The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.
SYMBOLIC TECHNIQUES FOR THE PERFORMANCE ANALYSIS OF GENERALISED STOCHASTIC PETRI NETS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ian Davies
January 2001

Supervised by
Prof Pieter S. Kritzinger
Abstract

Binary Decision Diagrams (BDDs) have been successfully used in sequential circuit theory, VLSI, and model checking. They form a highly memory efficient canonical representation of a Boolean function. In this dissertation, following on the success of BDDs in other fields, we investigate the applicability of symbolic techniques in the performance analysis of timed transition systems, particularly those of Generalised Stochastic Petri Nets (GSPNs).

We make use of symbolic methods, where states are represented implicitly rather than explicitly, primarily to conserve memory during the state space exploration process – a necessary step in the performance analysis pipeline. We have investigated the use of BDDs in two different ways. The first, our own novel technique, allows the user to effectively place an upper bound on the amount of memory to use during state space exploration. The second makes use of transition relations to find the successor states at each level of the state graph. Both of these techniques rely on a novel and efficient GSPN to BDD encoding function that we have derived.

We present a discussion on the impact of BDD variable ordering on the time efficiency of state the space exploration. We have developed a technique to find a good ordering of variables based on the static structure of the Petri net, but which does not require the GSPN to exhibit a regular repeating structure. Our techniques are equally applicable to regularly structured GSPNs, as to non-structured GSPNs.

Multi-Terminal Binary Decision Diagrams (MTBDDs) have been suggested as a potential memory-saving alternative to standard sparse matrix packages in solving the continuous time Markov chains (CTMCs) that result from the state space exploration. We discuss our experience with MTBDDs in solving CTMCs and present our proof-of-concept implementation of the conjugate gradient squared (CGS) algorithm, which we have not previously encountered in the literature.
Acknowledgements

The work that follows would not have been possible, or anywhere near as much fun, without the following people:

- Prof. Pieter Kritzinger for his expert supervision, guidance, great ideas, and for always reminding me of the goal of the work throughout this dissertation,
- Dr Markus Siegle for his help and for the initial proposal of this project,
- Dr Andrew Hutchison for his encouragement, and for proof-reading drafts of this dissertation,
- Dr William Knottenbelt for his help and interest in this work, and for responding so quickly and enthusiastically to my questions,
- Mr Marc Daya for his help with the initial concept of the invariant-based state encoding,
- The National Research Foundation for partially sponsoring this work,
- The Computer Science System Administrators, Sam Chetty and Matthew West, for consistently keeping our machines running,
- Elton and Michael, my lab partners and friends, and the rest of the Masters students, who have helped create a fun, friendly, working environment,
- My Family for their consistent love and support throughout the course of this work, even from very far away. Thank you also to my other family, the Jones’, for your help and kindness throughout the past two years,
- Natalie, where to begin? Thank you for being my constant support and guide throughout the course of my degree. Thank you for helping me put this document together. Even more, thank you for your unfailing belief in my abilities, your wonderful friendship, and your love.
Contents

Abstract iii

Acknowledgements iv

1 Introduction 1
   1.1 Motivation and Objective 2
   1.2 Known State Space Exploration Methods 3
   1.3 Dissertation Outline 4
   1.4 Contributions of this Dissertation 4

2 Generalised Stochastic Petri Nets 6
   2.1 Ordinary Petri Nets 6
      2.1.1 Analysis 8
         2.1.1.1 Reachability Graphs 8
         2.1.1.2 Invariants 10
   2.2 Time-Augmented Petri nets 12
      2.2.1 Stochastic Petri nets 12
      2.2.2 Immediate and Timed Transitions 13
      2.2.3 Analysis 14
         2.2.3.1 Simulation 14
         2.2.3.2 Steady State Analysis 15
3 Decision Diagrams

3.1 BDD Theory ................................................................. 16
  3.1.1 BDD Variable Ordering ........................................... 20
  3.1.2 BDD Manipulation Algorithms ................................. 21
    3.1.2.1 The If-Then-Else Algorithm .............................. 22
    3.1.2.2 The Restriction Algorithm ............................... 25

3.2 Application of BDDs .................................................. 27

3.3 MTBDD Theory ............................................................. 29
  3.3.1 Structure of MTBDDs .............................................. 30
  3.3.2 Complexity of MTBDDs .......................................... 32
  3.3.3 MTBDD Manipulation Algorithms .............................. 32
    3.3.3.1 The Apply Algorithm ..................................... 34
    3.3.3.2 The Inner-Product Algorithm ............................ 35
    3.3.3.3 The Vector-Matrix Multiply Algorithm .................. 36

4 Encoding Functions ..................................................... 38

4.1 Determining the Symbolic Representation of a GSPN State ........ 38
  4.1.1 Characteristic Functions ...................................... 38
  4.1.2 Encoding Functions .............................................. 39
    4.1.2.1 Analysis of Encoding Functions .......................... 39
    4.1.2.2 BDDs from GSPN States ................................... 40
  4.1.3 Place-Invariants for Encoding Functions .................... 41

4.2 One-Hot Encoding ...................................................... 42
  4.2.1 One-Hot Encoding a Single Place .............................. 43
  4.2.2 One-Hot Encoding a GSPN ....................................... 43
  4.2.3 Performance of the One-Hot Encoding Method ................ 43

vi
4.3 Binary Encoding ................................................. 44
   4.3.1 Binary Encoding a Single Place ......................... 44
   4.3.2 Binary Encoding a GSPN .................................. 44
   4.3.3 Performance of the Binary Encoding Method ............ 45
4.4 Computational Invariant Encoding ............................. 45
   4.4.1 Use of P-Invariants ..................................... 45
   4.4.2 An Algorithm for the Computational Invariant Encoding 46
   4.4.3 An Example of the Computational Invariant Encoding .... 48
   4.4.4 Performance of the Computational Invariant Encoding Method 48
4.5 Partitions and BDD Ordering .................................. 49
4.6 Conclusion ..................................................... 50

5 State Space Exploration Methods ................................. 51
   5.1 State Space Exploration Techniques .......................... 52
      5.1.1 Exhaustive and Probabilistic Techniques ............... 52
      5.1.2 Parallel and Distributed Techniques .................... 53
      5.1.3 Structural and Decomposition Techniques ............... 55
      5.1.4 Depth-First Exploration ................................ 55
      5.1.5 Breadth-First Exploration ............................... 56
      5.1.6 Recording Transition Rates .............................. 58
   5.2 BDD Data Storage .......................................... 60
      5.2.1 Breadth-First and Depth-First Searches .................. 61
         5.2.1.1 Incremental Reduction .............................. 61
         5.2.1.2 Frontier Set Calculation ............................ 65
         5.2.1.3 Detecting Enabled Transitions ....................... 66
      5.2.2 Vanishing State Elimination ............................ 67
5.3 Transition Relations .................................................. 69
  5.3.1 Basic Theory .................................................... 70
  5.3.2 Example ......................................................... 71
  5.3.3 The Relational Product ....................................... 72
  5.3.4 Partitioned Transition Relations ............................. 73
5.4 Transition Relations for GSPNs ..................................... 75
  5.4.1 Enabling Condition ............................................ 75
  5.4.2 Firing Function ............................................... 78
  5.4.3 Vanishing State Elimination .................................. 79
  5.4.4 Discussion ..................................................... 80

6 Markov Chain Solution Methods ..................................... 84
  6.1 Introduction ....................................................... 84
  6.2 The Method of Jacobi ............................................ 86
    6.2.1 MTBDD Implementation .................................... 87
  6.3 Conjugate Gradient Squared .................................... 88

7 Experimental Results ................................................ 91
  7.1 Performance Metrics ............................................. 91
    7.1.1 Transition Throughput ..................................... 92
    7.1.2 Mean Number of Tokens per Place ........................ 93
  7.2 Experimentation .................................................. 95
    7.2.1 Encoding Mechanism ....................................... 95
    7.2.2 Exploration Techniques ................................... 95
    7.2.3 The Effect of Variable Ordering .......................... 98
    7.2.4 Case Studies ............................................... 101
8 Conclusion

8.1 State Space Exploration Based on BDDs ................................. 112
8.1.1 CTMC Solutions Based on BDDs ........................................ 112
8.2 Future Work ......................................................... 113

A GSPN Models

A.1 The Dining Philosophers Problem ........................................ 114
A.2 The Courier Telecommunications Protocol .............................. 115
A.3 The Kanban Model .................................................... 116
A.4 The Flexible Manufacturing System .................................... 117
A.5 The parsys Model ..................................................... 117

Bibliography ................................................................. 118
Chapter 1

Introduction

This dissertation is concerned with techniques used in the performance analysis of concurrent communicating systems. We need to abstract from complex real-world systems to simplified mathematical models so that performance analysis can be carried out cost-effectively, in software.

Performance analysis is a process which can be broken down into four steps. The first step converts the real-world system to a mathematical model where analysis can be automated. We refer to a modeling formalism which defines the syntactic and semantic structure required to build such models. Some examples of modeling formalisms are finite-state automata, queueing networks, generalised stochastic Petri nets (GSPNs), stochastic process algebras, stochastic activity networks, SDL, and Estelle. For the purposes of this dissertation we have chosen GSPNs as our modeling formalism. GSPNs are graphically very simple, easy to learn, and are backed by mathematical principles.

The second step is the state space exploration of the model. The state of a model is described by the values of one or more state variables. State space exploration begins with the initial state of the model and determines all the states into which the model can evolve via state transitions.

A state graph can be constructed from this exploration, the arcs of which are labeled with the state transition that occurred. Performance variables can be recorded for each state transition describing the associated delay. In the case of GSPNs, these delays are exponentially distributed. Therefore, the state graph represents the underlying continuous-time Markov chain of the modeled process.

In the third step of performance analysis the Markovian steady state equations are solved and the stationary probability distribution for the states of the model derived. Finally, in the fourth step
these steady state probabilities are used to derive the performance metrics of the model, such as the transition throughput or the relative load on the different parts of the model. These metrics are then related back to the real-world system, and can be used to find design faults or system inefficiencies.

1.1 Motivation and Objective

There is a wide variety of tools available for the performance analysis of concurrent communicating systems [32]. These tools are all confronted by the same problem: state space explosion. This is where the number of states a system can increase exponentially with an increase in the number of parallel components [24] of the system. One does not typically create Markov chains by hand; one uses a high-level modeling tool, such as we have described, and generates the Markov chain from this. However, even apparently simple models can result in very large state graphs which cannot be stored in the memory of modern computer systems.

The state space explosion problem restricts the complexity of the system one is capable of analysing. Several methods already exist ranging from disk-based Markov solvers to tools which exploit the regularity of model structure to conserve memory. Disk-based methods are inherently as slow as the disk IO, and models do not always provide a uniform structure. There is a need for new techniques to reduce the impact of the state space explosion problem on memory resources.

This research was conducted in the Data Networks Architectures (DNA) Group at the University of Cape Town where tools have previously been created to this end. DNAmaca [32, 60], a Markov chain solver written in 1996 by W. Knottenbelt made use of probabilistic techniques to perform state space exploration. DNAmaca, even with a probabilistic search algorithm, still suffers from the state space explosion problem, slowing dramatically once virtual memory is required to store the found states.

We thus required an experimental tool-set to carry out research on new techniques for conserving memory during state space exploration, to improve upon the space efficiency of DNAmaca. The DNA Group has additionally created two tools for the creation and analysis of GSPNs: DNAnet, and DaNAMiCS. For direct compatibility with these tools our experimental tool-set has GSPNs as its underlying formalism.

The primary focus of this dissertation is then the efficient exploration of GSPN state spaces. We realise, however, that the technique used in finding the state space affects the efficiency of the
CHAPTER 1. INTRODUCTION

Markov solution process. We therefore discuss state space exploration techniques in the context of the performance analysis process. We present our investigation into Markov chain solution methods using our state space exploration storage method.

1.2 Known State Space Exploration Methods

The state of a system is represented by a state descriptor. This descriptor could be a string or list of integers and can, more generally, be a vector of bits. How the state of the system is translated, or encoded into a bit-vector, is discussed in Chapter 4. Classical well-known state space exploration methods have used two data structures for storing state descriptors: hash-tables, and search trees.

A hash-table is a data structure which stores data in key-value pairs. Each stored data element has an associated key which represents its location in the hash-table. The state descriptor is hashed using a one-way hash-function onto an integer value in the range \( \{0,l-1\} \) where \( l \) is the size of the hash-table. This value is then the key. It is possible for two distinct state descriptors to hash onto the same integer value and thus be stored in the same location in the array. This event, called a collision, can be dealt with in a number of ways depending on the state space exploration strategy. With reference to [61], we mention a probabilistic exploration strategy in Chapter 5, § 5.1.1.

Hash-tables are superior to simpler structures such as linked-lists because they provide \( O(1) \) access to any stored element, as opposed to linked-lists’ \( O(n) \) access. However, hashing techniques require large amounts of memory to be pre-allocated for the array which is larger than the number of states actually found in order to minimise collisions.

Search trees are sometimes preferred to hash-tables as a means to store the state space because they are a dynamic data structure, whereas with hash-tables, one must make an initial guess at the size of the state space and pre-allocate large amounts of memory for the exploration.

A commonly used search tree is the AVL tree [49], which is a balanced binary search tree where, whenever a new element is inserted, the tree must be re-balanced by "rotation" algorithms. The main disadvantage of AVL trees is the need for the extra balancing information which must be stored at each node, and overhead of re-balancing the tree.

The techniques we have researched for state space exploration are symbolic, where states are represented and stored implicitly instead of being individually enumerated. We have researched and implemented BDDs and an extension of BDDs called Multi-Terminal BDDs in our own software tool-set.
CHAPTER 1. INTRODUCTION

1.3 Dissertation Outline

We have organised this dissertation as follows. Chapter 2 contains a brief introduction to GSPNs. We follow with the theory of BDDs and MTBDDs in Chapter 3. Having covered the basic theory we then show, in Chapter 4, how GSPN markings can be symbolically represented by BDDs. Chapter 5 provides the theory for state space exploration and discusses classical state space exploration techniques. Later in that chapter, we describe two of the techniques we have researched and implemented in our tool-set, the first based on storage, and the later based on the dynamic behaviour of GSPNs. State space exploration using BDDs as the storage structure has, to our knowledge, not been done before in the way presented in this dissertation. Next, in Chapter 6, we show how Markov chains can be solved using MTBDD-based techniques. We present our implementation of the CGS Krylov subspace method which we have not found previously in the literature. Chapter 7 describes how we are able to calculate performance metrics using BDD-based techniques and shows our experimental test results from all our algorithms, referring to several GSPN models which can be found in Appendix A. Finally, Chapter 8 contains our conclusion, discussion of some results, and recommendations for future work.

1.4 Contributions of this Dissertation

Specific contributions of this dissertation are:

- We have described a technique for encoding the state descriptor of a GSPN based on its invariant constraints. This technique is similar to that of Haverkort [8] and Pastor [18], but uses heuristics to further save on the number of bits for the bit-vector state-descriptor of the GSPN. We could not find experimental results on bit-vector sizes in either of [8] or [18], so we compared our technique to that of a simple binary encoding and found significant improvements in the range of 30% to 80%. Refer to Chapter 4, Section 4.4.

- Realising that BDDs slow down the state space exploration by virtue of the fact that they are reduced, we implemented a technique which uses unreduced BDDs and compresses this data structure when it reaches a certain size. Implementing this technique allowed us to effectively place an upper-bound on the amount of volatile memory to be used by the state space exploration. This technique is novel in this respect, and certainly much more efficient than using transition relations in the context of GSPNs. Refer to Chapter 5, Section 5.2.1.1.
• We implemented two automated BDD variable ordering techniques with varying success. The technique of interleaving places in similar subnets of the GSPN gave the best results, with diminishing success for less regular models. Refer to Chapter 4, Section 4.5.

• Borrowing from the work of Siegle et al [24], we implemented the Jacobi iterative method for solving steady state equations of the form $Ax = b$. In addition to this we investigated the applicability of Krylov subspace techniques with MTBDDs as the underlying data structure. We found that MTBDDs were too slow for performing vector matrix multiplications, and suggest that one reverts to classical Markov chain solution methods until a more efficient MTBDD-based algorithm can be found. Refer to Chapter 6.
Chapter 2

Generalised Stochastic Petri Nets

This chapter contains the basic theory of Generalised Stochastic Petri nets — our chosen formalism for experimentation with state space exploration techniques. We first present ordinary Petri nets and describe their functionality. We then describe GSPNs, which have extended the original formalism by adding timing information, used for performance analysis.

2.1 Ordinary Petri Nets

Ordinary Petri nets (OPNs)\(^1\), simply known as Petri nets (PNs), are a graphical and mathematical technique for describing and analysing concurrent communicating systems. A states\(^2\) in Petri nets is represented by a vector of tokens on places. Structural information about the modeled system can be gained from the Petri net at design time and this can assist in the model solution process.

A Petri net is a weighted, directed, bipartite graph containing places and transitions. Weighted arcs may connect places to transitions or vice versa, but no two nodes of the same type may be connected. The marking \((M)\) of a Petri net assigns each place a nonnegative integer \(k\) and we say that if a place \(p\) is assigned the value \(k\), then it is marked with \(k\) tokens and we write \(M(p) = k\). The initial marking \((M_0)\) defines the starting distribution of tokens over the places of the net. Formally, a Petri net is a 5-tuple, \(PN = (P, T, F, W, M_0)\), where:

\(^1\)Petri nets are sometimes referred to as "ordinary" in the literature to distinguish them from time-augmented Petri nets, discussed in Section 2.2

\(^2\)States are also referred to as markings in the PN literature. In this dissertation we use the words interchangeably.
CHAPTER 2. GENERALISED STOCHASTIC PETRI NETS

\[ P = \{p_1, ..., p_m\} \text{ is a finite set of places} \]
\[ T = \{t_1, ..., t_n\} \text{ is a finite set of transitions} \]
\[ F \subseteq (P \times T) \cup (T \times P), \text{ is a set of arcs, the flow relation} \]
\[ W : F \rightarrow \mathbb{N} \text{ is a weight function: the weights of each of the arcs} \]
\[ M_0 : P \rightarrow \mathbb{N} \text{ is the initial marking} \]
\[ P \cap T = \emptyset \text{ and } P \cup T \neq \emptyset \]

(1)

Graphically, places are represented by circles and transitions by rectangular boxes. Arcs are lines with arrows between the places and transitions. The weight on the arc is shown by a numeric value unless it has the value one in which case the weight is usually omitted. The number of tokens on a place is shown by black dots for small values or numerically for larger values.

The pre- and post-sets of a particular place or transition are defined as follows:

- \( t \) is the preset of transition \( t \). It is the set \( p \subseteq P \) of places which have arcs to the transition \( t \). The places in the preset are sometimes called the input-places.
- \( t^* \) is the postset of transition \( t \). It is the set \( p \subseteq P \) of places which have arcs from the transition \( t \). Correspondingly, the places in the postset are sometimes called the output-places.
- \( p \) is the preset of place \( p \). It is the set \( t \subseteq T \) of transitions which have arcs leading to the place \( p \).
- \( p^* \) is the postset of place \( p \). It is the set \( t \subseteq T \) of transitions which have arcs leading from the place \( p \).

Thus far we have only considered the static structure of Petri nets. The dynamic behaviour is described by the flow relation which determines how many tokens are consumed from and produced onto the places of the net when a transition fires. The transition firing rules are as follows:

1. Transition \( t \) is enabled if \( \forall p \in P : M(p) \geq w(p, t) \), where \( w(p, t) \) is the weight of the arc from place \( p \) to transition \( t \). We write \( M[t > \) to show that transition \( t \) is enabled in marking \( M \).
2. A transition, \( t \), may fire if is is enabled.
CHAPTER 2. GENERALISED STOCHASTIC PETRI NETS

3. If enabled transition $t$ fires it removes $w(p, t)$ tokens from each of its input places and adds $w(t, p)$ tokens to each of its output places. $w(t, p)$ is the weight of the arc from transition $t$ to place $p$. We use the notation $M[t > M']$ to show that transition $t$ fires at marking $M$ and results in successor marking $M'$.

The presence or absence of tokens on a place in the Petri net indicates the value of a system condition or variable. The firing of transitions indicates the changing of state as events occur.

Petri nets can become large and unmanageable for real-world systems. The use of hierarchical modeling techniques structures such as subnets or Coloured Petri nets [29] alleviates this but the analysis of Petri nets by hand is far too complex and needs to be automated. We discuss this in the next section.

2.1.1 Analysis

Correctness analysis of Petri nets determines the functional properties of Petri nets. For example, we can determine whether our net could ever become deadlocked, or if any of the places could become unbounded. These terms are described in the following subsections.

The functional properties of a Petri net can be determined via exhaustive exploration of the reachability set [29]. The reachability set of a Petri net PN, denoted $R(PN)$, is the set of markings reachable from the initial marking via a sequence of transition-firings. This exhaustive analysis is highly memory-intensive and time-consuming. Some functional properties of Petri nets may be found by another less time-consuming method of analysis known as invariant analysis.

2.1.1.1 Reachability Graphs

If a marking $M'$ is immediately reachable from a marking $M$ we denote this as $M \rightarrow M'$ and this shows one-step reachability. To extend this we introduce $\rightarrow^*$ which is the reflexive and transitive closure of $\rightarrow$. In other words, $M \rightarrow^* M$ and $(M_i \rightarrow^* M_k) \land (M_k \rightarrow^* M_j) \Rightarrow (M_i \rightarrow^* M_j)$.

Now, for any marking $M'$, we say it is reachable from $M$ iff $M \rightarrow^* M'$.

The reachability graph is a directed, labeled graph of all the markings reachable from the initial marking. The labels on the arcs of the graph indicate which transition fired in order to change the marking.
CHAPTER 2. GENERALISED STOCHASTIC PETRI NETS

In the functional analysis of Petri nets, using reachability graphs, we are concerned with finding various properties of Petri nets such as liveness, the existence of home states, boundedness, and persistence. Formally,

1. A transition \( t \) is live iff \( \forall M \in R(PN) : \exists M' \in R(PN) : M \rightarrow^* M' \) and \( M'[t] > 0 \). It follows that a Petri net, \( PN \), is live iff all transitions \( t \in T \) are live. Said informally, a Petri net is live if, from any marking, it is ultimately possible, through some firing sequence, to enable every transition in the net.

2. A marking \( M \in R(PN) \) is a home state iff \( \forall M' \in R(PN) : M' \rightarrow^* M \).

3. A Petri net, \( PN \), is \( k \)-bounded iff \( \forall p \in P : \exists k \in \mathbb{N}_0 : \forall M \in R(PN) : M(p) \leq k \). Petri nets can be unbounded if there is some repeatable firing sequence which continually produces more tokens on a certain place or set of places. In this way that place or set of places could eventually contain an infinite or unbounded number of tokens.

4. A Petri net is said to be persistent if, for any two enabled transitions, the firing of one transition will not disable the other.

There are other functional properties of Petri nets which we do not cover here such as fairness and reversibility. The properties mentioned above can be found by a thorough analysis of the reachability graph [37].

If a Petri net is bounded it is also live if and only if all transitions appear as a label on the arcs of all final strongly connected components of the reachability graph. A final strongly connected component is defined [46] as \( R' \subseteq R(PN) \) iff \( \forall M \in R', M' \in R(PN) : (M \rightarrow^* M' \Rightarrow M' \rightarrow^* M \lor M' \in R') \). It follows that all states are home states if the reachability graph contains exactly one final strongly connected component.

Persistence is checked by iterating through each marking of the reachability set. Where there is more than one concurrently enabled transition one checks, for each child marking of that marking, if the same set of transitions is enabled except for the transition which caused the marking change from parent to child.

Reachability graph generation is very time-consuming and memory intensive. We now examine a simpler, more time- and space-conservative method called invariant analysis. Unfortunately invariant analysis cannot determine all the functional properties of Petri nets and one must still perform
the full reachability graph generation for complete results. However, some properties can be efficiently determined using invariants.

2.1.1.2 Invariants

Invariant theory allows one to derive functional properties of Petri nets by observing their static structure. Invariant analysis operates by viewing the Petri net static structure as a matrix equation.

The matrix is called the incidence matrix and describes the number of tokens that are consumed from and produced onto the places of the net. For a Petri net with \( n \) transitions and \( m \) places, the \((m \times n)\) incidence matrix \( C = [c_{ij}] \) where each \( c_{ij} \) is given by:

\[
c_{ij} = c_{ij}^+ - c_{ij}^-
\]  

Each \( c_{ij}^- = w(i, j) \) which defines \( C^- \) the backward incidence matrix where \( w(i, j) \) is the weight of the arc from the place \( i \) to the transition \( j \). Similarly, \( c_{ij}^+ = w(j, i) \) defines \( C^+ \) the forward incidence matrix where each \( w(j, i) \) is the weight from the transition \( j \) to the place \( i \).

Given the incidence matrix one can define formulas to represent the dynamic behaviour of Petri nets. Let a transition \( t_j \) be represented by the length \( n \) unit-vector \( e_j \) which is zero in all components except the \( j^{th} \) component, then transition \( t_j \) is enabled at a marking \( M \) if:

\[
M \geq e_j \cdot C^-
\]  

where \( M \) is a vector of length \( m \). It follows that enabled transition \( t_j \) fires at a marking \( M \) yielding a marking \( M' \), described by the following equation:

\[
M' = M + Ce_j
\]  

For a sequence of transition firings \( \sigma = t_1, t_2, ..., t_k \) we define a length \( n \) firing vector \( f = e_1 + e_2 + ... + e_k \) and generalise Equation 4 for an arbitrary number of transition firings to:

\[
M' = M + Cf
\]  

Of special interest are the cases where Equation 5 has the property that \( M' = M \). If \( Cf = 0 \) then the vector \( f \) is called a T-Invariant, or transition invariant, and represents a sequence of transition
firings which ultimately leave the marking unchanged. By solving the equation \( Cf = 0 \) we can find the Petri nets T-invariants. We say that a Petri net, \( PN \), is covered by positive T-invariants iff \( \forall t_i \in T : \exists \) T-invariant \( f \in \mathbb{Z}^n \) with \( f \geq 0 \) and \( w_i > 0 \), and with this property it holds that if a Petri net is live and bounded it is covered by positive T-Invariants.

Of more direct relevance to our work are P-Invariants or place invariants. P-Invariants can be derived my multiplying Equation 5 on the left by a length \( m \) vector \( v^T \) to yield:

\[
v^T M' = v^T M + v^T Cf
\]

Then if \( v^T C = 0 \) we say \( v^T \) is a P-Invariant and note that a Petri net, \( PN \), is covered by positive P-Invariants iff \( \forall p_i \in P : \exists \) P-Invariant \( v^T \in \mathbb{Z}^m \) with \( v^T \geq 0 \) and \( v_i > 0 \). A well-known result is that if a Petri net is covered by positive P-Invariants then it is bounded [46]. By multiplying each P-Invariant by the initial marking \( (v^T \cdot M_0) \), where \( M_0 \) is vector of length \( m \), we find an upper-bound on the number of tokens that can exist over the set of places \( p_i \in P \forall (v_i \neq 0) \).

Further, we can derive a set of equations of the form:

\[
C_{p_1}p_1 + \ldots + C_{p_m}p_m = U
\]

where each \( C_{p_i} \) is the coefficient of the \( i^{th} \) component in the respective P-Invariants, \( p_i \) represents each place of the Petri net and \( U \) is the value \( v^T \cdot M_0 \). \( U \) represents the highest number of tokens which can exist over the set of places in the invariant equation with non-zero coefficients. In this way we can find a maximum number of tokens which can exist on a place, which is very important for GSPN analysis using BDD data structures, described in Chapter 4.

Finding P-Invariants is an \( O(m^3) \) operation where \( m \) is the number of places in the Petri net. Using invariants as a boundedness check therefore saves time compared to an exhaustive exploration. However, it is important to note that if a Petri net is not covered by positive P-Invariants, it is not necessarily unbounded [46].

Once it is known that the system being modeled functions correctly, we can evaluate its performance. For example, we wish to be able to guarantee, mathematically, that a system can turn a certain number of jobs around in a certain amount of time, or that the load on certain parts of a system can never exceed some maximum.

Timing information must be added to the model for this kind of analysis. The dynamic behaviour of the resulting formalism is representative of a stochastic process and is called Stochastic Petri nets.
(SPNs). With SPNs we can derive performance results from our modeled system. In the next section we explain the time-augmentation of Petri nets; we introduce both Stochastic and Generalised Stochastic Petri nets.

2.2 Time-Augmented Petri nets

Timing information can be added to Ordinary Petri nets in several ways. One can either specify sojourn times on the tokens in places of the net so that there is some delay between when the tokens arrive at a place and when they are allowed to leave. These Petri nets are called Timed-Place Petri nets as the timing information is associated with each place.

There are two ways to associate timing information with the transitions of a Petri net. The first kind of Timed-Transition Petri net makes use of a preselection model. In a preselection model, when a transition becomes enabled it selects the tokens it requires from its input places and prevents other transitions from consuming them. Once its prespecified delay has elapsed it consumes the tokens and produces tokens on its output places as specified in the firing rules on page 7.

The second kind of Timed-Transition Petri nets uses a race model. Multiple transitions may be simultaneously enabled and rely on the same set of input places. Further, the Petri net in question may not be persistent in which case the firing of one of the enabled transitions first may disable one or more of the other concurrently enabled transitions. In a race model when a transition becomes enabled a timer starts. When the timer reaches zero the transition may fire provided it is still enabled and another transition has not consumed the tokens it needed in the meantime. The concurrently enabled transitions then race for the available tokens.

2.2.1 Stochastic Petri nets

Stochastic Petri nets (SPNs) make use of a race model. Each transition, \( t_i \), is assigned a possibly marking dependent firing variable called its firing rate, \( \lambda_i \), which affects the probability of it firing first at some marking. The firing time is exponentially distributed and the distribution of the random variable \( \chi_i \) of transition \( t_i \) is given by:

\[
F_{\chi_i}(x) = 1 - e^{-\lambda_i x}
\]
CHAPTER 2. GENERALISED STOCHASTIC PETRI NETS

This means that two or more transitions can be simultaneously enabled and one will fire first with a negative exponentially distributed probability, possibly disabling one or more of the others. There is therefore a sojourn time at each marking. The probability that transition $t_i$, with rate $\lambda_i$, fires first at a particular marking is given by:

$$P[t_i \text{ fires first at marking } m] = \frac{\lambda_i}{\sum_{t_j \in EN(m)} \lambda_j}$$  \hspace{0.5cm} (9)

where $EN(m)$ is the set of transitions enabled at marking $m$. Finally, the sojourn time spent in marking $m$ can be derived from the minimum of the independent exponentially distributed firing times:

$$P[\min(\chi_1, \ldots, \chi_n) \leq x] = 1 - e^{-(\sum_{i} \chi_i)x}$$  \hspace{0.5cm} (10)

With this convention in place, we can incorporate time into the Petri net modeling formalism simply by adding rates to each of the transitions.

2.2.2 Immediate and Timed Transitions

When modeling it is often convenient to only associate firing times with the more time-consuming processes in the system, that is, those events which have the largest impact on the system performance [36]. For this reason we introduce a class of transitions which fire in zero time. These transitions are called immediate transitions while the transitions associated with a firing rate are called timed transitions and the resultant new class of Petri nets with these two types of transitions are called Generalised Stochastic Petri nets (GSPNs).

The firing conditions for GSPNs are also modified in that immediate transitions always have priority over timed transitions. If no immediate transitions are enabled at some marking, the timed transitions compete for the enabling tokens in a race model as with SPNs. If one immediate transition is enabled at some marking it disables any concurrently enabled timed transitions. If more than one immediate transition is enabled at a marking there is a probability distribution or weight function defined over the immediate transitions which determines, probabilistically, which immediate transition will fire first. Formally, a GSPN is a 4-tuple, $GSPN = (PN, T_1, T_2, W)$, where:
CHAPTER 2. GENERALISED STOCHASTIC PETRI NETS

$PN$ is the underlying OPN
$T_1 \subseteq T$ is the set of timed transitions, $T_1 \neq \emptyset$
$T_2 \subseteq T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T = T_1 \cup T_2$
$W = (w_1, ..., w_n)$ where $w_i \in \mathbb{R}^+$
$W$ specifies the firing delays for timed transitions and the firing weight of immediate transitions

These extensions simplify the modeled SPN significantly and reduce the size of the reachability graph, as some markings are no longer reachable with the additional enabling constraint created by the differing priorities of transitions. It can be shown that GSPNs with their dual-priority transitions have equivalent modeling power to that of Turing machines [29].

2.2.3 Analysis

Timed Petri net analysis or performance evaluation provides metrics important for modern concurrent communicating systems. One can determine how often certain transitions fire, or how much time is spent in certain markings. In this way we can detect the potential over-utilisation of resources or the presence of bottlenecks in the real-world system. We will examine each of these concepts in detail with specific reference to GSPNs.

There are two methods for performance analysis of GSPNs: simulation and steady state analysis.

2.2.3.1 Simulation

Simulation is probably the most popular way of analysing real-world systems, despite its complexity and expense [7]. Typically, there is already a system in place to which time-stamps can be added and used in the analysis.

Simulation techniques in Petri nets exist for conducting quick performance tests. They are generally inaccurate and should only be used in the prototyping phase of Petri net modeling. A simulation of the GSPN starts with the initial marking and selects one of the transitions to fire, based on the enabled transitions' relative rates. This process is repeated at each successive marking for a certain number of firings. One records how often each marking is visited and how often each transition is fired. The higher the number of firings, the more accurate the results will be. Unfortunately, to obtain sufficiently accurate results, the firing sequence length would have to be very large.
2.2.3.2 Steady State Analysis

SPNs are isomorphic to continuous-time Markov chains (CTMCs [36]) and a \( k \)-bounded SPN is isomorphic to a finite Markov Chain. We are concerned with ergodic Markov chains which originate from ergodic SPNs. An ergodic SPN is simply a PN that contains home states. From an ergodic SPN we can determine the steady state probability distribution of markings by the matrix equation \( \pi Q = 0 \) and the condition that \( \sum_i \pi_i = 1 \) where \( \pi \) is the vector of steady state probabilities and \( Q \) is the *infinitesimal generator* whose elements show the transition of one marking to another as derived from the reachability graph.

A one-to-one correspondence exists between GSPNs and stochastic process states [36]. The two types of transitions in GSPNs provide two types of markings in the reachability graph. Those markings which enable immediate transitions are called *vanishing states*. This is because the transitions fire after zero time units. Correspondingly, markings which enable only timed transitions are called *tangible states*. Before the steady state computation is performed the vanishing states are eliminated.
Chapter 3

Decision Diagrams

BDDs have proved to be extremely versatile and efficient data structures in the area of model checking and VLSI. The purpose of this dissertation is to investigate the applicability of BDD methods to GSPNs for performance analysis. MTBDDs have been used successfully [20] in the algebraic manipulation of matrices. In this chapter we present the background theory of BDDs and MTBDDs necessary to understand our work.

3.1 BDD Theory

Binary Decision Diagrams (BDDs) are graph-based representations of Boolean functions [9]. They were first introduced by C.Y. Lee [35] in 1959 and further popularised by S.B. Akers [4] in 1978. In 1986 Randal E. Bryant published a seminal paper [9] on how the existing BDD structure could be modified to improve the compactness and time efficiency of the BDD manipulation algorithms. He placed ordering and reduction constraints (see page 18) on the nodes of the data-structure and formulated “Reduced Ordered Binary Decision Diagrams” or ROBDDs. These structures have become so integral to the area of sequential circuit verification, logic synthesis and process modeling and analysis, that they are now known simply as “BDDs”, with the assumption that the structure will always be reduced and ordered.

The origin of BDDs lies in the Shannon Cofactor Expansion of a Boolean function. Given a Boolean function $f : B^n \rightarrow B$, where $B$ is the range $\{0, 1\}$, the cofactors of $B$ are the resultant functions obtained by the partial evaluation of $f$. Thus, given any Boolean function $f(x_1, \ldots, x_n)$ we may use
any Boolean variable $x_i$, where $1 \leq i \leq n$, and write:

$$f \mid_{x_i=b} (x_1, ..., x_n) = f(x_1, ..., x_{i-1}, b, x_{i+1}, ..., x_n)$$

(12)

where $x_i$ is replaced by the constant $b$ and the result is the cofactor or restriction of $f$. Further, the Shannon Cofactor Expansion of $f$ with respect to a variable $x_i$ is given by:

$$f = x_i f \mid_{x_i=1} + \overline{x_i} f \mid_{x_i=0}$$

(13)

Carrying out this expansion recursively yields a binary tree with leaves 0 (false), or 1 (true)$^1$, where each internal node in the tree represents a function $f$. The low child of that node represents $f \mid_{x_i=0}$, and the high child $f \mid_{x_i=1}$, where this particular node is at depth $i$ in the BDD.

A BDD is a rooted, directed, acyclic graph (DAG). It contains a set of vertices $V$, each element of which has a certain index that describes its height in the BDD and a unique identifier that distinguishes it from the other vertices. The index of a BDD vertex $v$ can be obtained by the function $index(v)$, while the unique identifier of $v$ is obtained by the function $id(v)$.

There are two types of vertices in a BDD: non-terminal vertices that have two children vertices; and terminal vertices that have a certain Boolean value but no children. All BDD vertices can have any number of parent vertices, depending on the BDD structure. The two children of the non-terminal vertices are called the high and low child and, at a vertex $v$, are defined respectively by the functions $high(v)$ and $low(v)$. The value of the terminal vertex is in the Boolean range \{0, 1\} and, for a terminal vertex $v$ is defined by $value(v)$. This is clarified by the object-diagram in Figure 1.

Alternatively, a BDD is a function graph and it is defined recursively as follows:

1. if $v$ is a terminal vertex:

   (a) if $value(v) = 1$ then $f_v(x_1, ..., x_n) = 1$

   (b) if $value(v) = 0$ then $f_v(x_1, ..., x_n) = 0$

2. if $v$ is a non-terminal vertex with $index(v) = i$, then $f_v$ is defined by the function $f_v(x_1, ..., x_n) = \overline{x_i} \cdot f_{low(v)}(x_1, ..., x_n) + x_i \cdot f_{high(v)}(x_1, ..., x_n)$

---

$^1$Throughout this dissertation, we shall commonly use the value 0 (1) in mathematical formulae and the term false (true) in textual explanations.
We now discuss Bryant's extensions to this basic structure which allow more efficient operations to be performed on Boolean functions. The ordering extension, proposed by Bryant, imposes a restriction on the order vertices of the DAG such that, for any vertex \( v \), \( \text{index}(v) < \text{index}(\text{high}(v)) \) and \( \text{index}(v) < \text{index}(\text{low}(v)) \). The ordering of BDD nodes can have a dramatic effect of the size of the resultant BDD and thus an optimal ordering for a Boolean function should be chosen. Unfortunately, the selection of the appropriate ordering is itself an NP-Complete problem and so, in practice, a set of heuristics are used to decide upon the best ordering. The subject of ordering of BDD vertices will be dealt with in depth later in Chapter 5.

Subgraphs \( G \) and \( G' \) are isomorphic if there is a one-to-one mapping from each of the vertices in \( G \) onto each of the vertices in \( G' \). Isomorphism is defined recursively in that every vertex \( v \) in \( G \) should have a corresponding vertex \( v' \) in \( G' \) where the subgraphs of \( v \) and \( v' \) are themselves isomorphic. Bryant proved [9] that for any Boolean function, there is a unique, reduced, ordered BDD that represents that function.

Bryant defined a BDD as being reduced if it conforms to the following two requirements:

1. the BDD contains no vertex \( v \) such that \( \text{low}(v) = \text{high}(v) \) and,

2. no two vertices \( v \) and \( v' \) have isomorphic subgraphs.

Reduced function graphs form canonical representations of Boolean functions. This means that
two functions are equivalent if and only if the reduced, ordered BDD representation for the two functions are isomorphic.

Each level, or index of the BDD, represents a variable in the Boolean function. To demonstrate, consider the Boolean function: \( abc + \overline{a}bc + \overline{a}\overline{b}c \). The BDD for this function is shown in Figure 2. In this diagram, circles indicate internal or non-terminal vertices and the square represents the true terminal vertex. We have omitted the false terminal vertex, and all lines to it for clarity. Solid lines indicate high children, and lines through circles indicate low children. Level, or index, 1 corresponds to \( a \), index 2 corresponds to \( b \), and index 3 corresponds to variable \( c \). By tracing all the paths of the BDD which lead from the root to the true terminal, we build up all the possible bit sequences which, when substituted into the equation, would be equivalent to true. In our example there are three possible bit-sequences: 111, 101, and 001 which are exactly the sequences which would evaluate our function to true. If one follows the low child from the root there is only one possible path to the true terminal. If one follows the high path there is also only one root to the true terminal but this path does not visit any vertices at BDD index 2. For this path index 2 is a don't care index which means that, on this path, the value of the variable indexed by this BDD level does not affect the outcome of the Boolean function and its value can therefore be true or false.

![Figure 2: A simple reduced ordered BDD of the function \( abc + \overline{a}bc + \overline{a}\overline{b}c \).](image)

BDDs are a powerful data structure with the property that they can actually become smaller when more data is added to them. We show a graph in Figure 3 to demonstrate this. This graph shows the insertion of all the integer values between 0 and 4096 inclusive into a BDD with height 12. Each integer is stored in the BDD by its unsigned binary representation. The integer values are inserted in random order. We see that the number of vertices in the BDD grows rapidly at first to some peak value when approximately half of the integers have been inserted. The BDD size then drops off, eventually to zero, as all the integers are inserted. When we analyse the efficiency of BDD
algorithms we are interested in the peak BDD size and not the final BDD size as this shows the maximum amount of memory that will be consumed.

Figure 3: Inserting random states into a BDD.

3.1.1 BDD Variable Ordering

Essential in any introduction to BDDs is an understanding of the impact of variable ordering on the size of the BDD in terms of the number of vertices. Bryant explained [9] that, depending on how one chooses to order the variables making up the Boolean function, the number of nodes making up the BDD can be severely affected. Therefore, in order to achieve the smallest representation of the Boolean function using BDDs, one should pick the optimal ordering.

Unfortunately, Bryant also showed that finding the optimal ordering of variables is an NP-hard problem. At best, we can apply heuristics, based on our knowledge of our problem domain, to the ordering problem to find a good ordering that is not necessarily optimal. To illustrate this problem consider the two BDDs shown in Figure 4.
CHAPTER 3. DECISION DIAGRAMS

Figure 4: Showing how variable ordering can affect the size of the representative BDD.

This case is taken from Bryant’s paper [9] to illustrate the point that a poor choice of variable ordering can have disastrous effects on the memory efficiency of BDD-based techniques. The BDD on the left corresponds to the Boolean function: \( x_1 x_2 + x_3 x_4 + x_5 x_6 \), where the nodes at level \( i \) correspond to variable \( x_i \) as is the case for the BDD on the right which represents the function \( x_1 x_4 + x_2 x_5 + x_3 x_6 \). All that differs between these functions is a permutation of their variables and consequently the order in which they are evaluated in their respective BDDs, yet the BDD on the right has 15 vertices, while the one on the left has only 7.

It is evident that well-chosen variable ordering is necessary for the success of the application of BDDs. We will discuss variable ordering in detail in Chapter 4 for our particular case where we try to exploit the structure of GSPNs.

3.1.2 BDD Manipulation Algorithms

The Boolean operators and (\( \land \)), or (\( \lor \)), and the negation or complement unary operator, represented by a solid bar over the complemented variable, may be applied to BDD operands. In 1990, Brace et al [34] devised efficient techniques for manipulating Boolean functions based on the Boolean ternary operator if-then-else. They developed an algorithm, the if-then-else-algorithm or ite-algorithm, which allows one to perform most Boolean operations on BDDs. We will show here how the if-then-else algorithm functions, as well as the BDD restriction algorithm.
CHAPTER 3. DECISION DIAGRAMS

\[ \text{Table 1: Boolean operations performed using the ite operator.} \]

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ite(0, 0, 0)</td>
</tr>
<tr>
<td>( F \land G )</td>
<td>ite(F, G, 0)</td>
</tr>
<tr>
<td>( F \lor G )</td>
<td>ite(F, 1, G)</td>
</tr>
<tr>
<td>( F )</td>
<td>ite(F, 0, 1)</td>
</tr>
<tr>
<td>( F \oplus G )</td>
<td>ite(F, G, G)</td>
</tr>
<tr>
<td>( F \land \overline{G} )</td>
<td>ite(F, G, 1)</td>
</tr>
<tr>
<td>( F \lor \overline{G} )</td>
<td>ite(F, 0, G)</td>
</tr>
</tbody>
</table>

3.1.2.1 The If-Then-Else Algorithm

The if-then-else algorithm carries out the Boolean ternary operation \( R = \text{ite}(F, G, H) \) on input BDDs \( F, G, \) and \( H \):

\[
\text{ite}(F, G, H) = (F \land G) \lor (F \land H) \quad (14)
\]

With Equation 14 one can derive any of the three basic Boolean operators \( \land, \lor, \) and \( \neg \). For example, \( (F \land G) \equiv \text{ite}(F, G, 0) \), and \( (F \lor G) \equiv \text{ite}(F, 1, G) \). A full list of equivalences is shown in Table 1. The if-then-else algorithm thus provides us with an efficient base for performing Boolean operations on BDDs. The algorithm pseudocode is shown in Figure 5.

The algorithm takes three input BDDs and creates a reduced resultant BDD. We cater for each of the two reduction properties of BDDs described earlier by maintaining a unique table and by checking each vertex's high and low children before it is inserted into the BDD. No vertex \( v \) should exist such that \( \text{high}(v) = \text{low}(v) \) and, to this end, on line 9, we compare the then and else vertices, and if they are the same only returns one of them.

The unique table is a hashtable which keeps a record of all the created vertices. This table is checked on every call to \( \text{ite}(F, G, H) \) by a call to a function \( \text{findOrAddUniqueTable} \) to check if this combination has previously been called. \( \text{findOrAddUniqueTable} \) compares the input BDDs \( F, G, \) and \( H \) and determines whether this combination has previously been calculated. If so, the previously calculated vertex is returned, along with its unique subgraph.

The call to \( \text{findOrAddUniqueTable} \) is only called on line 11, after the recursive calls to \( \text{ite} \) on lines 7 and 8. This may lead to unnecessary computation and thus an efficient hash lookup table, the computed table is implemented. The computed table is a hash table with no collision management.
CHAPTER 3. DECISION DIAGRAMS

function \text{ite}(f : \text{Vertex}, g : \text{Vertex}, h : \text{Vertex}) : \text{Vertex}
begin
1 \text{if}(\text{terminal case})
2 \quad \text{return result}
3 \text{if}(\text{computed table has entry } \{f, g, h\})
4 \quad \text{return result}
5 \text{else}
6 \quad \text{let } v \text{ be the top variable of } \{f, g, h\}
7 \quad t = \text{ite}(f_v, g_v, h_v)
8 \quad e = \text{ite}(f_w, g_w, h_w)
9 \quad \text{if}(t = e)
10 \quad \text{return } t
11 \quad r = \text{findOrAddUniqueTable}(v, t, e)
12 \quad \text{insertComputedTable}(\{f, g, h\}, r)
13 \text{return } r
end.

Figure 5: BDD if-then-else Algorithm.

which means that new table entries simply replace existing entries. Unique entries are inserted into the computed table at the end of every call to \text{ite} on line 13 by the function \text{insertComputedTable}. More importantly, this table is checked first at line 3 on every call to \text{ite} to avoid but not eliminate unnecessary computation. The computed table increases the efficiency of the \text{ite} algorithm providing an efficient base for Boolean function manipulation.

The if-then-else algorithm has time complexity $O(|F| \cdot |G| \cdot |H|)$. But in \text{and} and \text{or} operations, the complexity is reduced to $O(|F| \cdot |G|)$, because the third BDD contains only one vertex, the \text{true} or \text{false} terminal depending on the operation. The complement operation can be performed using \text{ite}(F, 0, 1) which has order $O(|F|)$. But there is a simplification to this which allows us to simply swap the terminal values of \text{true} and \text{false}, and this operation is $O(1)$.

For example, consider the BDDs in Figure 6. The leftmost BDD represents the BDD for the Boolean function $P = abc + \overline{ab}c$ while the second BDD represents the Boolean function $Q = a\overline{bc}$, where level 1 in the diagram corresponds to variable $a$, level 2 to variable $b$, and so on. The number to the upper-right of the non-terminal node is an identification number, used for clarity in the explanations that follow. Dotted lines represent low children and solid lines represent high children.

We wish to combine these functions using the Boolean \text{or} ($\lor$) operator. As can be seen in Table 1, This is equivalent to \text{ite}(P, 1, Q). We will demonstrate how the \text{ite}-algorithm performs this
computation on BDDs.

Figure 7 shows the execution trace of the recursive function \texttt{ite}. The values shown in brackets indicate the parameters that \texttt{ite} is called with and we note that the \textit{true} terminal is denoted with a \texttt{T} and the \textit{false} terminal is denoted by an \texttt{F}. The numbers in the brackets refer to the \texttt{ids} of the vertices from the BDDs \texttt{P} and \texttt{Q}.

For example the first element in the initial parameter set \( (0, T, 0) \) refers to the vertex of \texttt{P} with \texttt{id} 0, the root vertex of \texttt{P}. The second parameter in the set is \texttt{T} which refers to the \textit{true} terminal. This originates from the equivalence Table 1 explained above. The last value in the parameter set is 0 referring to the 0\textsuperscript{th} vertex of \texttt{Q} which is the root node of \texttt{Q}.

The value shown in the circle above the parameters on certain levels of the execution trace indicates the \texttt{id} of the vertex that was created at that level of the \texttt{ite} functions progression. This \texttt{id} corresponds to a vertex in BDD \texttt{a} of Figure 8. Each of these values is inserted into the unique and computed hash tables. At certain levels of the execution trace we notice that parameter combinations have been visited before and the result for this combination is simply returned from the computed table.

At this point the resultant BDD of the operation would be shown by BDD 1 in Figure 8. This BDD represents all the possible paths of the result BDD, but is not reduced. The call to \((F, T, 2)\) at node 4 in the execution trace shows the use of the unique table which gives us the desired reduction. When this result is calculated, the call to \texttt{findOrAddUniqueTable} returns \((3, T, F)\) as the result for
3.7: The execution trace of the recursive function \( \text{ite} \) called with \( \text{ite}(P, \text{true}, Q) \).

\( (F, T, 2) \). This is because the subgraphs of \( (3, T, F) \) and \( (F, T, 2) \) are isomorphic. In each case their low child is the \text{true} terminal and their high child is the \text{false} terminal. The BDD at this stage of the computation is then shown by BDD 2 in Figure 8. When returning from the call to \( (2, T, 1) \) we find that its high child and its low child are identical, and thus by the comparison of high and low on line 9 of the algorithm in Figure 5 we only return the high child. In this way we build the reduced function graph for the result of the or operation which is shown by BDD 3 of Figure 8.

3.1.2.2 The Restriction Algorithm

There are many BDD manipulation algorithms, but we will only consider one more here: the BDD Restriction algorithm. Restricting a Boolean function has the effect of setting one of the Boolean argument variables to \text{true} or \text{false}, thereby eliminating it from the function. In BDDs this eliminates an entire level and returns a result BDD for the function with that represented variable removed.

The importance of the restriction algorithm is that it allows one to perform quantification over Boolean variables. For example, the Boolean function \( \exists v(f) \), where \( v \) is some Boolean variable and \( f \) is a Boolean function, is equivalent to \( f|_{v=0} \lor f|_{v=1} \) which is simply two restriction operations.
3. \( f|_{v=b \in B} \), where \( B = \{0, 1\} \), denotes the assignment of value \( b \) to Boolean variable \( v \) in function \( f \). The algorithm for restriction is shown in Figure 9.

The restriction algorithm has complexity \( O(|F|) \), where \( |F| \) is the height of the BDD. The algorithm proceeds by traversing the BDD. Since the BDD is strictly ordered it checks at each level to see whether any of the children vertices \( v \) of that level have \( \text{index}(v) = \text{restrictionIndex} \). The \text{restrictionIndex} is the index corresponding to the variable of the Boolean function that must be eliminated. If any of the child vertices meet this criterion then that child is eliminated in the following manner. If the variable is to be restricted to \text{false} then the pointer to the child is replaced by the child’s low pointer. Similarly, if the variable is to be restricted to \text{true} then the pointer to the child is replaced by the child’s high pointer.

**Multiple Quantifications**

Our use of BDDs requires the ability to quantify out more than one variable at a time. It is inefficient to run the restriction algorithm repeatedly for each variable we wish to quantify. We show an algorithm for computing the BDD with a set of its variables quantified in one pass of the BDD. We called the algorithm \text{restrictSet} and the pseudocode is shown in Figure 10.

Like the if-then-else algorithm, we make use of the \textit{unique} hash table. Not only does this ensure that our result will be a canonical representation of the Boolean function, it also avoids the recomputation of common paths of the BDD. Line 4 of the algorithm performs the quantification of the specific variable: \( f|_{v=0} \lor f|_{v=1} \) and lines 5 to 7 define the case when the current variable does not need to be quantified.


3.2 Application of BDDs

BDDs have been used very successfully in electronic circuit theory [31, 38], and in VLSI [9]. The memory efficient representation of Boolean functions, and the time efficient algorithms for manipulating them have resulted in very large state spaces being explored (more than $10^{20}$ states). We have investigated techniques to apply these advances to state space exploration of general GSPN models to try to achieve the same benefits. In this section we briefly describe the basic principles behind using BDDs in the state space exploration of GSPNs and other abstraction formalisms.

Coudert and Madre demonstrated in [38] that BDDs could be used to represent sets. Set elements can be enumerated and thus represented by some sequence of bits, or bit-vector. Each representative bit-vector represents a particular path from the root of the BDD to the true terminal. For example, the BDD in Figure 2 has possible paths: 111, 101, 001, each of which could represent some element of a set.

A Markov chain consists of a number of states and the transitions between them. Depending on the modeling formalism, each state of the system can be described by the values of its state variables.
These values make up what is known as the state descriptor. In GSPNs, this state descriptor is the marking, the numbers of tokens on each of the places of the GSPN. If we can represent the state descriptor of a model by a bit-vector, then it follows that we can "store" that state descriptor in a BDD taking advantage of the compactness BDDs offer.

This is exactly what happens in practice. One finds a Boolean function which evaluates to true given a particular set of Boolean input parameters. These parameters form the bit-vector for the state we wish to store and the Boolean function which evaluates them is the characteristic function of that state. The characteristic function, like any other Boolean function, is represented by a BDD. Coudert and Madre [38] showed that BDDs representing sets can be manipulated by the same operators that are used for Boolean algebra. For example, it can be shown that logical \textit{and} is equivalent to set intersection and logical \textit{or} is equivalent to set disjunction.

BDDs can not only be used as a storage site for the explored states of the model, they can also be used to represent the dynamic behaviour of the model by representing the models’ transitions. These BDDs are called \textit{transition relations} and they are able, by means of Boolean operators, to
CHAPTER 3. DECISION DIAGRAMS

determine the successor states of any particular state or set of states in a model. Each path in these
transition relation BDDs represents a potential mapping of one set of state variables onto another.
Our experience with using transition relations in the context of GSPN state space exploration is
discussed further in Chapter 5.

BDDs can therefore be used for the state space exploration of GSPN models. We have to be careful
in their use, however, as poor choices of exactly how the state descriptors are encoded into BDDs,
and how they are stored can dramatically affect the performance of the BDD structure in terms of
both memory and time efficiency. These considerations form a major part of this dissertation and
are expanded upon in detail in the chapters to follow.

We now describe MTBDDs, a variant of BDDs that have been used to efficiently store sparse ma-
trices [20]. We have investigated the applicability of MTBDDs in the steady state solution of the
CTMC derived from the state space exploration of GSPN models. The theory of CTMCs is well
presented in [36].

3.3 MTBDD Theory

MTBDDs are an extension of BDDs proposed in 1993 by McGeer et al [20]. Previous work by
McGeer showed that BDDs could be used to represent binary matrices, with entries in the Boolean
range \{0, 1\}. A BDD of depth \(n\) could be used to represent binary matrices of dimension \(2^{n/2} \times 2^{n/2}\)
or binary vectors of length \(2^n\). Further, it was suggested that there is no reason that BDDs only be
restricted to having terminal values of zero or one and that they need not only have two terminals.
[20] extends this idea by observing that any finite set could be used for the values of the terminal
nodes.

McGeer et al called this BDD expansion MTBDDs, or Multi-Terminal Binary Decision Diagrams.
McGeer shows that MTBDDs are a viable data structure for matrix representation and manipulation.
McGeer contends that MTBDDs are a superior data structure particularly for the representation of
very sparse matrices. In fact, in the context of MTBDDs, the matrix need not only be sparse. If
several of the elements are the same then a further advantage can be gained over standard sparse
matrix packages. Finally, McGeer proves that, in the worst case, MTBDDs are superior to sparse
matrix representations in terms of memory consumption.

\(^2\)A sparse matrix is defined to be a matrix where the number of zero-valued entries is much larger than the number of
non-zero valued entries.
Bahar et al [50] introduces the concept of applying BDDs to any algebra, and using them in a wide variety of symbolic algorithms. They called their resultant structures Algebraic Decision Diagrams (ADDs). The structure of ADDs is essentially the same as that of MTBDDs but [50] provides algorithms for solving a more general class of problems. We focus on the algorithms presented by McGeer et al as our use for MTBDDs is centred on solving matrix problems and we find the algorithms presented in [20] highly intuitive and easily adaptable to a wide class of matrix problems.

3.3.1 Structure of MTBDDs

An MTBDD represents a function of the type \( f : \{0, 1\} \rightarrow \mathbb{R} \). More generally, MTBDDs can represent functions of the form \( f : \tilde{D} \rightarrow \mathbb{R} \) where \( \tilde{D} \) is any finite set. One can encode the members of \( \tilde{D} \) using \( \lfloor \log_2 |\tilde{D}| \rfloor \) Boolean variables. In the case where \( \tilde{D} \) is the set of integers \( \{0, \ldots, n - 1\} \), then the MTBDD represents a vector of length \( n \). In the case where \( \tilde{D} \) is the finite set \( \{0, \ldots, n - 1\} \times \{0, \ldots, m - 1\} \), then the MTBDD represents a matrix of dimension \( n \times m \).

Thus, we can encode the individual positions of elements in a vector or matrix by a unique combination of Boolean variables. As an example, consider a vector of length \( m \). \( \lfloor \log_2 m \rfloor \) bits can be used to encode the positions of the vector. Figure 11 shows the MTBDD for the vector \((x_{00}, x_{01}, x_{10}, x_{11})\), where the two-bit index for each variable defines how the variable is stored. For example, terminal node 2, shown as a square, is on the path 01 from the root.

![MTBDD Diagram](Image)

Figure 11: An MTBDD for the vector \((1, 2, 3, 4)\).

Note that this MTBDD, like BDDs, is both reduced and ordered. The ordering constraint of MTBDDs is the same as for BDDs where there is a strict and constant order on the depth of the variables in the BDD used in the representation. The reduction constraint is identical: no two subgraphs of nodes at the same depth of the MTBDD may be isomorphic, and no node may exist such that its low
and high child are the same. In this way, MTBDDs form a canonical representation of the vector or matrix they encode. It is important to note that information about the length of the vector is lost in the MTBDD representation, and this must be somehow stored along with the data structure.

![MTBDD Diagram](image)

Figure 12: An MTBDD for the matrix in Equation 15.

In Figure 12 there is an MTBDD for the matrix:

\[
\begin{pmatrix}
1_{00} & 2_{01} & 3_{02} & 4_{03} \\
5_{10} & 6_{11} & 7_{12} & 8_{13} \\
9_{20} & 10_{21} & 11_{22} & 12_{23} \\
13_{30} & 14_{31} & 15_{32} & 16_{33}
\end{pmatrix} \tag{15}
\]

Each element of the matrix is subscripted with its position in the matrix. The position is a combination of integers which we can encode in a binary fashion in order to represent them with an MTBDD. A commonly used heuristic in the implementation of MTBDD packages [24] is to interleave the row and column variables of the MTBDD. Experience has shown that this leads to a more compact representation. Consequently the element 7 at position (1, 2) in the matrix is encoded by the bit-sequence 0110. This is made more concrete by the following example. A matrix can be thought of as a function \( f(x_0, y_0, x_1, y_1, \ldots) \) where the binary variables are interleaved. In this way we can describe the cofactors of the matrix \( M \) as follows:

\[
\begin{pmatrix}
f_{x_00y_0} & f_{x_00y_1} \\
f_{x_0y_0} & f_{x_0y_0}
\end{pmatrix} \tag{16}
\]

where each submatrix can be similarly decomposed. It is important to note that, since we represent
an MTBDD with Boolean variables, the dimension of every matrix we represent is always a power of two, and the excess entries we obtain in this way are automatically "padded" with zero elements.

3.3.2 Complexity of MTBDDs

As is the case with BDDs, MTBDDs gain much from the sharing of nodes by more than one parent. Different cofactors of the Shannon Expansion map onto identical functions. This means that MTBDDs are not only, on the face of it, a good representation for sparse matrices, where the zero entry is often repeated, but they are also a good representation for matrices where there are frequently repeated entries, such as block matrices\(^3\) or band matrices\(^4\). This property extends to the matrices for CTMCs, where the rates of a comparatively small number of transitions are stored.

In terms of the space complexity of MTBDDs, [50] claim, without proof, that ADDs, their analog of MTBDDs, are in the worst case, of equivalent size to classical sparse matrix representations. [20] proves that the MTBDD representation of a matrix of dimension \(n\), and containing \(m\) non-zero elements, has space complexity \(O(m \log n)\). Further, [20] proves that if \(R\) is any representation of a matrix of dimension \(n\) and contains \(m\) non-zero elements where \(m \ll n\), then there is at least one matrix \(M\) of dimension \(n\) and \(m\) non-zero elements such that \(|M|_R = O(m \log n)\). This implies that MTBDDs are the optimal representation for very sparse matrices. We will deal with the time complexity of MTBDD manipulation algorithms in our experimental results in Chapter 7.

3.3.3 MTBDD Manipulation Algorithms

We use MTBDDs for the representation and solution of CTMCs. How we accomplish this is described in detail in Chapter 6. In this section we present the main algorithms we will use in the solution process.

All the algorithms are formulated using the principles of recursion and cofactor expansion. The recursive nature of the algorithms is easily seen in each case by observing that the partial evaluation of the MTBDD for a matrix or vector reveals the submatrix or subvector for that evaluation, and a full evaluation of the MTBDD reveals a particular matrix or vector element. A partial evaluation of an MTBDD would involve setting certain of the variables to certain values, similar to the BDD restriction algorithm.

\(^3\)Block matrices have regularly repeated "blocks" of common entries.
\(^4\)Band matrices have regular repeated bands, diagonal or otherwise, of common entries.
MTBDDs gain much of their computational power from the fact that some subgraphs have identical parents. Consider the MTBDD in Figure 13 for an example of this. This MTBDD represents the \( 4 \times 4 \) matrix:

\[
\begin{pmatrix}
1 & 2 & 0 & 0 \\
3 & 4 & 0 & 0 \\
0 & 0 & 1 & 2 \\
0 & 0 & 3 & 4 \\
\end{pmatrix}
\]  \hspace{1cm}  (17)

Every node in the MTBDD represents a unique function and the result of applying this function is calculated exactly once and can be reused from any context, or parent node. This means that no operation is ever performed twice. This has two benefits, the first obviously being the speed of execution, and the second being that the MTBDD result graph of an operation is guaranteed as being unique. In practice this is carried out by using hashing tables, exactly as it was with BDDs. Each operation on every subgraph is “memorised” and can simply be returned if that specific set of parameters is called again. There are two hash tables: the \textit{computed} table which is implemented as a closed hash function with no collision detection; and the \textit{unique} table which ensures that the reduced property of MTBDDs is maintained.

It must be noted that some operations on MTBDDs are extremely simple to carry out and highly efficient. For example, consider scalar multiplication of matrices or vectors. This involves a simple update of each of the terminal values. A simple addition to the MTBDD implementation would add the terminal vertices to a list structure for direct linear access. Termwise element inversion can also
be carried out in this way. We have implemented our MTBDD package in this way along with this functionality.

3.3.3.1 The Apply Algorithm

The \textit{apply} algorithm, depicted in pseudocode in Figure 14, is for termwise operations such as matrix or vector addition or subtraction. The recursive procedure is realised by observing, for a matrix, that for $f \circ g$, where $\circ$ is any termwise operator, that:

$$
\begin{pmatrix}
\begin{array}{cc}
    f_{x0y0} & f_{x0y0} \\
    f_{x0y0} & f_{x0y0}
\end{array}
\end{pmatrix} \circ 
\begin{pmatrix}
\begin{array}{cc}
    g_{x0y0} & g_{x0y0} \\
    g_{x0y0} & g_{x0y0}
\end{array}
\end{pmatrix} = 
\begin{pmatrix}
\begin{array}{cc}
    f_{x0y0} \circ g_{x0y0} & f_{x0y0} \circ g_{x0y0} \\
    f_{x0y0} \circ g_{x0y0} & f_{x0y0} \circ g_{x0y0}
\end{array}
\end{pmatrix}
$$

(18)

\begin{verbatim}
function apply(f : Vertex, g : Vertex, o : Operator) : Vertex
    begin
        1    if (computed table has entry (f.g, operation)) return pre-computed result;
        2    if f is terminal
            begin
                3            if (g is a terminal) return o(f, g);
            4        else result = new NonTerminal(index(g), apply(f, low(g), o), apply(f, high(g), o));
            end;
        5    elseif (g is a terminal)
            result = new NonTerminal(index(f), apply(f, low(g), o), apply(f, high(g), o));
        6    elseif (index(f) == index(g))
            result = new NonTerminal(index(f), apply(low(f), low(g), o), apply(high(f), high(g), o));
        7    elseif (index(f) \neq index(g))
            result = new NonTerminal(index(f), apply(low(f), g, o), apply(high(f), g, o));
        8        else
            result = new NonTerminal(index(g), apply(f, low(g), o), apply(f, high(g), o));
        9    result = findOrAddUniqueTable(result, low(result), high(result));
        10   insertComputedTable(result);
        11  return result;
    end.
\end{verbatim}

Figure 14: An MTBDD termwise \textit{apply} Algorithm.

The formulation for a vector is very similar. Line 1 of the algorithm in Figure 14 is the case where this combination of parameters has been encountered before and thus the result of that computation can simply be returned. Line 3 is the base case where both of the MTBDDs have recursed down to terminal values. When both $f$ and $g$ are terminal values we can simply apply the operator to
3.3.3.2 The Inner-Product Algorithm

Performing vector inner-products with MTBDDs provides a very neat and elegant algorithm, which we have developed and implemented. The recursive nature of the algorithm can be built from the following basis, given two vectors \( f \) and \( g \) constructed respectively over the variables \( (x_0, \ldots, x_{n-1}) \), and \( (y_0, \ldots, y_{n-1}) \):

\[
f \cdot g = \sum_{i=0}^{n-1} x_i \cdot y_i
\]

An algorithm for finding inner-products can easily be derived and is shown in Figure 15. Note that the function \( \text{innerProduct} \) returns a terminal vertex representative of the scalar value. This is simply so that the inner production algorithm can also make use of the computed table, which stores MTBDD vertices. Once again, in this algorithm all MTBDD subgraphs are only computed once. We have not shown all the individual cases in the example as we did for the \( \text{apply} \) algorithm. In our implementation we made use of a function to automatically determine which vertices should be recursed upon, and which should remain the same. This also leads to a simpler algorithmic representation.

```plaintext
function innerProduct(f : Vertex, g : Vertex) : TerminalVertex
begin
1    if(computed table has entry \((f, g)\)) return pre-computed result;
2    if(f and g are both terminals) return \(f \cdot g\);
3    let \(v\) be the top variable of \(f\) and \(g\);
4    result = innerProduct\((f_v, g_v)\) + innerProduct\((f_{\bar{v}}, g_{\bar{v}})\);
5    result = findOrAddUniqueTable(result, \(low(result)\), \(high(result)\));
6    insertComputedTable(result);
7    return result;
end.
```

Figure 15: The Inner-Product Algorithm.
3.3.3.3 The Vector-Matrix Multiply Algorithm

The previous algorithms operate on vectors or matrices whose length or dimension is the same. We mentioned earlier that the length of a vector or the dimension of a matrix had to be stored somehow, as it is lost during the encoding process. When we wish to multiply a vector by a matrix, we need to somehow keep track of the size and shape of the vector and matrix we are multiplying.

```python
function vectorByMatrix(v : Vertex, m : Vertex, i : Integer) : Vertex
begin
1  if(computed table has entry (v, m)) return pre-computed result;
2  if(v and m are both terminals) return v \cdot m;
3  r_1 = apply(vectorByMatrix(v_{x_1}, m_{z_{i+1}}, i+1), vectorByMatrix(v_{x_i}, m_{z_{i+1}}, i+1), +);
4  r_2 = apply(vectorByMatrix(v_{x_i}, m_{z_{i+1}}, i+1), vectorByMatrix(v_{x_i}, m_{z_{i+1}}, i+1), +);
5  result = new NonTerminal(index(t), r_1, r_2);
6  result = findOrAddUniqueTable(result, low(result), high(result));
7  insertComputedTable(result);
8  return result;
end.
```

Figure 16: The MTBDD-based vector-by-matrix Algorithm.

Vector-matrix multiplication can be generalised to the following form, given a vector \( v \) of length \( n \), and a matrix \( M \) of dimension \( n \times m \):

\[
h(y_0, \ldots, y_{m-1}) = v(x_0, \ldots, x_{n-1}) \times M(x_0y_0, \ldots, x_{n-1}y_{m-1})
\]  

(20)

where \( h \) is the resultant vector of length \( m \). The recursive nature of the algorithm can easily be derived from Equation 20's simplified component-wise form:

\[
(h_{y_0}, h_{y_0}) = (v_{y_0}, v_{x_0}) \times \begin{pmatrix} M_{x_0y_0} & M_{z_0y_0} \\ M_{x_0y_0} & M_{y_0y_0} \end{pmatrix}
\]  

(21)

and from this we can write:

\[
h_{y_0} = v_{y_0} \cdot M_{x_0y_0} + v_{x_0} \cdot M_{x_0y_0}
\]

\[
h_{y_0} = v_{y_0} \cdot M_{x_0y_0} + v_{x_0} \cdot M_{x_0y_0}
\]  

(22)

From Equation 22 it is easy to derive the recursive algorithm for vector by matrix multiplication which is shown in Figure 16. This algorithm requires that we keep track of our relative position in
the vector. This is done by the index integer \( i \) which starts at zero and is incremented for each step. The index integer is necessary on Line 5, where a new non-terminal vertex must be created at that index in the result MTBDD.

In this algorithm, unlike in the \textit{apply} algorithm, the separate cases of having the \( v \) vertex with an index lower than the \( m \) vertex, or vice versa, are handled implicitly, as in the inner-product algorithm. This algorithm, like all the others also makes use of the unique hash table to ensure that a reduced MTBDD is built, and a computed hash table so that computation of shared subgraphs of the MTBDD is not unnecessarily re-applied.
Chapter 4

Encoding Functions

Central to the concept of using Binary Decision Diagrams for GSPN state-space exploration is the *encoding function*. Encoding functions allow one to symbolically represent a state by its *characteristic function* which, in turn, can be represented by a BDD. We will see in this chapter how the encoding function determines the height of the BDD. It is important to minimise the BDD height as it directly influences the time efficiency of BDD manipulation algorithms.

In Section 4.1 we discuss some general encoding principles and the use of Place-Invariants in encoding functions. In Sections 4.2 through 4.4 we discuss various encoding mechanism and their relative usefulness in state-space explorations using BDDs.

4.1 Determining the Symbolic Representation of a GSPN State

It has been shown previously (p. 16) that Binary Decision Diagrams (BDDs) are symbolic representations of a Boolean function. We wish to use BDDs to store all the possible states a particular GSPN can reach from its initial state. Thus, we must convert each state to a function that represents that state. This function is referred to as a *characteristic function*, and the function for converting the state to a characteristic function is called an *encoding function*.

4.1.1 Characteristic Functions

The characteristic function is based on the work of O. Coudert et al [38].

38
A Boolean function consists of Boolean variables and operators. i, an interpretation, is a function that associates each variable of a Boolean function with a value true or false. i evaluates a Boolean function by performing the operations of the logical connectors and yielding a true or false value for the function for a specific combination of Boolean variables. We say an interpretation, \( i \), satisfies a function \( c \) iff \( i(c) = \text{true} \). This means that the specific combination of true/false variables implied by \( i \) evaluates the function \( c \) to \( \text{true} \). The notation we use for this is \( \models_i c \).

A Boolean function \( c \) over variables \( b_1...b_n \) describes a function from \( \{0,1\}^n \) into \( \{0,1\} \). We consider \( c \) as its characteristic function. This defines a subset of \( \{0,1\}^n \) which we note as \( \chi(c) \).

\[
\chi(c) := \{i([b_1...b_n]) | \models_i c(b_1,...,b_n)\}. \tag{23}
\]

In other words, \( \chi(c) \) is the on-set of \( c \), that is: those variable combinations which are interpreted as true. For example, \( a \lor \neg b \) denotes the characteristic function of the set \( \{[1,0], [1,1], [0,0]\} \). We can thus define the characteristic function of a set and encode sets of Boolean variables in BDDs. In the next sections we discuss the encoding function which completes the mapping from a GSPN marking into a BDD.

### 4.1.2 Encoding Functions

All we require of an encoding function is a one-to-one mapping of GSPN states into \( \{0,1\}^n \). In the following section we present various techniques for this. The central aim of any encoding function should be to minimise the number of Boolean variables required in the mapping. This will, in turn, reduce the resultant height of the BDDs used to encode the state spaces. Shallower BDDs can store fewer nodes and limit how deeply a recursive manipulation algorithm can progress. Particularly in our implementation, discussed in the next chapter, the depth of the BDD directly affects the time taken to complete the state space exploration.

#### 4.1.2.1 Analysis of Encoding Functions

All that remains before presenting the actual techniques is to discuss measures to determine which encoding mechanisms are preferable. For our purposes, the most important metric is the height of the resultant BDD. The complexity of algorithms to determine the encoding and actually encode a GSPN marking are also an important consideration, but to a lesser extent as the vast majority of time
4.

For determining what constitutes a minimal BDD we use the following measure:

\[ EP = \frac{S_r}{S_p} \]  

(24)

where \( EP \) stands for Encoding Performance, \( S_r \) stands for the number of states actually reachable by a given GSPN, and \( S_p \) is the total number of states it can potentially identify uniquely using the encoded height of the BDD. Normally \( EP \ll 1 \) but, the closer this \( EP \) value is to one, the better the encoding.

4.1.2.2 BDDs from GSPN States

Mapping GSPN states into \( \{\text{true}, \text{false}\}^n \) in practice terms means representing each state by some unique bit-sequence. The uniqueness of these bit-sequences is obviously essential and we will show in the following sections how this can be achieved. For now, we will focus on the conversion process from bit-sequence to BDD.

Assume that some GSPN state is mapped into 10110. By taking the first (leftmost) variable as the root of a BDD, we see that the “root” of this bit-sequence is \text{true}. The resultant BDD would then have a non-terminal high child and its low child would be the \text{false} terminal. The \text{false} terminal in a BDD traversal implies the non-existence of a state, while the \text{true} terminal implies the existence of a state. So, following the high child to the second variable, represented by the next BDD index, we find that it is set to \text{false}. The root’s high child would then have a non-terminal low child, and its high child would also be the \text{false} terminal.

The BDD that results from continuing this process is shown in Figure 17. Notice that the added \text{true} terminal node increases the height of the BDD by one, thus a bit-sequence of length \( n \) would result in a BDD of height \( n + 1 \). This BDD only has one possible path to the \text{true} terminal, and thus only one state. The combination of decisions involved in traversing the BDD from root to \text{true} terminal builds a path from root to terminal and derives the state’s bit sequence. In the example, we would traverse as follows: high, low, high, high, low. This implies the above bit-sequence 10110 which, as we will see below, defines some state which is reachable by the GSPN being analysed.
The BDD in Figure 17 represents one state only. In practice there will be many states in a BDD. Thus, all paths from the root of a BDD to the true terminal indicate a reachable state. For example, the BDD shown in Figure 18 contains the states represented by the bit-sequences 10110, 10101, 11110 represented by the three possible paths from the root to the terminal node.

4.1.3 Place-Invariants for Encoding Functions

In most encoding methods we need to determine how many variables, or bits\(^1\), to allocate for each place in order that it be uniquely identified for each possible number of tokens it can contain.

Using Place-Invariants\(^2\) described in Chapter 2, we are able to find upper bounds of the total number of tokens that is allowable on a given place. From Equation 7, on page 11, it follows that the upper bound on a particular place, \(p_k\), is defined by:

\[^1\text{For simplicity, we refer to the encoding as a bit-sequence.}\]
\[^2\text{Place-Invariants are called P-Invariants, and for simplicity we will refer to them as “invariants”.}\]
4. \[ p_k = \frac{U}{(C_{p_k})} \]  

This defines the maximum number of tokens that can exist on a place assuming all other places constrained by this invariant, contain zero tokens.

Place-Invariants are not only used to help decrease the number of bits to use in an exploration using BDDs, they are essential to the proper functioning of the algorithm. We must know, before the exploration begins, how many bits we will use at most. It is not straightforward to dynamically increase the number of bits used, or the height of the BDD at run-time. State-space explorations can then safely be run only on bounded GSPNs, or rather, GSPNs whose places are covered by positive P-Invariants. If invariants are not known, then intelligent assumptions must be made about the maximum number of tokens that a place can hold.

In the sections that follow we present differing techniques for encoding GSPN states. As an example we use Figure 19 which shows the GSPN solution of the Readers/Writers problem where there are 7 potential readers and 3 potential writers.

![Figure 19: GSPN solution for the Readers/Writers problem.](image)

### 4.2 One-Hot Encoding

The one-hot encoding mechanism [18] is perhaps the simplest mechanism for encoding GSPN state spaces. One cannot simply enumerate each state as one is not certain, at the start, of the total number of states that will be explored and thus how many bits to allow for the height of the BDD. The one-hot encoding mechanism provides a simple means of representing tokens on places by bit-sequences.
4.2.1 One-Hot Encoding a Single Place

In the one-hot encoding, if the upper bound on a place is \( b \), then we must reserve \( b \) bits for that place. Then if that place contains \( i \) tokens, the \( i \)th bit is \textit{true} and all the others \textit{false}. This uniquely identifies the marking of the individual places.

Using the GSPN for the Readers/Writers problem shown in Figure 19, we will encode place \( p_0 \). First, however, we must determine the upper bound on the number of tokens \( p_0 \) can contain. This can be determined from the P-Invariants:

\[
\begin{align*}
p_0 + p_1 &= 7 \\
p_3 + p_4 &= 3 \\
p_1 + p_2 + 7p_4 &= 7
\end{align*}
\]

(26) \hspace{1cm} (27) \hspace{1cm} (28)

Place \( p_0 \) only appears in equation 26 where, by applying the equation 25, we find the upper-bound to be 7. If the place appeared in another equation, we take the highest bound calculated from all the equations in which it is a variable. We thus allocate 7 bits for place \( p_0 \). For the initial marking place \( p_0 \) contains 7 tokens, thus the bit sequence would be 1000000. If it contained 3 tokens the bit sequence would be 0000100. A string of all zeros indicates that the place contains no tokens.

4.2.2 One-Hot Encoding a GSPN

We have shown the encoding for a single place, and the encoding for the GSPN state is obtained by simply concatenating the bit-sequences for each of the places in some order. This order is important and optimising it is an NP-complete problem [9]. We can, however, find some heuristics for helping us to find the best order.

If we encoded all the places of the GSPN in Figure 19 and concatenated the individual bit-sequences, the resultant bit-sequence for the initial marking would be \( 1000000_{(p_0)} 000000_{(p_1)} 1000000_{(p_2)} 100_{(p_3)} 000_{(p_4)} \). In total this is 27 bits. This implies a BDD height of 28 as the additional level in the directed acyclic graph is for the terminal node.

4.2.3 Performance of the One-Hot Encoding Method

The total number of reachable states for the GSPN in Figure 19, on page 42, is 9. With 27 bits one can uniquely identify \( 2^{27} \) states. Applying formula 24 we find that the \( EP \) for the one-hot encoding
of this GSPN is $6.71 \times 10^{-8}$.

This method is thus not practical for use as the BDDs used in the exploration would be unmanageably large despite the fact that the one-hot encoding is very simple to implement and there is little overhead in determining the state encodings.

4.3 Binary Encoding

The binary encoding [17] is an improvement to the one-hot encoding in terms of the number of bits required to store the tokens on a place.

4.3.1 Binary Encoding a Single Place

A binary encoding means converting the decimal number of tokens on a place to its unsigned binary equivalent. Once again, the invariants are used to determine the upper-bound on the maximum number of tokens that can exist on a place. If $k$ is the maximum number of tokens that can exist on a place, $p$, then we need to reserve

$$b = \lceil \log_2 k \rceil$$  \hspace{1cm} (29)

bits for that place, where $b$ is the number of bits. For the GSPN in Figure 19 we determined in the previous section that place $p_0$ could maximally contain 7 tokens. Thus we must reserve 3 bits for place $p_0$. The initial marking of $p_0$ is $7_{10}$, which would be encoded as $111_2$.

4.3.2 Binary Encoding a GSPN

Applying the same principle as for the one-hot encoding we concatenate all the bit sequences from each of the places in order to form the encoding for the entire Petri net. The initial marking’s bit-sequence for the Petri net in Figure 19 is thus $111_{(p_0)}000_{(p_1)}111_{(p_2)}11_{(p_3)}00_{(p_4)}$. This is 13 bits in total, which implies a BDD height of 14 including the terminal vertex.
4.3.3 Performance of the Binary Encoding Method

As was previously mentioned, the total number of reachable states for the GSPN in Figure 19 is 9. With 13 bits we are able to uniquely identify 8192 states. This yields an encoding performance from Equation 24 of $1 \times 10^{-3}$. This is still not a good encoding performance ratio, but it significantly improves upon the one-hot encoding method. Additionally, there is very little extra overhead in computing the binary equivalents of the integer number of tokens.

4.4 Computational Invariant Encoding

The one-hot encoding mechanism, described previously has the advantage of simplicity of implementation but results in very deep BDDs that slow down any manipulation algorithms applied to them, and thus reduce the efficiency of a state-space exploration algorithm. Using a binary encoding gave a moderate improvement in space complexity by using the unsigned binary representation of the number of tokens on each place in the GSPN marking. Both of these methods yield unmanageably large BDDs and a significant improvement on this is required in order to make BDDs a viable means of exploring GSPN state spaces. The method presented in this section, the Computational Invariant Encoding method developed as part of the research of this thesis, uses the static GSPN information obtained from the P-Invariants of a GSPN to reduce the number of places that need to be stored to represent a given state.

The method described here is more algorithmic in nature and is not a straight encoding from a set of place markings into a set of Boolean variables. With this method we need to process the marking, using P-Invariants, before encoding it as its binary representation. In this section, we will refer to P-Invariants simply as invariants.

4.4.1 Use of P-Invariants

Invariants provide one with a static view of the GSPN. They define maximum upper-bounds on a place or set of places in the GSPN. We used this invariant information in the binary encoding in order to decide how many bits to allocate to store the number of tokens on each place. In this encoding method we divide the places of the GSPN into two categories: those which must be stored, and those which can be computed.
CHAPTER 4. ENCODING FUNCTIONS

Using the initial marking of a GSPN one can define invariant equations. In Equation 7 on page 11, $C_{p_k}$ is the invariant coefficient of place $p_k$ and $U$ is the constraint determined over the maximum number of tokens in the set of places $\{p_1, \ldots, p_n\}$. Rearranging this we find:

$$p_k = (U - (C_{p_1}p_1 + \ldots + C_{p_k-1}p_{k-1} + C_{p_{k+1}}p_{k+1} + \ldots + C_{p_n}p_n))/C_{p_k}$$ (30)

In the previous encoding methods we stored all places in the BDD. But, from this equation, if we know $\{p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n\}$ we can compute $p_k$. Thus it is only actually necessary to store $n - 1$ of the places in that equation. In this way we reduce the number of places, and thus the number of Boolean variables we need to store in the BDD.

4.4.2 An Algorithm for the Computational Invariant Encoding

This section discusses our algorithm for determining the computational invariant encoding of GSPN states. The intention of this encoding method is to reduce the height of the BDD used for the GSPN state space exploration. This means reducing the number of Boolean variables per state, or bits, required to uniquely identify each state. In the previous section we showed how one place could be eliminated from the BDD by manipulating the invariant equation. In general there will be more than one invariant defined over the GSPN. We can use each of these equations to eliminate places from the BDD. Here we present an algorithm which, using some heuristics, will reduce the number of places we need to store in the BDD.

The algorithm we explain here is depicted in pseudocode in Figure 20. The algorithm splits the places of the particular GSPN into two distinct sets. The first set contains the stored places, which are those places which will eventually be represented by their characteristic functions and the resultant BDD. The second is a set of computed places whose markings we can compute from the invariant equations. Note that we can re-arrange the invariant equations such that any computed place relies on only one invariant equation.

We not only want to minimise the number of places that is stored, but also the number of bits used for the places that are actually encoded in the BDD. Thus step one of the algorithm is to sort the set of equations. We sort them in order of increasing right-hand-side (RHS). This way the equation with the smallest RHS will be used first and from Equation 25 we see that the smaller $U$, the smaller the place bound and the smaller the number of bits required to store that place. Note that this is only effective in cases where the concerned place appears in more than one equation. In this case we
store it using the lowest possible bound. In the binary encoding method we had to pick the highest bound to store the equation to ensure we had sufficient bits to uniquely identify each state.

To further enhance the space efficiency of the algorithm, we also sort each equation's left-hand-side terms in order of decreasing coefficient. From Equation 25 we see that a place \( p_k \) will have a smaller bound if its coefficient, \( C_{p_k} \) is larger. As we will see below the last place in the equation is usually the place picked to be encoded. By sorting the terms into decreasing order we ensure that the place computed by this algorithm would require the highest number of bits to store. Therefore, we create a larger reduction in the height of the BDD by not storing this place but computing it.

```
initialise set \( S \) of stored places = \( \phi \)
initialise set \( C \) of computed places = \( \phi \)
Sort invariant equations on RHS
for each equation do
    Sort equation in order of decreasing coefficient
    for each equation \( e \) in order do
        for each place \( p \) in order do
            if \( (p \notin S) \land (p \notin C) \land (\neg(p \text{ is the last place in } e)) \)
                add \( p \) to \( S \)
            endif
            if \( p \) is the last place in \( e \)
                insert \( p \) at end of \( C \)
            endif
        endfor
    endfor
endfor
```

Figure 20: Algorithm for the Computational Invariant Encoding Method

With the equations sorted we can apply the algorithm to split the places into the stored and computed sets. This is a greedy algorithm and may not result in the optimal encoding, but it improves on the technique of simply using P-Invariants to remove one place per equation, as was described in [8].

We iterate through the sorted list of equations, and for each equation we pick the first \( p - 1 \) places to store where \( p \) is the number of places in this equation that are not already in the stored or computed sets. The places in the stored set are in an order that is chosen to yield the optimal BDD size and is discussed later in this chapter.
CHAPTER 4. ENCODING FUNCTIONS

Each stored place is encoded exactly once from one invariant. The computed places are also determined by exactly one equation. Next we present an example for the GSPN in Figure 19 on page 42.

4.4.3 An Example of the Computational Invariant Encoding

Sorting equations 26 to 28 on their RHS and coefficient values we find:

\[
\begin{align*}
    p_3 + p_4 &= 3 \quad (31) \\
    p_0 + p_1 &= 7 \quad (32) \\
    7p_4 + p_1 + p_2 &= 7 \quad (33)
\end{align*}
\]

From Equation 31 we add \( p_3 \) to the stored set, and \( p_4 \) to the computed set. \( p_3 \) has an upper bound of 3 and thus is allocated 2 bits in the state descriptor bit-sequence. From Equation 32 we add \( p_0 \) to the stored set, and \( p_1 \) to the computed set. \( p_0 \) has an upper bound of 7 and thus is allocated 3 bits in the state descriptor bit-sequence. In Equation 33 we see that \( p_4 \) is already in the computed list, as is \( p_1 \). Therefore we can calculate \( p_2 \) given the value in places \( p_3 \) and \( p_0 \). Re-arranging Equation 33 we find that \( p_2 = 7p_3 + p_0 - 21 \), so \( p_0 \) is added to the computed set.

In this case, the stored set would contain \( \{p_3, p_0\} \) and the computed set would contain \( \{p_4, p_1, p_2\} \). The final height of the BDD would be 6 as we require 5 bits in total for places \( p_3 \) and \( p_0 \). The initial marking of the GSPN for this encoding is then \( 11_{(p_3)}111_{(p_0)} \). The remaining places can be computed from this using the invariant equations.

4.4.4 Performance of the Computational Invariant Encoding Method

We have previously mentioned that there are 9 reachable markings from the GSPN shown in Figure 19. We use 5 bits to encode this GSPN using the computational invariant encoding method. With 5 bits we uniquely identify 32 states. Thus the encoding performance for this method, given by Equation 24 on page 40, is 0.28. This is a vast improvement over the previous two encoding methods.

There is, however, added overhead in this method of order \( O(E \times P) \) where \( E \) is the number of invariant equations and \( P \) is the number of places in the GSPN. For the saving in time and space complexity obtained by this method this overhead is justified.


4.5 Partitions and BDD Ordering

We discussed the importance of variable ordering in Chapter 3. Variable ordering can have a drastic effect on the amount of memory occupied by the BDDs used for the exploration process and also on the time taken for a complete state space exploration. It is known to be an NP-complete problem to \textit{a priori} decide on the optimal ordering of variables [9]. However, using heuristics, we can attempt to find a more optimal variable ordering than simply a random ordering. The approaches we consider in this section are specifically tailored for the computational encoding technique, described in the previous section.

In all of the previously discussed techniques, the number of tokens on certain places of the GSPN are binary encoded. The individual encodings are concatenated to derive a state bit-vector describing the current state. The variables for the individual places are stored in the logical order of least significant bit first. We noticed no difference between least significant bit versus most significant bit first. Therefore, in order to improve the encoding we look to the ordering of the places themselves. In the course of this dissertation we have experimented with four different GSPN partitioning techniques.

The first is merely a control group experiment. This is the \textit{random} partitioning algorithm which is used to prove the usefulness of our other techniques. The next technique tried to extract structure from the underlying GSPN by building partitions on the set of places in the GSPN. Although this method has proved disappointing, seldom improving on the place order chosen purely randomly, the discussion is interesting.

Thirdly, we tried to cluster the places together into logical partitions depending on the transitions for which they are input places. That is, for each transition, we would construct a partition consisting of that transition and the set of places connected to it via incoming arcs. If any of those places are already elements of other partitions, then the partitions are combined via a set union. Once this operation is completed we are left with distinct sets of places which enabled certain transitions. The partitions are ordered by the decreasing number of places they contain and the place order is derived from this sorted order. We hoped that ordering in this way would group together places that affect the same transitions and are likely to have their values changed in a similar way, allowing identical paths to be shared in the BDD.

Breaking the net into partitions in this way turns out to be the close to the worst thing one can do. The aggregation of places enabling the same transitions does not lead to a suitable place ordering. One needs to \textit{interleave} similarly affected places in the GSPN. This is true for the interleaving of
row and column matrices in MTBDDs, and the same theory [24] applies here. This concept resulted in the best automatically found BDD variable ordering, as is verified by our results in Chapter 7.

To try to automate this process for GSPN with no obvious structure, we grouped together those places whose number of incoming and outgoing arcs are equal. This would mean that they are perhaps in similar subnets of the GSPN. The technique we used for this simply arranged the places firstly in the order of increasing numbers of incoming arcs. The places with equal numbers of arcs were then sorted on their number of outgoing arcs. The results of this simple technique were better than the previously described method and this can be improved even further. We refer the reader to Section 7.2.3 on page 98 for a full comparison of the different variable orderings.

Choosing the correct ordering is an essential part of the state space solution process when using implicit techniques. In our experimentation, we have observed state space searches completing in less than half the time when a good place ordering is chosen. In our tool-set we allow the user, who has some knowledge of the modeled system to choose the place ordering they feel is best. In every experiment we conducted, exploiting the modeler’s knowledge of the GSPN always to determine the ordering always resulted in the best state space search times and most conservative memory usage.

4.6 Conclusion

We looked at three different encoding methods namely the one-hot encoding, the binary encoding, and the computational invariant encoding method. The one-hot encoding, while simple to implement, was shown to be inefficient in terms of space used. The binary encoding method offered a reasonable improvement in space efficiency with a slight increase in difficulty. The computational invariant encoding method, which was developed during this dissertation, gives a good improvement in space efficiency, but does require pre-processing to encode the state spaces.

Our invariant-based encoding was originally inspired by the work of Pastor et al [18], who take a similar approach, using BDDs to compute the encoding. After we had derived our own encoding, we discovered the work of Haerkort [8] who presents a similar concept, also using invariant equations to reduce the size of the stored bit-vector. However, no formal algorithm is presented. We developed our invariant-based encoding function independently and show, in Chapter 7, how our method of using heuristics to choose places to encode in the BDD that require the smallest number of bits greatly improves over a binary encoding.
Chapter 5

State Space Exploration Methods

Modern computer systems are complex both in the number of possible states in which they can exist, and in the time taken to find all these states. A popular way of analysing these complex systems is to abstract them into some form of simplified model, which may then be analysed.

Even simplified models, however, suffer from the state space explosion problem, where the number of reachable states from a system’s initial state is exponential in the number of parallel components of the system [24]. State spaces of even apparently simple models may be as large as $10^{10}$ and thus the descriptions of all of these states cannot be stored using conventional memory techniques. Furthermore, finding these states is often not possible in a realistic amount of time. In order to explore and analyse increasingly complex models we require advanced techniques for searching and storing state spaces.

The focus of this dissertation is symbolic methods for state space generation. Symbolic, or implicit, methods are very popular in the area of sequential circuit verification, VLSI, and model checking, and have been used with great success [31]. Symbolic methods have lately been referred to as “implicit” techniques, as an alternative to “explicit” techniques, where every state is explicitly represented in some data structure, as with search trees or hash-tables.

The central idea, which we will expand upon in later sections, is to represent the set of states explored by means of a characteristic function (cf. Ch. 4). Symbolic techniques can be thought of as tools for lossless data compression. They allow exhaustive state space analysis in a reasonable amount of time. In the area of VLSI they have extended the boundary on the size of the models one is able to explore and analyse.
We have investigated two methods for state space exploration using BDDs as the underlying storage data structure. The first simply uses BDDs as a structure for lossless data storage and is discussed in Section 5.2. The second, described in Section 5.3, uses BDDs as a storage structure, and also describes the way state changes in the model by means of transition relations.

In this chapter we give an overview of techniques for state space exploration, leading up to the techniques we have investigated. In Chapter 6 we discuss techniques for solving the Markov chain resulting from the generated state space.

5.1 State Space Exploration Techniques

In this section we discuss the various techniques for carrying out state space exploration.

5.1.1 Exhaustive and Probabilistic Techniques

Two approaches to state space exploration are exhaustive and probabilistic methods. Exhaustive methods find every possible state in the state graph. Probabilistic techniques perform an inexact state space search. That is to say that the entire state space is found within a certain confidence level.

Probabilistic techniques were introduced as a means to reduce state space exploration memory requirements. By reducing the number of states stored, and the size of each of the stored state descriptors, one can decrease the amount of memory required for the exploration process. Probabilistic techniques are particularly prominent in state space exploration algorithms involving hash tables as the explored state storage mechanism.

Collisions can occur when hash-tables are used as the state storage and lookup structure. Probabilistic techniques essentially turn collision detection off. That is, if a newly found state hashes onto a key that has already been used, the new state is simply discarded.

The success of a probabilistic technique depends heavily on the even distribution of the state descriptor hash function. A popular technique is to use one or more secondary hash functions. If the first hash function renders a collision, then the second function is applied, and so on.

Probabilistic techniques have some limitations. There are pathological cases where only one state is the “gateway” to large sub-graphs of the state graph, or cases where the one unexplored state
turns out to be a deadlocked state. Probabilistic methods do not provide an exact solution to the model in question, rather they provide, with a certain confidence level, information about the system’s correctness, or information about the system’s performance to within some range. With any probabilistic search the number of states found is always reported with its associated accuracy.

Probabilistic methods, although not providing an exact solution, are usually reasonably accurate [33] and conservative of memory. However, as explored state sizes grow, probabilistic techniques become less feasible. The state-space explosion problem still applies to probabilistic techniques, which rely on a fixed amount of memory for each found state.

5.1.2 Parallel and Distributed Techniques

A simple way to decrease both the memory and execution time implication of the state space explosion problem on the size of models we are able to analyse is to add more processors and distribute the state space exploration amongst them. Techniques we have considered thus far are known as sequential techniques where the state space is found incrementally by a single machine in either a breadth-first or depth-first fashion.

Parallel and distributed techniques assign a processor some portion of the state space which it is that processor’s responsibility to explore. Regardless of whether one uses a parallel cluster or distributed network the goal is the same: to minimise communication between nodes. Each node’s assigned portion of the state space is determined by a partitioning function. This is a hash function from a state $s \in S$ onto $\{0, ..., n - 1\}$, where $n$ is the number of processors.

Partition functions must be chosen in such a way to optimise the following three parameters:

1. **Spatial load.** This means that each processor should be assigned an equal portion of the state space, or that the partition function should distribute uniformly over the available processors.

2. **Temporal load.** Slightly more subtle, it is possible that the partitions created by the partition function are separated linearly in time, which means that all states which are the responsibility of processor 1 are found before any which are required by processor 2, and so on.

3. **Communication load.** Finally, the partitions should be loosely connected to one another, which means that processors should spend more time exploring the state space allocated to them than dispatching states to other processors.
Figure 21 shows four figures of possible state spaces. The graph in part (a) shows an ideal state space where each of the partitions, indicated by a different shade of grey and a number, is proportionately sized, there is little communication between partitions, and the partitions are distributed evenly throughout the state graph. (b), of Figure 21, on the other hand shows the pathological case where processor 1 is assigned nearly all the states to explore. (c) shows a sample state graph where there is a high degree of interconnectivity between the partitions of the state space. Finally, (d) shows the extreme case for a poorly temporally balanced partition function where partition 1 is explored, followed by 2, and so on. This exploration algorithm would be equivalent in time elapsed to a purely sequential algorithm. Partition functions need not be static. The partitions can be dynamically remapped if it is found that the function is performing poorly in any of the three categories mentioned above.

We have mentioned parallel and distributed techniques briefly here as we have experimented with
the application of these techniques to our own symbolic methods. We used the transitions of the GSPN as the partition. Each processor was assigned a certain number of transitions and was responsible for finding states reachable by firing those transitions. This approach, also used by Caselli et al in [54], is disappointing unless knowledge of the model is used to choose how the transitions should be divided amongst the processors. Caselli et al reported that execution times increased linearly with the size of the state space. There is, however, much scope for future work in this area.

5.1.3 Structural and Decomposition Techniques

Much success [6] has been gained recently from exploiting the regularity of structure from models. By finding repetitive sub-structures in a model one can speed up the exploration process, and minimise memory requirements.

Implicit techniques are known to be particularly efficacious when applied to models with a large degree of regularity in the structure. The implicit state storage mechanism of a technique such as BDDs means that several states can be stored along the same path in the BDD.

The approach is essentially “divide-and-conquer” where the net is decomposed into sub-structures which can then be independently explored and then later recombined. The process of decomposing the model can be automated using heuristics. The use of structural techniques has lead to some amazing state space search sizes. For example, the work of Ciardo [6] has been able to explore up to $10^{626}$ states in under a second. This was for an extremely regular net, however, and the results quickly taper off for less structured models.

5.1.4 Depth-First Exploration

Classical methods of exhaustive state space exploration techniques [33] use a depth first approach in deriving the state space. The algorithm for the depth-first exploration is shown in Figure 22.

The exploration starts by pushing the initial state onto a stack of unexplored states, $U$ (line 4). The iterative part of the algorithm then proceeds by popping the top state off the stack. This state is then used to find all the enabled transitions, which are subsequently fired to find the popped state’s immediate successor states (line 7). Each new state is checked against the set of states in the explored state table to see if it has been previously visited (line 8). States encountered for the first time are pushed onto the stack in the order in which they are found (line 9) and, finally, the state that
was originally popped from the stack at the start of the loop is inserted into the table of explored states (line 10).

In this algorithm the size of the stack of unexplored states is in proportion to the depth of the state graph and is thus very small. The chief concern in this algorithm is the size of the explored state table, and the efficiency of testing whether or not a state has been visited, or explored, previously.

5.1.5 Breadth-First Exploration

We discuss the breadth-first technique here as it is the generally accepted means of state space exploration for use with implicit data structures, such as BDDs. As with the the depth-first technique, there are three state holding structures: the new states $N$; the explored states $E$; and the unexplored states $U$. We will, for the purposes of this discussion, ignore the underlying types of the data structures used to hold these states, and refer to them simply as sets. The algorithm for breadth-first exploration is shown in Figure 23.

The set $N$ is initialised to contain only the initial state. The iterative part of this algorithm proceeds until there are no states left in the new state set. This new state set is often referred to as the frontier set. Figure 24 shows the algorithm graphically.
5.

<table>
<thead>
<tr>
<th>var</th>
<th>set $E$ of explored states</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>set $U$ of unexplored states</td>
</tr>
<tr>
<td></td>
<td>set $N$ of new states</td>
</tr>
<tr>
<td>begin</td>
<td>insert the initial state into $N$</td>
</tr>
<tr>
<td>while</td>
<td>$N \neq \emptyset$ do</td>
</tr>
<tr>
<td></td>
<td>let $U = \text{findSuccessorStatesFrom}(N)$</td>
</tr>
<tr>
<td></td>
<td>$E = E \cup N$ {add the newly explored states to the explored BDD}</td>
</tr>
<tr>
<td></td>
<td>$N = U - E$ {find the frontier set}</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Figure 23: Breadth-First State Space Exploration Algorithm.

Line 6 of the algorithm examines every state in the frontier set. For each state the enabled transition must be found and, if a transition is enabled, the one-step successor state recorded. All of the one-step successor states are then returned in a single set called the unexplored set.

On line 7 the frontier set is incorporated into the explored state set forming the new explored state set. Finally, on line 8 the new frontier set is calculated as all the elements in the unexplored state set that are not in the explored state set. Clearly, once the frontier set is empty, the state space exploration is finished.

Figure 24: Graphical representation of the Breadth-First Algorithm

This is a generic breadth-first algorithm; the function call to \text{findSuccessorStatesFrom}(N) can be modified, changing the way in which the new states are found. This function takes the frontier set
as input and returns the next state set. Exactly how we determine the successor states is critical for an efficient algorithm. The mechanism of this algorithm is discussed in great detail in Sections 5.2 and 5.3.

5.1.6 Recording Transition Rates

Performance analysis requires that the underlying infinitesimal generator matrix, which describes the CTMC for the model's behaviour, be generated during state space exploration. This square matrix has a positive entry of \( r \) at \( a_{ij} \) indicating a state-transition from state \( i \) to state \( j \). During the state space exploration process the values of \( r \) for each step need to be recorded for later use in the CTMC solution process. The theory of CTMCs can be found in [36], and two CTMC solution techniques we have researched are presented in Chapter 6.

Our state space exploration technique uses BDDs, primarily for the excellent “data compression” they offer. For this reason, we have also chosen to use MTBDDs (cf. Ch. 3) as the storage mechanism for the rate matrix, which stores the firing rates of each transition. We discuss how we have implemented the MTBDD matrix storage for each of our two techniques in the relevant sections below (Sections 5.2 and 5.3).

GSPNs have two differing types of transitions: immediate and timed transitions, where the immediate transitions have priority over the timed transitions. Immediate transitions are a useful modeling tool and can drastically cut down the size of the resultant state space. Most importantly, they increase the power of Petri nets to be equivalent to Turing machines [29]. For this reason we feel it is essential to support them in our experimental software tool-set.

A GSPN marking, or state, is vanishing if it enables any immediate transitions. Since immediate transitions have priority over timed transitions, the timed transitions are effectively disabled. Further, a zero state sojourn time is associated with immediate transitions, which means that there is a zero probability of being in a vanishing state at any point in time. Vanishing states must therefore be eliminated from the computation of the CTMC solution. They can either be eliminated as a step in the Markov solution process, or they can be eliminated on-the-fly where they are dealt with as they are encountered in the state space exploration process.

We choose to perform vanishing state elimination on-the-fly because of the memory saving obtained. From our experience there is usually a much greater number of total states than tangible
states derived from a GSPN model. Indeed, one of the major benefits of modeling with immediate transitions is the reduced size of the resulting state space which must be solved.

We solve the Markov chain derived from the state graph using MTBDDs (c.f Ch. 6). The memory saving of on-the-fly vanishing state elimination is not as pronounced because of the good memory saving offered by MTBDDs. A final reason to perform on-the-fly vanishing state elimination follows.

At the end of the exploration process, we will have an MTBDD for the sparse matrix of state-to-state rates. This matrix, $R$, is of the form:

$$R = \begin{pmatrix} R_{t,t} & R_{t,u} \\ P_{u,t} & P_{u,u} \end{pmatrix}$$

(34)

where $R_{t,t}$ is the rate matrix for transitions firing from tangible to tangible states via timed transitions and $R_{t,u}$ is the matrix of transitions firing from tangible to vanishing states, also via timed transitions. Similarly, the matrices $P_{u,t}$ and $P_{u,u}$ describe the transition weights for each of the immediate transitions that could affect moves from vanishing to tangible, or other vanishing states.

The $Q'$ matrix of Equation 34, which describes the transition rates from tangible to tangible states, once the vanishing states have been accounted for, can be extracted using the following formula:

$$Q' = R_{t,t} + R_{t,u}NP_{u,t}$$

(35)

where the matrix $N$ is defined by:

$$N = (I - P_{u,u})^{-1}$$

(36)

Solving for $N$ requires an inversion of the matrix $(I - P_{u,u})$. Matrix inversion, except in some specialised cases, is inefficient on MTBDD representative data structures [20]. This can be attributed to the fact that inversions typically involve a large amount of fill-in, which reduces the sparseness of the matrix, and thus causes a blow-up in memory resource needs. On-the-fly vanishing state elimination is therefore a desirable solution to removing the vanishing states from the Markov solution process.
Figure 25: A reachability graph demonstrating the effects of on-the-fly vanishing state elimination.

Consider the state graph on the left in Figure 25 where there are 8 states, 3 of which are vanishing, indicated by grey circles. After the vanishing states have been removed, as one can see in the right-hand graph of Figure 25, there are now only tangible states remaining and the relative rates between them have been calculated.

Sections 5.2.2 and 5.4.3 provide the details of how we have implemented vanishing state elimination for our two state space exploration techniques.

5.2 BDD Data Storage

Hashing methods work to reduce the amount of memory required to store the explored state space in two ways:

- by reducing the size of the state descriptors that must be stored in volatile memory for quick access, and by

- reducing the actual number of states that are stored, using probabilistic state descriptor hashing techniques.

The chief drawback of using hash-tables in this way is that they are probabilistic. The solution will not be absolute, in terms of correctness or performance of the system. We propose the use of BDDs as an efficient data compression technique for the storage of states during state space exploration of GSPNs.
In this chapter we comment upon our experiences in developing an efficient technique, building on the success of finding a suitable encoding for the GSPN in question. While Chapter 4 heavily relied upon the exploitable structure of GSPNs we stress that the techniques applied in this section are general and could be applied to any model of concurrent communicating systems where the modeling formalism is based on states and the transitions between them.

5.2.1 Breadth-First and Depth-First Searches

We have mentioned the two possibilities for algorithms for state space exploration, depth- and breadth-first searches. For the technique of data storage we choose breadth-first searches because they allow the exploration of all the successor states from a set of states in one logical iteration of the algorithm. Adding one state at a time to the BDD of the explored states is highly inefficient as the ite algorithm must be called once for each newly found state. Since each new state found will be a BDD with exactly one path and no nodes reduced, the total complexity for this exploration mechanism is \( O(D \cdot |E| \cdot |N|) \), where \( D \) is the depth of the BDD required by the encoding function, \( |E| \) is the size of the explored state BDD, and \( |N| \) is the total number of explored states. Furthermore, for every state found, an \( O(D) \) lookup must be done on the explored state BDD to see if this state has been previously inserted. This lookup would be \( O(D) \) in the worst case because there may be nodes along the path that represent don't care variables.

5.2.1.1 Incremental Reduction

Similarly to hashing methods, when BDDs are used simply as a storage structure we can also concentrate on making the encoding mechanism as efficient as possible. Although we did not implement this, it is even possible to make use of probabilistic state space exploration techniques. We have described the characteristic function which allows us to store sets of states in BDDs where each path from the root to the true terminal of the BDD represents a state (cf. Ch. 4).

The function \( \text{findSuccessorStatesFrom}(N) \), in the breadth-first search algorithm in Section 5.1.5 on page 56, must return all the states reachable in one step from the states in \( N \). We can find all the states stored in \( N \) by a simple BDD traversal. The algorithm for this process is shown in Figure 26.

During this traversal, we maintain an array of values indicating whether we took the low child path or the high child path, or whether we skipped a node, or number of nodes. If we do skip a number of nodes, then for each index, we need to set the corresponding path array value to a don't care
value, which will be used later in the call to `findSuccessorStatesFrom`. Whenever we reach the true terminal, we use the path array to set the marking of the Petri net, which we store as a dynamic structure in memory. The Petri net object is an entirely separate entity from the BDD used to store the states. The object diagram for our Petri net structure in memory is shown in Figure 27. Each place in the Petri net, as was discussed in Chapter 4, is represented by some sequence of bits, or levels, in the BDD. We have setup a BDD-index to place bit-index mapping structure which allows us, given a sequence of bits, to find which places are affected and how many tokens should be on those places.

For each set marking, we find and fire the enabled transitions. This process, encapsulated by the function `findSuccessorStatesFrom`, changes the marking of the Petri net, based on the transition firing. This new marking can then be inserted into the BDD denoted $U$ (see Fig. 26) representing the set of states found at this level of the state graph. These are the unexplored states, which will contribute to the frontier set in the next iteration of the state search algorithm.

The most time-consuming part of this algorithm is the collection of the successor states by the function `findSuccessorStatesFrom`. Initially, the BDD for the successor states, $U$, is empty and grows upon the insertion of states to some maximum size before shrinking with every new insertion,

```
procedure findSuccessorStatesFrom(BDD : U, Vertex : v, array of integers : path)
begin
  if (value(v) == 0) return
  elseif (value(v) == 1)
    foreach (combination of path) do
      findAllSuccessorsOf(path, U)
    end
  else
    begin
      set path[index(v)] to false
      fillInDon'tCares(index(v), index(low(v)), path)
      findSuccessorStatesFrom(U, low(v), path)
      set path[index(v)] to true
      fillInDon'tCares(index(v), index(high(v)), path)
      findSuccessorStatesFrom(U, high(v), path)
    end
end
```

Figure 26: `findSuccessorStatesFrom` detailed from the algorithm in Figure 23.
in accordance with the graph in Figure 3. We could make use of the ite algorithm in Figure 5, on page 23, to insert each newly found successor state into the set of new successor states. We would call the function ite with parameters $C$, $1$, and $S$ which is equivalent to performing the logical or ($\lor$) operation where $C$ is the collection of states found thus far at this level of the state graph and $S$ is the newly found state. Since this algorithm would be performed once for each newly found state the overall complexity of performing this algorithm would be $O(|C| \cdot |S| \cdot n)$, where $n$ is the number of states to be inserted at this level. This is, in practice, very time consuming and impractical.

A much more efficient means to add the new states is to realise that a new state simply means a new path in the BDD. This new path would simply be a worst-case $O(d)$ insertion. Unfortunately, since the BDDs are reduced we cannot simply add a new path to the existing BDD because this would create inconsistencies in the state found [9]. Reduced BDDs are a canonical representation of a Boolean function and therefore arbitrarily adding a new path to the BDD would invalidate this property.

We present a new algorithm to work around this problem. It is impossible to insert a new state into a reduced BDD by simply creating a new path. However, if the BDD is unreduced, we can simply add the nodes to build the new path from root to true terminal. This reduces the time complexity of adding a new state to $O(d)$, where $d$ is the height of the BDD.
CHAPTER 5. STATE SPACE EXPLORATION METHODS

```
procedure findSuccessorStatesFrom(BDD : U, BDD : T, Vertex : v, array of integers : path)
begin
  1 if (value(v) == 0) return
  2 elseif (value(v) == 1)
    begin
      3 foreach (combination of path) do
      4     findAllSuccessorsOff(path, T)
      5 if (|T| > maxSize)
      6     U = U ∪ T
      7     T = false
      end
        8 else
          begin
          9     set path[index(v)] to false
          10    fillInDonCares(index(v), index(low(v)), path)
          11    findSuccessorStatesFrom(U, T, low(v), path)
          12    set path[index(v)] to true
          13    fillInDonCares(index(v), index(high(v)), path)
          14    findSuccessorStatesFrom(U, T, high(v), path)
          end
    end.
end.
```

Figure 28: The algorithm for Incremental Reduction during state space exploration.

Unfortunately, unreduced BDDs lose their compactness — the characteristic which made them a good structure for data compression. While parent nodes are still re-used for paths in the unreduced version, isomorphic child sub-graphs are not re-used, thus leading to a blow-up in memory usage. The BDDs become large and unmanageable very quickly. Therefore, we cannot leave them in their unreduced form indefinitely.

Our technique makes use of two BDDs for the successor state computation. The first is an ordinary reduced BDD, U, which simply stores the states found thus far. The second is an unreduced temporary BDD denoted T. The temporary BDD has states added to it by simply inserting the new paths to the true terminal. This is done until the temporary BDD reaches a user-defined upper-bound in terms of the number of nodes used.

Once this upper-bound is reached, the ite algorithm is used to union T, the unreduced graph, with U, the reduced graph. Since the ite algorithm always returns a reduced BDD, this process has the effect of reducing the size of the BDD needed to store the newly explored states. The algorithm we developed is a modification of the algorithm in Figure 26 on page 62, but we must now also propagate the temporary BDD T throughout the recursive function and check whenever we have
added new states to $T$, that the BDD for $T$ does not exceed the user-defined size. This new version is shown in Figure 28. On line 7, the act of setting $T$ to \texttt{false} creates a new empty BDD.

For completeness, we also show the algorithm for \texttt{findAllSuccessorsOf(path, T)} in Figure 34. This algorithm is adapted for the case of using incremental reduction, where new paths are simply appended to the existing ones where possible. \texttt{findAllSuccessorsOf} makes a call to the procedure \texttt{addNewPath(newPath, T)}, for every transition that could have been fired from the set marking.

Since the reduced BDD is normally several orders of magnitude smaller than its unreduced counterpart, this algorithm allows the user to effectively place an upper-bound on the amount of memory used for the exploration. So, based on one’s system capabilities, one may select the memory size to be used by the exploration, and be guaranteed, up to a point, that the exploration will not crash due to memory exhaustion, or begin to degrade heavily in execution time when virtual memory is needed. Obviously, the lower one sets the upper-bound the closer the complexity of this algorithm becomes to inserting the states one at a time and so an optimal bound should be chosen for the user’s specific hardware capabilities.

Results of testing with these techniques can be found in Chapter 7.

### 5.2.1.2 Frontier Set Calculation

Hash functions take the current state and hash it onto a certain integer value. This integer value is then used as a lookup in the table of explored states to see if this state has been found before. This process involves a bit-wise comparison of the state hash values. BDDs can be used in a similar fashion. To detect whether a certain state has previously been found, one performs a traversal of the BDD for the explored states. The decisions of which children to follow in this traversal depends on the value of the state to be inserted.

The previous exploration algorithm we described found all the successors of a particular state. Some of these successors would have been previously found. In order to ensure that we do not re-explore them, they are removed at the next step in the algorithm. This is the set minus operation performed on line 8 of the algorithm in Figure 23 on page 57.

We have found this to be inefficient when used with our exploration method. This is because every new found state must be inserted by adding a new path to the BDD for temporary storage. This involves adding several new nodes to the BDD which is slow even though we have implemented
our own heap management. We have found it more time efficient and memory conservative to, for each found state, first check whether it is in the BDD for the explored states.

This involves a worst-case $O(D)$ comparison. The algorithm begins with the root of the explored states' BDD and uses a bit-array to determine the path of the traversal. If the value of the bit-array at the corresponding index to the level of the current non-terminal vertex in the BDD is true (false), then the high (low) child path is recursed further.

If the bit-array based recursive BDD traversal encounters the false terminal then that path, representative of the proposed state to be inserted, does not exist in the BDD of the explored states and must therefore be added to the BDD of the unexplored states. If the recursion encounters the true terminal then the state is present, and should not be inserted.

This technique is more efficient than the bit-wise comparison used in hashing techniques, because certain paths of the BDD contain don't care variables. These are variables, or levels in the BDD, which have been reduced out earlier in the exploration. These levels would obviously not be compared. Another speed-enhancement is that, since we never add a state to the temporary BDD which has already been explored, we know that the BDD for the unexplored states, which is built up at every iteration of the breadth-first search, will always be exactly the states in the frontier set. For this reason, we do not have to perform the set minus operation of line 8 of the algorithm in Figure 23.

As is shown in Chapter 7, this process of maintaining the frontier set leads to a moderate improvement in time efficiency.

5.2.1.3 Detecting Enabled Transitions

We now describe one further efficiency enhancement that has yielded moderate time improvements. On line 1 of the findAllSuccessorsOf algorithm in Figure 34, we use the bit values in the path array to set the marking of the Petri net stored as an object in memory.

For the case of the computational encoding (cf. Sec. 4.4), the computed places’ values can only be determined after all the stored places they rely on have been set. As we traverse down a particular branch of the BDD we may set the values of the bits corresponding to certain sorted places that, in turn, can be used to determine the values on computed places. Essentially, it is not necessary to wait until the entire bit-vector has been set before computing the marking on the Petri net.

In fact, waiting until the entire vector is known is very time wasteful. In the normal course of traversing the BDD bit values will be set for the upper indices and not changed until much later
when the algorithm proceeds down another child's path. This means that certain places will be
determined to have the same value again and again at lower levels of recursion.

If we can determine the number of tokens on a stored place as soon as the bit values are available
we can avoid this re-computation. Further, in the case of the computational encoding, since some
computed places can be determined based only on a small set of the stored places, these computed
places' values can also be found at a higher level of the recursion.

Transitions in the Petri net can consume tokens from either stored or computed places. Whether
they are enabled or disabled relies on values from both of these place sets. In exactly the same
way that we can determine computed places' values early in the recursion, we can also determine
whether certain transitions are enabled or not. This means that once the recursion reaches its base
case we do not have to check every transition to see if it is enabled. Some transitions' firing ability
will already be known.

This saves calling the isEnabled function of every transition for every path that is chosen through
the BDD, and since the isEnabled function requires the iteration through each of the transitions'incoming arcs to test if the attached places contain sufficient tokens, this leads to a moderate time
saving.

In order for this technique to be implemented, once the order of places stored in the BDD is known,
a "map" of BDD vertex indices can be built from the index onto the set of transitions. In fact this
map contains all information about which places' values are known at the various levels of the BDD.
In order to build up the transition map, we iterate through all the places. For each place we find the
transitions for which this place is a producer. For each such transition, we then iterate through the
remaining list of places, and see if this particular transition ever appears again as a consumer of any
of the remaining places. If we never find this transition again, then we know that this transition can
be determined to be enabled or not, purely depending on the values of stored places whose values
will have already been set in the progression of the recursive algorithm.

5.2.2 Vanishing State Elimination

Vanishing state elimination is implemented in the following manner. For every state that is en-
countered, it is checked to see if it enables any immediate transitions, if so the descendants of this
immediate state are explored in a depth-first fashion. We make use of a temporary "vanishing state"
stack for tracking the states we have not yet explored. The elements on this stack, of the form \( \{S_v, r\} \), aggregate the vanishing state \( S_v \) and the rate \( r \) at which that state is reached.

Any vanishing state, found in the usual exploration process, is pushed onto the vanishing state stack along with the rate of the transition that led to the state. Then the depth-first search continues until there are no more elements on the stack. At each iteration, the stack is popped and the one-step successors are found from the popped vanishing state. The tangible successor states are added to the set of explored states, and the rate is recorded as will be discussed in Chapter 6. The rate is calculated by multiplying the rate found in the popped element with the weight of the transition that fired to reach this tangible state. Any further vanishing states are pushed onto the vanishing state stack with rate equal to the popped element's rate multiplied by the weight of the fired transition.

Vanishing states can be handled by extending the functionality of the procedure `findAllSuccessorsOf` in Figure 34. The extended version of this for coping with the vanishing states is shown in Figure 35.

Vanishing states are never stored permanently. This decreases our storage space requirements, but also means it is possible to explore the same subspace derived from a vanishing state more than once, which is undesirable. Furthermore, it is possible for there to be a cycle of vanishing states which would not be detected and the search algorithm would loop forever. Consider the state graph in Figure 29, where vanishing states B, C, and D form a cycle. When we find B from A, the current algorithm would continuously loop through states B, C, and D.

![Figure 29: Vanishing states forming a timeless trap.](image)

Fortunately, such a collection of vanishing states constitutes a *timeless trap*. A timeless trap exists in the system where, beyond a certain point in the state graph, only immediate transitions can be enabled. This situation of transition activity with zero time elapsing is clearly indicative of a flawed or functionally incorrect system and, since the goal of the exploration is to carry out
performance analysis, we should report this functional error to the model designer. However, we do need to ensure that the state space exploration technique can detect such errors. We consider three mechanisms for dealing with timeless traps.

The first uses the weights of the immediate transitions. When a new vanishing state is pushed onto the stack, the pushed rate is calculated by multiplication of the weight of the fired transition with the rate most recently popped from the stack. A simple technique for detecting timeless traps would be to stop the depth-first search when the rate to be pushed drops below a certain minimum.

However, it is possible that all the transition weights are equal to 1, in which case the rate would always be the same and the above mechanism would fail. Another technique is to count the number of times the depth-first search has looped, and stop if this exceeds a certain maximum. Of course, it is possible that we do not set the maximum large enough, or that we set it too low, in which case certain tangible states would not be found.

A fool-proof method for handling the vanishing states would be to store them in a vanishing state table, but this then increases our memory requirements. A possible alternative would be to have a small cache of explored vanishing states, implemented as a hash-table. Each element of the cache contains the vanishing state and the set of tangible states which are its successors, and the associated rates. In combination with the depth-first iteration counter, one could not only decrease the possibility of missing certain states, but also save the re-exploration of the sub-graphs rooted by vanishing states.

5.3 Transition Relations

Thus far we have used BDDs to efficiently store the set of states found in the state space exploration process. The BDDs currently serve only as a storage data structure. In this section we discuss a technique whereby not only the state of the system can be stored with BDDs, but also the dynamic behaviour of the system as time progresses and transitions fire. This technique makes use of transition relations.

An encoding function for the state of a Petri net basically converts the state descriptor into some bit-vector or characteristic Boolean function. Similarly, with transition relations, we now find a Boolean function which, given the current values of the state variables, or Petri net places, determines the values of those variables, in the next state. It is a Boolean function that is built to describe how each bit of the state descriptor bit-vector can change upon the firing of transitions.
Transition relations are used in that part of the breadth-first state space exploration algorithm (Sec.5.1.5) which finds the successor states. The function that finds the successor states of a set of states is modified to use transition relations. The other functions to add the found states to the set of explored states and to compute the frontier set remain the same.

The use of transition relations allows us to avoid unreduced BDDs and further conserve memory during state space exploration. They have been successfully used by Clarke [20] et al., in the model checking of VLSI circuits, exploring more than $10^{20}$ states in reasonable time and with good memory utilisation. We discuss the theory of transition relations before [20] describing how we have adapted them to our specific needs in analysing GSPN models.

5.3.1 Basic Theory

A predicate transformation is an operation that, given an input set, returns the output set of all possible functions of the values in the input set. In terms of state space exploration a predicate transformation takes the set of currently unexplored states and returns the set of all possible successor states. A predicate transformation can be implemented as a disjunction or conjunction of individual transition relations.

We refer to a transition relation as a relation between the set of current unexplored states and the successor states reachable in one step given the firing of a particular transition $t$. If $N_t$ is a transition relation then we note $N_t(V, V')$ which describes the mapping of $V$, the present state variables, onto the $V'$ next state variables given that transition $t$ occurs.

$N_t$ is a conjunction of the individual relations $N_t(V, V')$ describing the behaviour of each state variable for a particular transition. These individual relations are defined by:

$$N_t(V, V') = (v_i' \leftrightarrow f_t(V))$$

(37)

where $f_t(V)$ is a function which operates on the present state variables and returns the value of the state variable $i$ ($v_i'$) for the firing of transition $t$. The transition relation $N_t$ is the conjunction of these individual terms, described as follows:

$$N_t(V, V') = N_{t_0}(V, V') \land ... \land N_{t_{n-1}}(V, V')$$

(38)
CHAPTER 5. STATE SPACE EXPLORATION METHODS

The reason for the conjunction is clear. Upon the firing of any particular transition, all the state variables (bits) affected by that transition will change according to the dynamic behaviour of the model. Thus $N_t$ is a conjunctive transition relation. Forming the total transition relation for the Petri net, or any formal system where there are states and transitions present, requires a disjunction of each of the individual transition relations. This is because, at any state, one or more transitions may fire. Thus the total transition relation for an ordinary Petri net is defined by the equation:

$$N(V, V') = N_{t_1}(V, V') \lor \ldots \lor N_{t_n}(V, V')$$

(39)

5.3.2 Example

A transition relation, when expressed as a BDD, consists simply of a list of paths expressed over a double copy of the variables used to describe the state of the system. For a particular transition, the relation would involve only those variables that are affected by the transition. The list of paths would include every possible state change that could be affected by that transition over the variables it manipulates.

![Diagram](image)

Figure 30: (a) A Simple Petri net, and (b) the transition relation BDD for $t_0$.

For example, consider the simple Petri net on the left in Figure 30. The transition relation for transition $t_0$ is shown on the right of Figure 30. Upon traversing this BDD, we find the possible paths we could follow are: 0010, and 1001.

It has been suggested that a good ordering of variables in a transition relation is to interleave the current and next state variables [20]. This is intuitive because, often, the current and next state
variables will be unchanged, which means that we can take advantage of the properties of reduced BDDs.

When we examine the list of paths above, we find the following state transitions are possible: 01 → 00, and 10 → 01. Converting from binary to integer representation, transition $t_0$ can potentially remove either 1 or 2 tokens from its input place $p_0$. Finally, note that the BDD variables are ordered from most significant to least significant bit.

### 5.3.3 The Relational Product

Given an input set of states the predicate transformation returns every possible state that can be reached in one step by firing any enabled transition. Each predicate transformation in our state space exploration takes the states $S_k$ at the current level $k$ of the state graph and returns the states at level $k + 1$, $S_{k+1}$. This can be formalised by:

$$S_{k+1}(V') = S_0(V') \vee \exists v \in V[S_k(V) \land N(V, V')]$$

The least fixed point of this equation is when $S_{k+1} = S_k$. This implies that, from the currently explored set of states it is impossible to find a state which is not already in that set. The most complex and time consuming part of Equation 40 is $\exists v \in V[S_k(V) \land N(V, V')]$, the determination of the new states set.

Clarke et al [31] found an algorithm for computing this result in one pass of the BDDs. This is known as the relational product algorithm and the benefit of their algorithm is that one never computes the BDD for $S_k(V) \land N(V, V')$, which can typically be quite large. We present their algorithm here for completeness.

The relationalProduct algorithm in Figure 31 takes the BDD for the transition relation and the BDD for the current set of unexplored states and computes the next state set in the form of a BDD. The function also takes a set of variables, usually implemented as a list of integers referring to the levels of the BDD that symbolically represent the current state variables. Similar to other BDD algorithms, a table of the unique vertices is maintained to ensure the returned BDD is reduced. The relationalProduct algorithm is $O(n^2)$ in the worst case [31].

Unfortunately, while this algorithm returns the set of successor states in one pass of the BDDs for the transition relation and the current states, it cannot report the actual state-to-state transitions; that
```plaintext
function relationalProduct(N(V, V') : Vertex, S(V) : Vertex, vars : Set) : Vertex
    {returned Vertex is the root of the "next-state" BDD}
begin
1     if ((N(V, V') =false) ∨ (S(V) =false)) return false
2    elseif ((N(V, V') =true) ∧ (S(V) =true)) return true
3    elseif ({(N(V, V'), S(V)} ∈ resultCache)) return previously computed vertices
4    else
5        let z be the topmost variable of S(V) and N(V, V')
6        h0 = relationalProduct(low(N(V, V')), low(S(V)), vars)
7        h1 = relationalProduct(high(N(V, V')), high(S(V)), vars)
8        if (z ∈ vars)
9            h = h0 ∨ h1
10        else
11            h = if-then-else(z, h0, h1)
12        insertInResultCache({(N(V, V'), S(V)}, h)
end
```

Figure 31: Algorithm for computing relational products.

is, how the states in the current state set are mapped to the states in the next state set. All that the relational product returns is the set of new states. The state-to-state information, along with the rates of such state transitions is required for later performance evaluation to construct and analyze the underlying Markov chain.

We have found no method, or been able to devise an efficient one, where the existing relational product algorithm can be modified to produce the rates. We thus perform the relational product in two steps: first combining the current state set with the transition relation using a simple BDD and operation; and then performing the existential quantifier on that result by using the restriction algorithm described in Figure 10.

### 5.3.4 Partitioned Transition Relations

N(V, V') is known as a monolithic transition relation, where the effects of firing each transition are combined into one function, and it is used in finding the set of successor states from a set of predecessor states via a predicate transformation. Monolithic transition relations are known [31] to become very large and unmanageable. For this reason it is better to use the relations for the individual transitions and find each result independently. This technique is known as using partitioned transition relations.
CHAPTER 5. STATE SPACE EXPLORATION METHODS

The result of each transition relation is determined separately. This involves finding the one-step successor states for each transition given the set of unexplored states. Each set of new states that is found is combined into a set of the total states found from this level of the state graph. From Equation 40, we see that the new states, $S'(V')$, can be computed from:

$$S'(V') = \exists_{v \in V} [S(V) \land N(V, V')]$$  \hspace{1cm} (41)

where $V$ is the set of current state variables and $V'$ is the set of next state variables. From the transition relations defined per transition in Equation 38, we derived Equation 39. Substituting this into Equation 41 we get:

$$S'(V') = \exists_{v \in V} [S(V) \land (N_0(V, V') \lor ... \lor N_n(V, V'))]$$  \hspace{1cm} (42)

By distributing the existential quantification over the disjunctions we find:

$$S'(V') = \exists_{v \in V} [S(V) \land N_0(V, V')] \lor ... \lor \exists_{v \in V} [S(V) \land N_n(V, V')]$$  \hspace{1cm} (43)

Computing these individual quantifications and \textit{and} operations and then combining the results is an efficient way of implementing the predicate transformation. In our experience, the monolithic transition relation and the BDD for $S(V) \land N(V, V')$ are both very large, unmanageable BDDs.

We show now our algorithm for computing the predicate transformation in Figure 32. As the individual transition relations are computed, the states are iteratively added to the BDD \textit{allStates}, which records all states found at this level of the state graph. On line 6, the function \textit{promote} converts all of the successor state variables to predecessor state variables for use in the next iteration of the algorithm. For our implementation, since the state variables are interleaved, this involves simply decrementing the index of each vertex in the BDD.

The technique of using partitioned transition relations is readily compatible with our MTBDD implementation. The actual transitions found on line 4 of the algorithm in Figure 32 resemble a MTBDD with the current state variables corresponding to the rows of the rate matrix and the next state variables corresponding to the columns. When we find the actual state-pairs for each transition we can simply multiply the \textit{true} terminal node of the resultant BDD by the rate of the transition that fired. This is then added to the MTBDD for the rate matrix of the system we are analysing.
We now discuss in more detail exactly how we compute the transition relations for the transition of GSPNs. We will first show the simple case for a binary encoding, and then show how we have extended this to cope with our computational encoding.

### 5.4 Transition Relations for GSPNs

We have not found an accurate description of how to compute transition relations from Petri nets in the literature, and the work we present here is extended from that found in a technical report of Pastor et al [17]. We use an example of a simple Petri net found in [17] and displayed in Figure 33, to illustrate the technique. Our formulas and example are based on the assumption that we are using our computational encoding mechanism, although the technique for the binary encoding is very similar, and in fact simpler.

#### 5.4.1 Enabling Condition

We create one transition relation per transition and the first step in defining the transition relation is in the construction of the BDD for the enabling condition of that transition ($E_t$). This is formalised by:

```c
function predicateTransformation(S(V) : BDD) : BDD
var
  allStates
begin
  foreach (transition t) do
    begin
      let $t_r$ be the BDD relation for $t$
      $n = \text{bddAND}(t_r, S(V))$
      save $n$ to storage (in our case, an MTBDD)
      restrict($n$, varStrn(S(V)))
      promote($n$)
      allStates = bddOR($n$, allStates)
    end
  return allStates
end
```

Figure 32: Algorithm for computing predicate transformations.
Figure 33: A simple Petri net from [17].

\[
E_t = \prod_{p \in \pi(t)} M(p) \geq w(p,t)
\]  

(44)

where \( M(p) \) is the number of tokens on place \( p \) and \( w(p,t) \) is the weight of the arc from \( p \) to transition \( t \). This equation ensures that there are sufficient tokens on each of the input places of transition \( t \). The condition \( M(p) \geq w(p,t) \) is realised by the following formula:

\[
M(p) \geq w(p,t) = (p^k > w^k) \lor (p^k = w^k \land p^{k-1} > w^{k-1}) \lor ... \lor (p^k = w^k \land ... \land p^2 = w^2 \land p^1 \geq w^1))
\]  

(45)

where the \( p^i \) and \( w^i \) indicate the \( i \)th bit of bit-vector for the place and its corresponding output arc.

We know that \( a \geq b \equiv (a + \bar{b}) \), \( a > b \equiv (a \cdot \bar{b}) \), and \( a = b \equiv (a \oplus b) \) so we can derive the following simplified equation for computing the enabling condition for a particular transition:

\[
E_t = \prod_{p \in \pi(t)} \left[ (p^1 + w^1) \cdot \prod_{j=2}^{K} \left( p^j \oplus w^j \right) \right] + \sum_{i=2}^{K} \left( (p^i \cdot w^i) \cdot \prod_{j=i+1}^{K} \left( p^j \oplus w^j \right) \right)
\]  

(46)

Some of the places in the above equation, as was explained in Chapter 4, will not be stored directly in the BDD, they are the "computed" places and the values on these places are only known in terms of the invariant equations. Consequently, we need to find the bit-vector for each computed place in terms of the variables of the stored places.
As an example consider the invariant equation found for the Petri net in Figure 33 on page 76, 2p₀ + p₁ + p₂ = 4. Assume we choose the stored place set to contain the places p₂ and p₁, then the value on place p₀ can be determined by p₀ = (4 − (p₁ + p₂))/2. Therefore, since p₀ is an input place for each of the transitions t₀ and t₁ and the arc from p₀ to each of these transitions has weight one, the firing constraint we require is p₀ ≥ 1, which, in terms of the invariant equation equates to (4 − (p₁ + p₂))/2 ≥ 1. This reduces to the inequality 2 ≥ p₁ + p₂. In order to determine the enabling condition we generalise Equation 44 to be of the form \( E_t = A \geq B \), where \( A \) and \( B \) are any bit-vectors of the same length.

Using our example, we work out every bit of the equation \( p₁ + p₂ \) using simple addition. There are three bits in the addition result which we represent with \( b₁ \) to \( b₃ \). \( p_i^j \) refers to the \( j^{th} \) bit of the value on place \( p_i \).

\[
\begin{align*}
  b₁ &= p₁^1 \oplus p₂^1 \\
  b₂ &= p₁^2 \oplus p₂^2 \oplus p₁^1 p₂^1 \\
  b₃ &= p₁^3 p₂^3 \oplus p₁^2 p₂^2 \tag{47}
\end{align*}
\]

The enabling condition, \( 2 \geq (p₁ + p₂) \), is found using the general form of Equation 46. In the general form \( p = 2 \), and \( w = (p₁ + p₂) \). Since \( w \) can contain at most 3 bits, 010 is used as the binary sequence for 2. Thus, the enabling conditions for transitions \( t₀ \) and \( t₁ \) can then be written as:

\[
E_{t₀} = E_{t₁} = 0 \cdot b₃ + ((1 \cdot b₂) \cdot (0 \oplus b₀)) + ((0 + b₁) \cdot (1 \oplus b₂) \cdot (0 \oplus b₃)) = (p₁^1 p₂^2 + p₁^2 p₂^1 (p₁^2 \oplus p₂^2)) (p₁^1 + p₂^3 (p₁^2 \oplus p₂^2)) \tag{48}
\]

Since transitions \( t₂ \) and \( t₃ \) only rely on stored places, their enabling condition is relatively simple to calculate. \( t₂ \) has input place \( p₁ \) and arc weight 2. Therefore we apply Equation 46 with \( p = p₁ \) and \( w = 2 \). The enabling condition for transition \( t₃ \) is similarly derived:

\[
\begin{align*}
  E_{t₂} &= (p₁^1 + 0) \cdot (p₁^2 \oplus 1) + (p₁^2 \cdot 1) \\
  &= p₁^2 \\
  E_{t₃} &= (p₂^1 + 0) \cdot (p₂^2 \oplus 1) + (p₂^2 \cdot 1) \\
  &= p₂^2 \tag{49}
\end{align*}
\]

This is clear from the Petri net in Figure 33 because as long as the most significant bit of the bit-representation of the tokens on either of the input places for transitions \( t₂ \) and \( t₃ \) is one, those places will contain sufficient tokens for the transitions to fire, which is greater than or exactly 2 tokens.
5.4.2 Firing Function

Once we have defined the enabling conditions for each transition in terms of a Boolean formula, whose representation can be stored as a BDD, we can create the actual relation which describes the mapping from predecessor state variables onto successor state variables.

The transition relation \( R_t(P, \mathcal{P}) \) can be written in the following form:

\[
R_t(P, \mathcal{P}) = E_t \cdot \left( \prod_{i=1}^{\left| S_t \right|} \prod_{j=1}^{K_{p_i}} (p_i^j = \delta_i^j(S)) \right)
\]

(50)

where \( P \) is the set of predecessor state variables and \( \mathcal{P} \) is the set of successor state variables. \( S_t \) is the set of places stored in the BDD and which are connected to the transition \( t \). That is \( S_t = \mathcal{P}_S \cap (t \cup t^\cdot) \). Note that the computed places are not necessary for the transition firing function. They are only used to make up the enabling condition. \( \mathcal{P}_S \) is the subset of \( P \), the places in the Petri net that are stored in the BDD. Finally, \( \delta_i^j \) is the transition firing function. This Boolean function describes how tokens are added to or removed from the output and input places of transition \( t \) for every bit of the output or input places:

\[
\delta_i^j = \begin{cases} 
  p_i^j \oplus w_i^{j,t} \oplus B_{j-1} & \text{if } p_i \in t \setminus t^\cdot \\
  p_i^j \oplus w_i^{j,p_i} \oplus C_{j-1} & \text{if } p_i \in t \setminus t^\cdot \\
  p_i^j | a^j | \oplus B_{j-1} & \text{if } a < 0 \text{ and } p_i \in t \cup t^\cdot \text{ where } a = W(t, p_i) - W(p_i, t) \\
  p_i^j | a^j | \oplus C_{j-1} & \text{if } a > 0 \text{ and } p_i \in t \cup t^\cdot \text{ where } a = W(t, p_i) - W(p_i, t)
\end{cases}
\]

(51)

where \( C \) and \( B \) are the Boolean addition and subtraction functions:

\[
C_j = \begin{cases} 
  0 & \text{if } j < 1 \\
  p_i^j \cdot w_i^{j,p_i} + C_{j-1}(p_i^j \oplus w_i^{j,p_i}) & \text{if } j \geq 1
\end{cases}
\]

(52)

\[
B_j = \begin{cases} 
  0 & \text{if } j < 1 \\
  p_i^j \cdot w_i^{j,p_i} + C_{j-1}(p_i^j \oplus w_i^{j,p_i}) & \text{if } j \geq 1
\end{cases}
\]

We use the equality \((a = b) \equiv (ab + a\overline{b})\) to evaluate the \( p_i^j = \delta_i^j(S) \) component of Equation 50.

Returning to our example, we see that it is not necessary to compute transition relations for place \( p_0 \)

\(^1\)In this section, we underline successor state variables to distinguish them from predecessor state variables.
as this place is not stored anywhere in the BDD. We use the collective tokens on places $p_1$ and $p_2$ in order to compute the tokens on $p_0$. This information is stored implicitly in the enabling functions for the transitions $t_0$ and $t_1$. The final transition relations for all the transitions of the Petri net seen in Figure 33 are:

$$
R_{t_0} = E_{t_0} \cdot \left( (p_1 = \overline{p_1}) \cdot (p_2 = (p_2^1 \lor p_1^1)) \cdot (p_2 = (p_2^2 \lor p_1^2)) \right) \\
R_{t_1} = E_{t_1} \cdot \left( (p_2 = \overline{p_2}) \cdot (p_2 = (p_2^2 \lor p_2^2)) \cdot (p_1 = \overline{p_1}) \cdot (p_2 = (p_2^1 \lor p_2^1)) \right) \\
R_{t_2} = E_{t_2} \cdot \left( (p_1^1 = p_1^1) \cdot (p_1^2 = \overline{p_1^2}) \right) \\
R_{t_3} = E_{t_3} \cdot \left( (p_2^1 = p_2^1) \cdot (p_2^2 = \overline{p_2^2}) \right)
$$

(53)

With these functions defined we can make use of the relational product algorithms to find the successor states of transitions given an input set of states. Computing the relations for immediate transitions is the same as that described above. They are, however used differently in the state space exploration algorithm.

### 5.4.3 Vanishing State Elimination

We have not found a description in the literature for removing the vanishing states on-the-fly in the state space exploration algorithm when making use of transition relations. We thus present our technique, in the context of the standard approach to state space searches on SPNs.

Once the transition relations have been found for each transition, including the immediate transitions, the initial state of the system is converted to a BDD by means of its characteristic function. This state is checked against all of the immediate transitions to check whether it is a vanishing state. The state is vanishing if, for any of the transitions the logical $\land$ ($\land$) of the initial state with the relation results in anything other than the $false$ BDD – a BDD consisting solely of the zero, or $false$ terminal.

Any non-zero BDDs that are found as a result of this test are combined using the BDD or operator. Once all of the immediate transitions have been accounted for, the combined BDD is then applied with the existential quantifier operator which removes the predecessor state variables. The remaining successor variables are promoted to predecessor variables and this set is designated as the initial state set, or initial frontier set.
For the remainder of the state space exploration, the frontier set is combined, using the and operator, for each timed transition, as described by the algorithm in Figure 31. Notice that only timed transitions can be enabled at this point as all vanishing states would have been eliminated at this point.

This resultant BDD is actually an MTBDD matrix containing the entry 1 for state pair. Applying the existential quantification removes the predecessor states and we are left with only the successor states. In the same way in which we dealt with the initial state, this set of successor states must be tested against each of the transition relations for the immediate transitions.

Once again, if any returned BDD is not the zero BDD, then this indicates that the immediate transition in question is enabled. The returned BDD in this case is an MTBDD comprising a matrix with 1 entries for each map of vanishing state onto a tangible or vanishing state. This matrix must be subtracted from the original rate matrix, as must the restricted predecessor states be removed from the original successor state set. The restricted successor states must be checked again, in a recursive fashion against each immediate transition. Any vanishing states subsequently found are treated in the same way, while the tangible states are kept and the effective rates of finding these states from the initial timed transition are maintained. Finally, all the state-to-state rates of each tangible to tangible state change are collected and added to the original MTBDD for the state-to-state rates.

This process may seem costly and tedious, but the alternative is to remove all the vanishing states at the end of the exploration process which requires a greater memory commitment.

5.4.4 Discussion

It is important to note that there are two sets of variables in the transition relation: the current and next state variables. The BDD for this relation will thus be double the height of the ordinary BDD required to simply store the states. Using heuristics, the current and next state variables are usually interleaved in the BDD as this seems to lead to a more compact BDD. Bryant [9] discusses BDD ordering for compactness in more detail.

Transition relations are a very elegant solution and have been successfully used in circuit theory, but they are much slower for GSPNs, even with the computational invariant encoding. The major weakness of this technique is the application of the quantitative restriction function, which is a great computational bottleneck. This technique will only become viable if this function can be improved upon. Additionally, saving the state-to-state rates slows the process considerably. Finally,
the process of finding the transition relations is very time consuming, although transition relations can be saved for later explorations and only computed once for each GSPN.
procedure findAllSuccessorsOf(array of integers : path, BDD : T)
begin
1  set Petri net marking using path
2  foreach (enabled transition, t) do
   begin
3       \( S = fire(t) \)
4       newPath = find new path \( S \)
5       addNewPath(newPath, root(T))
   end
end.

procedure addNewPath(array of integers : path, Vertex : v)
begin
1  if (index(v) == (maximum index - 1))
2     if (path[index(v)] == 1)
3       begin
4          high(v) = true
5          return
6       end
7     else
8       begin
9          low(v) = true
10         return
11       end
12  if (path[index(v)] == 1)
13     begin
14        if (high(v) == false)
15          begin
16             high(v) = new Vertex with index \( index(v) + 1 \) and both children false
17             addNewPath(path, high(v))
18          end
19        else if (path[index(v)] == 0)
20          if (low(v) == false)
21            begin
22               low(v) = new Vertex with index \( index(v) + 1 \) and both children false
23               addNewPath(path, low(v))
24            end
25     end
end.

Figure 34: Finding successor states using Incremental Reduction.
procedure findAllSuccessorsOf(array of integers : path, BDD : T)  
var  
vse : Stack  
begin  
1      set Petri net marking using path  
2      foreach (enabled transition, t) do  
3             $S = fire(t)$  
4             if ($S$ is vanishing)  
5                 $r = rateOf(t)$  
6                 push {$S, r$} onto vse  
7                 while (vse is not empty) do  
8                     $S_v = pop(vse)$  
9                     foreach (enabled transition, t) do  
10                        $S_f = fire(t)$  
11                        if ($S_f$ is vanishing)  
12                            push {$S_f, r$} onto vse  
13                           else  
14                               begin  
15                                   newPath = find new path from $S_f$  
16                                   addNewPath(newPath, root(T))  
17                               end;  
18                           end;  
19                     end;  
20                     newPath = find new path from $S$  
21                     addNewPath(newPath, root(T))  
22                 end;  
23             end;  
24         end;  
25     end.  

Figure 35: Finding successor states extension for coping with vanishing states.
Chapter 6

Markov Chain Solution Methods

6.1 Introduction

The primary focus of this dissertation is on state space exploration from GSPN models. The goal of the exploration is to record the data necessary to carry out performance analysis on the model. All the state space exploration techniques we have presented in this dissertation have been tailored to suit the needs of such analysis. As a secondary objective, we have investigated an MTBDD-based (c.f. Ch. 3) approach to the solution of the continuous time Markov chain underlying the GSPN.

Through the course of the state space exploration process we record both the states found and the individual state-to-state transition. These rates make up the rate matrix, $R$, from which we can derive the infinitesimal generator matrix $Q$. We can find the stationary probability distribution of the state space using the steady-state equations. This involves solving the following set of linear steady-state equations:

$$
\Pi Q = 0, \quad \text{and} \quad \sum_i \pi_i = 1
$$

(54)

where $\Pi$ is the vector of the steady-state probability distribution of length $|S|$, where $S$ is the state space. The steady-state equation is of the form:

$$
A\pi = b, \quad \text{where} \quad A = Q^T \quad \text{and} \quad b = 0^T
$$

(55)

There are several well-known methods of solving such equations. There are two broad categories:
CHAPTER 6. MARKOV CHAIN SOLUTION METHODS

- **Direct methods**: consisting of techniques such as LU-decomposition and Gaussian elimination. These techniques aim at rewriting the expressions for the steady-state probabilities in such a way that explicit expressions are formed for each state. Unfortunately, given the time complexity of these methods they are only viable for small state spaces of less than 1000 elements.

- **Iterative methods**: where direct methods fail on large state spaces we can make use of iterative methods, which are well suited to sparse matrix algorithms and data structures. Iterative techniques do not yield an exact solution of the system of equations, but rather a set of operations is iteratively performed until a desired level of accuracy is reached. It is not known *a priori* how many iterations will be required to reach the desired accuracy, and therefore one cannot predict how long the numerical solution of Equation 54 will take.

In this chapter we consider two iterative techniques that we have implemented. As mentioned, the techniques we employ for state space exploration are *implicit*, using BDDs as the underlying data structure. Using BDDs yields a good memory saving and thus simply saving the state transition rates to disk for a later disk-based solution is impractical. Additionally, the disk I/O severely slows the state space exploration algorithm. We could make use of any of the available sparse matrix packages or techniques, and we chose the MTBDD data structure, a close relative of BDDs, as part of the research for this dissertation.

We thus discuss the solution techniques with reference to the restrictions imposed by the MTBDD data structure which we use primarily to conserve memory. Particularly, MTBDDs are not feasible for the implementation of direct solving methods. MTBDDs are a sparse matrix technique and since direct methods manipulate the matrix $A$ causing "fill-in", where zero elements become valued elements, they are unsuitable. Furthermore, "fill-in" also occurs whenever a matrix inversion is performed.

We have implemented the Jacobi iteration method using MTBDDs as part of our experimental toolset. The theory for this was presented by Siegle *et al* [24], we present the theory here with details of our implementation. Additionally, we have examined the suitability of Krylov subspace techniques with MTBDDs as the underlying data structure. We have not encountered this anywhere else in the literature. We present the theory of the conjugate gradient squared (CGS) algorithm and describe our implementation.
6.2 The Method of Jacobi

Amongst the best known iterative methods are the Jacobi and the Gauss-Seidel methods. For classical sparse matrix techniques, Gauss-Seidel requires less memory than the method of Jacobi. However, the Gauss-Seidel technique involves a matrix inversion which causes fill-in, decreasing the memory efficiency of the sparse matrix technique. Furthermore, when MTBDDs are used as the storage data-structure, accessing the vector elements individually, as is required by the Gauss-Seidel iterative method, is not time efficient. The same applies to successive over-relaxation techniques, which are also unsuitable. The method of Jacobi, on the other hand, involves a vector-matrix multiplication per iterative step, with no matrix inversions. In terms of memory requirements, it is well-suited to implementation via MTBDDs.

The Jacobi method involves re-writing the \( i_{th} \) equation of Equation 55 as

\[
p_i = \frac{1}{|a_{ii}|} \left( \sum_{j \neq i} p_j a_{ij} \right)
\]  

(56)

\( a_{ii} \neq 0 \) is guaranteed when the linear system of equations to be solved is for the steady-state probabilities of an irreducible, aperiodic CTMC. The iterative procedure begins with an initial guess at the probability distribution, denoted \( p^0 \). This starting vector can be chosen in a number of ways. We can either set one element to 1 and leave the rest 0, we could set each variable in the vector to a random number in the range \((0, 1)\). Both of these techniques are unsuitable in an MTBDD environment. The first requires an \( O(D) \) operation, where \( D \) is the depth of the MTBDD, and the second requires and \( O(n \cdot D) \) operation where \( n \) is the length of the state probability vector, followed by a MTBDD reduce operation. In terms of MTBDDs, using a uniform probability distribution is the best choice, where every element is initially set the value \( 1/n \). This is an \( O(1) \) operation and the result is simply an MTBDD consisting of one terminal node with the value \( 1/n \). Given the starting vector or any other estimate for \( p \), the next estimate is calculated by:

\[
p_i^{(k+1)} = \frac{1}{|a_{ii}|} \left( \sum_{j \neq i} p_j^{(k)} a_{ij} \right)
\]  

(57)

Written in matrix form, Equation 57 becomes:

\[
p^{(k+1)} = D^{-1}(L + U)p^{(k)}
\]  

(58)
where \( D = \text{diag}(a_{ii}) \) and \( L \) and \( U \) are the lower and upper triangular parts of \( A \) respectively. Note that \( D^{-1} \) is found by the componentwise inversion of each of its entries. In MTBDDs this involves the inversion of the values in each of its terminal nodes, which are stored in a list and therefore easily accessed. A reason for using MTBDDs is the prospect that there will be much fewer terminal nodes than elements in the vector.

For computational simplicity we use the \textit{difference criterion} as our stopping condition. That is when \( \|p^{(k+1)} - p^{(k)}\| < \epsilon \), where \( \epsilon \) is the desired level of accuracy chosen by the modeler. This condition does not always imply that convergence has been met, in cases with slow convergence the difference between successive iterates could be quite slow. Checking the residual criterion to avoid this problem is computationally expensive however, and we have not implemented it for the method of Jacobi. In our package we subtract the elements of the successive iterates componentwise and then find the maximum element of the result, which we compare to \( \epsilon \).

### 6.2.1 MTBDD Implementation

Our experimental tool-set produces an MTBDD representing the rate matrix, \( R \), as a result of the state space exploration. This matrix has the source state number as the row index and the target state number as the column index and is equivalent to \( (L + U) \). To generate the matrix \( A \) we would need to set the diagonal elements of \( R \) to the negative of the row sums for each row, and then transpose the matrix. In MTBDDs information about how many rows and columns the matrix has is not stored implicitly, rather we have separate variables to record this. For simplicity, in our implementation, we do not transpose the matrix, we rather perform the vector-matrix multiplication on the left as opposed to on the right as it is in Equation 58.

In our implementation, the iteration matrix for the method of Jacobi is \( \Phi_J = R \cdot D^{-1} \). Each successive iteration is then calculated by \( p^{(k+1)} = \Phi_J p^{(k)} \). To find \( D \) we calculate the row sums of \( R \) by performing a vector by matrix multiplication of the row vector \( \mathbb{1} \) with \( R \). The result is a row vector with each entry \( i \) equal to the sum of the row of the matrix \( R \) at row \( i \). This must be diagonalised where each element is placed along the diagonal entries of a matrix with dimension equal to the length of the vector. The diagonalise operation is straight-forward for an \( m \)-reduced MTBDD, but not for a reduced MTBDD, as is our case. We have a primitive diagonalise function which places the vector elements on the diagonal of the matrix individually. This operation is \( O(D) \) for each element where \( D \) is the depth of the MTBDD for the diagonal matrix.
To obtain $D^{-1}$ we simply invert each element in a componentwise fashion, which involves iterating through each terminal entry. Finally, we perform a matrix by matrix operation $R \cdot D^{-1}$ to calculate $\Phi_J$. The generation of $\Phi_J$ can be summarised by the following equation:

$$
\Phi_J = \text{matrixMultiply}(R, \text{componentInvert}(\text{diagonalise}(\text{vectorMatrixMultiply}(1, R)))) \quad (59)
$$

The matrix multiply shown above is not the full algorithm described by McGeer et al [20]. It is optimised as described by Siegle [24] for the special case when we are multiplying a square matrix on the right by a diagonal matrix. In this case, we need simply multiply each column of $R$ by the corresponding entry in the diagonal matrix.

### 6.3 Conjugate Gradient Squared

Krylov subspace techniques are a collection of non-stationary iterative methods for solving systems of linear equations. They differ from stationary methods, such as the Jacobi method, in that each step involves computations that rely on information which is also changed at each step. Each iterate is generated using a shifted Krylov subspace associated with the coefficient matrix for the system of equations.

Improvements to the method of Jacobi, such as the Gauss-Seidel and successive over-relaxation techniques, are not feasible when implemented in MTBDDs. Krylov subspace techniques are an efficient [33] alternative to these techniques. We have investigated the CGS method as a preliminary study. It uses no matrix inversions and each step involves two inner-products and two vector-matrix multiplications per step.

In this section we describe the Conjugate Gradient Squared (CGS) algorithm. A detailed discussion of Krylov subspace techniques can be found in [47]. The pseudocode for the algorithm is shown in Figure 36. We use Greek letters ($\alpha, \beta, \gamma, ...$) to denote scalar values. Once again we do not transpose $Q$, we modify the vector-matrix multiplications for the same effect. Since the residual vector is readily available in the CGS algorithm we have implemented, we use it to check whether convergence has been reached. We say convergence has been reached when $\|r^{(k)}\|_{\infty}/\|x^{(k)}\|_{\infty} < \epsilon$ where $\|x^{(k)}\|_{\infty}$ is the infinity-norm, given by the maximum element of $x^{(k)}$ [33].
In this algorithm, adapted from [32], [33], and [47], for simplicity we have set the starting vector, \( z^{(0)} \), to have one element of 1, and the rest zero. Comparisons of the method of Jacobi with this technique will be made with this initial distribution. The residual vector update on line 21 of Figure 36 computes of the true residual \( r^{(k)} = -Az^{(k)} \) to avoid the cumulative errors which occur in finite precision arithmetic using a residual update method.

For a complete discussion of the results we obtained in experimentation with the CGS algorithm and the method of Jacobi, we refer the reader to Chapter 7.
function cgs Solve($R : MTBDD$)
begin

1. $A = \text{apply}(R, \text{diagonalise} (\text{vectorMatrixMultiply}(1, R)))$ .
2. $x^{(0)}$ is the first guess at the solution, with single entry 1
3. $\hat{r} = r^{(0)} = -\text{vectorMatrixMultiply}(A, x^{(0)})$
4. $\rho^{(0)} = 1$
5. $\kappa = 0$
6. $p^{(0)} = q^{(0)} = 0$

7. do

8. begin

9. $\kappa = \kappa + 1$

10. $\rho^{(\kappa)} = \hat{r} \cdot r^{(\kappa-1)}$

11. $\beta = \rho^{(\kappa)}/\rho^{(\kappa-1)}$

12. if ($\kappa == 1$)

13. $p^{(\kappa)} = u = r^{(\kappa-1)}$

14. else

15. begin

16. $u = r^{(\kappa-1)} + \beta q^{(\kappa-1)}$

17. $p^{(\kappa)} = u + \beta (q^{(\kappa-1)} + \beta p^{(\kappa-1)})$

18. end;

19. $v = Ap^{(\kappa)}$

20. $\alpha = \rho^{(\kappa)}/(\hat{r} \cdot v)$

21. $q^{(\kappa)} = u - \alpha v$

22. $w = u + q^{(\kappa)}$

23. $x^{(\kappa)} = x^{(\kappa-1)} + \alpha w$

24. $r^{(\kappa)} = -A x^{(\kappa)}$

25. end;

26. while ($||r^{(\kappa)}||_\infty / ||x^{(\kappa)}||_\infty < \epsilon$)

27. normalise $x^{(\kappa)}$

end

Figure 36: Conjugate Gradient Squared Algorithm
Chapter 7

Experimental Results

This chapter presents our experimental data along with a discussion of the results we have obtained through testing with our tool-set.

In Section 7.1 we show how we have implemented the penultimate step in the performance analysis pipeline: finding the performance metrics of the GSPN model under analysis. Section 7.2.5 presents the results of our experiments with our state space generator. We show how different encodings and partitions have affected the BDD memory usage. Finally, in Section 7.2.8, we discuss the results of our investigation into using MTBDDs in the steady-state solution of CTMCs.

7.1 Performance Metrics

The steady-state probability distribution vector shows, for every state, what the probability is of the model being in that state at any point in time. Performance "metrics" can be derived from this vector which are more meaningful than single probability values associated with each state.

The performance metric computations we have implemented are directly relevant to GSPN models, which are our choice of modeling formalism. Additionally, all of our techniques are BDD-based and we have not encountered, in the literature, examples of how to derive performance metrics based on BDDs. This is an important problem because the state description vector can also be a very large structure which is difficult to process using conventional memory storage.

We have implemented techniques to find each transitions' throughput, and to find the mean number of tokens on each of the GSPN places. When related back to the real-world system, transition
throughput indicates how frequently each of the system activities are used while the mean number of tokens on a place, or set of places, can be used to show the relative load on different parts of the system. For example, the mean number of tokens on a place could indicate the average number of people waiting in some part of a queuing network.

7.1.1 Transition Throughput

The throughput of a timed transition is given by its mean number of firings at the steady state. \( \overline{d}_j \) is the throughput at transition \( t_j \), given by:

\[
\overline{d}_j = \sum_{s_i \in EN_{t_j}} \pi_i \lambda_j
\]  

(60)

\( \lambda_j \) is the rate of timed transition \( j \) and the sum, in Equation 60, is over the steady state probabilities of all the states, \( S_i \), in which transition \( t_j \) is enabled.

<table>
<thead>
<tr>
<th>program Calculate Transition Throughputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
</tr>
<tr>
<td>1  foreach (transition ( t_j )) do</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>end;</td>
</tr>
</tbody>
</table>

![Figure 37: MTBDD-based calculation of transition throughput.](image)

Our technique for calculating the transition throughput for each transition is shown in Figure 37. We found transition relations, described in Chapter 5 on page 69, to be unsuitable for state space exploration of GSPN models. However, they turn out to be quite useful and efficient in the calculation of transition throughput.

A step in finding transition relations is the creation of the BDD for the enabling condition for each transition. The enabling condition is a BDD representing a Boolean function that evaluates to \textit{true} when the combination of variables, which represent values on the places of the transitions’ input set, satisfies the enabling rules for the transition. Put simply, there is a path in the enabling BDD for each transition from the root to the \textit{true} terminal for each state which enables that transition. We
can use this function to extract all the states from the steady-state probability vector \( \Pi \) which enable a particular transition.

![Figure 38: Illustrating the effect of multiplying an enabling function with a steady-state vector.](image)

Line 2 of the algorithm in Figure 37 finds the enabling condition for the particular transition. Note that it is not necessary to compute this every time performance analysis is done. If the GSPN is unchanged then the BDD for the enabling condition can simply be re-used. Line 3 performs a componentwise matrix multiply operation, described by the algorithm in Figure 14 on page 34, of the steady-state probability vector with the enabling condition of the transition. This has the effect, illustrated in Figure 38, of filtering out the steady-state probabilities relevant to a particular transition.

Once we have isolated the steady-state values corresponding to the states in which a particular transition is enabled, we use the function \( \text{sumComponents} \), on line 4, to sum the entries of the resulting vector and then multiply the result by the rate of the current transition.

This technique has the advantage of not needing each transition to be tested in each explored state to see if it is enabled. The enabling condition is found once for each transition and then a simple MTBDD \( \text{apply} \) algorithm is used which is \( O(|\Pi| \cdot |EN_{t_j}|) \) in time complexity where \( \Pi \) is the steady-state probability distribution vector and \( EN_{t_j} \) is the BDD of the enabling condition for the \( j^{th} \) transition.

### 7.1.2 Mean Number of Tokens per Place

The average number of tokens, \( \overline{m}_i \), on a place, \( p_i \), can be determined by:

\[
\overline{m}_i = \sum_{j=1}^{n} (\text{tokens on place } p_i \text{ in state } j) \times \pi_j
\]  

(61)
Where $n$ is the total number of explored states.

```
program Calculate Mean Tokens per Place
var
    path : array of Boolean,
    means : array of Real
begin
    calculateMeanTokensStep(root of MTBDD for state vector)
procedure calculateMeanTokensStep($v$: Vertex)
begin
    if ($v$ is terminal)
    begin
        use path to determine GSPN place values
        foreach ($p_i$ $\in$ $P$) do
            means[i] = means[i] + (($\text{tokens on } p_i$) $\times$ (value of $v$))
        end;
    else
        begin
            path[index($v$)] = false
            calculateMeanTokensStep(low($v$))
            path[index($v$)] = true
            calculateMeanTokensStep(high($v$))
        end;
    end.
end.
```

Figure 39: MTBDD-based calculation of transition throughput.

Figure 39 shows a simplified version of our algorithm for computing the mean number of tokens on a place. The mean tokens per place can be conveniently computed in one pass of the MTBDD for the steady-state probability vector. In the same way in which we determined the new states, described in Chapter 5 on page 61, the traversed path is recorded to an array called "path". We handle skipped vertices caused by reduced MTBDDs in the same way as described in Section 5.2.1.1. When the recursion reaches a terminal node of the MTBDD, we use the real-valued entry at the terminal to update the mean value of tokens on each place by the computation on line 5.

Once again, with this MTBDD-based technique it is not necessary to iterate through the entire set of explored states and, for each state, set the marking of the GSPN and update the mean number of tokens for every place. This would be an $O(|S|)$ operation where $S$ is the state space. In our case the operation is $O(|\Pi|)$, where $\Pi$ is the MTBDD for the steady-state probability vector. This
CHAPTER 7. EXPERIMENTAL RESULTS

technique is only effective in cases where $|\Pi| < |S|$.

7.2 Experimentation

In this section we present a discussion of our results from testing with our different techniques for state space exploration and steady state solution.

7.2.1 Encoding Mechanism

We have explored encoding mechanisms specifically for GSPNs, but it is possible to create an encoding scheme for any modeling formalism of a state-transition system. Our work greatly improves upon a binary encoding scheme where the binary representations of the numbers of tokens of each of the places are concatenated to form the state descriptor to be stored in a BDD.

We presented the computational invariant encoding scheme in Chapter 4 which uses the structural properties of Petri nets to reduce the number of places that must be stored in the BDD. In experimentation with this technique with several different GSPN models we found an average improvement of 58% on the height of the BDD over the use of binary encoding methods. In one example of the GSPN for the Trivial File Transfer Protocol (TFTP), shown in Figure 40, a saving of 33% of the height of the BDD was made over the binary encoding method. The TFTP model required a BDD height of 24 using our invariant-based method, down from the original height of 36 when encoded using the binary method.

The least improvement we encountered was 30% of the height required for a binary encoding while, in an extreme example, the computational encoding mechanism yielded an 80% reduction in the height of the BDD. It is important to note that the height of the BDD directly impacts the time taken for a BDD traversal, and consequently all BDD manipulation algorithms. Indeed, using the binary encoding technique we were unable to explore more than $10^3$ states, while with our invariant encoding method we have explored $10^8$ states.

7.2.2 Exploration Techniques

In Chapter 5 we described our approach to state space exploration by using BDDs as the underlying storage data structure. In Section 5.2.1.1, on page 61, we introduced our concept of "incremental
reduction”, a technique to take advantage of the time efficiency of adding states to unexpanded BDDs, and exploit the memory conservation offered by reduced BDDs for state storage.

Section 5.2.1.2 describes an enhancement to this basic technique. The standard breadth-first state space exploration algorithm, shown in Figure 23 on page 57, contains a line to remove the previously explored states from the set states found at the current level of the state graph, thereby maintaining a frontier set of unexplored states – equivalent to the unexplored state stack in the depth-first version.

Our technique attempts to take advantage of the speed BDDs offer in detecting whether a state has been found already or not. This amounts to an $O(D)$ operation where $D$ is the depth of the BDD used for the storage. In the worst case this will be the same order as a comparison of hashed states which also has to compare each bit of the new state with each bit of the stored state, assuming that pointer arithmetic is of similar computational effort as array-index arithmetic. In addition to the time and memory saved by not re-adding already explored states, we also avoid having to compute
the frontier set at every iteration in the breadth-first search.

![Graph showing speedup](image)

Figure 41: Graphs showing the speedup achieved by employing techniques to conserve the frontier set, and improving the transition enabled detector.

In Section 5.2.1.3, we discuss one further enhancement to our BDD traversal algorithm. We realised that it is not necessary to wait until the traversal reaches a BDD terminal node before the transitions can be checked whether they are enabled or not. Since the state description vector is built up as the BDD is traversed recursively from root to terminal node, token values on some places will be known before the terminal is reached, therefore certain transitions will have the token values set on all of their input places and whether they can fire or not will be known.

Figure 41 displays a graph which shows how the above two enhancements have improved the time efficiency of our algorithms. For this graph we used the GSPN for the classic dining philosophers problem. This GSPN exhibits an extremely regular order and the repeating subnet is shown in Figure 51 on page 114. On the graph's y-axis we have the speedup factor we obtained by implementing our techniques. On the x-axis we have placed the technique we used. The graph clearly shows that the technique of conserving the frontier set and the technique of optimising the enabled transition detection gave good speedups. With 15 diners, the speedup factor of using our new techniques gave a speedup factor of 5.

There are two interesting conclusions that can be drawn from this graph. The first is that the higher the number of transitions, the more effective our enabled transition detection technique was. Secondly, that the efficiency of the technique of preserving the frontier set becomes less noticeable on
CHAPTER 7. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Technique</th>
<th>Peak Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 Diners</td>
</tr>
<tr>
<td>Standard</td>
<td>217296</td>
</tr>
<tr>
<td>Frontier</td>
<td>164280</td>
</tr>
</tbody>
</table>

Table 2: Comparing the peak memory usage of a standard technique with one which preserves the frontier set.

deeper BDDs.

Finally, consider Table 2 where we show the relative peak memory usage of the "conserved frontier" technique, versus a standard technique. It is clear that preserving the frontier set in this way gives a good memory saving. In the case of 15 dining philosophers, we managed to save 20.9% of the peak memory required.

7.2.3 The Effect of Variable Ordering

Variable ordering is of critical importance to the success of any tool which uses BDDs as the underlying data structure. We presented three different techniques for ordering variables in BDDs based on a bit-vector from a GSPN state descriptor in Section 4.5 on page 49. In this section we present our experimental results in testing with these techniques. We compare all our results to a random ordering to show the importance of selecting an ordering according to some structured pattern, rather than haphazardly arranging the variables. We will not refer to ordering of variables in this discussion, rather we refer to the order of places in the BDD path sequence. As we have mentioned in Section 4.5, ordering the individual variables of the places had no significant effect on the resultant BDD size.

The four ordering methods we experimented with were:

1. Random: where the places' values which make up the bit-vector are arranged in an arbitrary order.

2. Transition-Based: where, as described in Section 4.5, the places are partitioned into sets which form the input set of the same group of transitions;

3. Arc-Based: where the places are sorted by similarity on their number of incoming and outgoing arcs. In this way we hope to interleave places in similar subsets of GSPNs with no regular structure;
Table 3: Comparing the peak memory usage of a standard technique with one which preserves the frontier set.

4. **User-Based**: where a user with insight into the model tried to pick the optimal ordering.

Both the arc-based and the user-based techniques tried to interleave the places of similar subnets of the GSPN under analysis. This has turned out to be the most efficient ordering technique. We have presented the data from our experiments in two ways.

Firstly, consider Table 3, where we show for each model we tested, and for each variable ordering method, the number of states explored, the time taken in seconds, and the peak memory required in bytes. The models we experimented with can be found in Appendix A, along with a brief description of each. "12 Diners" refers to 12 of the repeating subnet shown in Figure 51a.

We chose to scale these models so that they all generate approximately the same number of states, which we chose to be $10^5$, for comparison purposes. We made use of purely timed version of all the models in Appendix A. We did this to be able to easily generate large state spaces, by converting all the immediate transitions to timed transitions. We will now mention some specific points, relevant to each model, before discussing some general trends which we have experienced.

The Dining Philosophers problem is highly structured, with many repeating subnets. In this example, clustering together the places common to the input sets of the transitions of the GSPN, was the worst ordering we experienced. Even randomly ordering the places was better than clustering transition input sets together. The random ordering took nearly half the time of the transition-based method. Interestingly, with the dining philosophers problem, our own arc-based technique was superior to one chosen by a user with insight into the problem. Both the user and the algorithm use the same technique of trying to interleave subnets, however.

Again in the Kanban manufacturing system, shown in Figure 53 in Appendix A, randomly selecting the ordering was superior to clustering the transition input sets. The regularity of this model made it simple for a user to choose the optimal ordering, interleaving the places of the GSPN, and performing significantly better than the arc-based technique.
Both the Kanban system and the dining philosophers are extremely regular structures. Looking at less structured models, such as the FMS model, shown in Figure 54 in Appendix A, we find similar results, but not as pronounced. The transition ordering is still the worst attempt, and the arc- and user-based techniques are very similar.

The Courier telecommunications protocol, shown by the GSPN in Figure 52, has very little regular structure which we can exploit. The name "courier-1,1", in Table 3 implies that both places NNN and MMM have one token, in that order. It is interesting to note that, in this case, the random ordering is now worse than a transition-based approach, and in fact even better than the arc-based method. This suggests that, for this model, clustering the input places of transitions is a reasonable automated approach. However, the user, when deciding on the ordering, used the technique of interleaving places. Referring to Figure 52, the user interleaved places p0, p1, p33, and p34 of the first "row" of places, and continued in a similar trend for the remainder of the GSPN. In this way, the user tried to interleave the places of the sender "sub-net" with the places of the received "sub-net", resulting in a very good ordering, almost twice as good as either automated approach and approximately 2.5 times better than a random choice.

The parsys model, the GSPN shown in Figure 55, returned very similar results to the Courier model. The random choice was worst, bettered marginally by a transition-based technique. In this case the arc- and user-based method performance similarly. We can attribute this to the simplicity of the parsys model relative to the Courier model. It was more likely that our arc-based technique chose an optimal ordering of places on their number of incoming and outgoing arcs, as there were fewer places to order.

The graph in Figure 42 shows some interesting general trends we have found in the course of our experimentation. Figure 42 shows four bar-graphs, one for each of the ordering techniques with which we have experimented. Each bar corresponds to the number of states per second the technique allowed, given a particular model. For example, the leftmost bar shows that approximately 2,800 states per second were found using a random ordering on the Kanban model with 3 Kanbans. Finally, for each technique, the bars corresponding to each model are in the same order, which is that of decreasing regular structure. The most structured model is the Kanban model and the least structured is the parsys GSPN. We now examine the trends that can be deduced from this graph.

Firstly, on each of the four graphs, there is a decrease in states per second as the regularity of the model structure decreases. This confirms that, generally, symbolic methods are less efficient on less regular GSPNs.
CHAPTER 7. EXPERIMENTAL RESULTS

Both the Kanban system and the dining philosophers are extremely regular structures. Looking at less structured models, such as the FMS model, shown in Figure 54 in Appendix A, we find similar results, but not as pronounced. The transition ordering is still the worst attempt, and the arc- and user-based techniques are very similar.

The Courier telecommunications protocol, shown by the GSPN in Figure 52, has very little regular structure which we can exploit. The name “courier-1.1”, in Table 3 implies that both places \( N/N \) and \( MMM \) have one token, in that order. It is interesting to note that, in this case, the random ordering is now worse than a transition-based approach, and in fact even better than the arc-based method. This suggests that, for this model, clustering the input places of transitions is a reasonable automated approach. However, the user, when deciding on the ordering, used the technique of interleaving places. Referring to Figure 52, the user interleaved places \( p_0, p_1, p_{33}, \) and \( p_{34} \) of the first “row” of places, and continued in a similar trend for the remainder of the GSPN. In this way, the user tried to interleave the places of the sender “sub-net” with the places of the received “sub-net”, resulting in a very good ordering, almost twice as good as either automated approach and approximately 2.5 times better than a random choice.

The \textit{parsys} model, the GSPN shown in Figure 55, returned very similar results to the Courier model. The random choice was worst, bettered marginally by a transition-based technique. In this case the arc- and user-based method performance similarly. We can attribute this to the simplicity of the \textit{parsys} model relative to the Courier model. It was more likely that our arc-based technique chose an optimal ordering of places on their number of incoming and outgoing arcs, as there were fewer places to order.

The graph in Figure 42 shows some interesting general trends we have found in the course of our experimentation. Figure 42 shows four bar-graphs, one for each of the ordering techniques with which we have experimented. Each bar corresponds to the number of states per second the technique allowed, given a particular model. For example, the leftmost bar shows that approximately 2,800 states per second were found using a random ordering on the Kanban model with 3 Kanbans. Finally, for each technique, the bars corresponding to each model are in the same order, which is that of decreasing regular structure. The most structured model is the Kanban model and the least structured is the \textit{parsys} GSPN. We now examine the trends that can be deduced from this graph.

Firstly, on each of the four graphs, there is a decrease in states per second as the regularity of the model structure decreases. This confirms that, generally, symbolic methods are less efficient on less regular GSPNs.
Figure 42: A graph showing the relative performance of different variable ordering techniques in states per second.

The graphs also show that the user-based method was better than the others in almost every case. This implies that a user, with insight into the model, can achieve a better ordering than an a simple automated approach. Note though, that the arc-based technique is always similar to the user-based approach. This is because they both try to interleave the places of similar subnets. The graphs also show that interleaving the places of individual subnets yields a better ordering than clustering places into partitions based on the transitions to which they are connected.

Finally, as has been claimed in most papers that deal with the subject of BDDs, the graphs show that the ordering of GSPN places for state space exploration is critical for success when symbolic techniques are used.

7.2.4 Case Studies

We now present results from experimentation with three GSPN models. First we compare the memory usage of our BDD-based techniques with that of a classical state space exploration method. We use the GSPN for the Dining Philosophers problem, on page 114, for this comparison.

Next we chose the Kanban manufacturing system described on page 116 for its regular structure and easy scalability. We present results in terms of the memory used and the time taken find the state space. We compare our results with that of a classical approach. Finally, we use the GSPN for the
Table 4: Comparative performance of classical and BDD storage methods. Dashed entries indicate that the process ran out memory on a SPARC Station 5 with 236MB total memory.

parsys model, described on page 115, which has a non-regular structure. We compare our results for this model to those achieved with the Kanban model, to those achieved by a hashing-based state space exploration approach. We report on the memory and time efficiency of BDD-based techniques in each case.

Finally, we present a discussion of the results we obtained in experimentation with the MTBDD-based approach to solving the underlying CTMC.

7.2.5 The Dining Philosophers Problem

Table 4 shows results from the scalable Petri net model of the Dining Philosophers problem. It demonstrates how the use of BDDs in state-space explorations, in combination with our computational invariant encoding technique, drastically improves on the memory usage of both classical storage techniques as well as the binary encoding technique.

The binary encoding method could not cope with more than 500 states while the classical techniques failed on trying to explore more than $10^5$ states in 236MB of memory. The classical techniques were tested on DNAmaca [32], a Markovian analysis tool developed in 1996 in the DNA research group. The computational encoding mechanism out-performs both of these methods in terms of memory utilisation. In the case where there are 21 philosophers we succeeded in exploring the $10^8$ states with a maximum memory consumption of just 13.7MB memory.

7.2.6 Kanban Manufacturing System

The model for the Kanban manufacturing system is described in Section A.3 on page 116. We have used a version of the model consisting of purely timed transitions, simply so that we could generate
very large state spaces. We compare our state space exploration performance in terms of memory use and time taken with that of a classical tool, DNAmaca [32], which uses a dynamic probabilistic hashing technique. Since DNAmaca handles vanishing states in the same way as our experimental tool-set, the absence of vanishing states has no effect on the performance results we present.

Before we present the results, it is important to stress that our techniques perform an exact state space exploration while DNAmaca reports a 95% confidence level on complete state coverage, being probabilistic in nature. The graph in Figure 43 compares the memory efficiency of our state space exploration technique with that of DNAmaca. Note that “MADD” in the graphs refers to our own experimental tool-set. The graph in Figure 43 shows the comparison in the amount of memory require to store the state descriptors of each of the states.

![State Memory Usage for Kanban](image)

Figure 43: Comparison of MADD and DNAmaca’s memory utilisation for storing state descriptors.

From this graph, using our techniques, we notice a linear increase in the amount of memory required to store the state descriptors, with an exponential increase in the number of states. The classical approach shows an exponential memory increase as the number of states increases exponentially. At the extreme end of the graph we see that our techniques required just 4.6MB of memory to store the 11216376 states, while the classical approach required 288MB of memory.

Using our incremental reduction approach, we limited the size of the unreduced BDD, which stores the state descriptors, to contain 200,000 nodes (approximately 4.5MB memory). We limited the MTBDD for the rate matrix to contain 400,000 nodes (approximately 9MB memory). This means that during the state space exploration, our total memory requirements never exceeded 14MB. The graph in Figure 44 shows the final memory requirements of the classical rate matrix storage versus an MTBDD-based matrix storage technique.

In terms of memory requirements, BDDs for the state descriptors and MTBDDs for the rate matrix
are far superior to classical methods. The next two graphs show a comparison of their relative time efficiency. The first graph, in Figure 45, shows the relative performance of the two techniques for smaller state spaces.

It is clear that, for smaller state spaces, the classical approach is much faster: almost thrice as fast for $10^5$ states. Both have exponential time increases with an exponential increase in the size of the state space. This trend is easily explained by the fact that the classical approaches slow down when they start to use virtual memory. Our BDD-based approach almost never relies on virtual memory.

The graph in Figure 46 shows the next two data points: for $10^5$ and $10^6$ states. The trend continues as the use of virtual memory dramatically slows the state space exploration down. From this last graph we see that our techniques overtake the classical techniques at the million-state mark.

Our state space exploration is slower than that of a probabilistic hashing approach for smaller state spaces. However, once the state space becomes larger than volatile memory can accommodate,
classical techniques simply fail. Note that simply doubling the amount of volatile memory only doubles the number of states a classical method is capable of exploring, while this would exponentially increase the number of states a BDD-based technique could find. However, doubling the processor speed of the machine on which the exploration is done would halve the exploration time for a BDD-based approach.

7.2.7 The Parsys Model

We have already seen the impact of variable ordering on the number of states per second one is capable of generating using our BDD-based techniques. It is difficult to find a suitable ordering for GSPNs with non-regular structures. We now compare the results of experimentation with the Kanban model with that of a non-regularly structured model: the parsys model.

The graphs in Figures 47 and 48 show clearly that BDD-based memory storage out-performs classical storage techniques even for this non-structured model.

Our experimental tool-set, MADD, exhibits linear growth in memory requirements as the the state space increases almost exponentially in size. In fact, once we pass the 1.5 million state mark, the memory utilisation is fairly constant, due to our incremental reduction technique.

As for the Kanban model, we compare our final memory usage to store the rate matrix resulting from the state space exploration to that required by a classical method. The graph in Figure 48 shows the massive difference between classical and our BDD-based approach. Once again we have shown the final size of the matrix storage structure and note that the entire state space exploration never required more than 14MB of memory.
As a direct comparison between the Kanban and Parsys models, we compare the memory utilisation for the final rate matrix. The state space of the Kanban-5 model involves 57,854,496 transitions and requires an MTBDD size of 44.6KB to be stored. The height the MTBDD for this model’s rate matrix is 68. The state space of the Parsys-25 model involves 43,175,925 transitions and requires 474KB of memory with an MTBDD of height 72. This is approximately ten times more memory for MTBDDs of similar heights. We attribute this difference directly to the regularity of the Kanban model.

The same trend continues for time efficiency. Our state space engine can generate the 2.5 million states of the Kanban-5 model in approximately 70 minutes, while the Parsys-25 model requires 118 minutes to generate only 1.5 million states.

In comparison of the Parsys model with classical methods, we found similar results to the Kanban
CHAPTER 7. EXPERIMENTAL RESULTS

Figure 49: Comparison of MADD and DNAmaca's time efficiency for small state space exploration for the Parsys model.

model. The graph in Figure 49 shows the relative performance for smaller state spaces, with the classical techniques well out-performing our BDD-based method.

Figure 50: Comparison of MADD and DNAmaca's time efficiency for large state space exploration for the Parsys model.

However, as larger state spaces are explored the classical state hashing approach requires virtual memory to complete and our BDD-based technique begins to perform better in comparison. The next two data values are shown in the graph in Figure 50. Once again, at the million-state mark, BDDs begin to out-perform classical techniques in terms of memory efficiency. In both cases the state space exploration is slow, but with a BDD-based approach, the exploration has a greater chance of completing and not exhausting the available memory.
7.2.8 CTMC Solution with MTBDDs

MTBDDs excellent memory utilisation makes them ideal for storage of the rate matrix during state space exploration. At the end of the exploration one could easily convert the MTBDD to any standard sparse matrix representation and solve the infinitesimal generator matrix using a standard sparse matrix manipulation algorithm. We decided, as part of the research for this dissertation, to investigate the possibility of using MTBDDs directly in the solution of the CTMC.

Clarke et al [20] present algorithm for manipulating MTBDDs to carry out standard vector or matrix functions such as addition or vector by matrix multiplication. Furthermore, Siegle et al in [24] present some rules of thumb on using MTBDDs in a CTMC solution. The Jacobi method is an ideal candidate for a simple iterative algorithm that could be adapted for use with MTBDDs, and the theory for this was presented in [24]. Knottenbelt demonstrates in [33] that the CGS algorithm can be more efficient than the Jacobi method. Since CGS does not involve any matrix inversions, we considered it also as a candidate for implementation with MTBDDs, hoping to better the convergence times of the Jacobi method.

The results of our experiments were disappointing, but as Siegle points out 5 rules of thumb in his discussion of MTBDD-based CTMC solvers, we believe we can contribute to the discussion with our experiences. The bottleneck in the iterative solution process is the vector by matrix multiplication algorithm. This dually-recursive algorithm, presented in pseudocode in Figure 16 on page 36, is memory and time inefficient. The reason for the long execution time is obvious: at every recursive step of the algorithm three componentwise vector-additions must be carried out. Furthermore, unlike other BDD manipulation algorithms, where the recently computed vertex is stored in a hash-table, in the case of the vector by matrix multiplication the entire MTBDD for the sub-vector is stored. This leads to huge memory requirements in the size of the hash-table.

We worked to optimise this algorithm by first optimising the vector addition algorithm. This we did by minimising the size of the tables the addition algorithm requires for its computation. Further, we limited the number of intermediate result MTBDDs could be stored in the vector-by-matrix hash table. This table is used for efficiency, but its high memory usage makes it impractical to store all the intermediate results. In fact, we noticed improved efficiency by decreasing the size of this table.

In reference to Siegle’s rules of thumb 2 [24], which states:

Compressing \( m \) states into \( \lceil \log_2 m \rceil \) bits does not always result in the most space efficient encoding. It is always more promising to exploit the structure of the high-level
our experiences confirm this. We wrote a translator that converts the MTBDD-based rate matrix derived from the state space exploration to an $m \times m$ MTBDD. We found that the depth of the MTBDD certainly decreases substantially. For example, consider the parsys-5 model which had an MTBDD depth of 42. Since this model has only 1,122 states, we translated it to an MTBDD with depth 22. Interestingly, the MTBDD for the rate matrix contained 1,269 nodes while the translated MTBDD contained 3,636 nodes. So, in terms of memory and time efficiency when the vector by matrix algorithm is applied, rule of thumb 2 holds, in our experience.

We also concur with rule of thumb 5 which states that repetitive sub-blocks should be encoded "close" to one another. We have already seen the advantages of interleaving the similar elements of separate sub-nets in our discussion on variable ordering earlier in this chapter.

We have implemented the CGS and Jacobi methods as a proof of concept. Krylov subspace techniques are well-suited to solution via MTBDDs, and are an alternative to the CGS method. Unfortunately, an efficient alternative is required to the vector by matrix algorithm if either of these techniques are to be used. We managed to optimise our algorithm so that they could perform one vector by matrix operation in approximately 10 seconds for a state space size of 1,122 states, which is an improvement over our initial implementation. This is, however, far too slow to be viable in the solution process which often requires many hundreds of iterations, and the CGS algorithm requires 2 vector by matrix multiplications per step. A standard sparse matrix package performed one vector by matrix operation in 0.003 seconds.

Another important consideration is the size of the solution vector. Particularly in electronic circuit applications, the solution vector does not contain many distinct elements, which is ideal for an MTBDD-based application. However, for a suitable level of accuracy in timed transition systems, such as GSPNs, the solution vector is usually populated by non-repeating elements. We experimented with a technique to collapse terminal vertex entries that were the same to within some range. However, the accuracy of the solution suffered greatly as a result of this.

With particular reference to the solution techniques we have implemented we noted that, while the method of Jacobi generally converges smoothly using a high number of iterations, the CGS method exhibits erratic convergence behaviour generally with a smaller number of iterations. In our experience, both methods are sensitive to the starting vector – the initial guess at the probability distribution. We mentioned two types of starting vector in Chapter 6. The first contains the uniform
CHAPTER 7. EXPERIMENTAL RESULTS

distribution, which is an MTBDD consisting a single terminal node with entry $1/n$, where $n$ is the number of states. The second starting vector simply has one entry of 1, while the others are all zero. This is represented by an MTBDD with one path from root to an MTBDD terminal node with entry 1.

Both of these techniques were suitable for the method of Jacobi and are easily created. Neither, however, were suitable for use with the CGS algorithm. The CGS algorithm is highly sensitive to the starting vector which should not start off too close to the final answer, nor contain many zeros. Through testing with DNAmaca, we found that the ideal starting vector contains a random selection of entries. MTBDDs are not well suited to a random starting vector, which is typically a very large MTBDD with no terminal entries the same. However, in testing with this technique with the CGS algorithm we found that it is superior to the method of Jacobi.

Despite these drawbacks, if the vector by matrix algorithm can be improved, MTBDDs will be a viable alternative. However, this does not nullify the usefulness of MTBDDs in the state space exploration. MTBDDs are highly suited to the storage of the rate matrix during the exploration, and the MTBDD representation can be converted to a classical sparse matrix representation efficient CTMC solution. In this way we gain the memory and time efficiency of MTBDDs during state space exploration, and the time efficiency of standard sparse matrix methods during CTMC solution.

Using the GSPN of the FMS model [21] we compared classical probabilistic hashing-based techniques, using DNAmaca [32], to our own symbolic methods on a single workstation. DNAmaca required 250MB of memory for the state space of 4,459,455 states and 634MB of memory for the rate matrix of 38,533,968 transitions. DNAmaca reported a confidence level of 86% in the coverage of the state space. The classical exploration took in excess of 14 hours. Our own tool-set found the state space exhaustively using 4.8MB of memory for the state space and 0.8MB of memory for the final MTBDD of the rate matrix. The exploration was completed in 6.8 hours, less than half the time with a fraction of the memory requirements. Solution of the CTMC underlying this model was not feasible using MTBDDs. However, we have said that it is possible to convert from an MTBDD to a classical sparse matrix structure, and, using a disk-based solver, Knottenbelt [33] reports a solution time of 8.6 hours for this matrix.
Chapter 8

Conclusion

Without techniques to analyse concurrent communicating systems, we would be unable to ensure system requirements compliance on even basic levels. We need to be able to address and quantify such fundamental issues as the safety of systems like air-traffic controllers, or, as is prevalent today, railway system managers.

During the Gulf War, in 1991, cumulative errors in the clock of a system designed to track and destroy SCUD missiles caused United States defense software to be off by 0.34s, ultimately resulting in the deaths of 28 US soldiers.

Intel Corporation released a chip in 1995 containing the famous FDIV error. The error propagated from a simple programming transcription error. This simple, easily detected error cost Intel half a billion US dollars.

In June of 1996 the Ariane-5, a satellite-lifting rocket worth over 500 million US dollars, crashed into a marsh in French Guiana. An investigation blamed the error on inadequate software testing. In fact, the crash resulted from an unhandled floating-point exception that caused a memory dump into the area storing the guidance information for the rockets’ motors, and the disaster ensued.

These kinds of errors are not isolated and range from benign, to costly, to life-threatening. Clearly, in a world where the complexity of communicating systems is increasing so rapidly, there is a real need for efficient, cost-effective techniques for carrying out formal functional and performance analysis. We are chiefly interested in performance analysis.

One method for carrying out performance analysis begins by building a GSPN model of the real-world system. All the possible states of the GSPN are then enumerated, together with the transition
rates between each of the states. This is interpreted as a Markov chain from which it is possible to derive a set of steady-state equations whose solution yields the steady-state probability distribution of the chain. Once the low-level steady-state probabilities are known performance metrics of the original GSPN model can be derived. Completing the analysis life-cycle, analysts can use the model performance metrics to evaluate the efficiency or load on real-world system components.

This dissertation has presented techniques primarily for conserving memory during state space exploration and, secondly, for the solution of the resultant CTMC of the GSPN. In this chapter we present a discussion on our experiences, and comments on future work that may be done in this area.

8.1 State Space Exploration Based on BDDs

Immediately noticeable from the experimental results in Chapter 7 is that BDDs give a great memory saving over conventional state space exploration techniques such as AVL trees or hashing methods. Furthermore, where conventional methods may use probabilistic state searches, BDDs allow one to exhaustively search the state space without running out of volatile memory, and not requiring disk access.

We noted that when the structure of the model becomes less regular or repetitive, there is a definite decrease in the memory efficiency of BDD-based techniques. This confirms the comments of other published work in the area of symbolic techniques. However, even from non-regular models, it was possible to, relatively simply, create an automated technique to pick a good variable ordering.

The main drawback of using symbolic techniques is that, for GSPN models, they are much slower than classical techniques – particularly when the subject of interest is performance analysis. In this area, more efficient BDD manipulation algorithms are needed, and the GSPN state encoding system should be improved as the height of the storage BDD dramatically influences the speed of the state space exploration and CTMC solution process.

8.1.1 CTMC Solutions Based on BDDs

We implemented a proof of concept using MTBDDs in the solution of CTMCs via the Jacobi iterative method and the CGS algorithm, a Krylov subspace technique. We found that although MTBDDs are highly memory conservative, the algorithms to manipulate them are unusably slow.
CHAPTER 8. CONCLUSION

This is in accordance with the trend that the more memory one attempts to conserve, the more time consuming manipulation algorithms become. We have suggested a hybrid technique whereby the state space exploration is carried out using BDDs for the state descriptors and MTBDDs for the rate matrix. This gives good memory savings during state space exploration. Once the exploration is complete, the MTBDD rate matrix can be converted to any sparse matrix format, for a standard iterative solution.

8.2 Future Work

Our brief investigation into distributing the state space exploration over several networked machines gave disappointing results, but this area is certainly an interesting research direction. There are several important considerations to make. Firstly, a better state partition function would be required. We used one machine to control the state space exploration, which created a bottleneck for communication. This protocol could be improved upon.

An interesting exercise would be in combining probabilistic with symbolic techniques. We suggest this because probabilistic techniques make use of a hash function to compress the size of the state descriptor. In combination with our invariant encoding mechanism, we could further decrease the height of the BDD used for state storage, and speed up the state space exploration process.
Appendix A

GSPN Models

This section contains the set of models used for experimentation and testing in this dissertation, along with a brief description of each.

A.1 The Dining Philosophers Problem

The Dining Philosophers problem is a classical example of a resource-sharing system. There are $n$ philosophers sitting at a table with an infinite supply of spaghetti. There are also $n$ forks at the
table. Philosophers can be either thinking, or eating, or waiting to eat. To eat, a philosopher requires two forks, one from her left, and one from her right, as shown in Figure 51(a). Obviously, to eat, a philosopher requires both the left and right fork, preventing her neighbours from eating. If each philosopher decides to pick up their left (right) fork at the same time, then there will be a system deadlock as each philosopher waits for her right (left) neighbour to release their left (right) fork.

We added a simple arbitrator, a “waiter”, to solve the problem by controlling access to the dining table. Philosophers can now think at the bar, and when they are hungry move to the dining table to eat. The waiter ensures that, if there are a total of \( n \) philosophers, then no more than \( n - 1 \) may sit at the dining table at the same time. This eliminates the possibility of deadlock. This does not ensure fairness, however.

The GSPN of the dining philosophers form a group of identical subnets. The repeating subnet is shown in Figure 51(b). The “waiter” is modeled by a single place. By connecting more philosopher subnets together, we can scale the model in terms of the number of states it produces.

### A.2 The Courier Telecommunications Protocol

The Courier telecommunications protocol, the GSPN model for which is shown in Figure 52, operates at the application, session, and transport layers of the ISO network protocol stack. Messages flow from sender to receiver, and are passed between layers by “Courier” tasks which act as active data buffers. The transport layer fragments the data, modeled as two paths in Figure 52 from place \( p10 \) to \( p31 \). The first path (via \( t7 \)) carries all but the last data fragment to \( p31 \) and each fragment is acknowledged but no data is sent up to the application layer. The other path (via \( t8 \)) sends the last data fragment, which is acknowledged, and the data packet is sent up to the application layer. The courier protocol uses a sliding window. The size of the window, \( n \), is determined by the number of tokens on place \( NNN \), and the transport space, \( m \), is set by the number of tokens on the place \( MMM \).

By varying \( n \) and \( m \), we can generate very large state spaces. We took this model from [33] and made use of it because there is no easily discernible repeating substructure. We show how our techniques may also be applied to non-structured models.
A.3 The Kanban Model

The GSPN for the Kanban model, shown in Figure 53, is taken from [41].

The Kanban model is popular in the literature as it provides a regularly structured model with the ability to generate very large state spaces. The number of states is scaled by increasing the number of tokens of each of the places $p_2$, $p_6$, $p_{10}$, and $p_{14}$. We use the value $n$ to refer to the number of tokens on these places.

A Kanban is a card that contains all the activities to be performed on a product at each stage along the pipeline to product completion, along with what is still required to be done at later stages.
The cards are used to control work-in-progress, production, and inventory flow. A Kanban system consists of a set of such cards, one for each part to be manufactured, that travels between preceding and subsequent processes. Kanban systems are used to reduce costs in high-volume production lines by allowing organisations to reduce production lead-time and subsequently reduce the amount of inventory required.

A.4 The Flexible Manufacturing System

The GSPN for a flexible manufacturing system (FMS), shown in Figure 54, is taken from [21], and was also used in [33]. We use it as an example of a GSPN with some repetitive sub-structure.

Figure 54 contains a GSPN which models an assembly line consisting of three different machines (transitions t12, t8, and t3). The machines assemble four types of parts (places p0, p10, p15, and p9). Initially, there are $k$ unprocessed parts in places p0, p10, and p15. The parts in p9 are assembled from the parts in p0 and p10. By scaling $k$ we can produce very large state spaces.

There are three machines for processing p0. There is one machine which can process p10, or p15 if no p10 are present. The machine t3 assembles parts p0 and p10 to create p9. When any parts are finished they are shipped off, to be replaced exactly in quantity by fresh unprocessed parts.

A.5 The parsys Model

The GSPN for the parsys model, shown in Figure 55, was used by Caselli et al in [54] to show the effectiveness of their parallel state space generation techniques. They used it to show the effect of immediate transitions on the workload distribution amongst distributed processors. By increasing
the number of tokens on place \( p0 \), we can generate large state spaces. We use this model to test our techniques as it has no readily exploitable structure.
Bibliography


