2010

Comparison of encoding schemes for symbolic model checking of bounded petri nets

Nishtha Arora
Iowa State University

Comparison of encoding schemes for symbolic model checking of bounded petri nets

by

Nishtha Arora

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Andrew S. Miner, Major Professor
Samik Basu
Robyn Lutz

Iowa State University

Ames, Iowa

2010

DEDICATION

I would like to dedicate this thesis to my parents who have been a source of inspiration and encouragement to me all throughout my life. A very special thanks to Prof. Andrew Miner for his constant support and guidance. I learnt a lot from him and would like to thank him from all my heart. I would also like to take this opportunity to thank Junaid Babar and my committee members, Prof. Samik Basu and Prof. Robyn Lutz for their help and support.

Thanks to my siblings for their unconditional love that has helped me succeed at every step. Finally, I would like to thank my friend Shantanu, who instilled in me the confidence that I am capable of doing anything I put my mind to.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Petri nets are a graph based formalism used for modelling concurrent systems. Binary Decision Diagrams or Multi-Valued Decision Diagrams can be used in the analysis of systems modelled by Petri nets. An encoding scheme is required to be able to map the Petri net state to decision diagram values. Various encodings like One-hot scheme, Logarithmic scheme and Mdd scheme exist for this purpose. This thesis compares the performance of the existing encodings based on time and space taken to represent and analyze the system modelled as Petri net. It also introduces and compares a new encoding scheme called k-hot encoding and shows a gradual improvement in performance of the scheme with increasing values of k. The process of analyzing properties like deadlock and starvation is explained and a comparison is made between the encoding schemes based on the time taken by each to analyze these properties.

CHAPTER 1. OVERVIEW

1.1 Introduction

The software or hardware systems used nowadays are mostly huge and complicated. Thus, modelling and analyzing such systems has become complicated too. A representation for such a system can be used to evaluate the possible states the system can be in. One of the problems with representing such systems is the possibility of exponential growth in the number of states with the size of the system. This problem is well known as the *State Explosion* problem. The state explosion problem restricts huge concurrent systems from being represented and thus analyzing them or verifying properties for such system has become an issue. A lot of research work has been done in this context to make it possible to represent bigger and bigger systems efficiently and in a compact manner.

Efficiently modelling a system requires taking less space and time in representing the system. Decision Diagrams are used as they are capable of representing large systems in the form of compact data structures. In order to efficiently model concurrent systems, firstly a compact representation is chosen, secondly a good encoding scheme for the representation is selected and lastly a good enough ordering for the encoding variables is used.

The contributions of this thesis are as follows:

- This thesis evaluates and compares the existing encoding schemes(for Petri nets) namely One-hot scheme, Logarithmic scheme and MDD scheme based on their performance. The metrics used for evaluation are time and space taken to represent and analyze well known concurrent systems.

- A new encoding scheme called k-hot encoding is introduced which extends the One-hot and performs better with every increasing value of k.
- A comparison is made between the existing schemes and the new k-hot scheme using the same time and space metrics.
- Some of the systems are analyzed for properties like deadlock and starvation. This analysis is done using the existing and new schemes. A comparison is made between the time taken by each encoding scheme to analyze the given properties on the system.

Choosing the right encoding scheme is important for efficiently modelling a system, hence this evaluation helps in deciding the appropriate scheme for a system.

The remainder of this thesis is organized as follows. Chapter 2 of this thesis gives an introduction to the basic concepts like Decision Diagrams, Petri nets and Model checking. Chapter 3 explains in detail the existing encoding schemes. Towards the end, this chapter introduces a new encoding scheme and explains it in detail with the help of the Fork-join Petri net model. Chapter 4 compares the existing and proposed encoding schemes based on several metrics and lists the advantages and disadvantages of each scheme. The results of this comparison conclude when to use each encoding scheme. Chapter 5 summarizes the results of this thesis and describes some of the open problems in this context.

CHAPTER 2. REVIEW OF LITERATURE

2.1 Overview

This chapter introduces some of the key concepts related to modelling concurrent systems. Decision Diagrams have been popular data structures for representing huge concurrent systems. They generally provide a compact representation for sets of states and are discussed in detail in Section 2.2. Decision Diagrams can be further narrowed down to Binary Decision Diagrams, Multi Valued Decision Diagrams or more. Some of these types are also dicussed in this section. Section 2.3 introduces Petri Nets, which provide a front end used for modelling and analyzing concurrent systems. Lastly, Section 2.5 introduces Symbolic Model Checking [17]. Model checking refers to the process of verifying if a given model of the system satisfies a system property or not. Traditionally, the process of model checking consists of building the state graph and using temporal logic to verify a system property for that graph. In Symbolic Model Checking, BDD's are used for representing the state graph of the system.

2.2 Decision Diagrams

Decision Diagrams(DD) are used for representing points of decision and their possible results. It is possible to calculate the probability of an outcome/result using a DD [7]. They can be used for calculating and comparing the total cost for various alternative decisions and thus help in decision making [7]. Some important and popular types of Decision Diagrams are discussed below.

2.2.1 Binary Decision Diagrams

Binary Decision Diagrams(BDD's) [5, 9, 10] are directed acyclic graphs (DAGs) which represent a boolean function over a finite set of boolean variables $\{0,1\}^l \rightarrow \{0,1\}$. The vertices of the DAG are called nodes. These nodes could be terminal or non-terminal. The non-terminal nodes represent variables of the function represented by the BDD. Each non-terminal node of a BDD has exactly two children. A terminal node has no children and represents the values 0/false or 1/true.

To reach the right child denoted by $n(0)$, of a non-terminal node n, the value 0 or false is assigned to the corresponding variable of node n. Similarly, to reach the left child denoted by $n(1)$, the value 1 or true is assigned to the corresponding variable of node n. To evaluate the value of the function represented by the BDD, the DAG is traversed such that for every non-terminal node n, if the value of its corresponding variable is true, then $n(1)$ is evaluated next, else $n(0)$ is evaluated. On reaching the terminal node, the value of the terminal node is the value of the function with the given variable assignments.

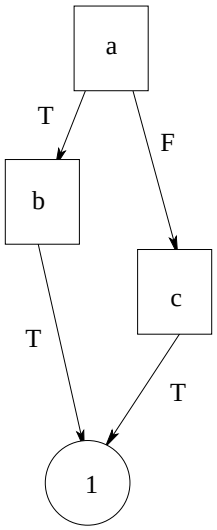


Figure 2.1 BDD representing function $f = ab' + a'bc$

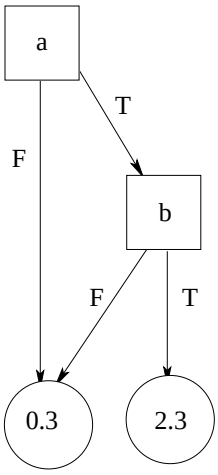
A BDD can be ordered by ordering its variables such that nodes occur according to the ordering in every path through the BDD. All non-terminal nodes in an ordered Decision Diagram representing variable x_k are said to belong to level k [23]. All terminal nodes belong to the level 0. The level of a non-terminal node is always greater than its child nodes. A *Reduced Ordered BDD*(ROBDD), is formed by reducing an ordered BDD as per the reduction rules of *Elimination* and *Redundancy*. The Elimination rule involves eliminating duplicate nodes. Two terminal nodes are duplicates if they have the same value. Two non-terminal nodes u and v are duplicates if they are labelled with the same variable and $u(0) = v(0)$ and $u(1) = v(1)$. The Redundancy rule eliminates redundant nodes. A non-terminal node u is redundant if $u(0) = u(1)$. This reduction process helps to reduce the size of the BDD's. This property is important as functionally equivalent ROBDD's are unique if the variable ordering is fixed [12]. Due to this canonical nature, verifying functional equivalence is reduced to verifying if the corresponding ROBDD's(with same variable order) are isomorphic or not.

A BDD can be *fully-reduced* or *quasi-reduced*. A BDD is fully-reduced if both reduction rules of *Elimination* and *Redundancy* are applied. On the other hand, quasi-reduction requires only the reduction rule of *Elimination*. Presence of all redundant nodes is required for quasi-reduction as otherwise there can be more than one way to quasi-reduce a BDD, thus no longer leaving the representation unique.

2.2.2 Algebraic Decision Diagrams

Algebraic Decision Diagrams(ADD's) [2] are also directed acyclic graphs representing functions. Unlike BDD's, ADD's represent functions of the form $\{0,1\}^l \rightarrow \mathbb{R}$ where $\mathbb{R}$ is the set of real numbers. They are also known as Multi Terminal BDD's.

Unlike a BDD, the ADD in Figure 2.2 has terminal values as real numbers. Thus ADD's basically extend BDD's by representing functions which can take a real value and not just 0 or 1. The number of terminal nodes in an ADD is equal to the number of distinct values which



a	b	f
T	T	2.3
T	F	0.3
F	T	0.3
F	F	0.3

Figure 2.2 Function f defined using a truth table on the right and ADD representing this function on the left

the function represented by the ADD can take.

Like BDD's, ADD's can be ordered by ordering their variables such that nodes occur according to the ordering in every path through the ADD. The reduction rules of *Elimination* and *redundancy* apply to ADD's as well. Functionally equivalent Reduced Ordered ADD's with fixed variable ordering are unique. Thus if two Reduced Ordered ADD's are isomorphic, then it implies that they are functionally equivalent [25].

2.2.3 Multi Valued Decision Diagrams

Decision Diagrams can be further generalized to represent multi-valued input functions of the form $\mathcal{X} \rightarrow \{0,1\}$ where $\mathcal{X}$ is the cross product $\mathcal{X} = \mathcal{X}_L \times \ldots \times \mathcal{X}_1$ of L finite sets and each $\mathcal{X}_k$, for $L \geq k \geq 1$, is of the form $\mathcal{X}_k = \{0, 1, \ldots, n_k - 1\}$. *Multi Valued Decision Diagrams*(MDD's) [14, 15] are directed acyclic graphs which can represent such functions. In MDD's, terminal nodes represent values 0/1 and non-terminal nodes have an integral number of children. A non-terminal node n representing variable x_k contains a vector of n_k pointers to other nodes(children). Figure 2.3 shows a Multi Valued Decision Diagram. In this MDD, the

non-terminal node x_4 points to the same child node for values 1 and 2, $x_4(1) = x_4(2)$. This is shown in Figure 2.3 as an arrow labelled with $1, 2$ from node x_4 pointing to the child node x_3.

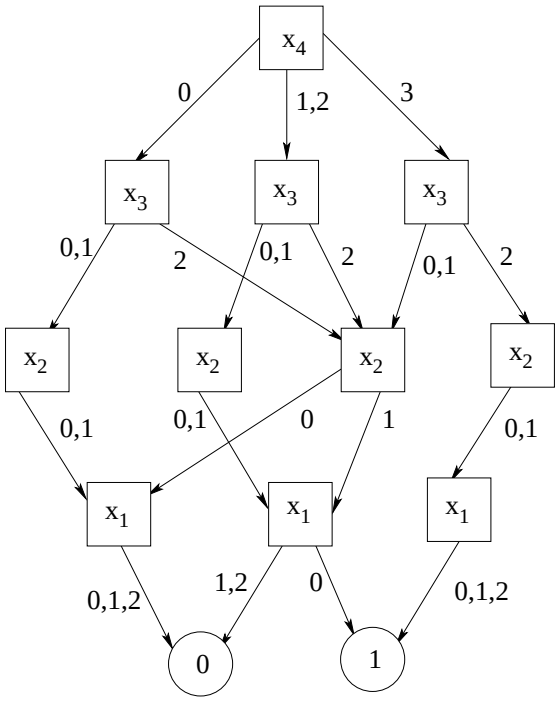


Figure 2.3 Example of an MDD

Two level k nodes u and v are said to be duplicates [14] if $u(i) = v(i)$ for all $0 \leq i \leq n_k$. A level k node u, is said to be redundant [14] if $u(i) = u(0)$ for all $0 \leq i \leq n_k$.

Like BDD's, MDD's are said to be ordered if in every path through the MDD, the nodes appear as per the ordering. Like BDD's, MDD's can be *fully-reduced* or *quasi-reduced*. Full reduction comprises of both reduction rules of *Elimination* and *Redundancy*. Quasi reduction on the other hand includes the *Elimination* reduction rule only. An MTMDD (Multi Terminal Multi Way Decision Diagram) [24] can be used to represent multi-valued input multi-valued output functions. In MTMDD, non-terminal nodes can point to an integral number of children at a level below and also there can be integral number of non-terminal nodes. Truth table for function f and its MTMDD representation is shown in Figure 2.5.

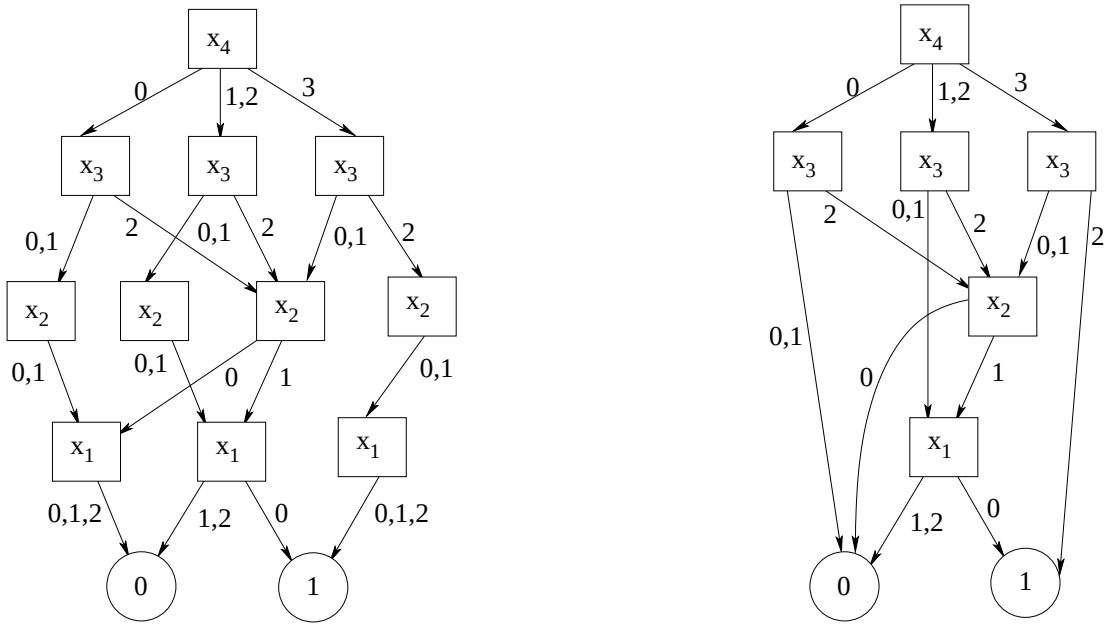


Figure 2.4 MDD on the left shows the quasi-reduced form of the MDD in Figure 2.3 and the one on the right shows its fully-reduced form.

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	2
2	0	2
2	1	2

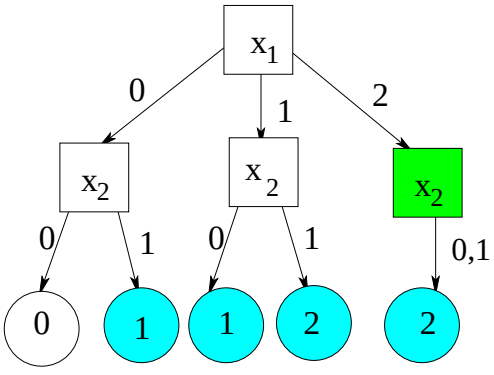


Figure 2.5 Truth table for function f on the left and its MTMDD representation on the right.

To fully reduce this MTMDD, the duplicate and redundant nodes are eliminated. Duplicate nodes are the terminal nodes with value 1 and 2(shaded terminal nodes). Removing these nodes, reduces the MTMDD to its quasi reduced form. In the quasi reduced MTMDD shown on the left in Figure 2.6, the right most node at level x_2 is a redundant node(shaded non terminal node) as it has identical children. Removing this node, reduces the quasi reduced MTMDD further to its fully reduced form as shown on the right in Figure 2.6.

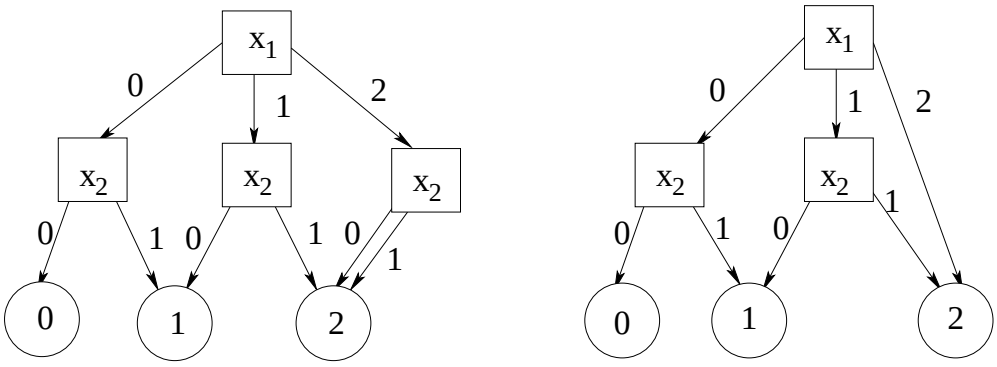


Figure 2.6 MTMDD on the left shows the quasi-reduced form of the MTMDD in Figure 2.5 and the one on the right shows its fully-reduced form.

2.2.4 Matrix Diagrams

Matrix Diagrams(MxDs) [18, 24] are used to encode functions of the form $(\mathcal{X} \times \mathcal{X}) \rightarrow \{0, 1\}$($\mathcal{X}$ is described in Section 2.2.3). In modelling a sytem, the variables used to represent the current state are called *unprimed* variables or from variables. Variables used to represent the next states are known as *primed* or to variables. Matrix diagrams can represent both primed and unprimed variables.

In an MxD, a non-terminal node n at level k encoding variables x_k and x'_k has $(n_k \times n_k)$ edges (pointers to other nodes) pointing to the level below, such that $n(a, a')$ points to the node that can be reached when $x_k = a$ and $x'_k = a'$. Identity patterns often occur in asynchronous

systems. A node n is an identity node if,

$$n\left(a, a'\right) = \begin{cases} 0, & \text{if } a \neq a' \\ n(0,0), & \text{otherwise} \end{cases} \tag{2.1}$$

Clearly, an Identity node reduction captures *no change* and *no dependency* behavior of the variable represented by the node. An MDD with $2k$ levels, where k is the number of variables, can be used to represent the same function as represented by an MxD. The benefit of MxD over MDD in this case, lies in the fact that MxDs are able to exploit identity patterns. Figure 2.7 shows an MxD with identity pattern on the left and its identity reduced version on the right. The middle node at level x_1(MxD on the left in Figure 2.7) is an identity node. Removing this identity node, generates the MxD on the right in Figure 2.7. As shown in this figure, node x_2 can reach node x_1 if $x_2 = 0$ and $x_2' = 2$(shown as label $(0, 2)$ on the arrow pointing from x_2 to x_1).

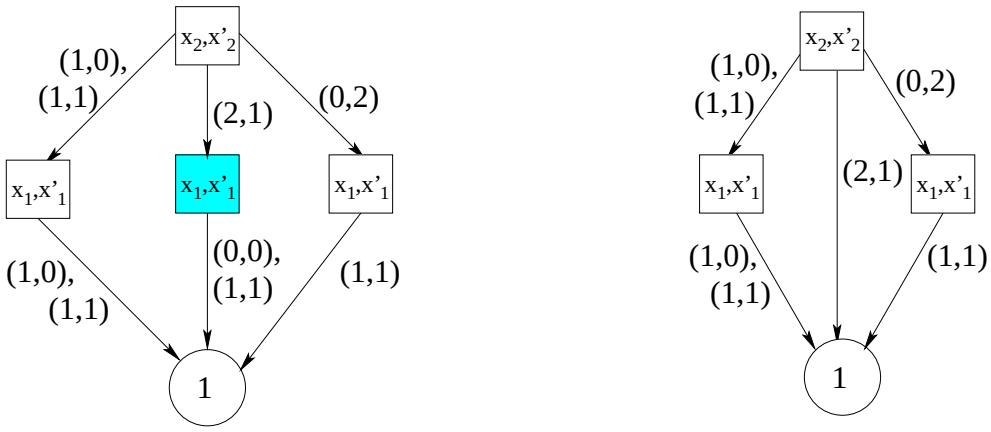


Figure 2.7 Identity reduced MxD [14]

An MxD can be either *fully-reduced*, *quasi-reduced* or *identity-reduced*. A fully-reduced MxD has no duplicate and redundant nodes; a quasi-reduced MxD has no duplicate nodes and an identity-reduced MxD has no identity and duplicate nodes.

2.2.5 Operations on DDs

Some of the operations that can be performed on DD's are AND, OR, NOT etc [20]. The algorithm described in this section can be used for many useful operations on MTMDD's. Hence this algorithm also works for operations on BDD's, MDD's and MTBDD's. It also works for MxD's with some small modifications.

Given two decision diagrams B_1(representing function f_1) and B_2(representing function f_2) and an operator $\oplus$ to be applied to these DD's, the root nodes v_1 and v_2 of the two DD's are passed to the funtion Apply shown in Algorithm 2.8 which returns the root node of the result DD(representing function f such that, $f = f_1 \oplus f_2$).

Each node v of a DD(over l variables) has:

- $v(j)$: If v is a non-terminal node such that value of the variable represented by v is j, then v points to $v(j)$
- level: represents level of v, could be 0 through l(terminal nodes have level 0)
- val: value of the node v if v is a terminal node

The algorithm can be broken down into 3 cases. The first case includes the trivial case where both v_1 and v_2 are terminal nodes(lines 2-4) and the operator specific cases(line 5). If both v_1 and v_2 are terminal nodes, the result DD consists of a terminal node with value $val(v_1) \oplus val(v_2)$. Few operator specific cases are $(1, \vee, v_2)$ and $(1, \wedge, v_2)$. In the process of computing operation $\oplus$ on v_1 and v_2, the value of a subgraph may be used more than once. In order to not evaluate the value of the subgraph more than once, a compute table T is maintained. An entry of set T is used to store the value computed by Apply on each node pair. The second case of the algorithm checks to see if there exists some u such that $(v_1, \oplus, v_2, u) \in T$(lines 6-8). In that case u is returned. The last case is when both nodes are non-terminal. If they are at the same level k, then a new node u is created at level k and the Apply procedure is recursively applied on all child nodes $v_1(j)$ and $v_2(j)$ to obtain $u(j)$ where $0 \leq j \leq n_k - 1$.

```
node Apply(node v1, operator ⊕, node v2)
 1: k := Max(level(v1), level(v2));
 2: if k == 0 then
 3:    return terminal node with value val(v1) ⊕ val(v2);
 4: end if
 5: check for operator specific cases eg. 1 ∨ x = 1, 1 ∧ x = x;
 6: if ∃ u such that (v1, ⊕, v2, u) ∈ T then
 7:    return(u);
 8: end if
 9: u := new level-k node;
10: for j = 0 to n_k − 1 do
11:    if level(v1) == level(u) then
12:       c1 := v1(j);
13:    else
14:       c1 := v1;
15:    end if
16:    if level(v2) == level(u) then
17:       c2 := v2(j);
18:    else
19:       c2 := v2;
20:    end if
21:    u(j) := Apply(c1, ⊕, c2);
22: end for
23: Reduce(u);
24: add (v1, ⊕, v2, u) to T;
25: return(u);
```

Figure 2.8 Apply method to compute $v_1 \oplus v_2$

Else if $level(v_1) = k$ and ($level(v_2) < k$ or v_2 is a terminal node), a new node u is created at level k and the Apply function is recursively applied on $v_1(j)$ and v_2 to obtain $u(j)$ for all $0 \leq j \leq n_k - 1$. The node u is then reduced(eliminating duplicate and redundant nodes) by calling the method Reduce. An entry $(v_1, \oplus, v_2, u)$ is added to T and u is returned.

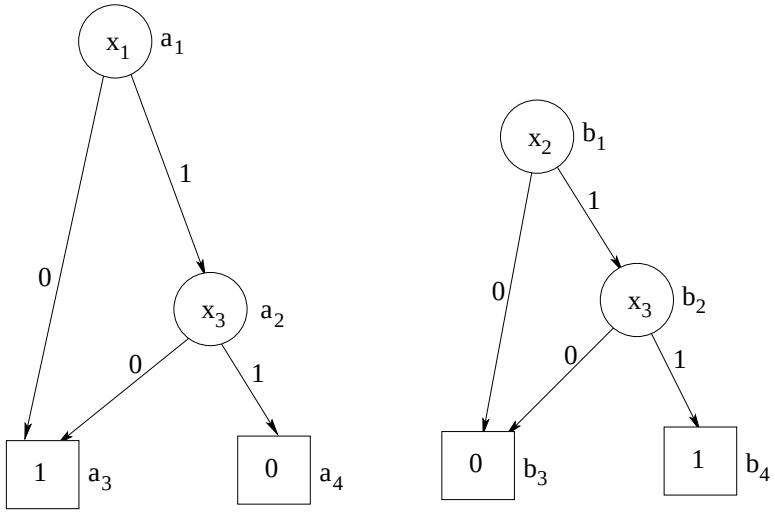


Figure 2.9 BDD for $\neg(x_1 \wedge x_3)$ on the left and $x_2 \wedge x_3$ on the right [4]

Figure 2.9 shows two BDD's B_1 and B_2 where B_1 represents function $f_1 = \neg(x_1 \wedge x_3)$ and B_2 represents function $f_2 = x_2 \wedge x_3$. Algorithm 2.8 is used compute the BDD for function f given by,

$$f(x_3, \ldots, x_1) = f_1(x_3, \ldots, x_1) \vee f_2(x_3, \ldots, x_1) \tag{2.2}$$

The function Apply(a_1, $\vee$, b_1) is called and it generates the BDD representing function f shown in Figure 2.10.

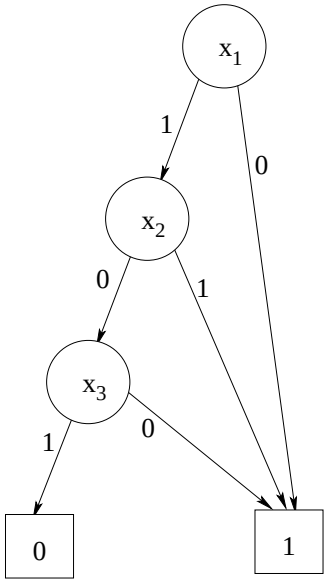


Figure 2.10 BDD for $\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$[4]

2.3 Petri Nets

A *Petri Net* [21, 25] is a weighted bipartite graph used to describe and model concurrent systems. It is generally used to model applications in which there is a flow of resources or concurrent behavior needs to be handled. It consists of places and transitions, and arcs are drawn pointing from a place to a transition and vice versa. A place in a Petri Net represents conditions. Transitions represent events that may occur. When the event represented by a transition occurs, the transition is said to have fired. An input arc from a place p to a transition t shows that the event represented by t requires the condition encoded in p. In this case, p is called the input place for t. Similarly, an output arc from t to p shows that the condition in p is generated as a result of the occurrence of event represented by t and in this case, p is called the output place for t. Every arc is assigned a weight, which is a non-negative integer used to determine the firing conditions of a transition and the token flow between the set of input and output places for that transition. A Marking $m \in \mathcal{M}$ of a Petri net is a mapping $m : \mathcal{P} \rightarrow \mathbb{N}$ where $\mathcal{P}$ is the set of places of the net and $\mathbb{N}$ is the set of naturals, representing the number of tokens at each place of the net. $\mathcal{M}$ is the set of all possible markings of the net.

A Petri net is said to be *1-bounded* or *safe* if each place in the initial marking of the net has a maximum of one token. A b-bounded PN has a maximum of b tokens at each place. Figure 2.11 shows a safe Petri net.

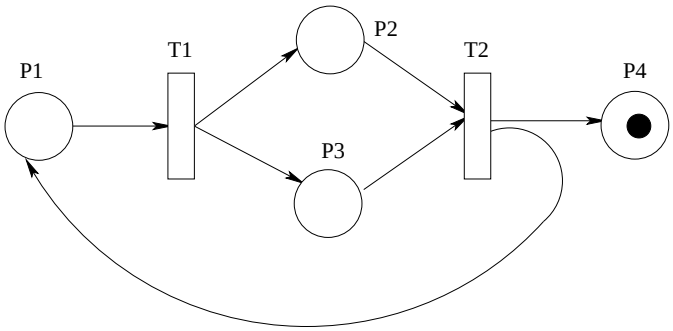


Figure 2.11 Petri Net

Petri nets can have inhibitor arcs. A transition having an inhibitor arc along with normal arcs, is enabled if all input places connected to the normal arcs have at least as many tokens as the weight of the arcs and all input places connected to the inhibitor arc do not have number of tokens greater or equal to the weight of the arc. If the weight of an inhibitor arc is ∞, then number of tokens in the input place cannot inihibit the transition. Figure 2.12 shows an example of Petri Net having both normal and inhibitor arcs. Inhibitor arcs are shown as arcs with circles.

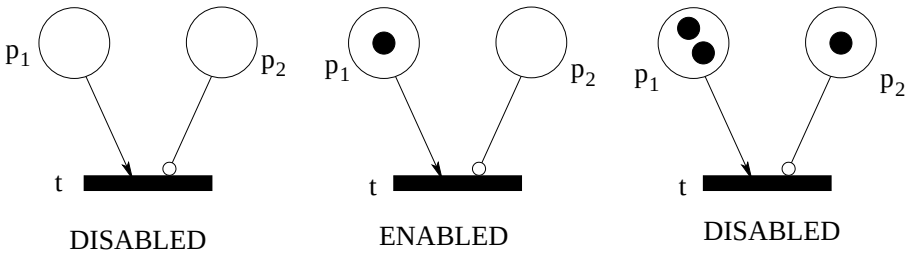


Figure 2.12 Petri Net with inhibitor arcs

A Petri Net PN, is represented as

$$PN = (\mathcal{P}, \mathcal{T}, \mathcal{W}, \mathcal{H}, M_0) \tag{2.3}$$

where,

- $\mathcal{P}$ is a finite set of places
- $\mathcal{T}$ is a finite set of transitions
- $\mathcal{W} : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \rightarrow \mathbb{N}$ is a multiset of arcs such that each arc is assigned a non-negative integer called the weight of the arc
- $\mathcal{H} : (\mathcal{P} \times \mathcal{T}) \rightarrow \mathbb{N} \cup \{\infty\}$ is a multiset of inhibitor arcs such that each inhibitor arc is assigned the weight of the arc
- M_0: $\mathcal{P} \rightarrow \mathbb{N}$ is the initial marking

Transitions cause the Petri net model to change state. A state is represented by marking the places with tokens. A transition $t \in \mathcal{T}$ is *enabled* in a marking m if for all places p, $m(p) \geq \mathcal{W}(p,t)$ and $m(p) < \mathcal{H}(p,t)$. This is called the enabling expression for transition t.

Multiple transitions can be enabled at the same time. An enabled Petri net transition may fire. This makes firing of transitions in Petri nets non-deterministic. On firing, a transition produces a new marking denoted by m'. The firing expression for transition t is,

$$\forall p \in \mathcal{P} : m'(p) = m(p) + \mathcal{W}(t,p) - \mathcal{W}(p,t) \tag{2.4}$$

Thus it removes tokens from each input place and adds tokens to each output place of the transition. The number of tokens removed is equal to the weight of the corresponding input arc and similarly, the number of tokens added is equal to the weight of the corresponding output arc. A marking m' is *reachable* from marking m if there is a sequence of firings from m that lead to m', denoted by $m \rightarrow m'$. All markings reachable from the initial marking form the

reachability set.

The transition function δ defines the new marking of the Petri net after an enabled transition fires. It transforms a set of markings $\mathcal{M}_1$ to a new set of markings $\mathcal{M}_2$ where $\mathcal{M}$ is the set of all markings.

$$\delta : 2^{\mathcal{M}} \times \mathcal{T} \rightarrow 2^{\mathcal{M}} \tag{2.5}$$

For a transition $t \in \mathcal{T}$,

$$\delta(\mathcal{M}_1, t) = \mathcal{M}_2 = \{m_2 \in \mathcal{M} : \exists m_1 \in \mathcal{M}_1,\ t \text{ is enabled in } m_1 \text{ and firing } t \text{ leads from } m_1 \text{ to } m_2\} \tag{2.6}$$

This set $\mathcal{M}_2$ is called the post-image of $\mathcal{M}_1$. The inverse image computation or pre-image computation results in a set of markings from which the current marking can be obtained. A *transition relation* contains pairs (m, m') where m' is reachable from m by firing a single transition, once(there is some t such that m enables t, and when t fires in m, produces new marking m').

A marking $m \in \mathcal{M}$ of a safe Petri net with n places can be represented using an encoding function which provides a binary mapping from $\mathcal{M} \rightarrow \mathcal{B}^n(\mathcal{B} = \{0, 1\})$, where the image of m is encoded into an element $(e_1, e_2, e_3, e_4, \ldots, e_n) \in \mathcal{B}^n$ such that $e_i = m(p_i)$. Using this representation a marking m such that $m(p_1) = 0$ and $m(p_2) = 1$ is encoded as [01].

This binary mapping allows using Binary Decision Diagrams to represent sets of markings. For instance a Petri net marking set $\mathcal{M} = \{m_1, m_2, m_3\}$ where $m_1(p_1) = 1$, $m_1(p_2) = 0$, $m_1(p_3) = 0$; $m_2(p_1) = 0$, $m_2(p_2) = 1$, $m_2(p_3) = 1$ and $m_3(p_1) = 1$, $m_3(p_2) = 0$, $m_3(p_3) = 1$ is represented using a BDD in Figure 2.13. The marking m_1 can be encoded as [100], m_2 can be encoded as [011] and m_3 can be encoded as [101] using the representation discussed earlier. The combinations of p_1, p_2, p_3 which belong to the set $\mathcal{M}$, point to the terminal node 1, and

all the other combinations point to the terminal 0.

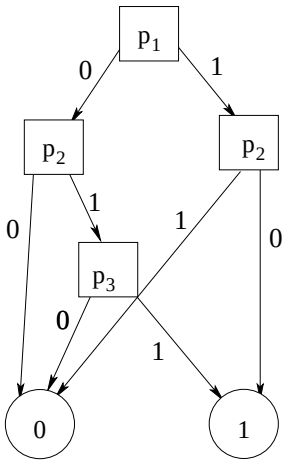


Figure 2.13 BDD for the marking set $\mathcal{M} = \{[100], [011], [101]\}$

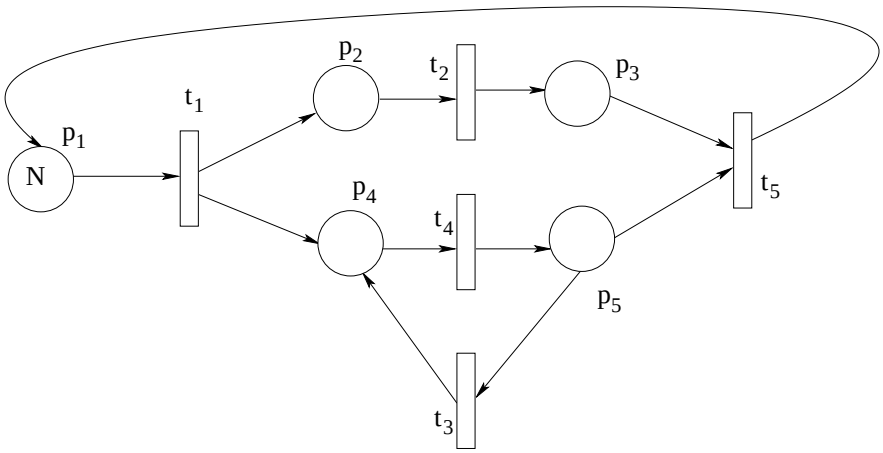


Figure 2.14 Petri net for Forkjoin Model

The forkjoin Petri net model is shown in Figure 2.14. It shows the forking of one main thread to create other threads that execute in parallel and then finally the joining of these forked threads back to the main thread. This Petri net shows a very common model of concurrent execution where the main process creates parallel executing sub-processes and waits for them to be finished before resuming execution.

The Forkjoin Petri net consists of 5 places(p_1, p_2, p_3, p_4 and p_5) and 5 transitions(t_1, t_2, t_3, t_4 and t_5). Each place has a bound of N. The initial state of this Petri net has N tokens in place p_1 which represents the main thread and 0 tokens at all other places. Transition t_1 forks the main thread p_1 to sub-threads and transition t_5 joins the sub-threads back to the main thread p_1. Thus, a *fork* is a transition with more than one output places and a *join* is a transition with more than one input places. Figure 2.15 also shows the reachability set of this net with $N = 1$.

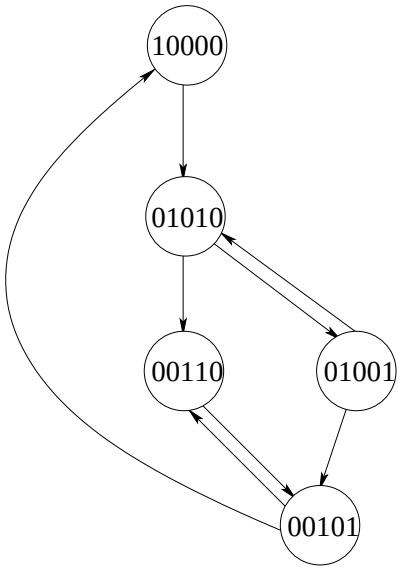


Figure 2.15 Reachability graph for Forkjoin model with $N = 1$

2.4 Building Transition relation and Reachability set

This section explains how the transition relation and reachability set for a Petri net model PN = ($\mathcal{P}$, $\mathcal{T}$, $\mathcal{W}$, $\mathcal{H}$, M_0) are built using the initial marking and the Petri net structure.

Algorithm 2.16 shows how enabling relation for a transition t is built. Enabling relation for t represents the possible current states in which t is enabled. It is built using the enabling expression for t. Hence, enabling relation is the relation for expression $\forall p \in \mathcal{P} : m(p) \geq \mathcal{W}(p,t)$ $\wedge\ m(p) < \mathcal{H}(p,t)$

Relation enablingRelation(transition t)
1: $V := \mathcal{M} \times \mathcal{M}$;
2: **for all** $p \in P$ **do**
3: $\quad R := \{(m, m') \in \mathcal{M} \times \mathcal{M} : m(p) \geq W(p,t)\}$;
4: $\quad V := V \cap R$;
5: $\quad R := \{(m, m') \in \mathcal{M} \times \mathcal{M} : m(p) < H(p,t)\}$;
6: $\quad V := V \cap R$;
7: **end for**
8: return(V);

Figure 2.16 Building enabling relation

Similarly, Algorithm 2.17 shows how firing relation for a transition is built. Firing relation for t represents the possible next reachable states when t is fired. It is built using the firing expression for t. Hence, firing relation is the relation for expression $\forall p \in \mathcal{P} : m'(p) = m(p) + \mathcal{W}(t,p) - \mathcal{W}(p,t)$.

Relation firingRelation(transition t)

1: $V :=$ Relation for $\mathcal{M} \times \mathcal{M}$;
2: **for all** $p \in P$ **do**
3: $\quad R := \{(m, m') \in \mathcal{M} \times \mathcal{M} : m'(p) = m(p) + W(t,p) - W(p,t)\}$;
4: $\quad V := V \cap R$;
5: **end for**
6: return(V);

Figure 2.17 Building firing relation

Taking the intersection of the two relations(enabling and firing) produces the transition relation for t. Algorithm 2.18 shows how the complete transition relation is built by taking the union of the relations for all transitions(built using Algorithms 2.16 and 2.17).

Relation buildTransitionRelation()

1: $Q := \emptyset$;
2: **for all** $t \in T$ **do**
3: $\quad F :=$ enablingRelation(t) $\cap$ firingRelation(t);
4: $\quad Q := Q \cup F$;
5: **end for**
6: return(Q);

Figure 2.18 Building overall Transition relation

Algorithm 2.19 shows how the Reachability set is built using the initial DD and Transition relation(obtained from Algorithm 2.18). The reachabilitySet DD initially consists of the initial marking. Next the post image operator is applied on the reachabilitySet to find the next reachable states denoted by postImage DD. Then, the union of postImage DD and reachabilitySet is done. This process of finding the post image and its union with the reachabilitySet is repeated until the post image operation does not create any new markings to be added to the reachability set.

```
DD BuildReachabilitySet(Relation T, DD M_0)
1: R := M_0;
2: P := ∅;
3: while P ≠ R do
4:     P := R;
5:     I := PostImageOperator(T, R);
6:     R := R ∪ I;
7: end while
8: return(R);
```

Figure 2.19 Algorithm for building Reachability Decision Diagram

The Petri net in Figure 2.14 has 5 transitions. The enabling and firing expressions for transition t_1 are $(m(p_1) \geq 1)$ and $(m'(p_1) == m(p_1) - 1 \wedge m'(p_2) == m(p_2) + 1 \wedge m'(p_4) == m(p_4) + 1)$. The enabling, firing and transition MxD for t_1 are shown in Figure 2.20.

2.5 Model Checking

Model Checking [3, 11] is an automatic procedure for verifying a specification over a given model. This process involves three steps. Firstly, Kripke structures or finite automata are used to formally model a system. Secondly, the specification is written using either classical or temporal logic. Specification in this context means a propositional logic formula which could be a safety property like no deadlock or liveness. Lastly, it is verified if the model satisfies the specification or not. A model checker takes as input, a finite state model and a specification expressed as a temporal logic formula and returns either a true result or an execution showing why the specification is not satisfied by the model. Due to the size of the model(Kripke structure/finite automata), model checkers also suffer from the state explosion problem as they need to verify the specification amongst every execution of the model. Due to their compact structure, Reduced Ordered BDD's are often used for representing the transition relation allowing verification of much larger models.

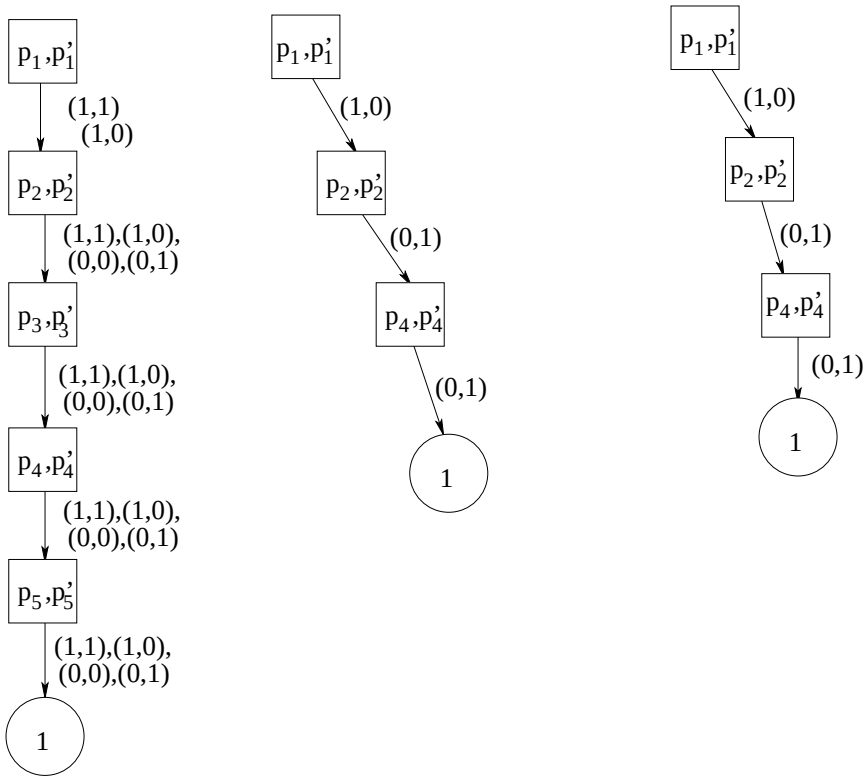


Figure 2.20 MxD for enabling expression on the left, firing expression in the middle and transition t on the right

Temporal logic in contrast with the classical logic, allows reasoning about the system behavior over time.

- The temporal operator G also called the Global operator, along with a proposition logic formula p as $\mathsf{G}p$, is true for a path if p holds at all states(point of time) along the path.
- The future temporal operator F when used with a propositional logic formula p as $\mathsf{F}p$, is true for a path if p holds at some state(points of time) along the path.
- The Until operator ($p\mathsf{U}q$) is true for a path if q holds at some state along the path, and p is true in all states before that state.
- The Next operator ($\mathsf{X}p$) is true for a path if p holds at the next state(points of time) along the path.

Computational Tree Logic also called CTL [6], is used to represent temporal logic. In CTL, a logical formula may consist of path quantifiers and temporal operators. Path quantifiers provided by CTL are, A: for all paths and E: there exists a path.

A CTL formula can be a state or a path formula. $M, s \models f$, denotes the state formula f holds in state s of model M. A state formula can be:

- An atomic proposition
- $p \wedge q$, $p \vee q$, $\neg\, p$, where p and q are state formulas
- $\mathsf{A}(g)$, $\mathsf{E}(g)$ where g is a path formula

A path formula can be $\mathsf{X}p$, $\mathsf{F}p$, $\mathsf{G}p$, $p\mathsf{U}q$, where p and q are state formulas. The CTL operators can be reduced to the set (EX, EU, EG) using the following equivalence,

- $\mathsf{AX}p = \neg\mathsf{EX}\neg p$
- $\mathsf{AF}p = \neg\mathsf{EG}\neg p$
- $\mathsf{EF}p = \mathsf{E}true\mathsf{U}p$
- $\mathsf{AG}p = \neg\mathsf{E}true\mathsf{U}\neg p$
- $\mathsf{A}p\mathsf{U}q = \neg\mathsf{E}[\neg q\mathsf{U}(\neg q \wedge \neg p)] \wedge \neg\mathsf{EG}\neg q$

Let p and q be propositional logic formulas, then

- $\mathsf{EX}p$: This formula is satisfied by a state s if for some path from s, the formula $\mathsf{X}p$ is satisfied.
- $\mathsf{E}p\mathsf{U}q$: This formula is satisfied by a state s if for some path from s, the formula $(p\mathsf{U}q)$ is satisfied.

- EGp: This formula is satisfied by a state s if for some path from s, the formula Gp is satisfied.

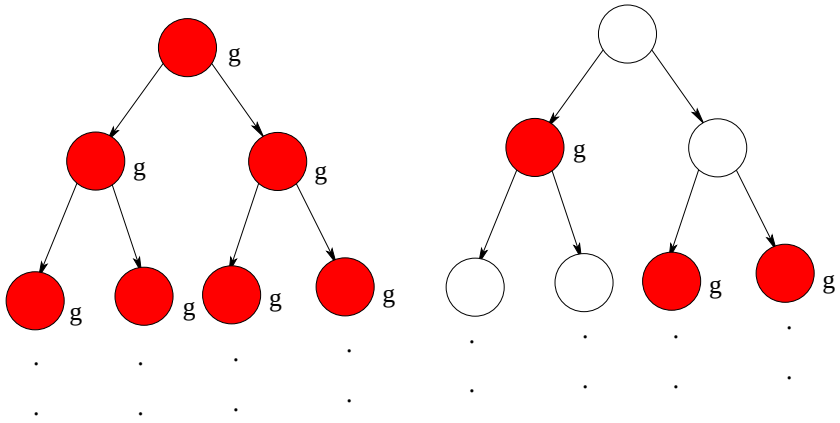


Figure 2.21 $M, s \models \mathsf{AG}g$ and $M, s \models \mathsf{AF}g$

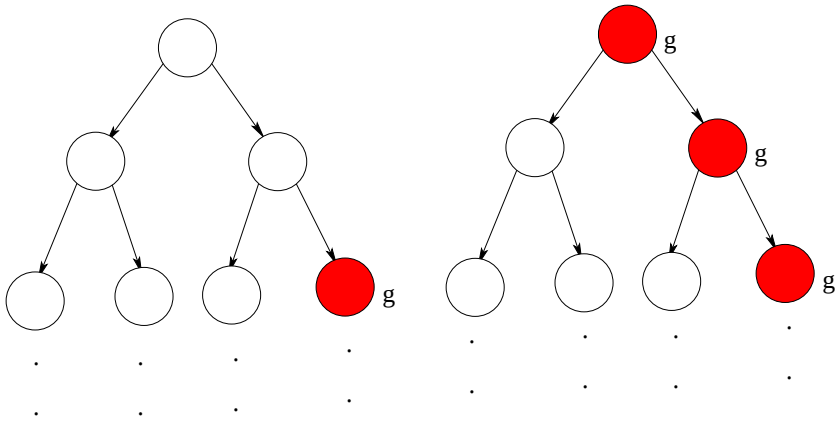


Figure 2.22 $M, s \models \mathsf{EF}g$ and $M, s \models \mathsf{EG}g$

Algorithms to implement the three basic CTL operators are shown. Figure 2.23 shows the algorithm for implementing the CTL formula EXp.

SetOfStates EX(LogicFormula p)
1: E := Satisfy(p);
2: T := Pre-Image(E);
3: return(T);

Figure 2.23 Algorithm for CTL formula $\mathsf{EX}p$

To implement the CTL formula $\mathsf{E}p\mathsf{U}q$ as shown in Figure 2.24, initially the set Y contains the set of states satisfying the logical formula q. Then any state that satisfies the logical formula p and can reach a state that satisfies $\mathsf{E}p\mathsf{U}q$, is added to the set until a fixed point is reached.

SetOfStates $\mathsf{E}p\mathsf{U}q$(LogicFormula p, LogicFormula q)
1: Y := Satisfy(q);
2: X := {};
3: Z := Satisfy(p);
4: **while** $Y \neq X$ **do**
5: $X := Y$;
6: $Y := Y \cup$ (Pre-Image(Y) $\cap$ Z);
7: **end while**
8: return(Y);

Figure 2.24 Algorithm for CTL formula $\mathsf{E}p\mathsf{U}q$

The algorithm for implementing the formula $\mathsf{EG}p$ is shown in Figure 2.25. The set of states satisfying $\mathsf{EG}p$ can be obtained by starting with the states satisfying the logical formula p. At each iteration, any state that cannot reach a state satisfying $\mathsf{EG}p$ is removed from the set. This iterative process continues until a fixed point is reached.

CTL model checking can be used to verify properties in systems modelled as Petri nets. It requires the generation of reachability graph, on which the model checking procedure is then applied. Figure 2.14 showed the Forkjoin Petri net model. Figure 2.26 shows the reachability graph of this petri net with $N = 9$.

SetOfStates $\mathsf{EG}p$(LogicFormula p)
1: $Y := \text{Satisfy}(p)$;
2: $X := \{\}$;
3: **while** $Y \neq X$ **do**
4: $\quad X := Y$;
5: $\quad Y := Y \cap \text{Pre-Image}(Y)$;
6: **end while**
7: return(Y);

Figure 2.25 Algorithm for CTL formula $\mathsf{EG}p$

Let f = (p_1 contains 8 tokens and p_2 contains 1 token and p_3 contains 0 tokens and p_4 contains 1 token and p_5 contains 0 token). Using CTL model checking, it is possible to check if the CTL formula $\mathsf{EX}(f)$ is satisfied or not. It can be seen from the reachability graph that the markings/states which satisfy the logical formula f are $\{81010\}$. Next, the states which satisfy the formula $\mathsf{EX}(f)$ form the set $\{90000\}$. As the initial state 90000 is included in this set, it is concluded that the given CTL formula is satisfied by the initial state. To evaluate the CTL formula $\mathsf{EG}(\#p_1 \geq 8)$ where $\#p_1$ represents number of tokens in p_1, all states satisfying $(\#p_1 \geq 8)$ are found. The set Y, satisfying $(\#p_1 \geq 8)$ is $\{90000, 81010, 80110, 81001, 80101\}$. Initially, the set X satisfying formula $\mathsf{EG}(\#p_1 \geq 8)$ is equal to set Y. Next, any state in Y that cannot reach a state in X needs to be removed. As every state in Y can reach a state in X, a fixed point has been reached and $\{90000, 81010, 80110, 81001, 80101\}$ is returned.

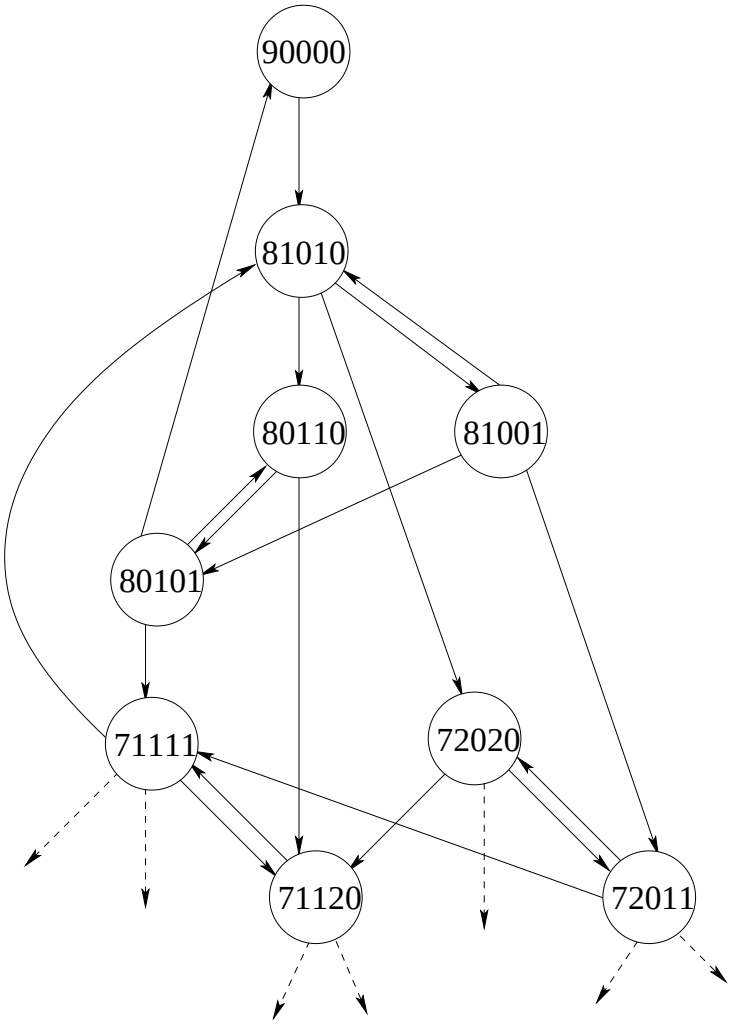


Figure 2.26 Reachability graph of the Petri net of Figure 2.14

CHAPTER 3. ENCODING SCHEMES

3.1 Overview

State encoding is a way to capture the state of a system. An encoding scheme basically maps the state of the system to decision diagram variable values. In case of Petri nets, capturing the state of the system would mean being able to capture the number of tokens at every place in the net at that time. Thus encoding a Petri net means encoding every place of the net using encoding variables. The size of the BDD can increase exponentially with increase in the number of encoding variables used to represent the Petri net model [16].

This Chapter explains the various encoding schemes and introduces a new encoding scheme called the k-hot encoding. Section 3.2 descibes the One-hot Encoding Scheme. Section 3.3 explains in detail the Logarithmic encoding. Native MDD encoding is explained in Section 3.4. Lastly, Section 3.5 introduces the new encoding scheme called k-hot encoding.

3.2 One-hot Encoding Scheme

Various encoding schemes have been proposed. The *One-hot encoding* scheme [15, 22] is extremely simple to implement. In an unsafe Petri net, a b-bounded place p uses b encoding variables under the One-hot scheme. Each of these b variables is used to encode the possibility of up to b tokens for that place. At most one of these b variables can be set at a time.

A marking m of a b-bounded Petri net with n places can be encoded using the One-hot scheme. Every place p of the Petri net, is encoded using b encoding variables, thus in total $n \cdot b$

encoding variables are needed to encode a marking m. $x_{p,i}$ represents value of the i^{th} encoding variable of place p.

$$x_{p,i} = \begin{cases} 1, & \text{if } m(p) = i \text{ where } 1 \leq i \leq b \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

To decode the marking at a place p(encoded using b boolean variables),

$$m(p) = \begin{cases} i, & \text{if there exists an } i \text{ such that } x_{p,i} = 1 \text{ where } 1 \leq i \leq b \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

Thus under One-hot encoding, an ADD/MTMDD for the function $f = m(p)$ is equivalent to DD(ADD/MTMDD) for the variable $x_{p,i}$ where $x_{p,i} = 1$. For safe Petri net models, this scheme uses the same number of encoding variables as the number of places in the safe Petri net. Thus this scheme is also known as one variable per place encoding scheme.

The Petri net in Figure 2.26 of Chapter 2 is a N-bounded Petri net. For $N = 9$, each place in this net can have a maximum of 9 tokens. As per the One-hot encoding scheme, each of these five places use nine variables to encode the number of tokens present. Table 3.1 shows the encoding for place p_1 of this Petri net. Thus the total number of encoding variables needed to encode a marking of this Petri net are 45.

$x_{p_1,1}$	$x_{p_1,2}$	$x_{p_1,3}$	$x_{p_1,4}$	$x_{p_1,5}$	$x_{p_1,6}$	$x_{p_1,7}$	$x_{p_1,8}$	$x_{p_1,9}$	$m(p_1)$
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	2
0	0	1	0	0	0	0	0	0	3
0	0	0	1	0	0	0	0	0	4
0	0	0	0	1	0	0	0	0	5
0	0	0	0	0	1	0	0	0	6
0	0	0	0	0	0	1	0	0	7
0	0	0	0	0	0	0	1	0	8
0	0	0	0	0	0	0	0	1	9

Table 3.1 Encoding for place p_1

As discussed in Chapter 2, transition function δ transforms a set of markings $\mathcal{M}_1$(also called the **from** state) to a new set of markings $\mathcal{M}_2$(also called the **to** state) such that for every marking $m_2 \in \mathcal{M}_2$, there exists a marking $m_1 \in M_1$ such that m_2 can be reached from m_1 on firing of a single transition which is enabled in m_1. For example, the transition relation transforms the marking [p_1:9, p_2:0, p_3:0, p_4:0, p_5:0] to marking [p'_1:8, p'_2:1, p'_3:0, p'_4:1, p'_5:0].

The transition function of Petri Net in Figure 2.26 can be represented as a function over 90 variables ($x_{p_1,1}$, $x_{p_1,2}$, ..., $x_{p_5,9}$, $x'_{p_1,1}$, $x'_{p_1,2}$, ..., $x'_{p_5,9}$), where the function evaluates to 1 if ($x'_{p_1,1}$, $x'_{p_1,2}$, ..., $x'_{p_5,9}$) can be reached from ($x_{p_1,1}$, $x_{p_1,2}$, ..., $x_{p_5,9}$) in one step. The enabling decision diagram of Algorithm 2.18 is is a graph over the 45 unprimed variables and the firing DD is a graph over the 45 primed variables. The reachability set of this Petri Net consists of 385 markings.

To encode the transition relation of a Petri net, both from and to states are encoded using the chosen encoding scheme. Section 2.4 of Chapter 2 showed how transition relation is built using the enabling and firing expressions of the net. The enabling and firing decision diagrams are built using one-hot encoding and hence the resulting transition relation is encoded as shown in Table 3.2. The unprimed variables encode the from state of the Petri net [p_1:9, p_2:0, p_3:0,

p_4:0, p_5:0]. The next state represented by primed variables has the marking [p'_1:8, p'_2:1, p'_3:0, p'_4:1, p'_5:0].

from state to state	$m(p_1)$ $m(p'_1)$	$m(p_2)$ $m(p'_2)$	$m(p_3)$ $m(p'_3)$	$m(p_4)$ $m(p'_4)$	$m(p_5)$ $m(p'_5)$
p_1:9,p_2:0,p_3:0,p_4:0,p_5:0	000000001	000000000	000000000	000000000	000000000
p'_1:8,p'_2:1,p'_3:0,p'_4:1,p'_5:0	000000010	100000000	000000000	100000000	000000000
p_1:8,p_2:1,p_3:0,p_4:1,p_5:0	000000010	100000000	000000000	100000000	000000000
p'_1:7,p'_2:2,p'_3:0,p'_4:2,p'_5:0	000000100	010000000	000000000	010000000	000000000
p_1:8,p_2:1,p_3:0,p_4:1,p_5:0	000000010	100000000	000000000	100000000	000000000
p'_1:8,p'_2:0,p'_3:1,p'_4:1,p'_5:0	000000010	000000000	100000000	100000000	000000000
p_1:8,p_2:1,p_3:0,p_4:1,p_5:0	000000010	100000000	000000000	100000000	000000000
p'_1:8,p'_2:1,p'_3:0,p'_4:0,p'_5:1	000000010	100000000	000000000	000000000	100000000
p_1:7,p_2:2,p_3:0,p_4:2,p_5:0	000000100	010000000	000000000	010000000	000000000
p'_1:6,p'_2:3,p'_3:0,p'_4:3,p'_5:0	000001000	001000000	000000000	001000000	000000000

Table 3.2 Part of the transition relation for the Petri net shown in Figure 2.26 for $N = 9$

The efficiency of this scheme can be improved by interleaving the primed and unprimed variables. Instead of storing all unprimed variables of place p_1 followed by its primed variables, an interleaved encoding stores $x_{p_1,1}$, the first unprimed variable for p_1 followed by its corresponding prime variable $x'_{p_1,1}$; then the second unprimed variable of p_1 which is $x_{p_1,2}$, followed by its primed variable $x'_{p_1,2}$ and so on. This ordering usually produces a more compact BDD as compared to the normal ordering [20, 26].

3.3 Logarithmic encoding

The *Logarithmic encoding* [15, 22] scheme uses same or fewer encoding variables as compared to the One-hot. For an unsafe Petri net with up to b tokens at each place, the log-based scheme uses $\lceil log(b+1) \rceil$ encoding variables to encode the tokens at every place. $x_{p,i}$ represents value of the i^{th} encoding variable of place p.

For a marking m, the value of $m(p)$ can be encoded as

$$x_{p,i} = a_i \bmod 2 \tag{3.3}$$

where $a_0 = m(p)$, $a_i = \lfloor a_{i-1}/2 \rfloor$ and $0 \leq i \leq \lceil log(b+1) \rceil - 1$.

The above encoding can be decoded to obtain the number of tokens at place p,

$$m(p) = \sum_{i=0}^{\lceil log(b+1) \rceil - 1} x_{p,i} \cdot 2^i \tag{3.4}$$

In order to build ADD/MTMDD representing the function $f = m(p)$, the decison diagrams for constant 2^i and variable $x_{p,i}$ are built for all i, such that $0 \leq i \leq \lceil log(b+1) \rceil - 1$. Next for all i, the multiplication operator is applied on DD's for 2^i and $x_{p,i}$. Addition of all such DD's results in the graph representing function f.

The Forkjoin Petri net model shown in Figure 2.26 with $N = 9$ can also be encoded under the Logarithmic encoding scheme where each of the five places of the Petri net use four variables to encode the number of tokens present. Thus under this scheme the total number of encoding variables required are 20 as opposed to the 45 variables required under the One-hot scheme. Place p_1 can be encoded using four boolean variables $x_{p_1,0}$, $x_{p_1,1}$, $x_{p_1,2}$ and $x_{p_1,3}$ as shown in Table 3.3.

As explained earlier, encoding the transition relation requires encoding both from and to states of the relation. With the log-based scheme, the from state will need four unprimed encoding variables and the to state will need four primed encoding variables. The encoded transition relation is shown in Table 3.4.

$x_{p_1,0}$	$x_{p_1,1}$	$x_{p_1,2}$	$x_{p_1,3}$	$m(p_1)$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	2
1	1	0	0	3
0	0	1	0	4
1	0	1	0	5
0	1	1	0	6
1	1	1	0	7
0	0	0	1	8
1	0	0	1	9

Table 3.3 Place p_1 encoded using variables $x_{p_1,0}$, $x_{p_1,1}$, $x_{p_1,2}$ and $x_{p_1,3}$

Table 3.4 shows the transition from state [p_1:9, p_2:0, p_3:0, p_4:0, p_5:0] to state [p'_1:8, p'_2:1, p'_3:0, p'_4:1, p'_5:0] and from state [p_1:8, p_2:1, p_3:0, p_4:1, p_5:0] to state [p'_1:7, p'_2:2, p'_3:0, p'_4:2, p'_5:0] and so on. Interleaving the primed and unprimed variables can increase the efficiency of this encoding scheme as well.

3.4 Native MDD Encoding

Petri net models encoded using the native *MDD encoding* [15, 22], are represented using Multi Valued Decision Diagrams. The MDD encoding scheme uses a single encoding variable to encode the number of tokens present at a place in the Petri net. For a b-bounded Petri net model, the number of tokens at each place can be encoded using a single integer variable with value equal to the number of tokens at that place.

For a given BDD, its corresponding MDD can be built by grouping l BDD variables (binary variables) into a single MDD variable (multi-valued variable). The advantage of an MDD over its corresponding BDD is that, if the BDD requires q memory accesses and the MDD requires z memory accesses then $z \leq q \leq l \cdot z$ [16]. Figure 3.1 shows a BDD and its corresponding MDD for the function:

$$f(x, y) = 1 \text{ if } x > y \tag{3.5}$$

from state to state	$m(p_1)$ $m(p'_1)$	$m(p_2)$ $m(p'_2)$	$m(p_3)$ $m(p'_3)$	$m(p_4)$ $m(p'_4)$	$m(p_5)$ $m(p'_5)$
p_1:9,p_2:0,p_3:0,p_4:0,p_5	1001	0000	0000	0000	0000
p'_1:8,p'_2:1,p'_3:0,p'_4:1,p'_5	0001	1000	0000	1000	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	0001	1000	0000	1000	0000
p'_1:7,p'_2:2,p'_3:0,p'_4:2,p'_5	1110	0100	0000	0100	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	0001	1000	0000	1000	0000
p'_1:8,p'_2:0,p'_3:1,p'_4:1,p'_5	0001	0000	1000	1000	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	0001	1000	0000	1000	0000
p'_1:8,p'_2:1,p'_3:0,p'_4:0,p'_5	0001	1000	0000	0000	1000
p_1:7,p_2:2,p_3:0,p_4:2,p_5	1110	0100	0000	0100	0000
p'_1:6,p'_2:3,p'_3:0,p'_4:3,p'_5	0110	1100	0000	1100	0000

Table 3.4 Part of the Transition relation of Petri net in Figure 2.26 with $N = 9$

where $0 \leq x \leq 2$ and $0 \leq y \leq 2$. Note that the BDD uses 4 variables to represent the function as opposed to the 2 variables used by the corresponding MDD. Hence $q \leq 2z$, where q and z are the memory accesses required by BDD and MDD of Figure 3.1.

In order to build an MTMDD representing the function $f = m(p)$, a decision diagram having an edge from x_p to terminal node $m(p)$ is built. MDD encoding of a Petri net also involves encoding its initial state, transition relation and the reachability set. The initial state of the Petri net of Figure 2.26 with $N = 9$ is encoded as shown in Figure 3.2.

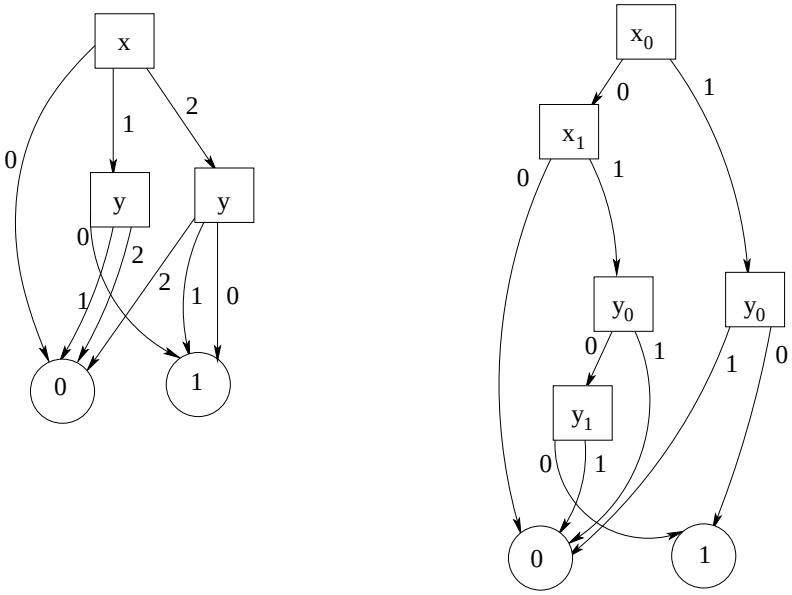


Figure 3.1 MDD for $x > y$ on the left and the corresponding BDD on the right

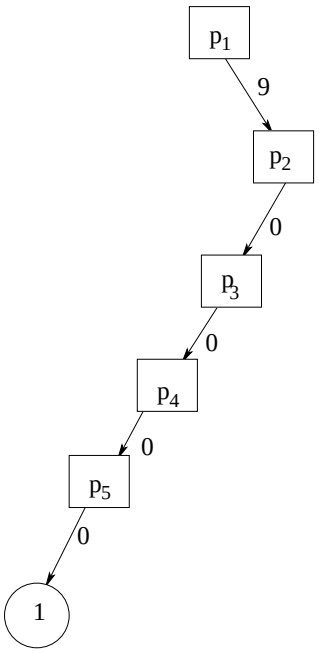


Figure 3.2 MDD representing the initial state of the Petri net shown in Figure 2.26 with $N = 9$

The transition relation of the Petri net of Figure 2.26 with $N = 9$ is encoded as shown in Figure 3.3.

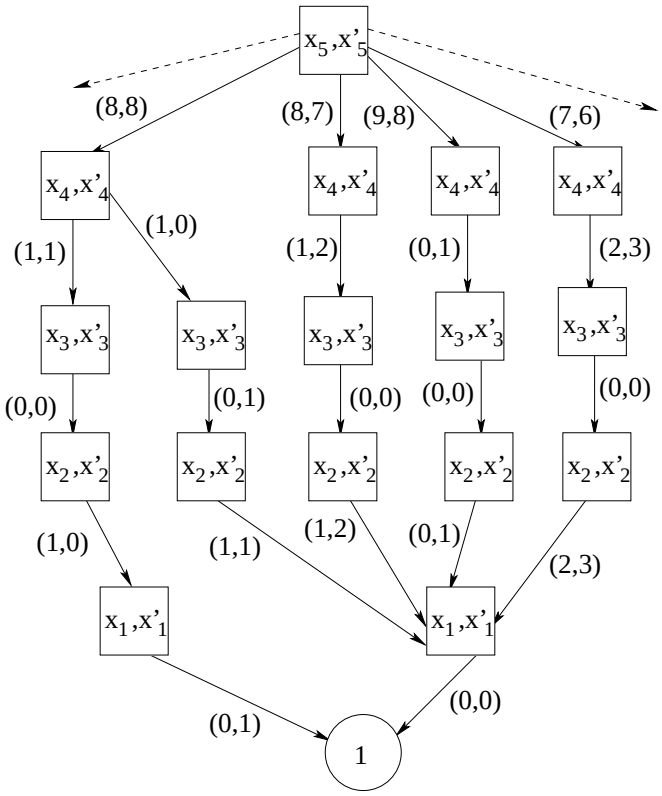


Figure 3.3 MxD representing a part of the transition relation of the PN shown in Figure 2.26 with $N = 9$

The reachability set of the Petri net of Figure 2.26 with $N = 9$ is encoded as shown in Figure 3.4.

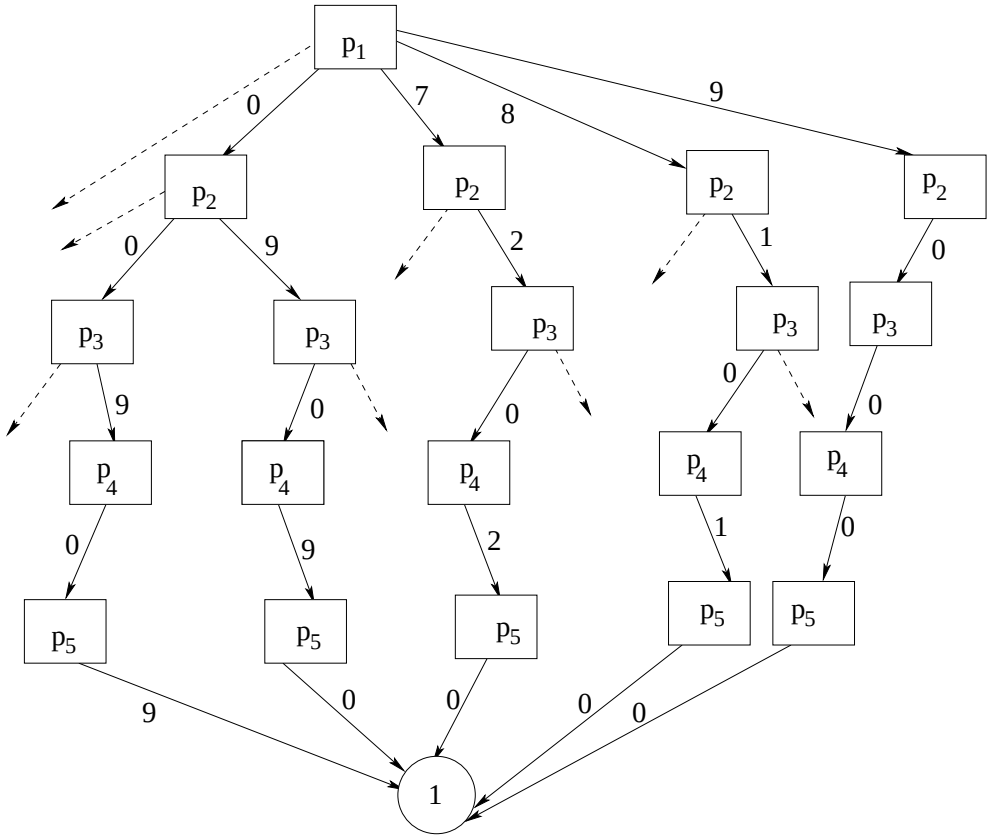


Figure 3.4 MDD representing a part of the reachability set of the PN shown in Figure 2.26 with $N = 9$

3.5 Proposed Encoding Scheme - K-Hot Encoding

This thesis proposes a new encoding scheme called the k-hot encoding. The factor k determines the number of variables used by the encoding. The idea behind this encoding scheme is to not necessarily use b encoding variables to encode b combinations(place p with bound b). If b variables are used, it is easier as only one variable is set at a time and the position of the set-variable determines the value at that place.

With k-hot encoding we use fewer variables depending on the value of k, and a maximum of k encoding variables can be set per place to represent a combination. If x variables are set for place(where $x \leq k$), then the last $x - 1$ variables must be set. A place p of the Petri net, with bound b is encoded using l variables depending on the factor k.

$$l = \left\lceil \frac{b}{k} + \frac{k-1}{2} \right\rceil, \text{ where } 1 \leq k \leq l \tag{3.6}$$

The values of k can be calculated for a fixed b such that if k variables are used and all of them are set, it is enough to cover b.

$$\frac{b}{k} + \frac{k-1}{2} \geq k \tag{3.7}$$

The maximum value of k such that it covers b when all variables are set is l. Hence for $k > l$, the number of variables does not decrease with increase in k. Therefore, k-hot is equivalent to (k+1)-hot for $k > l$. The value of the i^{th} variable of place p is represented as $x_{p,i}$. For a marking m, the value of $m(p)$ can be encoded with l boolean variables using Algorithm 3.5.

The number of tokens at a place is determined by adding the value of the set tokens. The number of tokens at a place p, encoded using l variables can be decoded as:

$$m(p) = x_{p,1} \ + \ 2x_{p,2} \ + \ \ldots \ + \ lx_{p,l} \tag{3.8}$$

```
Encode(p, l)
 1: t := m(p);
 2: while l ≥ 1 do
 3:    if t ≥ l then
 4:       x_{p,l} := 1;
 5:       t := t − l;
 6:       l := l − 1;
 7:    else
 8:       x_{p,l} := 0;
 9:       l := l − 1;
10:    end if
11: end while
```

Figure 3.5 Encoding place p using k-hot encoding

In order to build ADD/MTMDD representing the function $f = m(p)$, the decison diagrams for constant i and variable $x_{p,i}$ are built for all i, such that $1 < i < l$. Next for all i, the multiplication operator is applied on DD's for i and $x_{p,i}$. Addition of all such DD's results in the graph representing function f.

2-hot encoding would require 5 variables to encode every place p in the Petri net of Figure 2.26($N = 9$). These boolean variables are $x_{p,1}$, $x_{p,2}$, $x_{p,3}$, $x_{p,4}$ and $x_{p,5}$. A marking at p, which can be 0 through 9 is encoded as shown in Table 3.5.

$x_{p,1}$	$x_{p,2}$	$x_{p,3}$	$x_{p,4}$	$x_{p,5}$	$m(p)$
0	0	0	0	0	0
1	0	0	0	0	1
0	1	0	0	0	2
0	0	1	0	0	3
0	0	0	1	0	4
0	0	0	0	1	5
1	0	0	0	1	6
0	1	0	0	1	7
0	0	1	0	1	8
0	0	0	1	1	9

Table 3.5 Representing number of tokens at a place with bound 9 using 2-hot

Similarly, 3-hot encoding would require 4 variables to encode the place p.

$x_{p,1}$	$x_{p,2}$	$x_{p,3}$	$x_{p,4}$	$m(p)$
0	0	0	0	0
1	0	0	0	1
0	1	0	0	2
0	0	1	0	3
0	0	0	1	4
1	0	0	1	5
0	1	0	1	6
0	0	1	1	7
1	0	1	1	8
0	1	1	1	9

Table 3.6 Representing number of tokens at a place with bound 9 using 3-hot

An advantage of the k-hot scheme is the flexibility offered by this encoding. By setting different values of k, time and space metrics for computing the reachability set can be evaluated and compared. Interleaving the primed and unprimed variables can further improve the efficiency of this scheme.

The Forkjoin Petri net model introduced in Chapter 2 can be encoded using the k-hot encoding with different values of k. For $k = 1$, the k-hot encoding behaves exactly the same as the one-hot which was explained in Section 3.2. Hence, the following sub-sections explains the encoding of the Forkjoin model using $k = 2$ and $k = 3$.

3.5.1 Encoding Forkjoin model with k as 2

Using Equation(3.5), we can calculate the number of variables required to encode each place of this model using k-hot encoding with k as 2. Thus, we use 5 encoding variables per place in this case. The Transition relation for this Petri net is encoded as shown in Table 3.7.

from state to state	$m(p_1)$ $m'(p_1)$	$m(p_2)$ $m'(p_2)$	$m(p_3)$ $m'(p_3)$	$m(p_4)$ $m'(p_4)$	$m(p_5)$ $m'(p_5)$
p_1:9,p_2:0,p_3:0,p_4:0,p_5	00011	00000	00000	00000	00000
p'_1:8,p'_2:1,p'_3:0,p'_4:1,p'_5	00101	10000	00000	10000	00000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	00101	10000	00000	10000	00000
p'_1:7,p'_2:2,p'_3:0,p'_4:2,p'_5	01001	01000	00000	01000	00000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	00101	10000	00000	10000	00000
p'_1:8,p'_2:0,p'_3:1,p'_4:1,p'_5	00101	00000	10000	10000	00000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	00101	10000	00000	10000	00000
p'_1:8,p'_2:1,p'_3:0,p'_4:0,p'_5	00101	10000	00000	00000	10000
p_1:7,p_2:2,p_3:0,p_4:2,p_5	01001	01000	00000	01000	00000
p'_1:6,p'_2:3,p'_3:0,p'_4:3,p'_5	10001	00100	00000	00100	00000

Table 3.7 Part of the transition relation for Figure 2.26 with $N = 9$ using k-hot encoding with k as 2

3.5.2 Encoding Forkjoin model with k as 3

The k-hot encoding with k as 3, uses 4 variables to encode each place of the Petri net. Thus in all, 20 variables are used to encode the Initial state of this model. The encoded transition relation looks as follows:

from state to state	$m(p_1)$ $m'(p_1)$	$m(p_2)$ $m'(p_2)$	$m(p_3)$ $m'(p_3)$	$m(p_4)$ $m'(p_4)$	$m(p_5)$ $m'(p_5)$
p_1:9,p_2:0,p_3:0,p_4:0,p_5	0111	0000	0000	0000	0000
p'_1:8,p'_2:1,p'_3:0,p'_4:1,p'_5	1011	1000	0000	1000	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	1011	1000	0000	1000	0000
p'_1:7,p'_2:2,p'_3:0,p'_4:2,p'_5	0011	0100	0000	0100	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	1011	1000	0000	1000	0000
p'_1:8,p'_2:0,p'_3:1,p'_4:1,p'_5	1011	0000	1000	1000	0000
p_1:8,p_2:1,p_3:0,p_4:1,p_5	1011	1000	0000	1000	0000
p'_1:8,p'_2:1,p'_3:0,p'_4:0,p'_5	1011	1000	0000	0000	1000
p_1:7,p_2:2,p_3:0,p_4:2,p_5	0011	0100	0000	0100	0000
p'_1:6,p'_2:3,p'_3:0,p'_4:3,p'_5	0101	0010	0000	0010	0000

Table 3.8 Part of the Transition relation for Figure 2.26 with $N = 9$ using k-hot encoding with k as 3

CHAPTER 4. RESULTS

4.1 Overview

This chapter compares the proposed k-hot encoding scheme with the traditional encoding schemes using the Cudd and Meddly libraries. The metrics used for comparison are time and space to build the transition relation, time and space to build the reachability set and time taken to evaluate CTL formulas. Time and space metrics are measured in seconds and KB respectively. Section 4.2 describes the environment in which the experiments were run and also gives implementation specific details for both Cudd and Meddly in Section 4.2.1 and Section 4.2.2 respectively. Next, Section 4.3 describes the Fork-join Petri net model and compares the performance of various encoding schemes used to represent this model in Cudd and Meddly. Sections 4.4, 4.5, 4.3 and 4.7 give a similar comparison for Dining Philosopher, Swaps, Kanban and Tiles Petri net models respectively. Lastly, Section 4.9 concludes about the overall performance of encoding schemes based on results derived for individual models. In the following sections transition relation is referred to as TR, reachability set as RS, peak node count as PN, final node count as FN, peak memory as PM, final memory as FM and number of variables/levels as L.

4.2 Experimental setup

The expermients were run on a Linux machine with CPU speed of 800MHz, memory of 2Gb and memory speed of 533MHz. The comparison tool uses the PNFront library[19] to parse Petri nets. This library parses the set of places, bounds for each place, transition enabling and firing expressions which are used by the tool to build the transition relation and reacha-

bility set for the input PN model. The comparison tool supports both safe and unsafe Petri net models with extensions such as inhibitor arcs, marking dependent arc cardinalities and transition guards. Transition guards represent conditions under which a transition is enabled. Cudd 2.4.1[27] was used to obtain a Cudd-based implementation called CuddImpl and Meddly Revision-80[1] was used to obtain a Meddly-based implementation called MeddlyImpl.

4.2.1 CuddImpl

This sub-section explains the Cudd-based implementation. CuddImpl uses fully-reduced BDD's for transition relation. The basic process of building TR and RS was explained in Section 2.4 of Chapter 2. Cudd provides an excellent interface for manipulating BDD's and ADD's. The Cudd_bddApply operator can be used to apply several operations to two input BDD's. Some of the operations supported by Cudd_bddApply are multiplication(Cudd_bddTimes), addition(Cudd_bddPlus), or(Cudd_bddOr), minus(Cudd_bddMinus), divide(Cudd_bddDivide) etc. To build the relation for a transition t, the intersection of enabling and firing relations of t is computed. Building the enabling relation requires computing a relation R where $R = (m, m') : m(p) \geq W(p,t)$. Chapter 3 explained how a relation for $m(p)$ is built. If $W(p,t)$ is a constant, then BDD for the constant is built and Cudd_bddApply with Cudd_bddOneZeroMaximum as parameter can be used to build BDD for $m(p) \geq W(p,t)$. If $W(p,t)$ is a function of m of the form $m(p_k) \oplus m(p_j)$, then BDDs for $m(p_k)$ and $m(p_j)$ are computed and Cudd_bddApply function with appropriate parameter is used to build the final BDD. Similarly, the firing relation requires building the relation for $m'(p) = m(p) + W(t,p) - W(p,t)$. Once BDD's for $m(p)$, $W(t,p)$ and $W(p,t)$ are built, Cudd_bddApply with Cudd_bddPlus and Cudd_bddMinus can be used to build the firing relation. The intersection of enabling and firing BDD's can be easily computed using Cudd_bddApply method with Cudd_bddTimes as parameter. Once we have the relations for all transitions, union of all these relations is computed to obtain the final transition relation. Cudd_bddApply method with Cudd_bddOr as parameter can be used to obtain the union of two BDD's.

The Reachability set(initially consisting of initial marking), is built by finding post image of the reachabilty set and then computing the union of the post image with the reachability set as described in algorithm of Figure 2.19. The Post image operation is computed in Cudd as follows. Firstly, Cudd_bddApply method with Cudd_bddTimes as parameter is applied on the input BDD's - TR and RS, where RS initially contains just the initial marking. It produces a BDD relation such that, the from states are in RS and the next states are the next reachable states from RS. Next, the Cudd_bddCompose method is used to manipulate the BDD of the previous step such that the next states become the from states of the relation. Lastly, union(Cudd_bddApply method with Cudd_bddOr parameter) of this BDD with RS is obtained. This process is repeated until no new markings are to be added to RS.

In order to measure the time consumed in building TR and RS, a timer is set before starting the process of building the decision diagram and the value of the timer is recorded after the graph is built. The final node count of a DD is measured using the method Cudd_DagSize which takes as argument the decision graph whose node count needs to be measured. The peak node count in Cudd is calculated using Cudd_ReadPeakLiveNodeCount. This method computes the peak node count for the entire forest(containing both TR and RS). Thus, unfortunately the peak node count for building RS can not be computed using the Cudd interface in case when peak node count for building TR is more than RS. The peak and final memory consumed by a decision graph is computed by multiplying the peak and final node counts with the memory consumed by a single node of the graph. Therefore, peak memory is also computed for the entire forest and not the individual decision graph.

4.2.2 MeddlyImpl

This sub-section explains the Meddly-based implementation. MeddlyImpl uses identity-reduced MxD's for transition relation. The Meddly library features several methods including

the apply method(similar to the Cudd_bddApply operator in Cudd) which can be used to perform several operations on MDD's like union, intersection, difference, plus, minus, divide etc. To build the relation for a transition t, the intersection of enabling and firing relations of t is computed. Building the enabling relation requires computing a relation R where $R = (m, m') : m(p) \geq W(p,t)$. If $W(p,t)$ is a constant, then MxD for the constant is built and apply method with GREATER_THAN_EQUAL can be used to build MxD for $m(p) \geq W(p,t)$. If $W(p,t)$ is a function of m of the form $m(p_k) \oplus m(p_j)$, then MxDs for $m(p_k)$ and $m(p_j)$ are computed and apply method with appropriate parameter is used to build the final BDD. Similarly, the firing relation requires building the relation for $m'(p) = m(p) + W(t,p) - W(p,t)$. Once MxD's for $m(p)$, $W(t,p)$ and $W(p,t)$ are built, apply with PLUS and MINUS can be used to build the firing relation. The intersection of enabling and firing BDD's can be easily computed using parameter TIMES with apply method. Once we have the relations for all transitions, union of all these relations(computed using UNION with apply) is computed to obtain the final transition relation. The Meddly library also features a POST_IMAGE parameter with the apply method which can be used to obtain the post image of the reachability set. Hence, it is straight forward to implement the algorithm in Figure 2.19 using the Meddly interface.

In order to measure the time consumed in building TR and RS, a timer is set before starting the process of building the decision diagram and the value of the timer is recorded after the graph is built. The final and peak node counts of a DD are measured using getNodeCount and getPeakNumNodes respectively. The final and peak memory consumed by a decision graph are measured using the methods getCurrentMemoryUsed and getPeakMemoryUsed.

4.3 Fork-join model

The forkjoin Petri net model was discussed in Section 2.5. Properties of this model for various values of N are shown in Table 4.1. Table 4.1 shows the number of places and tran-

sitions in the model for different values of N. The maximum bound column represents the maximum number of tokens a place can have. Lastly, it also shows the number of markings in the reachability set of that model.

N	Places	Transitions	Max bound	Reachable markings
1	5	5	1	5
9	5	5	9	385
10	5	5	10	506
20	5	5	20	3,311
30	5	5	30	10,416

Table 4.1 Properties of Fork-join model

The variables for the model are $x_{p_1,1}, x_{p_1,2}, x_{p_1,3}, \ldots x_{p_5,N}$. The variable ordering used for these variables is $(x_{p_5,N}, x_{p_5,N-1}, \ldots, x_{p_1,3}, x_{p_1,2}, x_{p_1,1})$ from top to bottom level(where bottom level is the one right above the terminal nodes). Table 4.2 shows the Fork-join model in CuddImpl for various values of N. Table 4.3 shows the performance in MeddlyImpl for various values of N.

Table 4.2 shows the peak node count for TR and RS to be the same for some evaluations like 1-hot, 2-hot, 3-hot for 20-Fork-join and 2-hot, 3-hot for 30-Fork-join. For these evaluations, the peak node count for RS generation is less than the peak node count for TR generation and as Cudd interface measures the peak node count for the forest and not the individual graph, thus the peak node count for RS can not be measured for such cases.

Cudd												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
1	1-hot	5	0.0	76	37	1	0	0.0	1,381	12	21	0
1	log	5	0.0	76	37	1	0	0.0	1,381	12	21	0
9	1-hot	45	0.1	20,971	14,334	327	13	1.1	27,959	804	436	12
9	2-hot	25	0.0	11,010	8,144	172	7	0.3	15,874	410	248	6
9	3-hot	20	0.0	11,182	8,304	174	8	0.2	14,381	321	224	5
9	log	20	0.0	18,139	16,932	283	16	0.2	23,002	307	359	4
10	1-hot	50	0.3	28,035	18,872	438	18	1.7	36,875	1,038	576	16
10	2-hot	30	0.1	22,580	15,919	352	15	0.6	26,829	574	419	8
10	3-hot	25	0.0	11,182	8,304	174	8	0.3	15,541	370	242	5
10	log	20	0.0	17,898	16,712	279	16	0.3	23,836	350	372	5
20	1-hot	100	8.3	346,180	172,269	5,409	168	65.9	346,180	6,073	5,409	94
20	2-hot	55	1.8	224,063	126,053	3,500	123	11.7	224,063	2,989	3,500	46
20	3-hot	40	0.8	129,661	81,487	2,025	79	4.3	129,661	1,779	2,025	27
20	log	25	0.2	117,817	113,539	1,840	110	2.7	145,266	1,160	2,269	18
30	2-hot	80	13.0	1,028,821	481,869	16,075	470	159.7	1,028,821	8,504	16,075	132
30	3-hot	55	5.1	565,195	262,226	8,831	256	75.0	565,195	5,452	8,831	85
30	4-hot	45	4.5	585,504	307,972	9,148	300	32.4	585,504	4,350	9,148	67
30	5-hot	40	4.2	570,133	313,858	8,908	306	20.6	570,133	3,844	8,908	60
30	log	25	0.2	107,305	104,027	1,676	101	8.4	177,097	2,065	2,767	32

Table 4.2 Fork-join in CuddImpl

Meddly												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
1	1-hot	5	0.0	65	37	1	1	0.0	20	12	0	0
1	log	5	0.0	65	37	1	1	0.0	20	12	0	0
1	Mdd	5	0.0	65	37	1	1	0.0	20	12	0	0
9	1-hot	45	1.0	1,253	710	35	18	0.2	1,371	804	27	16
9	2-hot	25	0.3	537	352	15	9	0.0	746	410	15	8
9	3-hot	20	0.2	469	324	13	8	0.0	612	321	13	6
9	log	20	0.0	343	227	7	6	0.0	614	307	13	6
9	Mdd	5	0.0	227	133	9	6	0.0	148	88	5	2
10	1-hot	50	1.5	1,578	797	43	20	0.4	1,762	1,038	35	21
10	2-hot	30	0.4	764	439	21	11	0.2	1,044	574	21	11
10	3-hot	25	0.1	483	324	13	8	0.1	720	370	15	7
10	log	20	0.0	343	227	7	6	0.1	717	350	15	7
10	Mdd	5	0.0	247	145	10	7	0.0	174	102	6	3
20	1-hot	100	58.9	8,263	1,667	195	42	4.5	10,538	6,073	210	121
20	2-hot	55	5.9	2,768	874	68	22	2.6	5,621	2,989	115	61
20	3-hot	40	1.2	1,147	586	32	15	1.5	3,547	1,779	74	37
20	log	25	0.1	434	294	10	8	0.8	2,595	1,160	56	24
20	Mdd	5	0.0	447	265	27	19	0.3	521	297	29	14
30	2-hot	80	26.9	6,630	1,309	147	33	11.7	16,313	8,504	331	171
30	3-hot	55	7.6	3,244	904	79	23	8.2	10,871	5,452	225	111
30	4-hot	45	3.7	2,019	760	53	19	5.8	8,846	4,350	186	90
30	5-hot	40	2.3	1,748	792	49	20	5.1	7,912	3,844	168	80
30	log	25	0.1	434	294	10	8	1.5	4,978	2,065	109	44
30	Mdd	5	0.0	647	385	52	37	0.6	1,058	592	80	36

Table 4.3 Fork-join in MeddlyImpl

4.4 Dining philosopher model

The philosophers model is a safe Petri net model. Figure 4.1 shows a 3-Philosophers model. All three philosophers(ph_1, ph_2 and ph_3) have a fork(f_1, f_2 and f_3) on their left and right. A philosopher needs both forks to eat. But, as there are only three forks, only one philosopher can eat at a time. Philosopher ph_i is initially not-hungry(place nh_i has a token). When a philosopher becomes hungry, transition thinking(t_i) fires and a token each is added to places want-left(wl_i) and want-right(wr_i). If the fork on the left is available, transition get-left(gl_i) gets enabled which on firing removes a token from wl_i and adds it to hl_i. Similarly if the fork on the right is available, transition get-right(gr_i) gets enabled. On firing, gr_i removes a token from wr_i and adds it to hr_i. Once both hr_i and hl_i have a token each, transition eat(e_i) is enabled which on firing removes a token each from hl_i and hr_i and adds a token to nh_i leaving ph_i back in not hungry state and making left and right forks available.

Thus, a philosopher can eat only if both forks(left and right) are available. For ph_1 to eat, both f_1 and f_3 should be free, for ph_2 to eat, f_1 and f_2 should be free and for ph_3 to eat, f_2 and f_3 should be free. The variable ordering used for this model is ($f_3, hr_3, wr_3, hl_3, wl_3, nh_3$, $f_2, hr_2, wr_2, hl_2, wl_2, nh_2, f_1, hr_1, wr_1, hl_1, wl_1, nh_1$) from top to bottom level. This model can also be extended easily to N-Philosophers. For 4-Philosophers, group another philosopher say ph_4 such that ph_3, ph_4 share the same fork f_3 and ph_4, ph_1 share the same fork f_4.

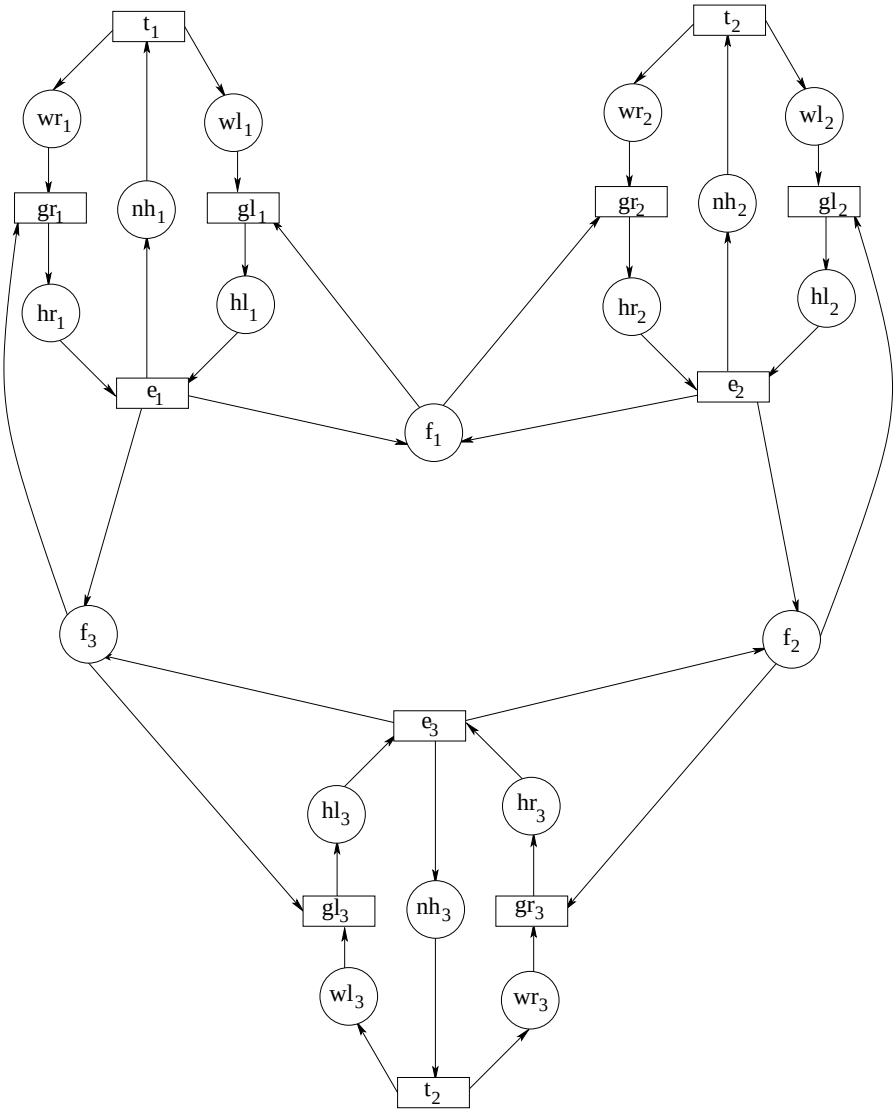


Figure 4.1 3 Dining philosophers model

Properties(count of places, transitions, reachable markings etc) of this model for various values of N are shown in Table 4.4. Table 4.5 shows the Philosophers model in CuddImpl with various values of N. For a safe PN model, all three encodings 1-hot, logarithmic and MDD are equivalent. Thus, the metrics are same for all encodings in Cudd and Meddly as shown in Tables 4.5 and 4.6.

N	Places	Transitions	Max bound	Reachable markings
5	30	20	1	1364
10	60	40	1	1,860,498
20	120	80	1	3,461,452,808,002
30	180	120	1	6,440,026,026,380,244,498
40	240	160	1	11,981,655,542,024,930,675,232,002
50	300	200	1	22,291,846,172,619,859,445,381,409,012,498
60	360	240	1	1,473,935,220,454,921,602,871,195,774,259,272,002

Table 4.4 Properties of Philosophers model

Cudd												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
5	1-hot	5	0.0	775	528	12	0	0.1	7,256	187	113	2
5	log	5	0.0	775	528	12	0	0.1	7,256	187	113	2
10	1-hot	10	0.2	1,680	1,195	26	1	3.5	31,432	450	491	7
10	log	10	0.2	1,680	1,195	26	1	3.4	31,432	450	491	7
20	1-hot	20	0.7	3,470	2,505	54	2	34.3	127,704	950	1,995	14
20	log	20	0.7	3,470	2,505	54	2	34.6	127,704	950	1,995	14
30	1-hot	30	4.4	5,260	3,815	82	3	284.4	286,341	1,450	4,474	22
30	log	30	4.4	5,260	3,815	82	3	284.1	286,341	1,450	4,474	22
40	1-hot	40	6.4	7,050	5,125	110	5	355.4	510,794	1,950	7,981	30
40	log	40	6.4	7,050	5,125	110	5	355.3	510,794	1,950	7,981	30
50	1-hot	50	19.4	8,840	6,435	138	6	2304.8	798,356	2,450	12,474	38
50	log	50	20.6	8,840	6,435	138	6	2304.7	798,358	2,450	12,474	38
60	1-hot	60	21.1	10,630	7,745	166	7	4403.6	1,149,769	2,950	17,965	46
60	log	60	21.4	10,630	7,745	166	7	4403.1	1,149,769	2,950	17,965	46

Table 4.5 Philosophers model in CuddImpl

Meddly												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
5	1-hot	5	0.0	681	528	15	14	0.0	638	187	13	4
5	log	5	0.0	681	528	15	14	0.0	638	187	13	4
5	Mdd	5	0.0	681	528	15	14	0.0	638	187	13	4
10	1-hot	10	0.1	1,496	1,195	33	32	0.7	2,218	450	68	10
10	log	10	0.1	1,496	1,195	33	32	0.7	2,218	450	68	10
10	MDD	10	0.1	1,496	1,195	33	32	0.7	2,218	450	68	10
20	1-hot	20	0.2	3,106	2,505	69	68	8.3	13,484	950	296	22
20	log	20	0.2	3,106	2,505	69	68	8.3	13,484	950	296	22
20	Mdd	20	0.2	3,106	2,505	69	68	8.3	13,484	950	296	22
30	1-hot	30	0.9	4,718	3,815	104	104	41.4	31,424	1,450	687	34
30	log	30	0.9	4,718	3,815	104	104	40.3	31,424	1,450	687	34
30	Mdd	30	1.0	4,718	3,815	104	104	41.1	31,424	1,450	687	34
40	1-hot	40	0.9	6,328	5,125	140	139	114.4	56,868	1,950	1,239	46
40	log	40	1.0	6,328	5,125	140	139	114.5	56,868	1,950	1,239	46
40	Mdd	40	0.9	6,328	5,125	140	139	114.4	56,868	1,950	1,239	46
50	1-hot	50	3.0	7,938	6,435	176	175	287.0	89,802	2,450	1,953	58
50	log	50	3.0	7,938	6,435	176	175	287.6	89,802	2,450	1,953	58
50	Mdd	50	3.1	7,938	6,435	176	175	287.3	89,802	2,450	1,953	58
60	1-hot	60	2.3	9,548	7,745	212	211	412.5	130,240	2,950	2,828	70
60	log	60	2.4	9,548	7,745	212	211	412.5	130,240	2,950	2,828	70
60	Mdd	60	2.4	9,548	7,745	212	211	412.7	130,240	2,950	2,828	70

Table 4.6 Philosophers model in MeddlyImpl

4.5 Swaps model

The swaps-3 model is shown in Figure 4.2. This model has 3 places(p_0, p_1 and p_2) and 2 transitions(t_1 and t_2). The variable ordering used for this model is (p_2, p_1, p_0) from top to bottom level. Transition t_1 is enabled when place p_0 has $\#p_0$ tokens and place p_1 has $\#p_1$ tokens. On firing, t_1 removes $\#p_0$ tokens from p_0 and adds $\#p_1$ tokens to it. It also removes $\#p_1$ tokens from p_1 and adds $\#p_0$ tokens to it. Hence, tokens get swapped between p_0 and p_1. Similarly, transition t_2 gets enabled when p_1 has $\#p_1$ tokens and p_2 has $\#p_2$ tokens. On firing, t_2 removes $\#p_1$ tokens from p_1 and adds $\#p_2$ tokens to it. It also removes $\#p_2$ tokens from p_2 and adds $\#p_1$ tokens to it. Hence, tokens get swapped between p_1 and p_2.

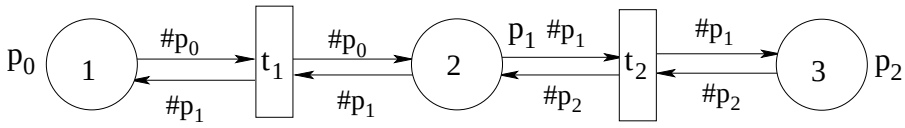


Figure 4.2 Swaps-3 model

This model can be easily extended to swaps-N model. For example, in order to extend it to swaps-4, another place p_3 and transition t_3 are added such that on firing, t_3 swaps the tokens between places p_2 and p_3. Thus any transition in this model exchanges the number of tokens in adjoining places. Properties of this model for various values of N are shown in Table 4.7.

N	Places	Transitions	Max bound	Reachable markings
6	6	5	6	720
7	7	6	7	5,040
8	8	7	8	40,320
9	9	8	9	362,880
10	10	9	10	3,628,800
11	11	10	11	39,916,800

Table 4.7 Properties of Swaps model

Tables 4.8 and 4.9 show the performance of swaps model in CuddImpl and MeddlyImpl for various values of N.

Cudd												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
6	1-hot	36	0.2	22,308	16,385	348	16	0.5	35,145	581	549	9
6	2-hot	24	0.1	9,941	7,422	155	7	0.3	18,886	425	295	6
6	log	18	0.0	4,594	3,246	71	3	0.2	11,412	299	178	4
7	1-hot	49	0.5	44,911	32,841	701	32	2.9	90,613	1,413	1,415	22
7	2-hot	28	0.1	15,239	11,411	238	11	1.2	41,958	900	655	14
7	log	21	0.0	5,814	4,027	90	3	0.7	25,239	629	394	9
8	1-hot	64	1.2	79,976	58,002	1,249	56	20.3	236,814	3,333	3,700	52
8	2-hot	40	0.3	31,409	23,301	490	22	11.0	132,518	2,524	2,070	39
8	3-hot	32	0.2	18,467	13,636	288	13	5.5	98,003	1,905	1,531	29
8	log	32	0.1	27,488	21,528	429	21	5.3	105,573	1,759	1,649	27
9	1-hot	81	2.9	134,791	97,972	2,106	95	144.7	671,499	7,685	10,492	120
9	2-hot	45	0.4	36,888	27,099	576	26	52.0	343,041	5,277	5,360	82
9	3-hot	36	0.3	40,461	25,993	632	25	41.4	268,864	3,918	4,201	61
9	log	36	0.1	32,817	25,112	512	24	41.3	265,699	3,838	4,151	59
10	1-hot	100	5.5	212,971	154,755	3,327	151	686.0	2,018,231	17,413	31,534	272
10	2-hot	60	1.3	76,338	56,716	1,192	55	397.3	1,183,929	13,861	18,498	216
10	3-hot	50	0.4	50,542	31,698	789	30	175.8	744,280	8,270	11,629	129
10	log	40	0.2	39,503	29,782	617	29	189.4	735,528	8,250	11,492	128
11	1-hot	121	10.4	317,318	229,643	4,958	224	4639.9	6,227,662	38,917	97,307	608
11	2-hot	66	1.6	86,213	63,686	1,347	62	1712.1	3,344,138	28,712	52,252	448
11	3-hot	55	2.5	220,012	132,758	3,437	129	1200.1	2,782,814	23,143	43,481	361
11	log	44	0.2	45,070	33,275	704	32	814.6	2,043,994	17,488	31,937	273

Table 4.8 Swaps model in CuddImpl

The node count and memory consumed by logarithmic encoding is less than k-hot for most cases. However, for Swaps-8 in CuddImpl, the final/peak node count and final/peak memory consumed for TR by logarithmic encoding is greather than that for 3-hot.

In most scenarios, k-hot encoding shows a gradual decrease in the node count and memory consumed for increasing values of k. But, for Swaps-11 in MeddlyImpl, the final/peak node count and memeory consumed for TR by 2-hot is better than that for 3-hot.

Meddly												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
6	1-hot	36	0.4	4,420	3,350	92	73	0.3	1,140	581	23	12
6	2-hot	24	0.3	2,912	2,362	63	53	0.2	794	425	17	9
6	log	18	0.0	1,693	1,625	38	37	0.1	591	299	13	6
6	Mdd	6	0.0	524	494	19	18	0.1	155	65	6	2
7	1-hot	49	1.3	8,416	6,144	177	132	1.4	3,422	1,413	71	30
7	2-hot	28	0.5	6,454	4,770	146	109	0.8	2,164	900	46	19
7	log	21	0.1	3,054	2,135	69	50	0.5	1,565	629	35	14
7	Mdd	7	0.0	1,148	773	46	32	0.3	375	129	14	5
8	1-hot	64	6.9	14,279	10,388	301	222	7.4	10,375	3,333	215	70
8	2-hot	40	1.6	9,367	6,620	204	146	4.3	7,614	2,524	161	54
8	3-hot	32	0.8	7,979	5,946	188	136	3.3	5,884	1,905	129	42
8	log	32	0.2	11,855	8,834	270	202	4.0	5,600	1,759	123	39
8	Mdd	8	0.0	1,698	1,140	73	50	1.3	924	257	40	12
9	1-hot	81	14.0	22,760	16,512	479	351	27.7	30,685	7,685	636	160
9	2-hot	45	2.1	11,743	8,086	256	179	15.0	20,464	5,277	438	113
9	3-hot	36	1.1	10,342	7,228	235	166	12.7	15,718	3,918	347	87
9	log	36	0.3	14,584	10,726	337	246	14.9	15,288	3,838	338	85
9	Mdd	9	0.1	2,398	1,607	111	75	5.5	2,421	513	111	26
10	1-hot	100	26.8	29,198	25,002	726	529	156.5	88,310	17,413	1,829	361
10	2-hot	60	5.5	21,542	14,923	466	326	80.9	67,211	13,861	1,424	293
10	3-hot	50	1.4	12,604	8,616	287	198	47.8	41,607	8,270	937	186
10	log	40	0.4	18,057	12,774	413	293	62.2	40,136	8,250	919	184
10	Mdd	10	0.1	3,266	2,186	163	109	21.1	5,827	1,025	301	56
11	1-hot	121	36.5	50,392	36,400	1,059	767	4450.9	246,988	38,917	5,110	806
11	2-hot	66	6.1	25,787	17,509	559	382	1684.7	175,451	28,712	3,738	608
11	3-hot	55	4.2	29,910	21,319	679	485	1186.2	143,593	23,143	3,126	502
11	log	44	0.4	21,626	14,978	495	344	791.4	106,813	17,488	2,443	391
11	Mdd	11	0.1	4,320	2,889	229	152	123.5	14,541	2,049	801	120

Table 4.9 Swaps model in MeddlyImpl

4.6 Kanban model

The Kanban Petri net model consists of 4 parallel processes which are synchronized amongst each other. The model is shown in Figure 4.3. The net has 4 stations. A part enters a station only if a kanban ticket is available (places *pkan*). The part is then processed in the station. If the processing completes correctly, transition *tok* fires and the part moves to the next station. Else, the part is sent back to the same station via transition *tback* for reprocessing. After passing station 1, a part splits into two parts, one each for station 2 and station 3. After the processing completes successfully at station 2 and station 3, the two parts join and move to station 4. The initial state of the model has N tokens at $pkan_1$, $pkan_2$, $pkan_3$ and $pkan_4$. All places have a bound of N. The variable oredering used by this model is $(pout_4, pkan_4, pback_4, pm_4, pout_3, pkan_3, pback_3, pm_3, \ldots, pout_1, pkan_1, pback_1, pm_1)$ from top to bottom level.

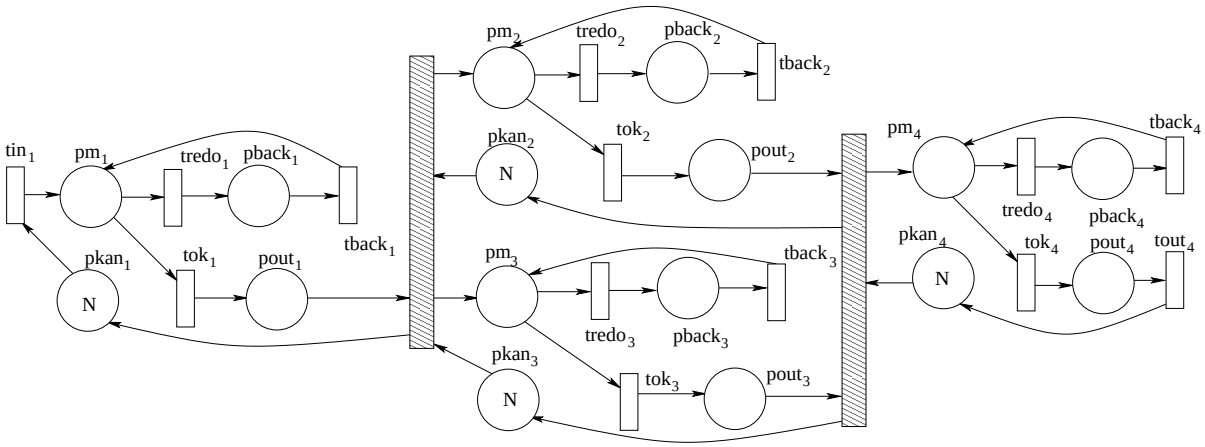


Figure 4.3 Kanban model

Properties of this model for various values of N are shown in Table 4.10.

N	Places	Transitions	Max bound	Reachable markings
1	16	16	1	160
3	16	16	3	58,400
5	16	16	5	2,546,432
7	16	16	7	41,644,800
8	16	16	8	133,865,325
9	16	16	9	384,392,800
10	16	16	10	1,005,927,208

Table 4.10 Properties of Kanban model

Tables 4.11 and 4.12 show performance of the Kanban model in CuddImpl and MeddlyImpl respectively.

As seen from Table 4.11, 3-hot encoding performs better than logarithmic with respect to node count and memory(for both TR and RS) for 8-Kanban in CuddImpl. 1-Kanban is a safe PN model and hence the performance of 1-hot and logarithmic is same for this model in CuddImpl and MeddlyImpl.

Cudd												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
1	1-hot	16	0.0	321	160	5	0	0.0	1,562	34	24	0
1	log	16	0.0	321	160	5	0	0.0	1,562	34	24	0
3	1-hot	48	0.2	22,759	16,797	355	16	1.1	30,547	227	477	3
3	2-hot	32	0.0	2,932	2,369	45	2	0.5	11,332	143	177	2
3	log	32	0.0	2,980	2,369	46	2	0.5	11,380	143	177	2
5	1-hot	80	2.7	258,402	169,841	4,037	165	13.2	258,402	672	4,037	10
5	2-hot	48	0.3	38,547	27,490	602	26	4.8	72,329	382	1,130	5
5	log	48	0.1	32,932	25,843	514	25	4.7	68,950	361	1,077	5
7	1-hot	112	130.5	2,067,911	1,172,033	32,311	1,144	323.3	2,067,911	1,449	32,311	22
7	2-hot	64	3.4	348,642	187,682	5,447	183	58.9	348,642	790	5,447	12
7	log	48	0.2	62,708	45,684	979	44	39.6	148,665	527	2,322	8
8	1-hot	128	517.8	3,469,820	1,921,133	54,215	1,876	758.1	3,469,820	1,987	54,215	31
8	2-hot	80	19.3	793,940	472,450	12,405	461	277.2	793,940	1,196	12,405	18
8	3-hot	64	4.1	348,642	187,682	5,447	183	179.7	395,923	943	6,186	14
8	log	64	4.3	915,246	670,122	14,300	654	139.8	915,246	854	14,300	13
9	1-hot	144	2402.1	7,504,474	4,016,745	117,257	3,922	1742.7	7,504,474	2,638	117,257	41
9	2-hot	80	20.7	793,940	472,450	12,405	461	488.8	834,391	1,406	13,037	21
9	3-hot	64	4.1	348,642	187,682	5,447	183	345.0	475,103	1,112	7,423	17
9	log	64	4.5	917920	672735	14342	656	289.1	936913	987	14639	15
10	2-hot	96	163.8	2,505,916	1,392,599	39,154	1,359	1250.8	2,505,916	1,976	39,154	30
10	3-hot	64	4.1	348,642	187,682	5,447	183	628.6	570,332	1,300	8,911	20
10	log	64	4.7	909,036	663,726	14,203	648	560.6	1,015,040	1,131	15,860	17

Table 4.11 Kanban model in CuddImpl

Meddly												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
1	1-hot	16	0.0	306	160	6	5	0.0	114	34	2	0
1	log	16	0.0	306	160	6	5	0.0	114	34	2	0
1	Mdd	16	0.0	306	160	6	5	0.0	114	34	2	0
3	1-hot	48	1.0	1,298	784	29	21	0.3	1,414	227	29	5
3	2-hot	32	0.3	676	392	15	11	0.2	938	143	20	3
3	log	32	0.0	684	392	15	11	0.2	941	143	20	3
3	Mdd	16	0.0	492	264	13	9	0.1	439	75	11	2
5	1-hot	80	6.1	2,479	1,486	54	39	7.9	7,139	672	147	15
5	2-hot	48	1.2	1,324	833	30	23	4.7	4,201	382	94	9
5	log	48	0.1	1,083	643	24	19	4.2	4,352	361	95	8
5	Mdd	16	0.0	676	368	21	15	2.0	1,354	128	37	3
7	1-hot	112	19.9	3,645	2,188	80	58	126.3	22,648	1,449	468	31
7	2-hot	64	2.6	2,151	1,310	49	35	60.3	12,689	790	282	17
7	log	48	0.1	1,083	643	24	19	38.3	9,770	527	216	12
7	Mdd	16	0.0	857	472	30	22	23.7	3,058	193	91	6
8	1-hot	128	21.5	4,334	2,539	93	67	293.6	36,260	1,987	742	43
8	2-hot	80	6.8	2,283	1,413	50	38	206.0	22,878	1,196	492	26
8	3-hot	64	3.6	2,151	1,310	49	35	121.7	18,482	943	403	21
8	log	64	0.2	1,488	902	33	26	125.8	18,636	854	398	19
8	Mdd	16	0.1	954	524	36	26	54.7	4,286	230	133	7
9	1-hot	144	64.6	4,811	2,890	105	76	738.0	55,067	2,638	1,127	56
9	2-hot	80	8.8	2,283	1,413	50	38	491.4	31,312	1,406	659	31
9	3-hot	64	4.1	2,151	1,310	49	35	306.7	24,735	1,112	550	25
9	log	64	0.2	1,488	902	33	26	260.4	24,032	987	541	22
9	Mdd	16	0.1	1,046	576	41	30	114.1	5,765	270	186	9
10	1-hot	160	71.9	5,471	3,241	119	85	1524.1	80,138	3,412	1,638	72
10	2-hot	96	17.4	2,765	1,764	63	47	924.6	49,126	1,976	1,033	43
10	3-hot	64	4.0	2,151	1,310	49	35	707.5	33,940	1,300	734	29
10	log	64	0.2	1,488	902	33	26	440.3	32,019	1,131	699	25
10	Mdd	16	0.1	1,076	628	47	34	226.5	7,535	313	253	11

Table 4.12 Kanban model in MeddlyImpl

4.7 Tiles model

Tiles(M,N) is another unsafe Petri net used for comparison. Figure 4.4 shows tiles(2,2) model. This model consists of 4 places($p_{11}, p_{12}, p_{21}, p_{22}$) and 8 transitions. The initial marking of the Petri net is $[p_{11} : 0, p_{12} : 1, p_{21} : 2, p_{22} : 3]$. The variable ordering used is $(p_{22}, p_{21}, p_{12}, p_{11})$ from top to bottom level. It is basically a 2-dimensional version of swaps model where the exchange of tokens takes place only when one of the places is empty. For example, if place p_{12} does not have any tokens then the transition e_{12} is enabled. On firing, e_{12} removes all tokens from place p_{11} and adds them to place p_{12}. As a result, p_{11} has 0 tokens which enables transition w_{11}. On firing, w_{11} removes all tokens from p_{12} and adds them to p_{11}. This model can be easily extended for different values of M and N. For an increase in the values of M, a sub-graph similar to g_1 shown in Figure 4.4 is added to the left of g_1. Hence, for tiles(3,2) a sub-graph similar to g_1 with places (x_{31} and x_{32}) and transitions (w_{31}, n_{21}, n_{22}, e_{32}, s_{31} and s_{32}) is added. Similarly, for an increase in the value of N, a sub-graph similar to g_2 shown in Figure 4.4 is added.

Properties of this model for various values of M and N are shown in Table 4.13.

$M \times N$	Places	Transitions	Max bound	Reachable markings
2×2	4	8	3	12
2×3	6	14	5	360
3×2	6	14	5	360
2×4	8	20	7	20,160
4×2	8	20	7	20,160
3×3	9	24	8	181,440
2×5	10	26	9	1,814,400
5×2	10	26	9	1,814,400

Table 4.13 Properties of Tiles model

Tables 4.14 and 4.15 show performance of the Tiles(M,N) model in CuddImpl and MeddlyImpl respectively.

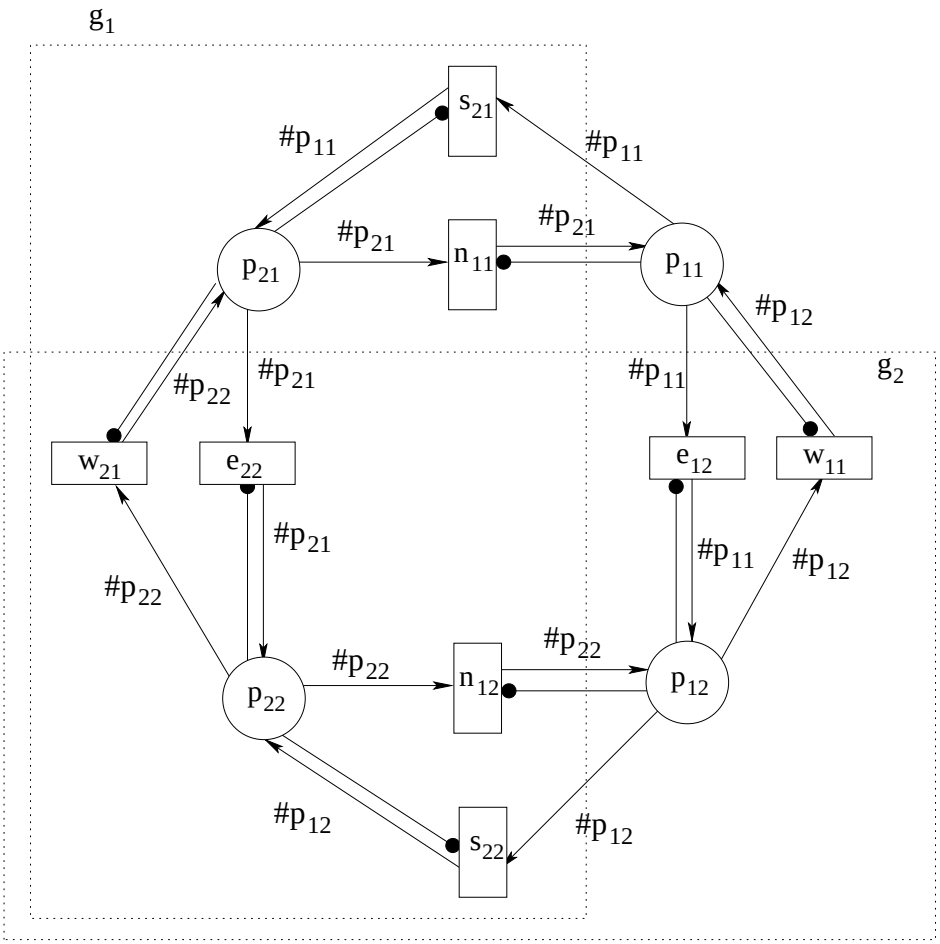


Figure 4.4 Tiles(2,2) model

Cudd												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
2×2	1-hot	12	0.0	832	540	13	0	0.0	1,982	53	30	0
2×2	2-hot	12	0.0	337	219	5	0	0.0	1,078	36	26	0
2×2	log	8	0.0	345	219	5	0	0.0	1,068	36	25	0
2×3	1-hot	30	0.1	5,941	4,007	92	3	0.4	12,717	691	198	10
2×3	2-hot	18	0.0	2,377	1,697	37	1	0.2	6,994	432	109	6
2×3	log	18	0.0	1,905	1,410	29	1	0.2	6,715	425	104	6
3×2	1-hot	30	0.1	5,308	3,336	82	3	0.3	10,788	691	168	10
3×2	2-hot	18	0.0	2,112	1,416	33	1	0.1	6,008	432	93	6
3×3	log	18	0.0	1,710	1,199	26	1	0.1	5,845	425	91	6
2×4	1-hot	56	0.5	21,482	14,837	335	14	26.9	192,164	5,143	3,002	80
2×4	2-hot	32	0.1	10,586	7,344	165	7	9.2	111,689	3,421	1,745	53
2×4	log	24	0.0	3,642	2,817	56	2	6.3	79,429	2,430	1,241	37
4×2	1-hot	56	0.4	17,066	10,260	266	10	18.4	119,766	5,143	1,871	80
4×2	2-hot	32	0.1	8,366	5,041	130	4	6.4	73,014	3,421	1,140	53
4×2	log	24	0.0	2,888	2,024	45	1	4.0	51,718	2,430	808	37
3×3	1-hot	72	0.9	31,947	21,118	499	20	128.8	801,294	12,623	12,520	197
3×3	2-hot	45	0.2	15,586	10,800	243	10	60.0	526,454	9,911	8,225	154
3×3	3-hot	36	0.2	11,677	7,838	182	7	40.3	416,685	7,581	6,510	118
3×3	log	36	0.0	10,019	7,696	156	7	40.5	404,438	6,827	6,319	106
2×5	1-hot	90	1.9	56,333	39,500	880	38	2064.9	6,343,758	29,787	99,121	465
2×5	2-hot	50	0.5	21,116	15,766	329	15	928.3	3,658,287	21,146	57,160	330
2×5	3-hot	40	0.3	15,805	11,493	246	11	631.1	2,927,929	16,006	45,748	250
2×5	log	40	0.1	13,529	10,946	211	10	630.3	2,911,142	15,346	45,486	239
5×2	1-hot	90	1.6	40,434	23,184	631	22	1205.7	2,746,699	29,787	42,917	465
5×2	2-hot	50	0.3	14,773	9,252	230	9	460.8	1,691,830	21,146	26,434	330
5×2	3-hot	40	0.2	11,131	6,693	173	6	290.1	1,359,777	16,006	21,246	250
5×2	log	40	0.1	9,528	6,768	148	6	288.9	1,322,711	15,346	20,667	239

Table 4.14 Tiles model in CuddImpl

Meddly												
N	Enc.	L	TR					RS				
			CPU	PN	FN	PM	FM	CPU	PN	FN	PM	FM
2×2	1-hot	12	0.0	430	347	11	8	0.0	73	53	1	1
2×2	2-hot	12	0.0	306	214	6	5	0.0	52	36	1	0
2×2	log	8	0.0	306	214	6	5	0.0	52	36	1	0
2×2	Mdd	4	0.0	184	129	4	3	0.0	26	17	0	0
2×3	1-hot	30	0.1	3,080	2,365	66	52	0.3	1,065	691	22	14
2×3	2-hot	18	0.0	2,015	1,604	45	37	0.2	725	432	15	9
2×3	log	18	0.0	1,855	1,489	42	34	0.2	750	425	16	9
2×3	Mdd	6	0.0	893	674	27	22	0.0	211	105	6	3
3×2	1-hot	30	0.2	2,318	1,979	56	44	0.2	1,035	691	21	14
3×2	2-hot	18	0.0	1,704	1,320	38	30	0.1	620	432	13	9
3×2	log	18	0.0	1,580	1,243	36	28	0.1	650	425	14	9
3×2	Mdd	6	0.0	748	531	23	17	0.0	179	105	5	3
2×4	1-hot	56	4.3	11,782	8,810	250	193	21.8	26,282	5,143	546	107
2×4	2-hot	32	1.0	8,781	7,022	195	159	8.7	15,932	3,421	348	74
2×4	log	24	0.1	4,168	3,030	92	69	5.3	12,017	2,430	278	55
2×4	Mdd	8	0.1	2,960	2,226	104	81	3.5	3,339	481	132	20
4×2	1-hot	56	6.9	8,547	5,990	183	132	11.9	19,546	5,143	405	107
4×2	2-hot	32	1.7	6,061	4,528	135	103	4.6	12,020	3,421	260	74
4×2	log	24	0.1	3,078	2,112	68	48	3.4	8,908	2,430	203	55
4×2	Mdd	8	0.1	2,050	1,372	73	51	1.3	2,421	481	97	20
3×3	1-hot	72	24.0	18,410	12,996	393	284	94.4	154,261	12,623	3,181	262
3×3	2-hot	45	5.4	11,316	8,069	244	178	53.7	101,876	9,911	2,173	210
3×3	3-hot	36	2.9	10,841	7,968	241	181	38.1	81,344	7,581	1,785	166
3×3	log	36	0.6	10,316	7,831	228	177	36.3	80,273	6,827	1,762	151
3×3	Mdd	9	0.1	4,298	3,027	162	117	14.5	17,377	989	765	46
2×5	1-hot	90	49.2	34,266	24,466	728	532	2046.7	975,343	29,787	20,266	617
2×5	2-hot	50	7.8	17,652	12,452	380	274	918.3	578,716	21,146	12,550	452
2×5	3-hot	40	3.8	17,028	12,417	379	282	621.8	464,598	16,006	10,609	354
2×5	log	40	1.0	16,108	12,021	355	271	618.2	482,098	15,346	10,850	340
2×5	Mdd	10	0.3	7,830	5,761	308	233	316.0	94,361	2,009	4,691	101
5×2	1-hot	90	45.9	20,331	13,580	436	297	728.4	535,041	29,787	11,143	617
5×2	2-hot	50	6.9	10,452	6,869	227	153	368.6	334,832	21,146	7,217	452
5×2	3-hot	40	3.6	9,824	6,638	220	151	287.6	269,443	16,006	5,981	354
5×2	log	40	0.7	9,341	6,599	208	149	279.5	262,647	15,346	5,809	340
5×2	Mdd	10	0.2	4,400	2,832	177	116	79.3	51,801	2,009	2,595	101

Table 4.15 Tiles model in MeddlyImpl

4.8 Performance in computing CTL formulas

Average time taken to compute CTL formulas such as deadlock for the N-Philosophers model, are also used as a metric for comparison. Table 4.16 shows the time taken in seconds for evaluating few CTL formulas in both Cudd and Meddly. The N-Philosopher model is used for this comparison. The CTL formulas used for comparison are,

$$\begin{aligned} start &= \textit{not-hungry}_1 \wedge \textit{not-hungry}_2 \wedge \ldots \wedge \textit{not-hungry}_n, \\ not\_dead &= EF start, \\ deadlock &= \neg not\_dead, \\ can\_deadlock &= EF deadlocked, \\ will\_deadlock &= AF deadlocked, \\ will\_eat_1 &= AF eat_1, \\ will\_eat_n &= AF eat_n, \end{aligned} \tag{4.1}$$

The CTL formula *start* is satisfied by a state where the places ($not - hungry_1$, $not - hungry_2$, ..., $not - hungry_n$) have a token each. In other words, *start* holds if all philosophers are not-hungry. $EF start$ is true for a state s if there exists a path from s where *start* holds for some state along the path. The CTL formula *deadlock* describes a state s such that there is no state reachable from s where all philosophers are not-hungry. Formula *can_deadlock* holds for a state s if there exists a path from s where *deadlock* holds for some state along the path. The CTL formula *will_deadlock* holds for a state s if for every path from s, there exists a state along the path where *deadlock* is satisfied. Next, $will\_eat_n$ holds in a state s if for every path from s, there exists a state along the path where transition eat_n is enabled($Philosopher_n$ can eat).

CTL formula evaluation						
N	CTL Formula	1-hot		Log		MDD
		CuddImpl	MeddlyImpl	CuddImpl	MeddlyImpl	MeddlyImpl
5	$will\_eat_1$	0.0	0.0	0.0	0.0	0.0
5	$will\_eat_2$	0.0	0.0	0.0	0.0	0.0
5	$can\_deadlock$	0.0	0.0	0.0	0.0	0.0
5	$will\_deadlock$	0.0	0.0	0.0	0.0	0.0
10	$will\_eat_1$	0.0	0.1	0.0	0.1	0.1
10	$will\_eat_2$	0.0	0.0	0.0	0.0	0.0
10	$can\_deadlock$	0.0	0.0	0.0	0.0	0.0
10	$will\_deadlock$	0.0	0.0	0.0	0.0	0.0
20	$will\_eat_1$	0.1	0.1	0.1	0.9	0.9
20	$will\_eat_2$	0.0	0.0	0.0	0.0	0.0
20	$can\_deadlock$	0.0	0.0	0.0	0.0	0.0
20	$will\_deadlock$	0.5	0.0	0.8	0.0	0.0
30	$will\_eat_1$	0.2	0.8	0.3	0.8	0.7
30	$will\_eat_2$	0.0	0.0	0.0	0.0	0.0
30	$can\_deadlock$	0.0	0.0	0.0	0.0	0.0
30	$will\_deadlock$	0.1	0.0	0.1	0.0	0.0
40	$will\_eat_1$	0.5	1.1	0.6	1.2	1.2
40	$will\_eat_2$	0.0	0.0	0.0	0.0	0.0
40	$can\_deadlock$	0.0	0.0	0.0	0.0	0.0
40	$will\_deadlock$	0.2	0.3	0.3	0.2	0.2

Table 4.16 Comparing time taken to evaluate CTL formulas

4.9 Discussion

As k-hot and logarithmic encoding schemes use boolean encoding variables, RS is expected to match in CuddImpl and MeddlyImpl and hence have the same final node count. The peak node count depends on the efficiency and order of operations performed and hence may differ in CuddImpl and MeddlyImpl. The results in the previous sections show same RS final node count in CudImpl and MeddlyImpl for k-hot and logarithmic schemes. For TR, CuddImpl uses fully reduced BDD's and MeddlyImpl uses identity reduced MxD's. Hence TR peak/final node count may not match for the two implementations. For a safe Petri net model, RS and TR both are expected to match because all three encodings are equivalent for a 1-bounded PN model.

MDD encoding clearly performs the best in terms of time taken to build TR and RS, peak/final memory consumed in the process and peak/final node count of TR and RS. Logarithmic encoding seems the best choice, when using BDD's. For most models in CuddImpl and MeddlyImpl, it performs the best in terms of time taken, peak/final node count and peak/final memory consumed. However, for some models like swaps8 in CuddImpl, 3-hot encoding uses fewer peak and final nodes for TR as compared to logarithmic. Both k-hot and logarithmic encodings perform better in MeddlyImpl than in CuddImpl with respect to time taken, node count and memory consumed in building TR and RS. For k-hot encoding in both CuddImpl and MeddlyImpl, a gradual decrease in consumption of time and nodes is seen with increasing values of k.

The current implementations in CuddImpl and MeddlyImpl do not support increase in bound during the generation of DDs. However, if Cudd and Meddly libraries provide the feature to dynamically assign variables to a DD, then dynamic increase in bound can be supported with slight modifications to the encodings. Encodings like k-hot and logarithmic can be modified to support this change by adding a new variables to accomodate the increase in bound. Mdd encoding uses a single variable per place, hence no change in the encoding would

be needed to support the increase in bound.

CHAPTER 5. SUMMARY AND FUTURE WORK

This thesis introduced a new encoding scheme called k-hot encoding. Existing encoding schemes were implemented and compared in BDD based Cudd library and MDD based Meddly library. The results in Chapter 4 showed that Mdd encoding performed far better than other encodings in terms of time taken, node count and memory consumed in building the transition relation and reachability set. Also, k-hot encoding showed a gradual improvement in time, memory and node count with increasing values of k.

This thesis can be extended to use existing variable ordering heuristics along with the encoding schemes discussed in Chapter 3. A lot of work has been done regarding ordering of the encoding variables [8, 13]. A good ordering can result in a significantly smaller BDD as opposed to a bad ordering. Thus a good ordering gives scope of being able to encode larger Petri net models. The behaviour of the newly introduced k-hot encoding along with the existing heuristics for ordering encoding variables, could throw more insight on the usability and efficiency of this scheme as compared to the conventional schemes like One-hot.

As discussed in Section 4.9, the encodings can be modified to support dynamic increase in bounds. The models discussed in this thesis have a fixed maximum bound specified for each place. With the option to dynamicaaly increase bounds, we would be able to encode models for which the maximum bound is not known at the start. The suggestions discussed above could make analyzing concurrent systems far more efficient. The first suggestion would help represent even larger concurrent systems using Petri nets. For any system to be modelled using a Petri net, the seconds suggestion would eliminate the need of knowing the bounds on each

place. Only the list of places and transitions would be required, which is much easier to realize for any system.

BIBLIOGRAPHY

[1] J. Babar and A. Miner. Meddly. https://meddly.svn.sourceforge.net/svnroot/meddly.

[2] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications, 1993.

[3] C. Baier and J. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[4] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[5] R. Bryant and A. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

[6] J. Burch, Jr. Edmund M. Clarke, K. Mcmillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[7] N. Chawla and D. Cieslak. *Evaluating Probability Estimates from Decision Trees.* American Association for Artificial Intelligence, 2006.

[8] G. Ciardo, G. Luttgen, and A. Yu. Improving Static Variable Orders via Invariants. *In Application and Theory of Petri Nets*, pages 83–103, 2007.

[9] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation.* Springer, 1998.

[10] R. Drechsler and D. Sieling. Binary Decision Diagrams in Theory and Practice. *International Journal on Software Tools for Technology Transfer*, 3(2):112–136, 2001.

[11] Jr. Edmund M. Clarke, M. Edmund, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] E.Hansen, R. Zhou, and Z. Feng. Symbolic Heuristic Search using Decision Diagrams. *In Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, LNCS 2371:83–98, 2002.

[13] O. Grumberg, S. Livne, and S. Markovitch. Learning to Order Bdd Variables in Verification. *Journal of Artificial Intelligence Research*, 18(1):83–116, 2003.

[14] T. Kam, T. Villa, R. Brayton, and A. Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[15] S. Malik, A. Srinivasan, and R. Brayton. Algorithms for Discrete Function Manipulation. *In Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 90–95, 1990.

[16] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni, and P. Scaglia. Fast Discrete Function Evaluation using Decision Diagrams. *International Conference on Computer-Aided Design*, page 402, 1995.

[17] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[18] A. Miner. Saturation for a General Class of Models. *IEEE Transactions on Software Engineering*, 32:559–570, 2006.

[19] A. Miner. Petri Net Parser. http://sourceforge.net/projects/pnparser/, 2009.

[20] A. Miner and D. Parker. Symbolic Representations and Analysis of Large Probabilistic Systems. *Validation of Stochastic Systems*, LNCS 2925:296–338, 2004.

[21] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[22] E. Pastor and J. Cortadella. Efficient Encoding Schemes for Symbolic Analysis of Petri Nets. *In Proceedings of the conference on Design, automation and test in Europe*, pages 790–795, 1998.

[23] E. Pastor, O. Roig, J. Cortadella, and R. Badia. In Petri Net Analysis using Boolean Manipulation. *15th International Conference on Application and Theory of Petri Nets*, pages 416–435, 1994.

[24] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Data Representation and Efficient Solution: A Decision Diagram Approach. *Formal Methods for Performance Evaluation*, LNCS 4486:371–394, 2007.

[25] J. Peterson. Petri Nets. *ACM Computing Surveys*, LNCS 2371:223–252, 1977.

[26] E. Reinhard, F. Thomas, and T. Dirk. Generating BDDs for Symbolic Model Checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[27] F. Somenzi. CUDD 2.4.2. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.