth-first search recursively. Once all nodes that can be reached from v have been visited, the search of v is complete. If some nodes remain unvisited, we select an unvisited node as a new start node. We repeat this process until all nodes of G have been visited. This algorithm has the complexity O(e) [9].

4.2.2 Breadth-First Searching (BFS)

In order to visit all node of a graph, Breadth-First Searching visits all nodes that are distance 1 form source at first, then visit all nodes that are distance 2 form source, and so on. We will use queue Q to put source into Q while it is not empty. The algorithm has the same complexity as DFS.

4.3 Data Structures and Graph Declarations

We define a link list structure to represent a given directed graph during the implementation of the Mason's rule as follows:

Structure 1: store net node information for a given net graph

```
struct list_node
    {
      struct list_node *next;
      int   id;
      double  weighted
      int   node_type;
      int   input_arc_number;
      int   in_p[MAX_INPUT_PLACE_NUMBER];
      int   out_p[MAX_OUTPUT_PLACE_NUMBER];
      int   visited_flag;
    }
struct list_node *place[MAX_NODE_NUMBER];
```

where

id: the identification number of the place;

weighted: a weight of directed graph, it refers to the transfer function in Petri net application node_type: the type of place, it declares a place of SOURCE(input), DESTINATION(output), MULTI_INPUT, or SINGLE_INPUT;

input: the number of input arcs of the place;

in_p[ ]: the id of input place;

out_p[ ]: the id of output place;

visited_flag: a check flag(initially 0) for searching forward paths and loops.

Structure 2: store the searching queue information during searching of loops and forward paths.

```
struct searching_queue
    {
     int   queue_no;
     int   p_queue_no;
     int   n_queue_no;
     int   place[MAX_PLACE_NUMBER];
    }
struct searching_queue queue[MAX_QUEUE_NUMBER];
```

where

queue_no: the number of current searching queue;

p_queue_no: the number of previous queue;

n_queue_no: the net queue numbers expanded from current queue.

place[ ]: the place numbers of searched places.

Declaration 1: A **SOURCE** denotes an original input place.

Declaration 2: A **DESTINATION** denotes an output place.

Declaration 3: A **SINGLE_INPUT** denotes a place that has only an input arc.

Declaration 4: A **MULTI_INPUT** denotes a place that has multiple-input arcs.

4.4 Loop Searching

A loop or cycle in a direct graph is a path of non-zero length whose endpoint coincides with its source. A loop is a sequence of vertices $v_0, v_1, \ldots, v_n$, n>1 such that $v_i = v_j$, $1 \le i < j \le n$, implies that i=1 and j=n. Given a graph G=(V, E), we wish to determine whether there are loops. The DEF algorithm can be used to solve this problem. If a back arc is encountered during a depth-first search of G, then clearly the graph has a loop. Conversely, if directed graph has a loop, then a back arc will always be encountered in any depth-first search of the graph. The combination of DFS and BFS algorithms is then applied to find all the loops.

Algorithm 1 (Loop Searching):

Step 1: Invoke the initialized net subroutine:

place[i]→out_p[ ]=the id whose node i has an output arc into a node;

place[i]→int_p[ ]=the id whose node i has an input arc from a node;

Determine place[i]→node_type;

Determine place[i]→input;

place[i]→visited_flag=0;

Step 2: If there no MULTI_INPUT place exists, then there doesn't exist loop(s) and the searching is done;

Else store all the MULTI_INPUT places in loop[]. The number of MULTI_INPUT place is S, and i=1;

Step 3: If i>S, then the searching is done;

Step 4: Start from the MULTI_INPUT place loop[i], Set current queue to 1(c_queue=1); Put the number of loop[i] in the place[] array of queue 1, n_queue_no=1, and visited_flag =1;

Step 5: If n_queue_no of queue 1 is equal to 0, then all adjacent out place of loop[i] has already been searched, i=i+1, goto Step 3;

Step 6: Search adjacent out place of loop[i]:

If there is only one adjacent out places, store this place number in the place[] of c_queue. The visited_flag of this adjacent output place is incremented by 1;

If there are N (N>1) adjacent out place, we expand N queues. Put the number of each adjacent out place in the place[] array of each expanded queue. For every expanded queue, p_queue_no=c_queue, n_queue_no=1, and visited_flag is incremented by 1.

The n_queue_no of c_queue is set N, then c_queue=c_queue+N.

Step 7: Check if the adjacent out place of current queue (c_queue) has been searched or not (visited_flag > 1 ?);

If yes, then check if this adjacent out place is the same as the starting MULTI_INPUT place or not?

If yes, then we have a loop and store these numbers of places of this loop in array loops[][]. For current queue, n_queueno is decreased by 1, goto Step 10;

Else this adjacent out place has been searched already. Therefore the n_queue_no of this current queue is decreased by 1. Goto Step 10;

Else (visited_flag < 1) n_queue_no of c_queue is decreased by 1;

Step 8: Check if the adjacent output place of current queue is a DESTINATION place or not?

If yes, then the n_queue_no of current queue is decreased by 1, goto Step 10;

Step 9: Check if the adjacent output place of current queue is an already searched MULTI_INPUT place or not?

If not, goto Step 5;

Else, the n_queue_no of current queue is decreased by 1;

Step 10: Check if the n_queue_no of current queue is zero or not?

If yes, we abandon this current queue. The n_queue_no of p_queue_no of this current queue must be decrement by 1. and visited_flag is also decreased by 1 for every place of this current queue; c_queue is decreased by 1; If c_queue=0 goto Step 5; Else goto Step 10; Else If c_queue=0 goto Step 5; Else goto Step 7.

4.5 Forward Path Searching

A path in a graph is a sequence of vertices v_0, v_1, ... ,v_n, with n ≥ 1, such that ∃ an arc (v_i, v_{i+1}) for i ≤ n-1, and v_i=v_j implies that i=j, 0≤i, j≤n. The vertex v_0 is the source of the path and v_k its endpoint; the n+1 is the length of the path.

The searching for forward paths starts from a given SOURCE_INPUT place to a given DESTINATION place. First of all, we visit all the adjacent places, and check to see if they are a DESTINATION place. We get a forward path if so. Otherwise, we check to see if it has already been searched. If it has already been searched, then this path is not a simple path, which implies that this is not a forward path. Therefore we abandon this path and choose another adjacent place and repeat the above procedures. This process will continue until all the adjacent places are processed.

Algorithm 2 (Forward Path Searching):

Step 1: Invoke initialized net subroutine:

initial queue, visited_flag and current queue;

Step 2: Start to search from the SOURCE_INPUT place. Put the number of SOURCE place into the place[] array in which queue number is 1, t he p_queue_no is set to 0 and n_queue_no is set to 1, and the visited_flag of this SOURECE_INPUT place is increased by 1, and c_queue=1;

Step 3: If n_queue_no of c_queue is equal to 0, then the searching is done.

Step 4: Find all adjacent place of c_queue:

If there exists only one adjacent out place, then put this place number into the place[] array of c_queue and n_queue_no of c_queue is set to 1. The visited_flag of this adjacent output place is increased by 1;

If there are N adjacent output place numbers stored in every place[] array of each new queue. Every n_queue_no of these N queues is set to c_queue+N; n_queue_no is set to 1; ans p_queue_no is set to c_queue. Further, every visited_flag of these N (N>1) adjacent output places is increased by 1. The n_queue_no of current queue is set to N, and then c_queue=c_queue+N;

Step 5: Check if the place in c_queue or the N adjacent output place are a DESTINATION place or not?

If yes, a forward path is obtained. Store this forward path in f_paths[][]. The n_queue_no of current queue is decreased by 1. Goto Step 7.

Step 6: Check if the visited_flag of these adjacent output place are greater than 1 or not?

If not, goto Step 3;

Else, this adjacent output place has already been searched. Therefore the n_queue_no of current queue is decreased by 1.

Step 7: Check if the n_queue_no of current queue is equal to zero or not?

If not, goto Step 6;

Else, if c_queue is equal to 1, the searching is done.

Else we abandon this c_queue, and the n_queue_no of p_queue_no of current queue is decreased by 1. Every visited_flag of this place of current queue is decreased by 1, c_queue=c_queue-1. Then goto Step 7.

4.6 Check Non-touching Loops

Assume we have got n loops by **Algorithm 1**, and they are stored in array loops[][], A method is, at first, to determine a vector set of combinations of comparing for the non-touching loops. The combinations of k nontouching loops are given by

$$\frac{n!}{(n-k)!\ k!} \qquad (2 \leq k \leq n)$$

Then we can check whether each combination of k loops are touching. This algorithm is derived as follows.

Algorithm 3 (Check Non-touching Loops):

/* Assume c is the index for loop i. n is the number of total loops. k is the number of loops to be compared for the non-touching case. k>=2. */

```
C[0] = -1;
```

```
for (i=1; i<=K; i++)   C[i] = i;

j = 1;

while (j!=0 )

      {

        for (i=1; i<=k i++) output_comb_index(C[i]);

        check_nontouching_loops(k);

        j = K;

        while (C[i] = n-k+j) j--;

        C[j]++;

        for (i=j+1; i<=K; i++) C[i] = C[i-1] +1;

      }
```

4.7 Check the Loops Touching Each Forward Path

In order to get Δ_k in Mason' formula, we need to find those loops which touch the ith forward path. If a loop L_i ($0<i\leq m$, m is the number of loops) touches the kth forward path (P_k), then we have $L_i = 0$ in Δ to obtain Δ_k. The algorithm for checking loops of touching path P_k is below:

Algorithm 4 (Check the Loops Touching Each Forward Path):

```
for ( pi=1; pi<=total_path+number; pi++ )

       {    j=0;

       for ( li=1; li<=total_loop_number; li++ )

            {   pj=0;

                 while (  paths[li][++lj]  !=0 )

                 {   lj=0;
```

```
                    while ( loops[li][++li] !=0 )
                    if ( paths[pi][pj]==loops[li][lj] )
                      {  loop_of_touching_path[pi][++j]=li;
                           goto  A;
                      }
                }
A:          ;
           }
     }
```

4.8 Complexity Analysis of the Algorithms and the Program

The complexity of these four algorithms can be analyzed as follows. For Algorithm 1, Step 1 has the complexity of O(MN) where M is the maximum number of the loops, and N is the maximum number of nodes, both predefined in the program. Step 2 takes O(n) where n is the number of nodes in the net; Step 3 takes O(1); Step 4 takes O(MN). Step 5 consists of assignment statements (i.e., O(1)). Since in the worst case, the '*goto step 5*' statement in Step 10 will be executed up to S time, we consider the complexity of Step 5 is O(S*1) (S is the maximum number which current queue is equal to 0); Step 6 takes $O(n_a)$ where n_a is the total number of output arcs in a node, which is less than the node number, i.e., n; Step 7 takes $O(M_{qc}*j*r)$ (where, j is the count number for searching a valid loop queue, r is the number of adjacent output places in a current queue, M_{qc} is the maximum number which the current queue is not equal to 0, and in the worst case the "*goto step 7*" statement in the step 10 will be executed up to M_{qc} time). Note that j<m, the number of loops, and r<n. Step 8 has complexity O(1); Step 9 takes $O(n_b)$ where n_b (<n) is the total number of the MULTI_INPUT place in the net; and Step 10 takes $O(M_{qc}*r)$. Therefore, the overall complexity of Algorithm 1 is $O(MAX(MN, M_{qc} *j*r))$.

Similar analysis can be conducted for the other three algorithms. Algorithm 2 has the same complexity as Algorithm 1, i.e., O(MAX(MN, M_{qn}*j*r)), O(a^n) for algorithm 3 (when it is invoked until k=n time), where a is the cost of *check_nontouching_loops(k)*(It is O(i^2*k)), and O(m^2n^2) for Algorithm 4.

Based above analysis, Algorithm 3 has the exponential growth with the number of loops. Therefore, the program will have a larger cost on the time when solving Mason's rule. When n grows large, the computer resources will be used up and thus this is not applicable for a very large system if we do not select an optimal algorithm to implement the checking non-touching loops.

Since Algorithm 3 has a larger amount of time cost (i.e., exponential growth), we present a procedure (Algorithm 5) for the *chech_nontouching_loops(k)* function to reduce the cost of the program in Algorithm 3 as follows:

Algorithm 5 (Reduce time cost when checking non-touching loops):

Step 1: Obtain the combinations for k=2;

Step 2: If all the combinations of k=2 are the non-touching loops, then all the combinations of 3≤k≤n are also non-touching loops (so that it is not necessary to continue to check the combinations of 3≤k≤n), and searching is done;

Step 3: If all the combinations of k=2 are touching loops, there are not any non-touching loops, and searching is done;

Step 4: If there are C nontouching combinations (C≥3) for all the combinations of k=2, we continue to check that if there are any combinations of the non-touching loops for k=3, ..., (n-1), (i.e., invoke *chech_nontouching_loops(k)* function for k=3, ..., (n-1).

Similarly, we can derive the wost-cast complexity of algorithm 5 has the exponential growth ($O(b^n)$). But, if the program invokes algorithm 5 instead of algorithm 3, its cost of checking non-touching loops will be greatly reduced. Thus, this result implies our scheme can be available for a more complicated net for the solution of Mason's rule.

CHAPTER 5

DESIGN SPECIFICATIONS OF CIMR

5.1 Development Environment of CIMR

The ***cimr*** is developed in UNIX using C language. UNIX provides us a very good programming environment. It can run on a range of computers from microprocessors to the largest mainframes. It is a good operating system, especially for programmers. C is a modern programming language and provides a fairly complete set of facilities for dealing with a wide variety of applications. C has all the useful data types, operators, control structures and a standard run-time library that includes useful functions for input/output, storage allocation, string manipulation, and other purposes. C programs are efficient and are generally quite portable across different computer hardware. The design of C also makes it natural to use top-down planning, structured and modular programming.

A utility program '***cimr***' runs under UNIX shell, *cimr* can process an input file while describes a state machine Petri net written by State Machine Petri Net Language (SMPNL), and creates an output file to describe transfer functions. The *cimr* also inspects the grammars of input file and create an error information list file.

5.2 Modules of CIMR

The functions of *cimr* are described as follows:

main(argc, argv):	the main model of the program;
read_input_file():	process input file of SMPNL;
initial():	build the adjacency matrix and adjacency list queue for a net;
print_matrix():	output the matrix elements;

build_adj_list():	build adjacency list queue;
initial_place_info_field():	initialize the information fields of *structure 1* in section 4.3;
initial_qrow():	initialize the information fields of *structure 2* in section 4.3;
initial_visited_flag():	initialize the check flag unit for searching forward paths and loops;
find_self_loops():	search all loops for a net;
check_same_loops():	check if there are any same loops;
find_forward_paths():	search all forward paths for a net;
combination(n_loop, k):	determine k loops (k=2, ... n) combinations to check that these loops which are touching or not;
check_nontouching_loops():	check non-touching loops for a net;
output_nontouching_loops():	output all non-touching loops to an output file and CRT;
loops_of_touching_path():	find all loops which touch any paths in a net;
appli_mason_formula():	solve Mason's formula;
calculate_delta():	solve the determinant of the graph of a net;
calculate_delta_i():	solve the cofactor of the path of a net.

CHAPTER 6

APPLICATION EXAMPLES OF CIMR

6.1 An Example of Performance Analysis of SPN

An application example is used to show application of Mason's rule to performance analysis of flexible manufacturing system (**FMS**). FMS is a system with automated machines, interconnected by automated material handling. The design, operation and control of these systems have to take into account numerous interactions occurring between concurrent and nondeterministic activities. The system is categorized as a discrete-event dynamic system. In these systems, the important performance parameters are machine utilizations, production rates, average queue size and waiting times.

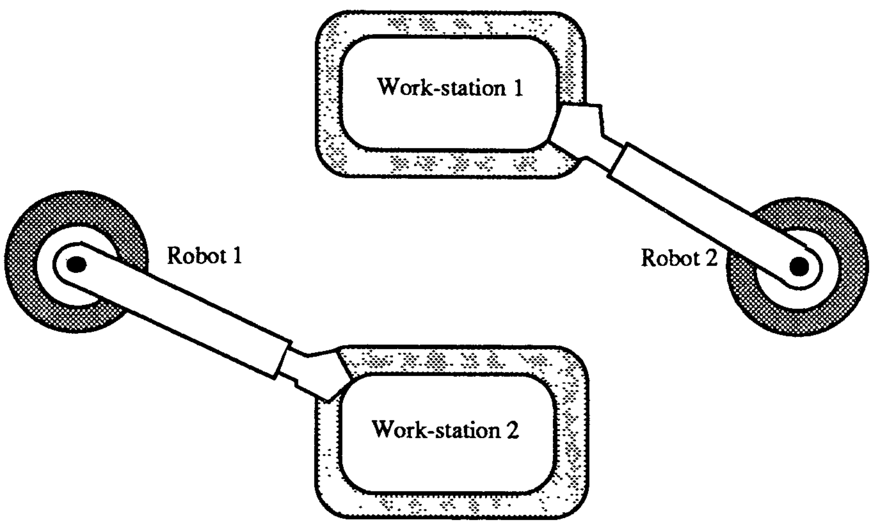
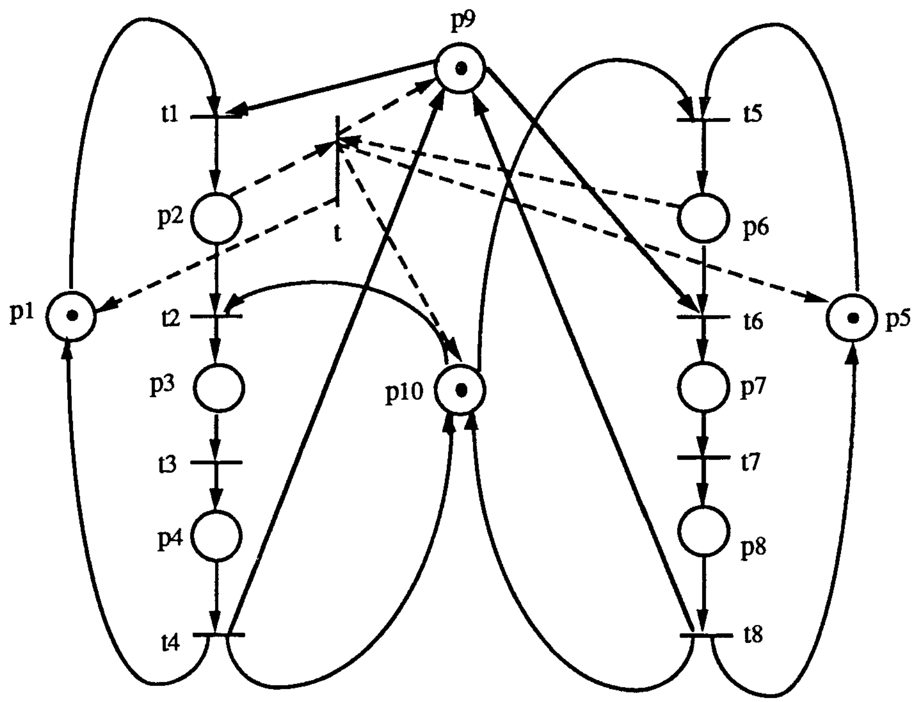


Figure 6.1 A flexible manufacturing system

Figure 6.1 [3] is consider as a FMS, which has two work-states (WS_1 and WS_2) and two robots (R_1 and R_2). The workstations do identical job, which is, assembly of parts with the help of both robots. We assume that each workstation acquires its right robot first and then the robot on its left, to assemble the parts.

Now, let's derive transfer functions by using CIMR method for performance evaluation of this system.

Figure 6.2 is an ASPN model for the system and its reachability graph. Firing of transition t_1 and t_5 leads to a system deadlock. The dash arcs and transition t is for the resolution of the deadlock. The deadlock rate can be found with the moment generating function based method.



t: Deadlock resolution

$m_0\,(1, 0, 0, 0, 1, 0, 0, 0, 1, 1)^T$

Figure 6.2 (a) An ASPN model

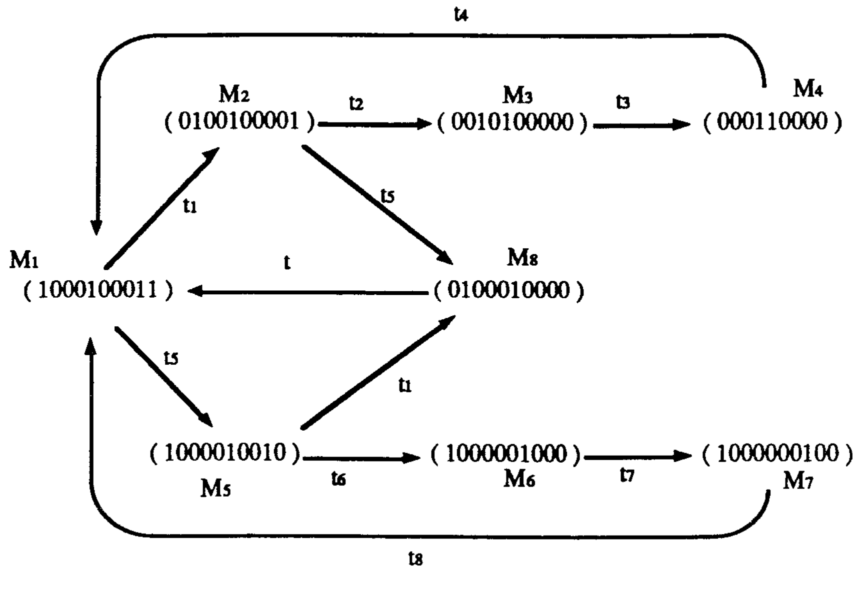


Figure 6.2 (b) The reachability graph of the ASPN in (a)

Figure 6.2 A Petri net model and its reachability graph

Time delays of transitions t_1, t_2, t_3, t_5, t_6 and t_7 are exponentially distributed random variables, and the firing times of transitions t_4, t_8, and t are constants: c_1, c_2 and c_3. The state machine Petri net (**Figure 6.3**) of the system is given from its reachability graph and assume the transfer functions of the each transition are derived as W_i $(i = 1...11)$ [3].

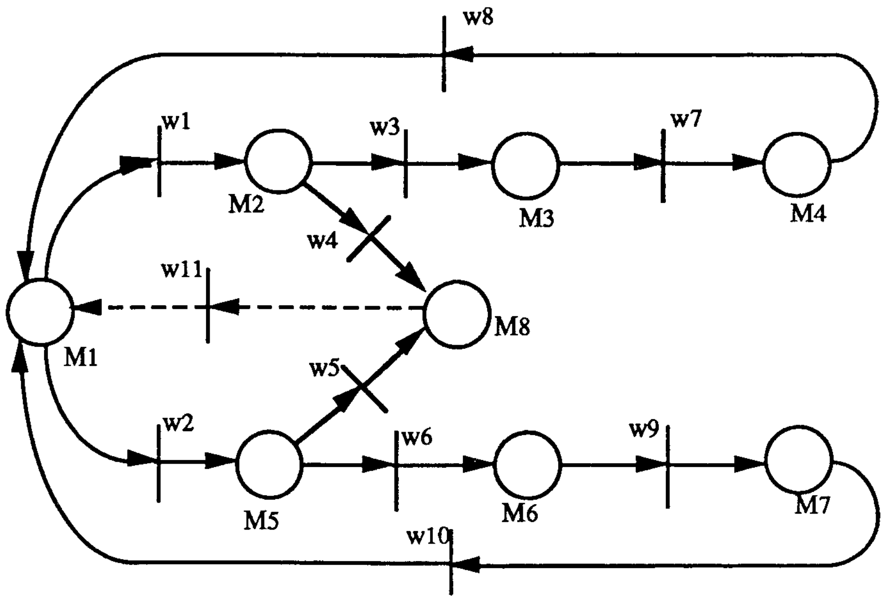


Figure 6.3 The state machine Petri net

In order to derive the deadlock rate, let the SOURCE place be M_0 and the DESTINATION place M_0' as shown in **Figure 6.4**. In order to apply the Mason's rule, we map a state machine Petri net into a directed graph with weight of W_i (**Figure 6.5**).

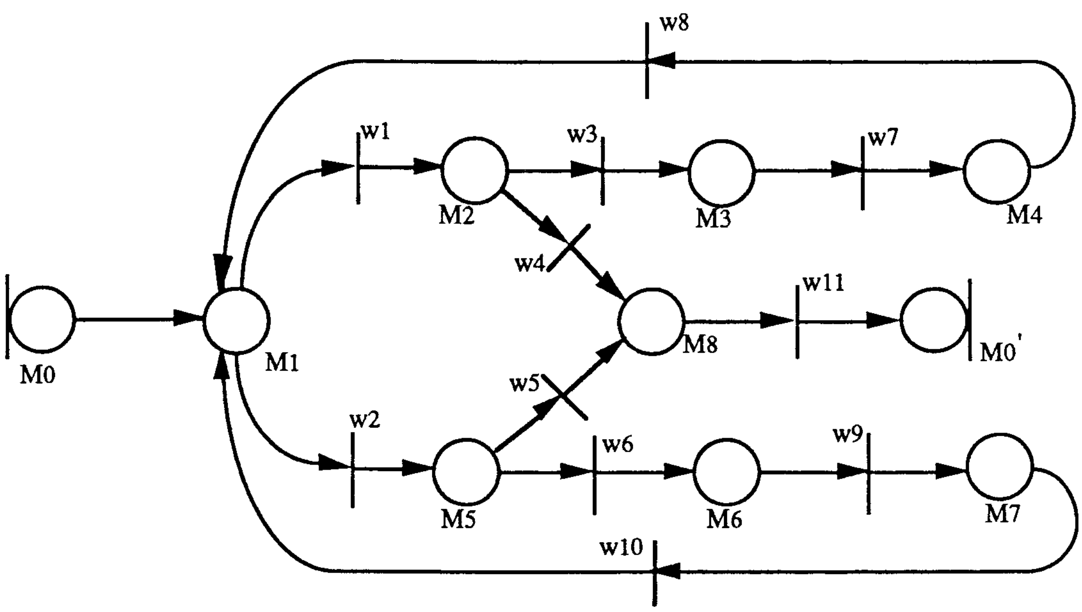


Figure 6.4 A net for deadlock rate by Source-Sink Solution

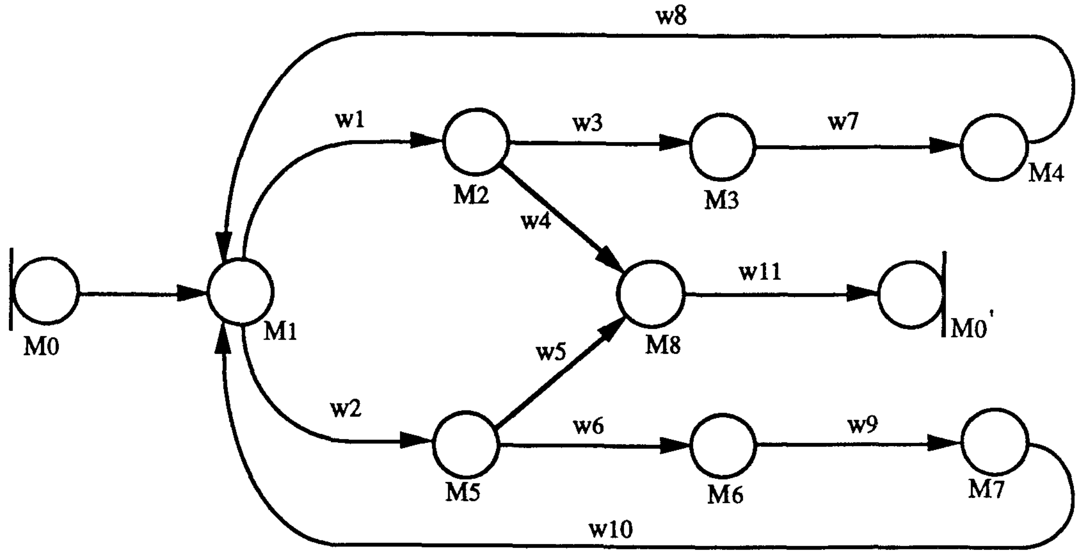


Figure 6.5 A mapped directed graph for the deadlock rate

The structures and information fields for the graph in **Figure 6.5** are given in **Table** 1.

Table 6.1. Node structures and information fields for **Figure 6.5**

	place[0]	place[1]	place[2]	place[3]	place[4]	place[5]	place[6]	place[7]	place[8]	place[0']
id	0	1	2	3	4	5	6	7	8	9
node_type	SOURCE	MULTI	SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	SINGLE	MULTI	SINGLE
input	0	2	1	1	1	1	1	1	2	1
in_p[]		in_p[0]=4 in_p[1]=7	in_p[0]=1	in_p[0]=2	in_p[0]=3	in_p[0]=1	in_p[0]=5	in_p[0]=6	in_p[0]=2 in_p[1]=5	in_p[0]=8
out_p[]	out_p[0]=1	out_p[0]=2 out_p[1]= 5	out_p[0]= 3 out_p[1]= 8	out_p[0]= 4	out_p[0]= 1	out_p[0]= 6 out_p[1]= 8	out_p[0]= 7	out_p[0]= 1	out_p[0]=9	
visited_f	0	0	0	0	0	0	0	0	0	0

The transfer function form input M_0 to output M_0' for the deadlock rate is derived by our computer algorithms for Mason's rule.

```
/*******************************************************************/
/* filename: ex1.i                                                 */
/*                                                                 */
```

```
/* This is an input file by using SMPN language. It describes    */
/* a State Machine Petri Net for Mason's rule application. After */
/* running command 'cimr filename.i[return]', you can get an     */
/* output file(filename.o), which records some useful information*/
/* of transfer function for performance analysis of a net.       */
/* Any text errors will reported to an output file(filename.e)   */
/* after compiling.                                              */
/*                                                               */
/***************************************************************/

net=9;          /* the total place numbers of the net place */
input=1;        /* determine the id of an input place */
output=9;       /* determine the id of an output place */
/* Determine the parameters of the State Machine Petri Net    */
/* node=(ptr1,ptr2,ptr3), where, 'ptr1' refers to id of input */
/* place, 'ptr2' refers to id of output place, and 'ptr3'     */
/* refers to the index of transfer function variables.         */
node=(1,2,1);
node=(1,5,2);
node=(2,3,3);
node=(2,8,4);
node=(3,4,7);
node=(4,9,8);
node=(5,6,6);
node=(5,8,5);
node=(6,7,9);
node=(7,1,10);
node=(8,1,11);
```

Figure 6.6 An input file of state machine Petri net

A description language of State Machine Petri Net is written as an input file of net (**Figure 6.6**). Use running command '**cimr**' under UNIX shell environment blow:

$cimr filename

After running the above command we can obtain an output file (filename.o) which presents all solutions related to evaluation of the Mason's rule for a net (**Figure** 6.7).

```
Output file: ex1.o

The adjacncy matrix(9x9):
-------------------------

0 1 0 0 2 0 0 0 0

0 0 3 0 0 0 0 4 0

0 0 0 7 0 0 0 0 0

8 0 0 0 0 0 0 0 0

0 0 0 0 0 6 0 5 0

0 0 0 0 0 0 9 0 0

10 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 11

0 0 0 0 0 0 0 0 0

* The elements which are not equal to zero in the matrix refer to the
index of the transfer function (Wi).

Adjacency List Queue:
--------------------

(1)--> w1(2)--> w2(5)

(2)--> w3(3)--> w4(8)

(3)--> w7(4)

(4)--> w8(1)

(5)--> w6(6)--> w5(8)

(6)--> w9(7)

(7)--> w10(1)

(8)--> w11(9)

(9)

* Wi refers to the transfer function.

Loop(s):
-------

L1:  (1)->(5)->(6)->(7)->(1)
     (W2 W6 W9 W10 )

L2:  (1)->(2)->(3)->(4)->(1)
     (W1 W3 W7 W8 )
```

```
The forward path(s) from node 1 to node 9:
------------------------------------------

P1: (1)->(5)->(8)->(9)

    (W2 W5 W11 )

P2: (1)->(2)->(8)->(9)

    (W1 W4 W11 )


Combinations for checking nontouching loops:
-------------------------------------------

k=2:

Non-touching_loops:
------------------


Loops touching forward path:
---------------------------

Loops of touching path P1:  L1 L2

Loops of touching path P2:  L1 L2

SOLUTION OF MASON'S RULE FOR THE INPUT NET:
==========================================

DELTA  = 1-(L1+L2)

DELTA1 = 1-(0+0)

DELTA2 = 1-(0+0)

MASON'S VALUE(out/in): T(s) = (P1 * DELTA1 + P2 * DELTA2 ) / DELTA

End of Execution!
```

Figure 6.7 A output file of Mason's rule computer implementation

---

Therefore, we get:

The first forward path :	$P_1 = w_1\ w_4\ w_{11}$,
The second forward path :	$P_2 = w_2\ w_5\ w_{11}$,
There are two self-loops:	$L_1 = w_1\ w_3\ w_7\, w_8$,
	$L_2 = w_2\ w_6\ w_9\, w_{10}$.
Non-touching loops:	None

Where, loop L_1 touches loop L_2. Therefore, the determinant is

$$\Delta = 1 - (L_1 + L_2)$$

The cofactor of the determinate alone path (P_1) is evaluated by removing the loops that touch path (P_1) from Δ (or P_1 touches all loops). Therefore, we have

$P_1=0,$

$P_2=0,$

$\Delta_1=1- 0 =1,$

Similarly, since P_2 touches all loops, the cofactor for path 2 is

$P_1=0,$

$P_2=0,$

$\Delta_2=1- 0 =1$

Total gain ($M_0 \rightarrow M_0'$): $T = (1/\Delta) \sum_{k=1} P_k \Delta_k$

Thus, the transfer function from source place $M_0(s)$ to sink place $M_0'(s)$ is:

$$\begin{aligned} W_E(s) &= \frac{M_0'(s)}{M_0(s)} \\ &= \frac{P_1 \Delta_1 + P_2 \Delta_2}{\Delta} \\ &= \frac{P_1 + P_2}{1 - L_1 - L_2} \\ &= \frac{(w_1 w_4 + w_2 w_5)\, w_{11}}{1 - w_1 w_3 w_7 w_8 - w_2 w_6 w_9 w_{10}} \end{aligned}$$

Its moment generating function is

$$M_E(s) = \frac{W_E(s)}{W_E(0)}$$

Then the mean recurrence time from initial state to deadlock state:

$$TR = \frac{\partial M_E(s)}{\partial s} \Big|_{s=0}$$

The mean sojourn time is

$$TS = \frac{\partial \bar{M}_E(s)}{\partial s}\Big|_{s=0}$$

where $\bar{M}_E$ is the moment generating function by replacing the transfer functions $w_i(s)$ $(i \neq 11)$ with one. Therefore, the deadlock rate is

$$R_d = \frac{TS}{TR}$$

Supposing the deadlock resolution time is a parameter, we can lead to a deadlock rate of the closed-form [3].

6.2 An Example of A Linear System

The example (**Figure 3.1**) in Chapter 2 can be solved by *cimr* for deriving transfer function as shown in **Figure 6.8** and **Figure 6.9**, where

W_1=1, W_2=G_1, W_3=G_2,

W_4=G_3, W_5=G_4, W_6=G_5,

W_7=G_6, W_8=G_7, W_9=G_8,

W_{10}=-H_4, W_{11}=-H_1, W_{12}=-H_2,

W_{13}=-H_3, W_{14}= 1

```
/*******************************************************************/
/*  filename: ex2.i                                                */
/*                                                                 */
/*  This is an input file by using SMPN language. It describes     */
/*  a State Machine Petri Net for Mason's rule application. After  */
/*  running command 'cimr filename.i[return]', you can get an      */
/*  output file(filename.o), which records some useful information */
/*  of transfer function for performance analysis of a net.        */
/*  Any text errors will reported to an output file(filename.e)    */
```

```
/*  after compiling.                                              */
/*                                                                */
/******************************************************************/

net=9;            /* the total place numbers of the net */
input=1;          /* determine a source id of a place */
output=9;         /* determine a destination id of a place */
/* Determine the parameters of the State Machine Petri Net      */
/* node=(ptr1,ptr2,ptr3), where, 'ptr1' refers to id of input   */
/* place, 'ptr2' refers to id of output place, and 'ptr3'       */
/* refers to the index of transfer function variables.          */
node=(1,2,1);
node=(2,3,2);
node=(3,4,3);
node=(4,5,4);
node=(4,7,8);
node=(5,6,5);
node=(6,5,10);
node=(6,7,6);
node=(6,8,9);
node=(7,3,12);
node=(7,8,7);
node=(8,2,13);
node=(8,6,11);
node=(8,9,14);
```

figure 6.8 Description file of the net shown in figure 1

```
Output file: ex2.o
The adjacncy matrix(9x9):
--------------------------
0 1 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0
0 0 0 3 0 0 0 0 0
0 0 0 0 4 0 8 0 0
0 0 0 0 0 5 0 0 0
```

```
0 0 0 0 10 0 6 9 0
0 0 12 0 0 0 0 7 0
0 13 0 0 0 11 0 0 14
0 0 0 0 0 0 0 0 0
```

* The elements which are not equal to zero in the matrix refer to the index of the transfer function(Wi).

```
Adjacency List Queue:
---------------------

(1)--> w1(2)
(2)--> w2(3)
(3)--> w3(4)
(4)--> w4(5)--> w8(7)
(5)--> w5(6)
(6)--> w10(5)--> w6(7)--> w9(8)
(7)--> w12(3)--> w7(8)
(8)--> w13(2)--> w11(6)--> w14(9)
(9)
```

* Wi refers to the transfer function.

```
Loop(s):
-------

L1: (2)->(3)->(4)->(7)->(8)->(2)
    (W2 W3 W8 W7 W13 )
L2: (2)->(3)->(4)->(5)->(6)->(8)->(2)
    (W2 W3 W4 W5 W9 W13 )
L3: (2)->(3)->(4)->(5)->(6)->(7)->(8)->(2)
    (W2 W3 W4 W5 W6 W7 W13 )
L4: (3)->(4)->(7)->(3)
    (W3 W8 W12 )
L5: (3)->(4)->(5)->(6)->(7)->(3)
    (W3 W4 W5 W6 W12 )
L6: (5)->(6)->(5)
    (W5 W10 )
L7: (6)->(8)->(6)
    (W9 W11 )
L8: (6)->(7)->(8)->(6)
    (W6 W7 W11)
```

```
The forward path(s) from node 1 to node 14:
-------------------------------------------

P1:  (1)->(2)->(3)->(4)->(7)->(8)->(9)
     (W1 W2 W3 W8 W7 W14 )
P2:  (1)->(2)->(3)->(4)->(5)->(6)->(8)->(9)
     (W1 W2 W3 W4 W5 W9 W14 )
P3:  (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)
     (W1 W2 W3 W4 W5 W6 W7 W14 )


Nontouching_loops:
-----------------

k=2:  { L1 L6 }, {L4 L6}, {L4 L7 }


Loops touching forward path:
---------------------------

Loops of touching path P1:  L1 L2 L3 L4 L5 L7 L8
Loops of touching path P2:  L1 L2 L3 L4 L5 L6 L7 L8
Loops of touching path P3:  L1 L2 L3 L4 L5 L6 L7 L8


SOLUTION OF MASON'S RULE FOR THE INPUT NET:
===========================================

DELTA  = 1-(L1+L2+L3+L4+L5+L6+L7+L8)+(L1*L6+L4*L6+L4*L7)
DELTA1 = 1-(0+0+0+0+0+L6+0+0)+(0*L6+0*L6+0*0)
DELTA2 = 1-(0+0+0+0+0+0+0+0)+(0*0+0*0+0*0)
DELTA3 = 1-(0+0+0+0+0+0+0+0)+(0*0+0*0+0*0)


MASON'S VALUE(out/in): T(s) = (P1 * DELTA1 + P2 * DELTA2 + P3 * DELTA3
) / DELTA


End of Execution!
```

Figure 6.9 Computer solution of Mason's rule for the net shown in **Figure 3.1**

6.3 An Example of A Complicated Net

Figure 6.10 is a net graph that has 25 loops and 5 forward paths. There are 19 places, Input is node 1 and output is node 19. The more loops, the more complicated to derive the transfer functions for a net graph. Therefore it is not easy to do that by hand. We use **cimr** to derive the transfer functions for the net . The results of **cimr** are shown in **Figure 6.11** (Input file of **cimr**) and **Figure 6.12** (Output file of **cimr**).

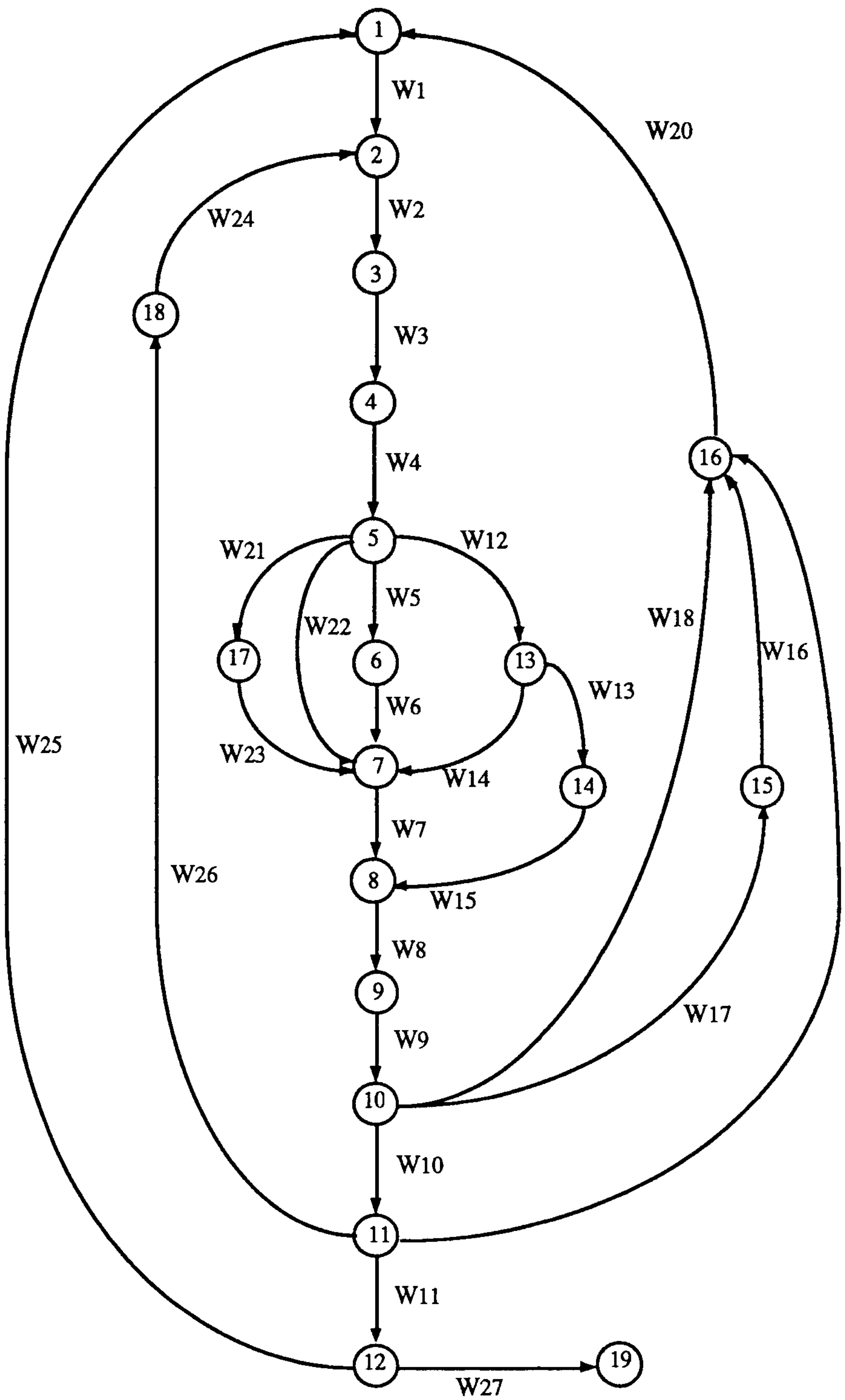


Figure 6.10 A Complicated Net

```
/*******************************************************************/
/*  filename: ex3.i                                                */
/*                                                                 */
/*  This is an input file by using SMPN language. It describes     */
/*  a State Machine Petri Net for Mason's rule application. After  */
/*  running command 'cimr filename.i[return]', you can get an      */
/*  output file(filename.o), which records some useful information */
/*  of transfer function for performance analysis of a net.        */
/*  Any text errors will reported to an output file(filename.e)    */
/*  after compiling.                                               */
/*                                                                 */
/*******************************************************************/

net=19;          /* the total numbers of the net place      */

input=1;         /* to determine the id of an input place  */

output=19;     /* to determine the id of an output place */

/* Determine the parameters of the State Machine Petri Net      */
/* node=(ptr1,ptr2,ptr3), where, 'ptr1' refers to id of input   */
/* place, 'ptr2' refers to id of output place, and 'ptr3'       */
/* refers to the index of transfer function variables.          */

node=(1,2,1);

node=(2,3,2);

node=(3,4,3);

node=(4,5,4);

node=(5,6,5);

node=(6,7,6);

node=(7,8,7);

node=(8,9,8);

node=(9,10,9);

node=(10,11,10);
```

```
node=(11,12,11);

node=(5,13,12);

node=(13,14,13);

node=(13,7,14);

node=(14,8,15);

node=(15,16,16);

node=(10,15,17);

node=(10,16,18);

node=(11,16,19);

node=(16,1,20);

node=(5,17,21);

node=(5,7,22);

node=(17,7,23);

node=(18,2,24);

node=(12,1,25);

node=(11,18,26);

node=(12,19,27);
```

Figure 6.11 Input file of cimr in Figure 6.10

```
Output file: ex3.o

The adjacncy matrix(19x19):
--------------------------

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 5 22 0 0 0 0 0 12 0 0 0 21 0 0
0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 10 0 0 0 17 18 0 0 0
0 0 0 0 0 0 0 0 0 0 0 11 0 0 0 19 0 26 0
25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 27
0 0 0 0 0 0 14 0 0 0 0 0 0 13 0 0 0 0 0
0 0 0 0 0 0 0 15 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16 0 0 0
20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 23 0 0 0 0 0 0 0 0 0 0 0 0
0 24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

* The elements which are not equal to zero in the matrix refer to the index of the transfer function(Wi).

Adjacency List Queue:

```
(1)--> w1(2)
(2)--> w2(3)
(3)--> w3(4)
(4)--> w4(5)
(5)--> w5(6)--> w22(7)--> w12(13)--> w21(17)
(6)--> w6(7)
(7)--> w7(8)
```

```
(8)--> w8(9)

(9)--> w9(10)

(10)--> w10(11)--> w17(15)--> w18(16)

(11)--> w11(12)--> w19(16)--> w26(18)

(12)--> w25(1)--> w27(19)

(13)--> w14(7)--> w13(14)

(14)--> w15(8)

(15)--> w16(16)

(16)--> w20(1)

(17)--> w23(7)

(18)--> w24(2)

(19)

* Wi refers to the transfer function.


Loop(s):
-------

L1: (1)->(2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(16)->(1)

    (W1 W2 W3 W4 W12 W13 W15 W8 W9 W18 W20 )

L2: (1)->(2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(15)->(16)-
>(1)

    (W1 W2 W3 W4 W12 W13 W15 W8 W9 W17 W16 W20 )

L3: (1)->(2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(11)->(16)-
>(1)

    (W1 W2 W3 W4 W12 W13 W15 W8 W9 W10 W19 W20 )

L4: (1)->(2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(11)->(12)-
>(1)

    (W1 W2 W3 W4 W12 W13 W15 W8 W9 W10 W11 W25 )

L5: (1)->(2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(16)->(1)
```

(W1 W2 W3 W4 W22 W7 W8 W9 W18 W20)

L6: (1)->(2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(15)->(16)->(1)

(W1 W2 W3 W4 W22 W7 W8 W9 W17 W16 W20)

L7: (1)->(2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(11)->(16)->(1)

(W1 W2 W3 W4 W22 W7 W8 W9 W10 W19 W20)

L8: (1)->(2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(11)->(12)->(1)

(W1 W2 W3 W4 W22 W7 W8 W9 W10 W11 W25)

L9: (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(16)->(1)

(W1 W2 W3 W4 W5 W6 W7 W8 W9 W18 W20)

L10: (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(15)->(16)->(1)

(W1 W2 W3 W4 W5 W6 W7 W8 W9 W17 W16 W20)

L11: (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(11)->(16)->(1)

(W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W19 W20)

L12: (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(11)->(12)->(1)

(W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W25)

L13: (2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(11)->(18)->(2)

(W2 W3 W4 W12 W13 W15 W8 W9 W10 W26 W24)

L14: (2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(11)->(18)->(2)

(W2 W3 W4 W22 W7 W8 W9 W10 W26 W24)

L15: (2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(11)->(18)->(2)

(W2 W3 W4 W5 W6 W7 W8 W9 W10 W26 W24)

L16: (7)->(8)->(9)->(10)->(16)->(1)->(2)->(3)->(4)->(5)->(17)->(7)

(W7 W8 W9 W18 W20 W1 W2 W3 W4 W21 W23)

L17: (7)->(8)->(9)->(10)->(16)->(1)->(2)->(3)->(4)->(5)->(13)->(7)

(W7 W8 W9 W18 W20 W1 W2 W3 W4 W12 W14)

L18: (7)->(8)->(9)->(10)->(15)->(16)->(1)->(2)->(3)->(4)->(5)->(17)->(7)

(W7 W8 W9 W17 W16 W20 W1 W2 W3 W4 W21 W23)

L19: (7)->(8)->(9)->(10)->(15)->(16)->(1)->(2)->(3)->(4)->(5)->(13)->(7)

(W7 W8 W9 W17 W16 W20 W1 W2 W3 W4 W12 W14)

L20: (7)->(8)->(9)->(10)->(11)->(18)->(2)->(3)->(4)->(5)->(17)->(7)

(W7 W8 W9 W10 W26 W24 W2 W3 W4 W21 W23)

L21: (7)->(8)->(9)->(10)->(11)->(18)->(2)->(3)->(4)->(5)->(13)->(7)

(W7 W8 W9 W10 W26 W24 W2 W3 W4 W12 W14)

L22: (7)->(8)->(9)->(10)->(11)->(16)->(1)->(2)->(3)->(4)->(5)->(17)->(7)

(W7 W8 W9 W10 W19 W20 W1 W2 W3 W4 W21 W23)

L23: (7)->(8)->(9)->(10)->(11)->(16)->(1)->(2)->(3)->(4)->(5)->(13)->(7)

(W7 W8 W9 W10 W19 W20 W1 W2 W3 W4 W12 W14)

L24: (7)->(8)->(9)->(10)->(11)->(12)->(1)->(2)->(3)->(4)->(5)->(17)->(7)

(W7 W8 W9 W10 W11 W25 W1 W2 W3 W4 W21 W23)

L25: (7)->(8)->(9)->(10)->(11)->(12)->(1)->(2)->(3)->(4)->(5)->(13)->(7)

(W7 W8 W9 W10 W11 W25 W1 W2 W3 W4 W12 W14)

The forward path(s) from node 1 to node 19:

P1: (1)->(2)->(3)->(4)->(5)->(17)->(7)->(8)->(9)->(10)->(11)->(12)->(19)

(W1 W2 W3 W4 W21 W23 W7 W8 W9 W10 W11 W27)

P2: (1)->(2)->(3)->(4)->(5)->(13)->(14)->(8)->(9)->(10)->(11)->(12)->(19)

(W1 W2 W3 W4 W12 W13 W15 W8 W9 W10 W11 W27)

P3: (1)->(2)->(3)->(4)->(5)->(13)->(7)->(8)->(9)->(10)->(11)->(12)->(19)

(W1 W2 W3 W4 W12 W14 W7 W8 W9 W10 W11 W27)

P4: (1)->(2)->(3)->(4)->(5)->(7)->(8)->(9)->(10)->(11)->(12)->(19)

(W1 W2 W3 W4 W22 W7 W8 W9 W10 W11 W27)

P5: (1)->(2)->(3)->(4)->(5)->(6)->(7)->(8)->(9)->(10)->(11)->(12)->(19)

(W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W27)

Combinations for checking non-touching loops:
----

k=2:

Non-touching_loops:

Loops touching forward path:

Loops of touching path P1: L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25

Loops of touching path P2: L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25

Loops of touching path P3: L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25

Loops of touching path P4: L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25

```
Loops of touching path P5:  L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13
L14 L15 L16 L17 L18 L19 L20 L21 L22 L23 L24 L25

SOLUTION OF MASON'S RULE FOR THE INPUT NET:
==========================================

DELTA  = 1-(L1+L2+L3+L4+L5+L6+L7+L8+L9+L10+L11+L12+L13+L14+L15+
L16+L17+L18+L19+L20+L21+L22+L23+L24+L25)
DELTA1 = 1-(0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0)
DELTA2 = 1-(0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0)
DELTA3 = 1-(0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0)
DELTA4 = 1-(0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0)
DELTA5 = 1-(0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0)

MASON'S VALUE(out/in): T(s) = (P1 * DELTA1 + P2 * DELTA2 + P3 * DELTA3
+ P4 * DELTA4 + P5 * DELTA5 ) / DELTA

End of Execution!
```

Figure 6.12 Output file of *cimr* in **Figure 6.10**

CHAPTER 7

CONCLUSIONS AND FURTHER RESEARCH

This thesis has presented an approach to implementation of the computerized solution for Mason's rule, and it is a part of software tool development for study, evaluation and analysis of ASPN. An executable program (**cimr**) written by C language has been developed. It is able to evaluate the Mason's rule under UNIX environment. A complicated example in which it is very difficult to derive transfer functions is also tested. The implementation will play a very important role in the automation of performance analysis using moment generating function based approach for arbitrary stochastic Petri nets. The results can also find their applications in reduction of linear control systems.

The methodology of Petri Net is a graphical tool for modeling and analysis of discrete system. However, the modeling, design and analysis for Petri nets need be automated with the help of the synthesis methodologies and computer software technology. In order to implement computerized performance evaluation, modeling and system analysis for nets, our further and partially completed research and development for a software package are as follows:

a) *ASPN language*

In order to implement a computerized performance evaluation, modeling and analysis, we define and describe an ASPN language as discrete event dynamic system programming language. The method of definition employed is referred to as Backus-Naur form [46], or NBF. We present a BNF description of the ASPN language as the grammar of Petri net language. The BNF definition of ASPN language includes:

- program definition (program, heading and block)
- variable definition (integer, float and character type)
- constant definition (letter, digit, integer, real, sign, string)

- statement definition (assignment statement, place statement, arc statement, marking statement, net input and output statement, function call statement, compound statement, empty statement etc.)

There are several main advantages to define an ASPN language:

- We can design, model and analyze a Petri net by using the methodology of programming language;
- A language of PN benefits to the computerized processing for the complicated net construct;
- The combination of the language and graphical methodologies will be helpful to develop a new and efficient software tool for DEDS.

b) *Compiler of ASPN language*

A compiler needs to be developed to transfer source code of ASPN program describing an ASPN into its object program code run under UNIX. The development of the compiler can be implemented by using a UNIX tool, *yacc* [45]. yacc is a parser generator, that is, a program for converting a grammatical specification of a language.

c) *Reachability Graph Generator (RGG)*

The reachability graph of a PN is a set of all reachable markings (states) from an initial marking m_0 (initial state). Given a PN, we can obtain as many new markings as the number of enabled transitions. From each new marking, we can reach more markings. This process results in a tree representation of the markings, which is known as the Reachability Graph. The generation of reachability graph in the program is done by the function firing( ) [42]. It is one of the key functions of the program and calls several other functions during execution. It also calls itself for next firings until it terminates upon some conditions [42].

d) *Library of ASPN function (LAF)*

It is necessary to develop a library of ASPN function. These functions shall include:

- PN property class

- PN analysis class
- Transfer function class
- Performance evaluation class

e) *Computer Implementation of Mason's Rule (CIMR)*

A running program '*cimr*' has been completed. It can be either as a command of a new software tool or as the function call of ASPN language.

f) *Graphical User Interface and Environment*

Our goal is to design a Graphical User Interface (GUI) to put the modeling and performance analysis of ASPN into a window environment with a better look and feel. GUI describes a user interface that makes use of windows, menus, and other graphical objects and that, to a large extent, allows users to interact with the application by pointing and cliking mouse button. From an application developer's point of view, a GUI is a combination of a window manager, a style guide, and a library of routines (toolkit) that can be used to build the interface [43].

X-window is a windowing system capable of organizing graphics output in a hierarchy of windows on the screen. This capability and the ability to accept inputs from keyboard and mouse make X-window ideal for handling user interaction [43].

A GUI has four components:

- Window system

The graphical *window system* organizes output on the display screen and performs the basic text and graphics drawing functions.

- Window manager

The *window manage* provides the mechanism by which, when several window are on the screen, users can indicate the window with which they intend to interact.

- Toolkit

The *toolkit* is a library of routines with a well-defined programming interface.

- Style guide

The *style guide* specifies the appearance and behavior of the user interface of an application.

The X programming interface, Xlib, allows you to create window and handle basic input and output to build any graphical user interface you want. It is used to design and build a GUI for modeling and analyzing of ASPN. **Figure 7.1** shows basic functional blocks for development of ASPN software.

Generally, a new software tool for ASPN based on above will be the integration of Petri nets, moment generating function concepts, programming design and graph environment. It will result in a powerful and unified tool for DEDS.

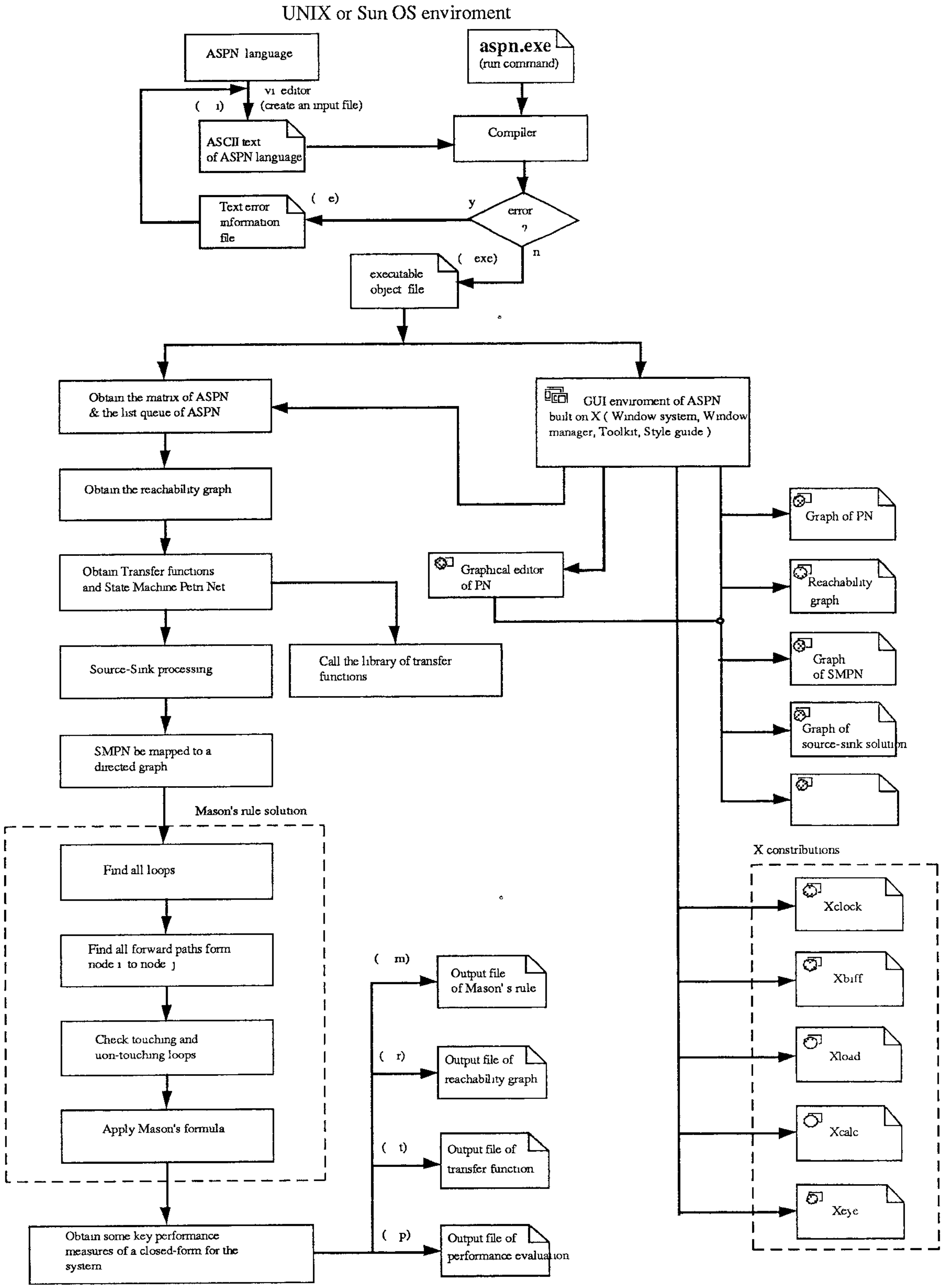


Figure 7.1 Basic functional blocks for development of ASPN software.

APPENDIX - SOURCE CODE OF CIMR

```
#ifndef LINT
static char sccsid[] = "@(#)Computer Implementation of Mason's Rule  V.1.00 02/15/92 ";
#endif

/************************************************************/

 /* This is the source code of the implementation of Mason's
    rule. It solves the transfer function form a given input
    node 'i' to a given output node 'j' in a State Machine
    Petri Nets. The program is also a part of the software
    developments for the computerized performance evaluation
    of Arbitrary Stochastic Petri Nets(ASPN).

    Author:   Xiaoyong Zhao
              Department of Computer & Information Science
              New Jersey Institute of Technology
*/
/************************************************************/


#include<stdio.h>
#include<string.h>
#include<math.h>
#include<memory.h>

#define FALSE  0
#define TRUE   1
#define MAX_NODE    50
#define MAX1   20
#define MAX2   1000
#define MAX_PATH   100
#define MAXLINE 128
#define DEFAULT_NO 30
```

```
#define SOURCE    10
#define SINGLE    11
#define MULTI     12
#define DESTIN    13


void read_input_file();
void initial();
void print_matrix();
void build_adj_list();
void initial_place_info_field();
void initial_qrow();
void initial_visited_flag();
void find_loops();
void check_same_loops();
void find_forward_paths();
void combination();
void check_nontouching_loops();
void output_nontouching_loops();
void loops_of_touching_path();
void appli_mason_formula();
void calculate_delta();
void calculate_delta_i();

struct list_node
     {
        struct list_node *next;
        int  id;
        int  w;
        int  node_type;
        int  input;
        int  in_p[MAX1+1];
        int  out_p[MAX1+1];
        int  visited;
     };
```

```
struct list_node *talloc();
struct list_node *place[MAX_NODE+1],*ptr;

struct operate
       {  int  qno;
          int  pqno;
          int  nqno;
          int  place[MAX1+1];
       };
struct operate qrow[MAX2+1];

unsigned int loopmat[MAX2+1][MAX_NODE+1];
unsigned int w_of_loop[MAX2+1][MAX_NODE+1];
unsigned int pathmat[MAX_PATH+1][MAX_NODE+1];
unsigned int w_of_path[MAX_PATH+1][MAX_NODE+1];
unsigned int comb_mat[MAX_NODE+1];
int nontouching_loops[51][MAX_NODE+1];
int loop_of_touch_path[MAX_PATH+1][MAX_NODE+1];

unsigned int matrix[MAX_NODE+1][MAX_NODE+1];
int cqueue;
int multi_place=0;
unsigned int ji,x,y,n,t=1;
unsigned int total_loop_number;
unsigned long int touching_index=0;
unsigned int nontouching_index=0;
int loop_in_delta[MAX2];
int check_nontouching_done=FALSE;

char filename[20];
char filename_o[20];
char filename_e[20];
char filename_m[20];
char strings[]="more -d ";
```

```
char strings1[]="SOLUTION OF MASON'S RULE FOR THE INPUT NET:\n";
char strings2[]="==========================================\n";
char strings3[]="* Wi refers to the transfer function.\n";
char strings4[]="* The elements which are not equal to zero in the metrix refer to the\n  index of the transfer function(Wi).";

FILE *fp_o;
FILE *fp_i;
FILE *fp_e;
FILE *fp_m;




main(argc,argv)
int argc; char *argv[];
{
int i=0;
char c;
char *p;

  if (argc<2)
     { printf( "Usage: cims filename\n");
       exit(1);
     }
  if ( (fp_i=fopen(argv[1],"r"))==NULL)
     { printf( "append: error opening file %s\n",argv[1]);
       exit(1);
     }
  p=strchr(argv[1],'.');
  if (p!=NULL) strcpy(p,"\0"); /* get the file name before the part of '.' */

  strcpy(filename_o,argv[1]);
```

```
strcpy(filename_e,argv[1]);
strcpy(filename_m,argv[1]);
strcat(filename_o, ".o");  /* get the file name of output file */
strcat(filename_e, ".e");  /* get the file name of error info. file */
strcat(filename_m, ".m");  /* get the file name of Mason' rule solution info. file */

fp_o=fopen(filename_o,"w");
fprintf(fp_o,"Output file: %s\n\n",filename_o);
fp_e=fopen(filename_e,"w");
fprintf(fp_e,"Output file: %s\n\n",filename_e);
fp_m=fopen(filename_m,"w");
fprintf(fp_m,"Output file: %s\n\n",filename_m);

read_input_file();
initial();
initial_place_info_field();
printf( "Start to search self loops ...");
find_loops();
printf( "\nObtain %d loops\n",total_loop_number);
printf( "Start to search paths ...");
find_forward_paths();
printf( "\nObtain %d paths\n",pathmat[0][0]);

printf( "Start to check nontouching loops ...");
fprintf(fp_o,"Combinations for checking nontouching loops:\n");
fprintf(fp_o,"-------------------------------------------
\n");
for (ji=2;ji<=total_loop_number;ji++)
  {
    fprintf(fp_o, "k=%d: ",ji);
    printf( "\nK = %d,",ji);
    combination(total_loop_number,ji);
```

```
        if (check_nontouching_done==TRUE) goto D;
      }
  D: printf( "\nChecking nontouching loops done\n");

    output_nontouching_loops();
    loops_of_touching_path();
    appli_mason_formula();

    fclose(fp_o);
    fclose(fp_e);
    fclose(fp_i);
    fclose(fp_m);

    strcat(strings,filename_o);
    system( strings);
    exit(0);
  }

/**************************************************************/

void read_input_file()
{
int line,i,j,jj,e,c;
int error_line_no[500];
int number;
int par_1,par_2,par_3;
char string[128];
char par[20];
char name1[20];
char ascii[20];
char *error[500];
char buffer[128];
char *pptr;
char *alloca();
   clear_matrix();
   line=i=e=0;
```

```
      while ( fgets(buffer,MAXLINE,fp_i)!=NULL) /* read a line */
         {  ++line;
            error[e]=alloca(128);
            pptr=buffer;
            for (i=1;i<=128;i++)
               string[i]='\0';
            i=0;
            j=0;
            while ( buffer[i]!='\n') /* move out '\t' and ' ' */
              {
               if ( buffer[i]!=' ' && buffer[i]!='\t')
                          string[j++]= buffer[i];
               i++;
              }
            string[j]='\0';

            if (string[0]=='/' && string[1]=='*')
               {     /* check expaination statment  */
               i=1;
               while ( string[++i]!='\n' && i<=128)
                 if (string[i]=='*' && string[i+1]=='/' && string[i+2]=='\0'  )
                    {  c=4;
                       goto A;
                    }
               goto B;
              }

            if (string[0]=='\0'|| string[0]=='\n')
               { c=4;
                 goto A;   /* if it is a space line */
               }


```

```
            /* check if there are explaination statments after a statment*/
            i=0;
            while ( string[i++]!=';') ;
            if (string[i]!='\0')
                  {  j=i;
                     if (string[i]=='/' && string[i+1]=='*' )
                        {
                           i++;
                           while ( string[++i]!='\n' && i<=128)
                           if (string[i]=='*' && string[i+1]=='/' && string[i+2]=='\0'  )
                              {
                                 string[j]='\0';
                                 goto C;
                              }
                           goto B;
                        }
                     else goto B;
                  }

   C:       if ( string[j-1]!=';' ) goto B; /* check if miss a ';' */


            i=0;
            while ( string[i]!= '=' && i<20) name1[i++]=string[i];
            name1[i++]='\0';

            if (strcmp(name1,"net")==0) c=0;
            else if (strcmp(name1,"input")==0) c=1;
                 else if (strcmp(name1,"output")==0) c=2;
                      else if (strcmp(name1,"node")==0) c=3;
                           else c=DEFAULT_NO;

```

```
A:      switch (c)
        { case 0:
                j=0;
                while ( string[i]!=';')
                  {
                   if ( string[i]<'0' || string[i]>'9') goto B;
                   ascii[j++]=string[i++];
                  }
                ascii[j]=string[i];
                if ((n=atoi(ascii))==0) goto B; /* get the node no. of the net */
                break;
          case 1:
                j=0;
                while ( string[i]!=';')
                  {
                   if ( string[i]<'0' || string[i]>'9') goto B;
                   ascii[j++]=string[i++];
                  }
                ascii[j]=string[i];
                if ((x=atoi(ascii))>n) goto B; /* get a source node id */
                break;
          case 2:
                j=0;
                while ( string[i]!=';')
                  {
                   if ( string[i]<'0' || string[i]>'9') goto B;
                   ascii[j++]=string[i++];
                  }
                ascii[j]=string[i];
                if ((y=atoi(ascii))>n) goto B; /* get a destination node id */
```

```
                    break;
               case 3:
                    if ( string[i++]!='(' )  goto B;
                    j=0;
                    while ( string[i]!=',')
                       if ( string[i]<'0' || string[i]>'9') goto B;
                       else ascii[j++]=string[i++];
                    ascii[j]='\0';
                    i++;
                    if ((par_1=atoi(ascii)) >n) goto B; /* get the first parameter for node statment */
                    j=0;
                    while ( string[i]!=',')
                       if ( string[i]<'0' || string[i]>'9') goto B;
                       else ascii[j++]=string[i++];
                    ascii[j]='\0';
                    i++;
                    if ((par_2=atoi(ascii))>n) goto B; /* get the second parameter for node statment */
                    j=0;
                    while ( string[i]!=')')
                       if ( string[i]<'0' || string[i]>'9') goto B;
                       else ascii[j++]=string[i++];
                    ascii[j]='\0';
                    par_3=atoi(ascii); /* get the 3th parameter for node statment */
                    if ( string[i]!=')' || string[i+1]!=';' ) goto B;
                    matrix[par_1][par_2]=par_3;
                    break;
            case 4: break;
  B:     default: error[e]=strcpy(error[e],pptr);
                    error_line_no[e++]=line;
```

```
                    break;
                }
            }
        if (e>0) /* output error info. for the input file */
          {  for (j=0;j<e;j++)
                { printf( "***** ERROR: line %d: %s\n", error_line_no[j], error[j]);
                  fprintf(fp_e, "***** ERROR: line %d: %s\n", error_line_no[j],error[j]);
                }
             printf( "ERROR(S): %d\n",e);
             fprintf(fp_e, "ERROR(S): %d\n",e);
             exit(1);
          }
}


/***************************************************************/

void initial()
{
int i,j;

print_matrix();
build_adj_list();
}


/***************************************************************/

clear_matrix()
{
int i,j;

```

```
for (i=1;i<=n;i++)
   for (j=1;j<=n;j++)
       matrix[i][j]=0;

}


/***************************************************************/

void print_matrix()
{
int i,j;

fprintf(fp_o,  "The adjacncy matrix(%dx%d):\n",n,n);
fprintf(fp_o,  "--------------------------\n");
for (i=1;i<=n;i++)
   for (j=1;j<=n;j++)
      {
       fprintf(fp_o,  "%d ",matrix[i][j]);
       if (j==n)
           {
             fprintf(fp_o, "\n");
           }
       }
fprintf(fp_o,"%s",strings4);
}



/*****************************************************************/

void build_adj_list()
{
int i,j;

for (i=1;i<=n;i++)
```

```
   {
    place[i]=talloc();
    if ( place[i]==0) { printf( "\n***** ERROR: invalid
address!\n");
                                  exit(1); }
    ptr=place[i];
    ptr->id=i;
    for (j=1;j<=n;j++)
        {
          if (matrix[i][j]!=0)
            {
              ptr->w=matrix[i][j];
              ptr->next=talloc();
    if ( ptr->next==0) { printf( "\n***** ERROR: invalid
address!\n");
                                  exit(1); }
              ptr=ptr->next;
              ptr->id=j;
              ptr->visited=0;
            }
        }
    ptr=NULL;
    }
fprintf(fp_o,  "\n\nAdjacency List Queue:\n");
fprintf(fp_o,  "--------------------\n");
for (i=1;i<=n;i++)
  {
  ptr=place[i];
  while (ptr!=NULL && ptr->id!=0 )
   {
    if (ptr->w==0)
          fprintf(fp_o,  "(%d)",ptr->id);
    else
          fprintf(fp_o,  "(%d)--> w%d",ptr->id,ptr->w );

    ptr=ptr->next;
```

```
      }
     fprintf(fp_o, "\n");
     }
fprintf(fp_o,"%s",strings3);
}



/*****************************************************************/

void initial_place_info_field()
{
int i,j,r;

for (i=1;i<=n;i++)
    {r=0;
     for (j=1;j<=n;j++)
         if (matrix[i][j]!=0) place[i]->out_p[r++]=j;
    }

for (j=1;j<=n;j++)
   { r=0;
     for (i=1;i<=n;i++)
         if (matrix[i][j]!=0) place[j]->in_p[r++]=i;
   }

for (i=1;i<=n;i++)
   { r=0;
     while ( place[i]->in_p[r++]!=0) place[i]->input++;
   }

for (i=1;i<=n;i++)  place[i]->visited=0;

for (i=1;i<=n;i++)
     if (place[i]->input>1) place[i]->node_type=MULTI;
     else place[i]->node_type=SINGLE;
```

```
}



/**************************************************************/

struct list_node *talloc()
{
int *malloc();
return((struct list_node*)malloc(sizeof(struct list_node)));
}



/**************************************************************/

/*
void copy_matrix()
{
int i,j;

  for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)
          m[i][j]=matrix[i][j];

}
*/


/**************************************************************/

void find_loops()

{
```

```
int i,j,r,pi=1,pj=1;
int N1,k,s=0;
int multi_place=FALSE;
int loop[MAX_NODE+1];
int pqindex[MAX2+1];

L1: ;
    initial_qrow();
    initial_visited_flag();
    for (i=0;i<=MAX2;i++)
        for (j=1;j<=MAX_NODE;j++)
            loopmat[i][j]=0;

    cqueue=1;
    place[x]->node_type=MULTI;

L2: for (i=1;i<=n;i++)
       if (place[i]->node_type==MULTI)
          {  loop[++s]=place[i]->id;
             multi_place=TRUE;
          } /* there are  multi-places */
    loop[0]=s;
    if (multi_place==FALSE) goto L12;
    i=1;
L3: if (i>s) goto L11;
L4: initial_qrow();
    initial_visited_flag();
    r=0;
    cqueue=1;
    qrow[1].qno=cqueue;
    qrow[1].pqno=0;
    qrow[1].nqno=1;
    qrow[1].place[r]=loop[i];
    place[ loop[i]]->visited=1;
    k=0;
    pqindex[++k]=1;
```

```
L5: if (qrow[1].nqno<=0)
    { i++;
      goto L3;
    }
L6: ;
    if (place[qrow[cqueue].place[r]]->out_p[0]==0) goto L7;
    if (place[qrow[cqueue].place[r]]->out_p[0]!=0 &&
                  place[qrow[cqueue].place[r]]->out_p[1]==0)
        { qrow[cqueue].place[r+1]=
place[qrow[cqueue].place[r]]->out_p[0];
          qrow[cqueue].nqno=1;
          place[qrow[cqueue].place[++r]]->visited++;
        }
    else {
          N1=0;
          while (place[qrow[cqueue].place[r]]->out_p[N1++]!=0)
               {  qrow[cqueue+N1].place[0]=
place[qrow[cqueue].place[r]]->out_p[N1-1];
                  qrow[cqueue+N1].qno=cqueue+N1;
                  qrow[cqueue+N1].nqno=1;
                  qrow[cqueue+N1].pqno=cqueue;
                  place[qrow[cqueue+N1].place[0]]->visited++;
               }
          qrow[cqueue].nqno=N1-1;
          cqueue=cqueue+N1-1;
          r=0;
          pqindex[++k]=cqueue;
         }

L7: if (place[qrow[cqueue].place[r]]->visited>1)
        if ( qrow[cqueue].place[r]==loop[i])
           {  /* get a loop */
             pj=0;
             for (j=1;j<=k;j++)
                {  r=0;
                   while ( qrow[pqindex[j]].place[r]!=0 )
```

```
                        loopmat[pi][++pj]=qrow[pqindex[j]].place[r++];
                     }

                  if (loopmat[pi][1]==loopmat[pi][pj])
                     {          /*don't need the last node because it is also the first node of the loop. */
                        loopmat[pi][pj]=0;
                        loopmat[pi][0]=pj-1;
                        pi++;
                     }
                  else for (i=0;i<=n;i++)
                          loopmat[pi][i]=0; /* This is not a loop, clear it. */
                  qrow[cqueue].nqno--;
                  goto L10;
               }
            else { qrow[cqueue].nqno--;
                     goto L10;
                  }
    L8: if (qrow[cqueue].place[r]==y)
            {  qrow[cqueue].nqno--;
               goto L10;
            }
    L9: for (j=1;j<=loop[0];j++)
             if ( qrow[cqueue].place[r]==loop[j] &&
                  place[qrow[cqueue].place[r]]->visited >1)
                {  qrow[cqueue].nqno--;
                   goto L10;
                }
        goto L5;
    L10: if ( qrow[cqueue].nqno==0)
             { qrow[qrow[cqueue].pqno].nqno--;
               r=0;
               while ( qrow[cqueue].place[r]!=0)
                      place[qrow[cqueue].place[r++]]->visited--;
```

```
                qrow[cqueue].qno=0; /* clear current queue */
                qrow[cqueue].pqno=0;
                qrow[cqueue].nqno=0;
                for (j=0;j<=MAX1;j++)
                    qrow[cqueue].place[j]=0;
                cqueue--;
                if (cqueue<=0) goto L5;
                pqindex[k]=cqueue;
                if (pqindex[k]==pqindex[k-1])  k--;
                r=0;
                goto L10;
              }
          else  { if (cqueue<=0) goto L5;
                   goto L7;
                 }
     L11: loopmat[0][0]=pi-1;
          check_same_loops();
          fprintf(fp_o,  "\n\nLoop(s):\n");
          fprintf(fp_o,  "--------\n");
          fprintf(fp_m,  "\n\nLoop(s):\n");
          fprintf(fp_m,  "--------\n");
          i=0;
          j=0;
          k=0;
          for (r=1;r<=loopmat[0][0];r++)
          {
          while ( loopmat[++i][j+1]!=0)
            {
            fprintf(fp_o, "L%d: ", ++k);
            fprintf(fp_m, "L%d: ", k);
            while ( loopmat[i][++j]!=0)
                 fprintf(fp_o,  "(%d)->", loopmat[i][j]);
            j=0;
     /* output the 1th id of the loop, because a loop will return
to its initial place */
            fprintf(fp_o, "(%d)\n",loopmat[i][1]);
```

```

/* get the transfer functions for each path. The transfer
function is noted as varible 'Wi'.  'strct list_node' has
recorded the 'Wi' in the list queue of a state machine Petri
net. Note: 'Wi' is a function vriable of 's' and is noted as
Wi(s) in Moment Generating Function -based method.
*/
       j=0;
       fprintf(fp_o,  "    (", i );
       while ( loopmat[i][++j]!=0)
         {
         ptr=place[loopmat[i][j]];
         if (loopmat[i][j+1]!=0)
            while ( ptr->next->id!=loopmat[i][j+1]) ptr=ptr->next;

         else {  /* The loop come back to the 1th node id */
               while ( ptr->next->id!=loopmat[i][1]) ptr=ptr->next;

              }
         w_of_loop[i][j]=ptr->w; /* obtain the index of 'Wi' */

         fprintf(fp_o,  "W%d ", w_of_loop[i][j]);
         fprintf(fp_m,  "W%d ", w_of_loop[i][j]);

         }
       j=0;
       fprintf(fp_o,  ")\n");
       fprintf(fp_m,  "\n");

       }
     }
     fprintf(fp_o,  "\n");
     return;
L12: printf( "There doesn't exist any loop(s) in the net.");
}

```

```
/*****************************************************************/


void check_same_loops()
{
unsigned long int add_sum[MAX2+1];
unsigned long int mult_sum[MAX2+1];
int i,j=1,v=0,k,r;
unsigned int lp[MAX2+1][MAX_NODE+1];


for (i=1;i<=MAX2;i++)
    {
     add_sum[i]=0;
     mult_sum[i]=1;
    }

for (i=1;i<=loopmat[0][0];i++)
  {
    while ( loopmat[i][j]!=0)
        add_sum[i]=add_sum[i]+loopmat[i][j++];
    j=1;
  }
j=1;
for (i=1;i<=loopmat[0][0];i++)
  {
    while ( loopmat[i][j]!=0)
        mult_sum[i]=mult_sum[i]*loopmat[i][j++];
    j=1;

  }
v=0;
for (i=1;i<=loopmat[0][0];i++)
   for (j=1;j<=loopmat[0][0];j++)
```

```
        if (i!=j && loopmat[j][0]!=0 && loopmat[i][0]!=0 && loopmat[i][0]==loopmat[j][0] )
            if (add_sum[i]==add_sum[j] && mult_sum[i]==mult_sum[j] )
                {
                    v=0;
                    while ( loopmat[j][++v]!=0)
                      if (loopmat[i][1]==loopmat[j][v]) break;
                    k=v;
                    r=0;
                    while ( loopmat[j][v]!=0) lp[j][++r]=loopmat[j][v++];
                    for (v=1;v<k;v++) lp[j][++r]=loopmat[j][v];
                    v=0;
                    r=0;
                    while ( loopmat[i][++r]!=0)
                        if (loopmat[i][r]!=lp[j][r]) v=1;
/*
        if (v==1)
         {
                v=0;
                fprintf(fp_m, "%d: ",i);
                while ( loopmat[i][v]!=0 )
                   fprintf(fp_m, "%d->",loopmat[i][v++]);
                fprintf( fp_m,"\n");
                v=0;
                fprintf(fp_m, "%d: ",j);
                while ( loopmat[j][v]!=0 )
                   fprintf(fp_m, "%d->",loopmat[j][v++]);
                fprintf( fp_m,"\n");
                fprintf(fp_m, "%d\n",j);
          v=0;
          }
*/
                v=0;
```

```
                    while ( loopmat[j][v]!=0 )
                          loopmat[j][v++]=0;
                    v=0;
                   }
     /* get total loops */
      j=1;
      r=0;
      for (i=1;i<=loopmat[0][0];i++)
           if ( loopmat[i][1]!=0 )
             {
              while ( loopmat[i][++r]!=0)
                    loopmat[j][r]=loopmat[i][r];
              while ( loopmat[j][r]!=0)
                    loopmat[j][r++]=0; /* when shift a loop, we must clear zeao  behind their varible. */
              j++;
              r=0;
             }
      total_loop_number= --j;
      loopmat[0][0]=j++;
      for (i=j;i<=MAX2;i++)
           for (r=0;r<=MAX_NODE;r++)
                loopmat[i][r]=0;
      return;
   }

/*****************************************************************/

   void find_forward_paths()
   {
   int i,j,r=0,k,pi=1,pj=1;
```

```
int N1;
int pqindex[MAX2+1];

P1: initial_qrow();
    initial_visited_flag();
    cqueue=1;
    for (i=1;i<=MAX_PATH;i++)
      { pqindex[i]=0;
        for (j=1;j<=MAX_NODE;j++)
            pathmat[i][j]=0;
      }
P2: r=0;
    qrow[1].place[r]=x;
    qrow[1].qno=cqueue;
    qrow[1].pqno=0;
    qrow[1].nqno=1;
    place[x]->visited++;
    cqueue=1;
    k=0;
    pqindex[++k]=1;

P3: if ( qrow[cqueue].nqno==0 ) goto P8;
P4: if ( place[qrow[cqueue].place[r]]->out_p[0]==0) goto P5;
    if ( place[qrow[cqueue].place[r]]->out_p[0]!=0 &&
           place[qrow[cqueue].place[r]]->out_p[1]==0 )
       { qrow[cqueue].place[r+1]=
place[qrow[cqueue].place[r]]->out_p[0];
         qrow[cqueue].nqno=1;
         place[qrow[cqueue].place[++r]]->visited++;
         goto P5;
       }
    else { N1=0;
           while (place[qrow[cqueue].place[r]]->out_p[N1++]
!=0)
                 {
qrow[cqueue+N1].place[0]=place[qrow[cqueue].place[r]]->out_p[N1-1];
```

```
                     qrow[cqueue+N1].qno=cqueue+N1;
                     qrow[cqueue+N1].nqno=1;
                     qrow[cqueue+N1].pqno=cqueue;
                /* place[qrow[cqueue+N1].place[0]]->visited++; */
                  }
               qrow[cqueue].nqno=N1-1;
               cqueue=cqueue+N1-1;
               place[qrow[cqueue].place[0]]->visited++;
               r=0;
               pqindex[++k]=cqueue;
             }

   P5: if ( qrow[cqueue].place[r]==y)
         {                 /* get a path */
           pj=0;
           for (i=1;i<=k;i++)
             { r=0;
               while ( qrow[pqindex[i]].place[r]!=0)
pathmat[pi][++pj]=qrow[pqindex[i]].place[r++];
             }
           pi++;
           qrow[cqueue].nqno--;
           goto P7;
         }
   P6: if ( place[qrow[cqueue].place[r]]->visited > 1)
         { qrow[cqueue].nqno--;
           goto P7;
         }
       else goto P3;
   P7: if ( qrow[cqueue].nqno==0)
        {
        if (cqueue==1) goto P8;
        else /* abandon this current queue */
         { qrow[qrow[cqueue].pqno].nqno--;
           r=0;
```

```
            while ( qrow[cqueue].place[r]!=0)
                  place[qrow[cqueue].place[r++]]->visited--;
            for (i=0;i<=MAX1;i++)
                  qrow[cqueue].place[i]=0;
            cqueue--;
            if ( qrow[cqueue].nqno==1) /* or say it !=0 */
                     place[qrow[cqueue].place[0]]->visited++;
            pqindex[k]=cqueue;
            if (pqindex[k]==pqindex[k-1]) k--;
            r=0;
            goto P7;
          }
         }
       else goto P6;

P8: fprintf(fp_o,"\n\nThe forward path(s) from node %d to node %d:\n",x,y);
       fprintf(fp_o,    "-------------------------------------------\n");
       fprintf(fp_m,"\n\nThe forward path(s) from node %d to node %d:\n",x,y);
       fprintf(fp_m,    "-------------------------------------------\n");
       i=0;
       j=0;
       while ( pathmat[++i][++j]!=0)
        {
          fprintf(fp_o, "P%d: ", i);
          fprintf(fp_m, "P%d: ", i);
          while ( pathmat[i][j+1]!=0)
             fprintf(fp_o,  "(%d)->", pathmat[i][j++]);
          fprintf(fp_o,  "(%d)", pathmat[i][j]);
          fprintf(fp_o,  "\n");

/* get the transfer functions for each path. The transfer
function is noted as varible 'Wi'. The 'strct list_node' has
```

```
recorded the 'Wi' in the list queue of a state machine Petri
net.  Note: 'Wi' is a function of vriable 's' and is noted as
Wi(s) in Moment Generating Function -based method.
*/
       j=0;
       fprintf(fp_o,  "    (", i );
       while ( pathmat[i][++j]!=0)
         {
         if (pathmat[i][j+1]!=0)
            {
             ptr=place[pathmat[i][j]];
             while (ptr->next->id!=pathmat[i][j+1]) ptr=ptr->next;
             w_of_loop[i][j]=ptr->w; /* obtain the index of 'Wi' */
             fprintf(fp_o,  "W%d ", w_of_loop[i][j]);
             fprintf(fp_m,  "W%d ", w_of_loop[i][j]);
            }
         }

       j=0;
       fprintf(fp_o,  ")\n");
       fprintf(fp_m,  "\n");


     }
    fprintf(fp_o, "\n\n");
    fprintf(fp_m,  "\n\n");
    pathmat[0][0]=i-1;
    return;
}



/*****************************************************************/
```

```
void initial_qrow()
{
int i,j;

for (i=1;i<=MAX2;i++)
{
  qrow[i].qno=0;
  qrow[i].pqno=0;
  qrow[i].nqno=0;
  for (j=1;j<=MAX1;j++)
      qrow[i].place[j]=0;
}
return;
}



/*****************************************************************/


void initial_visited_flag()
{
  int i;
  for (i=1;i<=n;i++)   place[i]->visited=0;
  return;
}



/*****************************************************************/

/* This combination algorithm is to check touching and
non-touching loops in the implementation of Mason's rule. We
can assign each loop an array index(cc[i]), where i=1,2,...n.
Then, we have to decide that which combination of k value be
```

```
compared for the touching and non-touching cases.
*/


void combination(n_loops,k)

int k, n_loops;
{
int i,j,r,cc[100];


    cc[0]= -1;
    for (i=1;i<=k;i++)  cc[i]=i;
    j=1;
    while ( j!=0)
        {
            /* get one of the combinations for k, begin from
comb_mat[k][1]. com_mat[][] will be update when call the combination
function each time. */

/*
            fprintf(fp_o, "( ");
*/
            for (i=1;i<=k;i++)
              {
                comb_mat[i]=cc[i];
/*
                fprintf(fp_o, "L%d ",comb_mat[i]);
*/
              }
/*
            fprintf(fp_o, "), ");
*/
            check_nontouching_loops(k);

            j=k;
```

```
            while ( cc[j]==n_loops-k+j)
                  j--;
            cc[j]++;
            for (i=j+1;i<=k;i++)
                  cc[i]=cc[i-1]+1;
          }
        /* we have got a set of combinations for k */
        fprintf(fp_o, "\n\n");

        /* we output the ID of the touching loops for k case */
        nontouching_loops[0][0]=nontouching_index;
        if (nontouching_loops[0][0]==0) check_nontouching_done=TRUE;
        /* after checking k=2, if all loops of k=2 are nontouching, the check done */

        t=touching_index+1;
    }


/****************************************************************/

/* After getting a combination of IDs of k loops, this
function will check if touching or non-touching for these
loops. After checking, we get a touching combination and save
the IDs of touching loops in touchin_loops[i][j], and begin
at i=1, j=1 */

void check_nontouching_loops(k)
int k;
{
int i,j,ii,iii,s;

   i=0, iii=1;
   j=1;
```

```
      /* begin to compare the elments of two loops to check them if touching each other. */
      while ( ++j<=k  )
        {
         while ( loopmat[comb_mat[iii]][++i]!=0 )
            {
            ii=0;
            while ( loopmat[comb_mat[j]][++ii]!=0  )
              if (loopmat[comb_mat[iii]][i]== loopmat[comb_mat[j]][ii] )
                   return;
            }
         i=0, iii++;
        }
      /* nontouching loops */
      nontouching_loops[++nontouching_index][0]=k;
      for (s=1;s<=k;s++)
            nontouching_loops[nontouching_index][s]=comb_mat[s];
      return;
   }



/****************************************************************/

   /*
   void output_touching_loops()
   {
   int i,j,k;

       i=1;
       k=2;

       fprintf(fp_o, "\nTouching_loops:\n");
       fprintf(fp_o,   "--------------");

```

```
     while ( i<=touching_loops[0][0])
         {
            fprintf(fp_o, "\nk=%d: ",k );
            while (touching_loops[i][0]==k)
              {
                j=0;
                fprintf(fp_o, "{ ");
                while ( touching_loops[i][++j]!=0)
                      fprintf(fp_o, "L%d ", touching_loops[i][j] );
                fprintf(fp_o, "}, ");
                i++;
              }
            fprintf(fp_o, "\n");
            k++;
         }
     fprintf(fp_o, "\n\n");

}
*/


/****************************************************************/
/* There are k=2,3,4,... combinations of nontouching loops.
We have obtained that the total number of nontouching loops
stored in array varible 'nontouching_loops[0][0]'.
'nontouching_loops[i][0]' stores the value 'k' of the
combination case of each nontouching loop. This subroutin is
to output each nontouching loop and their combination case
value 'k'. */

void output_nontouching_loops()
{
int i,j,k;

    i=1;
```

```
        k=2;

        fprintf(fp_o, "\nNon-ouching_loops:\n");
        fprintf(fp_o,   "-----------------");

        while ( i<=nontouching_loops[0][0])
            {
                fprintf(fp_o, "\nk=%d: ",k );
                while (nontouching_loops[i][0]==k)
                  {
                    j=0;
                    fprintf(fp_o, "{ ");
                    while ( nontouching_loops[i][++j]!=0)
                           fprintf(fp_o, "L%d
",nontouching_loops[i][j]);
                    fprintf(fp_o, "}, ");
                    i++;
                  }
                k++;
            }
        fprintf(fp_o, "\n\n");

    }


/*****************************************************************/
    /* This function gets the loops of touching a path. We check
    each path and see if there are any loops which touch this
    path.  Array 'loop_of_touch_path[pi][j]' stores the id of
    these loops, where 'pi' is the id of paths, 'j' is the id of
    these loops */

    void loops_of_touching_path()

    {
    int pi,pj,li,lj;
```

```
int i,j;

    fprintf(fp_o, "\n\nLoops touching forward path:\n");
    fprintf(fp_o,     "---------------------------");

    for (pi=1;pi<=pathmat[0][0];pi++)
       {
         fprintf(fp_o, "\nLoops of touching path P%d:  ",pi);

         j=0;
         for (li=1;li<=total_loop_number;li++)
           { pj=0;
             while ( pathmat[pi][++pj]!=0)
                 { lj=0;
                   while ( loopmat[li][++lj]!=0)
                         if ( pathmat[pi][pj]==loopmat[li][lj])
                            {  loop_of_touch_path[pi][++j]=li;
                               fprintf(fp_o, "L%d ",li);
                               goto D;
                            }
                 }
D:              ;
           }
       }
    fprintf(fp_o, "\n\n");
}


               /* output the terms behind the 1th term  */



/******************************************************************/

void appli_mason_formula()
{
```

```
int i,ii;

   fprintf(fp_o, "\n\n%s%s\n",strings1,strings2);
   fprintf(fp_m, "%s%s",strings1,strings2);

   calculate_delta();
   fprintf(fp_o, "\n");

   for (ii=1;ii<=pathmat[0][0];ii++)
        calculate_delta_i(ii);
   fprintf(fp_o, "\n");

   fprintf(fp_o, "\n\nMASON'S VALUE(out/in): T(s) = (");
   fprintf(fp_m, "\n\nMASON'S VALUE(in/out): T = (");
   for (i=1;i<pathmat[0][0];i++)
       {
         fprintf(fp_o,"P%d * DELTA%d + ",i,i);
         fprintf(fp_m,"P%d * DELTA%d + ",i,i);
       }
   fprintf(fp_o,"P%d * DELTA%d ) / DELTA\n",i,i);
   fprintf(fp_m,"P%d * DELTA%d ) / DELTA\n",i,i);

   fprintf(fp_o,     "\n\nEnd of Execution!");

   return;
}


/*************************************************************/
/* We need to determin the value of each loop for each
DELTAi. If a loop touches the ith path, then the value of the
loop is zero or the state of the loop is '-1' ( We use '-1'
in this function and the expresstion of the loops is refered
to loop_in_delta[]'). The loop will be removed from DELTA, if
it is '0' or '-1'. The meanning about this see the definition
of Mason's rule.
```

```
*/
void calculate_delta_i(pi)
int pi;
{
int i,j,r;

for (i=1;i<=total_loop_number;i++)
      {
         j=1;
         while (i!=loop_of_touch_path[pi][j] && loop_of_touch_path[pi][j]!=0) j++;
         if (loop_of_touch_path[pi][j]==0 )
            loop_in_delta[i]=j;  /* loop j does not touch the ith path */
         else loop_in_delta[i]= -1; /*loop touches the ith path and it will be removed from DELTA.    */

      }
loop_in_delta[i]=0;  /* the last unit is set to zero */



/******************************************************************/

/* We have already obtained the loops for DELTAi, i.e. if a
loop touches the ith path, it will be removed from DELTA (
where Li=-1 or Li=0). Therefore we can get DELTAi as follows:
*/
j=0;
if ( total_loop_number>0 )
    {
        fprintf(fp_o, "DELTA%d = 1-(",pi);
        fprintf(fp_m, "DELTA%d = 1-(",pi);

        /* ouptput the 1th loop behind '(' */
        if (loop_in_delta[++j]!= -1 && loop_in_delta[j]!=0)
```

```
              {
                fprintf(fp_o, "L%d",j);
                fprintf(fp_m, "L%d",j++);
              }
          else  if (loop_in_delta[j++]!=0)
                    {
                         fprintf(fp_o,"0");
                         fprintf(fp_m,"0");
                       }

          if ( total_loop_number>=j )
               for (i=j;i<=total_loop_number;i++)
                    if (loop_in_delta[i]!= -1)
                       {    fprintf(fp_o, "+L%d",i);
                            fprintf(fp_m, "+L%d",i);
                         }
                    else {
                            fprintf(fp_o, "+0");
                            fprintf(fp_m, "+0");
                            }

    /* output the multiply of nontouching loops */
          if ( nontouching_loops[0][0]>0 )
             {
              j=1;
              i=1;
              /* output the 1th term for the multip. of nontouching loops */
              if (loop_in_delta[nontouching_loops[i][j]]!= -1 )
                 {  /* is not the state '-1' */
fprintf(fp_o,")+(L%d",nontouching_loops[i][j]);
fprintf(fp_m,")+(L%d",nontouching_loops[i][j]);
                 }
              else {  fprintf(fp_o,")+(0");
```

```
                    fprintf(fp_m,")+(0");
                 }


/******   output the terms behind the 1th term  ********/
              for (i=1;i<=nontouching_loops[0][0];i++)
                {
                  while ( nontouching_loops[i][++j]!=0 )
                    {
                     if (loop_in_delta[nontouching_loops[i][j]]!= -1 )
                        {  /* is not the state '-1' */
                         fprintf(fp_o, "*L%d",nontouching_loops[i][j]);
                         fprintf(fp_m, "*L%d",nontouching_loops[i][j]);
                        }
                     else {  fprintf(fp_o,"*0");
                             fprintf(fp_m,"*0");
                          }
                    }

             if ( nontouching_loops[i+1][1]!=0 && i<= nontouching_loops[0][0])
                    {
                     if (loop_in_delta[nontouching_loops[i+1][1]]!= -1 )
                       {
fprintf(fp_o,"+L%d",nontouching_loops[i+1][1]);
fprintf(fp_m,"+L%d",nontouching_loops[i+1][1]);
                       }
                     else {  fprintf(fp_o,"+0");
                             fprintf(fp_m,"+0");
                          }
```

```

                    j=1;
                  }
              }
          }
      fprintf(fp_o, ")\n");
      fprintf(fp_m, ")\n");

   }
else  {   fprintf(fp_o, "DELTA%d = 1\n",pi);
          fprintf(fp_m, "DELTA%d = 1\n",pi);
      }
}






/**************************************************************/

void calculate_delta()
{
int i,j;

j=1;
i=1;
if ( total_loop_number>0 )
   {
      fprintf(fp_o, "DELTA  = 1-(L%d",i);
      fprintf(fp_m, "DELTA  = 1-(L%d",i);
      if ( total_loop_number>1 )
           for (i=2;i<=total_loop_number;i++)
                {
                  fprintf(fp_o, "+L%d",i);
                  fprintf(fp_m, "+L%d",i);
```

```
                    }
        if ( nontouching_loops[0][0]>0 )
          {            /* output the multiply of nontouching loops */
          j=1;
          i=1;
          fprintf(fp_o,")+(L%d",nontouching_loops[i][j]);
          fprintf(fp_m,")+(L%d",nontouching_loops[i][j]);
          for (i=1;i<=nontouching_loops[0][0];i++)
              {
              while ( nontouching_loops[i][++j]!=0 )
                {
                 fprintf(fp_o, "*L%d", nontouching_loops[i][j]);
                 fprintf(fp_m, "*L%d", nontouching_loops[i][j]);
                }
              if ( nontouching_loops[i+1][1]!=0 )
                 {  if (nontouching_loops[i+1][0]%2==0)
                    { /* for combination k=2,4,6,8,... */
                      fprintf(fp_o, "+L%d", nontouching_loops[i+1][1]);
                      fprintf(fp_m, "+L%d", nontouching_loops[i+1][1]);
                      j=1;
                    }
                    else {/* for combination k=3,5,7,9,...*/
                             fprintf(fp_o, "-L%d", nontouching_loops[i+1][1]);
                             fprintf(fp_m, "-L%d", nontouching_loops[i+1][1]);
                             j=1;
                         }
                 }
              }
          }
```

```
        fprintf(fp_o, ")\n");
        fprintf(fp_m, ")\n");
    }
else {
            fprintf(fp_o, "DELTA = 1\n");
            fprintf(fp_m, "DELTA = 1\n");
        }

}

```

REFERENCES

[1] Guo, D. L., F. DiCesare, and M. C. Zhou. "Moment Generating Function Approach to Performance Analysis of Extended Stochastic Petri Nets," *Proc. of IEEE Int. Conf. on Robotics and Automation,* pp. 1309-1314, Sacramento, CA, 1991.

[2] Guo, D. L, F. DiCesare, and M. C. Zhou. "A Moment Generating Function Based Approach for Evaluating Extended Stochastic Petri Nets, " To appear in *IEEE Trans. Automatic Control,* 1992

[3] Zhou, M. C., D. L. Guo and F. DiCesare. "Integration of Petri nets and Moment Generating Function Approaches for System Performance Evaluation," To appear in *J. of System Integration*, 1993.

[4] Molloy, M. K.. "Discrete Time Stochastic Petri Nets," *IEEE Transaction on Software Engineering,* 11(2), pp. 417-423, 1985.

[5] Holiday, M. A. and M. K. Vernon. "A Generalized Timed Petri Net Model for Performance Analysis," in *IEEE Proc. Int. Workshop on Timed Petri Nets,* Torino, Italy, July, 1985.

[6] Petri, C. A. "Kommunikation mit Automaten." Bonn: Institute fur Instrumentelle Mathematik, Schritten des IIM Nr. 3, 1962. Also, English translation, "Communication with automata" New York: Griffiss Air Force Base. Tech. Rep. RADCTR-65-377, vol. 1, Suppl. 1, 1996.

[7] Dorf, R. C. *Modern Control Systems,* Addison-Wisley, 1990.

[8] Kamen, E. *Introduction to signals and Systems*, MacMillan, New York, 1987.

[9] Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*, The Southeast Book company, 1983.

[10] Wang, C. H. "Computer-aided-manipulation of Mason's formula and its applications," *Pros. of IEEE Int. Conf. On Systems, Man and Cybernetics*, Charlottesville, VA, pp.493-500, 1991.

[11] Berlioux, P., and P. Bizard. *Algorithms 2: Data Structures & Algorithms,* John Wiley & Sons Ltd., 1990.

[12] Al-Jaar, R. Y., and A. A. Desrochers. "Petri Nets in Automation and Manufacturing," in *Advance in Automation and Robotics* (ed. G. N. Saridis), JAI Press, Greenwich, Conn., Vol. 2, 1991.

[13] Al-Jaar, R. Y., and A. A. Desrochers. "A Survey of Petri Nets in Flexible Manufacturing Systems," *Proceedings of the 1988 IMACS Conference*, Paris, France, July 1988.

[14] Peterson, T. L. *Petri Net Theory and the Modeling of Systems,* Prentice-Hall, Inc. Englewood Cliffs, 1981.

[15] Sifakis, J. "Use of Petri Nets for Performance Evaluation," in *Measuring, Modeling, and Evaluating Computer Systems, Acta Cybernet*, Vol. 4, No. 2, pp. 185-202, 1978.

[16] Cohen, G., D. Dubois, J. P. Quadrat, and M. Viot. "Linear System Theory for Discrete Event Systems," *Proceedings of the 23rd IEEE Conference on Decision and Control,* Las Vegas, Nev., pp. 539-544, Dec., 1984.

[17] Cohen, G., D. Dubois, J. P. Quadrat, and M. Viot. "A Linear-system -- Theoretic View of Discrete Event Processes and Its Use for Performance Evaluation in Manufacturing," *IEEE Transactions on Automatic Control,* Vol. AC-30, No. 3, pp. 210-220, March 1985.

[18] Molloy, M. K. "Performance Analysis Using Stochastic Petri Nets," *IEEE Tansactions on Computers,* Vol. C-3, No. 9, pp. 913-917, 1982.

[19] Agerwala, T., and Y. Choed-Amphai. "A Synthesis Rule for Concurrent Systems," *Proc. of Design Automation Conference,* pp. 305-311, 1978.

[20] Murata, T. "Petri Nets: Properties, Analysis and Application," Proc. of the IEEE, Vol 77, No.4, pp.541-580, 1989.

[21] Valette, R. "Analysis of Petri Nets by Stepwise Refinements," *J. of Comp. and Syst. Sci.*, 18, pp. 35-46, 1979.

[22] Chen, P., S. C. Bruell, and G. Ballo, "Alternative Methods for Incorporating Non-exponential Distributions into Stochastic Timed Petri Nets," *The third IEEE Int. Workshop an Timed Petri nets and Performance Models,* Tokyo, pp. 187-197, 1989.

[23] Cox, D. R. "A Use of Complex Probabilities in the Theory of Stochastic Process," *Proc. Cambridge Philosophical Society,* 51, pp. 313-319, 1955.

[24] Ho, Y. C. "Performance Evaluation and Perturbation Analysis of Discrete Event Dynamic Systems," *IEEE Trans. on Automatic Control,* 32(7), pp. 563-572, 1987.

[25] Marsan, M. A., G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. "The Effects of Execution Policies on the Semantics and Analysis of Stochastic Petri Nets," *IEEE Trans. on Software Engineering,* Vol. 15, No. 7, pp.832-846, 1986.

[26] Marsan, M. A., G. Balba, and A. Fumagalli. "An Accurate Performance Model of CSMA/CD Bus LAN," *Proc. of 7th European Workshop on Application and Theory of Petri Nets,* Oxford, England, 1986.

[27] Florin, G., and S. Natkin. "Necessary and Sufficient Ergodicity Condition for Open Synchronized Queuing Networks," *IEEE Transactions on Software Engineering,* Vol. 15, No.14, pp. 367-380, 1989.

[28] Dugan, J. B., A. Bobbio, A. Ciardo, and K. S. Trivedi. "The Design of a Unified Package for Solution of Stochastic Petri Net Models," *International Workshop on Timed Petri Nets,* Torino, Italy, pp.6-13, 1985.

[29] Gressier, E. " A Stochastic Petri Net Model for Ethernet," *Proceedings of Int.* Workshop on Timed Petri Nets, Torino, Italy, pp. 296-303, 1985.

[30] Dugan, J. B., A. Bobbio, and A. Ciardo. " Stochastic Petri Net Analysis of A Replicated File System," *Proc. of Int. Workshop on Petri Nets and Performance Models,* Wisconsin, 1987.

[31] Zhou, M. C., and F. DiCesare. "Adaptive Design of Petri Net Controllers for Error Recovery in Automated Manufacturing Systems," *IEEE Trans. on Systems, Man, and Cybernetics,* SMC-19, pp.963-973, 1989.

[32] Viswanadham, N., and Y. Narahari. "Stochastic Petri Net Models for Performance Evaluation of Automated Manufacturing Systems," *Information and Decision Technologies,* 14, pp. 125-142, 1989.

[33] Florin, G., and S. Natkin. "Evaluation Based upon Stochastic Petri Nets of the Maximum Throughput of A Full Dulex Protocol," *Informatic Fachberichte,* C. Girault and W. Reisig(eds.), 1982.

[34] Molloy, M. K. *On the Integration of Delay and Throughput Measure in Distributed Processing Models,* Ph. D. Thesis, UCLA, 1981.

[35] Marsan, M. A., G. Balbo, and G. Conte. "A Class of Generalized Stochastic Petri Nets for Performance Analysis of Multi-processor Systems," *ACM TOCS* Vol. 2, No. (2), pp. 93-122, 1984.

[36] Marsan, M. A., and G. Chiola. "On Petri Nets with Deterministic and Exponentially Distributed Firing Times," *Lecture Notes in Computer Science,* No. 254, pp. 132-145, 1986.

[37] Dugan, J. B., K. S. Trivedi, R. M. Geist, and V. F. Nicola. "Extended Stochastic Petri Nets: Application and Analysis," *Proc. PERFORMANCE 84,* pp. 507-519, Paris, France, 1984.

[38] Howard, R. A. *Dynamic Probabilistic Systems,* Jone Wiley, New York, NY, 1971.

[39] Kleirock, L. *Queueing Systems,* Wiley-Interscience. NY, 1975.

[40] Cumani, A. "ESP - A Package for the Evaluation of Stochastic Petri Nets with Phase-type Distributed Transition Times," *Proc. IEEE Int. Workshop on Timed Petri Nets,* Torino, Italy, 1985.

[41] Leung, Y. T., and R. Suri. "Performance Evaluation of Discrete Manufacturing Systems," *IEEE Control Systems,* 10(4), pp. 77-86, 1990.

[42] Jamwal, A. U. "Software Development for Analysis of Stochastic Petri Nets Using Transfer Function," Master thesis, NJIT, 1991.

[43] Barkakati, N. *X Window System Programming,* SAMS, 1991.

[44] Zhou, M. C., F. DiCesare and A. A. Desrochers (1992). "A Hybrid Methodology for Synthesis of Petri Net models for manufacturing systems," *IEEE Trans. on Robotics & Automation,* 8, pp. 305-361, 1992.

[45] Kernighan, B. W., and R. Pike. *The UNIX Programming Environment,* Prentice-Hall, Inc., 1984.