国 立 交 通 大 學

電 機 與 控 制 工 程 研 究 所

碩 士 論 文

派屈網輔助邏輯電路模型檢查技術之研究

Study on Petri Net-aided Model Checking Techniques

研究生：黃仕捷

指導教授：董蘭榮　博士

中華民國九十六年七月

Study on Petri Net-aided Model Checking Techniques

Advisor: Dr. Lan-Rong Dung

Graduate Student: Shih-Chieh Huang

July 2007

Graduate Institute of Electrical and Control Engineering

National Chiao Tung University

Hsinchu, Taiwan, ROC

Study on Petri Net-aided Model Checking Techniques

Graduate Student: Shih-Chieh Huang　　　　Advisor: Dr. Lan-Rong Dung

Department of Electrical and Control Engineering
National Chiao Tung University

Abstract

With the progress of semiconductor manufacturing techniques and the increasing of complexity of designs, to ensure the correctness of a design becomes a hard mission. To find out the bugs in a large and complex design is time consuming but significant works. The general verification method used by designers is simulation. The designers input appropriate signals to the design and observe if the outputs are correct to judge the correctness of the design. This verification method can not ensure that the design is completely conform to the specification. Clarke and Allen Emerson invented model checking techniques to recover the insufficiency of simulation based verification. In this paper, we propose a Petri net-aided model checking techniques to assist SMV model checker. In some cases, this technique can speed up the verification of **EF** and **EX** properties. We implement a simple program with C++ language to transfer a FSM (finite state machine) into a Petri net and verify the state machine. Then we show some examples to compare the verification time of PNV and SMV. Finally we make a conclusion that in some cases, PNV can reduce the verification time of **EF** and **EX** properties substantially.

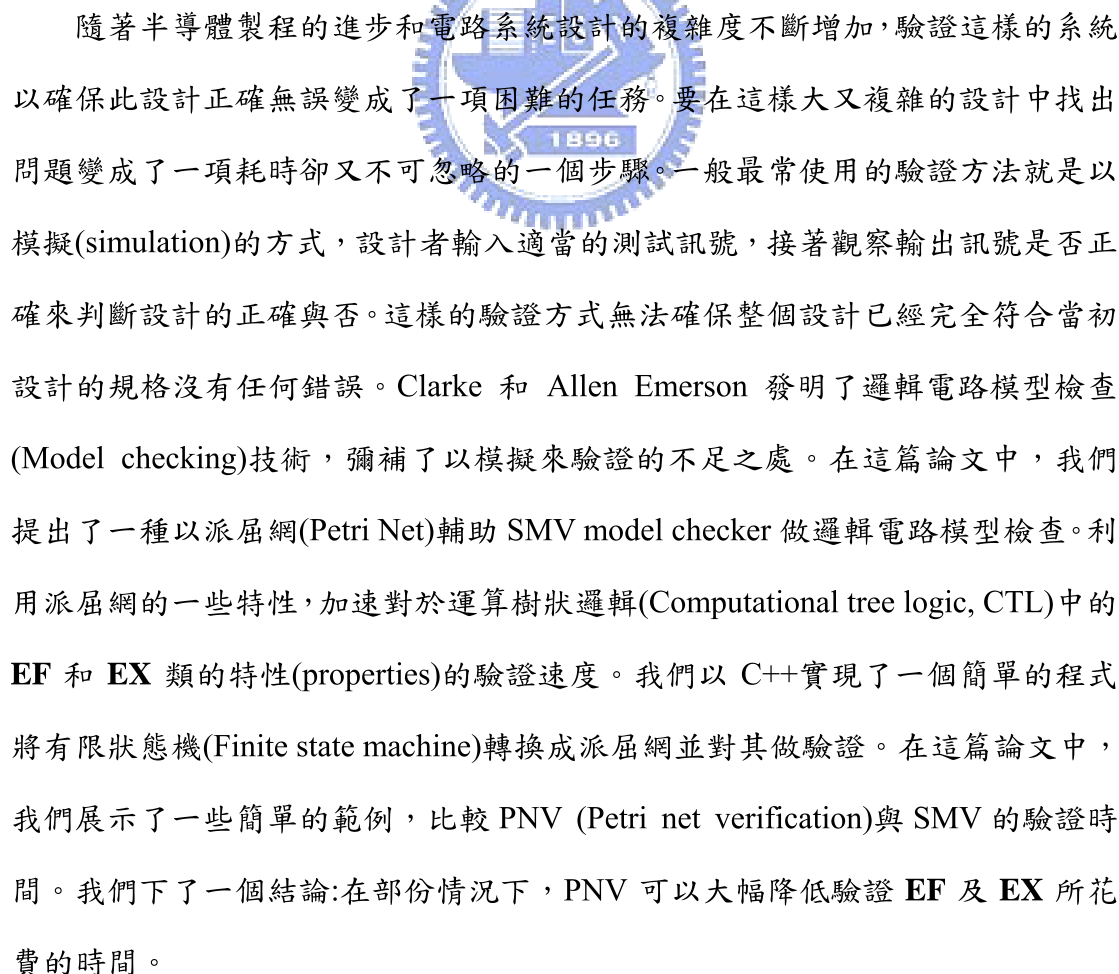
派屈網輔助邏輯電路模型檢查技術之研究

學生：黃仕捷　　　　　　　指導教授：董蘭榮　博士

國立交通大學

電機與控制工程學系研究所

摘要



隨著半導體製程的進步和電路系統設計的複雜度不斷增加，驗證這樣的系統以確保此設計正確無誤變成了一項困難的任務。要在這樣大又複雜的設計中找出問題變成了一項耗時卻又不可忽略的一個步驟。一般最常使用的驗證方法就是以模擬(simulation)的方式，設計者輸入適當的測試訊號，接著觀察輸出訊號是否正確來判斷設計的正確與否。這樣的驗證方式無法確保整個設計已經完全符合當初設計的規格沒有任何錯誤。Clarke 和 Allen Emerson 發明了邏輯電路模型檢查(Model checking)技術，彌補了以模擬來驗證的不足之處。在這篇論文中，我們提出了一種以派屈網(Petri Net)輔助 SMV model checker 做邏輯電路模型檢查。利用派屈網的一些特性，加速對於運算樹狀邏輯(Computational tree logic, CTL)中的 **EF** 和 **EX** 類的特性(properties)的驗證速度。我們以 C++實現了一個簡單的程式將有限狀態機(Finite state machine)轉換成派屈網並對其做驗證。在這篇論文中，我們展示了一些簡單的範例，比較 PNV (Petri net verification)與 SMV 的驗證時間。我們下了一個結論:在部份情況下，PNV 可以大幅降低驗證 **EF** 及 **EX** 所花費的時間。

誌謝

本篇論文得以順利完成，首先要感謝的是我的指導教授——董蘭榮教授。在碩士班的兩年間，董教授不厭其煩地指導我，當我陷入瓶頸及走錯方向時，董教授亦適時地指點我正確的方向，使我得以做出修正，並且提供非常豐富的資源，讓我能好好潛心於學習研究。這兩年的研究生活讓我獲益良多。

同時，也感謝實驗室的學長——江宗錫及楊學之，在我的求學過程中給予指點與幫助，以及同學們——何駿徹、賴信丞、呂俊衛及黃致惟，在課業與生活上的互相扶持、分擔紓解彼此的壓力，另外還有實驗室學弟們——謝博仁、賴貫康、簡嘉宏、莊詠麟，給了我一段美好的研究所時光。

在這段時間裡，我的女朋友筱珊在我身邊陪伴著我，當我煩悶及陷入低潮時，適時的幫我解悶和給我鼓勵，讓我的研究生活更加順利，在此感謝筱珊的陪伴。

最後要感謝我的父母，給我衣食無憂的環境，讓我可以專心於學校的事情，無後顧之憂的完成我的學業。

謹將此論文獻給所有關心我的人，在此致上最深的謝意。

Contents

ABSTRACT(IN ENGLISH) I

ABSTRACT(IN CHINESE)........ II

CONTENTS........ IV

LIST OF TABLES........ VI

LIST OF FIGURES VII

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND........ 4

2.1 MODEL CHECKING 4

2.2 SMV MODEL CHECKER 6

2.3 COMPUTATIONAL TREE LOGIC 7

2.4 PETRI NET 8

2.4.1 Basic Definition 8

2.4.2 Marking, State and Reachability........ 10

2.4.3 Matrix Equations........ 12

CHAPTER 3 MODEL CHECKING WITH PETRI NET 15

3.1 VERIFICATION FLOW 16

3.2 TRANSFORMATION FROM FSM INTO PETRI NET 17

3.2.1 Transformation of Places and Transitions....17

3.2.2 Transformation of Multiple Modules....22

3.3 MARKING GENERATOR....24

3.4 REACHABILITY CHECKING....26

3.4.1 Ranks are Equal to the No. of Transitions....28

3.4.2 Ranks are Less Than the No. of Transitions....30

3.4.3 Summary....33

CHAPTER 4 IMPLEMENTATION....35

4.1 INPUT CODING RULES....35

4.2 DATA STRUCTURE OF PNV....37

4.3 PROPERTY TO MARKING....40

4.4 VERIFICATION CORE....41

4.4.1 Elimination Methods....42

CHAPTER 5 EXPERIMENTAL RESULTS....47

5.1 COUNTER....47

5.2 GREATEST COMMON DIVISOR....51

5.3 AMBA....53

5.4 TRAFFIC LIGHT CONTROLLER....57

5.5 WIZARD'S REGISTRATION FLOW....59

CHAPTER 6 CONCLUSION AND FUTURE WORK....61

6.1 CONCLUSION....61

6.2 FUTURE WORK....62

REFERENCES....63

List of Tables

TABLE 5-1 TIME COST OF SMV AND PNV FOR COUNTER 50

TABLE 5-2 TIME COST OF SMV AND PNV FOR GCD.. 53

TABLE 5-3 TIME COST OF SMV AND PNV FOR AMBA 56

TABLE 5-4 TIME COST OF SMV AND PNV FOR TRAFFIC LIGHT CONTROLLER 59

TABLE 5-5 TIME COST OF SMV AND PNV FOR REGISTRATION 60

List of Figures

Figure 2-1 SMV Graphic User Interface [25]..6

Figure 2-2 CTL illustration..8

Figure 2-3 Petri net 1..10

Figure 2-4 Petri net 1 state 0..11

Figure 2-5 Petri net 1 state 1..11

Figure 3-1 Verification flow..16

Figure 3-2 FSM 1..19

Figure 3-3 Petri net of Figure 3-2..19

Figure 3-4 FSM 2..20

Figure 3-5 Petri net for Figure 3-4..20

Figure 3-6 FSM 4..20

Figure 3-7 Petri net 1..22

Figure 3-8 FSM 3..23

Figure 3-9 Petri net of Figure 3-6..23

Figure 3-10 Petri net combine from Figure 3-8..24

Figure 3-11 Flow of ranks = NO. of transition..29

Figure 3-12 Example of the ranks < NO. of transitions..30

Figure 3-13 Example of illustrating EF properties judging rules..32

Figure 3-14 Flow of ranks < NO. of transitions..33

Figure 3-15 PNV software verification flow..34

FIGURE 4-1 SMV CODE OF COUNTER 36
FIGURE 4-2 DATA STRUCTURE OF PNV 37
FIGURE 4-3 SMV CODE...39
FIGURE 4-4 TRANSITION MATRIX 41
FIGURE 4-5 ELIMINATING EXAMPLE 44
FIGURE 4-6 REDUCED ROW ECHELON FORM..46
FIGURE 5-1 FSM OF COUNTER 47
FIGURE 5-2 PETRI NET OF THE COUNTER 48
FIGURE 5-3 VERIFICATION RESULT OF COUNTER WITH PNV VERBOSE MODE 49
FIGURE 5-4 VERIFICATION RESULT OF COUNTER WITH PNV BRIEF MODE......................50
FIGURE 5-5 VERIFICATION RESULT OF COUNTER WITH SMV 50
FIGURE 5-6 FSM OF GCD..52
FIGURE 5-7 VERIFICATION RESULT OF GCD WITH PNV VERBOSE MODE.......................52
FIGURE 5-8 APB TO ASB READ TIMING (CAPTURED FROM [42])................................54
FIGURE 5-9 ASB TO APB WRITE TIMING (CAPTURED FROM [42]) 54
FIGURE 5-10 FSM OF THE COMMUNICATION OF ASB AND APB...................................55
FIGURE 5-11 VERIFICATION RESULT OF AMBA WITH PNV VERBOSE MODE..................56
FIGURE 5-12 TRAFFIC LIGHT CONTROLLER 57
FIGURE 5-13 FSMS OF TRAFFIC LIGHT CONTROLLER 58
FIGURE 5-14 THE REGISTRATION FLOW OF WIZARD 60

Chapter 1 Introduction

With the advancement of semiconductor manufacturing and complexity increasing of designs, to ensure that designs are correct consumes more and more time and efforts. Nowadays almost 80% of the overall design costs are paid for verification works. In industrial, simulation continues the mainstream for verification topics. However, simulation can only supply the presence of bugs rather than the absence. Formal verification technique has been getting much attention for its 100% design error coverage [2]. By using mathematical model, formal method conducts exhaustive exploration of all possible behaviors of design and proves or disproves the correctness of design intention underlying system specification or properties.

Model checking is a process of checking whether a given model satisfies given properties. The properties are expressed in computational temporal logic (CTL). This technique is a promising formal technique and it has widely used in industry and academy. A number of major companies including Intel, Motorola, ATT, Fujitsu and Siemens have started using model checking technique to verify their actual designs. Model checking allows ensuring that a finite state system does not violate properties it is supposed to conform with [3].

This technique was originally developed in 1981 by Clarke and Allen Emerson.

The main differences between model checking and simulation based verification are

1) Model checking can be performed automatically

2) When model checking detects an error, it produces meaningful results.

Typically, the user provides a high level representation of the model and the properties to be checked. The model checker will either stop with the answer true including that the model satisfies the properties, or give a counterexample to show why the model does not satisfy the properties.

The major method used by traditional model checkers is state traversing [2]. With the increasing of the complexity of designs, states of a system increase dramatically. State traversing method would cost more and more time to verify designs. In this paper, we provide a verification method based on modeling a system with Petri net to speed up the verification of **EF** and **EX** properties.

There are several ways to analyze a Petri net, and matrix equation is one of them [1]. In this paper we use the matrix equation method to analyze Petri nets. Because that the matrix equation method could solve the reachability issues of Petri net through linear algebra analysis [1, 34] and decrease the complexity of verification. Because of the feature of the matrix equation of a Petri net, we can use this method to speed up the verification of **EF** and **EX** properties.

First of all, we transfer the inputted finite state machine into a Petri net. According to the information of the Petri net, we generate the transition matrix of the Petri net. Then we put the properties into marking generator to transfer **EF** and **EX** properties into corresponding markings. If there are some properties which are not **EF** or **EX**, we may pass them to SMV directly. After getting transition matrix and markings, we start to verify the properties. If some properties are verified to be false or we can not process the properties, we will pass them to SMV. The main purpose for us to propose this thesis is to assist SMV to speed up verifying **EF** and **EX**

properties. If we meet the properties we can not process, we still pass them to SMV to verify.

The detailed verification methods are shown in chapter 3. In chapter 4, we illustrate the implementation of the verification software which is programming with C++ language. In chapter 5, there are some examples verified with the software we implemented. In chapter 6, we make some conclusions on this research works.

Chapter 2 Background

2.1 Model Checking

Model checking is an automatic technique for verifying correctness properties of finite-state reactive systems. This technique has been successfully applied to find out subtle errors in complicated industrial designs such as sequential circuits, communication protocols and digital controllers [3].

A reactive system consists of several components which are designed to interact with one another and with the system's environment. In contrast to functional systems, in which the semantics is given as a function from input to output values, a reactive system is specified by its temporal properties. A temporal property is a set of desired behaviors in time; the system satisfies the property if each execution of the system belongs to the set. From a logical standpoint, the system is described by a semantic Kripke-model, and a property is described by a logical formula. Arguing about system correctness, thus, amounts to determining the truth of formulas in models [3].

In order to perform such verification, one needs a modeling language in which the system can be characterized, a specification language for the formulation of properties, and a deductive calculus or algorithm for the verification process.

Usually, the system to be verified is modeled as a finite state transition graph, and the properties are formulated in an appropriate propositional temporal logic. An efficient search procedure is then used to determine whether or not the state transition graph satisfies the temporal formulas. When model checking was first developed in 1981, it was only possible to handle concurrent systems with a few thousand states. The discovery of how to represent transition relations using ordered binary decision diagrams (OBDD) changed the possibility of verifying systems with realistic complexity dramatically. By converting a formula to a BDD, a very concise representation of the transition relation may be obtained [3, 28].

Much of the success of model checking is due to the fact that it is fully automatic verification method. With model checking, all the user has to provide is a model of the system and a formulation of the properties to be proven. The verification tool will either terminate with an answer indicating that the model satisfies the formula or show why the formula fails to hold in the model. These counterexamples are particularly helpful in locating errors in the model or system [3, 26].

With the completely automatic approach it may be necessary for the model checking algorithm to traverse all reachable states of the system. This is only possible if the state space is finite. Whereas other automated deduction methods may be able to handle some infinite-state problems, model checking usually is constrained to a finite abstraction. In fact, model checking algorithms can be regarded as decision procedures for temporal properties of finite-state reactive systems [3, 26, 28].

2.2 SMV Model Checker

SMV is a symbolic model checking tool developed by Cadence Berkeley Labs. It allows users to formally verify temporal logic properties of finite state systems. We use the SMV language to describe the finite state systems which we want to verify by SMV model checker. Figure 2-1(Captured from Cadence SMV) is the Graphic user interface of Cadence SMV model checker [25].

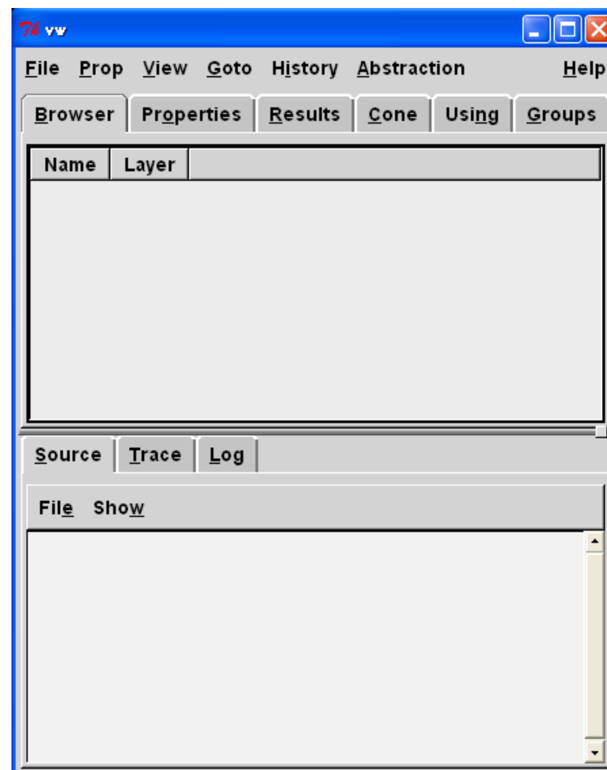


Figure 2-1 SMV Graphic User Interface [25]

The SMV language can be divided roughly into three parts – the definitional part, the structural part, and the expressions. The definitional part declares signals and their relationship with each other. It includes type declarations and assignments. The structural part combines the components declared in definitional part. It

provides language constructs for defining modules and structured data types and for instantiating them. Finally, the expression part specifies the properties that the user wants to verify, and the expressions are expressed by computational tree logic (CTL) [23, 24, 25, 33].

2.3 Computational Tree Logic

Computational Tree Logic (CTL) is branching-time logic: its formulas allow for specifying properties that take into account the non-deterministic, branching evolution of a FSM. The evolution of a FSM from a given state can be described as an infinite tree, where the nodes are the states of the FSM. The paths in the tree start at a given state are the possible alternative evolution of the FSM from that state. The CTL formulas are constructed path qualifiers and temporal operators.

Path qualifiers:

- **A** – "for all the paths"
- **E** – "some of the paths"

Temporal operators:

- **X***p* – "*p* holds *next* time"
- **F***p* – "*p* holds sometime in the *future*"
- **G***p* – "*p* holds *globally* in the future"
- *p***U***q* – "*p* holds *until* *q* holds"

There are eight CTL operators (**AX**, **AF**, **AG**, **A***p***U***q*, **EX**, **EF**, **EG**, **E***p***U***q*) can be used to express properties. They are illustrated in Figure 2-2.

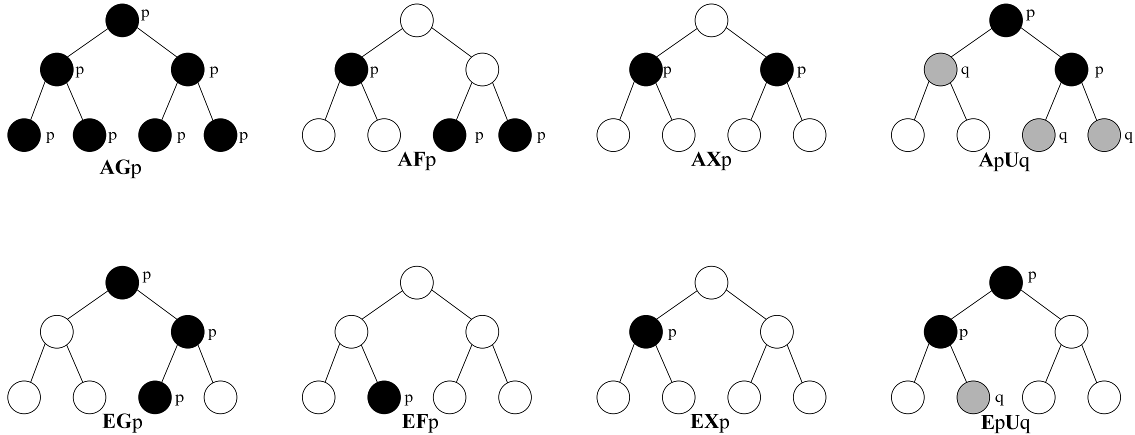


Figure 2-2 CTL illustration

CTL operators can be nested in an arbitrary way and can be combined using logic operators (!, &, |, ->, …). For example, **AG** (*p* -> **EX** *q*) means that "each occurrence of condition *p* is followed by at least one path that condition *q* occurs in the next state." **AG** (*p* & *q* -> **AF** *r*) means that "for all occurrence of condition *p* and condition *q* are followed by condition *r* occurs at one state for each path finally." [2, 4, 20]

2.4 Petri Net

2.4.1 Basic Definition

A Petri net *C* is a four-tuple *C* = (*P, T, I, O*), where *P* is a set of places, *T* is a set of transitions, *I* is an input function, and *O* is an output function. $P = \{ p_0, p_1, p_2, \ldots, p_n \}$ is a finite set of places, where n≧0. $T = \{ t_0, t_1, t_2, \ldots, t_n \}$ is a finite set of transitions, where n≧0. The two sets *P* and *T* are disjoint, that is, $P \cap T = \varphi$. The input function *I* and the output function *O* record the relationship between places and

transitions. For a transition t_i, $I(t_i)$ represents the set of the input places of t_i and $O(t_i)$ represents the set of output places of t_i. In other words, the input function I is a mapping from a transition t_i to a collection of places $I(t_i)$, and the output function O is a mapping from a transition t_i to a collection of places $O(t_i)$. A place p_i is an input place of a transition t_j if $p_i \in I(t_j)$; p_i is an output place of t_j if $p_i \in O(t_j)$.

A transition could have more than one input places and more than one output places. A transition also could have no input places or no output places. A transition which has no input places means that the transition is always firable and usually used to represent inputs. A transition which has no output places means that firing the transition only eliminates the tokens in its input places and no tokens will be created in any place.

A graphical representation of a Petri net is more convenient for illustrating the concepts of Petri net theory and easier for understanding. A Petri net graph consists of two elements, places and transitions. In a Petri net graph, a place is represented with a circle ○, and a transition is represented with a rectangle ▌. Places and transitions in a Petri net are connected with arrows. An arrow directed from a place to a transition defines the place to be an input place of the transition. An arrow directed from a transition to a place defines the place to be an output place of the transition. Multiple inputs to a transition are expressed by multiple arrows from the input places to the transition. Multiple outputs of a transition are represented by multiple arrows from the transition to multiple output places. However, a place can not connect to a place and a transition can not connect to a transition either. Figure 2-3 is an example of a Petri net graph.

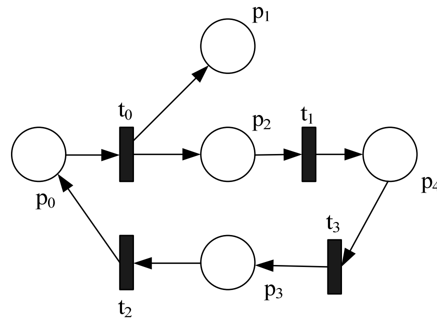


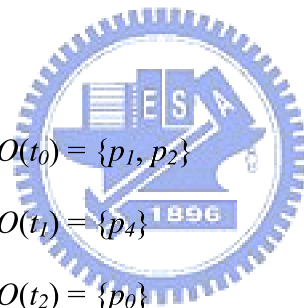
Figure 2-3 Petri net 1

There are five circles and four rectangles in Figure 2-3. The circles and rectangles represent places and transitions respectively. The following is the four-tuple representation of the Petri net graph in Figure 2-3.

$C = (P, T, I, O)$

$P = \{p_0, p_1, p_2, p_3, p_4\}$

$T = \{t_0, t_1, t_2, t_3\}$



$I(t_0) = \{p_0\}$ $O(t_0) = \{p_1, p_2\}$

$I(t_1) = \{p_2\}$ $O(t_1) = \{p_4\}$

$I(t_2) = \{p_3\}$ $O(t_2) = \{p_0\}$

$I(t_3) = \{p_4\}$ $O(t_3) = \{p_3\}$

2.4.2 Marking, State and Reachability

2.4.2.1 Marking

A marking μ is an assignment of tokens to the places of a Petri net. Tokens in a Petri net graph are represented with dots • in the circles which represent places. The states of a Petri net are determined by the number and distribution of tokens in the Petri net. A Petri net executes by firing transitions. During the execution of a Petri net, the number and distribution of tokens may be changed. A transition can be

fired if each of its input places has at least as many tokens in it as arrows from the place to the transition. When a transition has this condition, we say this transition is enabled (firable). The tokens in the input places of a transition and make the transition enable are the enabling tokens of the transition. Firing a transition will remove the enabling tokens from its input places and depose into each of its output places one token for each arrow from the transition to the places. For example, in Figure 2-4 the tokens in p_0 and p_3 are the enabling tokens of t_0 and t_2 respectively. After firing t_0, the tokens in p_0 will be removed and t_0 will depose one token into each of its output places p_1 and p_2. The result of firing t_0 is shown in Figure 2-5.

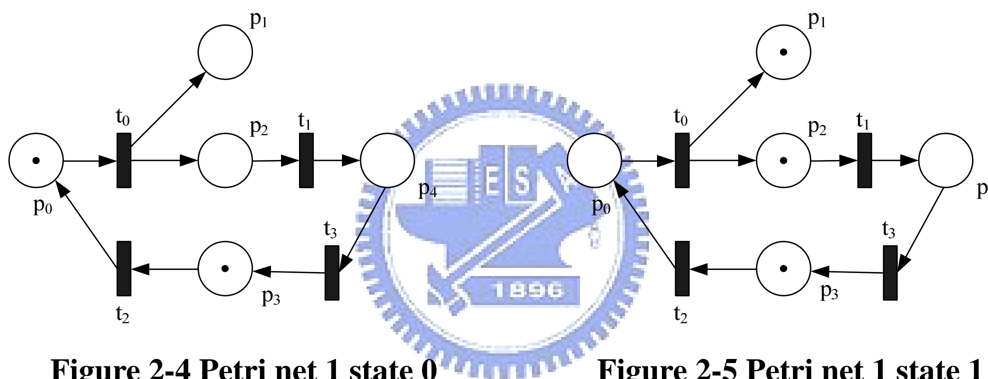


Figure 2-4 Petri net 1 state 0 **Figure 2-5 Petri net 1 state 1**

2.4.2.2 State

The states of a Petri net are defined by markings. The firing of a transition indicates a change in the state of the Petri net by changing the marking of the net. Since only enabled transitions can fire, the number of tokens in each place always remains non-negative when a transition is fired. The change in states caused by firing a transition is defined by a change function δ called the next-state function. If a transition t_i is enabled, $\delta(\mu, t_i) = \mu'$ indicates that firing t_i will change the marking from μ to μ'. When the marking of a Petri net is changed, the state of the Petri net is changed. For example, in Figure 2-4, the marking $\mu = [1\ 0\ 0\ 1\ 0]^T$.

After firing t_0, the marking will change from μ to $\mu' = [0\ 1\ 1\ 1\ 0]^T$.

2.4.2.3 Reachability

Given a Petri net $C = (P, T, I, O)$ and an initial marking μ^0, we can execute the Petri net by sequential transition firings. Firing an enabled transition t_i at the initial marking makes a new marking $\mu^1 = \delta(\mu^0, t_i)$. At the new marking μ^1, we can fire any enabled transitions to get a new marking. Assume that t_j is an enabled transition at μ^1, firing t_j will get a new marking $\mu^2 = \delta(\mu^1, t_j)$. This action can continue as long as there is at least one enabled transition at the new marking. If the Petri net reaches a marking in which no enabled transition, in other words, there is no transition can be fired, the execution of this Petri net must stop.

For a Petri net $C = (P, T, I, O)$ with marking μ, a marking μ^1 is immediately reachable from μ if there exists a transition $t_i \in T$ such that $\delta(\mu, t_i) = \mu^1$. If a marking μ^2 is reachable from μ^1 immediately, then we say that μ^1 and μ^2 are reachable from μ^0. In [1], they define the reachability set $R(C, \mu)$ of a Petri net C with marking μ to be all markings which are reachable from μ. A marking μ' is in $R(C, \mu)$ if there is at least one sequence of transition firings which will change the marking from μ to μ'. For example, the initial marking in Figure 2-4 is $\mu^0 = [1\ 0\ 0\ 1\ 0]^T$, after firing the sequence of transitions (t_0, t_1), we get $\mu^1 = [0\ 1\ 1\ 1\ 0]^T$ and $\mu^2 = [0\ 1\ 0\ 0\ 1]^T$. We say that $\mu^1, \mu^2 \in R(C, \mu^0)$.

2.4.3 Matrix Equations

There are several approaches to analyze a Petri net, and one of them is based on matrix equations. Each Petri net could be represented by a transition matrix. The matrix indicates the relationship between places and transitions. A transition matrix

is m (the number of places) rows by n (the number of transitions) columns. Each column represents one transition and records the input places and output places of this transition. The way of recording the relationship of places and transitions is that if there is an arrow directed from a place p_i to a transition t_j, we put -1 in the ith row by the jth column to mark that p_i is an input of t_j. If there are two arrows directed from p_i to t_j, then we put -2 in the corresponding position in the matrix and so on. If an arrow directed from t_j to p_k, we put 1 in the kth row by the jth column to mark that p_k is an output of t_j. If there are two arrows directed from t_j to p_k, we put 2 in the corresponding position in the matrix. After establishing the transition matrix of a Petri net, we can represent the Petri net by a matrix equation like the following.

$$\mu' = \mu + \mathbf{Ax} \tag{2.1}$$

Here μ is the initial state(marking) of the Petri net; **A** is the transition matrix records the relationship between places and transitions of the Petri net; **x** is a transition firing sequence matrix which records the firing times without firing order information of all transitions in the Petri net; μ' is the state(marking) of the Petri net after firing the transitions recorded in **x**. Of course, these fired transitions should be enabled at the state μ. This matrix equation indicates that we can get the next state μ' by adding the product of **A** and **x** to μ. Take Figure 2-4 for example,

$$\mu = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{matrix} \qquad \mathbf{A} = \begin{matrix} & \begin{matrix} t_0 & t_1 & t_2 & t_3 \end{matrix} \\ & \begin{bmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 \end{bmatrix} \end{matrix} \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{matrix}$$

At the state μ, the enabled transitions are t_0 and t_2. After t_0 fired, as shown in

Figure 2-5, we can get a new marking by solving the matrix equation

$$\mu' = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

To solve the matrix equation, we obtain $\mu' = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$. If we set $\mathbf{x} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, we can get another marking $\mu' = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ which represents the state firing t_0 and t_2 from the initial state μ [1].

Chapter 3 Model Checking with Petri Net

For the traditional model checking we told in section 2.1, the verification method is to spread the states from the initial state and then traverse all states to check that if all paths satisfy the properties which the system needs to hold. The drawback of the traditional model checking method is when designs become more and more large and complex, the states need to be traverse will grow dramatically. Because of this, the users need to spend more time verifying the designs through state traversing and the time will become considerable. In order to assist the state traversing based model checking, we propose a verification method based on Petri net. Because of the features of Petri net, the verification method we proposed can reduce the complexity of verifying **EX** and **EF** properties and economize on the time users spend verifying.

The method we used is to model a FSM in a Petri net and utilize the features of Petri net to reduce the complexity of verification. In this chapter, we will introduce the verification flow and the methods of our verification (Petri Net Verification, abbreviate to PNV).

3.1 Verification Flow

The verification flow of our Petri net based model checking is shown in Figure 3-1. The input of PNV is a SMV code with some constrain we made. The SMV code includes the description of FSMs and properties. The FSM parts include signals (inputs and outputs), states and state transition information. According to the information, we generate transition matrix and markings and verify the system.

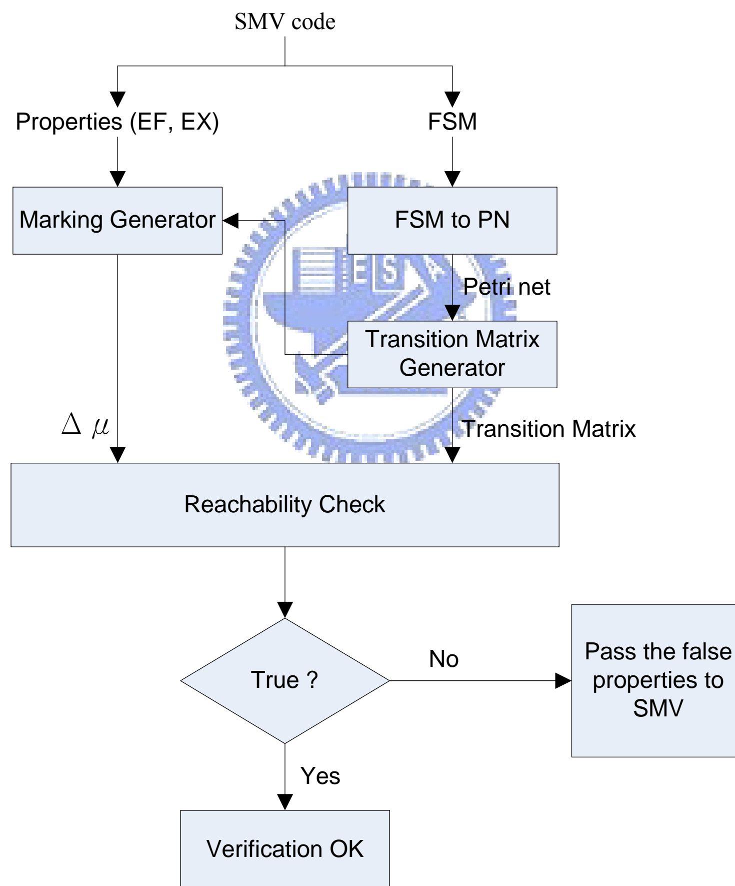


Figure 3-1 Verification flow

When we get a FSM and its properties, first, we transfer the FSM into a corresponding Petri net. Second, we generate the transition matrix based on the Petri net we get before by Transition Matrix Generator. Third, put the properties and

some data from the transition matrix into Marking Generator, we get corresponding markings. After getting the transition matrix and markings, we can start to verify the system. Because of the limitation of the verification methods, we can only process **EX** and **EF** properties. After verifying, if a property is false, we will pass the property to SMV to verify it again and generate a counterexample.

The verification flow is separated into several parts. In the following sections, we will talk about each part step by step. In the first part, we will talk about the transformation from FSM into Petri net. The generation method of transition matrix we have told in section 2.4.3, so we do not talk about it in this chapter. Second, we will discuss how the marking generator generates the corresponding markings based on the properties. Finally, we will introduce the main verification method of Petri net based model checking and the limitation of this method.

3.2 Transformation from FSM into Petri Net

3.2.1 Transformation of Places and Transitions

To transform a FSM into a Petri net, we need to know the relation between a FSM and a Petri net. A FSM is composed of the following five parts:

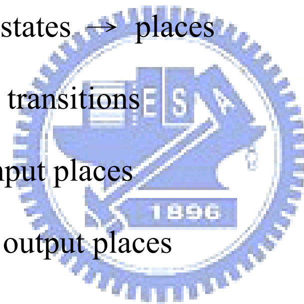
- Q : States
- Σ : Input alphabet
- Δ : Output alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ next-state function
- $\Gamma : Q \rightarrow \Delta$ output function

A Petri net is a four-tuple (*P, T, I, O*). As a FSM $(Q, \Sigma, \Delta, \delta, \Gamma)$ we define a Petri net (*P, T, I, O*) by

- $P = Q \cup \Sigma \cup \Delta$
- $T = \{t_{q,\sigma} | q \in Q \text{ and } \sigma \in \Sigma\}$
- $I(t_{q,\sigma}) = \{q, \sigma\}$
- $O(t_{q,\sigma}) = \{\Gamma(q)\}$

According to the illustration above, we could know that all states, inputs, and outputs in a finite state machine are represented with places in a Petri net and all events (arrows) in a state machine are represented by transitions in a Petri net [1]. The rules of transferring a FSM into a Petri net are:

1. Inputs, outputs, and states → places
2. Events (arrows) → transitions
3. Preconditions → input places
4. Post-conditions → output places



The arrows determine the state transitions in a FSM as well as the transitions in a Petri net, so an arrow in a FSM would be transferred into a transition in a Petri net. The input places and output places of the transition are the preconditions and post conditions of the arrow in the FSM respectively.

After discussing the transformation rules of FSMs and Petri nets, the following are some examples to show the transformation in different situations.

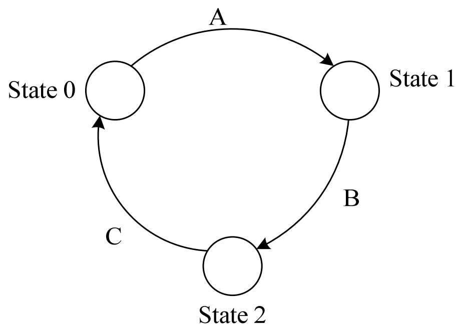


Figure 3-2 FSM 1

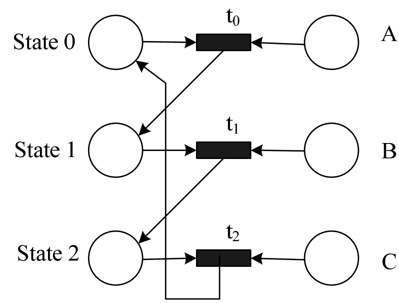


Figure 3-3 Petri net of Figure 3-2

Figure 3-3 is the Petri net transferred from the state machine in Figure 3-2. In Figure 3-2, there are three states (State 0, State 1, and State 2) and three signals (A, B, C). According to the transformation rule 1, there should be three places represent the states and another three places represent the control signals in the corresponding Petri net. There are three arrows in the state machine to determine the state transitions. According to the transformation rule 2, the Petri net needs to have three transitions to represent the three arrows, there are t_0, t_1, and t_2. Finally we connect the places and transitions bottom on the transformation rule 3 and 4 then the Petri net is built.

In the above example, we told about the state machine which changes states when control signals are 1(i.e. there is only one value being used for each signal). So there is only one place needs to be built to represent each signal in Petri net. In some state machines, states may change when the control signal is 0 and 1 both. In this kind of situation, we should generate two places to represent the two values of the signal. It is illustrated in the following example.

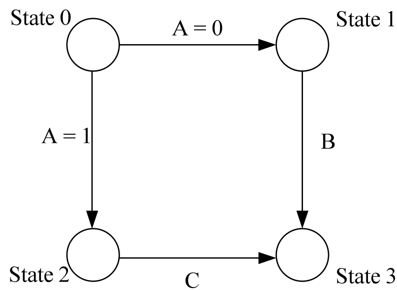


Figure 3-4 FSM 2

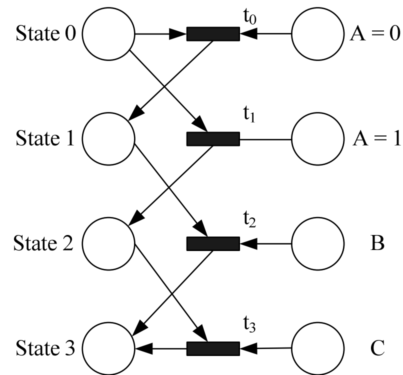


Figure 3-5 Petri net for Figure 3-4

In the FSM in Figure 3-4, State 0 may change to State 1 when A = 0 and may change to State 2 when A = 1. To transfer the FSM into a Petri net in this situation we should create two places to represent the two values of the signal A. The corresponding Petri net is shown in Figure 3-5. The transition t_0 controls the state transition from State 0 to State 1, and t_1 controls the state transition from State 0 to State 2.

In some cases, we could ignore to transfer some arrows and some values of signals into transitions and places. It is that when a value of a signal which does not control any state transitions, we could ignore the transformation of the value and the arrow. For example, in Figure 3-6, State 0 has a self-loop when A = 0. Assume the initial state is State 0, and A = 0, the state is still at State 0 until A = 1. This FSM could have the same behavior even if we ignore the transformation of the arrow controlled by signal A = 0. Because A = 0 does not determine any state transitions.

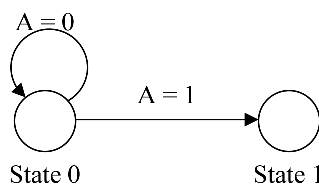


Figure 3-6 FSM 4

But in some other case, we still generate transitions and places even if the arrows and signals do not determine any state transitions. An example is shown in section 5.2.

From the three examples above, we can observe that in some situations we transfer one signal into one place, but in other situations we transfer one signal into two places. Theoretically speaking, we should always transfer a signal in a FSM into places based on its values. In other words, when a signal A can be 0 and 1, we should generate two places to represent A = 0 and A = 1. But some values of signals do not control any state transitions. For example, the value 0 of the signal A in Figure 3-2 do not control any state changes in the FSM, we don't need to generate a place to represent A = 0 and the transition controlled by A = 0 either. It is no effect that we do not generate the place A = 0 and the transition in the corresponding Petri net. Summarily speaking, when there are any arrows which do not determine state transitions, we could ignore the transformations of the arrows. We only transfer the signals and arrows which are used into places and transitions respectively. By doing this, we could make the corresponding Petri net more concise and the transition matrix of the Petri net smaller.

There is a special kind of transitions which do not have input places. The transitions are firable all the time and usually connected in front of inputs to provide the input places tokens. When we meet a situation that we don't know how many tokens we have to assign to an input signal (e.g. clk), we put an input transition in front of the place to generate tokens to the signal.

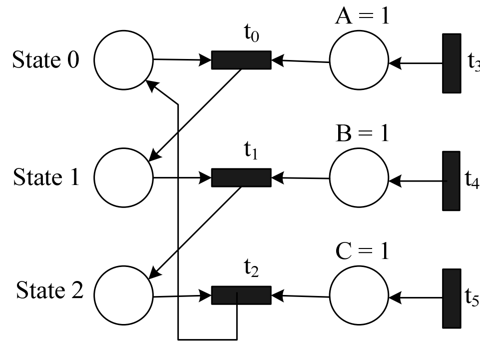
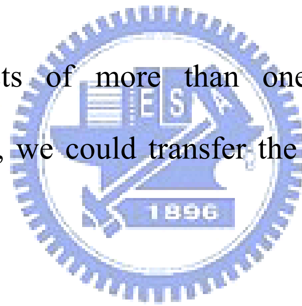


Figure 3-7 Petri net 1

In Figure 3-7, t_3 to t_5 are input transitions, they provide tokens to the inputs A, B and C respectively.

3.2.2 Transformation of Multiple Modules



When a system consists of more than one module and these modules communicate with each other, we could transfer the modules into a Petri net as the following steps.

1. Transfer each module into a Petri net individually
2. Merge the same places which are used in different Petri nets.

In Figure 3-8, it is a simple example to model bus communication protocol. FSM A is master and FSM B is slave. This system models the behavior of reading and writing. The initial states of the two state machines are A0 and B0.

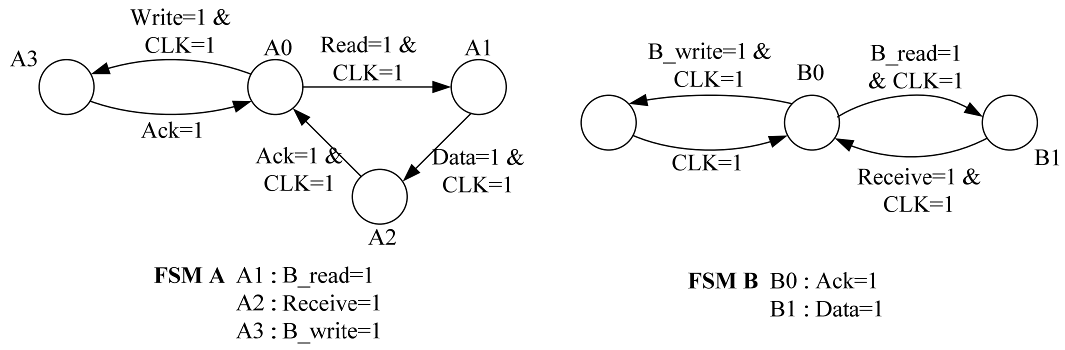


Figure 3-8 FSM 3

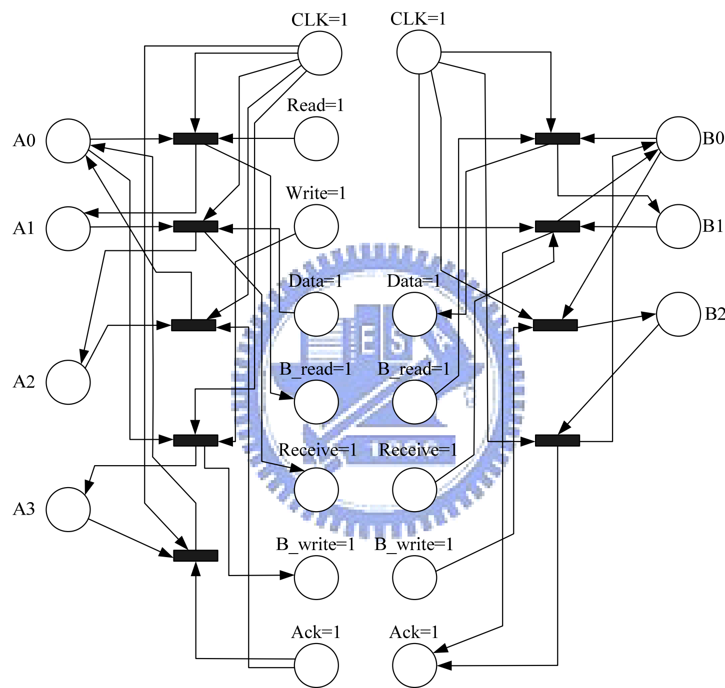


Figure 3-9 Petri net of Figure 3-6

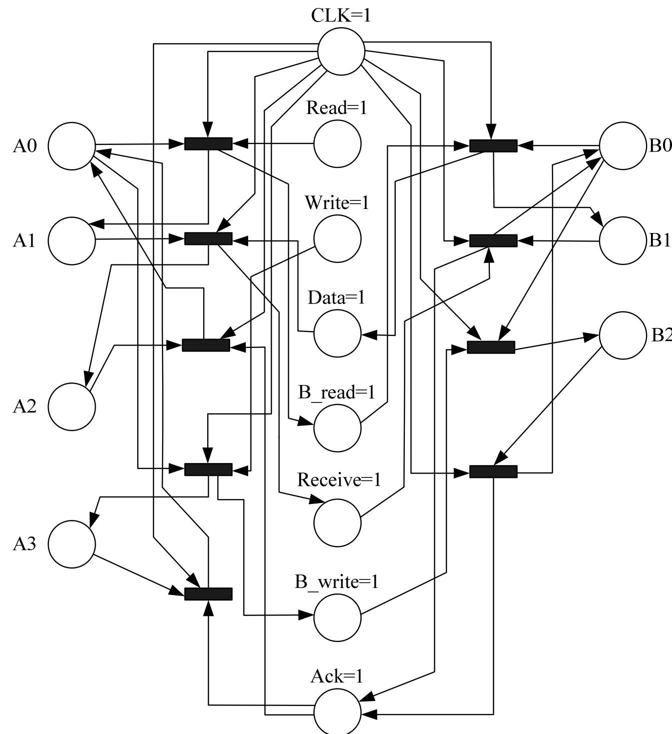


Figure 3-10 Petri net combine from Figure 3-8

According to the rules we told above, the first step is to transfer the two state machines into Petri nets individually. The transformation result is shown in Figure 3-9. The left Petri net is transferred from FSM A and the right one is transferred from FSM B. Because of the cooperation of these two FSMs, there are many places which appear in both Petri nets. Then we merge the same places of the two Petri nets. After merging some places of the two Petri nets, we get one Petri net shown in Figure 3-10. This Petri net is the result transferred from the system in Figure 3-8.

3.3 Marking Generator

The function of marking generator is to transfer the properties into corresponding markings. Markings represent the states of Petri nets. A property is written as

"O_1 (starting state → O_2 (ending state))", O_1 and O_2 are CTL operators. The starting state and ending state here can be mapping to μ and μ' in equation 2.1 respectively. To transfer a state into a marking, we put one token in each place which represents the state or signal mentioned in the state description. The expression of a token in matrix equation of a Petri net is putting 1 in the position which represents the place. In order to make the verification process convenient, we rewrite equation 2.1 as

$$\Delta \mu = \mathbf{Ax} \qquad (3.1)$$

where $\Delta \mu = \mu' - \mu$. The actual information we need is $\Delta \mu$. So, the rules of transferring the starting state and the ending state into $\Delta \mu$ are

1. In a marking matrix (μ), we put 1 in each place which represents the state mentioned in the starting state.
2. In another marking matrix (μ'), we put 1 in each place which represents the state mentioned in the ending state.
3. Through subtracting μ from μ', we get $\Delta \mu$.

To simplify the transformation steps, we change the way we do above. First we fill the matrix $\Delta \mu$ with 0. Then we add -1 instead of 1 in each place which represents the state mentioned in the starting state in a marking matrix and add 1 in each place which represents the state mentioned in the ending state in the same matrix. By doing this, we can directly get the matrix $\Delta \mu$.

3.4 Reachability Checking

We have introduced in section 2.4.3 that a Petri net could be analyzed by a matrix equation and the equation could be express as equation 3.1. According to linear algebra theorem [34], a linear system **Ax** = **b** is consistent if and only if the rank of **A** is equal to the rank of (**A** | **b**), where (**A** | **b**) is the augmented matrix. For example, from Figure 2-4 and 2-5 we can get

$$\mathbf{A}=\begin{bmatrix}-1&0&1&0\\1&0&0&0\\1&-1&0&0\\0&0&-1&1\\0&1&0&-1\end{bmatrix}\quad \Delta\mu=\begin{bmatrix}0\\1\\1\\1\\0\end{bmatrix}-\begin{bmatrix}1\\0\\0\\1\\0\end{bmatrix}=\begin{bmatrix}-1\\1\\1\\0\\0\end{bmatrix}\quad (\mathbf{A}\,|\,\Delta\mu)=\left[\begin{array}{cccc|c}-1&0&1&0&-1\\1&0&0&0&1\\1&-1&0&0&1\\0&0&-1&1&0\\0&1&0&-1&0\end{array}\right]$$

If the state in Figure 2-5 is reached from the state in Figure 2-4, the rank of **A** will be the same as the rank of matrix ($\mathbf{A}\,|\,\Delta\mu$). To evaluate the ranks of **A** and ($\mathbf{A}\,|\,\Delta\mu$), we use Gaussian elimination [34] to reduce the two matrices into reduced row echelon form and normalize the lead variables of each row. After going through the steps, we get two matrices in row echelon form like the following.

$$\text{Reduced }\ \mathbf{A}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\\0&0&0&0\end{bmatrix}\ \text{ and reduced }\ (\mathbf{A}\,|\,\Delta\mu)=\left[\begin{array}{cccc|c}1&0&0&0&1\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&0\end{array}\right]$$

The rank of **A** = 4 and the rank of ($\mathbf{A}\,|\,\Delta\mu$) = 4. This linear system is consistent and we can get solutions of the transitions directly from the most right column. The solution of the system is $\mathbf{x}=\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$. This solution means that we can reach the state of the Petri net in Figure 2-5 from the state in Figure 2-4 by firing the transition t_0 one

time.

Through this linear algebra method above, we can only know that this system has a solution or not. This condition can make us apply it to check **EX** and **EF** properties. Because as long as we can find any situation that makes the system satisfies the property which is **EX** or **EF**, we can say that the system satisfies the property. The other kinds of properties (**AX**, **AF**, **AG**…) may not have many advantages to be verified by this method because that to verify a system which satisfies these kinds of properties (**AX**, **AF**, **AG**…) needs to be proved that more than one situation or all situations of the system which satisfy the properties. So the main point in this paper is the methods to speed up verifying **EF** and **EX** properties with modeling a FSM in Petri net.

Before we start to solve the solutions of a matrix equation, we need to know the ranks of the matrices. When the ranks of **A** and (**A** $|\Delta\mu$) are equal, we can only know that the linear system has solutions. When the ranks of the matrices are equal to the number of its transitions, we can get a unique solution of the system. When the rank of the matrix is less than the number of its transitions, there are infinite solutions of the equation.

The solution of a matrix equation is a set of the firing times of transitions in a Petri net, and the firing times must be integers and greater than 0. Proving the equation is consistent is not enough to confirm that the system satisfies the property. We need to do some other process to ensure that there is at least one solution exists and satisfies the following two rules (Firing Times Rules).

1. The values of the firing times should be equal or greater than 0.
2. All the firing times should be integers.

In other words, when the ranks are equal to the number of transitions, we have to solve the solution and check if the solution conform the Firing Times Rules; when the ranks are less than the number of transitions, we also need to ensure that there exists at least one solution satisfies Firing Times Rules.

3.4.1 Ranks are Equal to the No. of Transitions

When the ranks of a matrix equation are equal to the number of its transitions, we can use the method we told above to verify the linear system and confirm that this system has at least one solution then check if the solution satisfies Firing Times Rules.

To verify an **EF** property we only need to solve the solution and check if the solution satisfies the Firing Times Rules. If the answer is YES and we can say that the **EF** property is true.

To verify an **EX** property we not only need to solve the solution and check if the solution satisfies the Firing Times Rules but also need to check one more rule which is:

- All fired transitions should not have any correlation with each other.

Proof:

1. For a FSM, there is only one state which the state machine is staying at. So in a Petri net transferred from a FSM, there is only one place which represents one of the states with a token in it.
2. For each transition, one of its input places must be a state place. In other words, firing a transition must cause a state transition.

Because of the two reasons, for a Petri net transferred from only one state machine, when we solve the firing sequence which has more than one transition

fired, the fired transitions must have causal relation with each other. For a Petri net transferred from more than one FSM, the Petri net can fire more than one transition at one time. But the fired transitions should not have casual relation with each other, too. When a firing sequence in which the fired transitions have casual relation with each other, the final state must be not the next state of the initial state.

The method of checking the rule is that the output places of each fired transition should not be other fired transition's input places. When the fired transitions do not have any casual relation with each other, we can ensure that the final state is the next state of the initial state. If there is a fired transition its input places are other fired transition's output places, it means that there is causal relationship between these two transitions. State transition causing by firing two successive transitions may not satisfy **EX** properties, i.e. the ending state is not the next state of the beginning state. The verification flow of ranks = NO. of transitions is shown in Figure 3-11.

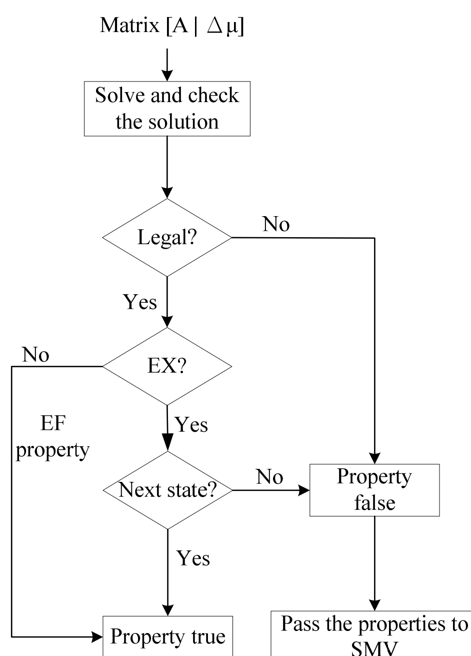
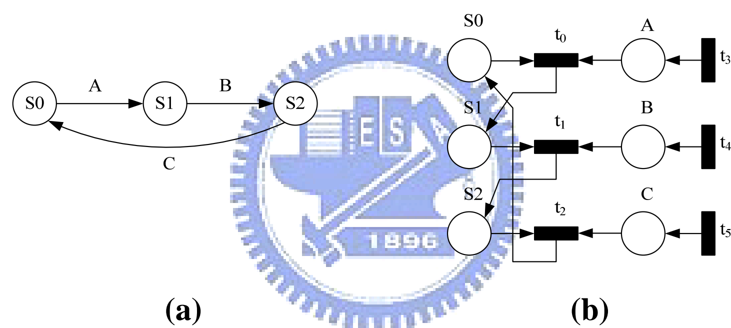


Figure 3-11 Flow of ranks = NO. of transition

3.4.2 Ranks are Less Than the No. of Transitions

The cause of that the ranks are less than the number of transitions is that the FSM has loops and when we transfer the state machine into a Petri net we generate transitions without input places and the output places are the control signals. For example, in Figure 3-12(a), the state machine has a loop and three control signals. We transfer the state machine into a Petri net in figure 3-12(b) and generate three transitions (t_3, t_4, t_5) without input places to put tokens into signal places.



	t0	t1	t2	t3	t4	t5	Δ μ
S0	-1	0	1	0	0	0	-1
S1	1	-1	0	0	0	0	0
S2	0	1	-1	0	0	0	1
A	-1	0	0	1	0	0	0
B	0	-1	0	0	1	0	0
C	0	0	-1	0	0	1	0

(c)

t0	t1	t2	t3	t4	t5	Δ μ
1	0	0	0	0	-1	1
0	1	0	0	0	-1	1
0	0	1	0	0	-1	0
0	0	0	1	0	-1	1
0	0	0	0	1	-1	1
0	0	0	0	0	0	0

(d)

Figure 3-12 Example of the ranks < NO. of transitions

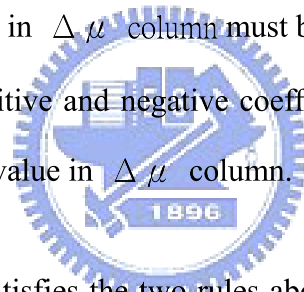
In this example, we set a property: "**AG** (S0 -> **EF** (S2))" to the FSM. The reduced matrix is shown in Figure 3-12(d). In this kind situation, the ranks will be less than the number of transitions and we can adjust some transitions to make the Petri net go through the loop one time of more. In this example we can adjust the firing times of t_5. In Figure 3-12(d), if we set $t_5 = 0$, the solution of the matrix equation will be (1, 1,

0, 1, 1, 0) and the state transition process is S0 → S1 → S2; if we set t_5 = 1, the solution will be (2, 2, 1, 2, 2, 1) and the state transition process is S0 → S1 → S2 → S0 → S1 → S2.

According to the illustration above, when the ranks of a matrix equation are less than the number of its transitions, we have to use some other methods to confirm that there is at least one solution satisfies the properties and the Firing Times Rules.

For **EF** properties, first we eliminate the matrix (**A** |$\Delta\mu$) into reduced row echelon form. Then we check each coefficient in each row to comprehend if the matrix obeys the following two rules.

1. For each row, when the signs of the coefficients in the **A** part are the same, the sign of the value in $\Delta\mu$ column must be also the same as the signs.
2. When there are positive and negative coefficients in the **A** part, we do not care the sign of the value in $\Delta\mu$ column.



When the matrix (**A** |$\Delta\mu$) satisfies the two rules above, we can say that this matrix equation could be found a solution set which satisfies the Firing Times Rules. The following is an example to illustrate the two rules above.

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	Δμ
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	1	0	0	0	0	1	0	0	1	0	0	0	1
2	0	0	1	0	0	-1	-1	0	-1	-1	-1	1	0	-1
3	0	0	0	1	0	1	1	0	0	0	0	-1	0	0
4	0	0	0	0	1	1	0	0	1	0	0	-1	0	0
5	0	0	0	0	0	0	0	1	1	1	0	0	0	1
6	0	0	0	0	0	0	0	0	0	0	0	0	1	1
7	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

Figure 3-13 (a)

	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	Δμ
0	1	0	0	0	0	0	0	0	0	0	0	0	0	-1
1	0	1	0	0	0	0	1	0	0	1	0	0	0	-1
2	0	0	1	0	0	-1	-1	0	-1	-1	-1	1	0	0
3	0	0	0	1	0	1	1	0	0	0	0	-1	0	0
4	0	0	0	0	1	1	0	0	1	0	0	-1	0	1
5	0	0	0	0	0	0	0	1	1	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	1	0
7	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

(b)

Figure 3-13 Example of illustrating EF properties judging rules

In Figure 3-13(a) row 0, the coefficients in the **A** part are zero except column t_0. The sign of (0, t_0) is positive and it is the same as (0, $\Delta\mu$). So we can find a legal solution of t0. In row 1, there are three non-zero coefficients and they are all positive, and the value (1, $\Delta\mu$) is also positive. We can find solutions of t_1, t_6, t_9 such that satisfy the equation and the Firing Times Rules like (t_1, t_6, t_9) = (1, 0, 0). When there are positive and negative coefficients in a row in the **A** part, it means that there is at least one loop in this system and we can adjust some transitions to change the firing times of other transitions. So for one row, no matter what the sign of column $\Delta\mu$ is, we always can find a solution set that satisfies the equation when there are positive and negative coefficients in the **A** part.

In Figure 3-13(b), the coefficient in row 0 column t_0 is positive but $\Delta\mu$ is negative. They have different signs and the solution of t0 must be -1, it violates the Firing Times Rules. In row 1, there are three positive coefficients in the **A** part but the value of $\Delta\mu$ is negative. We can not find any solutions of (t_1, t_6, t_9) which are positive integers and satisfy the equation.

For **EX** properties, we always need to solve the solution to verify it. Because even though the matrix equation has a solution which satisfies Firing Times Rules, we also need to check if the solution satisfies next state property. So, when we meet an

EX property and the ranks of the matrix equation are less than its number of transitions, we can not get a solution and also can not verify **EX** properties at this kind of situation. The verification flow of ranks < NO. of transitions is shown in Figure 3-14.

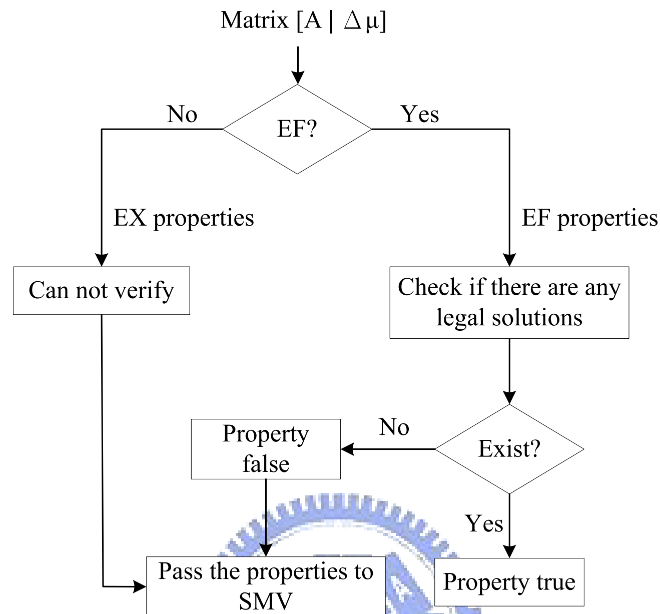


Figure 3-14 Flow of ranks < NO. of transitions

3.4.3 Summary

According to the illustration in the two section above, we could make a brief summary that when we find the ranks of the matrices **A** and ($\mathbf{A}$ | $\Delta \mu$) of a system are equal and the same as the number of its transitions, we will solve the matrix equation and check if the solution tallies with Firing Times Rules. When the ranks are equal and the solution satisfies the rules, we will show that the property is true. When we find that the ranks of the matrices are equal but they are less than the number of its transitions, if the property is **EF**, we will check if the matrix has solutions satisfies Firing Times Rules. When the property is **EX**, we can not verify it and will pass the property to SMV directly. When we find that the ranks of the two matrices are not

equal, we will directly show that the property is false and pass the property to SMV. The total verification flow is shown in Figure 3-15.

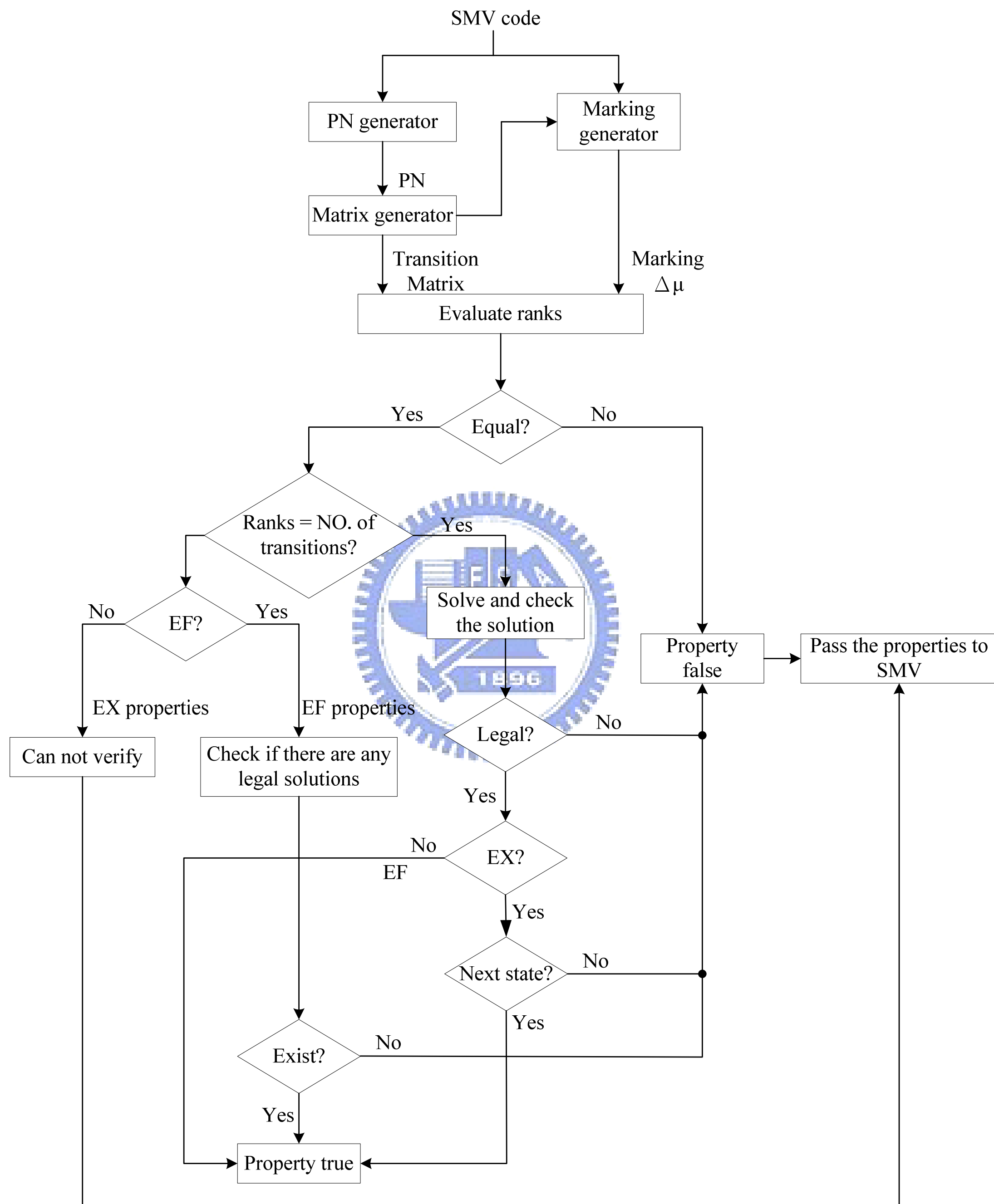
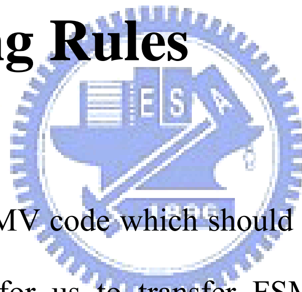


Figure 3-15 PNV software verification flow

Chapter 4   Implementation

In this section, we will discuss the implementation of PNV for each block in Figure 3-1.

4.1 Input Coding Rules



The input data of PNV is a SMV code which should obey the coding rules we made. The coding rules are made for us to transfer FSM descriptions into a Petri net structure more easily and conveniently. The codes which obey the coding rules are still readable by SMV model checker. The coding rules are

1. A signal which controls two state transitions should be named starting with "_", e.g. "_enable".
2. In the input SMV code, each symbol, signal name, and keyword should be separated by a blank space.
3. The properties should be described completely including the states and the values of signals.
4. Adding "--" at the end of the signal transition descriptions which do not need to be transferred into transition in Petri net.

The first rule is made for PNV to recognize which signal needs to be transferred into two places. The second rule is made for PNV to read the input SMV code easily. The purpose we make the third rule is to make marking generator generate markings easily. The fourth rule can make a Petri net and its transition matrix smaller in some situation. Even if we don't allow the fourth rule, the codes are still readable for PNV.

Figure 4-1 is a simple example of SMV code which obeys the coding rules above.

```
1  MODULE counter ( enable )
2  VAR
3     state : { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 } ;
4  ASSIGN
5     init ( state ) := 0;
6     next ( state ) := case
7        state=0 & enable=0 : 0 ;--
8        state=0 & enable=1 : 1 ;
9        state=1 : 2 ;
10       state=2 : 3 ;
11       state=3 : 4 ;
12       state=4 : 5 ;
13       state=5 : 6 ;
14       state=6 : 7 ;
15       state=7 : 0 ;
16       esac ;
17 MODULE main
18 VAR
19    cnt : counter ( en ) ;
20    en : boolean ;
21 SPEC
22    AG ( en=1 & cnt.state=0 -> EF ( cnt.state=7 ) )
23 SPEC
24    AG ( cnt.state=1 -> EX ( cnt.state=2 ) )
```

Figure 4-1 SMV code of counter

Figure 4-1 is a finite state machine of a counter. From line 1 to line 3 and from line 17 to line 20 are the definitional part. Line 1 declares a module called counter and enable is its input. Line 3 declares the state of the counter which has eight values from 0 to 7. From line 4 to line 16 are the structural part. Line 5 defines that the initial value of state is 0. From line 6 to line 16 define the state transitions of counter. Line 7 describes that when enable = 0 and state = 0, the next state is still 0. We don't need to generate any transition in Petri net for this description. So, according to the

4th coding rule, we add “--” at the end of line 7. Line 17 is main module in which the designer defines the connections of all sub-modules. From line 21 to line 24 are the expression part. Line 22 and line 24 are the properties of this system. The modules before main module are sub-modules which are used in main module. The descriptions in main module connect the whole sub-modules together and define properties.

4.2 Data Structure of PNV

The work of PNV is to transfer a SMV code into Petri net structure and to generate the transition matrix then verify it. Figure 4-2 is the data structure of a Petri net built by PNV.

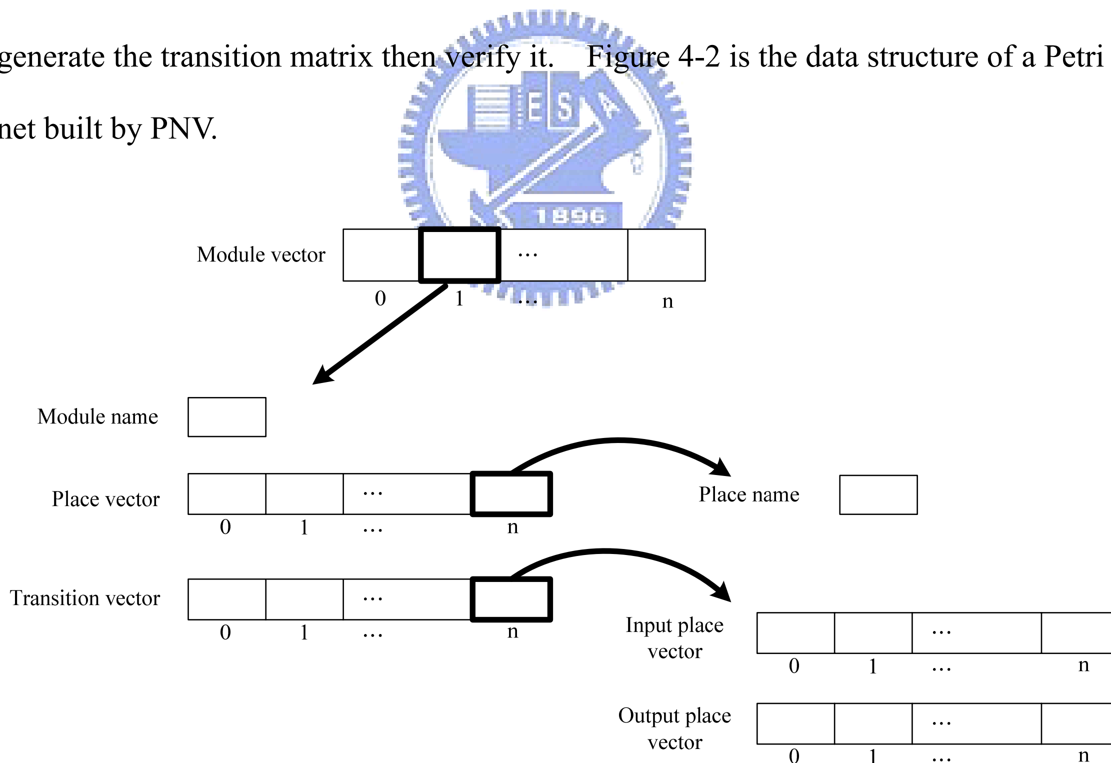


Figure 4-2 Data structure of PNV

There are three objects, Module, Place, and Transition, in the structure. For each module, there are three elements, a place vector, a transition vector, and a string, in it.

Each place has a string to represent its name. Each transition has two place vectors, one is to store the input places of the transition and the other is to store the output places of the transition [35].

The first step of building a Petri net is to generate modules. When PNV reads the keyword, MODULE, PNV pushes a new module which is named as the word after MODULE in the SMV code into the module vector. After creating a new module, PNV continues to build the places and transitions of this module according to the declarations in VAR part and ASSIGN part respectively. PNV creates one or two places to represent a signal which is declared as a Boolean signal. When there is a signal which is a set of signals like states, PNV creates places according to the number of signals in the set. For example, in Figure 4-1, PNV may push a module named "counter" in the module vector first and then push one place to represent the input signal, enable, and eight places to represent the signal, state (a set of signals), in the place vector which belongs to counter. When PNV reads a signal which is starting with "_" (coding rule 1st), PNV pushes two places into the place vector to represent the two values of the signal.

After generating all places, the next step is to build the connections of places and transitions. The descriptions in the ASSIGN parts in a SMV code express the relationship of places and transitions. PNV builds the connections of places and transitions based on the descriptions. In Figure 4-1 line 8, when PNV reads this description, PNV knows that there is a transition which has two input places, state=0 and enable=1, and one output place, state=1. Then PNV pushes these I/O data into the transition vector. After pushing all transitions into the transition vector, a module is built completely. Then PNV continues to build the following modules using the same methods.

After building all sub-modules, the last module must be main module. Main

module is a module which composes the sub-modules to become a complete system. The work of main module is to build the interconnections of the sub-modules included in main module. Figure 4-3 is an example that there are two sub-modules, fsm_a and fsm_b, in this system.

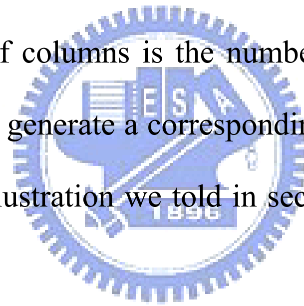
```
1 MODULE fsm_a ( _BWRITE , DSELAPB , PD )
2 VAR
3    stateA : { 0 , 1 , 2 , 3 , 4 , 5 , 6 } ;
4    _PSTB : boolean ;
5    PSEL : boolean ;
6    _PWRITE : boolean ;
7    BD : boolean ;
8 ASSIGN
9                .
10               .
11          (omitted)
12               .
13 MODULE fsm_b ( _PW , PSEL , _PSTB )
14 VAR
15   stateB : { 0 , 1 , 2 , 3 , 4 } ;
16   PD : boolean ;
17 ASSIGN
18               .
19               .
20          (omitted)
21               .
22 MODULE main
23 VAR
24   FA :  fsm_a ( _BW , DSEL , FB.PD ) ;
25   FB :  fsm_b ( FA._PWRITE , FA.PSEL , FA._PSTB ) ;
26   _BW : boolean ;
27   DSEL : boolean ;
28 ASSIGN
29   init ( _BW ) := 0 ;
30 SPEC
31               .
32               .
33          (omitted)
34               .
```

Figure 4-3 SMV code

The main module includes two sub-modules and declares two new signals in its VAR part. The two sub-modules included in main module are named FA and FB respectively. The signals in FA and FB will be named starting with “FA.” and “FB.” respectively. It is to mark the origin of the signals in main module. Line 24 and 25 in Figure 4-3 are ports mapping of the two sub-modules. When PNV reads line 24, PNV copies the places and transitions from the module fsm_a to main module. After copying the places and transitions, PNV changes the names of the places into starting with “FA.” and changes the names of the input signals from their original

names into the corresponding names that connect to the input ports. For example, in Figure 4-3 line 24, PNV copies the places from fsm_a and changes the names. Besides the input signals, all places' names would be changed starting with "FA.". But the input signals, _BWRITE, DSELAPB and PD, will be changed as _BW, DSEL and FB.PD respectively. _BW and DSEL are input signals from outside and FB.PD is an inner signal of the system from fsm_b. After processing the VAR part of main module, the whole system is built completely.

After building a whole Petri net structure, the following step is to generate the matrix. Because PNV stores places and transitions in the place vector and the transition vector respectively in main module, we can easily know that how many places and transitions in this system. The number of rows is the number of places of the system and the number of columns is the number of transitions. We get these data from the two vectors and generate a corresponding size matrix. Then PNV fills the matrix according to the illustration we told in section 2.4.3. After doing so, the matrix is accomplished.



4.3 Property to Marking

To verify a system represented with Petri net, we need to transfer its properties into markings. From the above works, each place stored in the place vector has an index to locate where it stored in the vector. We use the indexes to locate the positions of the places and create an n (the number of places) by 1 matrix to store the marking of each property. The methods of transfer a property into a marking is illustrated in section 2.4.2 and section 3.3. The information we need to verify is $\Delta\mu = \mu' - \mu$. A property description is beginning from its starting state and

finishing at the ending state. The starting state is μ and the ending state is μ'. According to the transformation methods in section 3.3, we can directly compose the generation of μ and μ' by adding 1 in the places of the ending state and adding -1 in the states in the starting state. So when PNV reads a property, it adds a corresponding number in each space in the marking matrix. For example, in Figure 4-1 the first property in line 22 describes that the starting state is en=1 and cnt.state=0, PNV adds -1 in the spaces which represent en=1 and cnt.state=0 and adds 1 in the spaces which represents cnt.state=7. Those spaces represent other places are still 0. Figure 4-4 is the transition matrix and the first marking in Figure 4-1.

	t0	t1	t2	t3	t4	t5	t6	t7	Δμ
en = 1	-1	0	0	0	0	0	0	0	-1
state = 0	-1	0	0	0	0	0	0	1	-1
state = 1	1	-1	0	0	0	0	0	0	0
state = 2	0	1	-1	0	0	0	0	0	0
state = 3	0	0	1	-1	0	0	0	0	0
state = 4	0	0	0	1	-1	0	0	0	0
state = 5	0	0	0	0	1	-1	0	0	0
state = 6	0	0	0	0	0	1	-1	0	0
state = 7	0	0	0	0	0	0	1	-1	1

Figure 4-4 Transition matrix

4.4 Verification Core

After the collection of markings and transition matrix, PNV can start to verify the system with the methods we have told in section 3.4. Assuming that the transition matrix is an m by n matrix and the markings are m by 1 matrix. First, PNV creates a new empty matrix and copy the transition matrix into the matrix and

then do Gaussian elimination to evaluate the rank of the transition matrix. Second, PNV copies the transition matrix into the new matrix again and copies the marking into the next column of the transition matrix, the matrix becomes ($\mathbf{A} \mid \Delta \mu$). Then PNV starts to evaluate the rank of the ($\mathbf{A} \mid \Delta \mu$) matrix. When the two ranks are unequal, this matrix equation does not have solutions. PNV will pass the property to SMV to verify it again and generate a counterexample. When the two ranks are equal and the same as the number of its transitions, PNV will solve the solutions of the equation and check if all solutions satisfy Firing Times Rules. When PNV gets an **EX** property, there is one more rule should be checked. It is that the fired transitions in the solution should not have causality. The reasons we have told in section 3.4.2.

When the property is **EF** and the ranks are equal but less than the number of its transitions, PNV will check if the equation has a solution satisfies Firing Times Rules. When the property is **EX** and the ranks are equal but less than the number of its transitions, PNV can not verify the property and will pass the property to SMV.

4.4.1 Elimination Methods

The elimination method of PNV is based on Gaussian elimination and some special methods. The methods are devised according to the characteristic of transition matrices to simplify the computational complexity of the elimination process.

A transition matrix of a Petri net has a characteristic that the most values in the matrix are zero. For each column, only the rows which represent the places connect to this transition have non-zero values. For each row, only the columns which represent the transitions connect to this place have non-zero values. In other words,

in a transition matrix, the most values are zero and the second most values are 1 or -1. The causes of this characteristic are that for each row, the non-zero columns are the transitions connected with the place. For a place, especially state place, there are not many transitions connect with it. This characteristic means we can utilize some specific methods to eliminate a transition matrix with fewer steps than using general Gaussian elimination. During generating a transition matrix, we would put the state places together to make the characteristic more obvious. By doing so, the work of elimination may be reduced outstandingly.

One of the methods we devise is that when we find one row with zero lead value, we can ignore this row in the elimination step. In other words, we check the representative value and decide if we need to eliminate this row. When the representative value is not zero, we have to eliminate it, but if the value is zero, we can skip the row. Though checking the representative value, we can save the work of eliminating unnecessary rows. According to this method, we can avoid some unnecessary calculation. So, we can say that the method devised from the characteristic of a transition matrix can make us more easily and quickly eliminate the transition matrices.

Because of the characteristic of a transition matrix, there is another method could be used to simplify the complexity of elimination. We check the values in the pivotal row and memorize the columns of the non-zero values. When we start to eliminate other rows, we can only calculate the non-zero columns and skip the other columns which are zero. By using this method, we can reduce the calculation when we meet a row which needs to be eliminated.

According to the methods above, the first step of elimination is to find out a non-zero value in the first column to be the pivotal row. When the pivotal row is not the first row, we have to exchange the pivotal row with the first row. Then we check

all values in the pivotal row and record the column indexes of the non-zero values. The next step is to eliminate the first column. We will check the values in the first column and eliminate the non-zero rows. The following is an example illustrating the steps we mentioned above.

	t0	t1	t2	t3	t4	t5	t6	t7
0	0	0	1	-1	0	0	0	0
1	-1	0	0	0	0	0	0	1
2	1	-1	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	1	-1	-1	0	0	0
5	0	0	0	0	1	-1	0	0
6	0	0	0	0	0	1	-1	0
7	0	0	0	0	0	0	1	0
8	-1	0	0	0	0	0	0	1

(a)

	t0	t1	t2	t3	t4	t5	t6	t7
0	-1	0	0	0	0	0	0	1
1	0	0	1	-1	0	0	0	1
2	1	-1	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	1	-1	-1	0	0	0
5	0	0	0	0	1	-1	0	0
6	0	0	0	0	0	1	-1	0
7	0	0	0	0	0	0	1	0
8	-1	0	0	0	0	0	0	1

(b)

	t0	t1	t2	t3	t4	t5	t6	t7
0	-1	0	0	0	0	0	0	1
1	0	0	1	-1	0	0	0	0
2	0	-1	0	0	0	0	0	1
3	0	1	0	0	0	0	0	0
4	0	0	1	-1	-1	0	0	0
5	0	0	0	0	1	-1	0	0
6	0	0	0	0	0	1	-1	0
7	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0

(c)

Figure 4-5 Eliminating example

Figure 4-5(a) is a transition matrix. For the first column, the (0, 0) position is zero, so row 0 can not be the pivotal row. We continue to check (1, 0) and find that it is a non-zero value, so we exchange row 0 and row 1 to become the matrix shown in Figure 4-5(b). After deciding the pivotal row, we start to check the items in the pivotal row and record the indexes of non-zero values. We find that column 0 and column 7 are not zero and we record the indexes 0 and 7. Then we start to eliminate the non-zero values in column 0 and only calculate the columns we have recorded.

In this example, we can eliminate the first column through four times of calculation with our methods instead of use full Gaussian elimination with sixty-four times of calculation.

For an m by n matrix, we assume that m and n are the same order and m = n + k. To find out the first pivotal row and exchange it to the first row, we have to check column 0 m times in the worse case and do 3n times to exchange the two rows. To find the second pivotal row, we have to check (m - 1) times and exchange 3(n - 1) times. The above operations we have to do for a matrix are

$$[m + 3n] + [(m - 1) + 3(n - 1)] + \ldots$$

$$= [(n + k) + 3n] + [(n + k - 1) + 3(n - 1)] + \ldots$$

$$= \sum_{i=1}^{n} (i + k) + 3i$$

$$= \sum_{i=1}^{n} 4i + k$$

$$= 2n\,(n + 1) + kn$$

$$= 2n^2 + (2 + k)\,n$$

The computational complexity of finding the pivotal row and exchanging it to the right position is $O(n^2)$.

To eliminate the first column, we have to check (m – 1) rows to figure out which rows need to be eliminated and we assume that there are c_i rows need to be eliminated. Before we start to eliminate, we need to check the values in the pivotal row and it cost n operations. After checking the pivotal row, we find out that there are d_i non-zero values. So the number of calculation we need to do is $c_i * d_i$. The total operations we have to do for eliminating a matrix into reduced row echelon form are

$$[n + (m - 1) + c_1 * d_1] + [(n - 1) + (m - 1) + c_2 * d_2] + \dots$$

$$= [n + (n + k - 1) + c_1 * d_1] + [(n - 1) + (n + k - 1) + c_2 * d_2] + \dots$$

$$= \sum_{i=1}^{n} i + (i + k - 1) + c_i * d_i$$

$$= \sum_{i=1}^{n} 2i + (k - 1) + c_i * d_i$$

$$= n\,(n + 1) + (k - 1)\,n + \sum_{i=1}^{n} c_i * d_i$$

$$= n^2 + kn + \sum_{i=1}^{n} c_i * d_i$$

The order of $\sum_{i=1}^{n} c_i * d_i$ term is about n. The computational complexity of the elimination step is $O(n^2)$. After the elimination step, we get a matrix in reduced row echelon form. When the rank of the matrix is equal to its transition number, we can get the firing times of each transition without more action. An example is shown in Figure 4-6, the matrix is in reduced row echelon form and the rank is the same as its transition number, so we can get the solutions of t_0 to t_5 are a to f without back substitution. Summarily, the total computational complexity of our elimination process is $O(n^2)$ and its order is lower than the computational complexity of original Gaussian elimination $O(n^3)$. The methods we use to eliminate a transition matrix can reduce the order of computational complexity from $O(n^3)$ to $O(n^2)$.

	t0	t1	t2	t3	t4	t5	Δμ
0	1	0	0	0	0	0	a
1	0	1	0	0	0	0	b
2	0	0	1	0	0	0	c
3	0	0	0	1	0	0	d
4	0	0	0	0	1	0	e
5	0	0	0	0	0	1	f

Figure 4-6 Reduced row echelon form

Chapter 5 Experimental Results

In this chapter, we will use PNV to verify five examples and interpret the rules we mentioned above. The simulation environment of the following examples is: "OS: Windows XP, CUP: AMD 3000+, RAM: 1GB."

5.1 Counter

Figure 5-1 is a finite state machine of a four-state counter. The initial state of the FSM is s0.

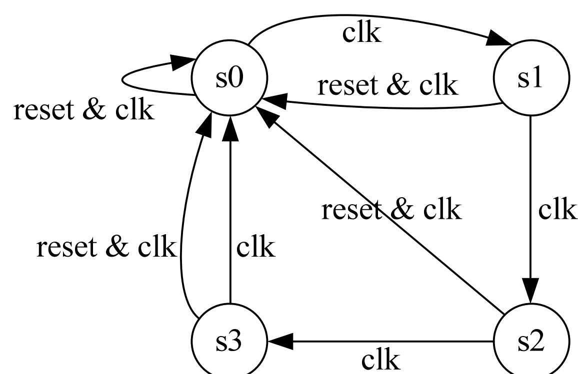
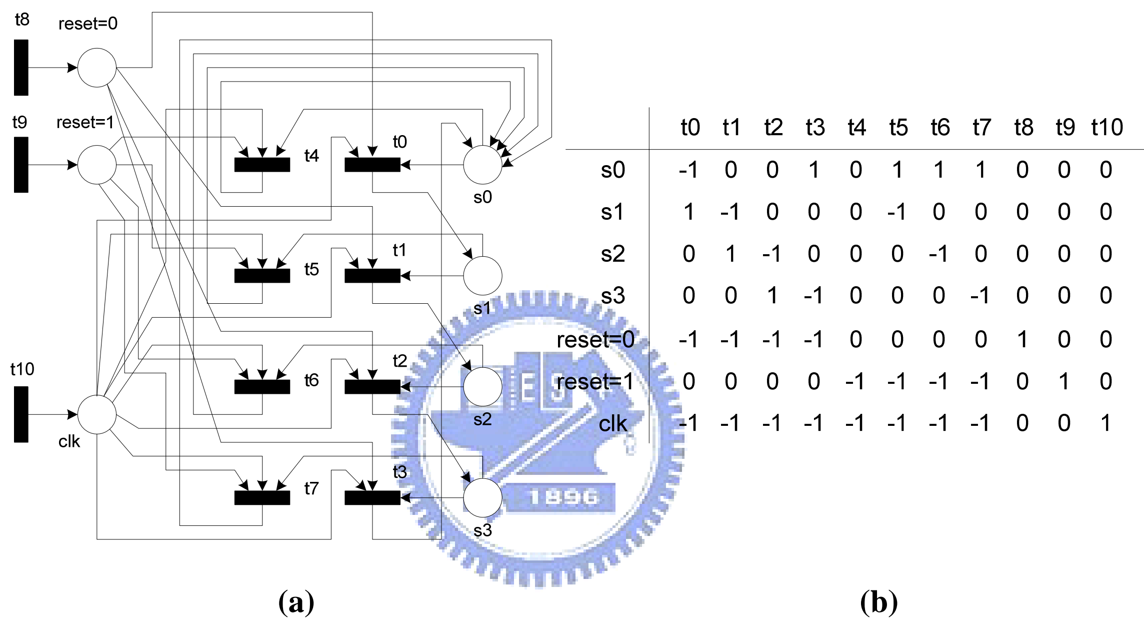


Figure 5-1 FSM of counter

Figure 5-2 is the Petri net graph and its transition matrix of the counter. In Figure

5-2(a), there are two places used to represent the reset signal. It is because that state transitions should always happened when reset = 0. So we need two places to indicate reset = 1 and reset = 0 as shown in Figure 5-2(a). The transitions t_0 to t_3 are working for state transition, t_4 to t_7 are working for reset, and t_8 to t_{10} are for input signals. The transitions t_0 to t_3 always need one input place which is reset = 0 to ensure that the system is not being reset.



	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
s0	-1	0	0	1	0	1	1	1	0	0	0
s1	1	-1	0	0	0	-1	0	0	0	0	0
s2	0	1	-1	0	0	0	-1	0	0	0	0
s3	0	0	1	-1	0	0	0	-1	0	0	0
reset=0	-1	-1	-1	-1	0	0	0	0	1	0	0
reset=1	0	0	0	0	-1	-1	-1	-1	0	1	0
clk	-1	-1	-1	-1	-1	-1	-1	-1	0	0	1

(a) **(b)**

Figure 5-2 Petri net of the counter

The property of this Petri net we set is "**AG** (state=0 -> **EF** (state=3))". This property means that there is at least one path starting from state = 0 and finally ending at state = 3. We verify this property by SMV and PNV respectively. Figure 5-3 is the verification result generated by PNV verbose mode. Figure 5-3(a) shows the places and the transitions generated by PNV. The first part in Figure 5-3(b) shows the transition matrix of the Petri net, the second part shows the matrix after being eliminated and the rank of this matrix, the third part shows the property and the corresponding marking, the fourth part shows the result of eliminated (**A** | $\Delta\mu$), and

the final part shows the verification result of the property and time cost. There is a brief mode of PNV shown in Figure 5-4, this mode only shows the result and time cost of the properties and it is faster than verbose mode. Figure 5-5 is the verification result generated by SMV.

```
C:\Documents and Settings\Porsche\My Documents\work\Petr...
Please enter the input file : cnt4.smv -i -v
NO. of places = 7
  0      cnt.state=0
  1      cnt.state=1
  2      cnt.state=2
  3      cnt.state=3
  4      clk=1
  5      _reset=0
  6      _reset=1

NO. of transitions = 11
Tran 0 :
    Input : cnt.state=0 _reset=1 clk=1
    Output : cnt.state=0
Tran 1 :
    Input : cnt.state=1 _reset=1 clk=1
    Output : cnt.state=0
Tran 2 :
    Input : cnt.state=2 _reset=1 clk=1
    Output : cnt.state=0
Tran 3 :
    Input : cnt.state=3 _reset=1 clk=1
    Output : cnt.state=0
Tran 4 :
    Input : cnt.state=0 _reset=0 clk=1
    Output : cnt.state=1
Tran 5 :
    Input : cnt.state=1 _reset=0 clk=1
    Output : cnt.state=2
Tran 6 :
    Input : cnt.state=2 _reset=0 clk=1
    Output : cnt.state=3
Tran 7 :
    Input : cnt.state=3 _reset=0 clk=1
    Output : cnt.state=0
Tran 8 :
    Input :
    Output : clk=1
Tran 9 :
    Input :
    Output : _reset=0
Tran 10 :
    Input :
    Output : _reset=1
```

(a)

```
C:\Documents and Settings\Porsche\My Documents\work\Petr...
Matrix A
   0   1   1   1  -1   0   0   1   0   0   0
   0  -1   0   0   1  -1   0   0   0   0   0
   0   0  -1   0   0   1  -1   0   0   0   0
   0   0   0  -1   0   0   1  -1   0   0   0
  -1  -1  -1  -1  -1  -1  -1  -1   1   0   0
   0   0   0   0  -1  -1  -1  -1   0   1   0
  -1  -1  -1  -1   0   0   0   0   0   0   1

Reduced A
   1   0   0   0   0  -1  -1  -2   0   1  -1
   0   1   0   0   0   2   1   1   0  -1   0
   0   0   1   0   0  -1   1   0   0   0   0
   0   0   0   1   0   0  -1   1   0   0   0
   0   0   0   0   1   1   1   1   0  -1   0
   0   0   0   0   0   0   0   0   1  -1  -1
   0   0   0   0   0   0   0   0   0   0   0
Rank(A) = 6


Property :
  AG ( cnt.state=0 -> EF ( cnt.state=3 ) )

Marking :
  -1
   0
   0
   1
   0
   0
   0

Reduced [A U]
   1   0   0   0   0  -1  -1  -2   0   1  -1   1
   0   1   0   0   0   2   1   1   0  -1   0   0
   0   0   1   0   0  -1   1   0   0   0   0   0
   0   0   0   1   0   0  -1   1   0   0   0  -1
   0   0   0   0   1   1   1   1   0  -1   0   0
   0   0   0   0   0   0   0   0   1  -1  -1   0
   0   0   0   0   0   0   0   0   0   0   0   0
Rank([A U]) = 6

TRUE!

Time cost of pnv : 13.0455 ms!
```

(b)

Figure 5-3 Verification result of counter with PNV verbose mode

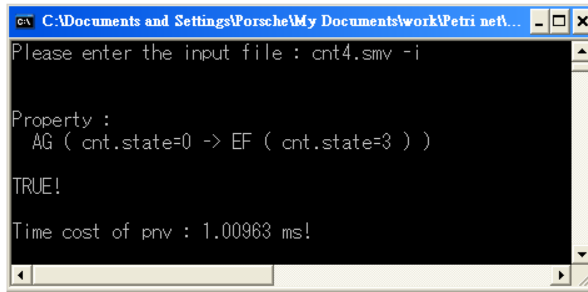


Figure 5-4 Verification result of counter with PNV brief mode

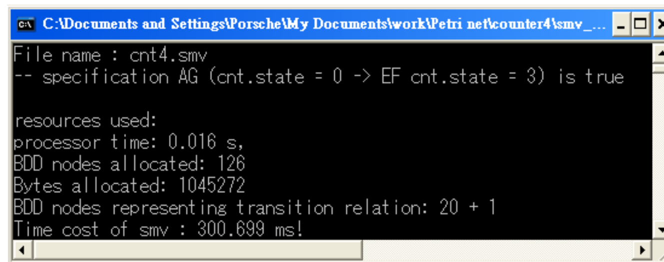


Figure 5-5 Verification result of counter with SMV

	SMV		PNV	
	Processor	Total	Verbose	Brief
Time (ms)	16	300.699	13.0455	1.00963
Memory (KB)	1184		1940	

Table 5-1 Time cost of SMV and PNV for counter

In this example, the rank of **A** and (**A** $|\Delta\mu$) are less than the number of transitions. We can not get an exact solution, so we verify this design by the methods illustrated in section 3.4.2.

The time costs of SMV and PNV are shown in Table 5-1. The SMV processor time is clocked by SMV and we don’t know what action included in this time interval and the accuracy is lower than the timer used by PNV. So we add another timer to clock the time cost of SMV, the result is shown in SMV Total. This result includes all action of SMV and its clocking condition is the same as PNV. Adding another timer to clock SMV makes us more convenient to compare the time costs of SMV and PNV. From Table 5-1 PNV part, we could notice that the most time cost by PNV is

to print out the result.

The processing time depends on the status of CPU, so the time cost of each processing may not be equal; even so, we still can find that the time which is shown in Table 5-1 costs of PNV are fewer than the time costs of SMV, especially at PNV brief mode.

5.2 Greatest Common Divisor

In this section, we will show an example about greatest common divisor (GCD) and verify it by SMV and PNV. Figure 5-6 is the FSM of GCD. The state s0 is idle mode, this machine starts when the start signal = 1 and the state transfers to s1. At s1, the machine transfers its state according to the inputs, u > v, u < v, and u = v. The state will transfer to s2 and s3 when u > v and u < v respectively. At these two states, the system will subtract the smaller number from the larger one then send the answer and the smaller number to a comparator. The comparator will identify which number is larger and send the result (u > v, u < v, or u = v) to GCD system. When the two numbers are equal, the state will transfer to s4 and the process is finished. The final number is the GCD of the original u and v.

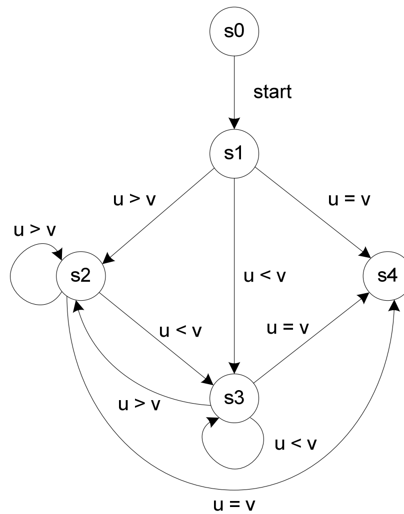


Figure 5-6 FSM of GCD

```
C:\Documents and Settings\Porsche\My Documents\work\Petri net\GCD\pn.exe
Property :
  AG ( gcd.state=1 -> EF ( gcd.state=0 & start=1 ) )

Marking :
   1
  -1
   0
   0
   0
   1
   0
   0
   0

Reduced [A U]
   1   0   0   0   0   0   0   0   0   0   0   0   0   0  -1
   0   1   0   0   0   0   1   0   0   1   0   0   0   0   0
   0   0   1   0   0  -1  -1   0  -1  -1   0  -1   1   0   0
   0   0   0   1   0   1   1   0   0   0   0   0  -1   0   0
   0   0   0   0   1   1   0   0   1   0   0   0  -1   0   0
   0   0   0   0   0   0   0   1   1   1   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   1   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
Rank([A U]) = 8

The solutions are not all positive!
FALSE!

Time cost of pnv : 25.2552 ms!
```

(a)

```
C:\Documents and Settings\Porsche\My Documents\work\Petri net\GCD\pn.exe
===============  SMV  ===============
-- specification AG (gcd.state = 1 -> EF (gcd.state = 0 &... is false
-- as demonstrated by the following execution sequence
state 1.1:
gcd.state = 0
start = 0
u = 0
v = 0
uv = 0

state 1.2:
start = 1

state 1.3:
gcd.state = 1
start = 0


resources used:
processor time: 0.015 s,
BDD nodes allocated: 466
Bytes allocated: 1045272
BDD nodes representing transition relation: 26 + 1

Time cost of smv : 241.413 ms!
```

(b)

Figure 5-7 Verification result of GCD with PNV verbose mode

	SMV		PNV	
	Processor	Total	Verbose	Brief
Time (ms)	31	285.795	25.2552	1.2306
Memory (KB)	1624		1944	

Table 5-2 Time cost of SMV and PNV for GCD

In this example, we show two **EF** properties and the second one is planned to be false deliberately to show the methods we told in section 3.4.2. In Figure 5-7(a), the property is false and we can see that the first row of the matrix Reduced [A U] violates the first rule in section 3.4.2. The solution of the first transition must be -1 and it is illegal. This property is verified as false by PNV then PNV will pass it to SMV and verify it again to generate a counterexample, it's shown in Figure 5-7(b). Table 5-2 shows the time costs of SMV and PNV.

5.3 AMBA

This example is to model the behavior of the communication between AMBA ASB and APB. We simplify the signals of the two components from Figure 5-8 and Figure 5-9. Figure 5-8 is the read timing graph and Figure 5-9 is the write timing graph. We ignore the BTRAN and BWAIT signals. In Figure 5-8, when DSELAPB = 1 means that APB is selected. Then the data will be sent to APB and PSEL will be set high. In the next clock cycle (c2 in Figure 5-8), PSTB will be set high and APB starts to read the data ASB needs [40, 42].

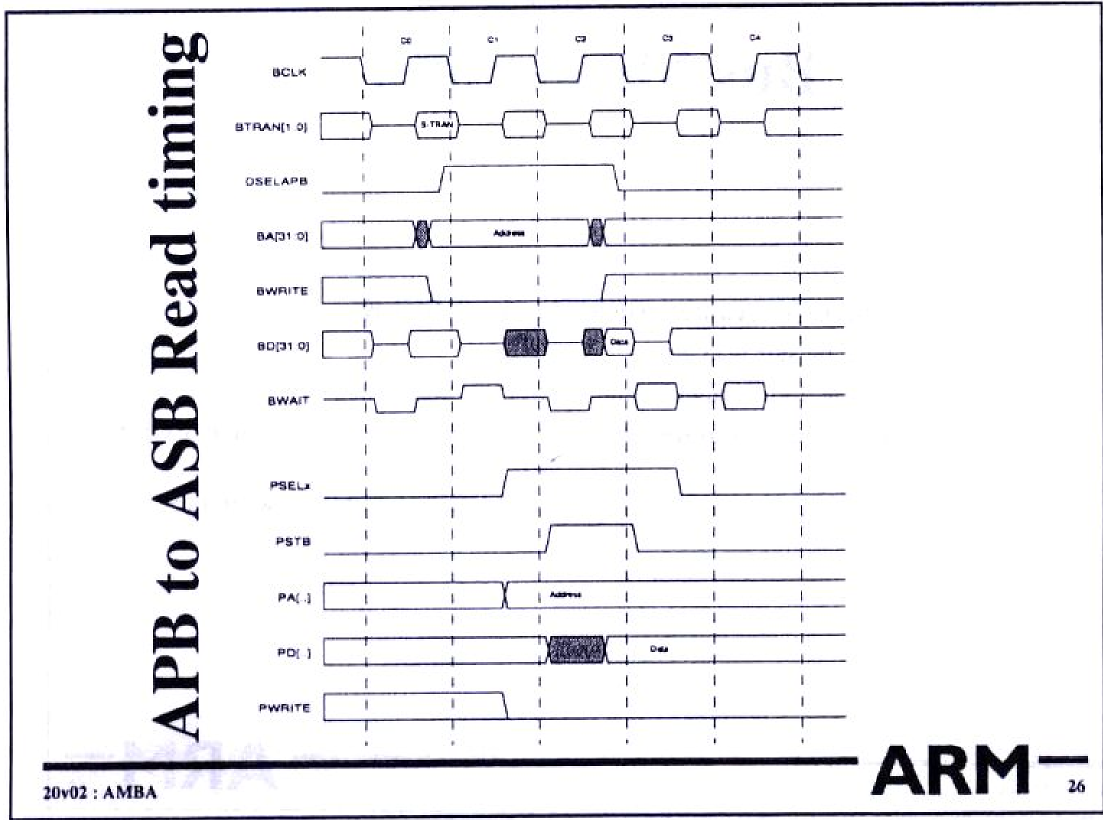


Figure 5-8 APB to ASB Read timing (Captured from [42])

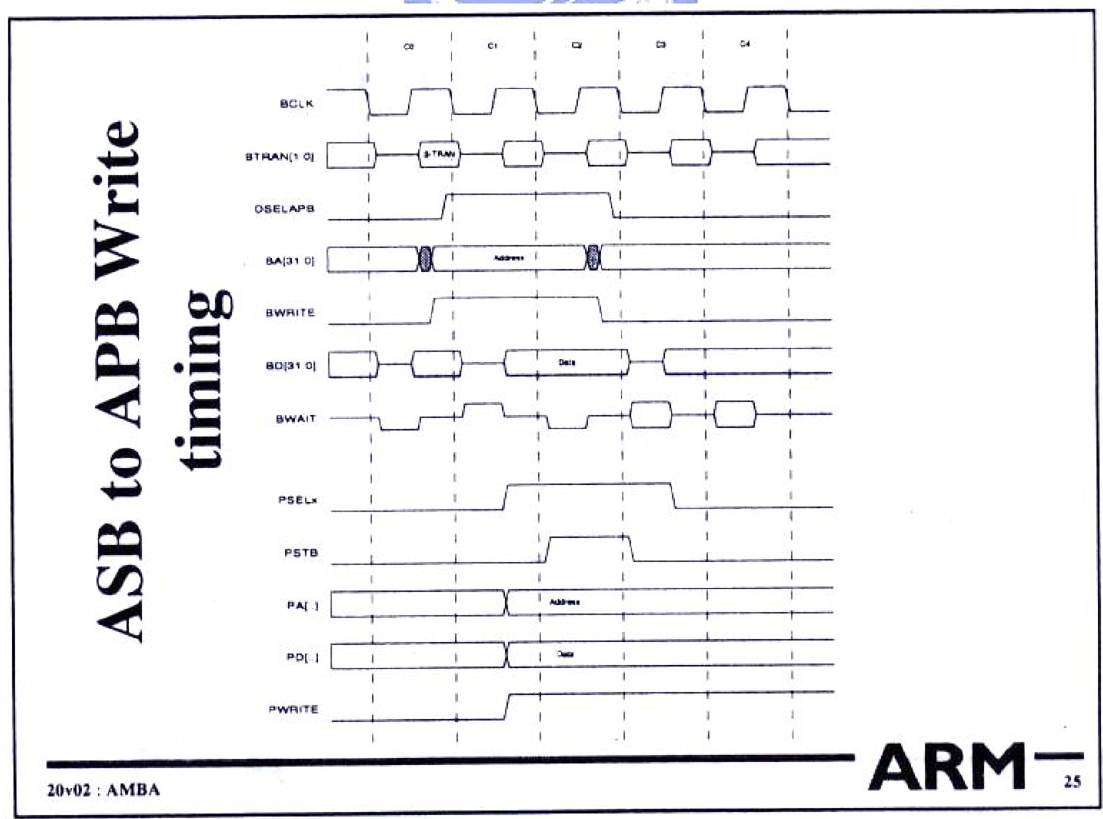


Figure 5-9 ASB to APB Write timing (Captured from [42])

Figure 5-9 is the timing graph of ASB writes data into APB. The BWRITE signal is high at this action and the same that APB start writing when PSTB is high.

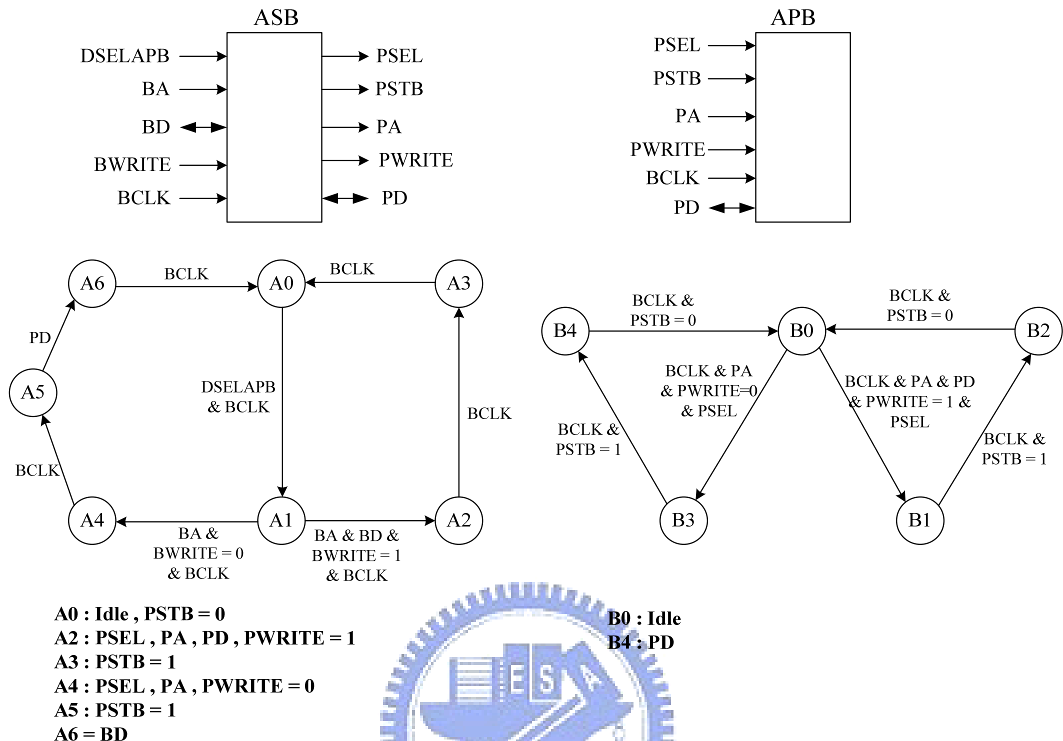


Figure 5-10 FSM of the communication of ASB and APB

Figure 5-10 are the FSMs we build according to the timing graph above. FSM A is ASB and FSM B is APB. The output signals of each state are shown under the FSMs. In this example, there are four properties set to be verified. The first property is that at the initial states A0 and B0, when DSELAPB is high and BWRITE is high, there exists a path which goes back to the initial states. The second property is to verify an **EX** property. In this example, the ranks are equal to the number of the transitions. So we can solve the solutions of the properties and can verify **EX** properties. The third and fourth properties are contrasts. The third property is to verify an **EF** property and the property is true. Then we change the **EF** operator into **EX**, it is the fourth property. These two properties are used to show that there is a path from the starting state to the ending state but the ending state is not the next state

of the starting state and PNV can recognize the if the ending state is the next state of the starting state.

The following are parts of the verification result of PNV and SMV.

```
C:\Documents and Settings\Porsche\My Documents\work\Petri net\AMBA\pn.exe
Property :
  AG ( FA.stateA=4 & FA._PWRITE=0 & FA.PSEL=1 & FB.stateB=0 -> EX ( FA.stateA=5  & FB.stateB=4 & FB.PD=1 ) )

Marking :
   0
   0
   0
   0
  -1
   1
   0
   0
   0
  -1
  -1
   0
   0
   0
   0
   0
  -1
   0
   0
   0
   1
   1

Reduced [A U]
   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   1   0   0   0   0   0   0   0   0   1
   0   0   0   0   0   0   1   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   1   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   1   0   0   0   0   1
   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   1   0   1
   0   0   0   0   0   0   0   0   0   0   0   0   0   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   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
   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   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
Rank([A U]) = 14

  T0  T1  T2  T3  T4  T5  T6  T7  T8  T9  T10  T11  T12  T13
   0   0   0   0   0   1   0   0   0   1    0    0    1    0
The solution of transitions is not the next state!
FALSE

Time cost of pnv : 85.0385 ms!
```

Figure 5-11 Verification result of AMBA with PNV verbose mode

	SMV		PNV	
	Processor	Total	Verbose	Brief
Time (ms)	16	408.82	85.0385	2.42684
Memory (KB)	1632		1956	

Table 5-3 Time cost of SMV and PNV for AMBA

Because the ranks and the number of transitions are equal, PNV can solve the solution for each property and shows the solutions in the verification result. In Figure 5-11, PNV finds that there is a solution of this property but the solution shows

that the ending state is not the next state of the starting state. So PNV determine that the property is false and pass the property to SMV. Table 5-3 shows the time costs of SMV and PNV.

5.4 Traffic Light Controller

This example is a crossroads traffic light controller. The illustration of the crossroads is shown in Figure 5-12.

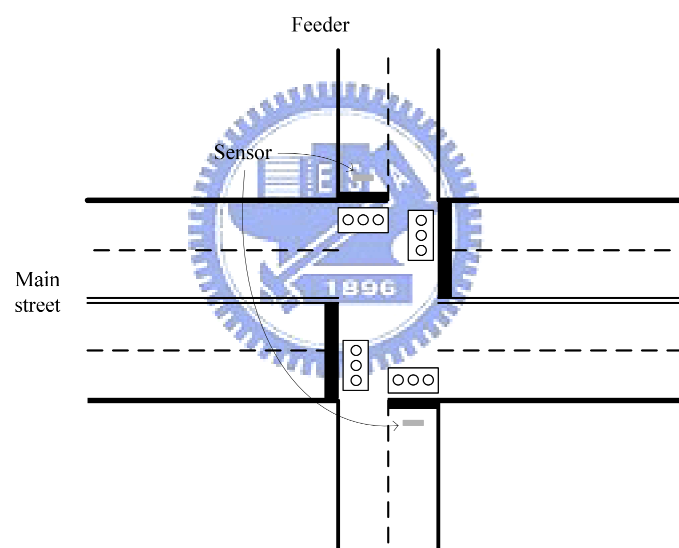
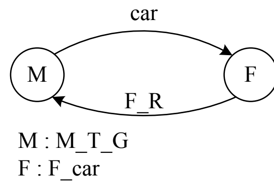
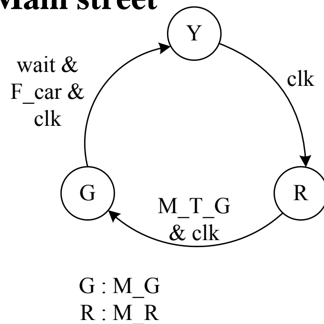


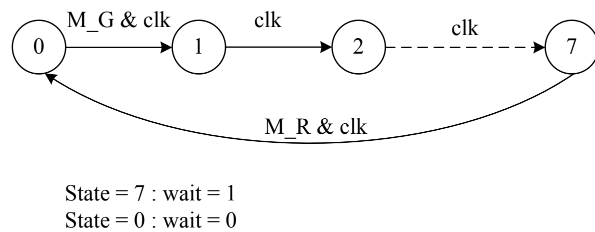
Figure 5-12 Traffic light controller

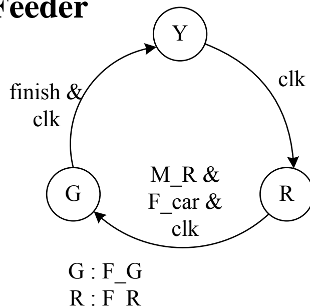
We assume that there are few cars on feeder, so the traffic lights of feeder are red at most time. There are two sensors on feeder shown in Figure 5-12, when there is a car which wants to cross the main street, the sensors will send a signal to the controller then the traffic lights of feeder will turn into green and the light of the main street will turn into red for a short time. After this, the feeder lights will go back to red and the main street lights will turn into green. When there is no cars want to cross the street on feeder, the traffic lights will always green for main street and red

for feeder. The behavior of the controller is shown in Figure 5-13.









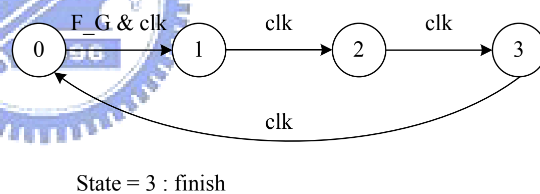


Figure 5-13 FSMs of traffic light controller

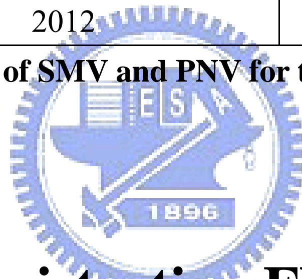
The initial state of each machine is: State controller: M; Main street: G; M_counter: 0; Feeder: R; F_counter: 0. At the beginning, M_counter starts to count for seven clock cycles to ensure that there is enough time for main street cars to pass. When there are any cars want to pass the street during the seven clock cycles, the traffic lights of feeder will not be turned into green. After seven clock cycles, when there is a car in feeder wants to pass the street, the controller will turn the lights of the main street into red and turn the lights of feeder into green; when there is no car waiting to pass in

feeder, the controller will still let the cars in main street pass. When the lights of feeder is turned into green and the lights of main street is turned into red, F_counter starts to count for three clock cycles then turns the feeder lights red and main street lights green.

In this example, we set a property which is: "At the initial state (main street lights are green and feeder lights are red), there exists one path to the state that main street lights are red and feeder lights are green." Table 5-4 shows the time costs of SMV and PNV.

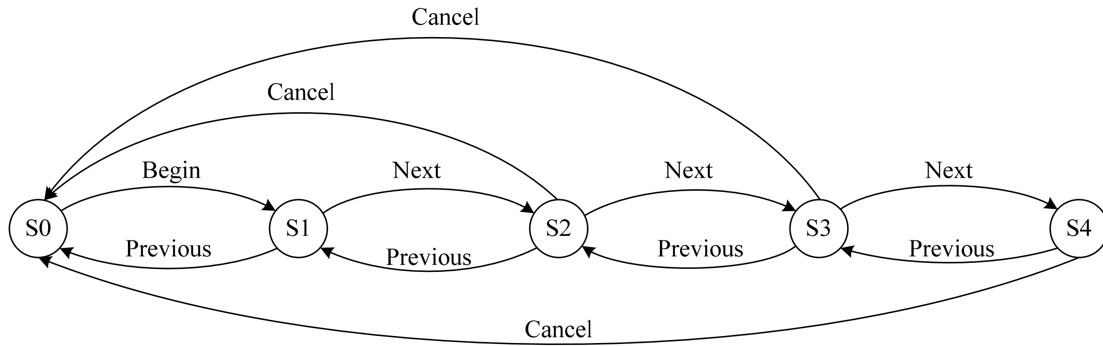
	SMV		PNV	
	Processor	Total	Verbose	Brief
Time (ms)	141	394.138	82.7625	2.42657
Memory (KB)	2012		1960	

Table 5-4 Time cost of SMV and PNV for traffic light controller



5.5 Wizard's Registration Flow

It is am example about the registration flow of Wizard we got it from .NET Framework Developer Center [43]. There are five states of this machine. The initial state is Default (S0). When the user wants to register and press Begin, the state will transfer into Register (S1). At Register, the user can choose Next to go to the next state Edit (S2) or Previous to go back to Register. At Edit (S2) and Edit user (S3), the user can choose Next or Previous to go forward or go back and also can choose Cancel to go back to Default. The last state is Finish which means the registration is done. If the user wants to change his setting, he can press Previous to go back or press Cancel to go to the initial state directly.



S0 : Default
S1 : Register
S2 : Edit
S3 : Edit user
S4 : Finish

Figure 5-14 The registration flow of Wizard

For this system, there is one property we set to be verified. It is "At the Default state and Begin signal is pressed, there exists a path to Finish state." Table 5-5 shows the time costs of PNV and SMV.

	SMV		PNV	
	Processor	Total	Verbose	Brief
Time (ms)	16	299.217	25.7013	1.23563
Memory (KB)	1628		1944	

Table 5-5 Time cost of SMV and PNV for Registration

Chapter 6 Conclusion and Future Work

6.1 Conclusion

In this paper, we introduce SMV and provide another verification method, PNV, based on modeling a FSM into a Petri net. Because of the verification method of PNV, we can use it to verify **EF** and partial **EX** properties. The purpose for us to propose this verification method is to assist SMV to speed up verifying **EF** and **EX** properties. We still verify the properties which PNV can not handle by SMV. The most advantageous situation of using PNV is that the **EF** properties are true and the **EX** properties can be verified by PNV and the properties are true. At this situation, we can only spend very short time and a little more memory usage than SMV on verifying these properties with PNV. When the properties are verified to be false by PNV, PNV will pass the properties to SMV to verify them again. PNV can identify the properties which are verified to be false by checking the ranks of the matrices and this work takes little time. The advantage of using PNV to verify **EF** and **EX** properties is that we can use little time on the work. The disadvantage of PNV is that PNV needs more memory than SMV to verify a design. Summarily speaking,

using PNV to assist SMV can earn the most benefit when the properties are true and can be processed by PNV. Even if the properties are verified to be false, we only spend little time more than only using SMV to verify them.

6.2 Future Work

The way we combine SMV and PNV now is calling SMV by PNV when it's necessary. It means that the two programs are still individual and they need to read the input file one time respectively. If SMV and PNV could be integrated into one program, the information of the input file can be shared between SMV part and PNV part and we don't need to waste time on reading the input file twice. Combining the two programs can also make users more convenient use the program.

The marking generator is not friendly enough for users. The generating method now is based on the properties and the user should describe the properties amply about each signal and state. If the marking generator could be made smarter, users could describe a property with only mentioning its states of some signals and the generator would filled with appropriate signals to make the property complete.

References

[1] James L. Peterson, *PETRI NET THEORY AND THE MODELING OF SYSTEMS*, Prentice-Hall, Inc. 1981

[2] Bruce Wile, john C. Goss, Wolfgang Roesner, *COMPREHENSIVE FUNCTIONAL VERIFICATION THE COMPLETE INDUSTRY CYCLE*, Elsevier Inc. 2005

[3] Edmund M. Clarke, Bernd-Holger Schlingloff, *HANDBOOK OF AUTOMATED RESONING*, Elsevier Science Publishers B.V. 2001

[4] Giovanni De Micheli, *SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS*, McGraw-Hill, Inc. 1994

[5] Varea M., Al-Hashimi B.M., Cortes L.A., Eles P., Zebo Peng, "Symbolic model checking of dual transition Petri Nets", Hardware/Software Codesign, CODES, pp.43 – 48, May 2002

[6] Zhenyu Chen, Conghua Zhou, Decheng Ding, "Automatic abstraction refinement for Petri nets verification", IEEE Int. High-Level Design Validation and Test Workshop, pp.168 – 174, Dec. 2005

[7] Karlsson, D., Eles, P., Zebo Peng, "Formal verification in a component-based reuse methodology", System Synthesis, 15th International Symposium, pp.156 – 161, 2002

[8] Weng X., Litz L., "Verification of logic control design using SIPN and model checking: methods and case study", American Control Conference, vol.6,

pp.4072 - 4076, June 2000

[9] Cortadella J., “Combining structural and symbolic methods for the verification of concurrent systems”, Application of Concurrency to System Design Int., pp. 2 – 7, March 1998

[10] Rodrigues C.L., Guerrero D.D.S., de Figueiredo J.C.A., “Model checking in object-oriented Petri nets”, IEEE Int. Systems, Man and Cybernetics, Vol 5, pp.4977 - 4982, Oct. 2004

[11] Pastor E., Cortadella J., Roig O., “Symbolic analysis of bounded Petri nets”, IEEE Trans. Computers, Vol 50, No. 5, pp.432 – 448, May 2001

[12] Wang J., Deng Y., Xu G., “Reachability analysis of real-time systems using time Petri nets”, IEEE Trans. Systems, Man and Cybernetics, Part B, Vol. 30, No. 5, pp.725 – 736, Oct. 2000

[13] Tsai J.J.P., Jennhwa Yang S., Yao-Hsiung Chang, “Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications”, IEEE Trans. Software Engineering, Vol 21, No. 1, pp.32 – 49, Jan. 1995

[14] Jiang J., Azzopardi D., Holding D.J., Carpenter G.F., Sagoo J.G., “Real-time synchronisation of multiaxis high-speed machines, from SFC specification to Petri net verification”, IEEE Control Theory and Applications, Vol 143, No. 2, pp.164 – 170, March 1996

[15] Tsung-Hsi Chiang, Lan-Rong Dung, Ming-Feng Yaung, “Modeling and formal verification of dataflow graph in system-level design using Petri net”, IEEE Int. ISCAS, Vol. 6, pp.5674 - 5677, May 2005

[16] Berthomieu B., Diaz M., “Modeling and verification of time dependent systems using time Petri nets”, IEEE Trans. Software Engineering, Vol. 17, No. 3, pp.259 – 273, March 1991

[17] Castelnuovo A., Ferrarini L., Piroddi L., "An incremental Petri net approach to production sequence modeling", IEEE Int. Automation Science and Engineering, pp.333 – 338, Aug. 2005

[18] Wang J., Deng Y., Zhou M., "Compositional time Petri nets and reduction rules", IEEE Trans. Systems, Man and Cybernetics, Part B, Vol. 30, No. 4, pp.562 – 572, Aug. 2000

[19] Zurawski R., "Petri net models, functional abstractions, and reduction techniques: applications to the design of automated manufacturing systems", IEEE Trans. Industrial Electronics, Vol. 52, No. 2, pp.595 – 609, April 2005

[20] Hadjidj R., Boucheneb H., "On-the-fly TCTL model checking for Time Petri Nets using state class graphs", ACSD Int. Application of Concurrency to System Design, pp.111 – 122, June 2006

[21] Yang S.J.H., Chu W., Lin S., Lee J., "Specifying and verifying temporal behavior of high assurance systems using reachability tree logic", IEEE Int. High-Assurance Systems Engineering Symposium, pp.150 – 156, Nov. 1998

[22] Emerson, E.A.; Namjoshi, K.S., "On model checking for non-deterministic infinite-state systems", IEEE Sym. Logic in Computer Science, pp.70 – 80, June 1998

[23] NuSMV source code, http://nusmv.irst.itc.it/

[24] SMV source code and execution file, Model Checking Group in Specification and Verification Center at CMU, http://www.cs.cmu.edu/~modelcheck/smv.html

[25] Cadence SMV model checker, Cadence Berkeley Labs, http://www.kenmcmil.com/smv.html

[26] J. R. Burch, E. M. Clarke, D. E. Long, "Symbolic model checking with partitioned transition relations", In *VLSI 91*, Edinburgh, Scotland, 1990.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit

verification using symbolic model checking", In *27th ACM/IEEE Design Automation Conference*, 1990.

[28] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic model checking: 10E20 states and beyond", In *LICS*, 1990.

[29] K. L. McMillan, "Symbolic model checking - an approach to the state explosion problem" PhD thesis, SCS, Carnegie Mellon University, 1992.

[30] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness, "Verification of the Futurebus+ cache coherence protocol", In L. Claesen, Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications. North-Holland, April 1993.

[31] E. M. Clarke, O. Grumberg, D. E. Long, "Model checking and abstract", ACM Symposium on Principles of Programming Languages, January 1992.

[32] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications", In ACM Trans. on Programming Languages and Systems, 8(2):244--263, 1986.

[33] Getting start with SMV tutorial, K. L. McMillan, Cadence Berkeley Labs 2001, http://www.cs.indiana.edu/classes/p515/readings/smv/CadenceSMV-docs/smv/tutorial/tutorial.html

[34] Steven J. Leon, *LINEAR ALGEBRA WITH APPLICATIONS*, Prentice Hall, 2002

[35] William Ford, William Topp, *DATA STRUCTURES with C++ using STL*, Prentice Hall, 2002

[36] E. M. Clarke and E. A. Emerson, Synthesis of synchronization skeletons for branching time temporal logic, In *Logic of Programs: Workshop, Yorktown Heights, NT, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981

[37] E. A. Emerson and J. Y. Halpern, "Sometimes" and "Not Never" revisited: On branching time versus linear time, *Journal of the ACM*, 33:151-178, 1986

[38] 洪維恩, *C++教學手冊(第二版)*, 博碩文化, April 2006

[39] 洪錦魁, *精通C語言*, 文魁資訊股份有限公司, March 2002

[40] Advanced Microcontroller Bus Architecture Specification

[41] Bloem R., Ravi K., Somenz F., "Symbolic guided search for CTL model checking", Design Automation Conference, pp.:29 – 34, June 2000

[42] ARM Powered Training documentation

[43] .NET Framework Developer Center, ASP.NET Technical Articles, Finite State Machines, Wizards, and the Web http://msdn2.microsoft.com/en-us/library/aa478972.aspx