IMPLEMENTATION OF SYMBOLIC MODEL CHECKING FOR PROBABILISTIC SYSTEMS

by

DAVID ANTHONY PARKER

A thesis submitted to the Faculty of Science
of the University of Birmingham
for the degree of
Doctor of Philosophy

School of Computer Science
Faculty of Science
University of Birmingham
2002

Abstract

In this thesis, we present efficient implementation techniques for *probabilistic model checking*, a method which can be used to analyse probabilistic systems such as randomised distributed algorithms, fault-tolerant processes and communication networks. A probabilistic model checker inputs a probabilistic model and a specification, such as "the message will be delivered with probability 1", "the probability of shutdown occurring is at most 0.02" or "the probability of a leader being elected within 5 rounds is at least 0.98", and can automatically verify if the specification is true in the model.

Motivated by the success of *symbolic* approaches to non-probabilistic model checking, which are based on a data structure called binary decision diagrams (BDDs), we present an extension to the probabilistic case, using multi-terminal binary decision diagrams (MTBDDs). We demonstrate that MTBDDs can be used to perform probabilistic analysis of large, structured models with more than 7.5 billion states, way out of the reach of conventional, *explicit* techniques, based on sparse matrices. We also propose a novel, hybrid approach, combining features of both symbolic and explicit implementations and show, using results from a wide range of case studies, that this technique can almost match the speed of sparse matrix based implementations, but uses significantly less memory. This increases, by approximately an order of magnitude, the size of model which can be handled on a typical workstation.

Acknowledgements

First and foremost, my thanks go to my supervisor, Marta Kwiatkowska, without whose unerring support, guidance and enthusiasm, this work would never have been completed. I am also indebted to Gethin Norman for his tireless help and advice which has been invaluable. Thanks must also go to Markus Siegle, Mark Ryan and Achim Jung for their helpful comments and to EPSRC MathFIT and QinetiQ for funding my studies. Finally, I would like to thank Nikki and my parents for their patience and support throughout.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The proliferation of computerised systems in all aspects of our lives places an increasing importance on the need for them to function correctly. The presence of such systems in safety-critical applications, coupled with their ever increasing complexity, means that the conventional method of checking that a system behaves as intended, testing it on a representative set of scenarios, is often inadequate.

A branch of computer science which aims to resolve this problem is *formal verification*. Of particular interest are formal verification techniques which can be automated. A prime example of this is *model checking*. To verify the correctness of some real-life system using model checking, one first constructs a model to represent it. This model identifies the set of all possible states that the system can be in and the transitions which can occur between them. One also specifies desirable or required properties of the real-life system to be verified. Examples are: "process 1 and process 2 are never in their critical sections simultaneously", "it is always possible to restart the system", and "whenever a request occurs, an acknowledgement will eventually be received".

A *model checker* then automatically checks, via a systematic analysis of the model, whether or not each of the specified properties is satisfied. The popularity and, more importantly, the uptake by industry of model checkers such as SMV, SPIN and FDR provide an indication of the success that these techniques have enjoyed.

One caveat of the model checking paradigm is that the results of verification are only as dependable as the accuracy of the model which has been constructed for analysis. For this reason, as verification techniques have become more efficient and more prevalent, many people have sought to extend the range of models and specification formalisms to which model checking can be applied.

A good example is the field of *probabilistic model checking*. The behaviour of many real-life processes is inherently stochastic. Examples include: probabilistic protocols or

randomised distributed algorithms, which can execute certain actions randomly; fault-tolerant systems, which are designed to operate knowing that some of its components are unreliable; and communication networks, whose varying and unpredictable patterns in usage can only be accurately modelled in a probabilistic fashion.

In probabilistic model checking, the model constructed for analysis is probabilistic. This is usually achieved by labelling transitions between states with information about the likelihood that they will occur. The specifications are also suitably modified. We now verify properties such as: "the message will be delivered with probability 1", "the probability of shutdown occurring is at most 0.02", and "the probability of a leader being elected within 5 rounds is at least 0.98".

As in the non-probabilistic case, algorithms for probabilistic model checking can be used to automatically determine whether or not a model satisfies its specification. While non-probabilistic model checking usually reduces to algorithms based on graph analysis, the probabilistic case typically also requires numerical computation. Despite the fact that the appropriate algorithms have been known since the 1980s, there has been a distinct lack of tools developed to support the technology. This was one of the primary motivations for our work. Over the past four years, we have developed a probabilistic model checker called PRISM. This thesis describes research carried out into developing the data structures and associated techniques required for the efficient implementation of such a tool.

The principal challenge when developing any kind of model checker is to overcome the so called *state space explosion problem*. Also known as 'largeness' or 'the curse of dimensionality', this states that there is a tendency for the number of states in a model to grow exponentially as the size of the system being represented is increased linearly. In practice, this means that models of even the most trivial real-life systems can contain many millions of states.

The reason that non-probabilistic model checking has been so successful in the real world is that an enormous amount of work has been put into developing efficient implementation techniques for it. One of the most successful approaches taken is known as *symbolic model checking*. This relies on a data structure called binary decision diagrams (BDDs). Because the models used in verification are inevitably described in some high-level formalism, they usually contain a degree of structure and regularity. BDDs can be used to exploit this regularity, resulting in an extremely compact representation for very large models. Efficient manipulation algorithms developed for the data structure then allow model checking to be performed on these models.

In this thesis, we consider the application of symbolic techniques to probabilistic model checking. To do so, we use a natural extension of BDDs, multi-terminal binary decision diagrams (MTBDDs). We have developed a complete implementation of probabilistic

model checking based on the MTBDD data structure. Using a wide range of case studies, we present an extensive analysis of the efficiency of this *symbolic* implementation and compare its performance to that of more conventional, *explicit* approaches based on sparse matrices. We show that MTBDDs can be used for probabilistic model checking of large, structured models with more than 7.5 billion states. Here, analysis with equivalent, explicit techniques is simply not feasible. We also demonstrate, however, that in many other cases the symbolic implementation has significantly higher time and memory requirements than the explicit version.

To combat this, we propose a novel, *hybrid* approach combining features of both the symbolic and explicit implementations. For typical examples, this technique is many times faster than using MTBDDs alone and can almost match the speed of the sparse matrix based implementation. Furthermore, by providing a significant reduction in the amount of memory required for verification, we increase the size of model which can be handled by approximately an order of magnitude.

Layout of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 gives a technical review of related work, identifying what research has already been done in this area and how the work in this thesis contributes to it. Chapter 3 introduces the relevant background material. This includes details of the probabilistic models, specification formalisms and model checking algorithms we consider, a description of our tool PRISM, and an introduction to the two principal data structures discussed in this thesis: sparse matrices and multi-terminal binary decision diagrams (MTBDDs). Chapters 4 to 6 contain the main contribution of the thesis. In Chapter 4, we consider the problem of developing an efficient representation for probabilistic models using MTBDDs. In Chapter 5, we use this representation to present a full, symbolic implementation of probabilistic model checking. We give results for a range of case studies, making comparisons with equivalent, explicit implementations based on sparse matrices and provide a detailed analysis of their respective performances. In Chapter 6, we present a novel, hybrid approach which combines symbolic and explicit techniques in order to improve the efficiency of the pure MTBDD implementation. We conclude in Chapter 7 by evaluating the success of the work and highlighting possible areas for future research.

Other Publications

Some of the work in this thesis has previously been published in jointly authored papers. In [dAKN+00], a symbolic implementation of model checking for MDPs was presented. This included an early version of the translation of the PRISM language into MTBDDs, as described in Section 4.3 and Appendix C. This translation was worked on by the author of this thesis along with Marta Kwiatkowska and Gethin Norman. The MTBDD implementation used to provide experimental results for [dAKN+00] was developed by the author. The latter is also true of [KKNP01], which presented results for MTBDD-based model checking of CTMCs. In addition, [KKNP01] included an improvement to the CSL model checking algorithm. This was developed jointly by the author and Gethin Norman.

The data structures and associated algorithms making up the hybrid approach of Chapter 6 are entirely the work of the author. A preliminary version of this was published in [KNP02b]. The author was also solely responsible for the development of the PRISM model checker, into which all the implementations described in this thesis have been built. A tool paper describing PRISM appeared in [KNP02a].

The majority of the modelling work to develop case studies for PRISM was done by Gethin Norman. These are used throughout this thesis to present experimental results. Finally, a joint journal paper with Holger Hermanns, Marta Kwiatkowska, Gethin Norman and Markus Siegle, summarising experiences with MTBDD-based analysis of probabilistic models, is to be published in [HKN+02]. This paper reports on results obtained independently with PRISM and IM-CAT, a tool developed at Erlangen.

Chapter 2

Review of Related Work

In this chapter we review work from the literature which is closely related to the topic of this thesis: the implementation of symbolic model checking techniques for probabilistic systems. We begin, in Section 2.1, by examining the development of probabilistic model checking itself. Next, Section 2.2 reviews BDD-based, symbolic model checking techniques, as applied in the non-probabilistic setting. Following this, in Section 2.3, we focus on MTBDDs, which are a natural extension of BDDs, suitable for extending symbolic model checking to the probabilistic case. We consider the application of MTBDDs to matrix representation and manipulation, the solution of linear equation systems and the analysis of probabilistic models. We also review some other approaches to the implementation of probabilistic model analysis. These include alternative extensions of BDDs and methods based on the Kronecker algebra. Finally, in Section 2.5, we summarise existing implementations of probabilistic model checking.

2.1 Probabilistic Model Checking

Temporal logics provide a way of reasoning about behaviour which varies over time. Their application in computer science was pioneered by Pnueli [Pnu77] who argued that existing techniques for formal verification were not adequate for concurrent, reactive systems, i.e. those comprising parallel, interacting components and which do not necessarily terminate.

Model checking, proposed independently by Clarke and Emerson [CE81] and by Queille and Sifakis [QS82], is a technique for automatically determining whether or not a finite-state model of a system satisfies a property specified in temporal logic. It does so via an exhaustive search of its state space, usually by executing some form of graph analysis over the model's transition relation. In the case where a property is not satisfied, a model checker also produces a counterexample: an execution of the system which illustrates

that the property does not hold. A model checking algorithm for the temporal logic CTL (Computation Tree Logic) was presented in [CES86].

Temporal logic based formal verification has been successfully extended to the realm of probabilistic models. Initial work in this area focused on the verification of qualitative formulas, i.e. those stating that some property holds with probability 0 or 1. The case for purely probabilistic systems, which we refer to in this thesis as discrete-time Markov chains (DTMCs), was considered in [LS82, HS84, CY88]. At the heart of all these approaches is the fact that, for qualitative formulas, the actual probabilities in the model are irrelevant: verification can be performed based only on an analysis of the underlying graph.

In a similar vein, qualitative property verification was considered for models which exhibit both probabilistic and nondeterministic behaviour, i.e. models such as Markov decision processes (MDPs). These included [HSP83, Pnu83, Var85, PZ86, CY88]. One of the main differences here is that a notion of fairness must be considered.

More difficult is the verification of quantitative specifications. For these, one must compute the exact probability that a given property is satisfied. The case for DTMCs was considered by Courcoubetis and Yannakakis [CY88, CY95], Hansson and Jonsson [HJ94], and Aziz et al. [ASB+95]. The latter two papers introduced the temporal logics PCTL and pCTL respectively. These are essentially equivalent, extending the non-probabilistic logic CTL with a probabilistic operator. This operator takes a probability bound as a parameter and allows one to write formulas such as $\mathcal{P}_{>p}[\psi]$, stating that the probability of event ψ occurring is greater than the bound p. It is shown that model checking for pCTL/PCTL reduces to solving a linear equation system. Aziz et al. [ASB+95] also introduced pCTL*, a combination of pCTL and the linear time logic LTL. This can be model checked using the techniques in [CY88, CY95].

The verification of quantitative properties for MDPs was considered by Courcoubetis and Yannakakis [CY90] and Bianco and de Alfaro [BdA95]. Model checking for pCTL is shown to reduce to the solution of a linear optimisation problem. Bianco and de Alfaro [BdA95] also presented an algorithm for pCTL*. As in the case for qualitative properties, it is often necessary to consider fairness in order to perform verification. In [BK98], Baier and Kwiatkowska showed how the algorithms for model checking quantitative properties can be extended to incorporate a notion of fairness based on that of Vardi [Var85]. A further improvement to their algorithm was given by Baier in [Bai98].

The model checking paradigm has also been extended to continuous-time Markov chains (CTMCs). Traditionally, these models have been analysed by standard performance analysis techniques, where steady-state (long run) and transient probabilities are computed and then translated into more meaningful, application-specific measures such as throughput and mean waiting time. In [ASSB96], Aziz et al. proposed the logic CSL

(Continuous Stochastic Logic) to provide a means of formally specifying properties of CTMCs. CSL can be seen as an extension of PCTL. It introduces a time-bounded until operator which can be used to reason about real-time behaviour of CTMCs. Model checking for the logic is shown to be decidable for rational time-bounds.

Baier et al. [BKH99] then extended CSL by adding an operator to reason above the steady-state behaviour of CTMCs and presented the first model checking algorithm for the logic. Steady-state probabilities are computed in the usual way by solving a linear equation system. For the time-bounded until operator, they proposed an algorithm based on an iterative method for the approximate solution of a Volterra integral equation system.

In [BHHK00a], Baier et al. presented an alternative method of model checking the CSL time-bounded until operator using a conversion to transient analysis. This allows standard, efficient computation techniques to be used. The algorithm was improved further by Katoen et al. in [KKNP01], where a relatively simple reformulation of the required computation results in an $\mathcal{O}(N)$ improvement in time complexity, N being the number of states in the CTMC being model checked.

2.2 Symbolic Model Checking

At the same time that model checking techniques were being extended to handle more complex modelling and specification formalisms, a great deal of work was being done to improve the efficiency of the original, non-probabilistic approach. Model checking had shown itself to be successful on relatively small examples, but it quickly became apparent that, when applied to real-life examples, explicitly enumerating all the states of the model is impractical. The fundamental difficulty, often referred to as the state space explosion problem, is that the state space of models representing even the most trivial real-life systems can easily become huge.

One of the most well-known approaches for combating this problem, and the one which this thesis focuses on, is symbolic model checking. This refers to techniques based on a data structure called binary decision diagrams (BDDs). These are directed acyclic graphs which can be used to represent Boolean functions. BDDs were introduced by Lee [Lee59] and Akers [Ake78] but became popular following the work of Bryant [Bry86], who refined the data structure and developed a set of efficient algorithms for their manipulation.

In terms of model checking, the fundamental breakthrough was made by McMillan. He observed that transition relations, which were stored explicitly as adjacency lists in existing implementations, could be stored symbolically as BDDs. Because of the reduced storage scheme employed by the data structure, BDDs could be used to exploit high-level structure and regularity in the transition relations. Furthermore, the graph analysis based

model checking algorithms for CTL could be implemented using efficient BDD operations.

In [BCM+90, McM93], these ideas were applied to give a symbolic version of the CTL model checking algorithm from [CES86] and it was demonstrated that large, regular models with as many as 10^{20} states could be verified. Improvements presented in [BCL91] pushed the limit even higher to 10^{120} states. These techniques were implemented in what became the well-known SMV model checker [McM93]. Perhaps one of the most notable achievements of this technology was the successful verification of the cache coherence protocol from the IEEE Futurebus+ standard [CGH+93].

2.3 Multi-Terminal Binary Decision Diagrams

This thesis considers the extension of established symbolic model checking techniques to the probabilistic case. In addition to algorithms based on graph analysis, probabilistic model checking requires numerical computation to be performed. While graph analysis reduces to operations on a model's transition relation and sets of its states, numerical computation requires operations on real-valued matrices and vectors. For this reason, the most natural way to extend BDD-based symbolic model checking is to use multi-terminal binary decision diagrams (MTBDDs). MTBDDs extend BDDs by allowing them to represent functions which can take any value, not just 0 or 1.

2.3.1 Matrix Representation and Manipulation

The idea of MTBDDs was first presented by Clarke et al. [CMZ+93], where two different extensions of BDDs are considered for the purpose of representing integer matrices: arrays of BDDs and BDDs with more than two terminals (MTBDDs). Focusing on the latter, they give methods for implementing standard matrix operations, such as scalar multiplication, matrix addition and matrix multiplication. Furthermore, they use these techniques to implement the Walsh transform, a common component of digital circuit design, and give positive experimental results. The reason for the success of this implementation seems to stem from the fact that the matrices used, Walsh matrices, contain only two distinct values (1 and -1) and are extremely regular.

This work is extended in [CFM+93], which begins by observing that MTBDDs can represent matrices over any finite set, for example a subset of the reals as opposed to of the integers. It then proceeds to examine in more detail the effectiveness of such a representation. The worst case complexity for the size of an MTBDD is analysed and compared to that of explicit storage schemes such as sparse matrices. It is shown that, in this respect, MTBDDs are the optimal representation for very sparse matrices.

In [CFM+93], it is also shown how the recursive nature of MTBDDs leads to elegant algorithms for the implementation of matrix operations. In particular, an improved algorithm for multiplication of matrices and vectors is presented. Inversion of upper or lower triangular matrices and L/U decomposition is also considered. The worst-case complexity for these operations is analysed, but experimental results are notably absent. While complexity analysis gives valuable insight into the performance of the algorithms, previous work on BDDs has shown that, despite having exponential worst-case or even average-case complexity, many operations have still proven to perform well in practice. Hence, it would seem wise to apply the algorithms to some real examples.

MTBDDs were also studied independently by Bahar et al. in [BFG+93], under the guise of ADDs (algebraic decision diagrams). The choice of terminology reflects the fact that the authors considered the data structure to be applicable not only to arithmetic over real-valued matrices, but also to operations in alternative algebras. For example, they show how MTBDDs can be used to implement a min-plus algebra based solution to the shortest path problem for weighted, directed graphs. More relevantly, from our perspective, they also give a detailed analysis of the application of MTBDDs to both the multiplication of real matrices and the solution of linear equation systems.

The authors present a novel matrix multiplication algorithm and compare it to the existing two [CMZ+93, CFM+93] on a number of large, benchmark examples. They conclude that their algorithm is comparable with the one in [CFM+93] and that both typically outperform the one in [CMZ+93]. Of particular interest is their comparison with an explicit implementation of multiplication using sparse matrices. Although they initially claim that the MTBDD and sparse matrix approaches are comparable in terms of speed, a later version of the paper [BFG+97] concedes that MTBDDs are actually much slower.

Bahar et al. also investigate MTBDD-based solution methods for linear equation systems. An implementation of Gaussian elimination is presented and analysed. Again, MTBDDs are found to be significantly slower than the equivalent sparse matrix version. An important contribution of the paper is the identification of the reason for this behaviour. The pivoting operations required for Gaussian elimination rely on the manipulation of individual matrix elements, rows and columns. These tend to destroy any regularity in the MTBDD, increasing its size and making the operations on it slow to perform. With this in mind, the authors suggest that iterative methods based on matrix multiplication may prove to be a more fruitful source of investigation.

2.3.2 Analysis of Probabilistic Models

Given that probabilistic models are often represented as matrices and that their analysis often entails numerical computation based on these matrices, such as the solution of linear equation systems, it becomes possible to perform the analysis using MTBDDs. In this section, we review work which has been undertaken in this area. We classify the various approaches according to the type of probabilistic model involved, considering first discrete-time Markov chains (DTMCs), then Markov decision processes (MDPs) and then continuous-time Markov chains (CTMCs).

Discrete-Time Markov Chains (DTMCs)

In [HMPS94, HMPS96], Hachtel et al. apply MTBDDs to the computation of steady-state probabilities of DTMCs using two iterative methods, the Power method and a modified version of the Jacobi method. They test their implementation on a number of DTMCs generated from benchmark circuits and present experimental results. These results demonstrate that analysis can be performed on some extremely large models, including one DTMC with 10^{27} states. The authors observe that the principal problem is the exponential increase in the number of distinct values in the solution vector. To circumvent this, they employ a strategy which actually computes an approximation to the correct solution by rounding all probabilities below a certain threshold to zero.

In a similar fashion, Xie and Beerel [XB97] used MTBDDs to compute the steady-state probabilities of DTMCs using the Power method. The techniques are used to analyse the performance of several asynchronous timed systems. While the implementation appears successful, results are only given for relatively small examples, with at most 30,000 states.

DTMC analysis using MTBDDs was also proposed by Baier et al. in [BCHG$^+$97], where a symbolic algorithm for PCTL model checking is given. The key part of the algorithm constitutes solving a linear equation system. As above, it is proposed to accomplish this using iterative methods. The authors also give an MTBDD-based model checking algorithm for the richer logic PCTL*, which we do not consider in this thesis.

Although [BCHG$^+$97] does not present any experimental results, a subset of the symbolic PCTL model checking algorithm proposed was implemented and described in the thesis of Hartonas-Garmhausen [HG98]. This resulted in the prototype tool ProbVerus [HGCC99]. In [HG98], these techniques are applied to two case studies, one based on a railway signalling system, and the other on a flexible manufacturing system. Although this work only covered a subset of PCTL (bounded until formulas only), it demonstrated that PCTL model checking of DTMCs could be successfully implemented in MTBDDs and applied to real-life examples.

Markov Decision Processes (MDPs)

In [BC98], Baier and Clarke presented the algebraic mu-calculus, an extension to the mu-calculus and, via an MTBDD representation of its semantics, restated how PCTL model checking of DTMCs could be implemented with MTBDDs. Baier [Bai98] then showed how this approach can be adapted to work with MDPs.

Kwiatkowska et al. [KNPS99] and de Alfaro et al. [dAKN$^+$00] presented the first implementations of PCTL model checking for MDPs using MTBDDs. In [KNPS99], model checking was performed using the Simplex method. This was found to perform poorly for the same reasons that [BFG$^+$93] found Gaussian elimination to do so: the Simplex method relies upon operations on individual rows and columns of matrices. These are slow to perform and cause a blow-up in MTBDD size. In [dAKN$^+$00], de Alfaro et al. presents a much improved version, focusing on the structured construction of MTBDDs and a BDD-based precomputation algorithm. Results for a number of extremely large, regular examples are given. It is also observed, as done previously in [HMPS94, HMPS96], that it is the MTBDD representation for the solution vector which limits the applicability of the techniques to even larger models.

An alternative usage of MTBDDs for MDP analysis can be found in [HSAHB99]. Although the application domain is very different, decision-theoretic planning, the solution algorithm implemented bears a marked similarity to the one used in PCTL model checking. One key difference is that, in their MDPs, the choices in each state are labelled with actions. Hence, for example, instead of computing the maximum probability of a given event, they would attempt to determine the best action to select in each state (a policy) in order to maximise this probability. Because of this, they store an MDP as a set of MTBDDs, one for each possible action which can be taken. The paper presents positive results for a large, structured MDP example. The performance of the MTBDD implementation is shown to be superior to both an existing version based on decision trees (which differ from decision diagrams in that they do not merge identical nodes) and one which uses an explicit representation. Because of differences between the three implementations, a detailed comparison of times is not attempted.

We also mention the work of [DJJL01] which introduces a technique for model checking a subset of PCTL on MDPs using abstraction and refinement. They also describe a prototype implementation of their approach which uses MTBDDs. The emphasis of the paper, however, is on improving the efficiency of model checking via abstraction and refinement, not through symbolic implementation.

Continuous-Time Markov Chains (CTMCs)

We now move on to the analysis of CTMCs. In [HMKS99], Hermanns et al. extended the work of [HMPS94, HMPS96], applying MTBDDs to the computation of steady-state probabilities for CTMCs. The paper describes how MTBDDs can be used to represent CTMCs and then gives symbolic algorithms for computing the steady-state probability distribution using iterative methods. These include the Power, Jacobi and Gauss-Seidel methods, the latter being implemented for the first time in MTBDDs.

The key contribution of the paper, though, is an in-depth discussion of methods for obtaining a compact MTBDD model encoding, something that is notably absent from prior work. This issue is particularly important since the efficiency of BDD-based data structures is largely dependent on their size, which in turn is dramatically affected by issues such as the encoding scheme chosen and the ordering of MTBDD variables.

The results of the paper are presented as a series of heuristics. The key lesson is that MTBDDs should be used to exploit structure in the model being represented. Such structure usually derives from the original high-level description of the system. The case for several formalisms, including process algebras and queueing models, is described in the paper. Empirical results are limited to memory requirements for the various encoding schemes discussed; performance details for the implementation of the iterative methods are not given.

In [BKH99], Baier et al. showed how full model checking of CSL can be performed symbolically. The two main problems are the computation of probabilities for the logic's steady-state and time-bounded until operators. The former can be done as in [HMKS99]. For the latter, an adaptation of MTBDDs, called MTDDs (multi-terminal decision diagrams) and a corresponding set of operations for manipulation are introduced.

The algorithm presented for model checking the CSL time-bounded until operator is based on solving an integral equation system, which in turn reduces to approximate evaluation of integrals using quadrature formulas. The new data structure is specifically tailored to this process. Some nodes in the MTDD remain as binary decisions, but others represent a choice between N values, where N corresponds to the number of components in the computation of the approximation. From a performance point of view, it is important to note that the size of the data structure will be affected by the choice of N, which in turn determines the quality of the approximations. Unfortunately, no implementation or experimental results were presented so it is difficult to judge the effect that the increase in complexity of the data structure would have.

Following the improved CSL model checking algorithm of [BHHK00a], an entirely MTBDD-based version of CSL model checking was proposed in [KKNP01] and a compar-

ison of its efficiency made with an equivalent implementation using sparse matrices. Two examples are given where MTBDDs perform significantly better, handling large, structured CTMCs with as many as 33 million states. It is also noted, however, that because of the size of the MTBDD for the solution vector, this is not the case for other examples.

Summary

In summary, there have been numerous applications of MTBDDs to the representation and analysis of probabilistic models. It is encouraging to note that there is clearly potential for the data structure to provide compact storage of these models. It is also evident that this is dependent both on the existence of structure and regularity in the models and on the use of heuristics to obtain suitable MTBDD encodings. We will consider these issues in Chapter 4.

Another positive observation which can be made is that MTBDDs have proved to be well suited to the implementation of a range of probabilistic model checking algorithms and are seemingly a natural extension of BDDs in this respect. One criticism that can be directed at existing research, though, is that, in many cases, algorithms have been translated into MTBDDs, but no implementation has been attempted. In others, where the techniques have been realised, a thorough analysis of their efficiency has often been lacking; in particular, a comparison with more traditional alternatives such as sparse matrices. This is a situation we have tried to redress in our contributions to the literature [dAKN+00, KKNP01]. In Chapter 5, we describe our MTBDD-based implementation of probabilistic model checking for three types of models, DTMCs, MDPs and CTMCs, and provide a detailed analysis of its efficiency using experimental results from a wide range of case studies.

Another conclusion which can be drawn from the existing descriptions of MTBDD-based analysis of probabilistic models is that a symbolic representation for solution vectors can often be problematic. We will also observe this phenomenon in Chapter 5 and, in Chapter 6, will present a novel solution to the problem.

2.3.3 Other Applications

For completeness, we also describe some additional applications of MTBDDs that have appeared in the literature. One such example is the work of Ruf and Kropf [KR97, RK98], who use MTBDDs for discrete-time model checking. Systems are modelled as labelled transition systems, each transition being associated with an integer delay. Specifications are written in QCTL (quantitative or quantized CTL) or CCTL (clocked CTL), temporal logics which extend CTL. The timed models can be represented as MTBDDs and model

checking algorithms are then formulated in terms of the data structure. Results are presented to show that, on a set of examples, this approach outperforms two existing methods: firstly a conversion to a larger, but equivalent CTL model checking problem solved with SMV [McM93]; and secondly using an alternative tool for discrete-time model checking, KRONOS [DOTY96]. While this work bears obvious similarities with our application, the main difference is that both the model and algorithms in question deal only with integer values, not reals. Hence, the proliferation of distinct values in the computation is far less problematic.

A second interesting application is that of [BBO+02] which presents a technique for eliminating incoherence from probabilistic data. MTBDDs are used to store probability distributions which are generated and manipulated using genetic algorithms. Experimental results show that compact representations of probability distributions can be achieved, but running times were not necessarily reduced as a result.

2.4 Alternative Approaches

2.4.1 Other Data Structures

A number of other data structures related to MTBDDs have been introduced in the literature. In this section we review some of the more relevant ones. Firstly, we mention Siegle's DNBDDs (decision node BDDs) [Sie99], which were introduced for the representation of CTMCs. While MTBDDs extend BDDs by allowing multiple terminals, DNBDDs retain the original structure of BDDs, but add additional information to certain edges between nodes. Rates in the CTMC being represented are now encoded by these edge labels, rather than the values on the terminals, as is the case with MTBDDs. DNBDDs were used in [HS99] to implement bisimulation algorithms for CTMCs and found to be well suited to this purpose by extending existing BDD algorithms for the non-probabilistic case. It is not clear, however, what advantages the data structure would present in terms of implementing the numerical solution of CTMCs.

Another extension of the basic BDD is the EVBDD (edge-valued BDD) [LPV96, VPL96]. This introduces additive weights onto edges between nodes. EVBDDs have been successfully applied to the formal verification of arithmetic circuits and to integer linear programming. They are, however, not applicable in our setting since they are limited to the representation of integer-valued functions. FEVBDDs (factored edge-valued BDDs) [TP97, Taf94] extend EVBDDs by including two weights on edges, one additive and one multiplicative, affording a more compact representation than EVBDDs and allowing functions to take fractional values. The data structure was applied to matrix

manipulation and the solution of linear equation systems. The latter was performed with Gaussian elimination using arbitrary precision arithmetic. Due to the size of the models we aim to consider, this is not an option for us.

Lastly, we describe PDGs (probabilistic decision graphs), introduced by Bozga and Maler in [BM99]. These are another BDD-based representation for vectors or matrices of probabilities, the basic idea being that nodes are labelled with conditional probabilities which are multiplied together to determine the final probability. The intention is to allow a compact representation of vectors or matrices which are structured, but contain many distinct values and would hence result in a large MTBDD. While it does indeed seem possible to achieve more compact storage in some cases, the experimental results presented indicate that, as computations progress, the size of the data structure often increases unmanageably. The proposed solution is to 'discretise' the probabilities, merging similar values together. The implications of this on numerical accuracy are not considered. At the end of Chapter 5, we observe that applying similar ideas in our case proved to be fruitless. In addition, the increased complexity of algorithms for manipulation of PDGs slows the implementation speed considerably.

Similar findings were made by Buchholz and Kemper in [BK01], where they adapt PDGs for use alongside Kronecker-based techniques (described in the next section). The PDG data structure is modified for this purpose, allowing more than two edges from each node, and new algorithms for manipulation are presented. Unfortunately, it seems that the performance, in terms of speed, is still unsatisfactory.

2.4.2 Kronecker-Based Approaches

We also consider an area of research which uses the Kronecker algebra to derive efficient methods for the analysis of probabilistic models. In common with the symbolic model checking approach considered in this thesis, these Kronecker-based techniques rely on exploitation of structure in probabilistic models to generate a compact representation and hence extend the range of models which can be analysed. For this reason, they are often referred to as structured analysis approaches. They can be seen as closely related to our hybrid approach in Chapter 6 since they use a structured representation for matrices but store vectors explicitly using arrays.

The basic idea is that the transition matrix of a CTMC can be represented as a Kronecker (tensor) algebraic expression of smaller matrices, corresponding to components of the system being modelled. The power behind the Kronecker approach is that it is only necessary to store these small, component matrices and the structure of the expression which combines them. Methods for analysing the CTMC such as steady-state probability

computation, which reduces to the iterative solution of a linear equation system, can be applied to this representation directly.

The Kronecker approach was first conceived by Plateau in [Pla85] and applied to CTMCs modelled in a formalism called stochastic automata networks. It was later extended by Donatelli [Don94] to certain classes of stochastic Petri nets, a commonly used formalism in performance analysis.

Further work [Kem96, CT96, BCDK97] removed some of the restrictions on the type of model to which the techniques could be applied and addressed a number of implementation issues. For example, early approaches such as [Pla85] operated over the product state space of the components, which may be considerably larger than the subset which is actually reachable. This means that larger vectors have to be stored and the time complexity of numerical solution is increased because spurious matrix entries have to be detected and ignored. Schemes such as binary search over ordered sets and multi-level data structures were used to allow solution using the reachable state space only. Another improvement was the development of methods whereby the relatively slowly converging Power method could be replaced by speedier alternatives such as Gauss-Seidel. The overall effect is that the Kronecker-based approaches can increase by approximately an order of magnitude the size of model which can be handled whilst maintaining solution speed comparable with sparse matrix implementations.

Of particular interest to us is recent work by Ciardo and Miner, presented in [CM99, Min00] and integrated within the tool SMART [CM96], into developing efficient data structures to perform Kronecker-based solution of CTMCs. They introduce a data structure called matrix diagrams for this purpose. This complements existing work by the authors [CM97] which considers structured approaches for computing and storing the reachable state space of stochastic Petri nets. The latter uses multi-valued decision diagrams (MDDs), a generalisation of BDDs. Matrix diagrams store the small Kronecker matrices in a BDD-like tree data structure. Ciardo and Miner developed efficient algorithms for extracting matrix columns from the data structure, which can be used to implement the Gauss-Seidel method for numerical solution. In [Min01], Miner presents an extension called 'canonical' matrix diagrams, which differ from conventional matrix diagrams by giving a canonical representation of matrices, rather than being tied to a particular Kronecker-algebraic expression. From the point of view of numerical computation, the data structure is identical. We will give a more technical discussion of the similarities and differences between our work and the Kronecker-based techniques after describing our approach in Chapter 6.

2.5 Tools and Implementations

As stated previously, one of the primary motivations for the work in this thesis was the limited number of available tools for probabilistic model checking. We conclude this chapter by reviewing the implementations which have been produced.

We are aware of two prototype probabilistic model checkers which were developed before the work in this thesis commenced. The first is TPWB (Time and Probability Workbench) [Fre94]. This implements the algorithms presented in [Han94] for model checking the logic TPCTL over systems described in the process algebra TPCCS, which allows both probability and discrete-time to be modelled. This incorporates model checking of PCTL over DTMCs, as described in [HJ94], which we consider.

The second is ProbVerus [HGCC99], an extension of the tool Verus [Cam96]. This supports model checking of DTMCs using a subset of PCTL (until formulas are restricted to the bounded variant) and was developed to accompany the work presented in [HG98]. Of particular interest from our point of view is the fact that, in contrast to TPWB, it was implemented using MTBDDs.

More recently, [HKMKS00] presented the tool $\mathsf{E} \vdash \mathsf{MC}^2$ (Erlangen-Twente Markov Chain Checker). This provides model checking of both CSL over CTMCs and PCTL over DTMCs. $\mathsf{E} \vdash \mathsf{MC}^2$ is not a symbolic model checker: it uses explicit data structures such as sparse matrices. The transition matrices for the DTMCs or CTMCs to be analysed are also specified explicitly: the user provides a list of all the states and transitions which make up the model. The format in which $\mathsf{E} \vdash \mathsf{MC}^2$ accepts this information allows it to be used in conjunction with other tools, such as TIPPtool [HHK+98], which constructs a CTMC from a process algebra description. TIPPtool, along with numerous other applications such as MARCA [Ste94], SMART [CM96], PEPA [GH94] and TwoTowers [BCSS98], supports conventional steady-state and transient analysis of CTMCs, but not probabilistic temporal logic model checking.

Our tool, PRISM [KNP02a], supports model checking of three types of probabilistic model: DTMCs, CTMCs and MDPs. As described in the previous paragraphs, there is a degree of overlap with other tools in terms of support for DTMC and CTMC analysis. As far as we are aware, though, PRISM is the only fully-fledged tool to provide model checking of MDPs. The only comparable implementation in this respect is that presented in [DJJL01] which implements the abstraction and refinement techniques, applicable to a subset of PCTL model checking, introduced in the paper. More information about PRISM can be found in Appendix A.

Chapter 3

Background Material

In this chapter, we introduce the necessary background material for this thesis. The first three sections cover the theory behind probabilistic model checking. Sections 3.1 and 3.2 introduce the relevant models and temporal logics, respectively, and Section 3.3 gives the corresponding model checking algorithms. In Section 3.4, we show how probabilistic models can be specified in practice using the PRISM language. The final three sections cover some of the practical issues which we need to consider for our implementation of probabilistic model checking. Section 3.5 reviews iterative methods for solving systems of linear equations. Sections 3.6 and 3.7 introduce the two main data structures used in this thesis: sparse matrices and multi-terminal binary decision diagrams (MTBDDs).

3.1 Probabilistic Models

Traditional model checking involves verifying properties of labelled state transition systems. In the context of probabilistic model checking, however, we use models which also incorporate information about the likelihood of transitions between states occurring. In this thesis, we consider three different types of probabilistic model: discrete-time Markov chains, Markov decision processes and continuous-time Markov chains.

3.1.1 Discrete-Time Markov Chains

The simplest of the models we consider are *discrete-time Markov chains* (DTMCs), which simply define the probability of making a transition from one state to another. They can be used to model either a single probabilistic system or several such systems composed in a synchronous fashion. We define a DTMC as a tuple $(S, \overline{s}, \mathbf{P}, L)$ where:

- S is a finite set of *states*

- $\overline{s} \in S$ is the *initial state*
- $\mathbf{P} : S \times S \to [0,1]$ is the *transition probability matrix*
- $L : S \to 2^{AP}$ is the *labelling function*.

An element $\mathbf{P}(s,s')$ of the transition probability matrix gives the probability of making a transition from state s to state s'. We require that $\sum_{s' \in S} \mathbf{P}(s,s') = 1$ for all states $s \in S$. Terminating states are modelled by adding a self-loop (a single transition going back to the same state with probability 1). The labelling function L maps states to sets of atomic propositions from a set AP. We use these atomic propositions to label states with properties of interest.

An execution of the system which is being modelled is represented by a *path* through the DTMC. A path ω is a non-empty sequence of states $s_0 s_1 s_2 \ldots$ where $s_i \in S$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. The ith state of ω is denoted by $\omega(i)$. A path can be either finite or infinite. For a finite path ω_{fin}, the last state is written $last(\omega_{fin})$. We say that a finite path ω_{fin} of length n is a *prefix* of the infinite path ω if the first $n+1$ states of ω are exactly as given by ω_{fin}. Unless stated explicitly, we always deal with infinite paths. The set of all (infinite) paths starting in state s is $Path_s$.

Since an execution of the system corresponds to a path, in order to reason about the probabilistic behaviour of the system, we need to be able to determine the probability that paths in a DTMC are taken. We do this in the standard way [KSK66] by, for each state $s \in S$, defining a probability measure $Prob_s$ on $Path_s$. The probability measure is induced by the transition probability matrix $\mathbf{P}$ in the following way. We first define the probability $\mathbf{P}(\omega_{fin})$ of a finite path ω_{fin} as $\mathbf{P}(\omega_{fin}) = 1$ if ω_{fin} consists of a single state s_0 and $\mathbf{P}(\omega_{fin}) = \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdots \mathbf{P}(s_{n-1}, s_n)$ in the general case where $\omega_{fin} = s_0 s_1 \ldots s_n$.

We next define the *cylinder set* $C(\omega_{fin})$, corresponding to a finite path ω_{fin}, as the set of all paths with prefix ω_{fin}. Then, let Σ_s be the smallest σ-algebra on $Path_s$ which contains all the sets $C(\omega_{fin})$ where ω_{fin} ranges over all finite paths starting in s. We define the probability measure $Prob_s$ on Σ_s as the unique measure with $Prob_s(C(\omega_{fin})) = \mathbf{P}(\omega_{fin})$. We can now quantify the probability that a DTMC behaves in a specified fashion by identifying the set of paths which satisfy this specification and, assuming that it is measurable, using the associated measure $Prob_s$.

3.1.2 Markov Decision Processes

The second type of model we consider, *Markov decision processes* (MDPs), can be seen as a generalisation of DTMCs. An MDP can describe both nondeterministic and probabilistic behaviour. It is well known that nondeterminism is a valuable tool for modelling concurrency: an MDP allows us to describe the behaviour of a number of probabilistic

systems operating in parallel. Nondeterminism is also useful when the exact probability of a transition is not known, or when it is known but not considered relevant. We define an MDP as a tuple $(S, \overline{s}, \mathit{Steps}, L)$ where:

- S is a finite set of *states*
- $\overline{s} \in S$ is the *initial state*
- $\mathit{Steps} : S \to 2^{\mathit{Dist}(S)}$ is the *transition function*
- $L : S \to 2^{AP}$ is the *labelling function*.

The set S, initial state $\overline{s}$ and labelling function L are as for DTMCs. The transition probability matrix $\mathbf{P}$, however, is replaced by Steps, a function mapping each state $s \in S$ to a finite, non-empty subset of $\mathit{Dist}(S)$, the set of all probability distributions over S (i.e. the set of all functions of the form $\mu : S \to [0,1]$ where $\sum_{s \in S} \mu(s) = 1$). Intuitively, for a given state $s \in S$, the elements of $\mathit{Steps}(s)$ represent *nondeterministic choices* available in that state. Each nondeterministic choice is a probability distribution, giving the likelihood of making a transition to any other state in S.

A *path* in the MDP is a non-empty sequence of the form $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \ldots$ where $s_i \in S$, $\mu_{i+1} \in \mathit{Steps}(s_i)$ and $\mu_{i+1}(s_{i+1}) > 0$ for all $i \geq 0$. As for DTMCs, $\omega(i)$ denotes the ith state of a path ω and $\mathit{last}(\omega_{\mathit{fin}})$ is the last state of a finite path ω_{fin}. Likewise, Path_s is the set of all (infinite) paths starting in state s.

Note that to trace a path through an MDP, both the nondeterministic and probabilistic choices have to be resolved. We assume that the nondeterministic choices are made by an *adversary* (also known as a 'scheduler' or 'policy'), which selects a choice based on the history of choices made so far. Formally, an adversary A is a function mapping every finite path ω_{fin} of the MDP onto a distribution $A(\omega_{\mathit{fin}}) \in \mathit{Steps}(\mathit{last}(\omega_{\mathit{fin}}))$. We denote by Path_s^A the subset of Path_s which corresponds to adversary A.

The behaviour of the MDP under a given adversary A is purely probabilistic. It can be described by a (usually infinite state) DTMC whose states are associated with finite paths of the original MDP and where the probability of making a transition between two such states is given by the probability distribution selected by A. More specifically, the state space of this DTMC is equal to the set of all finite paths in the MDP and its transition probability matrix $\mathbf{P}^A$ is defined as follows. For two finite paths ω and ω', $\mathbf{P}^A(\omega, \omega') = A(\omega)(s)$ if ω' is of the form $\omega \xrightarrow{A(\omega)} s$ and $\mathbf{P}^A(\omega, \omega') = 0$ otherwise. For a state $s \in S$, there is a one-to-one correspondence between the paths of this DTMC which start with the zero-length path s and the set of paths Path_s^A in the MDP. Hence, using the probability measure over DTMCs given in the previous section, we can define a probability measure Prob_s^A over the set of paths Path_s^A. The interested reader is referred to [BK98] for a more in-depth coverage of this material.

Reasoning about the precise probabilistic behaviour of an MDP for a single adversary is of limited use. We can, however, still verify meaningful properties of the MDP by computing the *maximum* or *minimum* probability that some specified behaviour is observed *over all possible adversaries*.

It is useful to extend this idea by computing the maximum or minimum probability for some class of adversaries. In particular, this allows the notion of *fairness* to be introduced. Consider an MDP which models several probabilistic systems running in parallel. Effectively, an adversary of the MDP selects which system is scheduled and when. It will often be impossible to verify basic properties of this system without making some assumptions about the fairness of this scheduling.

We will do this by distinguishing between *fair* and *unfair* adversaries. We use the definitions of fairness from [BK98], which are loosely based on those of [Var85]. For alternative notions of fairness in probabilistic systems, see for example [dA98, BK98]. We say that a path ω of an MDP is fair if, for states s occurring infinitely often in ω, each choice $\mu \in \mathit{Steps}(s)$ is taken infinitely often. We then define an adversary A to be fair if $\mathit{Prob}_s^A(\{\omega \in \mathit{Path}_s^A \mid \omega \text{ is fair}\}) = 1$ for all $s \in S$.

In our description of MDPs, we have adopted the terminology and notation of [BK98], although they refer to them as 'concurrent probabilistic systems'. The model can be also seen as a generalisation of the 'concurrent Markov chains' of [Var85] and is essentially the same as the 'simple probabilistic automata' of [SL94]. Note that, for our purposes, we omit some features which are frequently found in the definition of an MDP: action labels for nondeterministic choices and state- or choice-based rewards.

3.1.3 Continuous-Time Markov Chains

The final type of model, *continuous-time Markov chains* (CTMCs), also extend DTMCs but in a different way. While each transition of a DTMC corresponds to a discrete time-step, in a CTMC transitions can occur in real time. Each transition is labelled with a *rate*, defining the delay which occurs before it is taken. The delay is sampled from a negative exponential distribution with parameter equal to this rate. We define a CTMC as a tuple $(S, \overline{s}, \mathbf{R}, L)$ where:

- S is a finite set of *states*
- $\overline{s} \in S$ is the *initial state*
- $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$ is the *transition rate matrix*
- $L : S \to 2^{AP}$ is the *labelling function*.

The elements S, $\overline{s}$ and L are, again, as for DTMCs. The transition rate matrix $\mathbf{R}$, however, gives the rate, as opposed to the probability, of making transitions between states. For

states s and s', the probability of a transition from s to s' being *enabled* within t time units is $1 - e^{-\mathbf{R}(s,s')\cdot t}$. Typically, there is more than one state s' with $\mathbf{R}(s,s') > 0$. This is known as a *race condition*. The transition taken will be the one which is enabled first. Hence, it can be shown that the actual probability $\mathbf{P}(s,s')$ of moving from state s to state s' in a single step is $\mathbf{R}(s,s')/\sum_{s'' \in S} \mathbf{R}(s,s'')$ (unless s has no outgoing transitions, in which case we set $\mathbf{P}(s,s')$ to 1 if $s = s'$ and to 0 if $s \neq s'$). This information is captured by the *embedded Markov chain*, a DTMC $(S, \overline{s}, \mathbf{P}, L)$, where S, $\overline{s}$ and L are identical to the CTMC and $\mathbf{P}(s,s')$ is as just described. Note that we do allow self-loops, i.e. states s with $\mathbf{R}(s,s) > 0$. We also define the *generator matrix* $\mathbf{Q}$ of the CTMC as follows: $\mathbf{Q}(s,s') = \mathbf{R}(s,s')$ if $s \neq s'$ and $\mathbf{Q}(s,s') = -\sum_{s'' \neq s} \mathbf{R}(s,s'')$ if $s = s'$.

A path in a CTMC is a non-empty sequence $s_0 t_0 s_1 t_1 s_2 \ldots$ where $\mathbf{R}(s_i, s_{i+1}) > 0$ and $t_i \in \mathbb{R}_{>0}$ for all $i \geq 0$. The value t_i represents the amount of time spent in the state s_i. As with DTMCs and MDPs, we denote by $\omega(k)$ the kth state of a path ω, i.e. s_k. In addition, we denote by $\omega@t$ the state occupied at time t, i.e. $\omega(k)$ where k is the smallest index for which $\sum_{i=0}^{k} t_i \geq t$.

We again denote by $Path_s$ the set of all infinite paths starting in state s. The probability measure $Prob_s$ over $Path_s$, taken from [BKH99], can be defined as follows. If the states $s_0, \ldots, s_n \in S$ satisfy $\mathbf{R}(s_i, s_{i+1}) > 0$ for all $0 \leq i < n$ and $I_0, \ldots, I_{n-1}$ are non-empty intervals in $\mathbb{R}_{\geq 0}$, then the *cylinder set* $C(s_0, I_0, \ldots, I_{n-1}, s_n)$ is defined to be the set containing all paths $s'_0 t_0 s'_1 t_1 s'_2 \ldots$ where $s_i = s'_i$ for $i \leq n$ and $t_i \in I_i$ for $i < n$.

We then let Σ_s be the smallest σ-algebra on $Path_s$ which contains all the cylinder sets $C(s_0, I_0, \ldots, I_{n-1}, s_n)$ where $s_0, \ldots, s_n \in S$ range over all sequences of states with $s_0 = s$ and $\mathbf{R}(s_i, s_{i+1}) > 0$ for $0 \leq i < n$, and $I_0, \ldots, I_{n-1}$ range over all sequences of non-empty intervals in $\mathbb{R}_{\geq 0}$. The probability measure $Prob_s$ on Σ_s is then the unique measure defined inductively by $Prob_s(C(s_0)) = 1$ and $Prob_s(C(s_0, \ldots, s_n, I_n, s_{n+1}))$ equal to:

$$Prob_s(C(s_0, \ldots, s_n)) \cdot \mathbf{P}(s_n, s_{n+1}) \cdot \left(e^{-\inf I_n \cdot \sum_{s' \in S} \mathbf{Q}(s_n, s')} - e^{-\sup I_n \cdot \sum_{s' \in S} \mathbf{Q}(s_n, s')} \right)$$

In addition to path probabilities, we consider two more traditional properties of CTMCs: *transient* behaviour, which relates to the state of the model at a particular time instant; and *steady-state* behaviour, which describes the state of the CTMC in the long run. The transient probability $\pi_{s,t}(s')$ is defined as the probability, having started in state s, of being in state s' at time instant t. The steady-state probability $\pi_s(s')$ is defined as $\lim_{t \to \infty} \pi_{s,t}(s')$. For a certain class of CTMCs, namely those which are *ergodic*, the steady-state probability distribution can be shown to be independent of the initial state s. More information on this can be found in any standard text on CTMCs, e.g. [Ste94]. For simplicity, in this thesis we only deal with ergodic CTMCs.

3.2 Probabilistic Specification Formalisms

We now describe the formalisms that we use to specify properties of our models. As is usually the case with model checking, these are temporal logics; in this case PCTL, which is interpreted over DTMCs or MDPs, and CSL, which is interpreted over CTMCs.

3.2.1 PCTL

PCTL (Probabilistic Computational Tree Logic) [HJ94] is a probabilistic extension of the temporal logic CTL. It is essentially the same as the logic pCTL of [ASB+95]. The syntax of PCTL is as follows:

$$\begin{array}{rcl} \phi & ::= & \mathit{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi] \\ \psi & ::= & X\,\phi \mid \phi\,\mathcal{U}^{\leq k}\,\phi \mid \phi\,\mathcal{U}\,\phi \end{array}$$

where a is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0,1]$ and $k \in \mathbb{N}$. PCTL formulas are interpreted over a DTMC or an MDP. Note that each atomic proposition a must be taken from the set used to label the states of this DTMC or MDP.

In the syntax above we distinguish between state formulas ϕ and path formulas ψ, which are evaluated over states and paths, respectively. A property of a model will always be expressed as a state formula. Path formulas only occur as the parameter of the *probabilistic path operator* $\mathcal{P}_{\bowtie p}[\psi]$. Intuitively, a state s satisfies $\mathcal{P}_{\bowtie p}[\psi]$ if the probability of taking a path from s satisfying ψ is in the interval specified by $\bowtie p$.

As path formulas we allow the X (*next*), $\mathcal{U}^{\leq k}$ (*bounded until*) and $\mathcal{U}$ (*until*) operators which are standard in temporal logic. Intuitively, $X\,\phi$ is true if ϕ is satisfied in the next state; $\phi_1\,\mathcal{U}^{\leq k}\,\phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 is true up until that point; and $\phi_1\,\mathcal{U}\,\phi_2$ is true if ϕ_2 is satisfied at some point in the future and ϕ_1 is true up until then. In the following two sections, we formally define the semantics of PCTL over DTMCs and MDPs.

PCTL over DTMCs

For a DTMC $(S, \overline{s}, \mathbf{P}, L)$, state $s \in S$ and PCTL formula ϕ, we write $s \models \phi$ to indicate that ϕ is satisfied in s. Alternatively, we say that ϕ holds in s or is true in s. We denote by $\mathit{Sat}(\phi)$ the set $\{s \in S \mid s \models \phi\}$ of all states satisfying the formula ϕ. Similarly, for a path ω satisfying path formula ψ, we write $\omega \models \psi$. We can now give the formal semantics of PCTL over DTMCs. For a path ω:

$$\begin{array}{lcl} \omega \models X\,\phi & \iff & \omega(1) \models \phi \\ \omega \models \phi_1\,\mathcal{U}^{\leq k}\,\phi_2 & \iff & \exists i \leq k\,.\,(\omega(i) \models \phi_2 \wedge \omega(j) \models \phi_1\ \forall j < i) \\ \omega \models \phi_1\,\mathcal{U}\,\phi_2 & \iff & \exists k \geq 0\,.\,\omega \models \phi_1\,\mathcal{U}^{\leq k}\,\phi_2 \end{array}$$

and for a state $s \in S$:

$$\begin{array}{lcl} s \models \mathit{true} & & \text{for all } s \in S \\ s \models a & \iff & a \in L(s) \\ s \models \phi_1 \wedge \phi_2 & \iff & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \neg\phi & \iff & s \not\models \phi \\ s \models \mathcal{P}_{\bowtie p}[\psi] & \iff & p_s(\psi) \bowtie p \end{array}$$

where $p_s(\psi) = \mathit{Prob}_s(\{\omega \in \mathit{Path}_s \mid \omega \models \psi\})$. The probability Prob_s assigned to this set of paths is as defined in Section 3.1.1 and is provably measurable for all possible PCTL path formulas (see e.g. [Var85]).

PCTL over MDPs

The semantics of path formulas remain the same for MDPs as for DTMCs. However, as we saw in Section 3.1.2, the probability of a set of paths in an MDP can only be computed for a particular adversary. We denote by $p_s^A(\psi)$ the probability that a path from s satisfies path formula ψ under adversary A, i.e. $p_s^A(\psi) = \mathit{Prob}_s^A(\{\omega \in \mathit{Path}_s^A \mid \omega \models \psi\})$. To give the semantics of a PCTL formula $\mathcal{P}_{\bowtie p}[\psi]$, we choose a class of adversaries Adv (in this thesis, either all possible adversaries or only the fair ones) and then quantify over this set. We say that a state s satisfies $\mathcal{P}_{\bowtie p}[\psi]$ if $p_s^A(\psi) \bowtie p$ for *all* adversaries $A \in \mathit{Adv}$. Hence, the satisfaction relation is now parameterised by a class of adversaries Adv:

$$\begin{array}{lcl} s \models_{\mathit{Adv}} \mathit{true} & & \text{for all } s \in S \\ s \models_{\mathit{Adv}} a & \iff & a \in L(s) \\ s \models_{\mathit{Adv}} \phi_1 \wedge \phi_2 & \iff & s \models_{\mathit{Adv}} \phi_1 \wedge s \models_{\mathit{Adv}} \phi_2 \\ s \models_{\mathit{Adv}} \neg\phi & \iff & s \not\models_{\mathit{Adv}} \phi \\ s \models_{\mathit{Adv}} \mathcal{P}_{\bowtie p}[\psi] & \iff & p_s^A(\psi) \bowtie p \text{ for all } A \in \mathit{Adv} \end{array}$$

Additional Operators

From the basic syntax of PCTL, given above, we can derive a number of additional useful operators. Among these are the well known logical equivalences:

$$\begin{array}{rcl} \mathit{false} & \equiv & \neg\mathit{true} \\ \phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) \\ \phi_1 \rightarrow \phi_2 & \equiv & \neg\phi_1 \vee \phi_2 \end{array}$$

We also allow path formulas to contain the $\Diamond$ (*diamond*) operator, which is common in temporal logic. Intuitively, $\Diamond\phi$ means that ϕ is eventually satisfied, and its bounded

variant $\Diamond^{\leq k}\phi$ means that ϕ is satisfied within k time units. These can be expressed in terms of the PCTL until and bounded until operators as follows:

$$\begin{aligned} \Diamond\phi &\equiv \mathit{true}\ \mathcal{U}\ \phi \\ \Diamond^{\leq k}\phi &\equiv \mathit{true}\ \mathcal{U}^{\leq k}\ \phi \end{aligned}$$

When writing specifications for MDPs in PCTL, it may sometimes be useful to consider the *existence* of an adversary, rather than state that *all* adversaries satisfy some property. This can be done via translation to a dual property. For example, verifying that "there exists an adversary A for which $p_s^A(\phi_1\ \mathcal{U}\ \phi_2) \geq p$" is equivalent to model checking the PCTL formula $\neg\mathcal{P}_{<1-p}[\phi_1\ \mathcal{U}\ \phi_2]$.

A perceived weakness of PCTL is that it is not possible to determine the actual probability with which a certain path formula is satisfied, only whether or not the probability meets a particular bound. However, since the PCTL model checking algorithms proceed by computing the actual probability and then comparing it to the bound, this restriction on the syntax can be relaxed. If the outermost operator of a PCTL formula is the $\mathcal{P}_{\bowtie p}$ operator, we can omit the bound $\bowtie p$ and simply compute the probability instead. Furthermore, it may be useful to extend this idea by, for example, computing the probability that a path formula $\Diamond^{\leq k}\phi$ is satisfied for several values of k and then plotting a graph of this information.

For completeness, we should also discuss the limitations of PCTL. There are some useful properties of DTMCs and MDPs which cannot be expressed in the logic. The types of path formula, for example, supported by PCTL are quite limited. This could be solved by using the logic LTL (linear time temporal logic) which allows more complex path formulas. For example, we could compute the probability of the set of paths satisfying the path formula $\Diamond\phi_1 \wedge \Diamond\phi_2$, i.e. those where both ϕ_1 and ϕ_2 are eventually satisfied (but not necessarily at the same time). Note that this cannot be derived from the probabilities that the individual path formulas $\Diamond\phi_1$ and $\Diamond\phi_2$ are satisfied. We could add further expressivity by using the logic PCTL* [ASB+95, BdA95] which subsumes both PCTL and LTL.

Unfortunately this expressivity comes at a cost of increased model checking complexity. The model checking algorithms for PCTL on both DTMCs [CY88, HJ94] and MDPs [CY90, BdA95] are polynomial in the size of the model and linear in the size of the formula. Model checking for LTL and PCTL*, e.g. [Var85, CY88, ASB+95, BdA95, BK98], however, proceeds by translating the model to a larger one for a given property and is exponential in the size of the formula for DTMCs and at least doubly exponential in the size of the formula for MDPs.

3.2.2 CSL

The logic CSL (Continuous Stochastic Logic) was introduced in [ASSB96] and extended in [BKH99]. It is similar to the logic PCTL, but is designed to specify properties of CTMCs. CSL provides a way to describe steady-state and transient behaviour which are both elements of traditional CTMC analysis. It also allows specification of more involved properties using the probabilistic path operator of PCTL. The syntax is:

$$\begin{array}{rcl} \phi & ::= & \mathit{true} \mid a \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{S}_{\bowtie p}[\phi] \\ \psi & ::= & X\,\phi \mid \phi\,\mathcal{U}^{\leq t}\,\phi \mid \phi\,\mathcal{U}\,\phi \end{array}$$

where a is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0,1]$ and $t \in \mathbb{R}_{\geq 0}$.

As for PCTL, $\mathcal{P}_{\bowtie p}[\psi]$ indicates that the probability of the path formula ψ being satisfied from a given state satisfies the bound $\bowtie p$. Path formulas are the same for CSL as for PCTL except that the parameter t of the bounded until operator $\phi_1\,\mathcal{U}^{\leq t}\,\phi_2$ is a non-negative real rather than an non-negative integer. The formula holds if ϕ_2 is satisfied at some time instant in the interval $[0,t]$ and ϕ_1 holds at all preceding time instants. To avoid confusion, we will refer to this as the *time-bounded until* operator. The $\mathcal{S}$ operator describes the steady-state behaviour of the CTMC. The formula $\mathcal{S}_{\bowtie p}[\phi]$ asserts that the steady-state probability of being in a state satisfying ϕ meets the bound $\bowtie p$.

CSL over CTMCs

As with PCTL, we write $s \models \phi$ to indicate that a CSL formula ϕ is satisfied in a state s of a CTMC and denote by $\mathit{Sat}(\phi)$ the set $\{s \in S \mid s \models \phi\}$. Similarly, for a path formula ψ satisfied by path ω, we write $\omega \models \psi$. For a CTMC $(S, \overline{s}, \mathbf{R}, L)$, the semantics of CSL are:

$$\begin{array}{lcl} \omega \models X\,\phi & \iff & \omega(1) \text{ is defined and } \omega(1) \models \phi \\ \omega \models \phi_1\,\mathcal{U}^{\leq t}\,\phi_2 & \iff & \exists x \in [0,t]\,.\,(\omega@x \models \phi_2 \wedge \omega@y \models \phi_1\ \forall y \in [0,x)) \\ \omega \models \phi_1\,\mathcal{U}\,\phi_2 & \iff & \exists k \geq 0\,.\,(\omega(k) \models \phi_2 \wedge \omega(j) \models \phi_1\ \forall j < k) \end{array}$$

and:

$$\begin{array}{lcl} s \models \mathit{true} & & \text{for all } s \in S \\ s \models a & \iff & a \in L(s) \\ s \models \phi_1 \wedge \phi_2 & \iff & s \models \phi_1 \wedge s \models \phi_2 \\ s \models \neg\phi & \iff & s \not\models \phi \\ s \models \mathcal{P}_{\bowtie p}[\psi] & \iff & p_s(\psi) \bowtie p \\ s \models \mathcal{S}_{\bowtie p}[\phi] & \iff & \sum_{s' \models \phi} \pi_s(s') \bowtie p \end{array}$$

where $p_s(\psi) = \mathit{Prob}_s(\{\omega \in \mathit{Path}_s \mid \omega \models \psi\})$.

Additional Operators

As with PCTL, we can derive CSL operators for *false*, $\vee$, $\rightarrow$ and $\Diamond$. Note, however, that the bounded form of the diamond operator is $\Diamond^{\leq t}$, where t is a non-negative real, as in the time-bounded until operator. Again, like with PCTL, when the outermost operator of a CSL formula is $\mathcal{P}$ or $\mathcal{S}$, we can omit the bound $\bowtie p$ and just return the probability.

An extension specific to CSL is that the time-bound attached to a $\mathcal{U}$ or $\Diamond$ operator can be replaced with an arbitrary time interval, allowing formulas such as:

$$\mathcal{P}_{\bowtie p}[\phi_1 \; \mathcal{U}^{[t_1,t_2]} \; \phi_2]$$
$$\mathcal{P}_{\bowtie p}[\Diamond^{\geq t} \; \phi]$$

Model checking is described in [BHHK00a] and is not much more expensive, requiring an amount of work equivalent to model checking two of the simpler, time-bounded instances described above. For simplicity, we do not consider this case here.

3.3 Probabilistic Model Checking

We now summarise the model checking algorithms for the three cases discussed previously: PCTL over DTMCs, PCTL over MDPs and CSL over CTMCs. A model checking algorithm for either PCTL or CSL takes a model of the appropriate type, a formula ϕ in the logic, and returns the set $Sat(\phi)$ containing the states of the model which satisfy ϕ.

The overall structure of the model checking algorithm is the same in all three cases, and originates from the original CTL model checking algorithm presented in [CES86]. We first construct the parse tree of the formula ϕ. Each node of the tree is labelled with a subformula of ϕ and the root node is labelled with ϕ itself. Leaves of the tree will be labelled with either *true* or an atomic proposition a. Working upwards towards the root of the tree, we recursively compute the set of states satisfying each subformula. By the end, we have determined whether each state in the model satisfies ϕ.

Model checking of the non-probabilistic operators of the two logics is performed identically for DTMCs, MDPs and CTMCs: it is trivial to deduce from the model which states satisfy a given atomic proposition, and logical operators such as conjunction and negation are also simple. The non-trivial cases are the $\mathcal{P}$ and $\mathcal{S}$ operators. Here, it is necessary to compute the relevant probabilities and then identify the states which satisfy the bound given in the formula. The calculation of probabilities required for each operator is described in the following sections. The time complexity of these computations is, at worst, polynomial in the size of the model. Hence, the complexity for PCTL model checking over DTMCs and MDPs or CSL model checking over CTMCs is linear in the size of the temporal logic formula and polynomial in the size of the model.

3.3.1 PCTL Model Checking of DTMCs

For model checking the PCTL $\mathcal{P}_{\bowtie p}[\psi]$ operator against a DTMC $(S, \overline{s}, \mathbf{P}, L)$, we need to compute the probability that a path leaving each state s will satisfy the path formula ψ. When ψ is a PCTL next ($X\,\phi$), bounded until ($\phi_1\,\mathcal{U}^{\leq k}\,\phi_2$) or until ($\phi_1\,\mathcal{U}\,\phi_2$) formula, we calculate, for all states $s \in S$, the probabilities $p_s(X\,\phi)$, $p_s(\phi_1\,\mathcal{U}^{\leq k}\,\phi_2)$ and $p_s(\phi_1\,\mathcal{U}\,\phi_2)$, respectively. We then compute $Sat(\mathcal{P}_{\bowtie p}[\psi])$ as $\{s \in S \mid p_s(\psi) \bowtie p\}$. Because of the recursive nature of the PCTL model checking algorithm, we can assume that the relevant sets, $Sat(\phi)$, $Sat(\phi_1)$ or $Sat(\phi_2)$, are already known. The algorithms described in the following sections were first presented in [CY88, HJ94].

PCTL Next

It is easy to show that $p_s(X\,\phi) = \sum_{s' \in Sat(\phi)} \mathbf{P}(s, s')$. Assuming that we have a state-indexed vector $\underline{\phi}$ with $\underline{\phi}(s) = 1$ if $s \models \phi$ and 0 otherwise, we can compute the vector $\underline{p(X\,\phi)}$ of required probabilities as follows: $\underline{p(X\,\phi)} = \mathbf{P} \cdot \underline{\phi}$. This requires a single matrix-vector multiplication.

PCTL Bounded Until

We first divide all states into three disjoint sets: S^{no}, S^{yes}, and $S^{?}$. The sets $S^{no} = S \backslash (Sat(\phi_1) \cup Sat(\phi_2))$ and $S^{yes} = Sat(\phi_2)$ contain the states for which $p_s(\phi_1\,\mathcal{U}^{\leq k}\,\phi_2)$ is trivially 0 or 1, respectively. The set $S^{?} = S \backslash (S^{no} \cup S^{yes})$ contains all remaining states. For these states, we have:

$$p_s(\phi_1\,\mathcal{U}^{\leq k}\,\phi_2) = \begin{cases} 0 & \text{if } k = 0 \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot p_s(\phi_1\,\mathcal{U}^{\leq k-1}\,\phi_2) & \text{if } k \geq 1 \end{cases}$$

If we define the matrix $\mathbf{P}'$ as:

$$\mathbf{P}'(s, s') = \begin{cases} \mathbf{P}(s, s') & \text{if } s \in S^{?} \\ 1 & \text{if } s \in S^{yes} \text{ and } s = s' \\ 0 & \text{otherwise} \end{cases}$$

and abbreviate the vector of required probabilities $\underline{p(\phi_1\,\mathcal{U}^{\leq k}\,\phi_2)}$ to $\underline{p_k}$, the computation needed is as follows. For $k = 0$, $\underline{p_0}(s) = 1$ if $s \in S^{yes}$ and 0 otherwise. For $k > 0$, $\underline{p_k} = \mathbf{P}' \cdot \underline{p_{k-1}}$. In total, this requires k matrix-vector multiplications.

PCTL Until

As with the bounded until operator, one can identify the sets for which $p_s(\phi_1\,\mathcal{U}\,\phi_2)$ is trivially 0 or 1. In this case, it is useful to extend these sets to contain *all* states for which

Prob0($Sat(\phi_1), Sat(\phi_2)$)	
1.	$R := Sat(\phi_2)$
2.	$done :=$ **false**
3.	**while** ($done =$ **false**)
4.	$\quad R' := R \cup \{s \in Sat(\phi_1) \mid \exists s' \in R \,.\, \mathbf{P}(s, s') > 0\}$
5.	$\quad$ **if** $(R' = R)$ **then** $done :=$ **true**
6.	$\quad R := R'$
7.	**endwhile**
8.	**return** $S \backslash R$

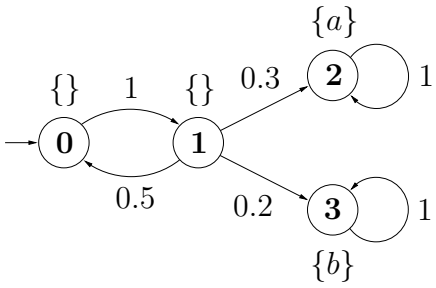
Figure 3.1: The Prob0 algorithm

Prob1($Sat(\phi_1), Sat(\phi_2), S^{no}$)	
1.	$R := S^{no}$
2.	$done :=$ **false**
3.	**while** ($done =$ **false**)
4.	$\quad R' := R \cup \{s \in (Sat(\phi_1) \backslash Sat(\phi_2)) \mid \exists s' \in R \,.\, \mathbf{P}(s, s') > 0\}$
5.	$\quad$ **if** $(R' = R)$ **then** $done :=$ **true**
6.	$\quad R := R'$
7.	**endwhile**
8.	**return** $S \backslash R$

Figure 3.2: The Prob1 algorithm

$p_s(\phi_1 \,\mathcal{U}\, \phi_2)$ is exactly 0 or 1. As above, we denote these sets S^{no} and S^{yes}. They can be determined with the fixpoint algorithms Prob0 and Prob1, respectively, described in Figures 3.1 and 3.2. Prob0 computes all the states from which it is possible, with non-zero probability, to reach a state satisfying ϕ_2 without leaving states satisfying ϕ_1. It then subtracts these from S to determine the states which have a zero probability. Prob1 computes the set S^{yes} in a similar fashion to Prob0 by first determining the set of states for which the probability is *less than* 1. These are the states from which there is a non-zero probability of reaching a state in S^{no} passing only through states satisfying ϕ_1 but not ϕ_2.

These two algorithms form the first part of the calculation of $p_s(\phi_1 \,\mathcal{U}\, \phi_2)$. For this reason, we refer to them as *precomputation algorithms*. It should be noted that for *qualitative* PCTL properties (i.e. where the bound p in the $\mathcal{P}_{\bowtie p}$ operator is either 0 or 1) or for cases where $p_s(\phi_1 \,\mathcal{U}\, \phi_2)$ happens to be either 0 or 1 for all states (i.e. $S^{no} \cup S^{yes} = S$), it suffices to use these precomputation algorithms. For *quantitative* properties with an arbitrary bound $\bowtie p$, numerical computation is also usually required. The precompu-



$$\mathbf{P} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.3 & 0.2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.3: A 4 state DTMC and its transition probability matrix

tation algorithms are still valuable, though, particularly since they determine the exact probability for the states in S^{no} and S^{yes}, whereas numerical computation typically only computes an approximation and may also be subject to round-off errors.

We still need to compute $p_s(\phi_1 \,\mathcal{U}\, \phi_2)$ for the remaining states $S^? = S \backslash (S^{no} \cup S^{yes})$. This can be done by solving the linear equation system in variables $\{x_s \mid s \in S\}$:

$$x_s = \begin{cases} 0 & \text{if } s \in S^{no} \\ 1 & \text{if } s \in S^{yes} \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot x_{s'} & \text{if } s \in S^? \end{cases}$$

and then letting $p_s(\phi_1 \,\mathcal{U}\, \phi_2) = x_s$. To rewrite this in the traditional $\mathbf{A} \cdot \underline{x} = \underline{b}$ form, we let $\mathbf{A} = \mathbf{I} - \mathbf{P'}$ where $\mathbf{I}$ is the identity matrix and $\mathbf{P'}$ is given by:

$$\mathbf{P'}(s, s') = \begin{cases} \mathbf{P}(s, s') & \text{if } s \in S^? \\ 0 & \text{otherwise} \end{cases}$$

and $\underline{b}$ is a column vector over states with $\underline{b}(s)$ equal to 1 if $s \in S^{yes}$ and 0 otherwise. The system $\mathbf{A} \cdot \underline{x} = \underline{b}$ can then be solved by any standard approach. These include direct methods, such as Gaussian elimination, or iterative methods, such as Jacobi and Gauss-Seidel. Since we are aiming to solve these problems for very large models, we will concentrate on iterative methods. This is discussed further in Section 3.5.

Example

We conclude our coverage of model checking for DTMCs with a simple example. Figure 3.3 shows a DTMC with four states $\{0, 1, 2, 3\}$. In our graphical notation, states are drawn as circles and transitions as arrows, labelled with their associated probabilities. The initial state is indicated by an additional incoming arrow. The atomic propositions attached to each state, in this case taken from the set $\{a, b\}$, are also shown.

We consider the PCTL formula $\mathcal{P}_{\geq 0.5}[\neg b \,\mathcal{U}\, a]$. Hence, we need to calculate the probability $p_s(\neg b \,\mathcal{U}\, a)$ for $s \in \{0, 1, 2, 3\}$. Since a is true in state 2 and neither a nor $\neg b$ are

true in state 3, the probabilities for these two states are 1 and 0, respectively. On this simple example, the precomputation algorithms Prob0 and Prob1 identify no further states with $p_s(\neg b \,\mathcal{U}\, a)$ equal to 0 or 1. Thus, $S^{no} = \{3\}$, $S^{yes} = \{2\}$ and $S^{?} = \{0, 1\}$. The remaining probabilities can be computed by solving the following system of linear equations: $x_0 = x_1$, $x_1 = 0.5\,x_0 + 0.3\,x_2 + 0.2\,x_3$, $x_2 = 1$ and $x_3 = 0$. This yields the solution $(x_0, x_1, x_2, x_3) = (0.6, 0.6, 1, 0)$. Letting $p_s(\neg b \,\mathcal{U}\, a) = x_s$, we see that the formula $\mathcal{P}_{\geq 0.5}[\neg b \,\mathcal{U}\, a]$ is satisfied by states 0, 1 and 2.

3.3.2 PCTL Model Checking of MDPs

For an MDP $(S, \overline{s}, \mathit{Steps}, L)$, we are again required to compute probabilities for PCTL next, bounded until and until operators. Since we need to determine whether or not the bound $\bowtie p$ is satisfied for *all* adversaries in some set Adv, we actually compute either the maximum or minimum probability for the formula, depending on whether the relational operator $\bowtie$ defines an upper or lower bound:

$$\begin{array}{lcl} s \models_{\mathit{Adv}} \mathcal{P}_{\leq p}[\psi] & \iff & p_s^{max}(\psi) \leq p \\ s \models_{\mathit{Adv}} \mathcal{P}_{< p}[\psi] & \iff & p_s^{max}(\psi) < p \\ s \models_{\mathit{Adv}} \mathcal{P}_{\geq p}[\psi] & \iff & p_s^{min}(\psi) \geq p \\ s \models_{\mathit{Adv}} \mathcal{P}_{> p}[\psi] & \iff & p_s^{min}(\psi) > p \end{array}$$

where $p_s^{max}(\psi) = \max_{A \in \mathit{Adv}}[p_s^A(\psi)]$ and $p_s^{min}(\psi) = \min_{A \in \mathit{Adv}}[p_s^A(\psi)]$. The computation of $p_s^{max}(\psi)$ or $p_s^{min}(\psi)$ differs depending on whether the set Adv is all adversaries or the set of fair adversaries. In fact this is only true for PCTL until formulas; for next and bounded until formulas, there is no difference. Intuitively, this is because they relate to paths of bounded length and fairness only places restrictions on the long-run (infinite) behaviour of the model. The model checking algorithms we give here are those of [CY90, BdA95] plus the extensions in [BK98, Bai98] for handling fairness and in [dA97, dAKN$^+$00] for the Prob1E precomputation algorithm.

PCTL Next

We first consider the PCTL next operator. This is very similar to the case for DTMCs:

$$p_s^{max}(X\,\phi) = \max_{\mu \in \mathit{Steps}(s)} \left\{ \sum_{s' \in \mathit{Sat}(\phi)} \mu(s') \right\}$$

$$p_s^{min}(X\,\phi) = \min_{\mu \in \mathit{Steps}(s)} \left\{ \sum_{s' \in \mathit{Sat}(\phi)} \mu(s') \right\}$$

Let m be the total number of nondeterministic choices in all states of the MDP, i.e. $m = \sum_{s \in S} |\mathit{Steps}(s)|$. We can represent the function Steps as an $m \times |S|$ matrix **Steps**,

where each row of the matrix corresponds to a single nondeterministic choice. This means that there are $|Steps(s)|$ rows corresponding to each state s, rather than just one, as for a DTMC. Assume we also have a state-indexed vector $\underline{\phi}$ with $\underline{\phi}(s)$ equal to 1 if $s \models \phi$ and 0 otherwise. We can carry out the computation above in two steps: firstly, a matrix-vector multiplication $\mathbf{Steps} \cdot \underline{\phi}$ which computes the summation for each nondeterministic choice, producing a vector of length m; and secondly, an operation which selects, from this vector, the maximum or minimum value for each state, reducing the length from m to $|S|$.

PCTL Bounded Until

Again, computation for the PCTL bounded until operator is very similar to the case for DTMCs. We first divide the set of all states into three disjoint subsets: S^{no}, S^{yes}, and $S^{?}$. The sets $S^{no} = S \backslash (Sat(\phi_1) \cup Sat(\phi_2))$ and $S^{yes} = Sat(\phi_2)$ contain the states for which $p_s^{max}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2)$ or $p_s^{min}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2)$ is trivially 0 or 1, respectively. The set $S^{?}$ is defined as $S \backslash (S^{no} \cup S^{yes})$ and contains all remaining states. For $s \in S^{?}$, we have:

$$p_s^{max}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{\mu \in Steps(s)} \left\{ \sum_{s' \in S} \mu(s') \cdot p_s^{max}(\phi_1 \, \mathcal{U}^{\leq k-1} \, \phi_2) \right\} & \text{if } k \geq 1 \end{cases}$$

$$p_s^{min}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2) = \begin{cases} 0 & \text{if } k = 0 \\ \min_{\mu \in Steps(s)} \left\{ \sum_{s' \in S} \mu(s') \cdot p_s^{min}(\phi_1 \, \mathcal{U}^{\leq k-1} \, \phi_2) \right\} & \text{if } k \geq 1 \end{cases}$$

If we assume, as above, that *Steps* is represented by the matrix **Steps**, then the computation of $p_s^{max}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2)$ or $p_s^{min}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_2)$ can be carried out in k iterations, each one similar to the process described for the PCTL next operator. Every iteration will comprise one matrix-vector multiplication and one maximum or minimum operation.

PCTL Until (All Adversaries)

For the PCTL until operator, we must distinguish between the case for all adversaries and the case for fair adversaries. We begin with the former. The latter is covered in the next section. We are required to compute either the probabilities $p_s^{max}(\phi_1 \, \mathcal{U} \, \phi_2)$ or $p_s^{min}(\phi_1 \, \mathcal{U} \, \phi_2)$, which we will abbreviate to p_s^{max} and p_s^{min}, respectively, in the remainder of this section. Likewise, we will use p_s^A to denote $p_s^A(\phi_1 \, \mathcal{U} \, \phi_2)$.

As for DTMCs, we first compute the sets S^{no} and S^{yes}, which contain the states with probability equal to exactly 0 or 1, respectively. When computing p_s^{max}, S^{no} contains all the states for which $p_s^A = 0$ for *every* adversary A. We determine these with the precomputation algorithm Prob0A. Conversely, S^{yes} contains all the states for which $p_s^A = 1$ for *some* adversary A. This is done with the precomputation algorithm Prob1E. The two algorithms are shown in Figures 3.4 and 3.5, respectively.

```
Prob0A(Sat(φ1), Sat(φ2))
1.  R := Sat(φ2)
2.  done := false
3.  while (done = false)
4.      R' := R ∪ {s ∈ Sat(φ1) | ∃μ ∈ Steps(s) . ∃s' ∈ R . μ(s') > 0}
5.      if (R' = R) then done := true
6.      R := R'
7.  endwhile
8.  return S\R
```

Figure 3.4: The Prob0A algorithm

```
Prob1E(Sat(φ1), Sat(φ2))
1.  R := S
2.  done := false
3.  while (done = false)
4.      R' := Sat(φ2)
5.      done' := false
6.      while (done' = false)
7.          R'' := R' ∪ { s ∈ Sat(φ1) | ∃μ ∈ Steps(s) .
.                  (∀s' ∈ S . μ(s') > 0 → s' ∈ R) ∧ (∃s' ∈ R' . μ(s') > 0) }
8.          if (R'' = R') then done' := true
9.          R' := R''
10.     endwhile
11.     if (R' = R) then done := true
12.     R := R'
13. endwhile
14. return R
```

Figure 3.5: The Prob1E algorithm

```
Prob0E(Sat(φ1), Sat(φ2))
1.  R := Sat(φ2)
2.  done := false
3.  while (done = false)
4.      R' := R ∪ {s ∈ Sat(φ1) | ∀μ ∈ Steps(s) . ∃s' ∈ R . μ(s') > 0}
5.      if (R' = R) then done := true
6.      R := R'
7.  endwhile
8.  return S\R
```

Figure 3.6: The Prob0E algorithm

Prob0A works in a similar fashion to the earlier algorithm Prob0. It first computes the states from which one can, under some adversary, with non-zero probability, reach a state satisfying ϕ_2 without leaving states satisfying ϕ_1. It then subtracts these from S to determine the states which have a zero probability for every adversary.

The Prob1E algorithm is more involved. It was first presented in [dAKN+00] but is actually an extension of a related algorithm from [dA97]. In essence, it is similar to Prob1, given earlier, in that it works by identifying states for which p_s^{max} is less than 1. Prob1E is based on the computation of a double fixpoint and is hence implemented as two nested loops. The outer loop computes a set of states R. By the end of the algorithm, R will contain all the states s for which $p_s^A = 1$ for *some* adversary A, as required. Initially R is set to S, i.e. all states. In each iteration of the outer loop, states are removed from R for which there is no adversary A with $p_s^A = 1$. The inner loop identifies the states to be removed as those which cannot reach a state in $Sat(\phi_2)$ without passing through either a state not in $Sat(\phi_1)$ or a state already removed by the algorithm (i.e. not in R).

When computing p_s^{min}, S^{no} contains all states for which $p_s^A = 0$ for *some* adversary A. This is computed with the algorithm Prob0E, given in Figure 3.6, which works in an almost identical fashion to Prob0A. The approach taken in Prob1E, however, is not applicable to the problem of determining states where $p_s^A = 1$ for *every* adversary A. Hence, we take S^{yes} to be the set of states $Sat(\phi_2)$ for which p_s^{min} is trivially 1.

The computation of p_s^{max} or p_s^{min} for the remaining states in $S^? = S \backslash (S^{no} \cup S^{yes})$ can then be performed by solving a linear optimisation problem over the set of variables $\{x_s \mid s \in S^?\}$. For the case of p_s^{max}, the problem is:

Minimise $\sum_{s \in S^?} x_s$ subject to the constraints

$$x_s \geq \sum_{s' \in S^?} \mu_s(s') \cdot x_{s'} + \sum_{s' \in S^{yes}} \mu_s(s')$$

for all $s \in S^?$ and all $\mu_s \in Steps(s)$

and for the case of p_s^{min}:

Maximise $\sum_{s \in S^?} x_s$ subject to the constraints

$$x_s \leq \sum_{s' \in S^?} \mu_s(s') \cdot x_{s'} + \sum_{s' \in S^{yes}} \mu_s(s')$$

for all $s \in S^?$ and all $\mu_s \in Steps(s)$

In either case, the problem admits a unique optimal solution and we can let $p_s^{max} = x_s$ or $p_s^{min} = x_s$, respectively. These results can be found in [BT91, CY90, dA97].

Linear optimisation problems can be solved using classic techniques such as the Simplex method. However, as with model checking of the PCTL until operator over DTMCs, these direct methods are not well suited to problems of the size we wish to deal with. Fortunately, there is an alternative method for computing the probabilities p_s^{max} and p_s^{min}. As shown in [Bai98], $p_s^{max} = \lim_{n\to\infty} p_s^{max(n)}$ where:

$$p_s^{max(n)} = \begin{cases} 0 & \text{if } s \in S^{no} \\ 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{?} \text{ and } n = 0 \\ \max_{\mu \in Steps(s)} \left\{ \sum_{s' \in S} \mu(s') \cdot p_s^{max(n-1)} \right\} & \text{if } s \in S^{?} \text{ and } n > 0 \end{cases}$$

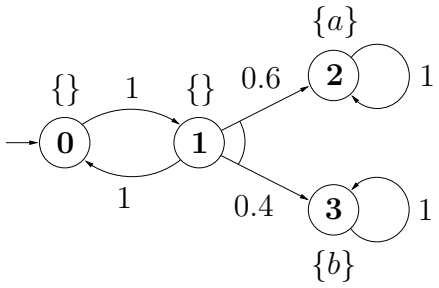
and $p_s^{min} = \lim_{n\to\infty} p_s^{min(n)}$ where:

$$p_s^{min(n)} = \begin{cases} 0 & \text{if } s \in S^{no} \\ 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{?} \text{ and } n = 0 \\ \min_{\mu \in Steps(s)} \left\{ \sum_{s' \in S} \mu(s') \cdot p_s^{min(n-1)} \right\} & \text{if } s \in S^{?} \text{ and } n > 0 \end{cases}$$

The values of p_s^{max} and p_s^{min} can now be approximated by an iterative computation, which calculates $p_s^{max(n)}$ and $p_s^{min(n)}$ for successive n, stopping when some convergence criterion has been satisfied. In our experiments, we have used the same convergence criterion as that applied in iterative methods for solving linear equation systems. This is discussed in Section 3.5. Note the similarity between a single iteration of this method and one of those required for the bounded until operator. Hence, assuming again that *Steps* is represented by a matrix **Steps**, each iteration can be performed in the same way using one matrix-vector multiplication and one maximum or minimum operation.

PCTL Until (Fair Adversaries Only)

We now describe the case of computing p_s^{max} and p_s^{min} over fair adversaries only, as presented in [BK98, Bai98]. The case for computing p_s^{max} actually remains unchanged, i.e. we can simply use the algorithm from the previous section. The reason for this is as follows. Because we are now considering a more restricted class of adversaries, the maximum probability clearly cannot increase. More importantly, though, it cannot decrease either. This is because, for any adversary A, we can always construct a fair adversary A' for which $p_s^{A'}(\phi_1 \,\mathcal{U}\, \phi_2) \geq p_s^{A}(\phi_1 \,\mathcal{U}\, \phi_2)$. Intuitively, the reason for this is that fairness only places restrictions on infinite behaviour and, for a path to satisfy an until formula, only some finite, initial portion of it is relevant.



$$\textbf{Steps} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ 0 & 0 & 0.6 & 0.4 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right)$$

Figure 3.7: A 4 state MDP and the matrix representing its transition function

For the case of computing p_s^{min}, this argument cannot be applied. The minimum probability over all fair adversaries *can* be higher than the minimum for all adversaries. In the next section, we show a simple example where this is true. Fortunately, using the technique of [Bai98], we can still compute p_s^{min} without much additional effort. The basic idea is to solve a dual problem, calculating the probability that $\phi_1 \,\mathcal{U}\, \phi_2$ is *not* satisfied. In doing so, we convert the problem to one where the maximum, not the minimum, probability is needed. For this, as we saw above, we can ignore fairness and use the method from the previous section. The desired probabilities can then be obtained from the computed probabilities by subtracting from 1.

The dual problem is constructed as follows. Let S^+ be the set of states for which $p_s^{max} > 0$. This can be computed using the Prob0A precomputation algorithm described above. Then, let $S^\#$ be the set $S^+ \backslash Sat(\phi_2)$. Using atomic propositions a^+ and $a^\#$ to label states in the sets S^+ and $S^\#$, respectively, it is shown in [Bai98] that, for a fair adversary A, $p_s^A(\phi_1 \,\mathcal{U}\, \phi_2) = 1 - p_s^A(a^\# \,\mathcal{U}\, \neg a^+)$. Hence, as described above, the problem reduces to computing $p_s^{max}(a^\# \,\mathcal{U}\, \neg a^+)$, for which fairness is irrelevant.

Example

We now give some simple examples of PCTL model checking over an MDP. Figure 3.7 shows an MDP with four states $\{0, 1, 2, 3\}$. As with DTMCs, states are drawn as circles and transitions as arrows, labelled with their associated probabilities. We join transitions corresponding to the same nondeterministic choice with an arc. This is only necessary when a choice can select more than one state with non-zero probability. Again, atomic propositions from the set $\{a, b\}$ label each state. We also give the matrix **Steps**, representing the transition function *Steps* of the MDP. Each row of the matrix corresponds to a single nondeterministic choice. For clarity, we separate the choices for each state with a horizontal line.

We begin by considering the formula $\mathcal{P}_{\leq 0.5}[\neg\, b \,\mathcal{U}\, a]$, model checking over all adver-

saries. Since "≤ 0.5" defines an upper bound, we are required to compute the maximum probabilities $p_s^{max}(\neg b \,\mathcal{U}\, a)$ for $s \in \{0,1,2,3\}$. We see immediately that $p_2^{max}(\neg b \,\mathcal{U}\, a) = 1$ and $p_3^{max}(\neg b \,\mathcal{U}\, a) = 0$. The precomputation algorithms Prob0A and Prob1E yield no additional states so we have $S^{no} = \{3\}$, $S^{yes} = \{2\}$ and $S^{?} = \{0,1\}$. The values of $p_s^{max}(\neg b \,\mathcal{U}\, a)$ for states 0 and 1 can be computed by solving the linear optimisation problem "Minimise $x_0 + x_1$ such that $x_0 \geq x_1$, $x_1 \geq x_0$ and $x_1 \geq 0.6$" which has the unique, optimal solution $(x_0, x_1) = (0.6, 0.6)$. Hence, our required vector of probabilities is $(0.6, 0.6, 1, 0)$ and the formula $\mathcal{P}_{\leq 0.5}[\neg b \,\mathcal{U}\, a]$ is satisfied only by state 3.

Note that an adversary of this MDP actually only has to make a choice in state 1 since $|Steps(s)| = 1$ for $s \in \{0,2,3\}$. Consider the adversary A which selects the 0.6/0.4 probability distribution the first time state 1 is reached. From this point on, state 1 can never be reached again and so A has no more decisions to make. It can be seen that the probabilities $p_s^A(\neg b \,\mathcal{U}\, a)$ are equal to the maximum probabilities $(0.6, 0.6, 1, 0)$. Furthermore, the adversary A is fair since state 1 cannot occur infinitely often in any path, from any state, corresponding to A. Hence, the values of $p_s^{max}(\neg b \,\mathcal{U}\, a)$ are the same when calculated over fair adversaries only. In fact, we already knew this since the model checking algorithm states that the computation remains unchanged.

Secondly, we consider the formula $\mathcal{P}_{\geq 0.5}[\neg b \,\mathcal{U}\, a]$. Since this contains a lower bound, we need to compute the minimum probabilities $p_s^{min}(\neg b \,\mathcal{U}\, a)$ for $s \in \{0,1,2,3\}$. We first perform model checking over all adversaries. As above, $p_2^{min}(\neg b \,\mathcal{U}\, a)$ and $p_3^{min}(\neg b \,\mathcal{U}\, a)$ are trivially 1 and 0, respectively. On this occasion, however, the precomputation algorithm Prob0E identifies that $p_s^{min}(\neg b \,\mathcal{U}\, a)$ is also 0 for states 0 and 1. Thus, $S^{no} = \{0,1,3\}$, $S^{yes} = \{2\}$, $S^{?} = \emptyset$ and $\mathcal{P}_{\geq 0.5}[\neg b \,\mathcal{U}\, a]$ is only satisfied in state 2. Like in the previous example, we can easily construct a simple adversary A which results in these minimum probabilities: the one which, in state 1, always selects the distribution leading to state 0 with probability 1. Clearly, under this adversary, it is impossible to reach state 2 from either state 0 or state 1. Hence, $p_s^A(\neg b \,\mathcal{U}\, a) = 0$ for $s \in \{0,1\}$.

Note, however, that this adversary A is not fair since, in any path, state 1 is returned to infinitely often but the same choice is made every time. Thus, it is possible that the probabilities $p_s^{min}(\neg b \,\mathcal{U}\, a)$ may be different under fair adversaries only. In fact, this *is* the case, as we now confirm by model checking.

Following the method outlined earlier, we begin by constructing the dual problem. We get $S^{+} = \{0,1,2\}$ and $S^{\#} = \{0,1\}$. Thus, we must now compute the probabilities $p_s^{max}(a^{\#} \,\mathcal{U}\, \neg a^{+})$ where $a^{\#}$ is satisfied in states 0 and 1, and $\neg a^{+}$ in state 3. Trivially, $p_3^{max}(a^{\#} \,\mathcal{U}\, \neg a^{+}) = 1$ and $p_2^{max}(a^{\#} \,\mathcal{U}\, \neg a^{+}) = 0$. The algorithms Prob0A and Prob1E identify no further states so $S^{yes} = \{3\}$, $S^{no} = \{2\}$ and $S^{?} = \{0,1\}$. The corresponding linear optimisation problem is "Minimise $x_0 + x_1$ such that $x_0 \geq x_1$, $x_1 \geq x_0$ and $x_1 \geq 0.4$"

which yields the solution $(x_0, x_1) = (0.4, 0.4)$. Subtracting the resulting probabilities $(0.4, 0.4, 0, 1)$ from 1, we get finally get that the values $p_s^{min}(\neg b \,\mathcal{U}\, a)$ are $(0.6, 0.6, 1, 0)$ and $\mathcal{P}_{\geq 0.5}[\neg b \,\mathcal{U}\, a]$ is satisfied in states 0, 1 and 2.

3.3.3 CSL Model Checking of CTMCs

In this section, we consider the computation of probabilities for the CSL next, time-bounded until, until and steady-state operators. CSL model checking was shown to be decidable (for rational time-bounds) in [ASSB96] and a model checking algorithm first presented in [BKH99]. We use these techniques plus the subsequent improvements made in [BHHK00a] and [KKNP01]. The CSL next and until operators depend only on the probability of moving from one state to another, not the time at which this occurs. Hence, in both cases, model checking can be performed on the embedded DTMC, proceeding as in Section 3.3.1 above. We consider the remaining two operators below.

CSL Time-Bounded Until

For this operator, we need to determine the probabilities $p_s(\phi_1 \,\mathcal{U}^{\leq t}\, \phi_2)$ for all states s. As in previous cases, the probability for states which satisfy ϕ_2 is trivially 1. We reuse the Prob0 algorithm from Figure 3.1 above to determine the set of states which cannot possibly reach a state satisfying ϕ_2 passing only through states satisfying ϕ_1. We denote this set of states S^{no}. The probabilities for the remaining states must be computed numerically. Originally, [BKH99] proposed to do this via approximate solution of Volterra integral equation systems. Experiments in [HKMKS00] showed that this method was generally slow and, in [BHHK00a], a simpler alternative was presented which reduces the problem to transient analysis. This is a well studied problem in performance modelling, for which efficient algorithms have been developed.

The basic idea is to modify the CTMC, removing all outgoing transitions from states that either satisfy ϕ_2 or are contained in S^{no}. Since a path in the new CTMC cannot exit a state satisfying ϕ_2 once it reaches one, and will never be able to reach a state satisfying ϕ_2 if it enters a state from S^{no}, the probability $p_s(\phi_1 \,\mathcal{U}^{\leq t}\, \phi_2)$ in the original CTMC is equal to the probability of being in a state satisfying ϕ_2 at time t in the modified CTMC, i.e. $\sum_{s' \models \phi_2} \pi_{s,t}(s')$. Hence, the problem reduces to computing the transient probabilities $\pi_{s,t}(s')$ for all states s and s' of the new CTMC.

For this, we will use a technique called *uniformisation* (also known as 'randomisation' or 'Jensen's method'). From the rate matrix $\mathbf{Q}$, we compute the *uniformised* DTMC $\mathbf{P}$, given by $\mathbf{P} = \mathbf{I} + \mathbf{Q}/q$ where $q \geq \max_{s \in S} |\mathbf{Q}(s, s)|$. Writing the vector of transient

probabilities $\pi_{s,t}(s')$ over all states $s' \in S$ as $\underline{\pi_{s,t}}$, we have:

$$\underline{\pi_{s,t}} \;=\; \underline{\pi_{s,0}} \cdot \sum_{i=0}^{\infty} \gamma_{i,q\cdot t} \cdot \mathbf{P}^i$$

where $\gamma_{i,q\cdot t}$ is the ith Poisson probability with parameter $q \cdot t$, i.e. $\gamma_{i,q\cdot t} = e^{-q\cdot t} \cdot (q \cdot t)^i / i!$. The vector $\underline{\pi_{s,0}}$ gives the probability of being in each state at time $t = 0$. Since we begin in state s, we have $\underline{\pi_{s,0}}(s')$ equal to 1 if $s' = s$ and 0 otherwise. The weighted sum of powers of $\mathbf{P}$ produces a matrix giving the probability of moving from any state in the CTMC to any other state in t time units.

To compute the transient probabilities numerically, we can truncate the infinite summation above using the techniques of Fox and Glynn [FG88]. For some desired precision ε, their method produces lower and upper bounds, L_ε and R_ε. Summing over these bounds is then sufficient to compute the transient probabilities to within the specified precision.

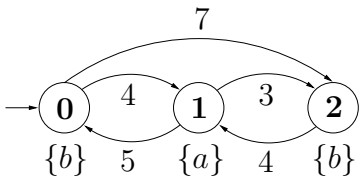
As shown in [KKNP01], uniformisation can be adapted to compute the probabilities for the CSL time-bounded until operator. Let $\underline{p(\phi_1\ \mathcal{U}^{\leq t}\ \phi_2)}$ be the required vector of probabilities $p_s(\phi_1\ \mathcal{U}^{\leq t}\ \phi_2)$ for each state s. Assuming that $\underline{\phi_2}$ is a vector with $\underline{\phi_2}(s)$ equal to 1 if $s \models \phi_2$ and 0 otherwise, and truncating the infinite summation as described:

$$\begin{aligned}\underline{p(\phi_1\ \mathcal{U}^{\leq t}\ \phi_2)} \;&=\; \left(\sum_{i=L_\varepsilon}^{R_\varepsilon} \gamma_{i,q\cdot t} \cdot \mathbf{P}^i\right) \cdot \underline{\phi_2} \\ &=\; \sum_{i=L_\varepsilon}^{R_\varepsilon} \left(\gamma_{i,q\cdot t} \cdot \mathbf{P}^i \cdot \underline{\phi_2}\right)\end{aligned}$$

Note that the inclusion of the vector $\underline{\phi_2}$ within the brackets is vital since it allows explicit computation of the matrix powers $\mathbf{P}^i$ to be avoided. Instead, each product $\mathbf{P}^i \cdot \underline{\phi_2}$ is calculated as $\mathbf{P} \cdot (\mathbf{P}^{i-1} \cdot \underline{\phi_2})$, reusing the computation from the previous iteration. The bulk of the work required thus reduces to R_ε matrix-vector multiplications.

CSL Steady-State

A state s satisfies the steady-state formula $\mathcal{S}_{\bowtie p}[\phi]$ if $\sum_{s' \models \phi} \pi_s(s') \bowtie p$. Therefore, to model check the $\mathcal{S}_{\bowtie p}[\phi]$ operator, we must compute the steady-state probabilities $\pi_s(s')$ for all states s and s'. However, since we only consider the case of ergodic CTMCs, the steady-state probabilities are in fact independent of the initial state s. Hence, we proceed by computing the unique steady-state probability distribution of the CTMC and summing over states satisfying ϕ. Depending on the bound $\bowtie p$, the formula $\mathcal{S}_{\bowtie p}[\phi]$ is then satisfied in either *all* states or *no* states. We denote the vector of steady-state probabilities for



$$\mathbf{R} = \begin{pmatrix} 0 & 4 & 7 \\ 5 & 0 & 3 \\ 0 & 4 & 0 \end{pmatrix} \qquad \mathbf{Q} = \begin{pmatrix} -11 & 4 & 7 \\ 5 & -8 & 3 \\ 0 & 4 & -4 \end{pmatrix}$$

Figure 3.8: A 3 state CTMC with its transition rate matrix and generator matrix

all states as $\underline{\pi}$. As shown in any standard text on CTMCs, such as [Ste94], this can be computed by solving the linear equation system:

$$\underline{\pi} \cdot \mathbf{Q} = \underline{0} \quad \text{and} \quad \textstyle\sum_{s' \in S} \pi(s') = 1$$

This system can be transposed as $\mathbf{Q}^T \cdot \underline{\pi}^T = \underline{0}$ and solved using standard methods, as for the PCTL until operator over DTMCs. Again, we will opt to solve this using iterative, rather than direct, methods. These are discussed in Section 3.5.

Example

We conclude this section with a simple example of CSL model checking. Figure 3.8 shows a 3 state ergodic CTMC. We use exactly the same graphical notation as for DTMCs, the only difference being that the transitions are now labelled with rates rather than probabilities. We consider the CSL formula $\mathcal{S}_{\geq 0.6}[b]$. This requires us to calculate the steady-state probability $\pi(s')$ for each state $s' \in \{0, 1, 2\}$. Abbreviating $\pi(s')$ to $x_{s'}$, we can determine these from the linear equation system:

$$\begin{aligned} -11\,x_0 + 5\,x_1 &= 0 \\ 4\,x_0 - 8\,x_1 + 4\,x_2 &= 0 \\ 7\,x_0 + 3\,x_1 - 4\,x_2 &= 0 \\ x_0 + x_1 + x_2 &= 1 \end{aligned}$$

which has solution $(x_0, x_1, x_2) = (\frac{5}{33}, \frac{1}{3}, \frac{17}{33})$. Since states 0 and 2 satisfy b, we compute $\sum_{s' \models b} \pi(s') = \frac{5}{33} + \frac{17}{33} = \frac{2}{3} \geq 0.6$ and hence the formula $\mathcal{S}_{\geq 0.6}[b]$ is satisfied in all states.

3.4 The PRISM Language

In this section, we describe how the probabilistic models from the previous sections are specified in practice using PRISM, the probabilistic model checker we have developed to implement the techniques in this thesis. Typically, real life applications will result in

models with a huge number of states. Clearly, it is impractical for the user to explicitly list every state and transition of these models. The solution is to provide a high-level specification formalism to describe them in a meaningful fashion. Examples for probabilistic models include stochastic process algebras and stochastic Petri nets.

PRISM has its own system description language for this purpose, based on the Reactive Modules formalism of [AH99]. There were two main reasons for this choice. Firstly, it provides a simple, consistent and intuitive way of specifying all three of the types of model we consider: DTMCs, MDPs and CTMCs. Secondly, the language is well suited to translation into MTBDDs, providing an ideal basis for experimentation into symbolic probabilistic model checking. In this section, we explain the basic ideas behind the language and give a number of illustrative examples. A formal specification of its syntax and semantics can be found in Appendix B.

3.4.1 Fundamentals

The two basic elements of the PRISM language are *modules* and *variables*. A model is defined as the parallel composition of several interacting modules. Each module has a set of integer-valued, *local variables* with finite range. We will often refer to these as *PRISM variables*. The *local state* of a module at a particular time is given by a valuation of its local variables. A *global state* of the whole model is a valuation of the variables for all modules.

A module makes a transition from one local state to another by changing the value of its local variables. A transition of the whole model from one global state to another comprises transitions for one or more of its component modules. This can either be *asynchronous*, where a single module makes a transition independently, the others remaining in their current state, or *synchronous*, where two or more modules make a transition simultaneously.

The behaviour of each module, i.e. the transitions it can make in any given state, are defined by a set of *commands*. Each command consists of a *guard*, which identifies a subset of the global state space, and one or more *updates*, each of which corresponds to a possible transition of the module. Intuitively, if the model is in a state satisfying the guard of a command then the module can make the transitions defined by the updates of that command. The probability that each transition will be taken is also specified by the command. The precise nature of this information depends on the type of model being described. This will be clarified in the following sections through a number of small examples.

dtmc

module M
v : [0..3] **init** 0;
[] $(v = 0) \rightarrow (v' = 1)$;
[] $(v = 1) \rightarrow 0.5 : (v' = 0) + 0.3 : (v' = 2) + 0.2 : (v' = 3)$;
[] $(v = 2) \rightarrow (v' = 2)$;
[] $(v = 3) \rightarrow (v' = 3)$;
endmodule

Figure 3.9: The PRISM language: Example 1

3.4.2 Example 1

As a first example, we consider a description of the 4 state DTMC from Figure 3.3. This is shown in Figure 3.9. The first line identifies the model type, in this case a DTMC. The remaining lines define the modules which make up the model. For this simple example, only a single module M is required.

The first part of a module definition gives its set of local variables, identifying the name, range and initial value of each one. In this case, we have a single variable v with range [0..3] and initial value 0. Hence, the local state space of module M, and indeed the global state space of the whole DTMC, is $\{0, 1, 2, 3\}$.

The second part of a module definition gives its set of commands. Each one takes the form "[] $g \rightarrow u$;", where g is the guard and u lists one or more updates. A guard is a predicate over all the variables of the model (in this case, just v). Since each state of the model is associated with a valuation of these variables, a guard defines a subset of the model's state space. The updates specify transitions that the module can make. These are expressed in terms of how the values of the local variables would change if the transition occurred. In our notation, v' denotes the updated value of v, so "$v' = 1$" implies simply that v's value will change to 1.

Since the model is a DTMC, where more than one possible transition is listed, the likelihood of each being taken is given by a discrete probability distribution. The second command of M shows an example of this. There is a clear correspondence between this probability distribution and the one in state 1 of the DTMC in Figure 3.3. When there is only a single possible transition, we assume an implicit probability distribution which selects this transition with probability 1. This information can be omitted from the description, as is done in the first, third and fourth commands of M. Note that, in order to ensure that the model described is a DTMC, we require that the guards of the module

are disjoint.

Since this DTMC is the one which we used for our PCTL model checking example in Section 3.3.1, we take this opportunity to illustrate how PCTL (and CSL) formulas are specified in PRISM. The only important issue to consider is how atomic propositions, the basic building blocks of the formulas, are expressed. The approach we take is to write them as predicates over PRISM variables. For example, in Section 3.3.1, we illustrated model checking of the PCTL formula $\mathcal{P}_{\geq 0.5}[\neg\, b\, \mathcal{U}\, a]$. Since a is satisfied only in state 2 and b only in state 3, the formula becomes $\mathcal{P}_{\geq 0.5}[\neg\, (v = 3)\, \mathcal{U}\, (v = 2)]$.

This small example is slightly misleading since it gives the impression that, to include an atomic proposition in a formula, we have to determine all the states satisfying it and then encode them as a predicate over PRISM variables. In more realistic examples, however, atomic propositions are used to label states with some property of interest. Since it is the PRISM variables themselves which determine the 'meaning' of a state, it is intuitive to describe atomic propositions in this way.

3.4.3 Example 2

Our second example, shown in Figure 3.10, describes an MDP. This is a slightly larger model than Example 1 and illustrates some additional features of the PRISM language, in particular demonstrating how several modules can be composed in parallel.

The model comprises two identical modules, M_1 and M_2, each with a single local variable, v_1 and v_2, respectively. The basic layout of the definition for each module is as described in Example 1. There are a few further points to note. Firstly, the commands here are more complex: we have used additional operators such as $\leq$, $\geq$ and $\wedge$ in the guards; and the updates contain arithmetic expressions such as "$v_1' = v_1 - 1$". Essentially, though, they both function exactly as before.

Secondly, the examples demonstrate that the guard of a command in one module can reference variables from another module. Note, for example, the presence of variable v_2, belonging to module M_2, in the third and fourth commands of module M_1. This applies only to guards, not updates: a module can only specify how its own local variables can change. Intuitively, this means that a module can read any variable in the model when deciding what transition to perform next, but can only update its own local variables when making the transition.

The complete model is constructed as the parallel composition of its component modules. The state space of the model is the product of the modules' state spaces. In this example, the state space of both M_1 and M_2 is $\{0, \ldots, 10\}$ so the global state space of the model is $\{0, \ldots, 10\} \times \{0, \ldots, 10\}$. By default, the parallel composition is asynchronous,

mdp

module M_1

v_1 : [0..10] **init** 0;

[] $(v_1 = 0) \rightarrow (v_1' = 1)$;

[] $(v_1 \geq 1) \wedge (v_1 \leq 9) \rightarrow 0.5 : (v_1' = v_1 - 1) + 0.5 : (v_1' = v_1 + 1)$;

[] $(v_1 = 10) \wedge (v_2 \leq 5) \rightarrow (v_1' = 10)$;

[] $(v_1 = 10) \wedge (v_2 > 5) \rightarrow (v_1' = 9)$;

endmodule

module M_2

v_2 : [0..10] **init** 0;

[] $(v_2 = 0) \rightarrow (v_2' = 1)$;

[] $(v_2 \geq 1) \wedge (v_2 \leq 9) \rightarrow 0.5 : (v_2' = v_2 - 1) + 0.5 : (v_2' = v_2 + 1)$;

[] $(v_2 = 10) \wedge (v_1 \leq 5) \rightarrow (v_2' = 10)$;

[] $(v_2 = 10) \wedge (v_1 > 5) \rightarrow (v_2' = 9)$;

endmodule

Figure 3.10: The PRISM language: Example 2

meaning that each transition of the composed model corresponds to an independent transition of a single module.

We assume that, in each global state of the model, some form of *scheduling* takes place to decide which module actually makes a transition. For MDPs, the scheduling is nondeterministic. The transitions which an individual module can make in each state of the model are defined by the description of that module. Note that, as in Example 1, there may be more than one possible transition available and, where this is the case, the module defines the probability that each is taken. Hence, each state of the resulting MDP comprises a nondeterministic choice between several probability distributions.

We clarify this using the example. Consider state $(0, 1)$ of the model, i.e. the state where $v_1 = 0$ and $v_2 = 1$. If module M_1 were scheduled, v_1 would change to 1 with probability 1. If module M_2 were scheduled, v_2 would change to either 0 or 2 with equal probability 0.5. Hence, in state $(0, 1)$ of the MDP, we have a nondeterministic choice between a probability distribution which selects state $(1, 1)$ with probability 1 or one which selects states $(0, 0)$ and $(0, 2)$ with equal probability 0.5.

In an MDP, scheduling due to parallel composition is not the only possible source of nondeterminism. We also allow *local nondeterminism*, where an individual module can choose between several transitions. Such behaviour is specified through multiple commands which have the overlapping guards. Returning to our example, let us assume

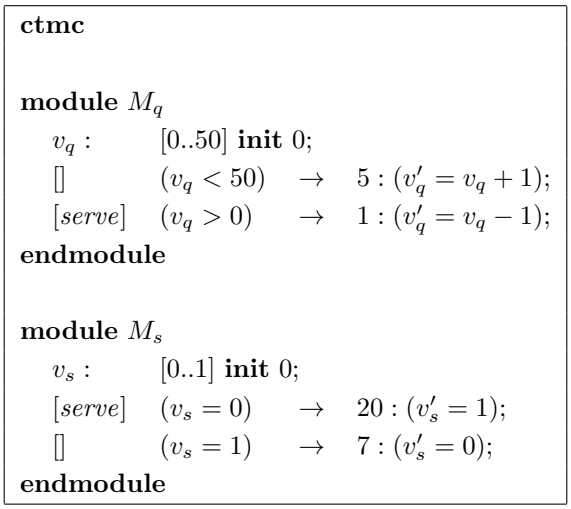


Figure 3.11: The PRISM language: Example 3

that module M_1 had an additional command "[] $(v_1 = 0) \rightarrow (v_1' = 2)$;". Then, the nondeterministic choice in state $(0, 1)$ of the resulting MDP would have, in addition to the two probability distributions described in the previous paragraph, a probability distribution which selects state $(2, 1)$ with probability 1.

Parallel composition of several PRISM modules is also applicable to DTMCs and CTMCs. We deal with the case for CTMCs in the next section. For DTMCs, we assume that the scheduling between modules is probabilistic and that each one is equally likely to be scheduled. This is not a particularly useful model of concurrency, but is included for completeness. In practice, DTMC models typically comprise either only a single module or several modules which are fully synchronised (i.e. move in lock-step).

3.4.4 Example 3

Figure 3.11 shows our third example, which describes a CTMC. This illustrates how CTMCs can provide an alternative to MDPs for modelling concurrent behaviour. It also demonstrates how the PRISM language can be used to model synchronisation between modules. The example comprises two modules, M_q and M_s, which represent a queue of jobs and a server, respectively. Module M_q has a single local variable v_q which represents the number of jobs currently in the queue. Module M_s has a single variable v_s. If $v_s = 0$, the server is free to accept a job; if $v_s = 1$, it is busy processing one.

The basic layout of the model description and the definition of each module is the same as in Examples 1 and 2. There are two main differences. Firstly, the updates of each command are assigned rates, not probabilities (i.e. the values are not necessarily

in the range $[0, 1]$). This is because we are describing a CTMC. Secondly, some of the commands in the modules are labelled with actions, placed in the square brackets at the start of the line. These are used for synchronisation and will be discussed shortly.

Like for an MDP or a DTMC, the parallel composition of several modules in a CTMC model results in the product state space of its components, in this case $\{0, \ldots, 50\} \times \{0, 1\}$. Again, we must consider the scheduling between the modules. In a CTMC, this concurrency is modelled naturally as a race condition. Consider state $(0, 1)$ in our example. Here, M_q can change v_q to 1 and M_s can change v_s to 0. These two transitions are specified as having rate 5 and 7, respectively. Hence state $(0, 1)$ of the CTMC will contain two transitions, one to state $(1, 1)$ with rate 5 and one to state $(0, 0)$ with rate 7. The module which is scheduled will be the one whose transition is enabled first.

In this case, the transitions occur *asynchronously*. However, transitions which correspond to commands labelled with actions take place *synchronously*. Consider, for example, the second command of M_q and the first command of M_s. These are both labelled with the action *serve*. Hence, their transitions will take place simultaneously. For example, in state $(50, 0)$ of the model, M_q can change v_q to 49 and M_s can change v_s to 1. In the CTMC, there will be a single, synchronous transition to state $(49, 1)$. This represents the server removing a job from the queue and beginning to process it.

We define the rate of a synchronous transition to be the product of its component rates. This is a slightly contentious issue and a number of alternative schemes can be found in the literature. Essentially, the source of the problem is that the behaviour of both the individual components and the combined components must be modelled using an exponential distribution. It is not possible, for example to make the time taken for the synchronous transition to occur equal to the maximum of its component transitions, since the maximum of two exponential distributions is not itself an exponential distribution. A good discussion of the various possible approaches and their relative merits and disadvantages can be found in [Hil94].

One common usage of our approach, as illustrated in the example above, is the following. One component is denoted *active* and proceeds with some non-negative rate. The other components are denoted *passive*, proceeding with rate 1. This means that the combined rate is simply that of the active component. The principal advantage of taking the product of the rates is that it provides us with a simple and consistent approach to the semantics and construction of all three of models: synchronisation can be applied in an identical fashion to DTMC and MDP models. Here, the probability of a synchronous transition occurring is equal to the product of the component probabilities. This is an intuitive approach since it equals the probability of all transitions occurring. In all cases, scheduling between a combination of asynchronous and synchronous behaviour

is identical to that of the purely asynchronous case: an equiprobable choice for DTMCs, a nondeterministic choice for MDPs and a race condition for CTMCs.

3.4.5 Additional Considerations

In this section, we have given an informal introduction to the PRISM language. Appendix B provides a formal description, explicitly stating its syntax and semantics. It also covers two topics we have not discussed here: global variables and reachability.

The PRISM language allows a model to contain *global variables* in addition to the local variables belonging to individual modules. A global state of the model is then a valuation for each local variable and each global variable. The values of global variables can be both read *and* modified by any module. This allows us to model shared memory, providing an alternative to synchronisation for inter-module interaction.

Reachability is another important issue. We say that a state s of a model is *reachable* if there exists a finite path starting in the initial state of the model and ending in s (note that this definition applies to all three types of model: DTMCs, MDPs and CTMCs). Since any execution of the model will only ever pass through states which are reachable, we can ignore the states which are not. The state space of a model defined in the PRISM language is the product of the state spaces of its modules, i.e. all possible valuations of all PRISM variables in the model. Typically, only some subset of these will be reachable. Hence, when translating a description in the PRISM language into a probabilistic model, we compute this reachable subset (a process we refer to as reachability) and remove all other states from the model.

3.5 Iterative Solution Methods

As we saw in Section 3.3, model checking for several operators of PCTL and CSL reduces to the solution of a system of linear equations. We can assume that this system is of the traditional form $\mathbf{A} \cdot \underline{x} = \underline{b}$, where $\mathbf{A}$ is a real-valued matrix, $\underline{b}$ is a real-valued vector, and $\underline{x}$ is a vector containing the solution to be determined. Since the systems we solve are derived from model checking problems, where matrices and vectors are indexed over a set of states S, we assume that matrices and vectors are of size $|S| \times |S|$ and $|S|$, respectively, and use the indexing $0, \ldots, |S|-1$.

Solving a system of linear equations is, of course, a well known and much studied problem. Typically, methods for their solution fall into two distinct classes: *direct* methods and *iterative* methods. Direct methods compute the exact solution (within the bounds of numerical error) in a fixed number of steps. Examples include Gaussian elimination

and L/U decomposition. Iterative methods compute successive approximations to the solution, terminating when the sequence of solutions has converged to within some pre-specified accuracy. Examples include the Power, Jacobi and Gauss-Seidel methods.

In this work we will only consider iterative methods. As with many applications, probabilistic model checking typically produces systems where $\mathbf{A}$ is large and sparse. Direct methods are not well suited to solving such systems because of a phenomenon known as *fill-in*. Modifications made to $\mathbf{A}$ as the computation progresses usually increase the number of non-zero elements. For large systems, the resulting growth in required storage space can make the computation infeasible. Fortunately, this is not the case for iterative methods. With the exception of some possible initialisation steps, the matrix is not modified at all during computation, and hence there is no increase in memory consumption.

There are also implications for the data structures used to implement the solution methods. For iterative techniques, we are free to select matrix storage schemes with strengths such as compactness and ease of access: we need not concern ourselves with the efficiency of modifications to the matrix. Conventionally, this argument is applied to sparse matrices, the traditional storage scheme for iterative methods, which we review in Section 3.6. The arguments will also apply, however, to the symbolic data structures, based on MTBDDs, which we introduce in Chapter 6.

In the following paragraphs we describe four of the most common iterative methods for solving systems of linear equations: Jacobi, Gauss Seidel, JOR and SOR. The general procedure for all four methods is as follows. Starting with some initial estimate, each iteration produces an increasingly accurate approximation to the solution of the linear equation system. The approximation computed in the kth iteration is denoted $\underline{x}^{(k)}$. We refer to this as the *solution vector* or *iteration vector*. Using the same notation, we denote the initial estimate as $\underline{x}^{(0)}$.

Each estimate $\underline{x}^{(k)}$ is computed from the previous one $\underline{x}^{(k-1)}$. The iterative process is stopped when the solution vector is judged to have converged sufficiently. The criteria for convergence can vary. One approach is to stop when the maximum difference between elements of consecutive vectors is less than some threshold ε:

$$\max_i \left|\underline{x}^{(k)}(i) - \underline{x}^{(k-1)}(i)\right| \;<\; \varepsilon$$

One potential problem with, though, is that if the solution vector contains very small values (i.e. less than ε), the process may be terminated prematurely. A better approach, and the one we adopt in this thesis, is to check the *relative* difference between elements:

$$\max_i \left(\frac{|\underline{x}^{(k)}(i) - \underline{x}^{(k-1)}(i)|}{|\underline{x}^{(k)}(i)|}\right) \;<\; \varepsilon$$

By default, we will take ε to be 10^{-6}.

3.5.1 The Jacobi Method

The Jacobi method is based on the observation that the ith equation of the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$:

$$\sum_{j=0}^{|S|-1} \mathbf{A}(i,j) \cdot \underline{x}(j) \quad = \quad \underline{b}(i)$$

can be rearranged as:

$$\underline{x}(i) \quad = \quad \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}(j) \right) / \mathbf{A}(i,i)$$

On the basis of this, in the Jacobi method, the ith element of the kth iteration vector is computed from the elements of the $(k-1)$th vector as:

$$\underline{x}^{(k)}(i) \quad := \quad \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j) \right) / \mathbf{A}(i,i)$$

Note that for the systems which arise from PCTL or CSL model checking, the diagonal elements $\mathbf{A}(i,i)$ will always be non-zero. For our purposes, it is also useful to express a single iteration of the Jacobi method in terms of matrices and vectors, rather than individual elements:

$$\underline{x}^{(k)} \quad := \quad \mathbf{D}^{-1} \cdot \left((\mathbf{L} + \mathbf{U}) \cdot \underline{x}^{(k-1)} + \underline{b} \right)$$

where $\mathbf{D} - (\mathbf{L} + \mathbf{U})$ is a partitioning of the matrix $\mathbf{A}$ into its diagonal, lower-triangular, and upper-triangular elements respectively, i.e. $\mathbf{D}$ contains all the diagonal entries of $\mathbf{A}$ and $\mathbf{L} + \mathbf{U}$ contains all the non-diagonal entries, negated. In this setting, the main operation required to perform a single iteration is a matrix-vector multiplication. We will see why this is useful when we implement symbolic versions of these algorithms in Chapters 5 and 6. Note also that the Jacobi method requires two vectors to be stored: one for the previous iteration's vector, and one for the current one.

3.5.2 The Gauss-Seidel Method

The Jacobi method can be improved by observing that the kth update to the ith vector element can actually use the new value $\underline{x}^{(k)}(j)$ rather than the old value $\underline{x}^{(k-1)}(j)$ for

$j < i$. This gives rise to the Gauss-Seidel method:

$$\underline{x}^{(k)}(i) \;:=\; \left(\underline{b}(i) - \sum_{j<i} \mathbf{A}(i,j) \cdot \underline{x}^{(k)}(j) - \sum_{j>i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j)\right) / \mathbf{A}(i,i)$$

This usually converges much faster than the Jacobi method. Furthermore, it only requires a single vector to be stored: after each new vector entry is computed, its old value is no longer required and can be overwritten.

As with the Jacobi method, we can show that one iteration of the Gauss-Seidel method can be performed using matrix-vector multiplication:

$$\underline{x}^{(k)} \;:=\; (\mathbf{D} - \mathbf{L})^{-1} \cdot (\mathbf{U} \cdot \underline{x}^{(k-1)} + \underline{b})$$

where $\mathbf{D}$, $\mathbf{L}$ and $\mathbf{U}$ are as for the Jacobi method above. In this case, though, it may not be a sensible option since the computation of the inverse of $(\mathbf{D} - \mathbf{L})$ may require significant extra work or result in fill-in. Furthermore, with this formulation, we again need to store two vectors rather than one.

3.5.3 Over-Relaxation Methods

Both the Jacobi and Gauss-Seidel methods can be improved with a technique called *over-relaxation*. In an iteration, the new value of each vector element is first computed as described above, and then subjected to a weighted average between itself and the corresponding vector element from the previous iteration. The weights for this average are determined according to a parameter ω. For example, using the Jacobi method as a basis, we get:

$$\underline{x}^{(k)}(i) \;:=\; (1-\omega) \cdot \underline{x}^{(k-1)}(i) \;+\; \omega \cdot \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j)\right) / \mathbf{A}(i,i)$$

This is known as the JOR (Jacobi Over-Relaxation) method. The same technique can be applied to the Gauss-Seidel method, in which case it is referred to as the SOR (Successive Over-Relaxation) method. The idea is that the methods converge faster for a well chosen value of ω. It is known that JOR and SOR will only converge for $\omega \in (0, 2)$. Unfortunately, it is usually either impossible or impractical to compute the optimum value of ω. Typically, one must rely on heuristics to select a good value. The interested reader is referred to, for example [Ste94], for more details. Note also that, technically, for $\omega < 1$, this technique should be referred to as *under-relaxation*, but for convenience it is often still known as over-relaxation.

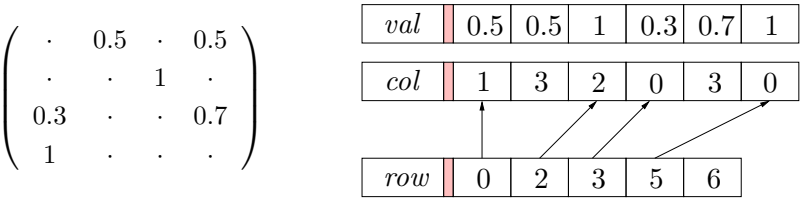


Figure 3.12: A 4×4 matrix and its sparse matrix representation

3.6 Sparse Matrices

Two of the probabilistic models we consider, namely DTMCs and CTMCs, are described by real-valued matrices. These matrices are often very large, but contain a relatively small number of non-zero entries. The transition matrix of a model with 10^6 states, for example, might have as few as 10 non-zero entries in each row. In this thesis, we will be considering *symbolic* data structures for the efficient storage and manipulation of such matrices. However, in order to assess the performance of our techniques, we will need to compare them with implementations based on more conventional, *explicit* storage mechanisms. We will use *sparse matrices*, the standard data structure for this purpose. The basic idea is that only the non-zero entries of the matrix are explicitly stored.

There are several different types of sparse storage scheme. The best one to use depends on what operations will be performed on the matrix. The scheme we will describe here is a common variant, often referred to as a *row-major* sparse matrix. Matrix entries are stored very compactly and are quick and easy to access. The main trade-off is that it is not efficient to make modifications to the matrix. The scheme is well suited to many typical applications of sparse matrices, such as the iterative solution methods we consider. In particular, matrix-vector multiplication, the key operation for such methods, can be performed very efficiently using this sparse matrix scheme.

Assume that we wish to store a real-valued matrix $\mathbf{A}$. As in the previous section, since the matrices we deal with represent probabilistic systems with some state space S, we assume that $\mathbf{A}$ is of size $|S| \times |S|$ and use the indexing $0, \ldots, |S|-1$. We denote the number of non-zero entries in $\mathbf{A}$ by nnz.

The sparse matrix data structure stores information about the row index, column index and value of each matrix entry in three separate arrays, *row*, *col* and *val*. The arrays *val* and *col* each contain nnz elements and store the actual value and column index of each matrix entry, respectively, ordered by row. The third array, *row*, stores indices into the other two arrays: the ith element of *row* points to the start of the entries of row i in *val* and *col*.

```
SparseMVMult(row, col, val, b)
1.  for (i := 0 ... |S|−1)
2.      res[i] := 0
3.      l := row[i]
4.      h := row[i + 1]−1
5.      for (j := l ... h)
6.          res[i] := res[i] + val[j] × b[col[j]]
7.      endfor
8.  endfor
9.  return res
```

Figure 3.13: Multiplication of a sparse matrix with a vector

Figure 3.12 shows an example of a matrix ($|S| = 4$, $nnz = 6$) and its corresponding sparse storage. For clarity, we show all zero entries of the matrix as dots. Both the matrix and the three arrays which store it are indexed from zero. To determine, for example, the entries in row 0 of the matrix, we read the values of $row[0]$ and $row[1]$. Since these are 0 and 2, respectively, we can deduce that row 0 contains two entries and that they are stored in positions 0 and 1 of the other two arrays. Hence, row 0 consists of entries $(0, 1) = 0.5$ and $(0, 3) = 0.5$. Note that *row* contains $|S| + 1$ elements so that the number of entries in the final row can be deduced.

The value of an arbitrary entry (r, c) in the matrix can be determined by looking up the rth and $(r + 1)$th entries in *row* and then checking the column indices $col[row[r]]$ to $col[row[r+1]–1]$ for the value c. If the value c is not present, then $(r, c) = 0$. If it is present, then (r, c) is non-zero and its value can be found by looking up the value at the same position in *val*. In fact, for our purposes, we never require access to a single, random entry, but access to all the entries at once, usually to perform a matrix-vector multiplication. The algorithm to perform such a multiplication, $\mathbf{A} \cdot \underline{b}$, is given in Figure 3.13. The matrix $\mathbf{A}$ is stored in the arrays *row*, *col* and *val*, as described above. The vector $\underline{b}$ and the vector representing the result $\mathbf{A} \cdot \underline{b}$ are stored in arrays b and *res*, respectively.

We will assume that matrix entries are stored as 8 byte doubles and indices as 4 byte integers. This puts an upper limit on *nnz* of 2^{32} ($\approx 4.3 \times 10^9$). Given the storage of a typical workstation, this is ample. Using these figures, the amount of memory required to store a sparse matrix is $12nnz + 4(|S| + 1)$ bytes.

Note that it is also possible to order the entries in the sparse matrix by column, reversing the roles of the *row* and *col* arrays. This is known as a *column-major* scheme. In terms of storage, row-major and column-major schemes are identical. The principal difference in practice is that the former is better suited to matrix-vector multiplication

($\mathbf{A} \cdot \underline{b}$) and the latter is better suited to vector-matrix multiplication ($\underline{b} \cdot \mathbf{A}$). Depending on the particular model checking algorithm being implemented, we may use either.

3.6.1 An Extension for MDPs

In this thesis, we will also be developing symbolic techniques for the analysis of MDPs. We will again wish to compare the performance of these methods with explicit alternatives. A comparison of this type is slightly more problematic than with DTMCs or CTMCs since there is no standard storage scheme for MDPs.

In fact, our requirements for MDP storage are slightly unusual. As described in Section 3.1.2, we consider the choice between different possible behaviours in each state of an MDP as being nondeterministic. We do not attempt to distinguish between these choices in any way. By contrast, in most other applications they are labelled with actions. An analysis of the MDP in this situation might entail, for example, determining the sequence of actions which maximises the probability of some event occurring. In our case, checking a typical PCTL formula would involve simply computing the maximum probability that the event occurs and verifying that it does not exceed some specified bound, not determining the actions.

Because of this, we can opt for a fairly simple data structure to store MDPs. We will use a trivial extension of sparse matrices. As discussed previously, we can think of an MDP as being represented by a matrix, but with each state described by several rows, rather than a single row as for DTMCs and CTMCs. Again, we store the value and column index of every matrix entry in two arrays *val* and *col*. This time, however, we require two levels of indexing to associate each entry with both a state of the MDP and a nondeterministic choice in that state. We name these arrays *row* and *nc*, respectively.

The idea is illustrated most clearly through a simple example. Figure 3.14 gives the matrix representation for a 4 state MDP and the sparse matrix based data structure which stores it. Again, we only show non-zero entries in the matrix. To determine the entries corresponding to state 0, we first read the values of $row[0]$ and $row[1]$. This reveals that state 0 contains two nondeterministic choices and that these have index 0 and 1 in the array *nc*. From $nc[0]$ and $nc[1]$, we see that the first nondeterministic choice comprises a single entry $(0, 1) = 1$. Likewise, from $nc[1]$ and $nc[2]$, we can deduce that the second choice comprises two entries: $(0, 1) = 0.5$ and $(0, 2) = 0.5$.

From the algorithms for PCTL model checking of MDPs in Section 3.3.2, we see that the key operation required is a combined matrix-vector multiplication and maximum or minimum operation. This can be implemented on the sparse data structure in a very similar way to ordinary matrix-vector multiplication. The algorithm for the 'maximum'

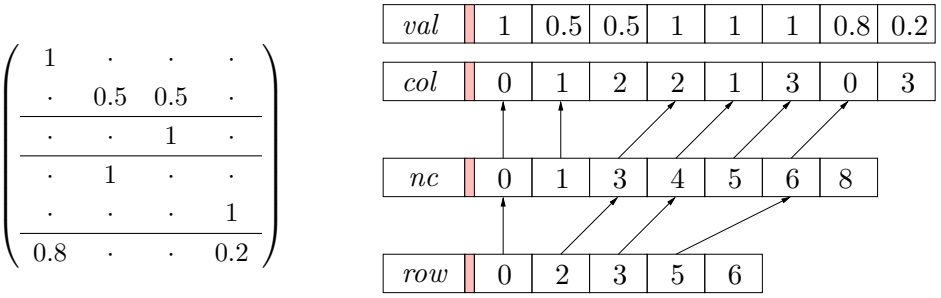


Figure 3.14: A 4 state MDP and its sparse storage

```
SparseMVMultMax(row, nc, col, val, b)
 1.  for (i := 0 ... |S|−1)
 2.      l_i := row[i]
 3.      h_i := row[i + 1]−1
 4.      res[i] := −1
 5.      for (j := l_i ... h_i)
 6.          l_j := nc[j]
 7.          h_j := nc[j + 1]−1
 8.          sum := 0
 9.          for (k := l_j ... h_j)
10.              sum := sum + val[k] × b[col[k]]
11.          endfor
12.          if (sum > res[i]) then res[i] := sum
13.      endfor
14.  endfor
15.  return res
```

Figure 3.15: Combined multiplication and maximum operation for an MDP

case is given in Figure 3.15. It takes an MDP, stored in arrays *row*, *nc*, *col* and *val*, a vector stored in an array b and returns the result in an array *res*. The corresponding algorithm to determine the minimum is almost identical.

Again storing matrix entries as 8 byte doubles and indices as 4 byte integers, the amount of memory required to represent an MDP with this scheme can be computed as $4(|S|+1)+4(nnc+1)+12nnz$ bytes, where $|S|$ is the number of states, nnz is the number of non-zero entries and nnc is the total number of nondeterministic choices over all states, i.e. $nnc = \sum_{s \in S} |Steps(s)|$.

3.7 Multi-Terminal Binary Decision Diagrams

We now introduce *multi-terminal binary decision diagrams* (MTBDDs), the data structure upon which much of the work in this thesis is based. MTBDDs are actually an extension of *binary decision diagrams* (BDDs), originally created by Lee [Lee59] and Akers [Ake78], popularised by Bryant [Bry86], and applied to model checking by McMillan, Clarke and others [BCM+90, McM93]. A BDD is a rooted, directed acyclic graph which represents a Boolean function of the form $f : \mathbb{B}^n \to \mathbb{B}$.

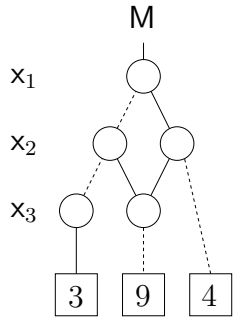
MTBDDs were first proposed in [CMZ+93] and then developed independently in [CFM+93] and [BFG+93]. In [BFG+93], they were christened algebraic decision diagrams (ADDs) but the two data structures are identical. MTBDDs extend BDDs by representing functions which can take values from an arbitrary set D, not just $\mathbb{B}$, i.e. functions of the form $f : \mathbb{B}^n \to D$. In the majority of cases, D is taken to be $\mathbb{R}$ and this is the policy we adopt here. Note that BDDs are in fact a special case of MTBDDs.

Let $\{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ be a set of distinct, Boolean variables which are totally ordered as follows: $\mathsf{x}_1 < \cdots < \mathsf{x}_n$. An MTBDD M over $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ is a rooted, directed acyclic graph. The vertices of the graph are known as *nodes*. Each node of the MTBDD is classed as either *non-terminal* or *terminal*. A non-terminal node m is labelled with a variable $\mathit{var}(\mathsf{m}) \in \underline{\mathsf{x}}$ and has exactly two children, denoted $\mathit{then}(\mathsf{m})$ and $\mathit{else}(\mathsf{m})$. A terminal node m is labelled by a real number $\mathit{val}(\mathsf{m})$ and has no children. We will often refer to terminal and non-terminal nodes simply as *terminals* and *non-terminals*, respectively.

The ordering $<$ over the Boolean variables is imposed upon the nodes of the MTBDD. For two non-terminals, m_1 and m_2, if $\mathit{var}(\mathsf{m}_1) < \mathit{var}(\mathsf{m}_2)$, then $\mathsf{m}_1 < \mathsf{m}_2$. If m_1 is a non-terminal and m_2 is a terminal, then $\mathsf{m}_1 < \mathsf{m}_2$. We require that, for every non-terminal m in an MTBDD, $\mathsf{m} < \mathit{else}(\mathsf{m})$ and $\mathsf{m} < \mathit{then}(\mathsf{m})$.

Figure 3.16(a) shows an example of an MTBDD. The nodes are arranged in horizontal levels, one per Boolean variable. The variable $\mathit{var}(\mathsf{m})$ for a node m can be found at the left end of the level which contains it. The two children of a node m are connected to it by edges, a solid line for $\mathit{then}(\mathsf{m})$ and a dashed line for $\mathit{else}(\mathsf{m})$. Terminals are drawn as squares, instead of circles, and are labelled with their value $\mathit{val}(\mathsf{m})$. For clarity, we omit the terminal with value 0 and any edges which lead directly to it.

An MTBDD M over variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ represents a function $f_{\mathsf{M}}(\mathsf{x}_1, \ldots, \mathsf{x}_n) : \mathbb{B}^n \to \mathbb{R}$. The value of $f_{\mathsf{M}}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$ is determined by tracing a path in M from the root node to a terminal, for each non-terminal m, taking the edge to $\mathit{then}(\mathsf{m})$ if $\mathit{var}(\mathsf{m})$ is 1 or $\mathit{else}(\mathsf{m})$ if $\mathit{var}(\mathsf{m})$ is 0. The function represented by the MTBDD in Figure 3.16(a) is shown in Figure 3.16(b). For example, the value of $f_{\mathsf{M}}(0, 1, 0)$ can be read as 9. We also use the notation $f_{\mathsf{M}}[\mathsf{x}_1 = 0, \mathsf{x}_2 = 1, \mathsf{x}_3 = 0]$ or $f_{\mathsf{M}}[\underline{\mathsf{x}} = (0, 1, 0)]$. This is often convenient



(a)

(b)

x_1	x_2	x_3	f_M
0	0	1	3
0	1	0	9
1	1	0	9
1	0	0	4
1	0	1	4
otherwise			0

Figure 3.16: An MTBDD M and the function it represents

when we wish to express the value of an MTBDD without explicit reference to the ordering of its Boolean variables.

We define a *minterm* of an MTBDD as a valuation of its Boolean variables which results in a non-zero value. The MTBDD in Figure 3.16, for example, has 5 minterms, each corresponding to a row in the table. Note that there is not necessarily a one-to-one mapping between paths in the MTBDD and minterms. For example, the path to the 4 terminal in Figure 3.16(a) corresponds to two different minterms.

For an MTBDD M, over variables $(\mathsf{x}_1, \ldots, \mathsf{x}_n)$, we define its *cofactor* $\mathsf{M}|_{\mathsf{x}_i=b}$ (where $b = 0$ or 1) as the MTBDD over variables $(\mathsf{x}_1, \ldots, \mathsf{x}_{i-1}, \mathsf{x}_{i+1}, \ldots, \mathsf{x}_n)$ representing the function $f_{\mathsf{M}}(\mathsf{x}_1, \ldots, \mathsf{x}_{i-1}, b, \mathsf{x}_{i+1}, \ldots, \mathsf{x}_n)$. Conversely, the function f_{M} for an MTBDD M, with top variable x_1, can be evaluated recursively using its cofactors:

$$f_{\mathsf{M}} = \mathsf{x}_1 \cdot f_{\mathsf{M}|_{\mathsf{x}_1=1}} + (1 - \mathsf{x}_1) \cdot f_{\mathsf{M}|_{\mathsf{x}_1=0}}$$

Note that, because of the recursive nature of the data structure, each node of an MTBDD is itself an MTBDD. Hence, we will often consider an MTBDD and the node which represents it (i.e. its root node) to be interchangeable.

The reason that MTBDDs can often provide compact storage is because they are stored in a reduced form. Firstly, if nodes m and m' are identical (i.e. $\mathit{var}(\mathsf{m}) = \mathit{var}(\mathsf{m}')$, $\mathit{then}(\mathsf{m}) = \mathit{then}(\mathsf{m}')$ and $\mathit{else}(\mathsf{m}) = \mathit{else}(\mathsf{m}')$), only one copy of the node is stored. We refer to this as *sharing* of nodes. Secondly, if a node m satisfies $\mathit{then}(\mathsf{m}) = \mathit{else}(\mathsf{m})$, it is removed and any incoming edges from levels above are redirected to its unique child. We will refer to this as a *skipped level*. Figure 3.17 illustrates the way that the MTBDD in Figure 3.16 has been reduced: (a) shows the full binary tree representing the same function; in (b), shared nodes have been merged; and in (c), nodes with identical children have also been removed, introducing skipped levels.

In this thesis, unless stated otherwise, we assume that all MTBDDs are fully reduced in this way. Under this assumption, and for a fixed ordering of Boolean variables, the data

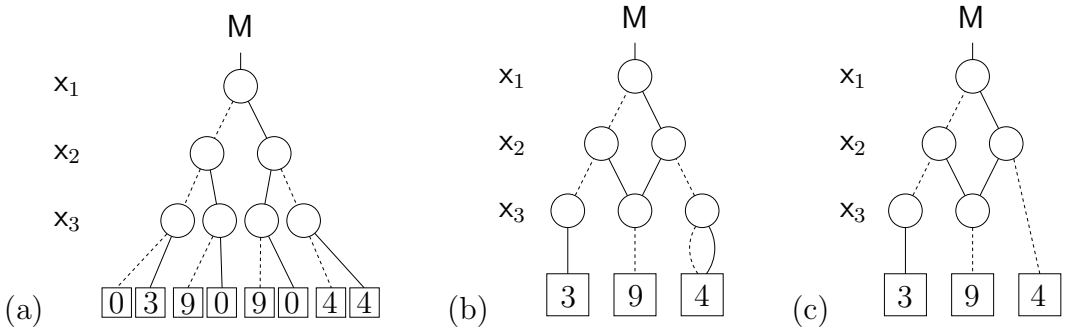


Figure 3.17: Reducing an MTBDD

structure can be shown to be *canonical*, meaning that there is a one-to-one correspondence between MTBDDs and the functions they represent. As we will see later, this canonicity property has crucial implications for performance.

Another important consideration from a practical point of view is *variable ordering*. The size of an MTBDD (i.e. number of nodes) representing a given function is extremely sensitive to the ordering of its Boolean variables. This has a direct effect on both the storage requirements for the data structure and the time needed to perform operations on it. We will consider this topic in more depth in Chapter 4.

The size of an MTBDD is also affected by the number of terminals it contains, or equivalently, the number of distinct values taken by the function which it represents. Compact MTBDDs can only be obtained by maximising sharing within the nodes of the graph. A high number of terminals reduces the capacity for sharing and so increases the number of nodes in the MTBDD.

3.7.1 Operations

Described below are all of the BDD and MTBDD operations needed for the model checking algorithms discussed in this thesis. We treat BDDs simply as a special case of MTBDDs. In the following, we assume that M, M_1 and M_2 are MTBDDs over the set of variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$.

- Const(c), where $c \in \mathbb{R}$, creates a new MTBDD with the constant value c, i.e. a single terminal m, with $val(\mathsf{m}) = c$.

- Apply($op, \mathsf{M}_1, \mathsf{M}_2$), where op is a binary operation over the reals (e.g. $+$, $-$, $\times$, $\div$, min, max, etc.), returns the MTBDD representing the function f_{M_1} op f_{M_2}. If M_1 and M_2 are BDDs, op can also be a Boolean operation ($\wedge$, $\vee$, $\rightarrow$, etc.). For

clarity, we allow Apply operations to be expressed in infix notation, e.g. $\mathsf{M}_1 \times \mathsf{M}_2 = \textsc{Apply}(\times, \mathsf{M}_1, \mathsf{M}_2)$ and $\mathsf{M}_1 \wedge \mathsf{M}_2 = \textsc{Apply}(\wedge, \mathsf{M}_1, \mathsf{M}_2)$.

- Not(M), where M is a BDD, returns the BDD representing the function $\neg f_{\mathsf{M}}$. As above, we may abbreviate Not(M) to $\neg \mathsf{M}$.

- Abs(M) returns the MTBDD representing the function $|f_{\mathsf{M}}|$, giving the absolute value of the original one.

- Threshold($\mathsf{M}, \bowtie, c$), where $\bowtie$ is a relational operator ($<$, $>$, $\leq$, $\geq$, etc.) and $c \in \mathbb{R}$, returns the BDD M' with $f_{\mathsf{M}'}$ equal to 1 if $f_{\mathsf{M}} \bowtie c$ and 0 otherwise.

- FindMin(M) returns the real constant equal to the minimum value of f_{M}.

- FindMax(M) returns the real constant equal to the maximum value of f_{M}.

- Abstract($op, \underline{\mathsf{x}}, \mathsf{M}$), where op is a commutative and associative binary operation over the reals, returns the result of abstracting all the variables in $\underline{\mathsf{x}}$ from M by applying op over all possible values taken by the variables. For example, $\textsc{Abstract}(+, (\mathsf{x}_1), \mathsf{M})$ would give the MTBDD representing the function $f_{\mathsf{M}|_{\mathsf{x}_1=0}} + f_{\mathsf{M}|_{\mathsf{x}_1=1}}$ and $\textsc{Abstract}(\times, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{M})$ would give the MTBDD representing the function $f_{\mathsf{M}|_{\mathsf{x}_1=0,\mathsf{x}_2=0}} \times f_{\mathsf{M}|_{\mathsf{x}_1=0,\mathsf{x}_2=1}} \times f_{\mathsf{M}|_{\mathsf{x}_1=1,\mathsf{x}_2=0}} \times f_{\mathsf{M}|_{\mathsf{x}_1=1,\mathsf{x}_2=1}}$. In the latter, $\mathsf{M}|_{\mathsf{x}_1=b_1,\mathsf{x}_2=b_2}$ is equivalent to $(\mathsf{M}|_{\mathsf{x}_1=b_1})|_{\mathsf{x}_2=b_2}$.

- ThereExists($\underline{\mathsf{x}}, \mathsf{M}$), where M is a BDD, is equivalent to $\textsc{Abstract}(\vee, \underline{\mathsf{x}}, \mathsf{M})$.

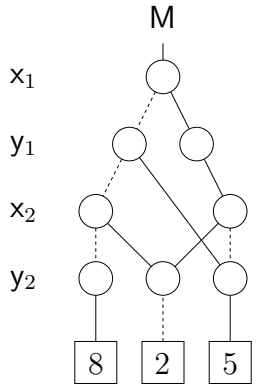
- ForAll($\underline{\mathsf{x}}, \mathsf{M}$), where M is a BDD, is equivalent to $\textsc{Abstract}(\wedge, \underline{\mathsf{x}}, \mathsf{M})$.

- ReplaceVars($\mathsf{M}, \underline{\mathsf{x}}, \underline{\mathsf{y}}$), where $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$, returns the MTBDD M' over variables $\underline{\mathsf{y}}$ with $f_{\mathsf{M}'}(b_1, \ldots, b_n) = f_{\mathsf{M}}(b_1, \ldots, b_n)$ for all $(b_1, \ldots, b_n) \in \mathbb{B}^n$.

3.7.2 Vectors and Matrices

From our point of view, one of the most interesting applications of MTBDDs is the representation of vectors and matrices. This was investigated in the original work which introduced the data structure [CFM+93, BFG+93]. Consider a real-valued vector $\underline{v}$ of length 2^n. We can think of $\underline{v}$ as a mapping from indices to reals, i.e. $\underline{v} : \{0, \ldots, 2^n-1\} \rightarrow \mathbb{R}$. Given an encoding of the 2^n indices into n Boolean variables, i.e. a bijection $enc : \{0, \ldots, 2^n-1\} \rightarrow \mathbb{B}^n$, we can represent $\underline{v}$ as an MTBDD v over variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$. We say that v represents $\underline{v}$ if and only if $f_{\mathsf{v}}[\underline{\mathsf{x}} = enc(i)] = \underline{v}(i)$ for $0 \leq i \leq 2^n-1$.

$$\mathbf{M} = \begin{pmatrix} 0 & 8 & 0 & 5 \\ 2 & 0 & 0 & 5 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 \end{pmatrix}$$



x_1	y_1	x_2	y_2	f_{M}	Entry in $\mathbf{M}$
0	0	0	1	8	$(0,1) = 8$
0	0	1	0	2	$(1,0) = 2$
0	1	0	1	5	$(0,3) = 5$
0	1	1	1	5	$(1,3) = 5$
1	1	1	0	2	$(3,2) = 2$
1	1	0	1	5	$(2,3) = 5$

Figure 3.18: A matrix $\mathbf{M}$ and an MTBDD M representing it

This idea extends easily to matrices. We can think of a 2^n by 2^n matrix, $\mathbf{M}$, as a mapping from $\{0,\ldots,2^n-1\} \times \{0,\ldots,2^n-1\}$ to $\mathbb{R}$. Again, assuming an encoding $enc : \{0,\ldots,2^n-1\} \to \mathbb{B}^n$, we can represent $\mathbf{M}$ by an MTBDD M over $2n$ variables, n of which encode row indices and n of which encode column indices. Using *row variables* $\underline{\mathsf{x}} = (\mathsf{x}_1,\ldots,\mathsf{x}_n)$ and *column variables* $\underline{\mathsf{y}} = (\mathsf{y}_1,\ldots,\mathsf{y}_n)$, we say that M represents $\mathbf{M}$ if and only if $f_{\mathsf{M}}[\underline{\mathsf{x}} = enc(i), \underline{\mathsf{y}} = enc(j)] = \mathbf{M}(i,j)$ for $0 \le i,j \le 2^n-1$. Note that we can also encode vectors and matrices with sizes which are not powers of two simply by padding them with extra elements, rows or columns, all set to 0.

Figure 3.18 shows an example: a matrix $\mathbf{M}$ and an MTBDD M representing it. The table on the right hand side illustrates how the matrix entries can be read off the MTBDD. We are representing a 4×4 matrix so we use two row variables $(\mathsf{x}_1, \mathsf{x}_2)$ and two column variables $(\mathsf{y}_1, \mathsf{y}_2)$. We use the standard binary encoding for integers, i.e. $0 \to 00$, $1 \to 01$, $2 \to 10$ and $3 \to 11$. Each minterm of the MTBDD represents a non-zero entry in the matrix. Consider, for example, the minterm $f_{\mathsf{M}}(1,1,0,1) = 5$. In this case, $\mathsf{x}_1 = 1$ and $\mathsf{x}_2 = 0$, so the row index is 2. Similarly, $\mathsf{y}_1 = 1$ and $\mathsf{y}_2 = 1$, giving a column index of 3. This corresponds to the matrix entry $(2,3) = 5$.

Observe that, in Figure 3.18, row and column variables are ordered alternately. This is a well known heuristic to minimise MTBDD size and will be discussed in Chapter 4. Another advantage of the ordering, though, is that it allows a convenient division of the matrix into its component submatrices. Figure 3.19 illustrates this. If the matrix $\mathbf{M}$ is represented by MTBDD M, the top and bottom halves of $\mathbf{M}$ are represented by the cofactor MTBDDs $\mathsf{M}|_{\mathsf{x}_1=0}$ and $\mathsf{M}|_{\mathsf{x}_1=1}$, respectively. Since x_1 is the top variable in the ordering, these cofactors can be found simply by following the *else* and *then* edges. By repeating this process on the next variable in the ordering, y_1, the four quadrants of $\mathbf{M}$ can be obtained. As well as providing convenient access to submatrices, this can be beneficial

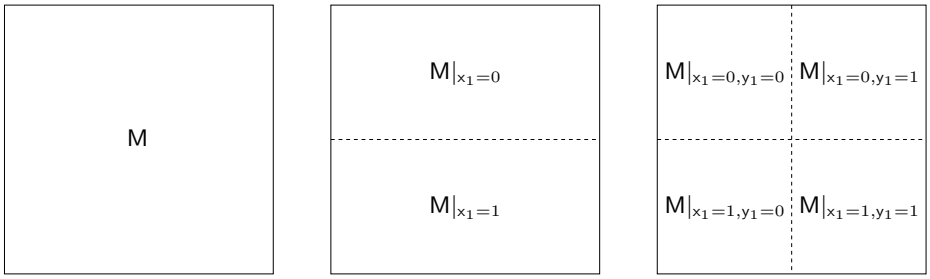


Figure 3.19: Submatrix access via cofactors

for implementing matrix operations which can be expressed recursively.

A good example of this is matrix-matrix multiplication. An algorithm to compute the product of two matrices represented as MTBDDs was first presented in [CMZ+93]. It was shown that by taking all the products of matrix entries first, and summing them afterwards, the multiplication could be performed with an Apply and an Abstract operation. In [CFM+93] an improved, recursive algorithm was proposed, based on the decomposition of a matrix into quadrants, as shown in Figure 3.19, and the fact that matrix-matrix multiplication can be computed in terms of these quadrants:

$$\begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 \\ \mathbf{A}_3 & \mathbf{A}_4 \end{pmatrix} = \begin{pmatrix} \mathbf{B}_1 & \mathbf{B}_2 \\ \mathbf{B}_3 & \mathbf{B}_4 \end{pmatrix} \cdot \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 \\ \mathbf{C}_3 & \mathbf{C}_4 \end{pmatrix} \quad \Longleftrightarrow \quad \begin{aligned} \mathbf{A}_1 &= \mathbf{B}_1 \cdot \mathbf{C}_1 + \mathbf{B}_2 \cdot \mathbf{C}_3 \\ \mathbf{A}_2 &= \mathbf{B}_1 \cdot \mathbf{C}_2 + \mathbf{B}_2 \cdot \mathbf{C}_4 \\ \mathbf{A}_3 &= \mathbf{B}_3 \cdot \mathbf{C}_1 + \mathbf{B}_4 \cdot \mathbf{C}_3 \\ \mathbf{A}_4 &= \mathbf{B}_3 \cdot \mathbf{C}_2 + \mathbf{B}_4 \cdot \mathbf{C}_4 \end{aligned}$$

In [BFG+93], the algorithm was improved further, observing that it was not always necessary to split the matrix at every level of recursion. The latter work also provided a comparison, in terms of empirical results on some benchmark matrices, of all three methods and concluded that, in general, their method performed better. Hence, in our implementation, we have used the algorithm of [BFG+93].

Note that, as we have seen earlier in this chapter, the primary operation which we need for probabilistic model checking is multiplication of a matrix and a vector, not of two matrices. Fortunately, the three algorithms can easily be adapted to this case. For an MTBDD M over variables $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ representing matrix $\mathbf{M}$, and an MTBDD v over $\underline{\mathsf{x}}$ representing column vector $\underline{v}$, we will denote by MVMult(M, v) the function which returns the MTBDD over $\underline{\mathsf{x}}$, representing the column vector $\mathbf{M} \cdot \underline{v}$. It is also trivial to adapt this algorithm to perform multiplication of a vector by a matrix.

Several other useful operations on vectors and matrices can be implemented with basic MTBDD functions. Any pointwise operation, such as addition or scalar multiplication, can be performed with Apply. Vectors and matrices can be transposed using the

ReplaceVars function. Lastly, we define Identity($\underline{x}, \underline{y}$) as the MTBDD (in fact, BDD) representing the identity matrix with row variables $\underline{x}$ and column variables $\underline{y}$.

3.7.3 Implementation Fundamentals

We conclude our coverage of BDDs and MTBDDs by discussing some of the more important aspects of how they are implemented in practice. Since this thesis is concerned with the development of efficient data structures and analysis of their performance, such low-level details are vital. All of our experimental work was implemented using the CUDD (Colorado University Decision Diagram) package of Somenzi [Som97], which, unlike the majority of BDD software, also supports MTBDDs. Most of the issues we discuss here are applicable to all decision diagram packages. Papers which discuss these in more detail can be found in, e.g. [BRB90, YBO+98].

We first examine how the data structures are stored. We have seen above that MTBDDs are kept in a reduced form where no identical nodes are duplicated. In practice, this is taken one step further. All of the MTBDDs in use at any one time are maintained in one large data structure, effectively a very large MTBDD with multiple root nodes. This means that nodes are not even duplicated across different MTBDDs. At any point during an MTBDD operation where a new node needs to be created, it is first verified whether or not such a node already exists and, if so, it is reused.

To ensure that such checks can be performed rapidly, the nodes are stored in a set of hash tables, one for each level (i.e. MTBDD variable). This structure is usually known as the *unique table*. Terminal MTBDD nodes are also stored in this fashion. However, since their values are stored as double precision floating points numbers, in practice, two terminals are considered to be identical if the difference between their values is less than some small threshold ε. In the CUDD package, the default value for ε is 10^{-12}. We found it necessary to lower this to 10^{-15}, almost equal to the precision of a double, to minimise round-off error in our numerical computations.

In normal operation, MTBDDs are constantly being created and destroyed. Hence, it is necessary to keep track of which nodes are currently needed and which are not. This issue is complicated by the fact that the nodes are stored in one shared structure and a given node may be contained in several MTBDDs. The solution is to keep a *reference count* on each node, denoting how many MTBDDs in current use contain it. If the reference count reaches zero, it is known as a *dead node*. Since removing a dead node from the unique table potentially involves restructuring the whole data structure, dead nodes are only removed periodically, typically when their number exceeds some predefined threshold. This process is know as *garbage collection*.

The total amount of memory required to store an MTBDD is directly proportional to its number of nodes. In the CUDD package, each non-terminal node comprises: one integer storing the level (i.e. which variable it is labelled with); two pointers, one for the *then* child and one for the *else* child; one integer storing the reference count; and one pointer used to maintain the node in a linked list as part of a hash table. Assuming that both integers and pointers are stored in 4 bytes, this gives a total of 20 bytes per node. Terminal nodes store slightly different information but this can still be fitted into the same amount of space.

The existence of the unique table is fundamental to the efficient operation of an MTBDD package. Clearly, storing the MTBDDs in this way minimises the amount of storage space required. More importantly, though, it also has ramifications for the speed at which MTBDD operations can be performed. Checking whether two MTBDDs are identical is easy: it reduces to verifying that the root nodes are stored in the same place in the unique table. Because of the canonicity property of MTBDDs, checking whether two MTBDDs represent the same function is equally trivial.

The true value of this comes in the implementation of another fundamental aspect of MTBDD packages: the *computed table*. This is essentially a cache for storing the results of operations on MTBDDs. Before any operation is performed, the cache is checked to see if it has been executed previously. If so, the result can simply be reused. If not, the result is computed, stored in the cache and then returned. Typically, many operations *are* repeated several times, particularly since operations are usually performed recursively and there is often a great deal of node sharing in MTBDDs. Lookups in this cache can only be implemented efficiently if it is quick to compare the equality of two MTBDDs.

In theory, any single operation is only ever performed once. Practically, of course, there is usually an upper limit on the amount of memory allocated to the cache so this is not true. It is still fair to say, though, that the computed table can result in a dramatic improvement in performance.

Chapter 4

Model Representation and Construction with MTBDDs

In this thesis, we consider an MTBDD-based implementation of probabilistic model checking. The first step in this process is to establish an efficient representation for the three types of models which we wish to analyse: DTMCs, MDPs and CTMCs. In this chapter, we consider several ways to minimise the size of the MTBDD representation of these models.

The size of an MTBDD is defined as the number of nodes contained in the data structure. This is particularly important because it affects not only the amount of memory required for storage, but also the amount of time required for manipulation. Typically, the time complexity of operations on MTBDDs is proportional to the number of nodes they contain.

It is well known that the size of an MTBDD representing a given function is extremely sensitive to both the way that the function is encoded into Boolean variables and the ordering that is chosen for these variables. We address both of these topics in detail. In Section 4.1, we will consider the situation for DTMCs and CTMCs, which are very similar. In Section 4.2, we will extend this to MDPs, which raise additional issues. For each type of model, we will present results for a number of case studies and compare the effectiveness of the MTBDD representation with that of equivalent, explicit alternatives, based on sparse matrices.

In the final section of this chapter, we will consider how these MTBDDs are constructed: the process of converting a model's description in the PRISM language into a symbolic representation of the corresponding DTMC, MDP or CTMC. This involves two stages: firstly, translation from the high-level model description into MTBDDs; and secondly, computation of the set of reachable states.

4.1 Representing DTMCs and CTMCs

4.1.1 Schemes for Encoding

We begin by considering the problem of representing DTMCs and CTMCs as MTBDDs. These two types of models are both described by real-valued matrices. From their inception, MTBDDs [CMZ+93, CFM+93, BFG+93] have been used to represent matrices, a process we described in Section 3.7. The basic idea is that a matrix can be thought of as a function mapping pairs of indices to real numbers. Given an encoding of these indices into Boolean variables, we can instead view the matrix as a function mapping Boolean variables to real numbers, which is exactly what an MTBDD represents.

In the simple example of Section 3.7, matrix indices were simply integers and we encoded them using their standard binary representation. In our case, however, the transition matrix of the DTMC or CTMC is indexed by states. Hence, what we actually need is an encoding of the model's state space into MTBDD variables. One approach is to enumerate the set of states in the model, assigning each one a unique integer, and then proceed as before. As we will see, though, by taking a more structured approach to the encoding, we can dramatically improve the efficiency of this representation.

The use of MTBDDs to represent probabilistic models has been proposed on a number of occasions, for example [BFG+93, HMPS94, BCHG+97]. The issue of developing an *efficient* encoding, however, was first considered by Hermanns et al. in [HMKS99]. One of the main contributions of the paper is a set of 'rules of thumb' for deriving compact MTBDD encodings of CTMCs from descriptions in high-level formalisms such as process algebras and queueing networks. This extends previous work [EFT91, DB95] which considers the efficient encoding of non-probabilistic process algebra descriptions into BDDs.

The key observation of Hermanns et al. is that one should try to preserve structure and regularity from the high-level description of the CTMC in its MTBDD encoding. For example, in the process-algebraic setting, a system is typically described as the parallel composition of several sequential components. They show that it is more efficient to first obtain a separate encoding for each of these components, and only then combine them into a global encoding. Regularity in the high-level description which can be reflected in the low-level MTBDD representation results in an increase in the number of shared nodes and, subsequently, a decrease in the size of the data structure.

In [dAKN+00], we described how these ideas can be be applied and extended to encode models described using the PRISM language. In this case, a model's state space is defined by a number of integer-valued PRISM variables and its behaviour by a description given in terms of these variables. Hence, to benefit from structure in this high-level description,

N	States	MTBDD Nodes	
		'Enumerated'	'Structured'
5	240	807	271
7	1,344	3,829	482
9	6,912	15,127	765
11	33,792	54,389	1,096
13	159,744	184,157	1,491
15	737,280	594,309	1,942

Table 4.1: MTBDD sizes for two different encoding schemes

there must be a close correspondence between PRISM variables and MTBDD variables.

To achieve this, we encode each PRISM variable with its own set of MTBDD variables. For the encoding of each one, we use the standard binary representation of integers. Consider a model with three PRISM variables, v_1, v_2 and v_3, each of range $\{0, 1, 2\}$. Our structured encoding would use 6 MTBDD variables, say $\mathsf{x}_1, \ldots, \mathsf{x}_6$, with two for each PRISM variable, i.e. $\mathsf{x}_1, \mathsf{x}_2$ for v_1, $\mathsf{x}_3, \mathsf{x}_4$ for v_2 and $\mathsf{x}_5, \mathsf{x}_6$ for v_3. The state $(2, 1, 1)$, for example, would become $(1, 0, 0, 1, 0, 1)$.

An interesting consequence of this encoding is that we effectively introduce a number of extra states into the model. In our example, 6 MTBDD variables encode $2^6 = 64$ states, but the model actually only has $3^3 = 27$ states, leaving 37 unused. We refer to these extra states as *dummy* states. To ensure that these do not interfere with model checking, when we store the transition matrix for the model, we leave the rows and columns corresponding to dummy states blank (i.e. all zero).

We now present some experimental results to illustrate the effect that the choice of encoding can have on the size of the MTBDD. We use a CTMC model of the cyclic server polling system of [IT90]. By varying N, the number of stations attached to the server, we consider several models of different sizes (for more information, see Appendix E). Table 4.1 shows statistics for each model. We give the number of states and the size of the MTBDD (number of nodes) which represents it for the two different encoding schemes described above: 'enumerated', where we assign each state an integer and encode it using the standard binary encoding; and 'structured', where we work from a high-level description, encoding each PRISM variable with its own set of MTBDD variables. It is clear from the table that the 'structured' encoding results in far more compact storage.

This encoding scheme has two other important advantages. These both result from the close correspondence between PRISM variables and MTBDD variables. Firstly, it facilitates the process of constructing an MTBDD, i.e. the conversion of a description in the PRISM language into an MTBDD representing the corresponding model. Since the

description is given in terms of PRISM variables, this can be done with an almost direct translation. We discuss this process in more detail in Section 4.3 and Appendix C.

Secondly, we find that useful information about the model is implicitly encoded in the MTBDD. As described in Section 3.4, when using PRISM, the atomic propositions used in PCTL or CSL specifications are predicates over PRISM variables. It is therefore simple, when model checking, to construct a BDD which represents the set of states satisfying such a predicate by transforming it into one over MTBDD variables. We will give some examples of this in Section 5.1. With most other encodings, it would be necessary to use a separate data structure to keep track of which states satisfy which atomic propositions.

4.1.2 Heuristics for Variable Ordering

Having chosen a suitable encoding of DTMCs and CTMCs into MTBDDs, we now move on to consider a second issue which can have dramatic effects on efficiency, that of variable ordering. It has long been known that BDDs are extremely sensitive to the ordering of their Boolean variables. For this reason, the problem has received an enormous amount of attention in the literature. Because MTBDDs and BDDs are so similar, many of the principles involved also apply in our case. One key result, from [THY93, BW96], is that the problem of determining the optimal variable ordering for a BDD is NP-hard. Because of this, the traditional approach to finding a good ordering is to rely on heuristics.

We have already seen one example of such an ordering heuristic. When building the MTBDD to represent a matrix in Section 3.7, we interleaved the Boolean variables encoding the row and column indices. The ordering was shown to provide a convenient decomposition of the matrix into its constituent submatrices. More importantly, though, this heuristic can dramatically reduce MTBDD size. The idea was first presented in [EFT91]. We can illustrate its usefulness on a simple example.

In Figure 4.1, we show two MTBDDs representing the 8×8 identity matrix. Both use variables $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3$ to encode row indices and $\mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$ to encode column indices. The first MTBDD (a) uses the interleaved variable ordering $\mathsf{x}_1 < \mathsf{y}_1 < \mathsf{x}_2 < \mathsf{y}_2 < \mathsf{x}_3 < \mathsf{y}_3$ and the second (b) uses $\mathsf{x}_1 < \mathsf{x}_2 < \mathsf{x}_3 < \mathsf{y}_1 < \mathsf{y}_2 < \mathsf{y}_3$. Even on this small example, the difference is clear. In fact, for the general case, the size of the MTBDD for the $2^n \times 2^n$ identity matrix is $\mathcal{O}(n)$ or $\mathcal{O}(2^n)$ for the interleaved and non-interleaved orderings respectively.

The effect of the variable ordering on this example can be explained as follows. Note that, in terms of MTBDD variables, the actual function being represented by the MTBDDs in Figure 4.1 is $(\mathsf{x}_1 = \mathsf{y}_1) \wedge (\mathsf{x}_2 = \mathsf{y}_2) \wedge (\mathsf{x}_3 = \mathsf{y}_3)$. Consider what happens when we traverse the MTBDDs from top to bottom, trying to determine the value of the function for some assignment of the variables. Effectively, each node encodes the values of all the

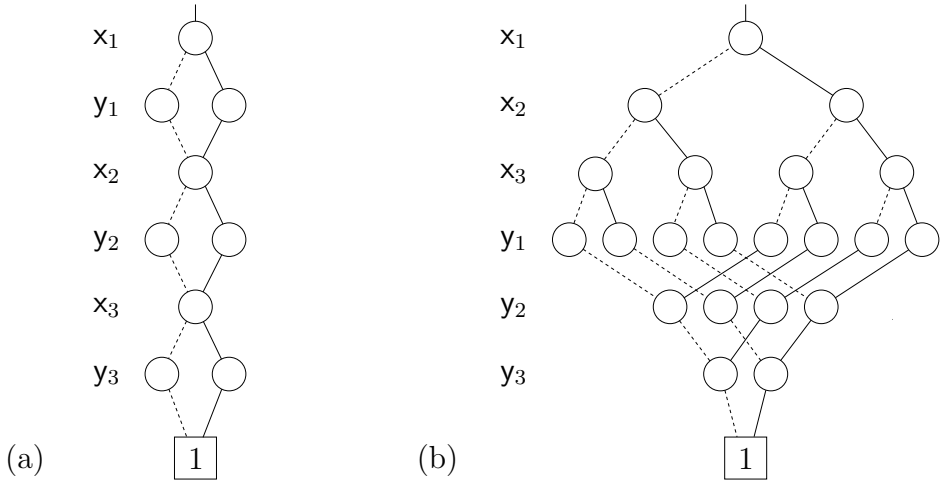


Figure 4.1: Alternative BDD variable orderings for the 8×8 identity matrix: (a) interleaved (b) non-interleaved

variables in levels above it. For example, in the first MTBDD, after two levels, we have established whether or not $\mathsf{x}_1 = \mathsf{y}_1$. If so, we will be positioned at the single node on the x_2 level. If not, we will have already moved to the zero constant node. In either case, from this point on, the values of x_1 and y_1 are effectively irrelevant, since in the function being represented, x_1 and y_1 relate only to each other. In the second MTBDD, however, there is a gap in the ordering between x_1 and y_1. After the first level, we 'know' the value of x_1 but cannot 'use' it until the fourth level. In the meantime, we must consider all possible values of x_2 and x_3, causing a blow-up in the number of nodes required.

This reasoning also explains why the interleaved variable ordering is appropriate for an MTBDD representing a transition matrix. Each traversal through the MTBDD corresponds to a single transition of the model. In a typical transition, only a few PRISM variables will actually change value, the rest remaining constant. Since there is a direct correspondence between PRISM variables and MTBDD variables, this argument also applies at the MTBDD level. This means that we have a similar situation to the identity matrix: generally, each y_i variable is most closely related to x_i. Hence, the interleaved variable ordering is beneficial. Table 4.2 demonstrates the effect of this on some typical transition matrices. We use the polling system case study of [IT90], as in the previous section, but the results are the same for all our examples. We present MTBDD sizes for both the interleaved and non-interleaved ordering. The difference is clear: it is simply not feasible to consider non-interleaved orderings.

Even once we have decided to opt for an interleaved variable ordering, there is still benefit to be derived from considering the positioning of the individual x_i and y_i variables

N	States	MTBDD Nodes	
		Interleaved	Non-interleaved
5	240	271	1,363
7	1,344	482	6,766
9	6,912	765	39,298
11	33,792	1,096	178,399
13	159,744	1,491	794,185

Table 4.2: MTBDD sizes for interleaved and non-interleaved variable orderings

within the overall ordering. Actually, we need only decide on an ordering for the x_i variables: the y_i variables will have to be in the same order. There are many other variable ordering heuristics to be found in the literature. Unfortunately, they are generally application dependent and hence not directly relevant to our situation. The general principles on which they are based, however, can still be used.

For example, we argued above that since an MTBDD node effectively encodes the values of all the variables in levels above it, it is wise to place closely related MTBDD variables as near to each other as possible in the ordering. By similar reasoning, it can be inferred that, if a single MTBDD variable is closely related to several others, it should benefit from being placed above all of them in the ordering.

The MTBDD variables encoding a single PRISM variable can be seen as closely related to each other. Furthermore, those relating to PRISM variables in the same module are more likely to be related to each other. Hence, our default ordering is to group MTBDD variables by module and, within that, by PRISM variable.

In fact, because of the close correspondence between MTBDD variables and PRISM variables, these heuristics can also be applied at the level of the PRISM language. Hence, it can be beneficial to choose an ordering for the PRISM variables within a module or for the ordering of the modules themselves according to similar rules.

We demonstrate this through an example, again using the polling system case study of [IT90]. The model consists of $N+1$ modules: a server and N stations. In ordering 1, we place the MTBDD variables for the server first, and those for the stations afterwards. In ordering 2, we do the opposite, with the variables for all stations first and then those for the server last. We would expect ordering 1 to perform better since, in the model, all stations communicate with the server but not with each other. Table 4.3 shows the results for several values of N, in each case giving the size of the MTBDD under both orderings. The conclusion we can draw from this is that, in some cases, it can be *extremely* beneficial to apply such heuristics.

N	States	MTBDD Nodes	
		Ordering 1	Ordering 2
5	240	271	417
7	1,344	482	1,236
9	6,912	765	5,415
11	33,792	1,096	17,804
13	159,744	1,491	67,098
15	737,280	1,942	264,122
17	3,342,336	2,469	1,312,887

Table 4.3: MTBDD sizes for two different variable orderings

4.1.3 Results

In the previous sections, we presented a number of techniques to achieve compact representations of DTMCs and CTMCs using MTBDDs. We now present some statistics to demonstrate the effectiveness of these techniques on a number of different examples. We do this by comparing the amount of memory required by MTBDDs to that of an equivalent explicit alternative, namely sparse matrices. Note that, in this section, we only consider the space requirements of the representations, and not timing factors such as the speed with which the data structures can be accessed or manipulated. This will be covered in the next chapter when we discuss model checking. Typically though, the smaller the MTBDD, the faster the operations on it will be.

Table 4.4 gives statistics for several different case studies: the polling system of [IT90], as used in the preceding sections; the Kanban manufacturing system of [CT96]; and the bounded retransmission protocol (BRP) of [HSV94]. The first two give rise to CTMCs, the third to a DTMC. For each one, we can construct models of varying size by changing a parameter N. In the polling system, N refers to the number of stations; in the Kanban system, N is the number of jobs in the system; and in the bounded retransmission protocol, N is the number of packets to be transmitted. For more information on the case studies and their respective parameters, see Appendix E.

The table gives the size of each model (both the number of states and the number of transitions) and the amount of memory required to store it (to the nearest kilobyte) both as an MTBDD and as a sparse matrix. For the MTBDD representation, the amount of memory needed is proportional to the number of nodes. We assume 20 bytes per node, as described in Section 3.7.3. For the sparse matrix representation, the memory usage is as described in Section 3.6. Note that, in all cases, we had to actually construct the MTBDD in order to compute the number of nodes. The largest sparse matrices, on the

Model	N	States	Transitions	Memory (KB)	
				MTBDD	Sparse
Polling system	5	240	800	5	10
	7	1,344	5,824	9	74
	9	6,912	36,864	15	459
	11	33,792	214,016	21	2,640
	13	159,744	1,171,456	29	14,352
	15	737,280	6,144,000	38	74,880
	17	3,342,336	31,195,136	48	378,624
Kanban system	3	58,400	446,400	48	5,459
	4	454,475	3,979,850	96	48,414
	5	2,546,432	24,460,016	123	296,588
	6	11,261,376	115,708,992	154	1,399,955
	7	41,644,800	450,455,040	186	5,441,445
	8	133,865,325	1,507,898,700	287	18,193,599
BRP	1,000	55,005	69,998	49	1,035
	2,500	137,505	174,998	53	2,588
	5,000	275,005	349,998	54	5,176
	10,000	550,005	699,998	56	10,352
	20,000	1,100,005	1,399,998	58	20,703

Table 4.4: Symbolic versus explicit storage for DTMCs and CTMCs

other hand, are too large to be stored, but the required space can be computed anyway. From Table 4.4, it is clear that, by exploiting structure, we can store significantly larger models using MTBDDs than we can with sparse matrices. Of course, it can be argued that the original, high-level description of a model constitutes an even more compact, structured representation. The MTBDD, however, provides direct access to the model's transition matrix.

4.2 Representing MDPs

4.2.1 Schemes for Encoding

Representing MDPs with MTBDDs is more complex than DTMCs or CTMCs since the nondeterminism must also be encoded. An MDP is not described by a transition matrix over states, but by a function *Steps* mapping each state to a set of nondeterministic choices, each of which is a probability distribution over states.

Assuming, however, that the maximum number of nondeterministic choices in any state is m, and letting S denote the set of states of the MDP, we can reinterpret *Steps*

as a function of the form $S \times \{1, \ldots, m\} \times S \to [0, 1]$. We have already discussed, in the previous section, ways of encoding a model's state space S into Boolean variables. If we encode the set $\{1, \ldots, m\}$ in a similar fashion, we can consider *Steps* as a function mapping Boolean variables to real numbers, and hence represent it as an MTBDD. We will use variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ and $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$ to range over source and destination states and variables $\underline{\mathsf{z}} = (\mathsf{z}_1, \ldots, \mathsf{z}_k)$ to encode $\{1, \ldots, m\}$. We will refer to $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ as row and column variables, as before, and $\underline{\mathsf{z}}$ as *nondeterministic variables*. The idea of encoding nondeterministic choice in an MDP as a third index to represent it as an MTBDD was proposed in [Bai98] but the application or practicality of it was not considered.

As discussed in Section 3.3.2, when implementing model checking of MDPs, it is often useful to consider *Steps* as non-square matrix, where each row corresponds to a single nondeterministic choice. This allows certain parts of the model checking algorithm to be conveniently expressed as matrix-vector multiplication. Similarly, we can think about the MTBDD representation described above as storing a matrix with $m \cdot |S|$ rows, encoded by variables $\underline{\mathsf{x}}$ and $\underline{\mathsf{z}}$, and $|S|$ columns, encoded by variables $\underline{\mathsf{y}}$. We will return to this idea when we consider an MTBDD-based implementation of model checking in the next chapter.

First, though, there are still several issues to address with our encoding. For example, it is likely that in some states, there will be less than m nondeterministic choices. Since we represent the MDP by a function of the form $S \times \{1, \ldots, m\} \times S \to [0, 1]$, some parts of it remain undefined. This problem will also arise when we encode $\{1, \ldots, m\}$ into Boolean variables. Unless m happens to be a power of two, we will be introducing extra indices, the meaning of which is undefined. This situation is analogous to the case with DTMCs and CTMCs, where we added dummy states with no transitions to our model. We will take the same approach here, effectively adding extra, empty probability distributions to some (or all) states.

We formalise this idea as follows. Let Steps be an MTBDD over the variable sets $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$, $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$ and $\underline{\mathsf{z}} = (\mathsf{z}_1, \ldots, \mathsf{z}_k)$. We say that Steps represents an MDP with state space S and transition function *Steps* if, for some state space encoding $enc : S \to \mathbb{B}^n$, for any $s \in S$:

- if $\mu \in \mathit{Steps}(s)$, then there exists $b \in \mathbb{B}^k$ such that:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$
- for any $b \in \mathbb{B}^k$, one of the following two conditions holds:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = 0$ for all $t \in S$

– there exists $\mu \in \mathit{Steps}(s)$ such that

$f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$

The first part ensures that every nondeterministic choice in each state of the MDP is encoded in the MTBDD and the second ensures that every choice encoded in the MTBDD does actually belong to the MDP. Note that it is acceptable for a choice to be included more than once in the MTBDD. In the definition of an MDP, $\mathit{Steps}(s)$ is a set and hence contains no duplicates. When constructing the MTBDD representing it, though, checking for duplicate choices would be a costly process so we ignore them. Fortunately, it is safe to do this because when we perform model checking on the MDP, we only ever compute the minimum or maximum probability of some event occurring; we never need to distinguish between individual nondeterministic choices.

Another issue to consider is how we actually encode the nondeterministic choices as MTBDD variables. Since the number of nondeterministic choices is typically very small compared to the number of states (i.e. $m \ll |S|$), one scheme would be to just enumerate them and encode them as integers, as we would do for an individual PRISM variable. The lesson we learnt from the previous section, however, was that it can be extremely beneficial to capture any high-level structure in our MTBDD encoding.

On example of such regularity is that in nearly all of the MDP case studies we have considered, most of the nondeterminism arises from the parallel composition (scheduling) of modules. Let us assume that this is in fact the only source of nondeterminism. In an MDP composed of m modules, there will be exactly m nondeterministic choices in every state, each corresponding to one of the modules being scheduled. This suggests a more structured encoding, using m nondeterministic variables, one for each module.

We illustrate the difference between these two schemes, which we will refer to as 'enumerated' and 'structured', through an example. Consider a model comprising four modules. In each state there will be four nondeterministic choices. Using the 'enumerated' scheme, we can encode these with 2 Boolean variables since $2^2 = 4$. Using the 'structured' scheme, we would require 4 variables. For simplicity, assume that the new variables appear at the start of the overall ordering. Figure 4.2 shows what the two MTBDDs would look like. The grey triangles denote the lower subgraphs of the MTBDDs, the exact structure of which is irrelevant here.

From the diagram, it seems clear that the 'enumerated' encoding will always be superior, in general requiring $\mathcal{O}(m)$, as opposed to $\mathcal{O}(m^2)$, nodes to encode the nondeterminism for m modules. In the next section, however, we will demonstrate that, for variable orderings other than the one used in Figure 4.2, this is not necessarily the case.

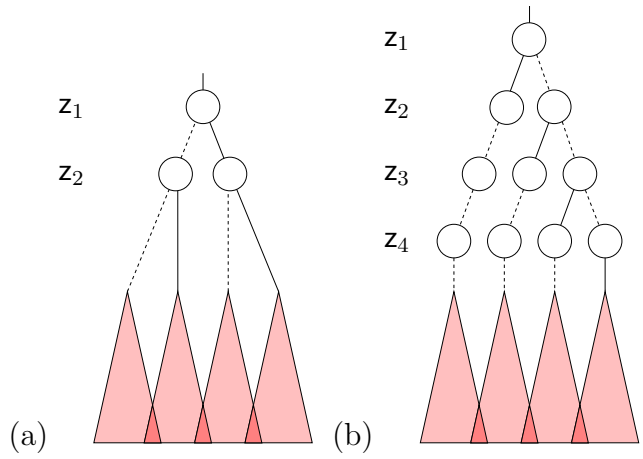


Figure 4.2: The two MTBDD encodings for nondeterminism in MDPs: (a) 'enumerated' (b) 'structured'

4.2.2 Heuristics for Variable Ordering

As for DTMCs and CTMCs, it is important to consider the ordering of the MTBDD variables. Since the previous reasoning still applies, it makes sense to retain our ordering heuristics for the row and column variables. All we need to consider, in fact, is where the nondeterministic variables are inserted into this ordering. The choice will depend on which of the two encodings from the previous section we opt for.

The obvious two policies would be to place all the nondeterministic variables either at the very top of the ordering or the very bottom. We will refer to these two options as 'top' and 'bottom'. Of these, 'top' would appear most promising option since the nondeterministic variables can be seen as related to all other variables: they affect every transition of the model.

The effect of this option has already been illustrated for each of the two encodings in Figure 4.2. Note that, since the only difference between the two MTBDDs is the encoding of the nondeterminism, the lower regions of the MTBDDs, i.e. all nodes labelled with row and column variables rather than nondeterministic variables, are exactly the same for both encodings. We have already made the observation that the maximum number of nondeterministic choices is typically much less than the number of states in the model. If this is the case, then the lower parts of the MTBDD represent the majority of the graph. Hence, there is actually very little difference between the two encodings.

If we focus on the 'structured' encoding, however, we have a third possibility for variable ordering. There are m nondeterministic variables, each corresponding to one module. Recall from the discussion of state space encoding in Section 4.1.1 that the row

Model	N	States	MTBDD Nodes				
			'Enumerated'		'Structured'		
			'Top'	'Bottom'	'Top'	'Bottom'	'Middle'
Mutual exclusion	3	10,744	28,500	28,557	28,503	28,590	23,159
	4	201,828	65,621	79,960	65,628	80,061	43,468
	5	6,769,448	136,836	233,641	136,845	233,777	78,292
	6	1.3×10^8	206,204	506,653	206,219	507,108	103,385
	8	4.5×10^{10}	381,163	-	381,192	-	153,571
	10	4.7×10^{13}	962,787	-	962,831	-	328,030
Coin protocol ($K = 8$)	4	84,096	2,290	1,973	2,297	1,995	1,398
	6	4,612,864	6,948	7,540	6,963	7,605	3,142
	8	2.2×10^8	15,684	25,936	15,713	26,220	5,587
	10	1.0×10^{10}	29,655	91,990	29,699	92,801	8,718
	12	4.3×10^{11}	50,079	342,087	50,145	346,005	12,536
	14	1.8×10^{13}	78,262	1,322,933	78,353	1,339,214	17,047

Table 4.5: MTBDD sizes for several different MDP variable orderings

and column variables for each module are grouped together. Since the nondeterministic variable for a module can be seen as related to its row and column variables, it would seem wise to group them all together. We will refer to this third ordering option as 'middle'. For a system of m modules, where the state space of the ith module is encoded by n_i row variables $(\mathsf{x}_{i,1}, \ldots, \mathsf{x}_{i,n_i})$ and n_i column variables $(\mathsf{y}_{i,1}, \ldots, \mathsf{y}_{i,n_i})$, the three orderings can be summarised as shown below. The parentheses are purely to aid readability.

- 'top' $(\mathsf{z}_1, \ldots, \mathsf{z}_m), (\mathsf{x}_{1,1}, \mathsf{y}_{1,1}, \ldots, \mathsf{x}_{1,n_1}, \mathsf{y}_{1,n_1}), \ldots, (\mathsf{x}_{m,1}, \mathsf{y}_{m,1}, \ldots, \mathsf{x}_{m,n_m}, \mathsf{y}_{m,n_m})$
- 'bottom' $(\mathsf{x}_{1,1}, \mathsf{y}_{1,1}, \ldots, \mathsf{x}_{1,n_1}, \mathsf{y}_{1,n_1}), \ldots, (\mathsf{x}_{m,1}, \mathsf{y}_{m,1}, \ldots, \mathsf{x}_{m,n_m}, \mathsf{y}_{m,n_m}), (\mathsf{z}_1, \ldots, \mathsf{z}_m)$
- 'middle' $(\mathsf{z}_1, \mathsf{x}_{1,1}, \mathsf{y}_{1,1}, \ldots, \mathsf{x}_{1,n_1}, \mathsf{y}_{1,n_1}), \ldots, (\mathsf{z}_m, \mathsf{x}_{m,1}, \mathsf{y}_{m,1}, \ldots, \mathsf{x}_{m,n_m}, \mathsf{y}_{m,n_m})$

We have experimented with combinations of encodings and variable orderings on several case studies which give rise to MDP models. Our findings are typified by the results, given in Table 4.5, for two examples: Rabin's randomised mutual exclusion algorithm [Rab82] and the coin protocol from Aspnes and Herlihy's randomised consensus protocol [AH90]. Both are parameterised by N, the number of processes modelled. The latter case study has an additional parameter, K, which we fix here at 8. Further details about these case studies and others we have used can be found in Appendix E. For each MDP, we give its size (number of states) and the number of nodes in the five MTBDDs which can represent it, one for each of the encoding/ordering pairs discussed. A dash indicates that the MTBDD could not be built due to memory constraints.

The results confirm our various predictions. When choosing the 'top' or 'bottom'

ordering, there is very little difference between the two encodings, but 'top' consistently performs much better than 'bottom'. More importantly, we find that using the 'structured' encoding and the 'middle' ordering results in the most compact MTBDD. This confirms our most important heuristic: that it is beneficial to exploit structure in the model.

The encoding schemes we have described here only handle the nondeterminism which results from inter-module scheduling. In the general case, there could also be nondeterministic choices made locally, within a module. There is usually no particular structure to such nondeterminism, but it is associated with an individual module. Hence, these choices could be encoded in a simple, unstructured fashion, and the nondeterministic variables used placed with the others for that module.

We are aware of two other examples in the literature which consider the representation of MDPs using MTBDDs. Firstly, we mention the work of Hoey et al. [HSAHB99, SAHB00], who implement techniques for decision-theoretic planning. Although the application domain is very different, the iterative algorithm implemented is similar to ours. In terms of representation, however, due to the nature of their algorithm, it is more convenient to store the MDP as a set of several MTBDDs. In our scenario, this would constitute storing separate MTBDDs, one for each of the possible nondeterministic choices.

A second instance can be found in [DJJL01] which presents a technique for model checking a restricted subset of PCTL over MDPs. Although the application domain in this case is very similar to ours, the emphasis of the paper is on the algorithm itself, not the implementation. They state only that the states of the MTBDD and the nondeterministic can be encoded into a set of Boolean variables. The compactness of the MTBDD representation is not considered.

4.2.3 Results

We now present some results to illustrate the effectiveness of the symbolic storage schemes for MDPs described in the previous two sections. As for DTMCs and CTMCs, we will consider the amount of storage space required by the MTBDD and compare this to that of the equivalent explicit storage scheme, described in Section 3.6.1. Any time-related issues will be dealt with in the next chapter where we consider model checking. Typically though, operations on smaller MTBDDs will be performed more quickly.

Table 4.6 gives statistics for the two MDP case studies from the previous section: Rabin's randomised mutual exclusion algorithm [Rab82] and the coin protocol of [AH90]. For each MDP, the table gives the size of the model (number of states, number of transitions and number of choices) and the amount of memory required (to the nearest kilobyte) by the MTBDD or sparse matrix to represent it. As before, we assume that each MTBDD

Model	N	States	Transitions	Choices	Memory (KB)	
					MTBDD	Sparse
Mutual exclusion	3	10,744	128,934	36,768	452	1,697
	4	201,828	3,379,072	912,320	849	43,951
	5	6,769,448	1.7×10^8	3.8×10^7	1,529	2,094,274
	6	1.3×10^8	3.9×10^9	8.7×10^8	2,019	5.0×10^7
	8	4.5×10^{10}	1.8×10^{12}	4.0×10^{11}	2,999	2.3×10^{10}
	10	4.7×10^{13}	2.7×10^{15}	5.1×10^{14}	6,406	3.4×10^{13}
Coin protocol ($K = 8$)	4	84,096	392,544	336,384	27	6,243
	6	4,612,864	3.2×10^7	2.8×10^7	61	504,239
	8	2.2×10^8	2.1×10^9	1.8×10^9	109	3.2×10^7
	10	1.0×10^{10}	1.2×10^{11}	1.0×10^{11}	170	1.8×10^9
	12	4.3×10^{11}	6.1×10^{12}	5.2×10^{12}	245	9.3×10^{10}
	14	1.8×10^{13}	3.0×10^{14}	2.5×10^{14}	333	4.6×10^{12}

Table 4.6: Symbolic versus explicit storage for MDPs

node takes 20 bytes. For the sparse storage scheme, the amount of memory required is computed as described in Section 3.6.1.

We find that MTBDDs can store structured MDP models extremely compactly. In fact, the representation is generally even more compact than for DTMCs and CTMCs. The main reason for this seems to be that there are fewer terminals in the MTBDDs, implying that there are less distinct values (i.e. probabilities) in the MDPs. This is not a property specifically of MDPs, but of the type of case studies we have considered: distributed randomised algorithms. Generally the probabilistic behaviour in these systems results from simple random choices, often from coin tosses with values such as $\frac{1}{2}$ or $\frac{1}{4}$.

4.3 Construction and Reachability

We have presented ways in which three types of probabilistic model can be encoded as MTBDDs. We have shown how to optimise these encodings and demonstrated that they can compare very favourably with explicit storage schemes. In this final section, we consider the issue of how these MTBDDs are actually constructed, i.e. how a system description in the PRISM language can be translated into an MTBDD which represents the corresponding probabilistic model.

This translation proceeds in three phases. The first task is to establish an encoding of the model's state space into MTBDD variables. This process was described in the preceding sections of this chapter. Secondly, using the correspondence between PRISM

and MTBDD variables provided by this encoding, an MTBDD representing the model is constructed from its description. Thirdly, we compute from the constructed model the set of reachable states. All unreachable states, which are of no interest, are then removed. The next two sections describe these second and third phases (construction and reachability) in more detail. Note that all the models for which statistics have been presented in this chapter have been built in this fashion.

4.3.1 Construction

The construction process converts a model described in the PRISM language into an MTBDD representing the corresponding DTMC, MDP or CTMC, as defined by the semantics in Appendix B. One of the motivations behind the design of the language was to allow for an efficient translation into MTBDDs. In this section, we describe the translation process informally through a simple example. Appendix C contains a formal description of the entire process and a proof that the construction is correct. A preliminary version of this translation was presented in [dAKN+00].

Figure 4.3 shows a simple DTMC described in the PRISM language. It comprises two modules, M_1 and M_2, each with a single variable, v_1 and v_2, respectively, of range [0..1]. Hence, the global state space of the model is $\{0,1\} \times \{0,1\} = \{(0,0),(0,1),(1,0),(1,1)\}$. Our first task is to encode this state space into Boolean variables. Since the PRISM variables v_1 and v_2 are themselves Boolean, we will simply use row variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \mathsf{x}_2)$ and column variables $\underline{\mathsf{y}} = (\mathsf{y}_1, \mathsf{y}_2)$, where x_1 and y_1 correspond to variable v_1 and x_2 and y_2 to variable v_2.

Since the encoding gives such a close correspondence between PRISM variables and MTBDD variables, the model description can be translated almost directly. First, an MTBDD is constructed for each command in the description. Each one defines the *local* behaviour of a particular module for some subset of the *global* state space. Hence, it is represented by an MTBDD over the row variables for the whole system and the column variables for that module. Figure 4.4 demonstrates the translation for the first and third commands of module M_1. Note how non-primed and primed PRISM variables are translated into row and column variables, respectively.

The MTBDD describing the behaviour of an entire module is computed by summing those for all of its commands. The MTBDD representing the whole DTMC is then obtained by combining the MTBDDs for all its modules. The precise details of this are left to Appendix C. Here, we simply aim to demonstrate that the translation process is direct and relatively simple. Consequently, the construction is also usually quite fast. Section 4.3.3 presents some results to illustrate this.

dtmc

module M_1
$v_1 :$ [0..1] **init** 0;
[] $(v_1 = 0) \wedge (v_2 = 0) \rightarrow (v_1' = 1)$;
[] $(v_1 = 0) \wedge (v_2 = 1) \rightarrow (v_1' = 0)$;
[] $(v_1 = 1) \rightarrow 0.4 : (v_1' = 0) + 0.6 : (v_1' = 1)$;
endmodule

module M_2
$v_2 :$ [0..1] **init** 0;
[] $(v_2 = 0) \wedge (v_1 = 0) \rightarrow (v_2' = 1)$;
[] $(v_2 = 0) \wedge (v_1 = 1) \rightarrow (v_2' = 0)$;
[] $(v_2 = 1) \rightarrow 0.4 : (v_2' = 0) + 0.6 : (v_2' = 1)$;
endmodule

Figure 4.3: Description of a small DTMC in the PRISM language

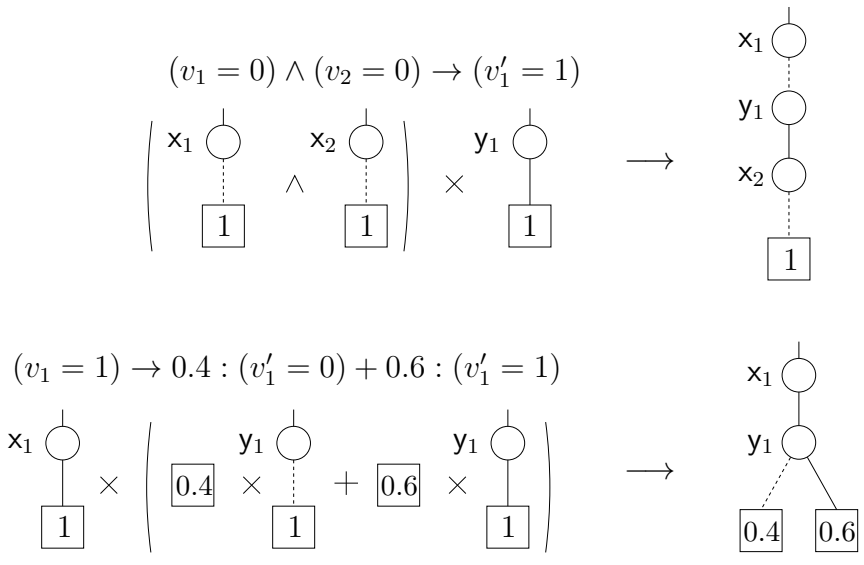


Figure 4.4: MTBDD construction for two commands of module M_1

4.3.2 Reachability

Computing the set of reachable states of the model can be done via a breadth-first search of the state space starting with the initial state. This is well suited to implementation as a BDD fixpoint computation. We first compute BDDs representing the initial state and the transition relation for the underlying graph of the model. Reachability is then performed iteratively using these two BDDs. The implementation is very similar to that of the precomputation algorithms for PCTL and CSL model checking, discussed in the next chapter. We give full details of the BDD algorithm for reachability in Appendix C.

It should be noted that, in non-probabilistic model checking, determining the reachable states of a model may actually be sufficient for model checking. In our case, though, we usually need to perform probability calculations. Since these must be performed on the entire, reachable model, we consider reachability to be part of the construction phase.

Another interesting observation which we make here is that the removal of unreachable states from the model often causes a slight increase in the size of its MTBDD. This is despite the fact that both the number of states and the number of transitions in the model decreases. The explanation for this phenomenon is that the regularity of the model is also reduced and, as we have seen repeatedly in this chapter, structure and regularity are the chief sources of MTBDD efficiency. It is, however, impractical to retain the unreachable states since this would result in extra work being performed at the model checking stage.

4.3.3 Results

Finally, we present some statistics to demonstrate the performance of the construction and reachability phases. Table 4.7 gives statistics for a selection of different case studies (see Appendix E for further details about each example). For each one, we give the number of states in the model, the time taken for construction (conversion from the PRISM language to an MTBDD), and both the number of fixpoint iterations and the total time required to compute the set of reachable states. As will always be the case in this thesis, timings given refer to actual (wall) time as run on a 440 MHz 512 MB Sun Ultra10. It can be seen that we can construct very large models with quite reasonable speed: in all cases the whole process takes less than 80 seconds. This means that we can now concentrate on the implementation and efficiency of the actual model checking process, which is our main area of interest.

Model	Parameters	States	Construction (sec.)	Reachability	
				Iterations	Seconds
Polling system	$N = 18$	7,077,888	0.47	37	0.52
Kanban system	$N = 6$	11,261,376	0.41	85	3.99
FMS	$N = 8$	4,459,455	7.11	65	36.1
BRP	$N = 2,500$	137,505	8.02	15,011	71.3
Dining philosophers	$N = 4, K = 6$	3,269,200	6.38	36	65.1
Coin protocol	$N = 10, K = 6$	7,598,460,928	1.22	729	17.4
FireWire	$N = 3,000$	2,238,333	13.1	3,050	27.2

Table 4.7: Performance statistics for construction and reachability

Chapter 5

Model Checking with MTBDDs

Having presented techniques to construct and store probabilistic models with MTBDDs in an efficient manner, we now consider the problem of carrying out probabilistic model checking using the same data structure. The first three sections of this chapter describe how this can be implemented. We cover model checking of PCTL over DTMCs and MDPs and of CSL over CTMCs. In doing so, we reuse existing algorithms and results from [BFG+93, HMPS94, BCHG+97, KNPS99, HMKS99, BKH99, dAKN+00, KKNP01]. We will see that there is a great deal of similarity between the various algorithms.

The main contribution of this chapter is a thorough investigation into the practical applicability of these techniques. In Section 5.4, we present experimental results for MTBDD-based probabilistic model checking on a wide range of case studies. We also compare the performance of our symbolic implementation with that of an explicit version based on sparse matrices. In Section 5.5, we analyse the performance of the symbolic implementation in more detail and consider some potential improvements.

5.1 The Main Algorithm

As we saw in Chapter 3, the model checking algorithms for both PCTL and CSL take a formula in the logic, along with a model of the appropriate type, and return the set of states which satisfy the formula. The first requirement of our implementation of these model checking algorithms is that the inputs and outputs can be represented as MTBDDs. We have seen in the previous chapter how, by encoding its state space with Boolean variables, a model can be represented as an MTBDD. Using the same state space encoding, we can represent a state-indexed vector, as described in Section 3.7. Furthermore, by representing a set of states S' by its characteristic function (i.e. the function $f_{S'}$ for which $f_{S'}(s)$ equals 1 if $s \in S'$ and 0 otherwise), we can also represent sets of states using BDDs.

ϕ	$\textsc{Sat}(\phi)$
true	$\textsc{Const}(1)$
a	$\textsc{Encode}(a)$
$\phi_1 \wedge \phi_2$	$\textsc{Sat}(\phi_1) \wedge \textsc{Sat}(\phi_2)$
$\neg\phi$	$\neg\textsc{Sat}(\phi)$

Figure 5.1: Model checking for non-probabilistic PCTL and CSL operators

This gives us all that is required.

In the description of our implementation, we will assume the following. For DTMCs, MDPs and CTMCs, the transition probability matrix $\mathbf{P}$, transition function *Steps* and transition rate matrix $\mathbf{R}$ are represented by MTBDDs P, Steps and R respectively. We use $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ as row variables, $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$ as column variables, and $\underline{\mathsf{z}} = (\mathsf{z}_1, \ldots, \mathsf{z}_k)$ as nondeterministic variables. Hence, P and R are over $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$, and Steps is over $\underline{\mathsf{x}}$, $\underline{\mathsf{y}}$ and $\underline{\mathsf{z}}$. For state-indexed vectors and sets of states we use an (MT)BDD over $\underline{\mathsf{x}}$.

The model checking algorithms for PCTL and CSL proceed by traversing the parse tree for the logical formula and recursively evaluating the set of states which satisfy each subformula. Hence, it suffices to provide one algorithm for each logical operator. For the non-probabilistic operators (*true*, a, $\wedge$, $\neg$), common to PCTL and CSL, model checking is trivial and performed identically for both logics. The operations required are given in Figure 5.1. We denote by $\textsc{Sat}(\phi)$ the function which computes the BDD for the set of states satisfying the formula ϕ. Note that since the model checking algorithm is recursive, we can assume that any subformulas have already been model checked. For example, when computing $\textsc{Sat}(\neg\phi)$, we can assume that the result of $\textsc{Sat}(\phi)$ is known.

Model checking for *true*, $\wedge$ and $\neg$ uses only straightforward BDD operations. For an atomic proposition a, we denote by $\textsc{Encode}(a)$ the function which returns the BDD representing the set of states which satisfy a. Fortunately, as observed in Section 4.1.1, this is also easy to generate. This is because, in practice, atomic propositions are predicates over PRISM variables and since we adopt a structured approach to encoding the model's state space, there is a direct correspondence between PRISM variables and MTBDD variables. Consider, for example, a single PRISM variable v with range $\{0, 1, 2, 3\}$, encoded as described in Section 4.1.1 by two MTBDD variables x_1 and x_2. Figure 5.2 shows BDDs corresponding to some simple predicates. More complex predicates can be created using logical connectives such as $\vee$ and $\wedge$. Their BDDs can be computed using the corresponding BDD $\vee$ and $\wedge$ operations.

One further point we should note about the main model checking algorithm is that we must be careful to only include reachable states. We saw in the previous chapter that

v	x_1	x_2
0	0	0
1	0	1
2	1	0
3	1	1

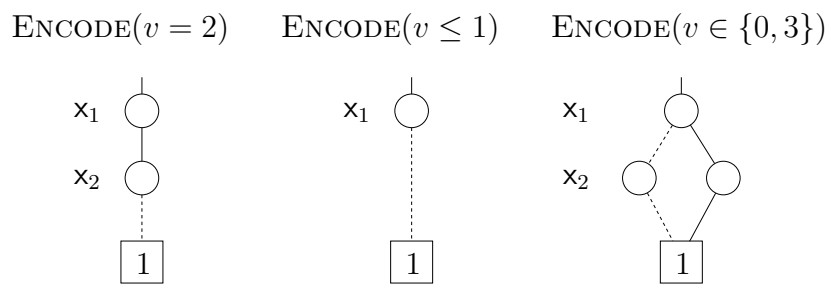


Figure 5.2: BDD encodings for atomic propositions

the MTBDD encoding of a model's state space will typically include some unreachable or non-existent states. Since these are uninteresting, we remove them by performing a conjunction with a BDD reach representing the set of reachable states. So, for example, Sat($\neg\phi$) would actually be computed as reach $\wedge \neg$ Sat(ϕ).

The remaining PCTL and CSL operators which we need to consider are the probabilistic operators, $\mathcal{P}_{\bowtie p}[\cdot]$ and $\mathcal{S}_{\bowtie p}[\cdot]$. These are non-trivial and constitute the bulk of the work required for model checking. The overall procedure is the same for both operators. In each case, we are required to compute a vector of probabilities (either $p_s(\cdot)$, $p_s^{max}(\cdot)$ or $p_s^{min}(\cdot)$ for $\mathcal{P}_{\bowtie p}[\cdot]$, or $\pi(s)$ for $\mathcal{S}_{\bowtie p}[\cdot]$), one for each state, and compare the probabilities with the bound ($\bowtie p$) to establish which states satisfy the formula. Assuming that we have computed an MTBDD probs representing the vector of probabilities for all states, the BDD representing the result is Threshold(probs, $\bowtie, p$).

The calculation of probabilities is covered in the next two sections. We begin with the implementation of the precomputation algorithms, which usually constitute the first phase of the process. We then consider the situations where numerical computation is required. In both cases, rather than describe all the different algorithms in their entirety, we give the implementation of a few representative examples. All the algorithms can be found in Appendix D.

5.2 Precomputation Algorithms

The first step in the model checking algorithm for many of the probabilistic operators of PCTL and CSL is to execute one or more precomputation algorithms. These determine, via a graph analysis of the model, the states for which the relevant probability is exactly 0 or 1. Probabilities need then only be computed for the remaining states. The precomputation algorithms are important because, in some cases, they produce enough information for the second and more expensive numerical computation phase to be skipped. Even

```
Prob0(phi_1, phi_2)
1.  sol := phi_2
2.  done := false
3.  while (done = false)
4.      sol' := sol ∨ (phi_1 ∧ ThereExists(y, T ∧ ReplaceVars(sol, x, y))
5.      if (sol' = sol) then done := true
6.      sol := sol'
7.  endwhile
8.  return ¬sol
```

Figure 5.3: The MTBDD version of the Prob0 algorithm

when the second phase is carried out, it only computes an approximation to the exact probabilities and is subject to round-off errors. The precomputation algorithms, however, present no such drawbacks.

The algorithms we need, as described in Section 3.3, are Prob0, Prob1, Prob0A, Prob1E, and Prob0E. We will concentrate on Prob0, which is sufficient to explain all the necessary implementation details. The MTBDD versions of all five algorithms can be found in Appendix D.

Prob0 is used when model checking a PCTL until formula $\mathcal{P}_{\bowtie p}[\phi_1 \,\mathcal{U}\, \phi_2]$ over a DTMC. It computes the set of states for which $p_s(\phi_1 \,\mathcal{U}\, \phi_2) = 0$. The algorithm was given in Figure 3.1. We assume that the sets $Sat(\phi_1)$ and $Sat(\phi_2)$ have already been computed, and are represented by phi_1 and phi_2, BDDs over the variables $\underline{\mathsf{x}}$. The algorithm only considers the existence of a transition between two states, not the actual probability of making that transition. Hence, it is more efficient to use a BDD T, representing the transition relation of the underlying graph, than to use the MTBDD P representing the transition probability matrix. This can be computed as $\mathsf{T} := \textsc{Threshold}(\mathsf{P}, >, 0)$. The Prob0 algorithm will return a BDD over variables $\underline{\mathsf{x}}$ representing the required set of states. Figure 5.3 gives the MTBDD algorithm for Prob0.

Like all the precomputation algorithms, Prob0 uses a fixpoint computation to generate its result. The key aspects of the implementation of the fixpoint are as follows. The set of states being computed is stored in a BDD sol over variables $\underline{\mathsf{x}}$. First, sol is initialised to some value (line 1). The algorithm then repeatedly computes a new set sol' based on sol (line 4) and compares it to the previous set (line 5). The iterative process is stopped when the set does not change, i.e. the BDDs sol and sol' are identical. The computation is guaranteed to terminate since there are finitely many states. The set generated by the fixpoint is then given by sol. Note that, in this case, the required result is actually the states *not* accumulated by the fixpoint, hence we negate sol before returning it.

As for the actual computation performed in each iteration, the original algorithm translates easily into BDDs. Note the close correspondence between line 4 of Figure 3.1 and line 4 of Figure 5.3:

- $R' := R \cup \{s \in Sat(\phi_1) \mid \exists s' \in R \,.\, \mathbf{P}(s,s') > 0\}$
- $\mathsf{sol}' := \mathsf{sol} \vee (\mathsf{phi}_1 \wedge \textsc{ThereExists}(\underline{\mathsf{y}}, \mathsf{T} \wedge \textsc{ReplaceVars}(\mathsf{sol}, \underline{\mathsf{x}}, \underline{\mathsf{y}}))$

Furthermore, this consists entirely of simple BDD operations which can be performed efficiently. The other important part of the algorithm, checking if the BDDs sol and sol′ are identical, can be done in constant time due to the canonicity property of (MT)BDDs and the efficient data structures used to store them, as discussed in Section 3.7.

5.3 Numerical Computation

We require algorithms to compute probabilities for the PCTL next, bounded until and until operators over both DTMCs and MDPs, and the CSL next, time-bounded until, until and steady-state operators over CTMCs. As in the previous section, there is a considerable amount of overlap between the various cases. Hence, we limit our coverage to a few examples which illustrate all of the relevant points. A complete description of all the MTBDD algorithms can be found in Appendix D.

5.3.1 The PCTL Until Operator for DTMCs

As our first example, we consider an MTBDD implementation of the numerical computation for the PCTL until operator, $\mathcal{P}_{\bowtie p}[\phi_1 \,\mathcal{U}\, \phi_2]$, over DTMCs. This algorithm was discussed in Section 3.3.1. The MTBDD version is given in Figure 5.4. It is essentially the same as the one proposed in [BCHG+97]. The probabilities are computed by the MTBDD algorithm PctlUntil. It takes as input two BDDs, phi_1 and phi_2, representing the sets of states $Sat(\phi_1)$ and $Sat(\phi_2)$, respectively, and returns an MTBDD representing the vector of probabilities $p_s(\phi_1 \,\mathcal{U}\, \phi_2)$ for each state s. It also uses the MTBDD P representing the transition probability matrix of the DTMC.

The first step (lines 1–3) determines the sets S^{no}, S^{yes} and $S^?$. This uses the precomputation algorithms Prob0 and Prob1, the implementation of which was described in the previous section. Secondly (lines 4–6), the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$ is constructed, as described in Section 3.3.1. The matrix $\mathbf{A}$ and vector $\underline{b}$ are represented by MTBDDs A and b respectively. We first build the matrix $\mathbf{P}'$ (represented by P′) which is equal to the matrix $\mathbf{P}$ but with the rows corresponding to states not in $S^?$ set to zero. This is done by a pointwise multiplication, using the Apply operator, of P and $\mathsf{s}_?$, the

PctlUntil($\mathsf{phi}_1, \mathsf{phi}_2$)	
1.	$\mathsf{s_{no}} := \textsc{Prob0}(\mathsf{phi}_1, \mathsf{phi}_2)$
2.	$\mathsf{s_{yes}} := \textsc{Prob1}(\mathsf{phi}_1, \mathsf{phi}_2, \mathsf{s_{no}})$
3.	$\mathsf{s_?} := \neg(\mathsf{s_{no}} \vee \mathsf{s_{yes}})$
4.	$\mathsf{P}' := \mathsf{s_?} \times \mathsf{P}$
5.	$\mathsf{A} := \textsc{Identity}(\underline{\mathsf{x}}, \underline{\mathsf{y}}) - \mathsf{P}'$
6.	$\mathsf{b} := \mathsf{s_{yes}}$
7.	$\mathsf{probs} := \textsc{SolveJacobi}(\mathsf{A}, \mathsf{b}, \mathsf{b})$
8.	**return** probs

Figure 5.4: The PctlUntil algorithm

latter representing $S^?$. The matrix $\mathbf{A} = \mathbf{I} - \mathbf{P}'$ is then constructed using the Identity and Apply functions. In line 7, the solution of the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$ is computed, using the Jacobi iterative method. This constitutes the bulk of the work and is contained in a separate algorithm, SolveJacobi.

As stated in Section 3.3, we use iterative methods for solving linear equation systems, rather than alternative, direct methods such as Gaussian elimination or L/U decomposition. This is because we are aiming to study large probabilistic models, which will produce very large linear equation systems. Direct methods usually require modifications to the matrix $\mathbf{A}$, which are costly both in terms of space and time.

This argument applies regardless of the data structure being used. In our case, however, it is particularly relevant. Work by [BFG+93] showed that MTBDDs are very poorly suited to methods such as Gaussian elimination. As we saw in the previous chapter, the effectiveness of MTBDDs relies heavily on them being used to store regular, structured information. Modifications to the matrix $\mathbf{A}$, such as those made by Gaussian elimination, inevitably lead to a significant loss in regularity and a consequent blow-up in the size of the MTBDD. Furthermore, the operations required to perform these modifications work on individual elements, rows and columns of the matrix. These are particularly difficult to implement on inherently recursive data structures such as MTBDDs. The iterative methods we use, on the other hand, do not modify the matrix throughout the computation and can be implemented with matrix-vector multiplication, for which efficient MTBDD algorithms exist.

The problem of implementing iterative solution methods using MTBDDs was first considered in [HMPS94], which implemented steady-state probability calculation using the Power method. [HMKS99] extended this work, also presenting MTBDD algorithms for the Jacobi and Gauss-Seidel methods. In Figure 5.5, we give the function SolveJacobi, the MTBDD implementation of the Jacobi method.

```
SolveJacobi(A, b, init)
 1.  d := Abstract(max, y, A × Identity(x, y))
 2.  A′ := A × Const(−1) × ¬Identity(x, y)
 3.  sol := init
 4.  done := false
 5.  while (done = false)
 6.      sol′ := MVMult(A′, sol)
 7.      sol′ := sol′ + b
 8.      sol′ := sol′ ÷ d
 9.      if (MaxDiff(sol, sol′) < ε) then
10.          done := true
11.      endif
12.      sol := sol′
13.  endwhile
14.  return sol
```

Figure 5.5: The SolveJacobi algorithm

The algorithm can be compared to the description of the Jacobi method we gave in Section 3.5. Note that there were two alternatives presented there: one expressed in terms of operations on individual matrix elements; and one in terms of matrix-vector multiplication. We select the latter because, as described above, it is far more efficient to implement in MTBDDs. The first three lines of Figure 5.5 set up the MTBDDs A', d and sol which will be used in the main iterative loop. In terms of the description of the Jacobi method in Section 3.5.1, A' corresponds to the matrix $\mathbf{L} + \mathbf{U}$ and d stores the diagonal values from the matrix $\mathbf{D}$. The MTBDD sol represents the solution vector.

The main part of SolveJacobi is the loop in lines 4–13. Each iteration computes the next approximation to the solution vector, sol', from the previous one, sol. This is done with one matrix-vector multiplication and a pointwise addition and division on a vector. Each iteration also contains a convergence check which compares sol and sol' to determine whether or not the method should be terminated. Various stopping criteria can be used, as discussed in Section 3.5. We check if the maximum relative difference between elements of the two vectors is below some threshold ε. We assume the presence of a function MaxDiff($\mathsf{v}_1, \mathsf{v}_2$) which computes this difference between two vectors represented by MTBDDs v_1 and v_2. This could be done with basic MTBDD operations, e.g. FindMax(Abs(($\mathsf{sol}' - \mathsf{sol}$) ÷ sol')). In fact, there are also built-in operations in the CUDD package which can be used to compute this directly from the MTBDDs.

The JOR method can be implemented as a simple modification of the Jacobi method (see Appendix D for the exact details). To encode Gauss-Seidel though, or the related SOR

```
PctlUntilMax(phi_1, phi_2)
 1.  s_no := Prob0A(phi_1, phi_2)
 2.  s_yes := Prob1E(phi_1, phi_2)
 3.  s_? := ¬ (s_no ∨ s_yes)
 4.  Steps' := s_? × Steps
 5.  probs := s_yes
 6.  done := false
 7.  while (done = false)
 8.        probs' := MVMult(Steps', probs)
 9.        probs' := Abstract(max, z, probs')
10.        probs' := probs' + yes
12.        if (MaxDiff(probs, probs') < ε) then
13.              done := true
14.        endif
15.        probs := probs'
16.  endwhile
16.  return probs
```

Figure 5.6: The PctlUntilMax algorithm

method, is more difficult. For efficiency reasons, we must rely on the matrix formulation of the method given in Section 3.5, which is again based on matrix-vector multiplication. The implementation of this in MTBDDs is considered in [HMKS99] but the need to compute a matrix inverse and the amount of extra work this entails make it an unattractive option. This is unfortunate, because these two methods usually require significantly less iterations to converge, which could have a marked effect on the overall time required for model checking.

5.3.2 The PCTL Until Operator for MDPs

As our second example, we describe the model checking algorithm for the PCTL until operator, $\mathcal{P}_{\bowtie p}[\phi_1 \ \mathcal{U} \ \phi_2]$, over MDPs. We consider the case where $\bowtie p$ defines an upper bound, and hence the algorithm computes the probabilities $p_s^{max}(\phi_1 \ \mathcal{U} \ \phi_2)$. In this case, it makes no difference whether we consider all adversaries or fair adversaries only. The model checking algorithm was given in Section 3.3.2. The MTBDD version, PctlUntilMax, can be seen in Figure 5.6. We first presented this in [dAKN$^+$00].

As was the case in the previous section, there exist alternative, direct methods to perform this model checking problem. It can be reformulated as a linear programming (LP) problem, as detailed in Section 3.3.2, and solved with classic LP techniques such as the Simplex algorithm. We investigated the applicability of MTBDDs to the Simplex

method in [KNPS99] and found that the situation is similar to that of Gaussian elimination: its reliance on access to individual elements, rows and columns of the matrix makes it impractical for symbolic implementation.

The overall structure of PctlUntilMax is very similar to PctlUntil from the previous section: firstly, we use precomputation algorithms to determine the sets S^{no}, S^{yes} and $S^{?}$; secondly, we make modifications to the MTBDD representing the model; and thirdly, we execute an iterative method using this modified MTBDD.

Note that an MDP is represented by a transition function *Steps*, not a transition matrix, as is the case for a DTMC. As described in Section 3.3.2 though, from an implementation point of view it is useful to think of *Steps* as a non-square matrix **Steps**, where there are several rows corresponding to each state of the MDP. The key component of each iteration of the MDP model checking algorithms then becomes a matrix-vector multiplication using this non-square matrix.

We can use the same idea here. The MTBDD Steps, representing the MDP, is over row variables $\underline{x}$, column variables $\underline{y}$, and nondeterministic variables $\underline{z}$. If we instead treat Steps as representing a matrix with $\underline{x}$ and $\underline{z}$ for row variables and $\underline{y}$ for column variables, we can perform this operation using the MTBDD MVMult function. This is done in line 8 of Figure 5.6. The other important step, computing the maximum probability for each state, can be carried out easily using the Abstract$(\max, \cdot, \cdot)$ operator. The convergence check is performed as in the previous PctlUntil algorithm.

5.3.3 Other Operators

The implementation of the numerical computation for the remaining PCTL and CSL operators is not particularly different from the ones described above. PCTL bounded until for DTMCs and MDPs requires a fixed number of iterations similar to those for the unbounded variants and PCTL next needs a single such iteration. For CSL, the steady-state operator requires solution of a linear equation system, which can be done as for PCTL until formulas over DTMCs. The calculations for the CSL time-bounded until operator are slightly more involved but the basic operations required are no different. As discussed in Chapter 2, symbolic implementations of the computation required for CSL model checking were first proposed in [HMKS99, BKH99].

The full versions of all algorithms for PCTL and CSL model checking of DTMCs, MDPs and CTMCs are in Appendix D. Crucially, the overall format of the methods is the same in all cases: a number of initialisation steps followed by one or more iterations, the key constituent of each being the multiplication of a matrix and a vector.

5.4 Results

The MTBDD-based model checking algorithms for PCTL and CSL described in the previous sections have all been implemented in the PRISM model checker. Using the tool, we have applied the techniques to a number of case studies. This section presents some of the results of this work. We assess the performance of our symbolic implementation and, where applicable, compare it to that of equivalent, explicit approaches based on sparse matrices. Since most interesting properties for our case studies will make use of the probabilistic operators of PCTL and CSL, our analysis focuses on these.

We begin by considering cases where model checking can be done entirely with precomputation algorithms, typically when we are considering qualitative formulas where the bound p is 0 or 1. There are actually many case studies where this is true. One of the most common sources is models of randomised distributed algorithms, which give rise to MDPs. Since the algorithms are randomised, it is usually necessary to verify properties such as "the probability that the algorithm eventually terminates is 1", weaker than the more usual "the algorithm will always eventually terminate".

In Table 5.1, we present results for three such case studies: Rabin's randomised mutual exclusion [Rab82], Lehmann and Rabin's randomised dining philosophers [LR81], and the coin protocol from Aspnes and Herlihy's randomised consensus protocol [AH90]. In each case, we model check a PCTL property of the form $\mathcal{P}_{\geq 1}[\Diamond \phi]$, which reduces to an until formula, and compute minimum probabilities over fair adversaries only. Hence, the precomputation algorithm is used is Prob0A. The actual properties checked and details of the model parameters (N and K in the table) can be found in Appendix E.

For each example, we give the size of the MDP (number of states), the number of fixpoint iterations performed by the precomputation algorithm and the total time required for model checking. In these, and all the other results presented in this thesis, we measure actual (wall) time, running on a 440 MHz 512 MB Sun Ultra10.

We see from the table that, for these examples, model checking is fast and efficient, in some cases for extremely large models (up to 10^{13} states). We have not attempted to compare these results with an explicit implementation. It is clear, though, from the model sizes given above and the comparisons presented in Chapter 4, that many of these models could not even be constructed explicitly due to memory limitations.

Although the success of these techniques is encouraging, particularly because they can be applied to many randomised distributed algorithm case studies, in this thesis we are more concerned with the problem of implementing numerical computation. Since the former only requires BDD fixpoint algorithms on state transition systems, it is similar to existing, symbolic implementations of non-probabilistic model checking, which have

Model	Parameters	States	Iterations	Time (sec.)
Mutual exclusion (N)	3	10,744	4	0.50
	4	201,828	4	1.13
	5	6,769,448	4	1.67
	6	1.3×10^8	4	3.62
	8	4.5×10^{10}	4	4.92
	10	4.7×10^{13}	4	12.8
Dining philosophers (N, K)	$3, 4$	28,940	9	1.54
	$3, 5$	47,204	8	1.86
	$3, 6$	69,986	8	2.87
	$3, 8$	129,254	8	4.78
	$4, 4$	729,080	11	31.4
	$4, 5$	1,692,144	11	26.2
	$4, 6$	3,269,200	10	40.3
	$4, 8$	8,865,024	9	80.7
Coin protocol (N, K)	$2, 8$	1,040	53	0.04
	$4, 8$	84,096	105	0.57
	$6, 8$	4,612,864	157	2.93
	$8, 8$	2.2×10^8	209	10.3
	$10, 6$	7.6×10^9	201	20.4

Table 5.1: Results for precomputation-based model checking

Model	Parameters	States	Iterations	Time per iteration (sec.)	
				MTBDD	Sparse
Coin protocol (N, K)	$2, 8$	1,040	6,132	0.02	0.0003
	$4, 8$	84,096	21,110	0.34	0.04
	$6, 8$	4,612,864	42,967	1.83	3.37
	$8, 8$	222,236,672	70,669	4.56	-
	$10, 6$	7,598,460,928	63,241	9.79	-
FireWire (N)	200	68,185	169	0.01	0.02
	400	220,733	375	0.02	0.07
	600	375,933	581	0.04	0.12
	800	531,133	789	0.05	0.17
	1,000	686,333	995	0.06	0.22
	2,000	1,462,333	2,027	0.11	0.47
	3,000	2,238,333	3,015	0.17	0.71
Tandem queue (N)	500	501,501	534	0.12	0.22
	1,000	2,003,001	1,049	0.15	0.89
	1,500	4,504,501	1,563	0.20	1.96
	2,000	8,006,001	2,077	0.25	-
	3,000	18,009,001	3,103	0.56	-
	4,000	32,012,001	4,128	0.47	-
	5,000	50,015,001	5,153	0.48	-
	6,000	72,018,001	6,177	0.57	-
	7,000	98,021,001	7,201	0.61	-

Table 5.2: Positive results for MTBDD-based numerical computation

already received considerable attention in the literature. Hence, we concentrate on situations where precomputation algorithms are either not applicable or where they can be applied but numerical computation is still required. Unsurprisingly, these cases require considerably more effort.

We start by highlighting a number of examples where our MTBDD implementation performs extremely well. Table 5.2 present results for these. The case studies in question are the coin protocol of [AH90], the FireWire root contention protocol model of [SV99], and the tandem queueing network of [HMKS99]. The first two are MDP models, for which we model check a PCTL until formula. The third is a CTMC model, where we check a CSL time-bounded until formula. Appendix E gives more details.

For each model, Table 5.2 shows its size (number of states), the number of iterations required for model checking, and the time required for both the MTBDD and sparse matrix implementations. For the latter, we give the average time per iteration. This allows for a good, general comparison between the two data structures, regardless of the

actual iterative method required or its convergence on a particular example. Note that the time required for other parts of model checking, such as the initial setup up of the iterative solution method, is always negligible in comparison. A dash in the table indicates that model checking could not be completed because of memory constraints.

For each of the three case studies, MTBDDs are faster than sparse matrices on all but the very smallest examples. In addition, for both the coin protocol and tandem queue case studies, the MTBDD implementation can handle considerably larger models. The reason for this is the differing memory usage of the two approaches.

Unfortunately, it is not feasible to carry out a detailed comparison in this respect. While the storage requirements for the matrix and solution vectors are easy to compute for the sparse matrix implementation, it is difficult to determine exact figures for the MTBDD version. Each individual operation performed creates new MTBDDs. These are kept in a shared data structure so their storage always overlaps to some extent. Also, memory freed by MTBDDs becoming redundant is only released periodically via garbage collection. Furthermore, a significant portion of the space required is taken up by the cache for the computed table, whose size is altered dynamically by the MTBDD package.

What we can see from our empirical results, though, is that we have examples where MTBDDs can be used but where sparse matrices cannot. On the largest coin protocol model, for example, which has 7.5 billion states, the MTBDD computation could be performed comfortably in 512 MB of RAM, while the storage of the matrix alone would have required $1,300$ GB for the explicit implementation.

The results presented in Table 5.2 are again very encouraging. Unfortunately, they are not typical. More often than not, the MTBDD implementation is easily outperformed by its sparse matrix counterpart. Table 5.3 illustrates the performance of the two approaches on some typical examples: two CTMC models, the Kanban manufacturing system of [CT96] and the cyclic server polling system of [IT90]; a DTMC model, the bounded retransmission protocol (BRP) of [HSV94]; and an MDP model, the randomised dining philosophers algorithm of Lehmann and Rabin [LR81]. For the CTMCs, we compute the steady-state probabilities in order to model check a CSL steady-state formula; for the DTMC, we check a PCTL until property; and for the MDP, we check a PCTL bounded until formula. As above, we give the size of the model, the number of iterations required and the average time per iteration using MTBDDs and sparse matrices.

Here, the MTBDD implementation is always much slower than the sparse matrix equivalent, by several orders of magnitude in the worst cases. MTBDDs also perform poorly in terms of memory. While the upper limit for models which can be analysed by the sparse matrix implementation remains the same (a few million states), the MTBDD version runs out of memory on examples with as few as $70,000$ states.

Model	Parameters	States	Iterations	Time per iteration (sec.)	
				MTBDD	Sparse
Kanban System (N)	1	160	101	0.03	0.001
	2	4,600	166	2.22	0.002
	3	58,400	300	45.5	0.05
	4	454,475	466	-	0.41
	5	2,546,432	663	-	2.70
Polling system (N)	8	3,072	310	0.31	0.001
	10	15,360	406	13.1	0.01
	12	73,728	505	-	0.05
	14	344,064	606	-	0.30
	16	1,572,864	709	-	1.56
BRP (N)	500	27,505	3,086	0.38	0.01
	1,000	55,005	6,136	0.67	0.01
	1,500	82,505	9,184	0.99	0.02
	2,000	110,005	12,230	1.26	0.03
	2,500	137,505	15,274	1.53	0.04
Dining philosophers (N, K)	$3, 4$	28,940	22	0.12	0.01
	$3, 5$	47,204	26	0.14	0.01
	$3, 6$	69,986	30	0.16	0.02
	$3, 8$	129,254	38	0.22	0.04
	$4, 4$	729,080	22	1.24	0.19
	$4, 5$	1,692,144	27	2.57	0.48
	$4, 6$	3,269,200	31	4.62	1.00

Table 5.3: Negative results for MTBDD-based numerical computation

5.5 Analysis

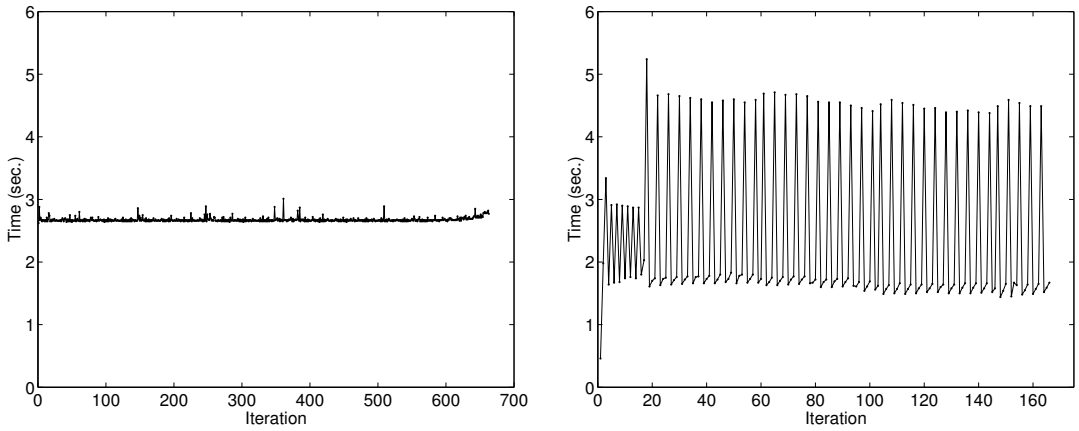
We now analyse our results in more detail. In particular, we want to determine why MTBDDs are good for some cases of numerical computation and bad for others. We begin by trying to identify the factors which cause MTBDDs to perform poorly. In the previous section, we measured the performance of an implementation by the average time per iteration. Whilst this is a good measure for comparisons, it is also revealing to examine the *actual* time required for each iteration.

In Figures 5.7 and 5.8, we plot this information for the steady-state probability computation of two CTMCs. Figures 5.7 and 5.8 show the Kanban system and polling system examples, respectively. In both cases, (a) gives the times for sparse matrices and (b) the times for MTBDDs. We select models such that the average time per iteration is approximately the same for the two implementations. We are not concerned with the actual times, only the pattern in iteration time.

The most obvious difference is that the iteration times for sparse matrices remain relatively constant, whilst those for MTBDDs vary greatly. In fact, this variation in the latter is the result of two factors. The repeated jumps which can be observed in the graph are due to the fact that the MTBDD package periodically performs garbage collection and then reorganises the unique table. This is done when the package determines that there are a significant number of dead nodes which should be freed. Typically, entries in the computed table which correspond to these nodes are also removed. All other MTBDD operations are paused while this process is carried out. This phenomenon is particularly clear in Figure 5.7(b), where it can be seen to occur every 3 or 4 iterations.

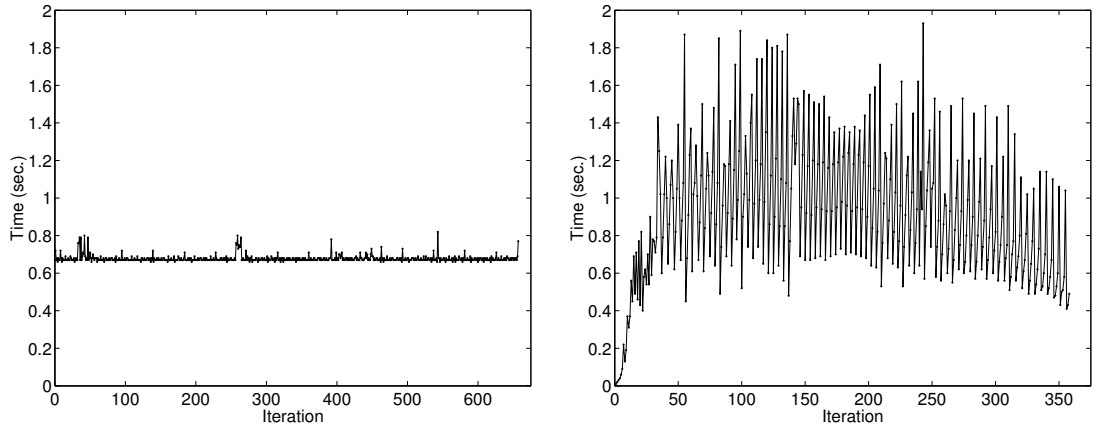
The second characteristic which can be observed is an increase in iteration time as the computation progresses. This is especially evident in Figure 5.8(b) where the time can be seen to initially be almost zero and then grow very rapidly. This can also be seen in Figure 5.7(b), where the increase is even more sudden.

An examination of the iterative algorithms we perform for model checking suggests that the most costly operation performed is the multiplication of a matrix and a vector. Empirical results confirm this. For both the sparse matrix and MTBDD implementations, the time complexity of this operation depends purely on the size of the data structures storing the matrix and the vector. We know that the matrix is never modified: this is one of the fundamental properties of iterative solution methods. The vector, however, is updated at each iteration. In the sparse matrix implementation, it is stored in an array of fixed size so these alterations will have no effect on the size of the data structure. By contrast, the MTBDD representation of the iteration vector may change significantly. The size of the MTBDD (and hence the time required for the multiplication operation)



(a) (b)

Figure 5.7: Actual iteration times for the Kanban system case study: (a) Sparse matrices ($N = 5$) (b) MTBDDs ($N = 2$)



(a) (b)

Figure 5.8: Actual iteration times for the polling system case study: (a) Sparse matrices ($N = 15$) (b) MTBDDs ($N = 9$)

depends on the structure and regularity of the vector. This will inevitably change as the computation progresses.

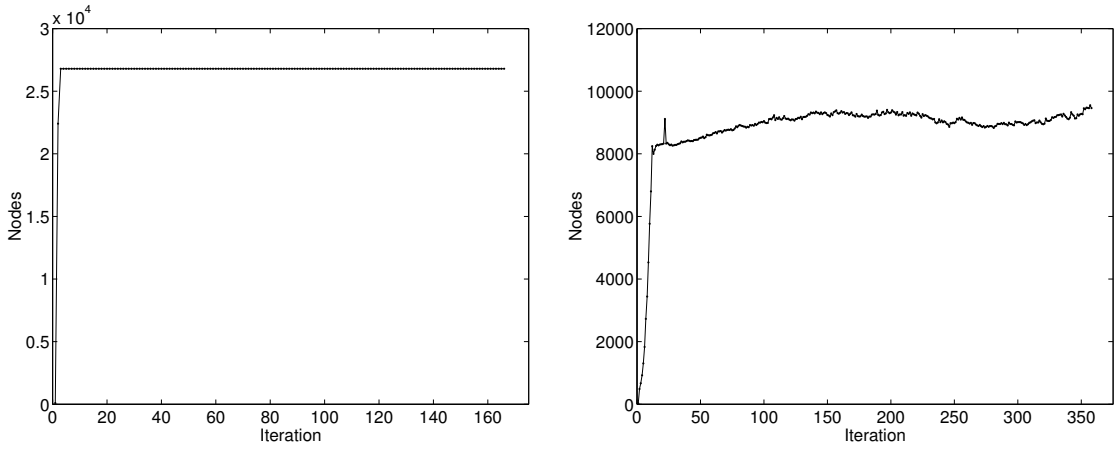
To investigate this, we now plot the size (number of nodes) of the MTBDD representing the vector for each iteration. Figure 5.9 shows this for the same computations as in Figures 5.7(b) and 5.8(b). There is a clear correspondence between the growth in the MTBDD size for the vector and the increase in time per iteration. Note, for example, how in the Kanban system example, the MTBDD size jumps almost immediately to its maximum value and remains the same, as is reflected in the iteration times. We conclude that it is this growth of the MTBDD for the vector which is responsible for the slow run-times observed in the symbolic implementation. From the size of the MTBDDs involved, it is clear that this is also the reason that the implementation runs out of memory for the larger examples in Table 5.3. In addition to the storage required for the MTBDDs themselves, the memory required for the computed table will also increase.

One likely factor to explain the increase in the size of the MTBDD is the number of terminals it contains. This corresponds to the number of distinct values in the iteration vector. We would expect this to increase as the computation progresses. This is indeed the case, as confirmed by the graphs in Figure 5.10, where we plot the number of terminals in the vector MTBDD for each iteration. Note the close correspondence between these two graphs and the two in Figure 5.9.

The correlation between these two sets of statistics is unsurprising. Compact MTBDDs are obtained by exploiting structure and regularity. Clearly, as the number of distinct terminal nodes increases, the capacity for sharing will decrease. For the Kanban example (Figure 5.10(a)), the number of terminals quickly reaches 4,600, equal to the number of states in the model. This is actually the worst possible case for MTBDD size. As well as affecting the number of nodes, these factors will also reduce the chance of duplicate operations being performed, severely limiting the effectiveness of the cache, usually one of the main sources of efficiency in MTBDD computations.

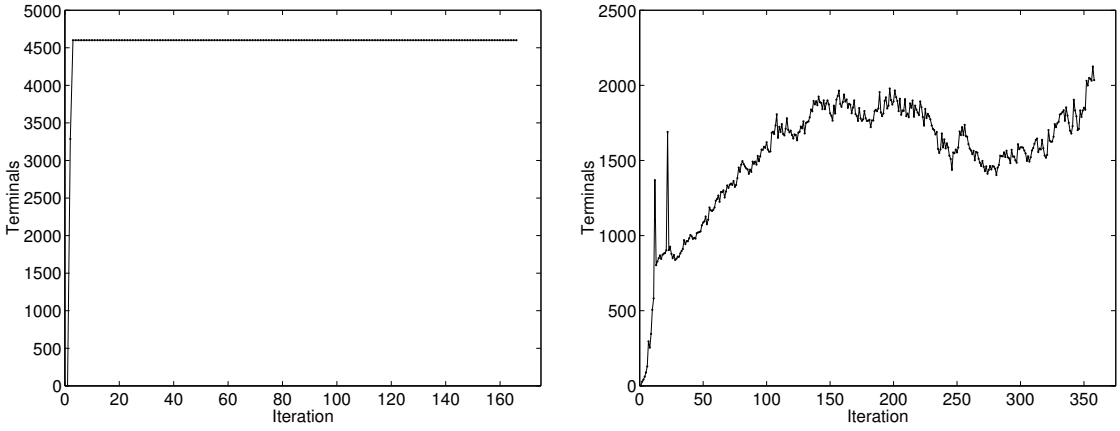
For comparison, we plot the equivalent graphs for some of the cases where MTBDDs performed well. Figures 5.11(a) and (b) show plots of the number of terminals in the iteration vector MTBDD for two examples from Table 5.2: the FireWire root contention protocol model ($N = 400$) and the coin protocol model ($N = 2, K = 8$).

The most obvious difference is that the number of terminals for these MTBDDs is much lower than in the previous examples. For the FireWire example, this number is extremely small, remaining below 6 for the whole computation. This is despite the fact that the model has more than 200,000 states. For the coin protocol, the number is slightly higher, levelling off at about 130, but this is still significantly less than in our examples where MTBDD perform badly. Note that, in this second example, we have only plotted



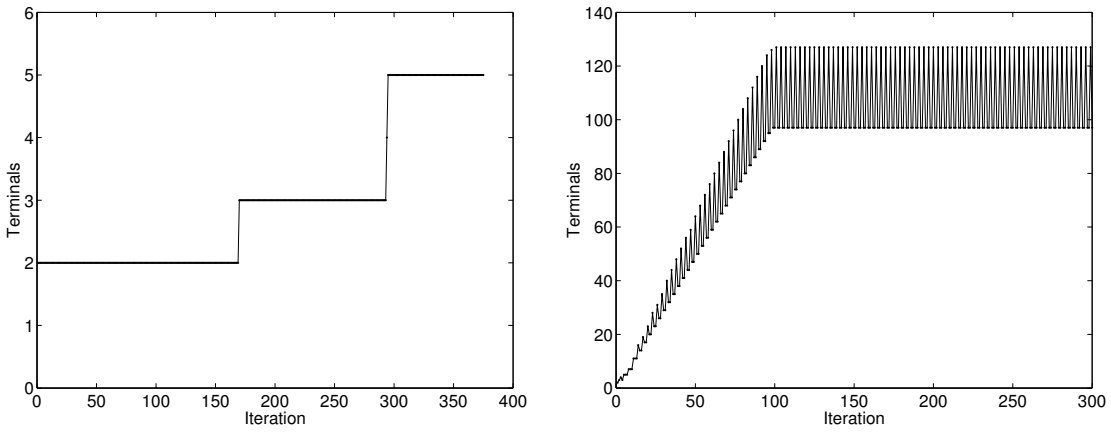
(a) (b)

Figure 5.9: MTBDD size for the iteration vector during steady-state computation: (a) Kanban system ($N = 2$) (b) Polling system ($N = 9$)



(a) (b)

Figure 5.10: Number of terminals at each iteration for steady-state computation: (a) Kanban system ($N = 2$) (b) Polling system ($N = 9$)



(a) (b)

Figure 5.11: Number of terminals at each iteration for PCTL model checking: (a) FireWire model ($N = 400$) (b) Coin protocol ($N = 2, K = 8$)

results for the first 300 iterations. The whole computation actually takes more than 6,000 iterations, but we have verified that the number of terminals grows no larger in this period. The curve of the graph where the number of terminals levels off is also suspiciously sharp, suggesting that other factors, such as round-off error might be keeping the figure artificially low. To rule this out, we performed the same computation using arbitrary precision arithmetic and obtained identical results.

We observe that the models on which MTBDDs perform impressively are usually MDPs. In Chapter 4, we found that the compactness of the MTBDD representation was better for MDPs than other types of model. We reasoned that this was due to the fact that our primary source of MDP case studies was randomised distributed algorithms. The probabilistic behaviour in such systems is often limited to a small number of simple, random choices such as coin tosses. Therefore, in the matrices representing the corresponding MDPs, there are very few distinct values and most most non-zero entries are 1. This results in compact, regular MTBDDs with a small number of terminals. It seems that, consequently, the solution vector in these cases can also remain relatively structured and compact, causing a much slower growth in size and improved time and memory requirements.

Potential Solutions

We conclude this chapter by describing some some potential solutions we have considered for alleviating the performance problems of MTBDD-based numerical computation. We

first look at approaches taken in the non-probabilistic case. BDD-based fixpoint computations, such as those used in this thesis for reachability (Section 4.3.2) and precomputation algorithms (Section 5.2), or elsewhere for non-probabilistic model checking, can be seen as an analogue of our MTBDD-based numerical methods. Both are based on iterative computations: in our case, using MTBDDs to represent a real-valued matrix and a real-valued vector; in the non-probabilistic case, using BDDs to represent a transition relation and a set of states.

Although the problems are not as severe in the non-probabilistic case, it is frequently still necessary to try and curb the growth of the BDDs during the computation. A common approach is to experiment with *dynamic variable reordering*. While an initial ordering for the BDD variables is usually chosen to minimise the size of the transition relation, if the size of the BDD for the set of states becomes unmanageable, it is often worth reordering the variables to reduce the BDD size. The literature contains many studies into heuristics for determining when and how to perform such reordering. Unfortunately, in our case, the lack of structure in the solution vector which causes the growth in MTBDD size seems to be influenced mainly by the number of terminals. This will remain the same, regardless of the ordering chosen.

This suggests that a good direction to take would be to try and reduce the number of terminals in the vector MTBDD. One possible technique would be to round off the solution vector to some lower precision at each iteration. In this way, terminals with distinct (but approximately equal) values, perhaps the result of round-off errors, would be merged, reducing the size of the MTBDD. We implemented a prototype version of this idea and discovered that, not only was there no dramatic decrease in MTBDD size, but in several cases the iterative solution method would now either not converge or would converge to the wrong answer. It is well known that the limited precision of floating point arithmetic can cause round-off errors in computations. It seems that artificially limiting the precision further only accentuates these problems.

Chapter 6

A Hybrid Approach

In the previous two chapters, we demonstrated the applicability of MTBDDs to the construction and analysis of probabilistic models. MTBDDs proved to be extremely well suited to building and storing large, structured models. Furthermore, model checking of qualitative properties could be performed efficiently on these models using reachability-based precomputation algorithms. When we focused on model checking of quantitative properties, based on numerical computation, we also witnessed positive results, finding several instances where the MTBDD implementation easily outperformed the sparse matrix equivalent. In the majority of these cases, though, the opposite was true and the symbolic implementation performed poorly.

We successfully identified the main cause of these problems. All the numerical calculations required for model checking are implemented using iterative methods, based on successive computations of an approximation to a solution vector. The update to this vector at each iteration is based on a multiplication by a matrix, derived in some way from the transition matrix of the model. Despite the fact that the MTBDD representation for this matrix is often extremely compact, we found that the size of the same representation for the vector usually increases rapidly as the computation progresses and the vector becomes less and less regular. Consequently, the time required to perform operations on the vector grows large, slowing the computation down. Furthermore, in many instances, the increase in memory consumption actually makes model checking infeasible.

By comparison, in our explicit implementation, the solution vector is stored in an array. The entries of the vector can be accessed and modified much more quickly when represented in this fashion. Furthermore, the size of the data structure remains constant as the computation progresses. The disadvantage of this implementation, though, is that the memory requirements for the matrix can become unwieldy, limiting the size of model checking problem to which the techniques can be applied.

In this chapter, we present a novel, hybrid approach to the implementation of numerical solution. We will represent the matrix as an MTBDD, allowing us to benefit from the compact storage schemes developed in Chapter 4 and also to make use of the more efficient aspects of our symbolic implementation such as model construction and precomputation algorithms. The iteration vector, though, will be stored in an array, as in the explicit approach. The problem we now face is implementing numerical iterative methods using these two fundamentally different data structures.

6.1 Mixing MTBDDs and Arrays

We begin by considering the numerical computation required for model checking DTMCs and CTMCs. The complications introduced by the case for MDPs will be addressed at a later stage. As we observed when developing the pure MTBDD-based implementation in Chapter 5, model checking for all probabilistic operators of PCTL and CSL can be performed using iterative methods, the key step of each iteration being the multiplication of a matrix and a vector.

The only restriction implied by this is that, when solving a system of linear equations (for PCTL until or CSL steady-state formulas), we limit ourselves to the Jacobi and JOR methods, rather than more efficient alternatives such as Gauss-Seidel or SOR. In fact, we will later discuss ways of relaxing this restriction. For now, though, we are free to focus on the simpler problem of performing a single operation, matrix-vector multiplication, knowing that this is sufficient to implement full PCTL and CSL model checking. The problem we address in the following sections is how to perform this operation when the matrix is stored as an MTBDD and the vector as an array.

6.1.1 A First Algorithm

Our approach will be to emulate the matrix-vector multiplication algorithm for sparse matrices given in Figure 3.13. This multiplies a matrix $\mathbf{A}$, stored in arrays *row*, *col* and *val*, by a vector, stored in an array b, and places the result in an array *res*. Crucially, we observe that the algorithm uses each element of $\mathbf{A}$ exactly once during this process.

In a sparse matrix setting, these elements can be obtained by simply reading along the three arrays, *row*, *col* and *val*. In this case, the matrix entries are obtained in order, row by row, but this is not actually required for matrix-vector multiplication. The same result can be achieved by initially setting every element of *res* to zero, and then performing the operation $res[r] := res[r] + v \times b[c]$ for all entries "$(r, c) = v$" of $\mathbf{A}$ in any order.

With this in mind, we observe that all the non-zero entries of a matrix, as represented

by an MTBDD, can be extracted via a depth-first traversal of the data structure, i.e. an exhaustive exploration of all paths through the MTBDD which lead to a non-zero terminal. The entries will not be produced in any meaningful order, but this represents the quickest way of extracting them all in a single pass.

Figure 6.1 shows the algorithm TraverseMtbdd, which performs this depth-first traversal. It does so by recursively splitting the problem into the traversal of four smaller MTBDDs. This corresponds to the decomposition of the matrix represented by the MTBDD into four submatrices, as previously illustrated in Figure 3.19.

The recursion bottoms out either when the algorithm comes across a submatrix containing no non-zero entries, represented by the zero terminal, or when it locates a matrix entry, i.e. when it reaches a non-zero terminal. In the latter case, the function UseMatrixEntry(r, c, v) is called, where r, c and v correspond to the row index, column index and value of the matrix entry found. For matrix-vector multiplication, UseMatrixEntry(r, c, v) would perform the operation $res[r] := res[r] + v \times b[c]$. We use this generic function to indicate that the traversal algorithm could just as easily be used for any operation which requires an explicit list of all matrix entries.

In a call to the algorithm TraverseMtbdd(m, i, r, c), m is the current MTBDD node, i is the current level of recursion and r and c are used to keep track of row and column indices. For a matrix represented by an MTBDD M, the algorithm is called at the top level as TraverseMtbdd$(\mathsf{M}, 1, 0, 0)$. Recall from Section 3.7 that we consider an MTBDD and the node which represents it (its root node) to be interchangeable.

The current level of recursion, i, has to be tracked explicitly in order to deal with skipped levels in the MTBDD. This corresponds to our observation in Section 3.7 that a single path through an MTBDD can correspond to several minterms and hence to several matrix entries. We assume that the matrix being represented by M uses row variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ and column variables $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$. Skipped levels can be detected by comparing the positions of the variable for the current node, $var(\mathsf{m})$, and that of either x_i or y_i in the MTBDD variable ordering.

The key part of the algorithm is the calculation of the row and column indices. In an MTBDD, each index is associated with its Boolean encoding, i.e. an element of $\mathbb{B}^n$. An explicit representation such as an array is indexed by integers. We need a way to convert one to the other. We will assume that the rows and columns of the matrix are indexed from 0 to $2^n - 1$ and are encoded using the standard binary representation for integers. By noting the path that we have taken through the MTBDD and by summing the appropriate powers of two when we take a *then* edge, we can compute the indices as required. This is exactly what TraverseMtbdd does. Note that the computation of row and column indices is performed independently, using variables r and c, respectively.

```
TraverseMtbdd(m, i, r, c)
1.  if (m = Const(0)) then
2.      return
3.  else if (i = n + 1) then
4.      UseMatrixEntry(r, c, val(m))
5.      return
6.  endif
7.  if (var(m) > x_i) then
8.      e := t := m
9.  else
10.     e := else(m)
11.     t := then(m)
12. endif
13. if (var(e) > y_i) then
14.     ee := et := e
15. else
16.     ee := else(e)
17.     et := then(e)
18. endif
19. if (var(t) > y_i) then
20.     te := tt := t
21. else
22.     te := else(t)
23.     tt := then(t)
24. endif
25. TraverseMtbdd(ee, i + 1, r, c)
26. TraverseMtbdd(et, i + 1, r, c + 2^(n−i))
27. TraverseMtbdd(te, i + 1, r + 2^(n−i), c)
28. TraverseMtbdd(tt, i + 1, r + 2^(n−i), c + 2^(n−i))
```

Figure 6.1: The TraverseMtbdd algorithm

When a *then* edge is followed from a node at the ith level of recursion, 2^{n-i} is added to the appropriate variable, r or c.

Alternatively, we can consider TraverseMtbdd to function as follows. Each call to TraverseMtbdd(m, i, r, c) extracts the submatrix represented by node m and computes the corresponding local indices of its elements, relative to that submatrix. The actual row and column indices are calculated by adding the offsets r and c, respectively, to the local indices. Since, at this level of recursion, the algorithm splits a square matrix of size 2^{n-i+1} into four submatrices of size 2^{n-i}, the offsets, r and c, for the next level are computed by adding 2^{n-i} where appropriate.

A simple example should demonstrate this process more clearly. Figure 6.2 shows a 4×4 matrix $\mathbf{M}$ and the MTBDD M which represents it. Note that $\mathbf{M}$ was derived from a model with an unreachable state and the corresponding row and column have been filled with zeros. For clarity, we mark these as '$-$'s, not '0's. This issue is irrelevant now, but will be important when we reuse the example later.

Figure 6.3 gives a table, explaining how the TraverseMtbdd algorithm works on this example. Each row of the table corresponds to a single matrix entry. These are listed in the order in which TraverseMtbdd would have extracted them. The first five columns of the table describe the path taken through the MTBDD, i.e. the value of each MTBDD variable and of the terminal reached. The next four columns give the components of the sums to compute the row and column indices. The final column shows the resulting matrix entry.

The table also illustrates why the methods we present here are only applicable to iterative methods which can be implemented with matrix-vector multiplication. The order in which the matrix entries are extracted is determined by the interleaving of the row and column variables in the MTBDD. The entire top-left quadrant of the matrix will be extracted before any entries of the top-right quadrant are. Hence, we generally do not obtain a whole row at a time. Notice, for example, in Figure 6.3, that entry $(1, 1)$ is extracted before entry $(0, 3)$. Iterative methods which rely on utilising rows or columns one at a time, such as Gauss-Seidel, could not be implemented in this way.

One way to resolve this would be to opt for a non-interleaved MTBDD variable ordering such as $\mathsf{x}_1 < \cdots < \mathsf{x}_n < \mathsf{y}_1 < \cdots < \mathsf{y}_n$ but, as we saw in Section 4.1.2, this is not feasible because the size of the MTBDD becomes unmanageable. Alternatively, we could extract every row or column individually, performing one traversal of the MTBDD for each. It seems likely, though, that this would result in a considerable slow-down.

There remains one crucial problem with the approach outlined in this section. Recall from Chapter 4 that our MTBDD encoding of a model typically results in the inclusion of unreachable states. Hence, the matrix represented by the MTBDD will have empty

$$\mathbf{M} = \begin{pmatrix} 0 & 3 & - & 6 \\ 0 & 3 & - & 4 \\ - & - & - & - \\ 5 & 0 & - & 4 \end{pmatrix}$$

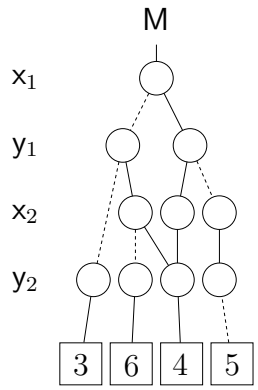


Figure 6.2: A matrix $\mathbf{M}$ and the MTBDD M which represents it

Path					Indices				Entry of $\mathbf{M}$
x_1	y_1	x_2	y_2	f_{M}	x_1	y_1	x_2	y_2	
0	0	0	1	3	-	-	-	1	$(0,1) = 3$
0	0	1	1	3	-	-	1	1	$(1,1) = 3$
0	1	0	1	6	-	2	-	1	$(0,3) = 6$
0	1	1	1	4	-	2	1	1	$(1,3) = 4$
1	0	1	0	5	2	-	1	-	$(3,0) = 5$
1	1	1	1	4	2	2	1	1	$(3,3) = 4$

Figure 6.3: Illustration of the TraverseMtbdd algorithm on M

rows and columns corresponding to these states. The number of such states is potentially large and, for some of our case studies, is orders of magnitude larger than the number of reachable states. The traversal algorithm presented above takes no account of this, and so effectively we are dealing with a much larger matrix than necessary.

Normally, we are happy to do this since it results in a compact MTBDD representation. The difficulty here, though, is that vectors, and hence the arrays storing them, need to be of the same size. Since these are stored explicitly, this puts unacceptable, practical limits on the size of problems with which we can deal. Note that this is not a problem with sparse matrices, because we assume that the sparse matrix (and hence the solution vector) is only over reachable states.

6.1.2 Our Improved Method

What we need is a way of determining the actual row and column index of each matrix element, in terms of reachable states only. We will assume that the set of reachable states, say $\hat{S} \subseteq S$, is indexed from 0 to $|\hat{S}|-1$ and that these are in the same order that they were in S. This means we can adopt a similar approach to the previous section, recursively computing submatrix entries locally and using offsets to determine the actual indices.

In the previous section, we increased the offsets r and c at each level of recursion by a fixed power of two, corresponding to the size of the submatrices on the next level. Now, the increase will instead depend on the number of rows or columns in these submatrices which correspond to reachable states. Our approach will be to precompute this information and store it on the MTBDD so that it can be read off as we traverse the data structure.

More precisely, we will construct a new MTBDD from the old one, in which each node is labelled with an *offset*. For a row node, this offset will indicate how many 'reachable rows' there are in the top submatrix and, for a column node, it will indicate how many 'reachable columns' there are in the left submatrix. In both cases, the submatrix in question corresponds to taking the *else* edge (as illustrated previously in Figure 3.19). This means that, when tracing a path from the root node of the MTBDD to one of its terminals, representing some matrix entry, the actual row and column indices of this entry can be determined by summing the offsets on nodes labelled with row and column variables, respectively, from which the *then* edge was taken. In this section, we will describe this new data structure in more detail and present a modified traversal algorithm for it. In the following section, we will explain precisely how it is constructed.

We christen this data structure an *offset-labelled MTBDD*. It is essentially an MTBDD, but with two important differences. Firstly, as described above, every non-terminal node is labelled with an integer value, which will be used to compute reachable row and col-

umn indices. Secondly, the MTBDD does not need to be fully reduced. In fact, our requirements in this respect are slightly more complex, as we now explain.

Recall from Section 3.7 that there are two types of reduction performed to minimise the size of an MTBDD: merging of shared nodes, i.e. those on the same level and with identical children; and removal of nodes for which the *then* and *else* edges point to the same node, introducing one or more skipped levels. In an offset-labelled MTBDD, the second type of reduction is only permitted when both the *then* and *else* edges point to the zero terminal. This means that every edge from every node points either to a node in the level immediately below it or directly to the zero terminal. Consequently, we now have a one-to-one correspondence between paths through the offset-labelled MTBDD leading to a non-zero terminal and the entries of the matrix which it represents.

There are two reasons for doing this. Firstly, the traversal process can be sped up because we no longer need to check for skipped levels at every node. Secondly, and more importantly, we need the offsets which are used to compute matrix entry indices to be on every level and these are stored on the nodes. Note that the exception for edges going directly to the zero terminal is safe because we are only interested in extracting non-zero matrix entries. In fact, since our matrices are typically very sparse, removing this exception would slow the extraction process down considerably.

Furthermore, in offset-labelled MTBDDs the first type of MTBDD reduction, merging of shared nodes, is not compulsory. When we describe the construction process in the next section, we will see instances where it is important *not* to merge shared nodes. Note that the principal reason for maintaining fully reduced MTBDDs in normal use is to preserve the canonicity property of the data structure. As described in Section 3.7.3, this allows extremely efficient manipulation of MTBDDs by facilitating features such as caching of intermediate results in the computed table. Offset-labelled MTBDDs are not canonical but do not need to be. The data structure will be constructed once, used to perform numerical computation via traversal and then discarded: no manipulation is required.

In Figure 6.4, we give the algorithm, TraverseOffsets, which is used to traverse the new, offset-labelled MTBDD data structure and extract the matrix entries. Compare the new algorithm in Figure 6.4 with the old version in Figure 6.1. The key differences are as follows. Firstly, we do not need to check for skipped levels. This removes a significant amount of effort. The exception to this is that we must still check for edges which skip directly to the zero terminal. Secondly, in the recursive portion of the algorithm, instead of adding 2^{n-i} to the indices r and c, we use the node offsets, denoted $\mathit{off}(\mathsf{m})$ for node m.

Figure 6.5 shows an example of the new data structure representing the same matrix $\mathbf{M}$ as in the previous section. Compare this to the MTBDD in Figure 6.2. Note the addition of the offset labels and also the insertion of an extra node on the path to the 3 ter-

```
TraverseOffsets(m, i, r, c)
1.  if (m = Const(0)) then
2.      return
3.  else if (i = n + 1) then
4.      UseMatrixEntry(r, c, val(m))
5.      return
6.  endif
7.  e := else(m)
8.  t := then(m)
9.  if (e ≠ Const(0)) then
10.     TraverseOffsets(else(e), i + 1, r, c)
11.     TraverseOffsets(then(e), i + 1, r, c + off(e))
12. endif
13. if (t ≠ Const(0)) then
14.     TraverseOffsets(else(t), i + 1, r + off(m), c)
15.     TraverseOffsets(then(t), i + 1, r + off(m), c + off(t))
16. endif
```

Figure 6.4: The refined traversal algorithm TraverseOffsets

minal. Figure 6.6 explains the traversal of the data structure by the TraverseOffsets algorithm. Again, each row corresponds to a single matrix entry. The table gives both the path through the MTBDD corresponding to this entry and the offsets which have been summed to determine its row and column indices. The table in Figure 6.6 can be compared to the equivalent one for the TraverseMtbdd algorithm in Figure 6.3. Since only 3 of the 4 states are reachable, all references to state 3 in the previous table are replaced with 2 in the new one.

The offset-labelled MTBDD and the traversal algorithm TraverseOffsets can now be used to efficiently perform matrix-vector multiplication where the matrix is stored as an MTBDD and the vector as an array. This allows us to implement iterative numerical methods, as required for probabilistic model checking, using these two data structures. First though, we need to consider how this new data structure can be generated.

6.2 The Construction Process

6.2.1 Generating the Offsets

In the next two sections, we describe the process of constructing an offset-labelled MTBDD. The first issue we resolve is how to generate the offsets which will label the nodes. These

$$\mathbf{M} = \begin{pmatrix} 0 & 3 & - & 6 \\ 0 & 3 & - & 4 \\ - & - & - & - \\ 5 & 0 & - & 4 \end{pmatrix}$$

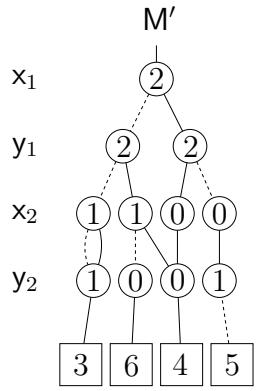
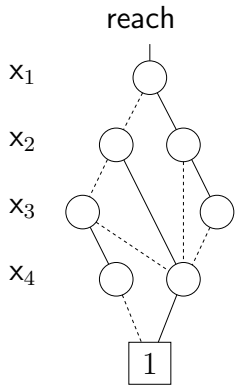


Figure 6.5: A matrix $\mathbf{M}$ and the offset-labelled MTBDD M' which represents it

Path					Offsets				Entry of $\mathbf{M}$
x_1	y_1	x_2	y_2	$f_{\mathsf{M}'}$	x_1	y_1	x_2	y_2	
0	0	0	1	3	-	-	-	1	$(0,1) = 3$
0	0	1	1	3	-	-	1	1	$(1,1) = 3$
0	1	0	1	6	-	2	-	0	$(0,2) = 6$
0	1	1	1	4	-	2	1	0	$(1,2) = 4$
1	0	1	0	5	2	-	0	-	$(2,0) = 5$
1	1	1	1	4	2	2	0	0	$(2,2) = 4$

Figure 6.6: Illustration of the TraverseOffsets algorithm on M'



#	$(\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4)$
0	$(0, 0, 0, 1)$
1	$(0, 0, 1, 0)$
2	$(0, 1, 0, 1)$
3	$(0, 1, 1, 1)$
4	$(1, 0, 0, 1)$
5	$(1, 0, 1, 1)$
6	$(1, 1, 0, 1)$

Figure 6.7: An example of the BDD reach and the states it represents

are used to determine the row or column index, in terms of reachable states, of each entry in the matrix being represented. As we have seen, for a node m where $var(\mathsf{m})$ is a row variable, the offset gives the number of rows in the submatrix represented by $else(\mathsf{m})$ which correspond to reachable states. If $var(\mathsf{m})$ is a column variable, the offset gives the number of columns which correspond to reachable states.

We assume that the necessary information about which of the potential 2^n states are reachable is stored in a BDD reach. Figure 6.7 shows an example: a BDD over four variables $(\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4)$ which encodes a set of 7 states taken from $\mathbb{B}^4$. The table in the figure lists the reachable states in order, assigning them indices from 0 to 6.

The states which are reachable in a certain portion of the state space are represented by the appropriate subgraphs of reach. Using the example from Figure 6.7, the set of states for which $\mathsf{x}_1 = 0$, i.e. those of the form $(0, \cdot, \cdot, \cdot)$, is represented by the BDD $else(\mathsf{reach})$. This information can be used to determine the rows and columns of a particular submatrix which correspond to reachable states: the number of such rows or columns is equal to the number of minterms in the appropriate subgraph of reach. Hence, we will use reach to compute the required offsets. For convenience, we will store these values on the nodes of the BDD reach itself.

The first step is to modify reach slightly by removing all skipped levels, except those which go directly to the zero terminal. This ensures that there are nodes at each level of every path through the BDD on which to store offsets. We do this by checking for edges which skip one or more levels and inserting extra nodes. Empirical results show that this usually results in only a very slight increase in the size of the BDD. The process is performed by the recursive algorithm ReplaceSkippedLevels, given in Figure 6.8. The top-level call is:

$$\mathsf{reach}' := \textsc{ReplaceSkippedLevels}(\mathsf{reach}, 1)$$

```
ReplaceSkippedLevels(m, i)
1.  if (marked(m) = true) then
2.      return m
3.  else if (m = Const(0)) then
4.      marked(m) := true
5.      return m
6.  else if (i = n + 1) then
7.      marked(m) := true
8.      return m
9.  else if (var(m) > x_i) then
10.     e := t := ReplaceSkippedLevels(m, i + 1)
11.     M' := CreateNode(x_i, e, t)
12.     marked(m') := true
13.     return m'
14. else
15.     else(m) := ReplaceSkippedLevels(else(m), i + 1)
16.     then(m) := ReplaceSkippedLevels(then(m), i + 1)
17.     marked(m) := true
18.     return m
19. endif
```

Figure 6.8: The ReplaceSkippedLevels algorithm

On each recursive call, the algorithm checks to see if a level is skipped at the current position in the MTBDD, adds an extra node if this is the case, and then recurses downwards. The level of recursion, i, is tracked and used to check for skipped levels in the same way as in the TraverseMtbdd algorithm. The function CreateNode($\mathsf{x}_i, \mathsf{e}, \mathsf{t}$) returns a non-terminal MTBDD node m with $var(\mathsf{m}) = \mathsf{x}_i$, $else(\mathsf{m}) = \mathsf{e}$ and $then(\mathsf{m}) = \mathsf{t}$. It first checks to see if an identical node has already been created and, if so, reuses it. Otherwise, a new node is created. For the purposes of this algorithm, we have added an extra field, *marked*, to each BDD node. This stores a Boolean variable, used to mark nodes which have already been dealt with by the algorithm. Using this approach, we can ensure that the algorithm only visits each node once, rather than many times, as would usually be the case in a recursive traversal of the BDD.

Next, we proceed by labelling each node m in the modified BDD reach' with the number of minterms in its child $else(\mathsf{m})$. This value can be computed recursively by adding the number of minterms of its two children and is done by the recursive algorithm ComputeOffsets(m, i), given in Figure 6.9. As in previous algorithms, i denotes the level of recursion. The top-level call would be ComputeOffsets($\mathsf{reach}', 1$).

We add two fields, off_e and off_t, to each node of the BDD to store the number of

```
ComputeOffsets(m, i)
 1.  if (m = Const(0)) then
 2.      return 0
 3.  else if (i = n + 1) then
 4.      return 1
 5.  else if (off_e(m) ≠ −1) then
 6.      return off_e(m) + off_t(m)
 7.  else
 8.      off_e(m) := ComputeOffsets(else(m), i + 1)
 9.      off_t(m) := ComputeOffsets(then(m), i + 1)
10.      return off_e(m) + off_t(m)
11.  endif
```

Figure 6.9: The ComputeOffsets algorithm

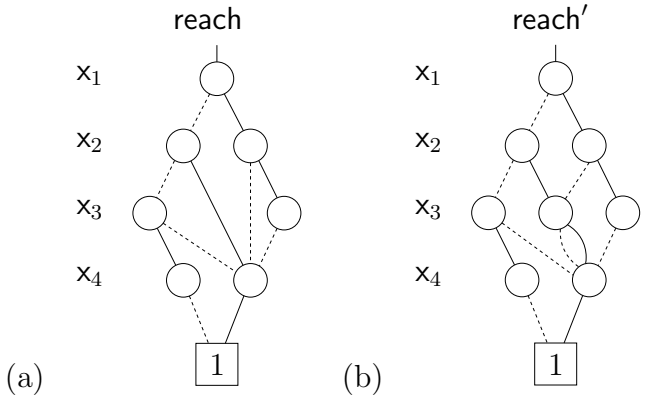
minterms in its *else* and *then* branches respectively. Initially, these are both set to -1 on all nodes, allowing us to keep track of which nodes have been visited dealt with. A call to ComputeOffsets$(\mathsf{m}, \cdot)$ computes values for off_e and off_t, stores them on m, and then returns the sum of the two, i.e. the number of minterms for m. When the algorithm terminates, we remove the two labels from each node and replace them with a single label off, which is given the value of off_e.

This process could be carried out by storing just off_e, not off_t. This way, however, we do not need to calculate the value of off_t each time we revisit a node. Hence, the time complexity is proportional to the number of nodes of the BDD rather than the number of minterms or paths.

Figure 6.10 gives a simple illustration of the process of generating these offsets: (a) shows the BDD reach from the earlier example in Figure 6.7; (b) shows the modified BDD reach' created by applying ReplaceSkippedLevels to reach; in (c), the ComputeOffsets algorithm has labelled each node with the two offsets off_e and off_t; in (d), the offset off_t has been removed from each node. Figure 6.10(e) shows a table explaining how the indices of each state can now be read off the final offset-labelled BDD reach'. Each row of the table corresponds to path through the BDD which computes the index for a single state. This index can be obtained by summing the offsets on nodes along the path from which the *then* edge was taken.

6.2.2 Constructing the Offset-Labelled MTBDD

We now discuss how an offset-labelled MTBDD is built. Assume that we have a matrix, represented by an MTBDD M, and an indexing of reachable states, represented by an



(e)

Path				Offsets				Index
x_1	x_2	x_3	x_4	x_1	x_2	x_3	x_4	
0	0	0	1	-	-	-	0	0
0	0	1	0	-	-	1	-	1
0	1	0	1	-	2	-	0	2
0	1	1	1	-	2	1	0	3
1	0	0	1	4	-	-	0	4
1	0	1	1	4	-	1	0	5
1	1	0	1	4	2	-	0	6

Figure 6.10: Labelling the BDD reach with offsets

offset-labelled BDD reach'. Essentially, we want to combine the two into a single data structure which, when traversed by the algorithm TraverseOffsets from Figure 6.4, will produce the correctly indexed matrix entries, as defined by M and reach'.

The construction process works by modifying the existing MTBDD M. As when modifying reach in the previous section, we first remove from M all occurrences of skipped levels, except those that go directly to the zero terminal. Again, the motivation for this is to ensure that offsets can be stored at every level of the MTBDD. For this, we reuse the ReplaceSkippedLevels algorithm of Figure 6.8:

$$\mathsf{M}' := \textsc{ReplaceSkippedLevels}(\mathsf{M}, 1)$$

using variables $(\mathsf{x}_1, \mathsf{y}_1, \ldots, \mathsf{x}_n, \mathsf{y}_n)$, instead of just $(\mathsf{x}_1, \ldots, \mathsf{x}_n)$.

The remainder of the construction process entails labelling each node of M' with the correct offset. This is done via a recursive traversal of M'. At the same time, two concurrent traversals of the offset-labelled BDD reach' are performed. These are used to determine what the offset for each node should be, one for nodes labelled with row variables and one for those labelled with column variables.

When we encounter a node of M' for the first time, this process works fine. However, when nodes are revisited, a problem quickly becomes apparent: there is a potential clash in terms of offset labelling when two different paths pass through the same node. This situation will occur when the entries in two submatrices are identical, but the patterns of reachable states in the rows or columns are different. Our solution will simply be to add into the MTBDD an identical copy of the node, but labelled with the correct offset. This is why, in our earlier definition of offset-labelled MTBDDs, we did not insist that all shared nodes must be merged.

The algorithm to perform the whole process is given in Figure 6.11. It is recursive, but since row and column nodes are treated slightly differently, it alternates between two functions, LabelMtbddRow and LabelMtbddCol. Both take three parameters, m, row and col, which are the current nodes in the traversal of M' and in the two traversals of reach', respectively. The top-level call is:

$$\mathsf{M}'' := \textsc{LabelMtbddRow}(\mathsf{M}', \mathsf{reach}', \mathsf{reach}')$$

We assume that each node of the MTBDD M' being labelled has two extra labels *row* and *col*. These will store pointers to BDD nodes and are initially all set to null. Although we only intend to label each node m of M' with a single integer offset, for the duration of the algorithm, what we actually store on m is two pointers to nodes of the BDD reach'. Since the latter are themselves labelled with the required offset, this information is easily accessed indirectly. More specifically, in a recursive call to either $\textsc{LabelMtbddRow}(\mathsf{m}, \mathsf{row}, \mathsf{col})$

```
LabelMtbddRow(m, row, col)
 1.  if (m is a terminal node) then
 2.      return m
 3.  else if (row(m) = row and col(m) = col) then
 4.      return m
 5.  endif
 6.  e := LabelMtbddCol(else(m), else(row), col)
 7.  t := LabelMtbddCol(then(m), then(row), col)
 8.  if (row(m) = null) then
 9.      else(m) := e
10.      then(m) := t
11.      row(m) := row
12.      col(m) := col
13.      return m
14.  else
15.      return CreateNode(var(m), e, t, row, col)
16.  endif
```

```
LabelMtbddCol(m, row, col)
 1.  if (m is a terminal node) then
 2.      return m
 3.  else if (row(m) = row and col(m) = col) then
 4.      return m
 5.  endif
 6.  e := LabelMtbddRow(else(m), row, else(col))
 7.  t := LabelMtbddRow(then(m), row, then(col))
 8.  if (col(m) = null) then
 9.      else(m) := e
10.      then(m) := t
11.      row(m) := row
12.      col(m) := col
13.      return m
14.  else
15.      return CreateNode(var(m), e, t, row, col)
16.  endif
```

Figure 6.11: The LabelMtbddRow and LabelMtbddCol algorithms

or LabelMtbddCol(m, row, col), the labels $row(\mathsf{m})$ and $col(\mathsf{m})$ will be set to row and col, respectively. The actual offset which will eventually label the node is $\mathit{off}(row(\mathsf{m}))$ if $var(\mathsf{m})$ is a row variable and $\mathit{off}(col(\mathsf{m}))$ if it is a column variable.

The reasoning behind this approach is as follows. As stated above, there are potential clashes in terms of the offset which should label an MTBDD node. If we labelled each MTBDD node with the actual offset, we would have to traverse every possible path through the MTBDD to ensure that we had dealt with all possible clashes. Our approach, however, is more efficient. On visiting a node for the second time, we check that the nodes $row(\mathsf{m})$ and $col(\mathsf{m})$ are equal to row and col. If this is the case, we know not only that the (implicitly stored) offset is correct, but that the same is true for all nodes below it in the MTBDD. This speeds the whole process up considerably.

After the algorithm terminates, we replace the two *row* and *col* fields on each node m with a single integer field *off*, which contains the final offset. If $var(\mathsf{m})$ is a row variable, then this value is set to $\mathit{off}(row(\mathsf{m}))$ and if $var(\mathsf{m})$ is a col variable, then it is set to $\mathit{off}(col(\mathsf{m}))$.

Figure 6.12 gives an example of an offset-labelled MTBDD constructed in this way: (a) shows a 4×4 matrix and (b) the MTBDD M which represents it. Note that this example again includes an unreachable state. Figures 6.12(c) and (d) show the offset-labelled BDD reach' and the offset-labelled MTBDD M'', respectively. Firstly, note the skipped level on the left of the x_2 level in the original MTBDD and the extra node introduced in M'' to resolve it. Secondly, note the two extra nodes added at the bottom right as a result of offset clashes. For clarity, we also give, in Figure 6.12(e), a table explaining how the TraverseOffsets algorithm would operate on this particular offset-labelled MTBDD M''. It can be verified that each matrix entry obtained by tracing a single path through M'' can alternatively be determined by tracing one path through M and two paths through reach'.

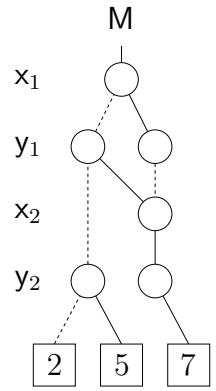
Correctness

The correctness of the construction algorithm can be argued as follows. We require that the offset-labelled MTBDD produced by the algorithm is such that, when traversed by the algorithm TraverseOffsets, all non-zero entries of the matrix and their corresponding (reachable) row and column indices are extracted. For this, we require that every path through the MTBDD from the root node to a non-zero terminal corresponds to an entry $(r, c) = v$, i.e. the terminal in question is labelled with v and r and c can be determined by summing the offsets on nodes labelled with row and column variables, respectively, from which *then* edges were taken.

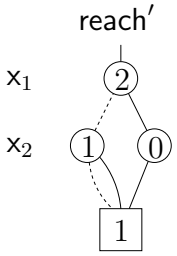
(a)

$$\mathbf{M} = \begin{pmatrix} 2 & 5 & - & 0 \\ 2 & 5 & - & 7 \\ - & - & - & - \\ 0 & 7 & - & 0 \end{pmatrix}$$

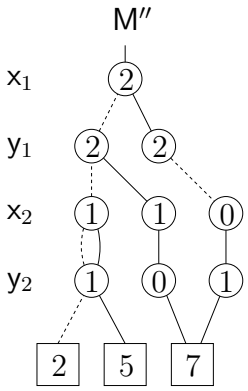
(b)



(c)



(d)



(e)

Path					Offsets				Entry of $\mathbf{M}$
x_1	y_1	x_2	y_2	$f_{M''}$	x_1	y_1	x_2	y_2	
0	0	0	0	2	-	-	-	-	$(0,0) = 2$
0	0	0	1	5	-	-	-	1	$(0,1) = 5$
0	0	1	0	2	-	-	1	-	$(1,0) = 2$
0	0	1	1	5	-	-	1	1	$(1,1) = 5$
0	1	1	1	7	-	2	1	0	$(1,2) = 7$
1	0	1	1	7	2	-	0	1	$(2,1) = 7$

Figure 6.12: Construction of an offset-labelled MTBDD M″

The set of non-zero entries is defined by the original MTBDD M and its corresponding function $f_{\mathsf{M}} : \mathbb{B}^{2n} \to \mathbb{R}$. The index, be it row or column, of each state (in terms of reachable states only) has already been shown to be defined by the offset-labelled BDD reach' which maps each path to the 1 terminal to an integer by summing offsets. The combined data structure must capture both sets of information. We demonstrate each of the two cases separately.

Firstly, we show, by induction, that the offset-labelled MTBDD preserves the original function $f_{\mathsf{M}} : \mathbb{B}^{2n} \to \mathbb{R}$. For the base case, we observe that terminals, at the bottom of the recursion, are returned unchanged. For non-terminal nodes, if the node is being visited for the first time or has been visited already and the *row* and *col* labelling is the same, it is also returned unchanged, except for the possible addition of offset information which is irrelevant here. The only instance where the structure of the MTBDD is actually changed is when a new node is created because of an offset clash. In this case, the *then* and *else* edges of the new node are attached to nodes computed by recursion on the *then* and *else* edges of the original node. By induction, these are correct with respect to f_{M}.

Secondly, we show that the MTBDD returned has the correct offsets. We observe that, by the recursive nature of the algorithm, it considers every path from the root to a non-zero terminal. If it reaches the bottom of a path, i.e. a non-zero terminal, it will have taken the corresponding choices in row and col and (indirectly) labelled each node with the correct offset. Where necessary, new nodes may have been added to ensure that this is the case. The only situation where some portion of a path is not considered is when the current node m has already been visited and $\mathit{row}(\mathsf{m})$ and $\mathit{col}(\mathsf{m})$ are the same as the values of row and col at the current point in the recursion. Since the traversal of M, row and col from then on will be exactly the same as last time the node was visited, the offsets on all nodes below this point must be correct.

Efficiency

The time complexity of the construction algorithm is determined by the size of the MTBDD produced by it. If no new nodes need to be added, then each of the nodes in the original MTBDD is only dealt with once. If this is not the case, additional work is required to create each extra node. Extra nodes are either a result of skipped levels or offset clashes. Unfortunately, it is very difficult to quantify how often either of these will occur in a typical MTBDD. The hope is that, in a structured MTBDD, most identical subtrees in the MTBDD will correspond to identical sets of reachable states and few nodes will need to be added. We will see the empirical results in the next section.

6.3 Results

We have integrated the hybrid approach described in first part of this chapter into our probabilistic model checker PRISM. We have used the techniques to implement model checking for the PCTL bounded until, PCTL until, CSL time-bounded until and CSL steady-state operators. Where solution of a linear equation system is required, we can use either the Jacobi or JOR methods.

This section presents some results to illustrate the effectiveness, in terms of both time and space requirements, of our approach. Our analysis is divided into two parts: first, we consider the construction of the offset-labelled MTBDD and second, the implementation of iterative numerical methods, based on repeated traversal of the data structure using the TraverseOffsets algorithm. We also compare the performance of the hybrid approach with that of the two rival alternatives, namely MTBDDs and sparse matrices.

6.3.1 The Construction Process

As discussed previously, the strength of the algorithm for constructing the offset-labelled MTBDD is the fact that its time complexity is proportional to the number of nodes created, not the number of paths in the original MTBDD. For this reason, the time for construction should generally be less than a single iteration of the solution process. Given that many such iterations will be required, our main concern is that the amount of memory required to store the matrix does not increase significantly.

In Table 6.1, we present statistics for the construction process. In this and the following sections, we show results for four model checking case studies: the Kanban manufacturing system of [CT96] and cyclic server polling system of [IT90], CTMC models for which we compute steady-state probabilities using the JOR ($\omega = 0.9$) and Jacobi methods, respectively; the bounded retransmission protocol (BRP) of [HSV94], a DTMC model for which we model check a PCTL until formula using the Jacobi method; and the flexible manufacturing system (FMS) of [CT93], a CTMC model for which we check a CSL time-bounded until property. The sizes of all four models can be varied by changing a parameter N, the meaning of which for each case can be found in Appendix E.

For each model, Table 6.1 gives the number of nodes in the original MTBDD ("Before"), the number of nodes in the newly constructed, offset-labelled MTBDD ("After"), the percentage increase, and the time taken for the construction process.

We conclude that the increase in the number of nodes is typically small. For the three CTMC examples, we always observe a growth of less than 5%. The results for the DTMC example, the bounded retransmission protocol (BRP), are not quite as impressive, but

Model	N	MTBDD Nodes		Increase	Time
		Before	After	(%)	(sec.)
Kanban system	1	499	499	0	< 0.01
	2	1,685	1,685	0	< 0.01
	3	2,474	2,474	0	< 0.01
	4	4,900	4,900	0	0.01
	5	6,308	6,308	0	0.02
	6	7,876	7,876	0	0.04
Polling system	8	608	629	3.45	< 0.01
	10	921	948	2.93	< 0.01
	12	1,282	1,315	2.57	0.01
	14	1,707	1,746	2.28	0.01
	16	2,188	2,233	2.06	0.01
	18	2,745	2,796	1.86	0.01
BRP	500	2,268	3,032	33.7	0.02
	1,000	2,340	3,126	33.6	0.02
	1,500	2,410	3,242	34.5	0.03
	2,000	2,412	3,220	33.5	0.03
	2,500	2,486	3,320	33.5	0.03
FMS	3	10,423	10,880	4.38	0.06
	4	26,744	27,941	4.48	0.36
	5	47,750	49,877	4.45	1.17
	6	73,256	76,638	4.62	2.89
	7	113,902	118,978	4.46	6.92
	8	203,495	211,563	3.96	22.6

Table 6.1: Statistics for the offset-labelled MTBDD construction process

still satisfactory: on average, the data structure increases in size by about a third. As we will see in the next section, the amount of memory required to store this data structure is still significantly smaller than that required by the equivalent sparse matrix. We also note that the total time required for construction in small in all cases.

We have analysed further the reasons for the increase in MTBDD size. As described previously, nodes will be added either as a result of skipped levels in the original MTBDD or because of offset clashes. We determined that, in every example above, all the additional nodes were a result of offset clashes. It seems that the compactness of the MTBDDs for these models is purely a result of shared nodes, not skipped levels.

6.3.2 The Solution Process

Secondly, and more importantly, we consider the performance of the numerical solution phase. Although the exact computation carried out varies depending on the instance of probabilistic model checking being performed, this phase always comprises a number of iterations, the key element of each being a single traversal of an offset-labelled MTBDD using the TraverseOffsets algorithm. Table 6.2 shows timing statistics for the four model checking case studies described in the previous section. For each example, it lists the size of the model (number of states), the number of iterations performed and the average time per iteration for each of our three implementations. Here, as always, we give actual (wall) times measured on a 440 MHz 512 MB Sun Ultra10. A '-' symbol indicates that an example could not be model checked due to memory limitations. For the Kanban system, polling system and FMS case studies, we increased the parameter N as far as possible for each implementation. For the BRP examples, this was not the case since the computation for some of the larger models we built suffered from round-off errors.

The results can be summarised as follows. The new hybrid approach represents a marked improvement over the original MTBDD implementation. There is an impressive reduction in average iteration time for all examples. In the best cases (Kanban system for $N = 3$ and polling system for $N = 10$), it is improved by a factor of more than 100. Unfortunately, the times remain much slower than for sparse matrices.

The most noteworthy result, though, is that our hybrid approach can handle larger models than both of the other implementations. This is a result of their relative memory requirements. As in the previous chapter, we do not attempt a detailed comparison with the memory requirements for MTBDDs since accurate information is very difficult to obtain. We can see from Table 6.2, however, that many of the examples cannot be performed using the MTBDD implementation due to insufficient memory. A comparison of the memory required for hybrid and sparse matrix approaches is given in Table 6.3.

Model	N	States	Iterations	Time per iteration (sec.)		
				MTBDD	Sparse	Hybrid
Kanban system	1	160	101	0.03	0.001	0.001
	2	4,600	166	2.22	0.002	0.03
	3	58,400	300	45.5	0.05	0.35
	4	454,475	466	-	0.41	4.79
	5	2,546,432	663	-	2.70	26.1
	6	11,261,376	891	-	-	123.8
Polling system	8	3,072	310	0.31	0.001	0.004
	10	15,360	406	13.1	0.01	0.03
	12	73,728	505	-	0.05	0.16
	14	344,064	606	-	0.30	0.89
	16	1,572,864	709	-	1.56	4.46
	18	7,077,888	814	-	-	23.1
BRP	500	27,505	3,086	0.38	0.01	0.12
	1,000	55,005	6,136	0.67	0.01	0.24
	1,500	82,505	9,184	0.99	0.02	0.36
	2,000	110,005	12,230	1.26	0.03	0.47
	2,500	137,505	15,274	1.53	0.04	0.59
FMS	3	6,520	569	6.13	0.002	0.11
	4	35,910	733	48.3	0.02	0.84
	5	152,712	825	-	0.10	3.35
	6	537,768	968	-	0.37	13.3
	7	1,639,440	1,040	-	1.14	33.6
	8	4,459,455	1,182	-	-	102.0

Table 6.2: Timing statistics for numerical solution using the hybrid approach

Model	N	Vector storage (KB)	Matrix storage (KB)		Total storage (KB)	
			Sparse	Hybrid	Sparse	Hybrid
Kanban system	1	3 × 1	8	10	11	13
	2	3 × 36	348	33	455	141
	3	3 × 456	5,455	48	6,823	1,417
	4	3 × 3,551	48,414	96	59,066	10,748
	5	3 × 19,894	296,588	123	356,270	59,805
	6	3 × 87,979	-	154	-	264,092
Polling system	8	3 × 24	186	12	258	84
	10	3 × 120	1,110	19	1,470	379
	12	3 × 576	6,192	26	7,920	1,754
	14	3 × 2,688	32,928	34	40,992	8,098
	16	3 × 12,288	168,960	44	205,824	36,908
	18	3 × 55,296	-	55	-	165,943
BRP	500	4 × 215	441	59	1,301	919
	1,000	4 × 430	883	61	2,602	1,781
	1,500	4 × 645	1,324	63	3,902	2,643
	2,000	4 × 859	1,766	63	5,203	3,499
	2,500	4 × 1,074	2,207	65	6,504	4,361
FMS	3	4 × 51	451	213	655	416
	4	4 × 281	2,711	546	3,835	1,668
	5	4 × 1,193	11,990	974	16,762	5,746
	6	4 × 4,201	42,744	1,497	59,548	18,302
	7	4 × 12,808	129,853	2,324	181,085	53,556
	8	4 × 34,839	-	4,132	-	143,490

Table 6.3: Memory requirements for numerical solution using the hybrid approach

We present statistics for the same four case studies: the amount of memory to store the matrix; the amount to store the iteration vectors; and the total memory, which is the sum of these. All values are rounded to the nearest kilobyte.

For the hybrid implementation, the memory required to store the matrix is determined by the number of nodes in the offset-labelled MTBDD. Each node requires 20 bytes, as was the case for a standard MTBDD. Although we must store an additional integer (the offset) on each node, we can reuse the space previously occupied by the node's reference count (see Section 3.7.3). This is not required because we never need to manipulate this new data structure: we simply create it once, traverse it repeatedly and then discard it. The memory required to store the matrix in sparse format is computed as described in Section 3.6.

In both cases, we assume that all vectors are stored as arrays of doubles (8 bytes per

entry). The number of arrays required varies depending on the model checking problem. All the iterative methods require at least two, to store the previous and current approximations. We also store the diagonal entries of the matrix in an array because this gives a significant improvement in speed. Additionally, for CSL time-bounded until we require one to keep track of the weighted sum of vectors, and for PCTL until over DTMCs we need one to store the vector $\underline{b}$ of the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$. For CSL steady-state, this is not necessary since $\underline{b}$ is the zero vector. The actual number of arrays needed for each example is given in the table.

From Table 6.3, the main conclusion we draw is that, despite the potential increase in MTBDD size and storage of additional information, the amount of memory required to store the matrix using the hybrid approach is still far less than with sparse matrices. At best, the ratio is four orders of magnitude. In fact, the limiting factor, in terms of memory, now becomes the storage of the iteration vectors.

On the 512 MB Sun Ultra10 used for these experiments, the largest probabilistic model successfully analysed with the hybrid technique had over 11 million states; with sparse matrices, the largest had about 2.5 million states. The choice of machine for our experiments was dictated by the fact that we needed a single-user workstation in order to generate accurate timings. To test the scalability of our approach, we also ran experiments from the same case studies on a shared machine with 1 GB of RAM. On this, the largest model handled by the hybrid approach had 41 million states, compared to 7 million for the sparse matrix implementation. In general, we find that we can increase the size of solvable model by approximately a single order of magnitude.

6.4 Optimising the Hybrid Approach

6.4.1 The Basic Idea

We now present an improvement to our hybrid approach which will result in a significant speedup for numerical solution. Every iteration of the process requires access to each matrix entry exactly once. In the hybrid approach described in this chapter, this is achieved by a recursive traversal of an offset-labelled MTBDD, tracing every path from the root node to a non-zero terminal. It is hardly surprising that the sparse matrix implementation, in which the matrix entries can be read directly off the arrays storing them, is faster.

It is important to note that, despite our best efforts in the preceding chapters to minimise the size of the MTBDD representing the matrix, for the hybrid approach the number of nodes affects only the storage space required and has no bearing on traversal

speed. The compactness of the MTBDD is a result of shared nodes in the data structure, representing structure in the high-level model. In terms of the matrix, these shared nodes correspond to identical submatrices. However, since the actual row and column indices of the entries in these submatrices are different for each occurrence, the entries are extracted separately each time. In fact, each shared node is visited once for every time it occurs on a path to a non-zero terminal in the MTBDD.

In this section, we modify the offset-labelled MTBDD data structure and its traversal algorithm in order to exploit the data structure's compactness in terms of time as well as space. The key idea is that, although the actual row and column indices of entries in each repeated submatrix are different each time it occurs in the overall matrix, the local indices, relative to that submatrix are the same. This is of course fundamental to the working of the recursive traversal algorithm which computes the indices.

The optimisation we propose is, for some subset of the nodes in the MTBDD, to cache an explicit version of the submatrix corresponding to each node and then reuse it each time the node is visited in future, rather than having to traverse all the other nodes below that one. To compute the actual indices of the submatrix on each visit, we can simply add the computed offsets (r and c in TraverseOffsets) to the local indices. Since the traversals in each iteration are identical, we will compute and store these matrices once, at the beginning of the solution process, and then use them on every iteration.

6.4.2 Implementing the Optimisations

The first task is to decide for which nodes we will store an explicit matrix. There is a clear compromise here between time and space. If we opt to compute the matrices for nodes near the top of the MTBDD, we are likely to significantly improve the time to extract its entries, but would lose our advantage in terms of memory usage. Conversely, if we only stored the matrices for a few nodes at low levels in the MTBDD, we would require little extra memory, but would also achieve only a small improvement in terms of traversal time.

One situation we want to avoid is duplication of information. For example, if we store the matrix for two nodes which occur on a common path, the submatrix for the lower node is effectively being stored twice. With this in mind, we propose to adopt the following simple scheme. We will select a particular level of the MTBDD and then compute matrices for all the nodes on that level. For simplicity, we will limit our choice of levels to those containing row variables. This is because our existing traversal algorithm is based on a four way recursive split, so at each level the current node is labelled with a row variable. One advantage of this scheme is that it is guaranteed not to duplicate

information. Secondly, it is easy to control the amount of optimisation, and hence the compromise between time and space, by selecting a level higher or lower in the MTBDD.

We also need to consider *how* the matrices will be stored. Since all but the very smallest submatrices of a large, sparse matrix will also be sparse, it seems sensible to adopt the same sparse storage scheme we have been using to date. The next question is *where* they will be stored. One approach would be to store the matrices in a cache and then check the cache each time we come across a node to see if its matrix has been precomputed. However, since this would require a great deal of searching at every step of every iteration, and since the number of nodes in our MTBDD is relatively small, we instead add a pointer to each MTBDD node. For nodes with precomputed matrices, we store a pointer to the matrix. For nodes without, we leave the pointer null.

Once these decisions have been made, the optimised version of the hybrid approach can be implemented fairly easily using our existing algorithms. First, we add an extra field, *sm*, to each MTBDD node which can hold a pointer to a sparse matrix. Initially, these pointers are all set to null. For a specified level of the MTBDD, we then compute the sparse matrices corresponding to each node on that level. These matrices can be obtained using the existing TraverseOffsets algorithm, where UseMatrixEntry stores each entry in a sparse matrix, rather than using it to perform a matrix-vector multiplication as was the case previously. Each matrix is then attached to its corresponding node via the pointer *sm*.

The new traversal algorithm, TraverseOpt is shown in Figure 6.13. It is basically an extension of TraverseOffsets, with lines 3–12 added. At each point in the recursion, the algorithm checks the pointer $sm(\mathsf{m})$ of the current node m to see if it is non-null. If so, the sparse matrix is extracted and no further recursion is required for that node. If the pointer is null, traversal continues as before.

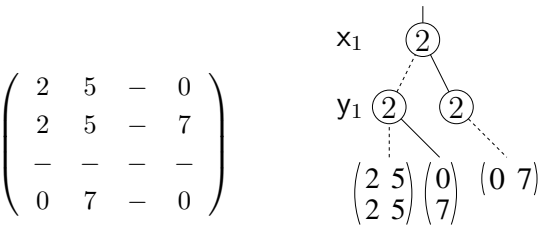
The extraction of matrix entries from the sparse matrix $sm(\mathsf{m})$ is done in exactly the same way that we saw in the sparse matrix-vector multiplication algorithm of Figure 3.13. The sparse matrix is stored in three arrays, *row*, *col*, and *val*, and *dim* gives the size of the matrix. The main difference in the extraction process is that the current offsets r and c are added to the row and column indices to compute the correct values.

6.4.3 Example

Figure 6.14 illustrates the idea of this optimisation through an example, using the matrix and MTBDD from Figure 6.12. We compute the submatrices for all nodes on the x_2 level. In the diagram, we denote this by replacing each node and any nodes under it by the matrix itself. Since it is such a small example, none of the three nodes which have been

```
TraverseOpt(m, i, r, c)
 1.  if (m = Const(0)) then
 2.      return
 3.  else if (sm(m) ≠ null) then
 4.      (row, col, val, dim) := sm(m)
 5.      for (j := 0 ... dim−1)
 6.          l := row[j]
 7.          h := row[j + 1]−1
 8.          for (k := l ... h)
 9.              UseMatrixEntry(r + j, c + col[k], val[k])
10.          endfor
11.      endfor
12.      return
13.  else if (i = n + 1) then
14.      UseMatrixEntry(r, c, val(m))
15.      return
16.  endif
17.  e := else(m)
18.  t := then(m)
19.  if (e ≠ Const(0)) then
20.      TraverseOpt(else(e), i + 1, r, c)
21.      TraverseOpt(then(e), i + 1, r, c + off(e))
22.  endif
23.  if (t ≠ Const(0)) then
24.      TraverseOpt(else(t), i + 1, r + off(m), c)
25.      TraverseOpt(then(t), i + 1, r + off(m), c + off(t))
26.  endif
```

Figure 6.13: The optimised traversal algorithm TraverseOpt



Path		Offsets		Local	Global
x_1	y_1	x_1	y_1	entry	entry
0	0	-	-	$(0,0) = 2$	$(0,0) = 2$
				$(0,1) = 5$	$(0,1) = 5$
				$(1,0) = 2$	$(1,0) = 2$
				$(1,1) = 5$	$(1,1) = 5$
0	1	-	2	$(1,0) = 7$	$(1,2) = 7$
1	0	2	-	$(0,1) = 7$	$(2,1) = 7$

Figure 6.14: An illustration of the optimisation idea on a small example

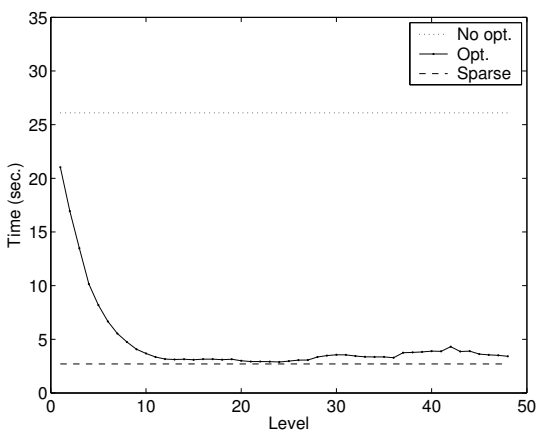
turned into sparse matrices are actually shared nodes. In a more realistic example, each matrix would be reachable along many paths. In addition, the matrices themselves are not particularly sparse. Again, in a larger example, this would not be the case.

The table in Figure 6.14 shows how data structure would be used. Traversal of the MTBDD would result in three paths, each leading to a sparse matrix. The local entries of each submatrix are given in the table, along with the actual entries in the global matrix, which are computed by adding the row or column indices shown.

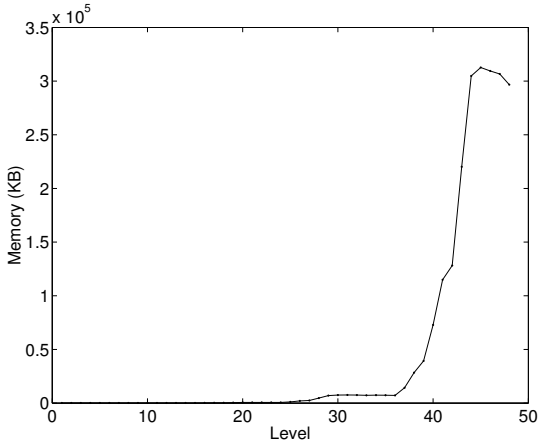
6.4.4 Results

In order to assess the improvement which this optimisation provides, we experimented with replacing the nodes on different levels in the MTBDDs for several case studies. In Figure 6.15, we plot graphs for three such examples, selected from the set for which we presented results in the previous section. In each one, for the full possible range of levels, we graph both the time per iteration of numerical solution and the amount of space required to store the offset-labelled MTBDD and small sparse matrices. For comparison, on the timing graphs, we also plot the time per iteration for the sparse matrix and (unoptimised) hybrid implementations. Note that we only present a graph for one example of each case study. We are interested in the pattern of the graph and this was found to be almost identical for examples taken from the same case study.

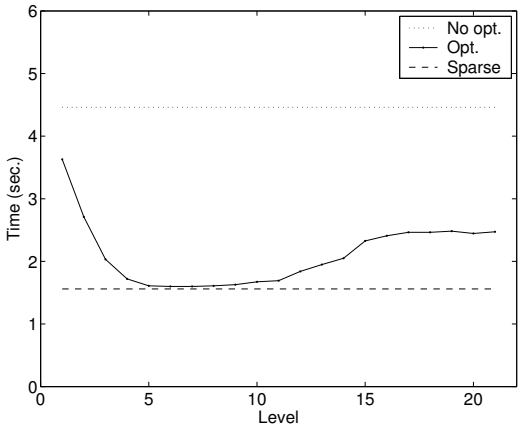
For the purpose of these experiments, we count the number of levels starting from the



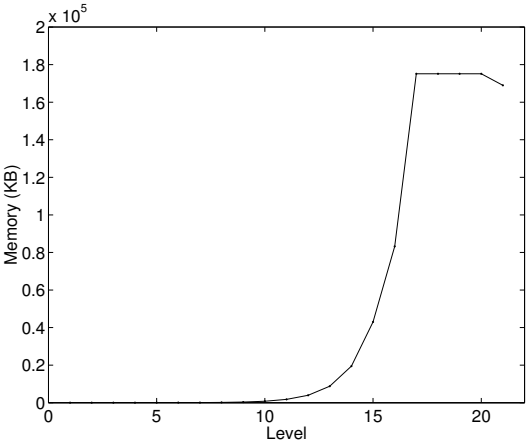
(a) Iteration time (Kanban, $N = 5$)



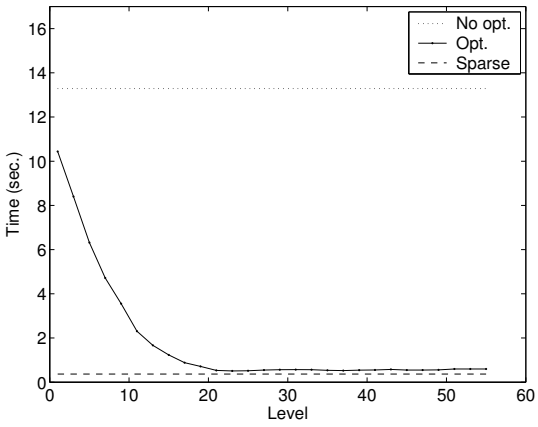
(b) Memory usage (Kanban, $N = 5$)



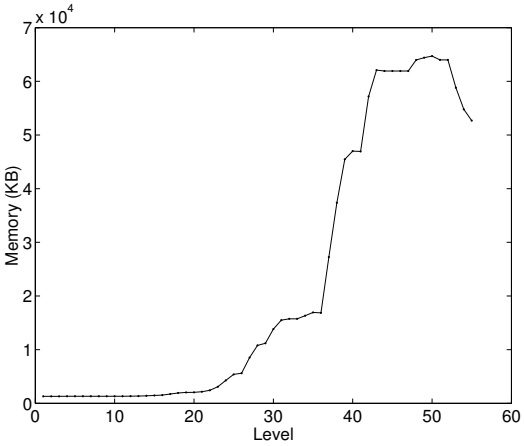
(c) Iteration time (Polling, $N = 16$)



(d) Memory usage (Polling, $N = 16$)



(e) Iteration time (FMS, $N = 6$)



(f) Memory usage (FMS, $N = 6$)

Figure 6.15: Statistics for the optimised hybrid approach

bottom of the MTBDD. Hence, a value of 1 indicates that we have replaced nodes in the bottom level, a value of 2 refers to the second bottom level, and so on. Recall that we only replace nodes labelled with row variables. Hence the ith level is actually the $2i$th from bottom level in the MTBDD. The maximum level we can try is equal to half the height of the MTBDD, or equivalently, the number of Boolean variables used to encode the global state space.

We can obtain several useful pieces of information from the graphs, the first and most obvious of which, is that by optimising our hybrid technique we can dramatically improve the time required for each iteration of numerical solution. We can generally now match the speed of the sparse matrix implementation. Furthermore, this reduction can be made without a significant increase in memory usage.

It is also instructive to examine the graphs in more detail. There is a general pattern which can be observed in all cases. The time per iteration drops very quickly initially, flattens out and then increases slightly at the end. Note that the increase at the end is less pronounced in the Kanban system and FMS examples, since the range of times involved, and hence the scale of the graph, is greater. Realistically, any point in the middle region of the graph gives a satisfactory time.

The steep drop at the beginning of the timing graph can be explained as follows. Moving from left to right along the graph, the level is increased, i.e. we replace nodes further up the MTBDD. By the nature of the depth-first recursive traversal algorithm, the lower nodes will be visited far more often than nodes further up the MTBDD. Hence, the impact of saving work at lower nodes is more pronounced. This also explains why the total improvement is better for the Kanban system and FMS examples which have more levels.

Secondly, we consider the pattern of the graph for memory usage. Here we note that the memory stays low for the first portion of the graph and then increases rapidly towards the right-hand end. This is expected, since the nodes higher up in the MTBDD correspond to larger submatrices, which are likely to contain more non-zeros, and hence take up more space when stored explicitly.

For the right-most point on the graph, the memory usage is the same as when storing the entire matrix explicitly. Note though that this is not the maximum value in each graph. Because of the way the sparse matrix representation works, storing one matrix requires less memory than storing the submatrices which comprise it separately. The top few levels of an MTBDD are likely to contain very few shared nodes. Hence, replacing nodes on these levels is actually more expensive than just replacing the root node.

We hypothesise that the slight increase in iteration time for higher levels is caused by the larger amount of space needed. This theory is supported by a visible correspondence

between the right-hand sections of the graphs for timing and memory usage. In these cases, there is not a great deal of difference between the work done by the hybrid implementation for different levels. The main difference is that the blocks of memory (for each sparse matrix) which have to be accessed are of different sizes. Smaller blocks can be moved in and out of RAM more easily. Larger blocks will be broken into smaller pieces by the operating system anyway, but it is more efficient for us to specify which subblocks we need and when, rather than rely on the operating system's algorithms for doing so. Furthermore, it is likely that using smaller blocks of memory allows us to take more advantage of low-level memory caching.

Fortunately, the amount of memory required to store the sparse matrix corresponding to a given MTBDD node is relatively easy to compute. It depends only on the dimension of the matrix and the number of entries it contains. This information can be computed for each node of the MTBDD in a single recursive traversal of the data structure. This allows us to calculate in advance how much memory would be required to compute and store the matrices for nodes on each level of the MTBDD. Hence, given some upper limit on memory, we can determine the highest level on which nodes can be replaced without exceeding the memory limit.

Finally, we present experimental results to illustrate the time and space requirements of our optimised hybrid approach on the four running examples. We use the scheme described above to select a level for which to store explicit matrices, taking an upper memory limit of 1024 KB. Table 6.4 gives timing statistics. This includes the time taken to construct the new data structure, i.e. the process of selecting a level and then computing and storing the sparse submatrices for that level, and the average time per iteration of numerical solution. For comparison, we also include the iteration times for the three other implementations: MTBDDs, sparse matrices and the (unoptimised) hybrid approach.

The results are very satisfying. Firstly, we note that the construction process for the optimised hybrid technique is very fast. More importantly, we find that we have significantly improved the average time per iteration of numerical solution using the optimised version of our hybrid approach. The time is now comparable with those for sparse matrices.

Furthermore, we are again able to handle larger models than with sparse matrices. This is due to the relative storage requirements of the methods, illustrated in Table 6.5. In each case, it gives both the memory used to store the matrix and the total memory. The latter figure includes the storage requirements for iteration vectors. These remain the same as in Figure 6.3 and are not repeated.

Clearly, the optimised approach will need more memory than the unoptimised version: firstly, we have to store an extra pointer on each node; and secondly, we have to store the

Model	N	States	Construction (sec.)	Time per iteration (sec.)			
				MTBDD	Sparse	Hybrid	Hybr. opt.
Kanban system	1	160	0.01	0.03	0.001	0.001	0.001
	2	4,600	0.05	2.22	0.002	0.03	0.003
	3	58,400	0.10	45.5	0.05	0.35	0.05
	4	454,475	0.12	-	0.41	4.79	0.53
	5	2,546,432	0.09	-	2.70	26.1	3.19
	6	11,261,376	0.11	-	-	123.8	16.1
Polling system	8	3,072	0.01	0.31	0.001	0.004	0.001
	10	15,360	0.02	13.1	0.01	0.03	0.01
	12	73,728	0.04	-	0.05	0.16	0.05
	14	344,064	0.04	-	0.30	0.89	0.31
	16	1,572,864	0.04	-	1.56	4.46	1.67
	18	7,077,888	0.05	-	-	23.1	8.51
BRP	500	27,505	0.19	0.38	0.01	0.12	0.01
	1,000	55,005	0.25	0.67	0.01	0.24	0.02
	1,500	82,505	0.33	0.99	0.02	0.36	0.03
	2,000	110,005	0.32	1.26	0.03	0.47	0.04
	2,500	137,505	0.33	1.53	0.04	0.59	0.05
FMS	3	6,520	0.22	6.13	0.002	0.11	0.003
	4	35,910	0.33	48.3	0.02	0.84	0.03
	5	152,712	0.27	-	0.10	3.35	0.16
	6	537,768	0.31	-	0.37	13.3	0.57
	7	1,639,440	0.32	-	1.14	33.6	2.55
	8	4,459,455	0.45	-	-	102.0	8.17

Table 6.4: Timing statistics for the optimised hybrid implementation

Model	N	Matrix storage (KB)			Total storage (KB)		
		Sparse	Hybrid	Hybr. Opt.	Sparse	Hybrid	Hybr. Opt.
Kanban system	1	8	10	20	11	13	23
	2	348	33	387	455	141	495
	3	5,455	48	866	6,823	1,417	2,234
	4	48,414	96	858	59,066	10,748	11,511
	5	296,588	123	671	356,270	59,805	60,353
	6	-	154	944	-	264,092	264,881
Polling system	8	186	12	201	258	84	273
	10	1,110	19	657	1,470	379	1017
	12	6,192	26	794	7,920	1,754	2,522
	14	32,928	34	804	40,992	8,098	8,868
	16	168,960	44	815	205,824	36,908	37,679
	18	-	55	829	-	165,943	166,717
BRP	500	441	59	512	1,301	919	1,372
	1,000	883	61	726	2,602	1,781	2,446
	1,500	1,324	63	1,055	3,902	2,643	3,635
	2,000	1,766	63	1,011	5,203	3,499	4,447
	2,500	2,207	65	1,006	6,504	4,361	5,302
FMS	3	451	213	706	655	416	910
	4	2,711	546	1,556	3,835	1,668	2,680
	5	11,990	974	1,885	16,762	5,746	6,657
	6	42,744	1,497	2,645	59,548	18,302	19,449
	7	129,853	2,324	3,666	181,085	53,556	54,898
	8	-	4,132	5,686	-	143,490	145,042

Table 6.5: Memory requirements for the optimised hybrid implementation

small sparse matrices. Despite these increases, though, it is evident that, in general, our approach still requires far less memory than sparse matrices.

The results here were obtained, like all the others in this thesis, on a 440 MHz 512 MB Sun Ultra10. We were able to run all the same experiments as in the previous section. In addition, we again confirmed the scalability of our techniques on a larger, shared machine. As before, we find that in general, we are able to increase the size of models which can be analysed by approximately one order of magnitude.

6.5 Extending to MDPs

The hybrid approach described in the previous sections dealt specifically with the problem of performing a matrix-vector multiplication, as required by the model checking algorithms

for DTMCs and CTMCs. We now extend our technique to handle MDPs. As ever, the complication is the addition of nondeterminism.

From an implementation point of view, the difference can be explained as follows. In a standard matrix-vector multiplication, as required for DTMC or CTMC model checking, each entry of the resulting vector, corresponding to a particular state in the DTMC or CTMC, is computed from a summation of products. In an iteration of model checking for an MDP, the value for a state is determined by computing a summation of products for each nondeterministic choice in that state, and then selecting either the maximum or minimum of these values. For clarity, in the remainder of this section, we will consider the case where the maximum is taken. The case for minimum, however, is almost identical.

In our MTBDD implementation, described in Section 5.3.2, we thought of the MDP as being represented by a non-square matrix. This allowed each iteration to be performed using a matrix-vector multiplication and then a maximum operation, both of which can be done efficiently with MTBDDs. The disadvantage of this approach is that it creates an intermediate result: a vector of size equal to the number of nondeterministic choices in the MDP. This is potentially much larger than the number of states in the MDP. Hence, in situations where vectors are stored explicitly with arrays, such as in our sparse matrix or hybrid implementations, this approach is not practical. We have already seen that the storage required for vectors represents the limiting factor for the applicability of these techniques.

In Section 3.6.1, we showed how this problem could be resolved in the sparse matrix case, by combining the matrix-vector multiplication and maximum operation. The algorithm considers each state, one by one, computing and storing its maximum value before moving on to the next. Note that this is only possible because, in our sparse matrix storage scheme for MDPs, matrix entries are ordered by state.

As we saw in the previous sections though, with our hybrid approach, entries from matrices for DTMCs and CTMCs are not extracted from MTBDDs in a meaningful order. This is because of the interleaving of their row and variable variables. Since in our preferred encoding of MDPs, as presented in Chapter 4, the MTBDD variables corresponding to rows, columns and nondeterministic choices are again interleaved, this will also apply to the extraction of matrix entries for MDPs. Therefore, we cannot perform a direct analogue of the sparse matrix operation.

We have already concluded that sacrificing the interleaving of row and column variables is impractical due to the large size of the resulting MTBDDs. Hence, the only feasible way to resolve this situation is the following. We revert to the 'top' MTBDD variable ordering for MDPs described in Section 4.2.2, where all the nondeterministic variables come first in the ordering, followed by all the row and column variables, interleaved as for

```
MdpIterMax({M_i}_{i=1..m}, b)
 1.  for (i := 0 ... |S|−1)
 2.      res[i] := −1
 3.  endfor
 4.  for (i = 1 ... m)
 5.      for (j := 0 ... |S|−1)
 6.          tmp[j] := −1
 7.      endfor
 8.      for (each element (r, c) = v of M_i)
 9.          if (tmp[r] = −1) then tmp[r] := 0
10.          tmp[r] := tmp[r] + v × b[c]
11.      endfor
12.      for (j := 0 ... |S|−1)
13.          if (tmp[j] > res[j]) then res[j] := tmp[j]
14.      endfor
15.  endfor
16.  return res
```

Figure 6.16: Pseudo-code for a single iteration of MDP solution

DTMC and CTMC models. In this way, we effectively partition the MDP into several, say m, transition matrices. A depth-first traversal of the MTBDD for the MDP will extract each matrix, in its entirety, one by one. This provides a way of performing the combined multiplication and maximum operation using just two arrays to store vectors. Essentially, we will perform m individual matrix-vector operations, one for each matrix. We will use one array to store the result of a single multiplication operation and a second to keep track of the maximum value found so far for each state.

Figure 6.16 shows the pseudo-code for this process. We assume that the m matrices are $\mathbf{M}_1, \ldots, \mathbf{M}_m$, the vector to be multiplied by is stored in an array b, and that there are $|S|$ states. The resulting vector is put in an array *res*, and an array *tmp* is used to store each individual matrix-vector multiplication. In our implementation, the implicit loop over all matrix entries in line 8 would be done by traversing the appropriate MTBDD.

We must be careful to remember that, since there are not necessarily m choices in every state, some matrices may not contain a row for every state. This must be taken into consideration when computing the maximum. Because we only deal with probability distributions, every value in every vector will always be in the range $[0, 1]$. Hence, we initialise our vector so that all entries are -1, guaranteed to be below this range, and then states which have been missed are easily detected.

Hence, we have found a way to apply our hybrid technique to MDP models. Further-

more, we can just reuse our existing algorithms, since we do the same as before, but for several matrices. This applies to each and every phase of the process, i.e. for construction we perform offset labelling and explicit submatrix construction m times, and in each iteration of numerical solution we carry out m traversals. The disadvantages are that, firstly, we have resorted to our second choice of MDP encoding so the MTBDDs will not be as small as possible, and secondly, we require an extra vector.

6.5.1 Results

We now present experimental results of applying the hybrid approach, as just described, to some MDP model checking case studies. In our work, we found a distinct lack of such examples for which numerical computation was actually required, the majority being solvable with precomputation algorithms alone. Of the few remaining cases, most were the examples where we found MTBDDs outperformed explicit approaches. Tables 6.6 and 6.7 present statistics for three instances: Lehmann and Rabin's randomised dining philosophers algorithm [LR81], the coin protocol of [AH90] and the FireWire root contention protocol model of [SV99]. For the first, we check a PCTL bounded until property, and for the second two a PCTL until property. In each case we use a lower probability bound, and hence compute minimum probabilities. For the PCTL until properties, though, we model check over fair adversaries only so this actually reduces to the problem of computing maximum probabilities.

Table 6.6 gives timing statistics: the time for construction (which includes offset labelling *and* explicit submatrix construction), the number of iterations required for model checking and the average time per iteration. For comparison, we also give the time per iteration for the equivalent MTBDD and sparse matrix implementations.

The results are clearly not as good as for DTMCs and CTMCs. The times for construction remain small, but the time per iteration is poor in comparison to the other implementations. We already know that MTBDDs perform very well on these examples, particularly for the coin protocol and FireWire case studies, but the comparison with sparse matrices is disappointing.

The main problem seems to be that the fast, efficient algorithms we have developed to explicitly extract matrix entries from MTBDDs are not as well suited to the computation required for model checking of MDPs. We are forced to split the MDP into m separate matrices, where m is the maximum number of nondeterministic choices in any state. From examination of our case studies, we see that, in typical MDPs, most states will have significantly fewer that m choices. Hence, a lot of time is wasted traversing MTBDDs and computing maximums or minimums, work that the sparse matrix version does not do.

Model	Param.s	States	Construction (sec.)	Iterations	Time per iteration (sec.)		
					MTBDD	Sparse	Hybrid
Dining philosophers (N, K)	$3, 4$	28,940	0.27	22	0.12	0.01	0.02
	$3, 5$	47,204	0.47	26	0.14	0.01	0.03
	$3, 6$	69,986	0.50	30	0.16	0.02	0.05
	$3, 8$	129,254	0.80	38	0.22	0.04	0.11
	$4, 4$	729,080	6.53	22	1.24	0.19	0.81
	$4, 5$	1,692,144	15.3	27	2.57	0.48	2.04
	$4, 6$	3,269,200	38.1	31	4.62	1.00	3.91
Coin protocol (N, K)	$2, 8$	1,040	0.01	6,132	0.02	0.0003	0.0004
	$4, 8$	84,096	0.06	21,110	0.34	0.04	0.07
	$6, 8$	4,612,864	0.15	42,967	1.83	3.37	6.21
	$8, 8$	2.2×10^8	-	70,669	4.56	-	-
	$10, 6$	7.6×10^9	-	63,241	9.79	-	-
FireWire (N)	200	68,185	0.35	169	0.006	0.02	0.05
	400	220,733	0.50	375	0.02	0.07	0.27
	600	375,933	0.48	581	0.04	0.12	0.47
	800	531,133	0.50	789	0.05	0.17	0.73
	1,000	686,333	0.46	995	0.06	0.22	0.91
	2,000	1,462,333	0.51	2,027	0.11	0.47	2.09
	3,000	2,238,333	0.50	3,015	0.17	0.71	3.22

Table 6.6: Timing statistics for the hybrid approach on MDPs

Model	Parameters	MTBDD nodes increase (%)	Matrix storage (KB)		Total storage (KB)	
			Sparse	Hybrid	Sparse	Hybrid
Dining philosophers (N, K)	$3, 4$	114	663	1,292	1,341	2,196
	$3, 5$	123	1,157	1,406	2,264	2,882
	$3, 6$	133	1,791	1,522	3,432	3,710
	$3, 8$	147	3,495	1,790	6,525	5,830
	$4, 4$	169	16,357	4,442	33,445	27,226
	$4, 5$	166	43,564	6,147	83,224	59,027
	$4, 6$	176	91,763	8,740	168,386	110,904
Coin protocol (N, K)	$2, 8$	48	32	43	56	75
	$4, 8$	45	3,247	697	5,218	3,321
	$6, 8$	42	183,539	935	291,653	145,087
FireWire (N)	200	90	2,083	963	3,682	3,095
	400	95	7,074	1,190	12,249	8, 090
	600	95	12,163	1,288	20,974	13,037
	800	95	17,252	1,386	29,702	17,986
	1,000	95	22,341	1,095	38,427	22,543
	2,000	95	47,787	1,193	82,062	46,893
	3,000	95	73,232	1,291	125,693	71,239

Table 6.7: Memory requirements for the hybrid approach on MDPs

On a more positive note, we see that the memory usage is still more favourable for the hybrid approach. Table 6.7 gives the amount of memory required for the hybrid and sparse matrix implementations. We give both the memory to store the matrix and the total amount, which also includes arrays for the iteration vectors. In addition, we show the percentage increase in the number of nodes for the construction of the offset-labelled MTBDDs. The figures for the latter are significantly higher than those observed for DTMCs and CTMCs. One possible reason for this is that, in our current implementation, we perform the construction of offset-labelled MTBDDs and explicit submatrices separately for each of the m matrices. Hence, it is likely that the memory usage could be improved by exploiting any sharing which occurs between these matrices. Despite all this, because of the small size of the original MTBDDs, the sparse matrix storage still requires far more memory than the hybrid approach. This means that, even with the addition of an extra vector (4 for the hybrid approach, as opposed to 3 with sparse matrices), the total memory usage is still less for the symbolic implementation than the explicit one.

6.6 Extending to Gauss-Seidel and SOR

Recall that, so far, our symbolic implementations of iterative numerical methods have been restricted to those based on matrix-vector multiplication. This includes both the pure MTBDD implementation of Chapter 5 and the hybrid approach presented in this chapter. For some parts of our probabilistic model checking algorithms (e.g. for the PCTL until operator over MDPs and the CSL time-bounded over CTMCs), this limitation is irrelevant since there is only one viable solution method and this can be efficiently expressed in terms of matrix-vector multiplication.

For others, this is not the case. In particular, model checking for the PCTL until operator over DTMCs and the CSL steady-state operator over CTMCs requires solution of a linear equation system. Here, our symbolic implementations are limited to the Jacobi and JOR methods, while more efficient alternatives such as Gauss-Seidel and SOR are not feasible. In this section, we propose an extension to our hybrid approach which allows a modified form of these more efficient methods to be applied.

Note that an implementation of Gauss-Seidel can easily be extended to SOR in exactly the same fashion that Jacobi can be extended to JOR. Hence, in the remainder of this section, we will focus on the Gauss-Seidel method. The method presented is, however, equally applicable to SOR.

6.6.1 Jacobi versus Gauss-Seidel

Gauss-Seidel has two principal advantages over the Jacobi method. Firstly, it generally converges quicker, reducing the total time required for solution. Secondly, the solution vector can be stored using a single array, rather than two as needed for Jacobi. We have already seen that the memory requirements for storing vectors constitute the limiting factor for our hybrid approach, so this would present a significant advantage.

The difference between the two methods is the way in which the solution vector is updated at each iteration. Assume we are solving the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$, where $\mathbf{A}$ is an $|S| \times |S|$ matrix and $\underline{b}$ a vector of length $|S|$. We describe the iterative methods in terms of how $\underline{x}^{(k)}(i)$, the ith element of the kth vector, is obtained from $\mathbf{A}$, $\underline{b}$, and the previous vector $\underline{x}^{(k-1)}$. For the Jacobi method:

$$\underline{x}^{(k)}(i) \quad := \quad \left(\underline{b}(i) - \sum_{j \neq i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j) \right) / \mathbf{A}(i,i)$$

and for the Gauss-Seidel method:

$$\underline{x}^{(k)}(i) \quad := \quad \left(\underline{b}(i) - \sum_{j < i} \mathbf{A}(i,j) \cdot \underline{x}^{(k)}(j) - \sum_{j > i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j) \right) / \mathbf{A}(i,i)$$

```
JacobiIteration(A', d)
1.  for (i := 0 ... |S|−1)
2.      x^new(i) := 0
3.  endfor
4.  for (each element (r, c) = v in A')
5.      x^new(r) := x^new(r) + v · x(c)
6.  endfor
7.  for (i := 0 ... |S|−1)
8.      x(i) := (x^new(i) + b(i))/d(i)
9.  endfor
```

```
GaussSeidelIteration(A', d)
1.  for (i := 0 ... |S|−1)
2.      x^new := 0
3.      for (each element (r, c) = v in row i of A')
4.          x^new := x^new + v · x(c)
5.      endfor
6.      x(i) := (x^new + b(i))/d(i)
7.  endfor
```

Figure 6.17: Implementation of an iteration of Jacobi and Gauss-Seidel

Note how the Gauss-Seidel method uses the most recent approximation of the solution vector available. Hence, to compute $\underline{x}^{(k)}(i)$, it uses $\underline{x}^{(k)}(j)$ for $j < i$ but $\underline{x}^{(k-1)}(j)$ for $j > i$. Intuitively, this is why Gauss-Seidel converges faster than Jacobi.

Pseudo-code illustrating how a single iteration of each method is implemented in practice is shown in Figure 6.17. Note that we do not use the matrix $\mathbf{A}$ directly. Instead, we assume that the non-diagonal elements of $\mathbf{A}$ are stored, negated, in a matrix $\mathbf{A}'$, and that the diagonal elements of $\mathbf{A}$ are stored in a vector $\underline{d}$, i.e. for $0 \leq i, j \leq |S|-1$:

- $\mathbf{A}'(i,j) = -\mathbf{A}(i,j)$ if $i \neq j$ and 0 otherwise
- $\underline{d}(i) = \mathbf{A}(i,i)$

In practice, diagonal and non-diagonal matrix entries are often stored separately in this way because they are treated differently by the algorithm. The reason we negate the non-diagonal entries is that it allows us to clarify the link between these methods and matrix-vector multiplication. Note how lines 1–6 of the Jacobi iteration essentially correspond to multiplying the matrix $\mathbf{A}'$ by vector $\underline{x}$ and placing the result in $\underline{x}^{new}$. This is why we were able to construct our symbolic implementations of the Jacobi method using matrix-vector multiplication. Gauss-Seidel, which accesses each row of the matrix $\mathbf{A}'$ individually,

cannot be formulated in this way. Hence, we have only been able to implement it with sparse matrices.

The relative memory requirements of Jacobi and Gauss-Seidel are also clarified by the pseudo-code in Figure 6.17. Both methods use a vector $\underline{x}$ of size $|S|$ to store the current approximation to the solution. By the end of the iteration, this will have been updated to contain the next approximation. The Jacobi method requires an additional vector $\underline{x}^{new}$ of equal size to store intermediate results For Gauss-Seidel, on the other hand, a single, scalar variable x^{new} suffices.

6.6.2 Pseudo Gauss-Seidel

We now propose a modified version of Gauss-Seidel which we will able to implement symbolically using our hybrid approach. The method is essentially a compromise between the Jacobi and Gauss-Seidel methods. We will refer to it as *Pseudo Gauss-Seidel*. Let us assume that our matrix $\mathbf{A}$ is divided into a number of blocks. In the above, we indexed $\mathbf{A}$ over $0, \ldots, |S|-1$ where S is the state space of the probabilistic model being analysed. Assume now that S is divided into m contiguous partitions $S_1, \ldots, S_m$. Using this, the matrix $\mathbf{A}$ can be split into m^2 blocks, $\{\mathbf{A}_{p,q} \mid 1 \leq p, q \leq m\}$, where the rows and columns of block $\mathbf{A}_{p,q}$ correspond to the states in S_p and S_q, respectively, i.e. block $\mathbf{A}_{p,q}$ is of size $|S_p|$ by $|S_q|$. We introduce the additional notation $N_p = \sum_{i=1}^{p-1} |S_i|$. For $1 \leq p \leq m$, partition S_p includes indices N_p up to $N_{p+1} - 1$. Finally, we denote by $block(i)$ the block containing index i, i.e. the unique value $1 \leq p \leq m$ such that $N_p \leq i < N_{p+1}$. Pseudo Gauss-Seidel can then be described as:

$$\underline{x}^{(k)}(i) \;:=\; \left(\underline{b}(i) \;-\; \sum_{j<N_{block(i)}} \mathbf{A}(i,j) \cdot \underline{x}^{(k)}(j) \;-\; \sum_{j \geq N_{block(i)}, j \neq i} \mathbf{A}(i,j) \cdot \underline{x}^{(k-1)}(j) \right) / \mathbf{A}(i,i)$$

The method works as follows. Each iteration is divided into m phases. In the pth phase, the method updates elements of the solution vector corresponding to the states in S_p. It does this using the most recent approximation for each entry of the solution vector available, i.e. it uses values from the previous iteration for entries corresponding to sets $S_p, \ldots, S_m$ and values from earlier phases of the current iteration for entries corresponding to sets $S_1, \ldots, S_{p-1}$. We can relate Pseudo Gauss-Seidel to the previous two iterative methods by considering Jacobi to be the case where $m = 1$ and Gauss-Seidel to be the case where $m = |S|$.

The pseudo-code for a single iteration of Pseudo Gauss-Seidel is shown in Figure 6.18. The notation used is the same as in Figure 6.17: $\mathbf{A}$ is split into $\mathbf{A}'$ and $\underline{d}$, the solution vector is stored in $\underline{x}$ and intermediate results are stored in the vector $\underline{x}^{new}$. The matrix

```
PseudoGaussSeidelIteration(A', d)
1.   for (p := 1 ... m)
2.       for (i := 0 ... |S_p|−1)
3.           x^new(i) := 0
4.       endfor
5.       for (each element (r, c) = v in a block A'_{p,q})
6.           x^new(r − N_p) := x^new(r − N_p) + v · x(c)
7.       endfor
8.       for (i := 0 ... |S_p|−1)
9.           x(N_p+i) := (x^new(i) + b(N_p+i))/d(N_p+i)
10.      endfor
11.  endfor
```

Figure 6.18: Implementation of an iteration of Pseudo Gauss-Seidel

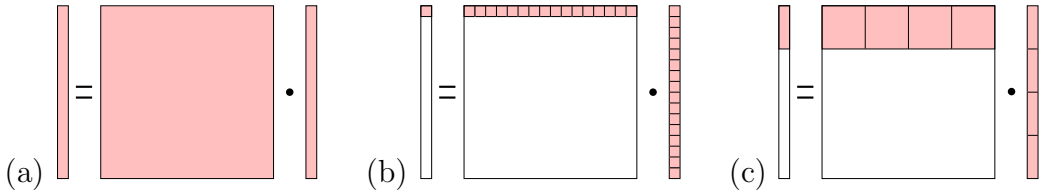


Figure 6.19: Matrix access for (a) Jacobi (b) Gauss-Seidel (c) Pseudo Gauss-Seidel

$\mathbf{A}'$ is split into m^2 blocks in an identical fashion to $\mathbf{A}$.

Note that the pth phase of an iteration only requires access to entries from the pth row of blocks in $\mathbf{A}'$, i.e. $\mathbf{A}'_{p,q}$ for $1 \leq q \leq m$. This can be compared to Jacobi, where all the entries of $\mathbf{A}'$ were used in one go, and to Gauss-Seidel, where each of the $|S|$ steps in an iteration used a single row of $\mathbf{A}'$. Figure 6.19 illustrates this idea graphically: (a) refers to Jacobi, (b) to Gauss-Seidel, and (c) to Pseudo Gauss-Seidel. In each case, the figure demonstrates the part of an iteration which uses the elements of matrix $\mathbf{A}'$ and the elements of vector $\underline{x}$ to calculate new entries for the solution vector. In the pseudo-code for Jacobi, Gauss-Seidel and Pseudo Gauss-Seidel, this corresponds to lines 1–6, 2–5 and 2–7, respectively.

The intuition behind the Pseudo Gauss-Seidel method is as follows. Firstly, since each iteration now uses newer approximations for *some* elements of the solution vector (each phase uses the values computed by earlier phases of the same iteration), we would hope that the method converges more quickly than Jacobi. Secondly, the vector $\underline{x}^{new}$, used to store intermediate results, need only be of size $\max_{p=1,\ldots,m} |S_p|$, rather than $|S|$, as in Jacobi. Basically, we would expect to obtain the same benefits of the Gauss-Seidel

method, only to a lesser extent. The performance of the algorithm will vary, depending on the number of blocks that the matrix is split into and the maximum size of these blocks.

Note that, although Pseudo Gauss-Seidel is based on a division into blocks, it is entirely different from the existing 'Block Gauss-Seidel' iterative method. The latter is an attempt to improve the Gauss-Seidel method by reducing the number of iterations required for convergence. We, on the other hand, hope to require less iterations than Jacobi, but would not expect to use less than Gauss-Seidel.

On the subject of convergence, since we have introduced a non-standard iterative method, it is important to check that it actually converges. Fortunately, because of its similarity to the existing Jacobi and Gauss-Seidel methods, Pseudo-Gauss Seidel can be seen to exhibit similar behaviour in this respect. We rely on existing results for Jacobi and Gauss-Seidel, as presented for example in [Ste94].

Jacobi and Gauss-Seidel belong to a class of methods which can be described by *splittings*. An iterative method of this type, for the solution of the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$, is associated with a splitting of the matrix $\mathbf{A} = \mathbf{M} - \mathbf{N}$. An iteration of the method can then be written as $\underline{x}^{(k)} := \mathbf{M}^{-1} \cdot \mathbf{N} \cdot \underline{x}^{(k-1)} + \mathbf{M}^{-1} \cdot \underline{b}$. For Jacobi, $\mathbf{M}$ contains the diagonal entries of $\mathbf{A}$ and $\mathbf{N}$ the negated non-diagonal entries. For Gauss-Seidel, $\mathbf{M}$ contains the diagonal and lower-triangular entries of $\mathbf{A}$, while $\mathbf{N}$ contains the negated upper-triangular entries. Pseudo Gauss-Seidel is also based on a splitting, where $\mathbf{M}$ is defined as follows:

$$\mathbf{M}(i,j) = \begin{cases} \mathbf{A}(i,j) & \text{if } j < N_{block(i)} \text{ or } j = i \\ 0 & \text{otherwise} \end{cases}$$

and $\mathbf{N}$ is equal to $\mathbf{M} - \mathbf{A}$. If the splitting $\mathbf{A} = \mathbf{M} - \mathbf{N}$ is such that both $\mathbf{M}^{-1}$ and $\mathbf{N}$ are non-negative, then it is known as a *regular splitting*.

In [Ste94], it is shown that solving the linear equation system $\underline{\pi} \cdot \mathbf{Q} = \underline{0}$ to compute steady-state probabilities for a CTMC is equivalent to solving a system of the form $(\mathbf{I} - \mathbf{P}^T) \cdot \underline{x} = \underline{0}$, where $\mathbf{P}$ is a stochastic matrix (one where each row contains positive entries summing to one). The convergence properties for both Jacobi and Gauss-Seidel then rely on the fact that, for such a system, the methods are based on regular splittings. Crucially, if $\mathbf{M}$ can be obtained from $\mathbf{I} - \mathbf{P}^T$ only by setting non-diagonal elements to zero, then the splitting is regular. As can be seen from above, this is the case for all three of the methods we consider: Jacobi, Gauss-Seidel *and* Pseudo Gauss-Seidel.

Note that, in the other instance where we require solution of a linear equation system, namely model checking the PCTL until operator over a DTMC, the system is of the form $(\mathbf{I} - \mathbf{P}) \cdot \underline{x} = \underline{b}$, where $\mathbf{P}$ is a stochastic matrix. Using again the results from [Ste94], splittings of $\mathbf{I} - \mathbf{P}$ for Jacobi, Gauss-Seidel and Pseudo Gauss-Seidel are always regular.

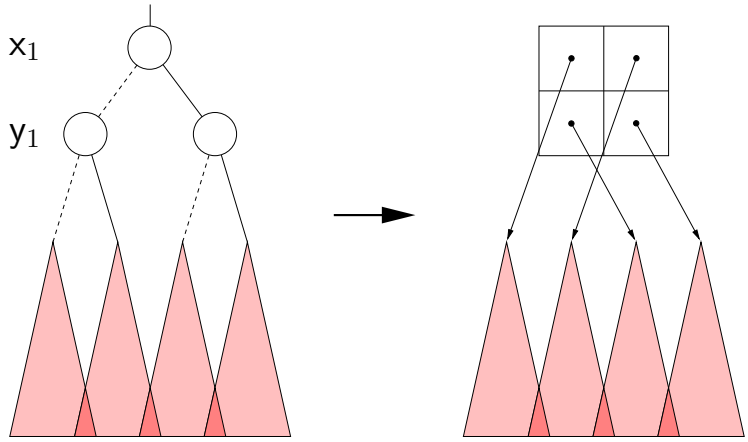


Figure 6.20: Storing subgraphs of an offset labelled MTBDD explicitly

6.6.3 Implementation

The reason that we have introduced the Pseudo Gauss-Seidel method is because it can be implemented with the hybrid approach we developed earlier in this chapter. We have already seen, on several occasions, how submatrices of a matrix represented by an (offset-labelled) MTBDD correspond to subgraphs of the same data structure. To extract a row of matrix blocks, as required by Pseudo Gauss-Seidel, we can simply extract each block in the row individually, using our existing MTBDD traversal algorithm TraverseOpt on the appropriate subgraphs of the offset-labelled MTBDD.

The exact procedure we adopt is as follows. Assume that we have an offset-labelled MTBDD M representing the matrix $\mathbf{A}'$, as used by the iterative methods in the previous section. We begin by selecting an integer k. By performing a traversal of the top $2k$ levels of M (i.e. k row variables and k column variables), we can split it into $(2^k)^2$ smaller offset-labelled MTBDDs, each representing a submatrix of $\mathbf{A}'$. We then explicitly store pointers to each of these MTBDDs, for example in a two-dimensional array. Figure 6.20 illustrates this idea for $k = 1$; the grey triangles denote the lower portions of the offset-labelled MTBDD. We can now perform Pseudo Gauss-Seidel, letting $m = 2^k$. The entries in each of the m^2 blocks of $\mathbf{A}'$ are extracted using the TraverseOpt algorithm and the pointers to each block's offset-labelled MTBDD can be accessed quickly and easily from their explicit storage.

The number of blocks for which we store pointers will grow exponentially as k is increased. Furthermore, because our matrices are sparse, many of these blocks will only contain zeros. Hence, we actually store a sparse matrix data structure of MTBDD pointers rather than a 2-dimensional array. Each row can still be accessed quickly, exactly as for a

conventional sparse matrix, but the items extracted from the data structure are pointers to MTBDD nodes, not floating point values.

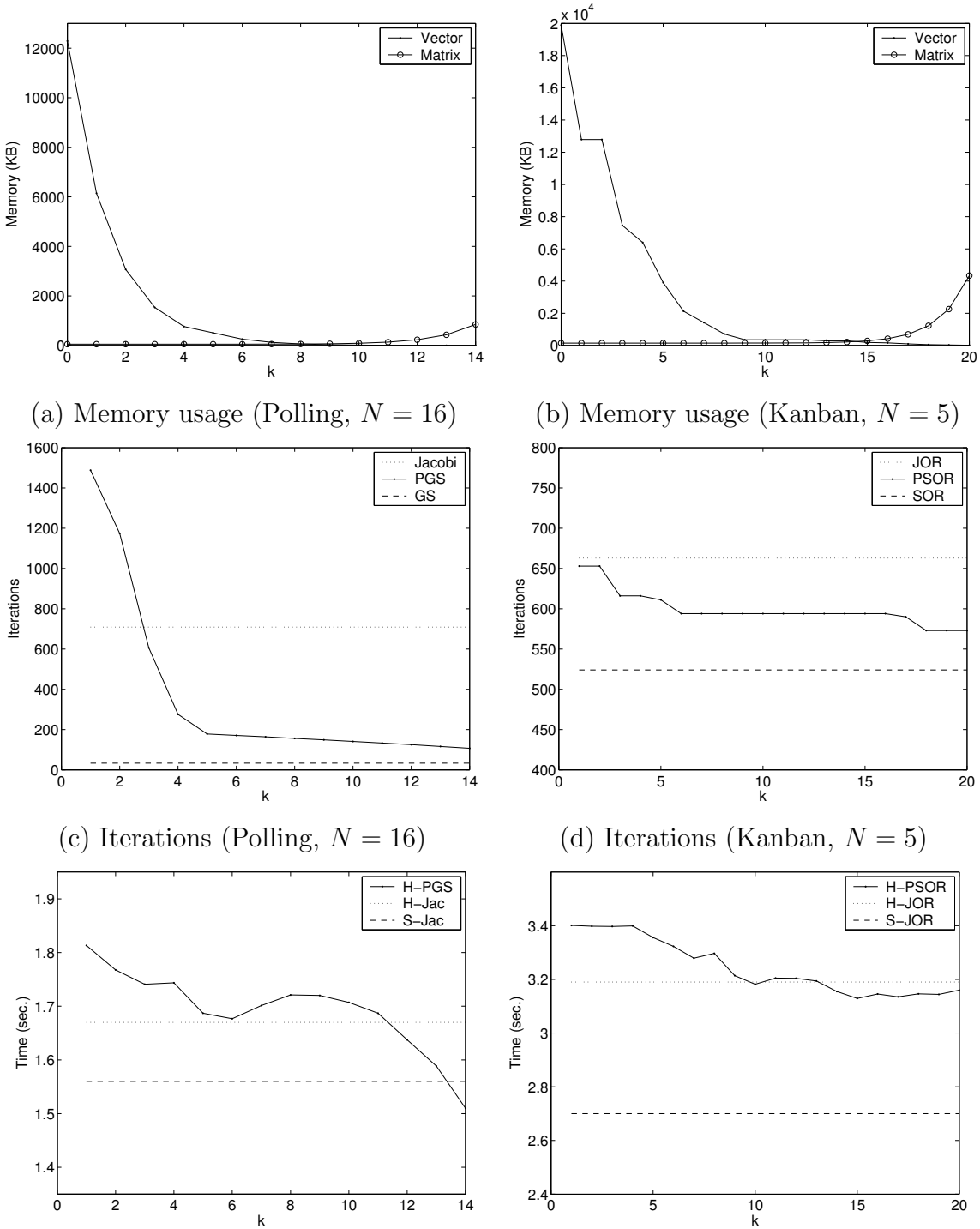
6.6.4 Results

We have integrated the method outlined above into our symbolic model checker, extending our existing implementation of the Jacobi method into Pseudo-Gauss Seidel and the JOR method into 'Pseudo SOR' in identical fashion. We then applied these to the model checking case studies from the previous sections. Clearly, these techniques are only applicable when solving a linear equation system, i.e. when model checking either a PCTL until formula for a DTMC or a CSL steady-state formula for a CTMC. Hence, we use the polling system, bounded retransmission protocol (BRP) and Kanban system case studies. For the first two, we previously used the Jacobi method so we will now apply Pseudo Gauss-Seidel. For the Kanban system, we used JOR ($\omega = 0.9$) since Jacobi did not converge. In this case, we apply Pseudo SOR.

We begin by considering two specific examples, the polling system for $N = 16$ and the Kanban system for $N = 5$. We will use these to investigate the effect that the choice of k has on the performance of the algorithm. Figure 6.21 shows three plots for each of the two examples.

Figures 6.21(a) and (b) show how the memory usage varies. We plot both the amount needed to store the vector of intermediate results ($\underline{x}^{new}$) and the amount for explicit storage of pointers into the MTBDD storing the matrix ($\mathbf{A}'$). The variation in memory is exactly as we would expect. As k increases, the size of the matrix blocks becomes smaller so the vector $\underline{x}^{new}$ decreases in size. Conversely, the increase in the number of blocks causes the memory required for storing the MTBDD pointers to rise. It seems though, that by choosing a sensible value of k, we can produce a large drop in the memory for the vector without a significant increase in that for the matrix.

In Figures 6.21(c) and (d), we show the number of iterations taken for Pseudo Gauss-Seidel (or Pseudo SOR) to converge for each value of k. For comparison, we also give the number required by the standard iterative methods. We compare Pseudo Gauss-Seidel to Jacobi and Gauss Seidel, and Pseudo SOR to JOR and SOR. As hoped the number of iterations required decreases as we increase the value of k. For the Kanban system, the range of values is neatly bounded above and below by the number of iterations for the two standard methods. For the polling system, we see that for very small values of k, Pseudo Gauss-Seidel requires more iterations than either Jacobi or Gauss-Seidel. The overall conclusion, however, is that we can successfully reduce the required number of iterations using the 'Pseudo' methods.



(a) Memory usage (Polling, $N = 16$)

(b) Memory usage (Kanban, $N = 5$)

(c) Iterations (Polling, $N = 16$)

(d) Iterations (Kanban, $N = 5$)

(e) Iteration time (Polling, $N = 16$)

(f) Iteration time (Kanban, $N = 5$)

Figure 6.21: Statistics for Pseudo Gauss-Seidel and SOR

Finally, Figures 6.21(e) and (f) give the average time per iteration of Pseudo Gauss-Seidel or Pseudo SOR. The amount of work done in each iteration does not differ greatly from that done in the original hybrid implementation (i.e. Jacobi or JOR). The time for the latter is shown by the dotted, horizontal line. We see that for small values of k, we require slightly more time, attributed to the extra book-keeping performed, and for larger k, we need slightly less time. This is because it is faster to extract the MTBDDs for each matrix block from explicit storage than it is to obtain them by traversing the top few levels of the MTBDD every iteration. For reference, we also show the average time per iteration for sparse matrices (Jacobi or JOR) as a dashed, horizontal line. Interestingly, for large values of k on the polling system example, we can now outperform the explicit implementation. This seems to be because, as described in Section 6.4.4, it is more efficient to manipulate small matrices than to work with the entire matrix.

In summary, we see that increasing the value of k reduces the memory for vector storage, the number of iterations *and* the average time per iteration. The only negative consequence is that the matrix storage grows. Fortunately, we can adopt a similar approach to that taken in Section 6.4, when selecting a level for which to compute explicit submatrices. The amount of memory which would be required for the explicit storage of MTBDD pointers for a given value of k is easy to compute in advance. Hence, we choose an upper limit on memory and then determine the maximum value of k for which this limit is not exceeded. We will again take this limit to be 1024 KB. Note that, on smaller examples, these two schemes (division into matrix blocks and explicit storage of submatrices) could potentially interfere with each other. Where there is a possible conflict, we give preference to the techniques of this section, i.e. division into matrix blocks.

Finally, we present experimental results for the techniques described above on the full range of examples from our three case studies. Tables 6.8 and 6.9 show statistics for timing and memory usage, respectively. For reasons of space, we identify each model with a letter, as explained in the key below each table.

Table 6.8 shows the number of iterations required for convergence, the average time per iteration, and the total time for three different implementations: Gauss-Seidel (GS) using sparse matrices, and Jacobi (Jac) and Pseudo Gauss-Seidel (PGS) using the hybrid approach. For the Kanban system example, these are replaced by SOR, JOR and Pseudo SOR (PSOR), respectively. We conclude that changing the hybrid approach to use the 'Pseudo' methods reduces both the average iteration time and the number of iterations, the net result being a useful decrease in the total time required. The overall time for sparse matrices is still quicker. This is largely because the number of iterations is even less using (real) Gauss-Seidel or SOR. We have still achieved a clear improvement in our hybrid implementation though.

Model	Iterations			Time per iteration (sec.)			Total time (sec.)		
	GS	Jac	PGS	GS	Jac	PGS	GS	Jac	PGS
A	25	310	25	0.001	0.001	0.003	0.03	0.31	0.08
B	27	406	27	0.01	0.01	0.02	0.27	4.06	0.54
C	29	505	64	0.06	0.05	0.06	1.74	25.3	3.84
D	31	606	87	0.32	0.31	0.30	9.92	188	26.1
E	33	709	107	1.64	1.67	1.51	54.1	1,184	162
F	-	814	127	-	8.51	8.03	-	6,927	1,020
G	2,046	3,086	2,557	0.01	0.01	0.01	20.5	30.9	25.6
H	4,071	6,136	5,088	0.01	0.02	0.02	40.7	123	102
I	6,095	9,184	7,617	0.02	0.03	0.03	122	276	229
J	8,117	12,230	10,146	0.03	0.04	0.04	244	489	406
K	10,140	15,274	12,674	0.04	0.05	0.05	406	764	634
	SOR	JOR	PSOR	SOR	JOR	PSOR	SOR	JOR	PSOR
L	140	101	140	0.001	0.001	0.001	0.14	0.10	0.14
M	136	166	142	0.002	0.003	0.003	0.27	0.50	0.43
N	235	300	258	0.05	0.05	0.05	11.8	15.0	12.9
O	367	466	414	0.45	0.53	0.52	165	247	215
P	524	663	590	2.79	3.19	3.14	1,462	2,115	1,853
Q	-	891	794	-	16.1	15.6	-	14,336	12,386

Key					
Polling system		BRP		Kanban system	
A	$N = 8$	G	$N = 500$	L	$N = 1$
B	$N = 10$	H	$N = 1,000$	M	$N = 2$
C	$N = 12$	I	$N = 1,500$	N	$N = 3$
D	$N = 14$	J	$N = 2,000$	O	$N = 4$
E	$N = 16$	K	$N = 2,500$	P	$N = 5$
F	$N = 18$			Q	$N = 6$

Table 6.8: Timing statistics for Pseudo Gauss-Seidel and SOR

Model	Matrix storage (KB)			Vector storage (KB)			Total storage (KB)		
	GS	Jac	PGS	GS	Jac	PGS	GS	Jac	PGS
A	186	201	273	48	72	48	234	273	321
B	1,110	657	569	240	360	240	1,350	1,017	809
C	6,192	794	648	1,152	1,728	1,152	7,344	2,522	1,800
D	32,928	804	1,384	5,376	8,064	5,376	38,304	8,868	6,760
E	168,960	815	1,215	24,576	36,864	24,577	193,536	37,679	25,792
F	-	829	1,524	-	165,888	110,596	-	166,717	112,120
G	441	512	673	645	860	645	1,086	1,372	1,318
H	883	726	761	1,290	1,720	1,289	2,172	2,446	2,050
I	1,324	1,055	745	1,935	2,580	1,934	3,258	3,636	2,679
J	1,766	1,011	795	2,577	3,436	2,578	4,344	4,447	3,373
K	2,207	1,006	723	3,222	4,296	3,223	5,429	5,302	3,946
	SOR	JOR	PSOR	SOR	JOR	PSOR	SOR	JOR	PSOR
L	8	20	147	2	3	3	10	23	150
M	348	387	850	72	108	72	4,19	495	922
N	5,455	866	607	912	1,368	914	6,372	2,234	1,521
O	48,414	858	738	7,102	10,653	7,138	55,515	11,511	7,876
P	296,588	671	912	39,788	59,682	39,889	336,376	60,353	40,801
Q	-	944	1373	-	263,937	176,198	-	264,881	177,570

Key					
Polling system		BRP		Kanban system	
A	$N = 8$	G	$N = 500$	L	$N = 1$
B	$N = 10$	H	$N = 1,000$	M	$N = 2$
C	$N = 12$	I	$N = 1,500$	N	$N = 3$
D	$N = 14$	J	$N = 2,000$	O	$N = 4$
E	$N = 16$	K	$N = 2,500$	P	$N = 5$
F	$N = 18$			Q	$N = 6$

Table 6.9: Memory requirements for Pseudo Gauss-Seidel and SOR

Table 6.9 gives the total memory required for each of the three implementations just described, and shows how this is broken down into storage for the matrix and vectors. We achieve an impressive reduction in memory for the vectors and, thanks to only an, at worst, minor increase for the matrix, a similar reduction for the total memory. Since the storage requirements for the sparse matrix implementation are still dominated by the explicit representation of the matrix, it is again easily outperformed by the hybrid approach in this respect.

6.7 Comparison with Related Work

As we pointed out in Chapter 2, the basic idea behind our hybrid technique for probabilistic model checking is similar to that of Kronecker-based approaches for CTMC analysis. They both use a compact, structured representation of the model and explicit storage for vectors to perform numerical computation using iterative methods. Of particular interest is the implementation of Ciardo and Miner [CM99] which uses decision diagram data structures. Now, having now presented our technique in detail, we give a more in-depth comparison of the two approaches.

To recap, the idea of Kronecker-based techniques is that the transition matrix of a CTMC is defined as a Kronecker (tensor) algebraic expression of smaller matrices, corresponding to components of the overall model. It is only necessary to store these small matrices and the structure of the Kronecker expression; iterative solution methods can be applied directly to this representation. Extracting or accessing a single element of the matrix requires several multiplications and a summation. Hence, as with our approach, ingenious techniques must be developed to minimise the resulting time overhead during numerical solution. Typically, these disadvantages are outweighed by the substantial savings in memory and corresponding increase in size of solvable model.

It turns out that the two approaches share a number of issues. A good example is the need to differentiate between reachable and unreachable states. Early Kronecker approaches worked over the entire set of possible states. This increases the size of the vectors which need to be stored and means that additional work is required to detect entries in the transition matrix corresponding to unreachable states. Improvements to the Kronecker approach, such as Kemper's use of binary search over ordered sets [Kem96] and Ciardo and Miner's use of multi-level data structures [CM97, CM99] to store the state space, relieved these problems to a certain extent. Our approach is comparable to that of [CM99] in that both use decision diagrams to compute and store the set of reachable states, and use offsets to determine state indices. The difference is that [CM99] uses *multi-valued* decision diagrams (MDDs) and we use *binary* decision diagrams (BDDs).

Another common issue is the selection of an iterative method for numerical solution. Early Kronecker approaches used the Power or Jacobi methods. More recent work such as [BCDK97, CM99] has given algorithms for the Gauss-Seidel method, a more attractive option since it usually requires considerably fewer iterations to converge and needs less memory. The drawback is that, to implement the method, each row or column of the matrix must be extracted individually, a process not ideally suited to techniques relying on structured storage of the matrix.

An alternative, and the one which is most directly related to our approach, is the 'interleaving' idea of [BCDK97]. Here, all matrix entries are accessed in a single pass of the data structure, comparable with the depth-first traversal of MTBDDs we adopt. The advantage is that many of the multiplication operations performed to compute the matrix entries can be reused when carried out in this order. The sacrifice made to obtain this saving is that the technique is restricted to the Power and Jacobi methods.

The Kronecker implementation of Ciardo and Miner [CM99, Min00] has further similarities with our work. They have developed a data structure called a *matrix diagram* which stores the Kronecker representation for a CTMC in a tree-like data structure. Like an MTBDD, the tree is kept in reduced form to minimise storage requirements. Furthermore, Ciardo and Miner use a combination of MDDs for reachability and state space storage, and matrix diagrams for matrix storage and numerical solution. This is analogous with our use of BDDs and MTBDDs, respectively.

Despite the numerous similarities, there remain fundamental differences between our hybrid approach and the various Kronecker-based techniques. Firstly, the amount of work required to extract a single matrix entry differs significantly. For an offset-labelled MTBDD, this constitutes tracing a path from its root node to a terminal, reading and following one pointer at each level. The Kronecker representation is also a multi-level system, but extracting an entry is slower, requiring a floating point multiplication to be performed at each level. In some cases, several such products must be computed and then summed to determine the final value.

Secondly, our MTBDDs are based on binary decisions whereas data structures for storing Kronecker representations, such as matrix diagrams, are based on multi-valued decisions. From an alternative viewpoint, the former encodes state spaces using Boolean variables and the latter does so using finite, integer-valued variables. The relative merits of each are hard to judge. Multi-valued variables might been as a more intuitive way to structure a given model's state space. On the other hand, we have found the flexibility of Boolean variables useful for developing efficient storage schemes for MDPs. Kronecker methods have only been used to store models which can be represented as a square, two-dimensional matrix, i.e. CTMCs and DTMCs.

With all of the above factors in mind, we conclude this section by presenting a comparison of the space and time efficiency of the two approaches. We begin by considering the amount of memory required for matrix storage. It seems that the Kronecker-based representations are more compact than MTBDDs in this respect. As an example, we use the Kanban system case study of Ciardo and Tilgner [CT96], used both in this thesis and in numerous other related sources in the literature. According to the statistics in [CM99], for example, we find that our offset-labelled MTBDD representation requires several times more memory than matrix diagrams for this model. Fortunately, this comparison is largely irrelevant since the space requirements of both approaches are dominated almost entirely by storage for vectors, not matrices.

From a time perspective, an exact comparison is more problematic. Even given the facility to run contrasting implementations on the same hardware, it is hard to make a fair evaluation without a detailed knowledge of their workings. Generally, matrix diagrams and sophisticated Kronecker implementations both claim, like us, to be comparable to sparse matrices in terms of speed. We are not aware, though, of an instance where Kronecker-based methods actually outperformed explicit techniques, as our hybrid approach did on the polling system case study. Given that an iteration of numerical solution essentially reduces to extracting matrix entries from the structured matrix representation, our observations above would suggest that offset-labelled MTBDDs should be faster than Kronecker-based data structures in this respect.

In fairness, though, Kronecker-based implementations, such as matrix diagrams, are often tailored towards performing the Gauss-Seidel method. While this usually entails more work per iteration, the total number of iterations can often be significantly reduced. The number of vectors which must be stored is also reduced by one.

We, on the other hand, have opted to focus on developing fast implementations of iterative methods which can be implemented using matrix-vector multiplication such as the Jacobi method. We have taken steps to address some of the limitations of Jacobi by investigating Pseudo Gauss Seidel, which exhibits numerous advantages of conventional Gauss-Seidel, only to a lesser extent.

More importantly, in this thesis we have concentrated on a wider range of analysis methods for probabilistic models. While steady-state probability computation requires the solution of a linear equation system, amenable to Gauss-Seidel, many other problems reduce to alternative iterative methods. Such problems include computing transient probabilities for CTMCs, model checking CSL time-bounded until properties for CTMCs and model checking PCTL properties for MDPs. In these cases, our approach is at a distinct advantage.

Chapter 7

Conclusions

7.1 Summary and Evaluation

The aim of this work was to develop an *efficient* probabilistic model checker. We set out to investigate whether BDD-based, symbolic model checking techniques, so successful in the non-probabilistic setting, could be extended for this purpose. The approach we have taken is to use MTBDDs, a natural extension of BDDs.

In terms of efficiency, we are concerned with minimising both the time and space requirements of model checking. When working with BDD-based data structures, complexity analysis is generally unhelpful; despite often having exponential worst-case complexity, it is well known that on realistic examples exhibiting structure, symbolic techniques can dramatically outperform other alternatives. For this reason, we have opted to rely on empirical results to gauge the efficiency of our techniques. By applying our work to a wide range of case studies, we aim to make such comparisons as fair as possible.

One of the main motivations for the work in this thesis was the lack of existing implementations of probabilistic model checking. Consequently, there is limited scope for making comparisons of our work with other tools. Instead we have chosen to implement an alternative version of our model checker based on more conventional, explicit data structures and use this to judge the efficiency of our techniques. This allows for fair comparisons, ensuring that tests can be carried out on identical examples and solution methods, and under the same conditions. Since probabilistic model checking requires computations on large matrices with relatively few non-zero entries, the obvious candidate for an explicit representation is sparse matrices. Fortunately, it is relatively simple to produce an efficient implementation of this data structure.

As demonstrated in the preceding chapters, we have successfully applied MTBDDs to the process of probabilistic model checking for two temporal logics, PCTL and CSL,

and for three types of model, DTMCs, CTMCs and MDPs. We found that there was a significant amount of commonality between the various cases.

In Chapter 4, we showed that, by applying heuristics for encoding and variable ordering, MTBDDs could be used to quickly construct a very compact representation of extremely large probabilistic models from their high-level description in the PRISM language. The heuristics for MDPs are the first to be presented in the literature. Furthermore, because of the close relationship between MTBDDs and BDDs, we were able to perform reachability and model checking of qualitative temporal logic properties efficiently on models with more than 10^{13} states.

The real focus of the thesis, however, has been on model checking of quantitative properties, for which numerical computation is required. In Chapter 5, we demonstrated that, for some case studies, this could be performed very efficiently with MTBDDs. On a desktop workstation of relatively modest specification, we were able to analyse models with as many as 7.5 billion states. The best results proved to be for MDP models, typically of randomised, distributed algorithms. Clearly, models of this size could not be handled explicitly under the same conditions.

We also found, in concurrence with existing work on symbolic analysis of probabilistic models, that MTBDDs were often inefficient for numerical computation because they provide a poor representation of solution vectors, which exhibit no structure and contain many distinct values. In Chapter 6, we presented a novel hybrid approach to combat this, combining our symbolic, MTBDD-based approach and the explicit version. Initially, we implemented this hybrid approach for the model checking of DTMCs and CTMCs, relying on the Jacobi and JOR methods for solving linear equation systems. Thanks to memory savings from the compactness of our MTBDD representation, we were able to analyse models approximately an order of magnitude larger than with explicit techniques. Typically, we also maintained a comparable solution speed.

We then showed how this hybrid approach could be extended for model checking of MDPs. Although the results were not as impressive as for DTMCs in terms of solution speed, we still required less memory than explicit approaches. As a second extension, we modified our hybrid approach to allow numerical solution of linear equation systems using a modified version of Gauss-Seidel called Pseudo Gauss-Seidel. We succeeded both in reducing the number of iterations for convergence and in producing a significant improvement in terms of the amount of memory required for vector storage. We also managed to reduce the average iteration time, in one case actually outperforming sparse matrices.

We concluded Chapter 6 by presenting a comparison of our hybrid approach and Kronecker-based techniques for CTMC analysis, in particular the matrix diagram data structure of Ciardo and Miner. Although the origins of these areas of work are different,

the two approaches have a lot in common. Generally, the performance of the two is quite similar; both can handle state spaces an order of magnitude larger than explicit approaches, while maintaining comparable solution speed. One of the main differences is that Kronecker techniques are amenable to the more efficient Gauss-Seidel method, whereas ours are not. We have gone some way towards redressing the balance with the implementation of Pseudo Gauss-Seidel. More importantly, though, our approach applies to a wider range of probabilistic models and solution methods, several of which, such as transient analysis of CTMCs and model checking of MDPs, make no use of Gauss-Seidel.

The potential value of our implementation is illustrated by the response we have received from the release of our model checker PRISM. To date, the tool has been downloaded by more than 350 people and we have received a pleasing amount of positive feedback from those who have managed to use PRISM to analyse interesting case studies. These include applications of the tool to probabilistic anonymity protocols [Shm02], and power management strategies [NPK+02]. A promising collaboration [DKN02] with the KRONOS model checker [DOTY96] has also been established.

7.2 Future Work

There are many possible areas in which the work in this thesis could be developed further. These can be divided into two broad classes. Firstly, there are a number of ways in which the functionality of our model checker PRISM could be improved. It would be desirable, for instance, to extend the range of modelling and specification formalisms which are supported. One example is the temporal logics LTL or PCTL*, which would allow more expressive specifications of DTMCs and MDPs than is possible with PCTL.

We would also like to augment our probabilistic models with information about costs and rewards, and support model checking of these using appropriately extended specification formalisms. The work in [BHHK00b, HCH+02], which presents the temporal logics CRL and CSRL for CTMC analysis of this nature, and the work of de Alfaro [dA97] for MDPs, would provide a good starting point. On a similar theme, we are interested in examining real-time extensions to our existing probabilistic models, for example by adding timed automata style clocks to the PRISM language. Initial work in this direction can be found in [DKN02].

Another improvement would be to consider the possibility of providing counterexamples for model checking. This has always been one of the most attractive features of non-probabilistic model checkers, but represents a non-trivial extension in the probabilistic setting since a single path of a model cannot be used to illustrate whether or not a temporal logic formula is satisfied. A related feature is the provision of information

about adversaries for MDPs (also known as policies or schedulers). Currently, our model checker computes the minimum or maximum probability, over a set of adversaries, that some behaviour is observed but does not generate the actual adversary which produces this behaviour. Such information would constitute useful feedback to the user of a model checker.

Lastly, it would certainly be beneficial to extend our probabilistic model checking with techniques for abstraction, such as data-type reduction and assume-guarantee reasoning. These have met with success in the non-probabilistic setting but would represent a significant challenge to emulate in the probabilistic case.

From our point of view, it would be interesting to investigate whether the improvements to the functionality of PRISM described above could be efficiently integrated into the symbolic implementation we have already developed. The fact that the techniques proposed here have already proved to be applicable to a wide range of model checking problems is an encouraging sign in this respect.

A second class of potential areas for future work are those which focus on improving the efficiency of our existing model checking implementation. For example, in this thesis, we have highlighted a number of cases where MTBDDs alone perform well, but have generally focused on techniques to improve efficiency where this is not the case. It would be interesting to investigate further the limits of the MTBDD approach where it does perform well and examine potential areas for improvement such as dynamic variable reordering and cache management. There are also aspects of our hybrid approach which warrant additional development, such as the version for MDP model checking.

As observed at the end of Chapter 6, it would also be interesting to explore in more depth the relationship between our work and Kronecker-based approaches, in particular the matrix diagram data structure of Ciardo and Miner. One example is the choice between binary and multi-valued decisions in data structures. There is some evidence that operations traditionally implemented in BDDs such as reachability and CTL model checking can be performed more efficiently with MDDs, as used in some Kronecker-based approaches. It might be beneficial to try modifying our data structures to use multi-valued decision instead of binary ones. Conversely, some of the optimisations we have developed for our hybrid approach in Chapter 6 might be applicable to Kronecker or matrix diagram implementations.

One of the main factors which limits the applicability of both our hybrid approach and the Kronecker-based techniques is the memory requirements for explicit vector storage. Any method of alleviating this problem would be extremely welcome. Two directions for research are potentially promising: firstly, so called ‘out-of-core’ methods where memory usage is reduced by storing data on disk, retrieving and updating it as required; secondly,

parallel or distributed implementations, where storage and, ideally, memory requirements can be lessened by spreading work over several processors or computers. Preliminary work in the first of these areas can be found in [KMNP02]. Since both these techniques usually require the matrix to be split up in some way, there is scope for reusing the techniques developed for this purpose in the Pseudo-Gauss Seidel implementation of Section 6.6.

7.3 Conclusion

In conclusion, we have successfully demonstrated the feasibility of MTBDD-based probabilistic model checking on a broad range of case studies, including examples of real-world randomised protocols. We have been able to perform analysis of extremely large, structured models, some of which are clearly out of the reach of explicit approaches.

This illustrates that established symbolic model checking techniques *can* be extended to the probabilistic case. In keeping with results from this field, we confirm that the most important heuristic for symbolic approaches is to maximise the exploitation of structure and regularity which derives from high-level model descriptions.

We also draw similar conclusions to other work on structured methods for probabilistic model analysis such as Kronecker-based techniques; namely that, while compact model storage can increase by approximately an order of magnitude the size of model which can be handled, the need to store one or more explicit solution vectors of size proportional to the model being analysed is often the limiting factor. This suggests that an important focus for future work should be the development of techniques and methodologies for abstraction, which concentrate on reducing the size of model which must be analysed, rather than on finding compact storage for large models.

Appendix A

The PRISM Model Checker

This appendix gives a brief overview of PRISM (**Pr**obabilist**i**c **S**ymbolic **M**odel Checker), the tool which has been developed to implement the techniques described in this thesis. PRISM supports model checking of PCTL over discrete-time Markov chains (DTMCs) and Markov decision processes (MDPs) and of CSL over continuous-time Markov chains (CTMCs). The algorithms used are those described in Section 3.3.

The tool takes two inputs: a model description and a properties specification. The former is written in the PRISM language, the syntax and semantics of which are given in Appendix B. PRISM parses this description, constructs a model of the appropriate type, either a DTMC, an MDP or a CTMC, and then determines the set of reachable states of this model. The properties specification consists of a number of temporal logic formulas in either PCTL or CSL, depending on the model type. These are parsed and then model checked. The architecture of the PRISM tool is summarised in Figure A.1.

The basic underlying data structures of PRISM are BDDs and MTBDDs. These are used for the construction and storage of all models, as described in Chapter 4 and Appendix C, for reachability, and for the precomputation phases of model checking. For the numerical computation aspect of model checking, PRISM provides three distinct engines. The first is a pure MTBDD implementation, as described in Chapter 5 and Appendix D; the second is explicit, based on sparse matrices; and the third uses the hybrid approach described in Chapter 6.

There are two versions of PRISM available, one based on a graphical user interface and one on a command line interface. Screenshots of each are shown in Figures A.2 and A.3, respectively. Binary and source-code versions of the tool can be obtained from the PRISM web site at www.cs.bham.ac.uk/~dxp/prism. The site also includes tool documentation, a number of relevant papers and a large amount of information about more than twenty case studies which have been developed with PRISM.

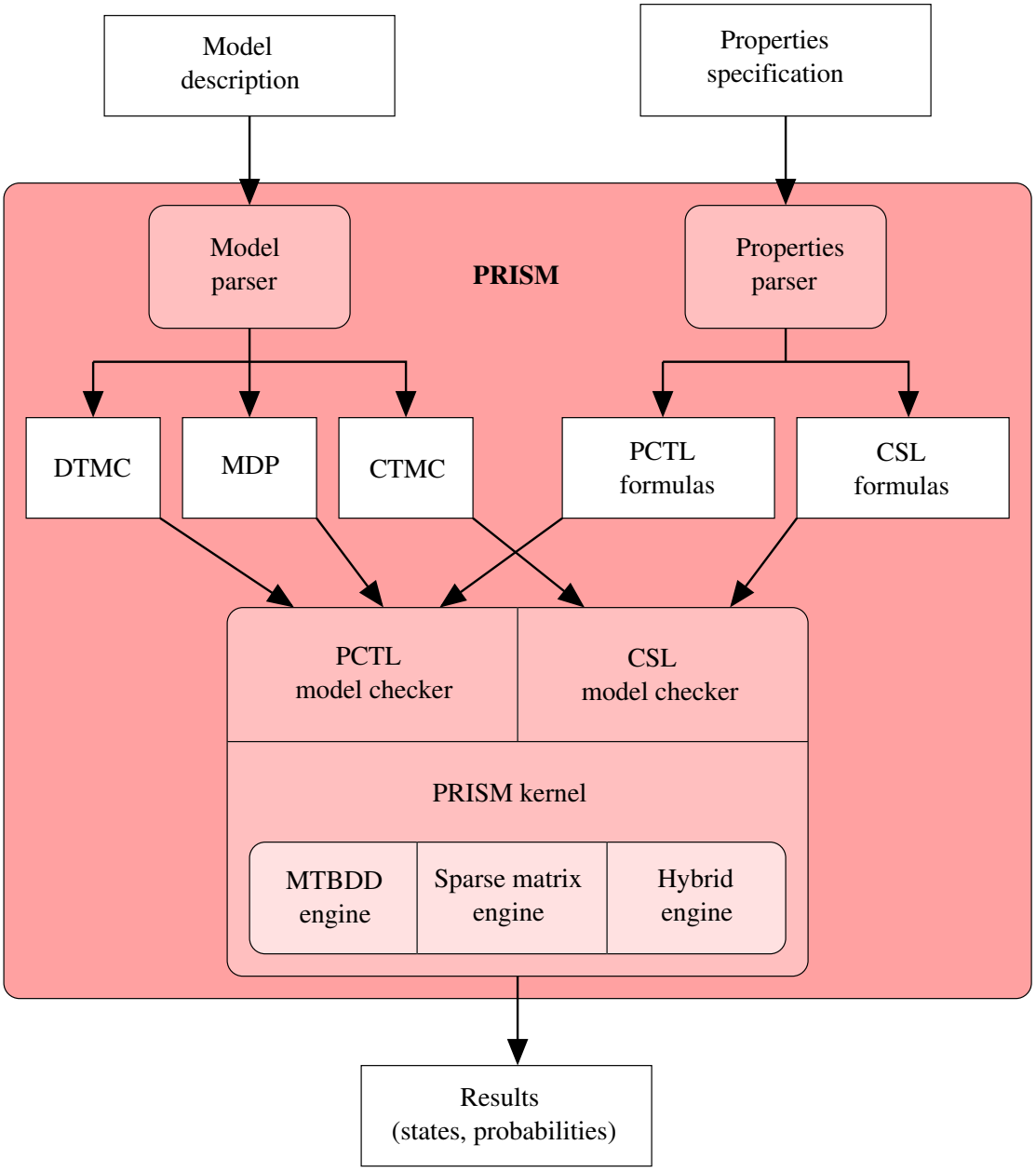


Figure A.1: The architecture of the PRISM model checker

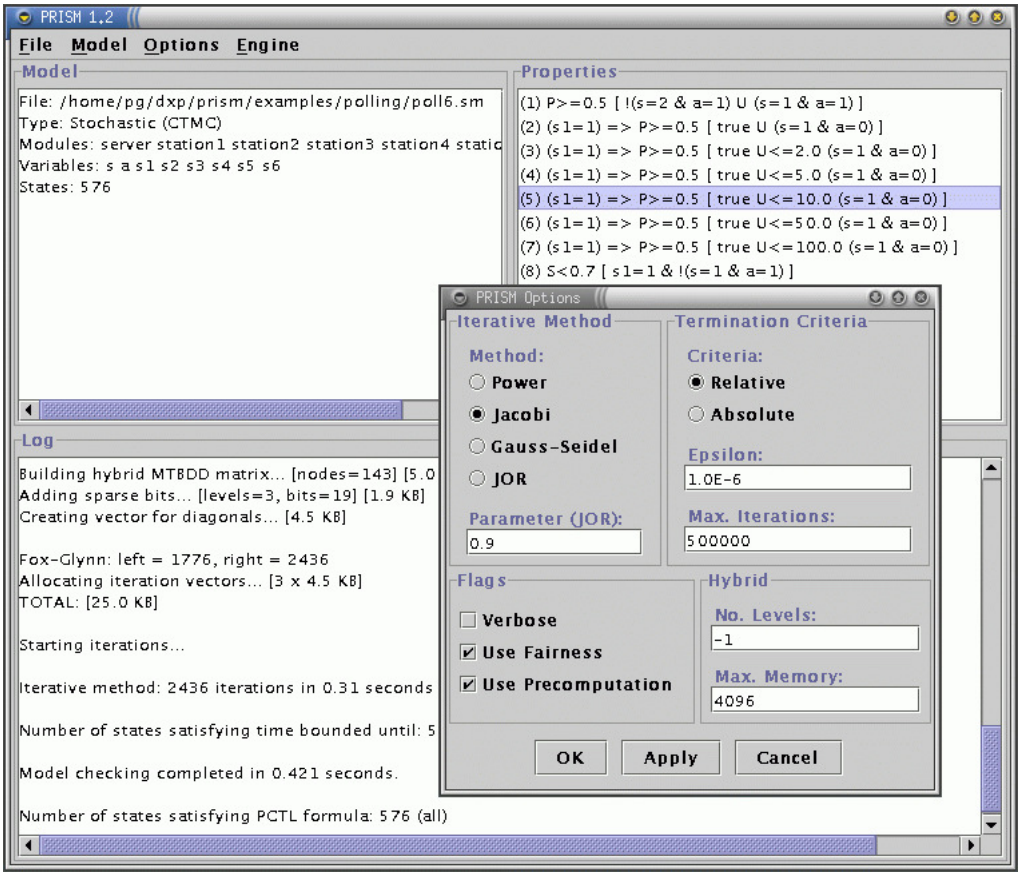


Figure A.2: Screenshot of the PRISM graphical user interface

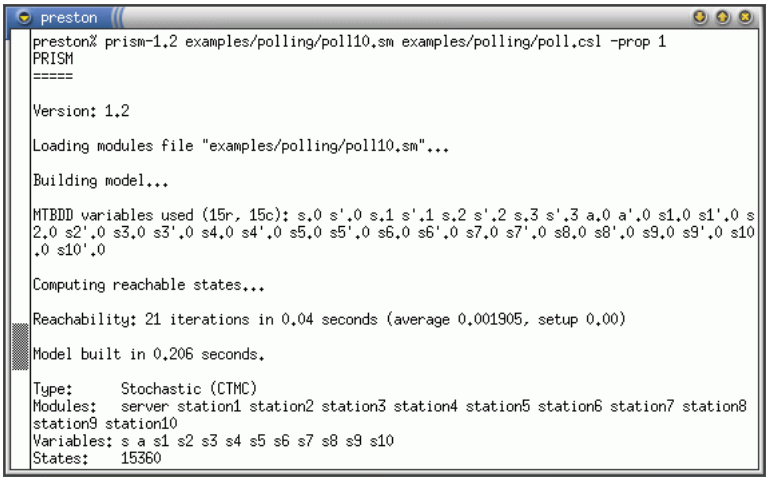


Figure A.3: Screenshot of the PRISM command line interface

Appendix B

The PRISM Language

In Section 3.4, we gave an informal presentation of the PRISM language and some simple examples of models described in it. This appendix provides a formal definition. Section B.1 defines the syntax of the language and Sections B.2, B.3 and B.4 give its semantics when describing DTMCs, CTMCs and MDPs, respectively. In these sections, we consider the restricted scenario where neither global variables nor synchronisation between modules are permitted. In Sections B.5 and B.6, we show how the basic syntax and semantics can be modified to handle these extensions. Finally, in Section B.7, we consider the issue of reachability.

B.1 Syntax

In this section, we give the syntax of a model described in the PRISM language. A model is defined as a set of m modules $M_1, \ldots, M_m$. Each module M_i is a pair (Var_i, C_i) where Var_i is a set of integer-valued, local variables with finite range and C_i is a set of commands. The variable sets Var_i define the local state space of each module which in turn define the global state space of the whole model. We denote by Var the set of all local variables in the model, i.e. $Var = \bigcup_{i=1}^{m} Var_i$. Furthermore, we suppose that each variable $v \in Var$ has an initial value $\overline{v}$.

The behaviour of module M_i is defined by the set of commands C_i. Each command $c \in C_i$ takes the form $(g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$, comprising a guard g and a set of pairs (λ_j, u_j) where $\lambda_j \in \mathbb{R}_{>0}$ and u_j is an update for each $1 \leq j \leq n_c$. A guard g is a predicate over the set of all local variables Var. Each update u_j of a command in C_i corresponds to a possible transition of module M_i. This is described in terms of the effect the transition would have on the variables in Var_i. If Var_i contains n_i local variables $v_1, \ldots, v_{n_i}$, then an update takes the form $(v_1' = expr_1) \wedge \cdots \wedge (v_{n_i}' = expr_{n_i})$ where each $expr_j$ is an expression

in terms of the variables in Var. We use v' to denote the new value of a variable v. Note that, in practice, an update may leave the values of some variables in Var_i unchanged. We allow this information to be omitted from the description of the update.

The constants λ_j in a command are used to assign probabilities or rates to each of its updates. These values will determine the probability or rate attached to transitions in the model which correspond to these updates. For a description of a DTMC or an MDP, we require that $\lambda_j \in (0, 1]$ for $1 \leq j \leq n_c$ and that $\sum_{i=1}^{n_c} \lambda_j = 1$. In the case of a CTMC, each λ_j can take any value in $\mathbb{R}_{>0}$.

Example

To illustrate the above ideas more clearly, we relate them to a simple example:

dtmc

module M_1
v : [1..6] **init** 1;
[] $(v = 1)$ $\rightarrow$ $1 : (v' = 2)$;
[] $(v > 1) \wedge (v < 6)$ $\rightarrow$ $0.5 : (v' = v - 1) + 0.5 : (v' = v + 1)$;
[] $(v = 6)$ $\rightarrow$ $1 : (v' = 5)$;
endmodule

This model defines a DTMC and consists of a single module M_1. The description of the module is split into two parts, giving its local variables and commands, respectively. From the first line, we see that M_1 has a single local variable v with range $\{1, \ldots, 6\}$ and initial value 1. Hence $Var = Var_1 = \{v\}$ and $\overline{v} = 1$.

The other three lines describe the module's set of commands C_1. In our notation, each command $(g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$ is written "[] $g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_{n_c} : u_{n_c}$;". For example, the second command of the three has the guard "$(v > 1) \wedge (v < 6)$" and two updates, "$(v' = v - 1)$" and "$(v' = v + 1)$", each assigned probability 0.5.

B.2 DTMC Semantics

We now give the semantics of the PRISM language, which are defined in terms of either DTMCs, CTMCs or MDPs, depending on which is being described. We first demonstrate the case for DTMCs and then show what modifications are required for the other types of model.

A DTMC is described by a state space, an initial state, and a probability transition matrix $\mathbf{P}$, as described in Section 3.1.1. We define the local state space S_i of module M_i to be the set of all valuations of the variables in Var_i. The global state space S, i.e. the

state space of the DTMC, is the product of the m local state spaces, i.e. $S = S_1 \times \cdots \times S_m$. We can express a global state $s \in S$ as a tuple of local states $(s_1, \ldots, s_m)$, where $s_i \in S_i$ for $1 \leq i \leq m$. The initial state of module M_i, denoted $\overline{s}_i$, is determined by the initial values of the variables in Var_i. The initial state of the DTMC is $\overline{s} = (\overline{s}_1, \ldots, \overline{s}_m)$.

To determine $\mathbf{P}$, we first have to consider the behaviour of each individual module. To determine the behaviour of module M_i, we must examine each of its commands. Consider a command $c \in C_i$ of module M_i where $c = (g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$. Since the guard g is a predicate over the variables in Var and each state of the DTMC is a valuation of these variables, g defines a subset of the global state space S. We denote this set of states $S_c = \{s \in S \mid s \models g\}$. The command c describes the behaviour of module M_i when the model is in a state $s \in S_c$. For a DTMC, we require that the guards of the commands in a module do not overlap, i.e. that the sets S_c for $c \in C_i$ are pairwise disjoint.

Each update u_j of c corresponds to a transition that M_i can make when the model is in a state $s \in S_c$. The transition is defined by giving the new value of each variable in Var_i as an expression in terms of the variables in Var. Hence, we can think of u_j as a function from S_c to S_i. If $Var_i = \{v_1, \ldots, v_{n_i}\}$ and u_j is $(v'_1 = expr_1) \wedge \cdots \wedge (v'_{n_i} = expr_{n_i})$, then for each state $s \in S_c$:

$$u_j(s) = (expr_1(s), \ldots, expr_{n_i}(s))$$

Using the value λ_j associated with each update u_j, the command c defines, for each $s \in S_c$, a function $\mu_{c,s} : S_i \to \mathbb{R}_{\geq 0}$ where for each $t_i \in S_i$:

$$\mu_{c,s}(t_i) = \sum_{j \in \{j \,|\, u_j(s) = t_i\}} \lambda_j$$

Note that, by the restrictions placed on the constants λ_j in the semantics, for a DTMC $\mu_{c,s}$ defines a probability distribution over S_i. Intuitively, this gives the probability of module M_i moving to each local state in S_i when the global state is s. To determine the behaviour of M_i in every global state, we have to combine the information for all commands in C_i. We denote this by a function $\mathbf{P}_{i,ind} : S \times S_i \to [0, 1]$ where for each $s \in S$ and $t_i \in S_i$:

$$\mathbf{P}_{i,ind}(s, t_i) = \begin{cases} \mu_{c,s}(t_i) & \text{if } s \in S_c \text{ for some } c \in C_i \\ 0 & \text{otherwise.} \end{cases}$$

Since we are presently ignoring the issue of synchronisation between modules, each transition of the whole model corresponds to a single module being scheduled and making an independent transition, the others remaining in the same state. We define the effect that

module M_i's transitions have on the whole model using the function $\mathbf{P}_i : S \times S \to [0,1]$. For states $s = (s_1, \ldots, s_m)$ and $t = (t_1, \ldots, t_m)$ in S:

$$\mathbf{P}_i(s,t) = \begin{cases} \mathbf{P}_{i,ind}(s,t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

Finally, we define the probability transition matrix $\mathbf{P}$ of the overall model. In each global state, some subset of the m modules can independently make transitions. We assume a uniform probability of each one being scheduled. Hence, we define $\mathbf{P} : S \times S \to [0,1]$ as:

$$\mathbf{P}(s,t) = 1/n_s \left(\sum_{i=1}^{m} \mathbf{P}_i(s,t) \right)$$

where n_s is the number of modules which can make a transition in state s, i.e. $n_s = |\{M_i \mid s \in S_c \text{ for some } c \in C_i\}|$. Since, by definition, the probabilities for each module sum to 1, this can also be computed as $n_s = \sum_{i=1}^{m} \sum_{t \in S} \mathbf{P}_i(s,t)$, i.e. by summing the values in each row of the summation of the $\mathbf{P}_i$ matrices above.

Example

We now show how these semantics for DTMCs apply to the simple example from the previous section. We saw that $Var = Var_1 = \{v\}$. Hence, $S = S_1 = \{1, \ldots, 6\}$ and $\overline{s} = \overline{s}_1 = 1$. Module M_1 has three commands which we label c_1, c_2 and c_3. Then, $S_{c_1} = \{1\}$, $S_{c_2} = \{2, \ldots, 5\}$ and $S_{c_3} = \{6\}$. We take the command c_2 as an example. Its two updates, which we denote u_1 and u_2, define functions from $\{2, \ldots, 5\}$ to $\{1, \ldots, 6\}$. For example, $u_1(2) = 1$, $u_1(3) = 2$, $u_1(4) = 3$ and $u_1(5) = 4$. The command c_2 defines four different probability distributions over $\{1, \ldots, 6\}$: $\mu_{c_2,2}$, $\mu_{c_2,3}$, $\mu_{c_2,4}$ and $\mu_{c_2,5}$. For example, $\mu_{c_2,2}$ selects 1 and 3 with equal probability 0.5.

B.3 CTMC Semantics

For a CTMC, the semantics are almost identical to DTMCs. The local state space S_i of module M_i, global state space S and initial state $\overline{s}$ are as before. For a command $c \in C_i$, the set S_c and functions $\mu_{c,s} : S_i \to \mathbb{R}_{\geq 0}$ are also defined in identical fashion. Note that, in this case, $\mu_{c,s}$ does not define a probability distribution over S_i, but a mapping from S_i to non-negative reals since, as described in Section 3.1.3, the transitions in a CTMC are labelled with rates rather than probabilities. Also, we allow the sets S_c for commands in a module to overlap, i.e. a state $s \in S$ can be contained in S_c for several commands $c \in C_i$. It is often convenient to define the transitions for one state across several commands in

this way. This is possible for a CTMC because there is no need to check that transitions in each state form a probability distribution.

The transition rate matrix $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$ for the CTMC is constructed in a similar fashion to the matrix $\mathbf{P}$ for DTMCs. The independent behaviour of module M_i is given by the function $\mathbf{R}_{i,ind} : S \times S_i \to \mathbb{R}_{\geq 0}$ where for each $s \in S$ and $t_i \in S_i$:

$$\mathbf{R}_{i,ind}(s, t_i) = \sum_{c \in \{c \in C_i \mid s \in S_c\}} \mu_{c,s}(t_i)$$

Note that the summation captures behaviour split over several commands. As for DTMCs, only one module makes a transition at a time. The effect on the whole model is defined by $\mathbf{R}_i : S \times S \to \mathbb{R}_{\geq 0}$, where for states $s = (s_1, \ldots, s_m)$ and $t = (t_1, \ldots, t_m)$ in S:

$$\mathbf{R}_i(s, t) = \begin{cases} \mathbf{R}_{i,ind}(s, t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

Finally, we combine the above to form the transition rate matrix $\mathbf{R} : S \times S \to \mathbb{R}_{\geq 0}$:

$$\mathbf{R}(s, t) = \sum_{i=1}^{m} \mathbf{R}_i(s, t)$$

Note that there is no need to modify the rates in any way, as we did with the probabilities when constructing a DTMC. As mentioned in Section 3.4.4, we model the concurrency arising from scheduling between modules using the race condition which exists in each state of a CTMC with multiple transitions.

B.4 MDP Semantics

With MDPs, we again take a similar approach. The local state space S_i of module M_i, global state space S and initial state $\overline{s}$ are as in the previous two sections. For a command $c \in C_i$, the set S_c and functions $\mu_{c,s} : S_i \to \mathbb{R}_{\geq 0}$ are also defined in the same way. As for DTMCs, each function $\mu_{c,s}$ is a probability distribution over S_i. Unlike DTMCs, though, we allow the guards of commands in a module to overlap. Hence, several probability distributions may be enabled in one state. This is interpreted as *local nondeterminism* within the module.

As described in Section 3.1.2, an MDP is defined by a function $Steps : S \to 2^{Dist(S)}$, mapping each state $s \in S$ to a finite, non-empty subset of $Dist(S)$, the set of all probability distributions over S (i.e. the set of all functions of the form $\mu : S \to [0, 1]$ where $\sum_{s \in S} \mu(s) = 1$). We begin by, for each module M_i, defining a function $Steps_{i,ind} : S \to 2^{Dist(S_i)}$ which associates each state $s \in S$ with a set of probability distributions over S_i:

$$Steps_{i,ind}(s) = \{\mu_{c,s} \mid c \in C_i \text{ and } s \in S_c\}$$

As before, we convert this to a function which gives probability distributions over the global state space, assuming that all other modules remain in the same state. We will denote this $Steps_i : S \to 2^{Dist(S)}$. Let $s = (s_1, \ldots, s_m)$ be a state in S. We define $Steps_i(s)$ as follows. For each $\mu_i \in Steps_{i,ind}(s)$, the set $Steps_i(s)$ contains the distribution $\mu \in Dist(S)$, where for any state $t = (t_1, \ldots, t_m)$ in S:

$$\mu(t) = \begin{cases} \mu_i(t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

Finally, we combine the above to determine $Steps : S \to 2^{Dist(S)}$ for the overall model. The scheduling between modules in a state is nondeterministic so we simply take the union of the set of all probability distributions for each module.

$$Steps(s) = \bigcup_{i=1}^{m} Steps_i(s)$$

B.5 Global Variables

The basic definition of the language can be extended with the inclusion of global variables. These are variables which can be both read and modified by any module. In terms of the syntax, we assume that the set of all variables Var now includes a set of global variables Var_{glob}. The initial value, $\overline{gv}$, of each global variable $gv \in Var_{glob}$ is also specified.

A guard g of a command c is still a predicate over all the variables in Var, but this now includes Var_{glob}. Similarly, each update in a command now also specifies how the global variables are modified. Hence, if $Var_i = \{v_1, \ldots, v_{n_i}\}$ and $Var_{glob} = \{gv_1, \ldots, gv_{n_{glob}}\}$, an update in a command $c \in C_i$ takes the form:

$$(v'_1 = expr_1) \wedge \cdots \wedge (v'_{n_i} = expr_{n_i}) \wedge (gv'_1 = gexpr_1) \wedge \cdots \wedge (gv'_{n_{glob}} = gexpr_{n_{glob}})$$

where each $expr_j$ and $gexpr_j$ is an expression in terms of the variables in Var. As before, the update may leave the values of some variables unchanged, in which case this information can be omitted from the update.

In terms of the semantics, we define S_{glob} to be the set of all valuations of the variables Var_{glob} and the initial valuation $\overline{s}_{glob} \in S_{glob}$ to be given by the value $\overline{gv}$ of each global variable gv. The global state space S and initial state $\overline{s}$ of the model are then $S_1 \times \cdots \times S_m \times S_{glob}$ and $(\overline{s}_1, \ldots, \overline{s}_m, \overline{s}_{glob})$, respectively.

For a command $c \in C_i$ of module M_i, the set S_c is defined as in the previous sections. If $c = (g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$, each update u_j now corresponds to a function $u_j : S_c \to S_i \times S_{glob}$, where for any state $s \in S_c$:

$$u_j(s) = (expr_1(s), \ldots, expr_{n_i}(s), gexpr_1(s), \ldots, gexpr_{n_{glob}}(s))$$

The function $\mu_{c,s} : S_i \times S_{glob} \to \mathbb{R}_{\geq 0}$ is then defined as follows. For states $s \in S_c$ and $t_{i,glob} \in S_i \times S_{glob}$:

$$\mu_{c,s}(t_{i,glob}) = \sum_{j \in \{j \,|\, u_j(s)=t_{i,glob}\}} \lambda_j$$

The above applies to DTMCs, CTMCs and MDPs. The remaining part of the semantics differs for each type of model but is easily obtained from the basic semantics. Hence, we demonstrate the case for DTMCs only.

For states $s \in S$ and $t_{i,glob} \in S_i \times S_{glob}$, the function $\mathbf{P}_{i,ind}$ is defined as:

$$\mathbf{P}_{i,ind}(s, t_{i,glob}) = \begin{cases} \mu_{c,s}(t_{i,glob}) & \text{if } s \in S_c \text{ for some } c \in C_i \\ 0 & \text{otherwise} \end{cases}$$

Since the parallel composition is asynchronous, only one module is scheduled at any one time. Hence, we can assume that this module assumes responsibility for the global variables. For states $s = (s_1, \ldots, s_m, s_{glob})$ and $t = (t_1, \ldots, t_m, t_{glob})$ in S, we define the function $\mathbf{P}_i : S \times S \to [0, 1]$ as:

$$\mathbf{P}_i(s, t) = \begin{cases} \mathbf{P}_{i,ind}(s, (t_i, t_{glob})) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

The final composition remains unchanged:

$$\mathbf{P}(s, t) = 1/n_s \left(\sum_{i=1}^{m} \mathbf{P}_i(s, t) \right)$$

where $n_s = |\{M_i \mid s \in S_c \text{ for some } c \in C_i\}|$.

B.6 Synchronisation

We now extend our definition of the PRISM language to include synchronisation. This allows modules to make transitions simultaneously, as well as independently. On the assumption that this would typically be used as an alternative modelling tool to global variables, we present synchronisation as an orthogonal extension to the basic definition of the PRISM language.

Adopting the mechanism used in many process algebras, we label local transitions of modules with actions from a fixed alphabet and require that transitions labelled with the same action occur simultaneously. Our definition is based on the standard parallel composition from the process algebra CSP [Hoa85, Ros98].

The changes to the syntax of the PRISM language are as follows. We assume a set of action labels Act. Each command of a module is either specified as being *independent* or is

labelled with an action $a \in Act$. We partition each command set C_i into the subsets $C_{i,ind}$ and $\{C_{i,a} \mid a \in Act\}$ where independent commands are placed in $C_{i,ind}$ and commands labelled with a in $C_{i,a}$. The set of all actions used by a module M_i, sometimes referred to as its 'alphabet', is denoted A_i, i.e. $A_i = \{a \in Act \mid C_{i,a} \neq \emptyset\}$.

Since each transition of a module corresponds to one of its commands, its transitions are now also defined as being either independent or labelled with an action $a \in Act$. Intuitively, in each global state of the composed model, either one of the n modules makes an independent transition or, for some $a \in Act$, the set of modules $\{M_i \mid a \in A_i\}$ all make a synchronous, a-labelled transition. In the latter case, the synchronous transition can only occur if *all* the modules in $\{M_i \mid a \in A_i\}$ are able to participate. We now define this process formally for each of the three types of model.

Discrete-Time Markov Chains (DTMCs)

The definitions of S_i, S, $\overline{s}$, S_c and $\mu_{c,s}$ remain the same as in the asynchronous case. For each module M_i, we then define two functions, $\mathbf{P}_{i,ind} : S \to S_i$ and $\mathbf{P}_{i,a} : S \to S_i$, which correspond to independent and a-labelled transitions, respectively. For states $s \in S$ and $t_i \in S_i$:

$$\mathbf{P}_{i,ind}(s,t_i) = \begin{cases} \mu_{c,s}(t_i) & \text{if } s \in S_c \text{ for some } c \in C_{i,ind} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{P}_{i,a}(s,t_i) = \begin{cases} \mu_{c,s}(t_i) & \text{if } s \in S_c \text{ for some } c \in C_{i,a} \\ 0 & \text{otherwise} \end{cases}$$

As before, we extend these definitions to describe the behaviour of the whole model. The function $\mathbf{P}_i : S \to S$ corresponds to the independent behaviour of module M_i and the function $\mathbf{P}_a : S \to S$ to the case where modules synchronise over action $a \in Act$. For states $s = (s_1, \ldots, s_m)$ and $t = (t_1, \ldots, t_m)$ in S:

$$\mathbf{P}_i(s,t) = \begin{cases} \mathbf{P}_{i,ind}(s,t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{P}_a(s,t) = \begin{cases} \prod_{i \in \{i \mid a \in A_i\}} \mathbf{P}_{i,a}(s,t_i) & \text{if } s_j = t_j \text{ when } a \notin A_j \\ 0 & \text{otherwise} \end{cases}$$

Note that, as would be expected, the probability of several modules making a transition is equal to the product of the component probabilities. Finally, we define the overall transition probability matrix $\mathbf{P} : S \to S$ as:

$$\mathbf{P}(s,t) = 1/n_s \left(\sum_{i=1}^{m} \mathbf{P}_i(s,t) + \sum_{a \in Act} \mathbf{P}_a(s,t) \right)$$

In this case, we assume that there is an equal probability of either an independent or an action-labelled transition occurring. Hence, n_s is equal to the number of modules which can make an independent transition in s plus the number of synchronous transitions which can occur. Again, we can compute n_s by adding all entries of the row corresponding to s in the summation above.

Continuous-Time Markov Chains (CTMCs)

As for DTMCs, the definitions of S_i, S, $\overline{s}$, S_c and $\mu_{c,s}$ remain the same as in the asynchronous case. For each module M_i, we define two functions, $\mathbf{R}_{i,ind} : S \to S_i$ and $\mathbf{R}_{i,a} : S \to S_i$, corresponding to independent and a-labelled transitions, respectively. For states $s \in S$ and $t_i \in S_i$:

$$\mathbf{R}_{i,ind}(s,t_i) = \sum_{c \in \{c \in C_{i,ind} \,|\, s \in S_c\}} \mu_{c,s}(t_i)$$

$$\mathbf{R}_{i,a}(s,t_i) = \sum_{c \in \{c \in C_{i,a} \,|\, s \in S_c\}} \mu_{c,s}(t_i)$$

These are combined, as for DTMCs, to produce the functions $\mathbf{R}_i : S \to S$ and $\mathbf{R}_a : S \to S$. For states $s = (s_1, \ldots, s_m)$ and $t = (t_1, \ldots, t_m)$ in S:

$$\mathbf{R}_i(s,t) = \begin{cases} \mathbf{R}_{i,ind}(s,t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{R}_a(s,t) = \begin{cases} \prod_{i \in \{i \,|\, a \in A_i\}} \mathbf{R}_{i,ind}(s,t_i) & \text{if } s_j = t_j \text{ when } a \notin A_j \\ 0 & \text{otherwise} \end{cases}$$

Note that the rate of a synchronous transition is obtained by multiplying the individual rates. A common approach taken when modelling the synchronisation of several transitions is to assume that one is *active* and has a non-negative rate, and the others are *passive* with rate equal to 1. Hence, under multiplication, the overall rate is simply equal to that of the active one. While there is some debate in the literature as to the best way of combining rates for synchronisation, we opt for this method because it is simple and allows a consistent approach to finding the probability or rate of a synchronous transition in all three types of model: DTMCs, CTMCs and MDPs. We can now define the overall transition rate matrix $\mathbf{R} : S \to S$. For states $s, t \in S$:

$$\mathbf{R}(s,t) = \sum_{i=1}^{m} \mathbf{R}_i(s,t) + \sum_{a \in Act} \mathbf{R}_a(s,t)$$

Markov Decision Processes (MDPs)

For MDPs, the definitions of S_i, S, $\overline{s}$, S_c and $\mu_{c,s}$ are the same as for DTMCs and CTMCs. Like in the previous two sections, we split the behaviour of the MDP into two cases, independent and synchronising, defining functions $Steps_{i,ind} : S \to 2^{Dist(S_i)}$ and $Steps_{i,a} : S \to 2^{Dist(S_i)}$. For states $s \in S$:

$$Steps_{i,ind}(s) = \{\mu_{c,s} \mid c \in C_{i,ind} \text{ and } s \in S_c\}$$

$$Steps_{i,a}(s) = \{\mu_{c,s} \mid c \in C_{i,a} \text{ and } s \in S_c\}$$

We then convert these to functions which give probability distributions over the global state space, $Steps_i : S \to 2^{Dist(S)}$ and $Steps_a : S \to 2^{Dist(S)}$. Let $s = (s_1, \ldots, s_m)$ be a state in S. We define $Steps_i(s)$ as follows. If $\mu_i \in Steps_{i,ind}(s)$, then $Steps_i(s)$ contains the distribution $\mu \in Dist(S)$, where for any state $t = (t_1, \ldots, t_m)$ in S:

$$\mu(t) = \begin{cases} \mu_i(t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

Again, letting $s = (s_1, \ldots, s_m)$ be a state in S, we define $Steps_a(s)$ as follows. If μ_i is a probability distribution in $Steps_{i,a}(s)$ for each i such that $1 \leq i \leq m$ and $a \in A_i$, then $Steps_a(s)$ contains the distribution $\mu \in Dist(S)$ where, for any state $t = (t_1, \ldots, t_m)$ in S:

$$\mu(t) = \begin{cases} \prod_{i \in \{i \mid a \in A_i\}} \mu_i(t_i) & \text{if } s_j = t_j \text{ when } a \notin A_j \\ 0 & \text{otherwise} \end{cases}$$

Finally, we construct the function $Steps : S \to 2^{Dist(S)}$ representing the MDP. For each state $s \in S$:

$$Steps(s) = \left(\bigcup_{i=1}^{m} Steps_i(s) \right) \bigcup \left(\bigcup_{a \in Act} Steps_a(s) \right)$$

B.7 Reachability

A description in the PRISM language, be it corresponding to a DTMC, a CTMC or an MDP, defines a set of states, one of which is the initial state, and the behaviour of the model in each of these states. We say that a state s' is *reachable* from another state s if there exists a finite path in the model starting in s and ending in s'. We define the set of reachable states of the model as the set of all states which are reachable from its initial state. Typically, we are only interested in this reachable portion of the model and hence all unreachable states can be removed. In addition, it is often useful to check for the presence of *deadlock* states. We define a deadlock state as one which is reachable but has no outgoing transitions.

Appendix C

MTBDD Model Construction

In Section 4.3, we discussed the process of constructing an MTBDD from a model description in the PRISM language and gave a simple example. In this appendix, we describe the method formally and in full. Our presentation is based on the syntax of the language given in the previous appendix. We then use the corresponding semantics to show that the construction process is correct. As in Appendix B, we initially assume that neither global variables nor synchronisation are present.

C.1 Discrete-Time Markov Chains (DTMCs)

We start with the case for DTMCs. We assume an encoding $enc : S \to \mathbb{B}^n$ of the global state space into MTBDD variables, as discussed in Chapter 4. Furthermore, we assume that the local state space of each module M_i is encoded by $enc_i : S_i \to \mathbb{B}^{n_i}$ where $\sum_{i=1}^{m} n_i = n$ and such that $enc(s_1, \ldots, s_m) = (enc_1(s_1), \ldots, enc_m(s_m))$. We will construct the MTBDD P, which represents the DTMC's transition probability matrix $\mathbf{P}$, as described in Section 3.7.2. For this we will use row variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ and column variables $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$. We also assume that $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ are partitioned into subsets $\underline{\mathsf{x}}_1, \ldots, \underline{\mathsf{x}}_m$ and $\underline{\mathsf{y}}_1, \ldots, \underline{\mathsf{y}}_m$, respectively, corresponding to the encodings $enc_1, \ldots, enc_m$.

Following the semantics in Section B.2, we begin by considering a command $c \in C_i$ of the module M_i. Let $c = (g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$. The guard g is a predicate over PRISM variables, defining a set $S_c \subseteq S$. As described in Section 4.1.1, our encoding enc of S is based on a translation from PRISM variables to MTBDD variables. Hence, it is trivial to build a BDD g over variables $\underline{\mathsf{x}}$ representing the guard g, i.e. such that $f_{\mathsf{g}}[\underline{\mathsf{x}} = enc(s)]$ is equal to 1 if $s \in S_c$ and 0 otherwise for all $s \in S$. This process was illustrated in the simple example of Section 4.3.1. It is also identical to the construction of BDDs to represent atomic propositions from PCTL or CSL formulas, as shown previously

in Figure 5.2.

In a similar fashion, we can encode each update u_j, representing a function from S_c to S_i. Since $S_c \subseteq S$, we represent u_j as a BDD u_j over variables $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}_i$, i.e. $f_{\mathsf{u}_j}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc(t_i)]$ is equal to 1 if $u_j(s) = t_i$ and 0 otherwise for all $s \in S$ and $t_i \in S_i$. The construction of this BDD was also demonstrated in the example of Section 4.3.1. The MTBDD mu_c representing the command c is then obtained as follows:

$$\mathsf{mu}_c := \mathsf{g} \times \sum_{j=1}^{n_c} \textsc{Const}(\lambda_j) \times \mathsf{u}_j$$

Having constructed mu_c for each command $c \in C_i$ of a module M_i, we combine them to produce the MTBDD $\mathsf{P}_{i,ind}$:

$$\mathsf{P}_{i,ind} := \sum_{c \in C_i} \mathsf{mu}_c$$

Then, we build the MTBDD P_i, multiplying $\mathsf{P}_{i,ind}$ by identity matrices for the other $m-1$ modules. This encodes the fact that module M_i moves independently and so the other modules remain in the same state:

$$\mathsf{P}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{P}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

where id_j is the BDD $\textsc{Identity}(\underline{\mathsf{x}}_j, \underline{\mathsf{y}}_j)$. Finally, having constructed P_i for each of the m modules M_i, we compute the MTBDD P:

$$\mathsf{P} := \textsc{Unif}\left(\sum_{i=1}^{m} \mathsf{P}_i\right)$$

where $\textsc{Unif}(\mathsf{M}) = \mathsf{M} \div \textsc{Abstract}(+, \underline{\mathsf{y}}, \mathsf{M})$.

Proof of Correctness

We need to show that the MTBDD P, constructed above, represents the transition probability matrix $\mathbf{P}$ of the DTMC, as defined by the semantics in Section B.2. From above, we have that, for a command $c = (g, (\lambda_1, u_1), \ldots, (\lambda_{n_c}, u_{n_c}))$ of module M_i and for any $s \in S$, $t_i \in S_i$ and $t \in S$:

$$f_{\mathsf{g}}[\underline{\mathsf{x}} = enc(s)] = \begin{cases} 1 & \text{if } s \in S_c \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\mathsf{u}_j}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc_i(t_i)] = \begin{cases} 1 & \text{if } u_j(s) = t_i \\ 0 & \text{otherwise} \end{cases}$$

Hence, for any $s \in S$, $t_i \in S_i$ and $t \in S$, we have the following:

$$f_{\mathsf{mu}_c}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc_i(t_i)] = \begin{cases} \mu_{c,s}(t_i) & \text{if } s \in S_c \\ 0 & \text{otherwise} \end{cases}$$

by the definition of $\mu_{c,s}$. Then:

$$f_{\mathsf{P}_{i,ind}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc_i(t_i)] = \mathbf{P}_{i,ind}(s, t_i)$$

by the definition of $\mathbf{P}_{i,ind}$ and since s is contained in S_c for at most one command $c \in C_i$. This gives:

$$f_{\mathsf{P}_i}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t)] = \mathbf{P}_i(s, t)$$

by the definition of $\mathbf{P}_i$ and since $f_{\mathsf{id}_j}[\underline{\mathsf{x}}_j = enc_j(s_j), \underline{\mathsf{y}}_j = enc_j(t_j)]$ is equal to 1 if $s_j = t_j$ and 0 otherwise for all $s_j, t_j \in S_j$. Finally, as required:

$$f_{\mathsf{P}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t)] = \mathbf{P}(s, t)$$

by the definition of $\mathbf{P}$ and since, in the summation of matrices $\mathbf{P}_i$, each row corresponding to state s sums to n_s because it comprises n_s probability distributions.

C.2 Continuous-Time Markov Chains (CTMCs)

The process for CTMCs is very similar to the one for DTMCs. We use the same row variable sets $\underline{\mathsf{x}}, \underline{\mathsf{x}}_1, \ldots, \underline{\mathsf{x}}_m$, column variable sets $\underline{\mathsf{y}}, \underline{\mathsf{y}}_1, \ldots, \underline{\mathsf{y}}_m$ and state space encodings $enc, enc_1, \ldots, enc_m$ as in the previous section. Assuming also the same definition of the MTBDD mu_c, we construct for each module M_i:

$$\mathsf{R}_{i,ind} := \sum_{c \in C_i} \mathsf{mu}_c$$

$$\mathsf{R}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{R}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

where id_j is the BDD IDENTITY($\underline{\mathsf{x}}_j, \underline{\mathsf{y}}_j$). We then combine these to form the MTBDD R, representing the transition rate matrix $\mathbf{R}$:

$$\mathsf{R} := \sum_{i=1}^{m} \mathsf{R}_i$$

Proof of Correctness

The proof of correctness is also very similar. We need to show that the MTBDD R, constructed above, represents the transition rate matrix $\mathbf{R}$ of the CTMC, as defined by the semantics in Section B.3. Applying the same arguments as for DTMCs, we have that for any $s \in S$, $t_i \in S_i$ and $t \in S$:

$$f_{\mathsf{R}_{i,ind}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc_i(t_i)] = \mathbf{R}_{i,ind}(s, t_i)$$

$$f_{\mathsf{R}_i}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t)] = \mathbf{R}_i(s, t)$$

$$f_{\mathsf{R}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t)] = \mathbf{R}(s, t)$$

C.3 Markov Decision Processes (MDPs)

We now extend the process to MDPs. The main difference is the presence of nondeterminism, either from the scheduling between modules or from local nondeterministic choices within a module. We use the same row variable sets $\underline{\mathsf{x}}, \underline{\mathsf{x}}_1, \ldots, \underline{\mathsf{x}}_m$, column variable sets $\underline{\mathsf{y}}, \underline{\mathsf{y}}_1, \ldots, \underline{\mathsf{y}}_m$ and state space encodings $enc, enc_1, \ldots, enc_m$ as in the previous two sections. In addition, we use a set of k nondeterministic variables $\underline{\mathsf{z}}$. We assume that $\underline{\mathsf{z}}$ is partitioned into two sets, $\underline{\mathsf{z}}_s = (\mathsf{z}_{s,1}, \ldots, \mathsf{z}_{s,k_s})$ to encode nondeterminism arising from scheduling between modules, and $\underline{\mathsf{z}}_l = (\mathsf{z}_{l,1}, \ldots \mathsf{z}_{l,k_l})$ to encode local nondeterminism within a module, i.e. $k = k_s + k_l$. Following our definition from Section 4.2.1, we require that, for any state $s \in S$, the MTBDD Steps representing the MDP defined by *Steps* satisfies:

- if $\mu \in Steps(s)$, then there exists $b \in \mathbb{B}^k$ such that:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$
- for any $b \in \mathbb{B}^k$, one of the following two conditions holds:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t), \underline{\mathsf{z}} = b] = 0$ for all $t \in S$
  - there exists $\mu \in Steps(s)$ such that
    $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}} = enc(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$

As with DTMCs and CTMCs, we begin by constructing an MTBDD for each module M_i. For MDPs, we must take additional steps to handle local nondeterminism. Here, the guards within a module are allowed to overlap, i.e. a state $s \in S$ of the MDP can be in S_c for more than one command $c \in C_i$. Where this is the case, there will be several nondeterministic choices in s, one corresponding to each command.

At the MTBDD level, we encode this using the MTBDD variables $(\mathsf{z}_{l,1}, \ldots, \mathsf{z}_{l,k_l})$. We need to ensure that for a state s, each local nondeterministic choice available in s is encoded with a distinct element of $\mathbb{B}^{k_l}$. To do so, we partition M_i's set of commands C_i into a number, say n_l, of subsets, $C_{i,1}, \ldots, C_{i,n_l}$, such that the guards of the commands within each subset do not overlap. All of the nondeterministic choices corresponding to commands in a given subset $C_{i,j}$ can then be encoded with the same element. We assume that $b_{l,1}, \ldots, b_{l,n_l}$ are distinct elements of $\mathbb{B}^{k_l}$. For $1 \leq j \leq n_l$, subset $C_{i,j}$ will be encoded with $b_{l,j}$. Hence, we must ensure that k_l satisfies $2^{k_l} \geq n_l$. Note that n_l is bounded above by the number of commands $|C_i|$.

The construction proceeds as follows. Taking the same definition of the MTBDD mu_c for a command c as in the previous two sections, we build the MTBDD $\mathsf{Steps}_{i,ind}$ for each module M_i as follows:

$$\mathsf{Steps}_{i,ind} := \sum_{j=1}^{n_l} \left(\mathsf{b}_{l,j} \times \sum_{c \in C_{i,j}} \mathsf{mu}_c \right)$$

where $\mathsf{b}_{l,j}$ is a BDD over variables $\underline{\mathsf{z}}_l$, encoding $b_{l,j} \in \mathbb{B}^{k_l}$, i.e. $f_{\mathsf{b}_{l,j}}(b)$ is equal to 1 if $b = b_{l,j}$ and 0 otherwise for all $b \in \mathbb{B}^{k_l}$. Then, like in the previous two sections, we multiply each MTBDD $\mathsf{Steps}_{i,ind}$ by identity matrices corresponding to the other $m-1$ modules, encoding the fact that module M_i moves independently:

$$\mathsf{Steps}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{Steps}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

where id_j is the BDD IDENTITY$(\underline{\mathsf{x}}_j, \underline{\mathsf{y}}_j)$. We then combine the MTBDDs Steps_i to build the final MTBDD Steps. As described in Section 4.2.1, we use m nondeterministic variables to encode the scheduling between the m modules. Since we use variables $(\mathsf{z}_{s,1}, \ldots, \mathsf{z}_{s,k_s})$ for this purpose, we set $k_s = m$. For $i = 1, \ldots, m$, we will encode module M_i with $b_{s,i} \in \mathbb{B}^{k_s}$ where $b_{s,1} = (1, 0, \ldots, 0)$, $b_{s,2} = (0, 1, 0, \ldots, 0)$, etc. Then:

$$\mathsf{Steps} := \sum_{i=1}^{m} \mathsf{b}_{s,i} \times \mathsf{Steps}_i$$

where $\mathsf{b}_{s,i}$ is a BDD over variables $\underline{\mathsf{z}}_s$, encoding $b_{s,i} \in \mathbb{B}^{k_s}$, i.e. $f_{\mathsf{b}_{s,i}}(b)$ is equal to 1 if $b = b_{s,i}$ and 0 otherwise for all $b \in \mathbb{B}^{k_s}$.

Proof of Correctness

Reasoning as for DTMCs we have that, for any $s \in S$ and $t_i \in S_i$:

$$f_{\mathsf{mu}_c}[\underline{\mathsf{x}} = enc(s), \underline{\mathsf{y}}_i = enc_i(t_i)] = \begin{cases} \mu_{c,s}(t_i) & \text{if } s \in S_c \\ 0 & \text{otherwise} \end{cases}$$

From this and from the definition of $Steps_{i,ind}$ it follows that, for any $s \in S$:

- if $\mu_i \in \mathit{Steps}_{i,ind}(s)$, then there exists $b \in \mathbb{B}^{k_l}$ such that:
  - $f_{\mathsf{Steps}_{i,ind}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}}_i = \mathit{enc}_i(t_i), \underline{\mathsf{z}}_l = b] = \mu_i(t_i)$ for all $t_i \in S_i$
- for any $b \in \mathbb{B}^{k_l}$, one of the following two conditions holds:
  - $f_{\mathsf{Steps}_{i,ind}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}}_i = \mathit{enc}_i(t_i), \underline{\mathsf{z}}_l = b] = 0$ for all $t_i \in S_i$
  - there exists $\mu_i \in \mathit{Steps}_{i,ind}(s)$ such that $f_{\mathsf{Steps}_{i,ind}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}}_i = \mathit{enc}_i(t_i), \underline{\mathsf{z}}_l = b] = \mu_i(t_i)$ for all $t_i \in S_i$

since the sets S_c for commands c within each set $C_{i,j}$ are disjoint and each set $C_{i,j}$ is encoded by a unique $b_{l,j} \in \mathbb{B}^{k_l}$. Hence, for any $s \in S$:

- if $\mu \in \mathit{Steps}_i(s)$, then there exists $b \in \mathbb{B}^{k_l}$ such that:
  - $f_{\mathsf{Steps}_i}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}}_l = b] = \mu(t)$ for all $t \in S$
- for any $b \in \mathbb{B}^{k_l}$, one of the following two conditions holds:
  - $f_{\mathsf{Steps}_i}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}}_l = b] = 0$ for all $t \in S$.
  - there exists $\mu \in \mathit{Steps}_i(s)$ such that $f_{\mathsf{Steps}_i}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}}_l = b] = \mu(t)$ for all $t \in S$

by the definition of Steps_i and since $f_{\mathsf{id}_j}[\underline{\mathsf{x}}_j = \mathit{enc}_j(s_j), \underline{\mathsf{y}}_j = \mathit{enc}_j(t_j)] = 1$ if $s_j = t_j$ and 0 otherwise for all $s_j, t_j \in S_j$. Finally, as required, for any $s \in S$:

- if $\mu \in \mathit{Steps}(s)$, then there exists $b \in \mathbb{B}^k$ such that:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$
- for any $b \in \mathbb{B}^k$, one of the following two conditions holds:
  - $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = 0$ for all $t \in S$
  - there exists $\mu \in \mathit{Steps}(s)$ such that $f_{\mathsf{Steps}}[\underline{\mathsf{x}} = \mathit{enc}(s), \underline{\mathsf{y}} = \mathit{enc}(t), \underline{\mathsf{z}} = b] = \mu(t)$ for all $t \in S$

since each $\mu \in \mathit{Steps}(s)$ is in $\mathit{Steps}_i(s)$ for some i and is encoded by $b \in \mathbb{B}^k$ where $b = (b_{s,i}, b_{l,j})$, letting $b_{s,i} \in \mathbb{B}^{k_s}$ and $b_{l,j} \in \mathbb{B}^{k_l}$ be as defined previously.

C.4 Global Variables

To handle global variables, we add extra MTBDD variable sets $\underline{\mathsf{x}}_{glob}$ and $\underline{\mathsf{y}}_{glob}$ and assume that the state space of the global variables S_{glob} is encoded by enc_{glob} such that $enc(s_1, \ldots, s_m, s_{glob}) = (enc_1(s_1), \ldots, enc_m(s_m), enc_{glob}(s_{glob}))$. References to global variables in either guards or updates are translated in exactly the same fashion and the construction process for all three types of model remains unchanged.

C.5 Synchronisation

Now, we extend our description to include synchronisation between modules. Again, the construction closely follows the description of the semantics. The definition of the MTBDD mu_c for a command c remains the same as in the asynchronous case. Below, we define the construction for each of the three types of model. The proofs of its correctness with respect to the semantics are similar to those in the previous sections and are omitted.

Discrete-Time Markov Chains (DTMCs)

Independent commands are treated as before, i.e. for each module M_i:

$$\mathsf{P}_{i,ind} := \sum_{c \in C_{i,ind}} \mathsf{mu}_c$$

$$\mathsf{P}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{P}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

Action-labelled commands are handled as follows. For each $a \in Act$:

$$\mathsf{P}_{i,a} := \sum_{c \in C_{i,a}} \mathsf{mu}_c$$

$$\mathsf{P}_a := \prod_{i \in \{i \,|\, a \in A_i\}} \mathsf{P}_{i,a} \times \prod_{i \in \{i \,|\, a \notin A_i\}} \mathsf{id}_i$$

Then, the MTBDD representing the DTMC is constructed by combining the MTBDDs corresponding to its independent and synchronising behaviour:

$$\mathsf{P} := \textsc{Unif}\left(\sum_{i=1}^{m} \mathsf{P}_i + \sum_{a \in Act} \mathsf{P}_a\right)$$

where, as before, $\textsc{Unif}(\mathsf{M}) = \mathsf{M} \div \textsc{Abstract}(+, \underline{\mathsf{y}}, \mathsf{M})$.

Continuous-Time Markov Chains (CTMCs)

The construction process for CTMCs is almost identical to that for DTMCs. First, for each module M_i:

$$\mathsf{R}_{i,ind} := \sum_{c \in C_{i,ind}} \mathsf{mu}_c$$

$$\mathsf{R}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{R}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

Then, for each action $a \in Act$:

$$\mathsf{R}_{i,a} := \sum_{c \in C_{i,a}} \mathsf{mu}_c$$

$$\mathsf{R}_a := \prod_{i \in \{i \,|\, a \in A_i\}} \mathsf{R}_{i,a} \times \prod_{i \in \{i \,|\, a \notin A_i\}} \mathsf{id}_i$$

Finally:

$$\mathsf{R} := \sum_{i=1}^{m} \mathsf{R}_i + \sum_{a \in Act} \mathsf{R}_a$$

Markov Decision Processes (MDPs)

The process for MDPs is slightly more complicated. Again, we will use a set of k nondeterministic variables $\underline{\mathsf{z}}$ and assume that it is split into two sets, $\underline{\mathsf{z}}_s = (\mathsf{z}_{s,1}, \ldots, \mathsf{z}_{s,k_s})$ to encode nondeterminism arising from scheduling between modules, and $\underline{\mathsf{z}}_l = (\mathsf{z}_{l,1}, \ldots \mathsf{z}_{l,k_l})$ to encode local nondeterminism within a module.

Independent commands are treated in similar fashion to the asynchronous case. For each module M_i, we partition $C_{i,ind}$ into a number, say n_l, of subsets, $C_{i,ind,1}, \ldots, C_{i,ind,n_l}$, such that the guards of the commands within each subset do not overlap. Taking $b_{l,1}, \ldots, b_{l,n_l}$ to be distinct elements of $\mathbb{B}^{k_l}$, we encode each subset $C_{i,ind,j}$ with $b_{l,j}$:

$$\mathsf{Steps}_{i,ind} := \sum_{j=1}^{n_l} \left(\mathsf{b}_{l,j} \times \sum_{c \in C_{i,ind,j}} \mathsf{mu}_c \right)$$

where $\mathsf{b}_{l,j}$ is a BDD over variables $\underline{\mathsf{z}}_l$ encoding $b_{l,j}$. We then, as before, multiply by the identity matrices for the other $m-1$ modules:

$$\mathsf{Steps}_i := \mathsf{id}_1 \times \cdots \times \mathsf{id}_{i-1} \times \mathsf{Steps}_{i,ind} \times \mathsf{id}_{i+1} \times \cdots \times \mathsf{id}_m$$

Now, we consider synchronisation over each action $a \in Act$. For this case, we require separate nondeterministic variables for each module. Hence, we assume that $\underline{\mathsf{z}}_l$ is partitioned into m subsets $\underline{\mathsf{z}}_{l,1}, \ldots, \underline{\mathsf{z}}_{l,m}$ where subset $\underline{\mathsf{z}}_{l,i}$ contains $k_{l,i}$ variables, i.e. $\sum_{i=1}^{m} k_{l,i} = k_l$.

The MTBDD $\mathsf{Steps}_{i,a}$ is constructed from the MTBDDs mu_c for commands c in $C_{i,a}$. As above, we partition this set of commands into a number, say n_l, of subsets

$C_{i,a,1}, \ldots, C_{i,a,n_l}$, such that the guards of the commands within each subset do not overlap. Taking $b_{l,i,1}, \ldots, b_{l,i,n_l}$ to be distinct elements of $\mathbb{B}^{k_{l,i}}$ for $1 \leq j \leq n_l$, we encode subset $C_{i,a,j}$ with $b_{l,i,j}$:

$$\mathsf{Steps}_{i,a} := \sum_{j=1}^{n_l} \left(\mathsf{b}_{l,i,j} \times \sum_{c \in C_{i,a,j}} \mathsf{mu}_c \right)$$

The MTBDDs $\mathsf{Steps}_{i,a}$ for modules M_i which synchronise on action a are then combined with identity matrices for those which do not:

$$\mathsf{Steps}_a := \prod_{i \in \{i \,|\, a \in A_i\}} \mathsf{Steps}_{i,a} \times \prod_{i \in \{i \,|\, a \notin A_i\}} (\mathsf{b}_{l,i,0} \times \mathsf{id}_i)$$

where id_i is the BDD IDENTITY($\underline{\mathsf{x}}_i, \underline{\mathsf{y}}_i$) and $\mathsf{b}_{l,i,0}$ encodes $b_{l,i,0} \in \mathbb{B}^{k_{l,i}}$ where $b_{l,i,0} = (0, \ldots, 0)$. In this way, Steps_a represents all global transitions corresponding to action a and each nondeterministic choice in a state is encoded with a unique element of $\mathbb{B}^{k_l}$.

Finally, the independent transitions for each module and the synchronous transitions for each action are combined. Since all of these can potentially occur in any state, we must encode nondeterminism between them. As before we use the k_s nondeterministic variables in $\underline{\mathsf{z}}_s$. For this purpose, we require $m + |Act|$ distinct elements of $\mathbb{B}^{k_s}$, which we will denote $b_{s,i}$ for $1 \leq i \leq m$ and $b_{s,a}$ for $a \in Act$. To do this, we set $k_s = m + |Act|$ and use elements of the form $(1, 0, \ldots, 0)$, $(0, 1, 0, \ldots, 0)$, etc. as in the asynchronous case. The final construction step is then:

$$\mathsf{Steps} := \sum_{i=1}^{m} (\mathsf{b}_{s,i} \times \mathsf{Steps}_i) + \sum_{a \in Act} (\mathsf{b}_{s,a} \times \mathsf{Steps}_a)$$

where $\mathsf{b}_{s,i}$ and $\mathsf{b}_{s,a}$ are BDDs over the variables $\underline{\mathsf{z}}_s$ encoding $b_{s,i}$ and $b_{s,a}$, respectively.

C.6 Reachability

After construction of a DTMC, CTMC or MDP, we compute its set of reachable states and remove any unreachable ones. Reachability is performed by executing a breadth-first search, implemented as a BDD fixpoint algorithm. We first build the MTBDD T over variables $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$, representing the transition relation of the underlying graph. Depending on the model type, this is constructed as:

- T := THRESHOLD($\mathsf{P}, >, 0$)
- T := THRESHOLD($\mathsf{R}, >, 0$)
- T := THEREEXISTS($\underline{\mathsf{z}}$, THRESHOLD($\mathsf{Steps}, >, 0$))

We assume that the initial state $\overline{s} \in S$ is represented by a BDD init over variables $\underline{\mathsf{x}}$, i.e. $f_{\mathsf{init}}[\underline{\mathsf{x}} = enc(s)]$ is equal to 1 if $s = \overline{s}$ and 0 otherwise for all $s \in S$. The algorithm to compute the BDD representing the set of reachable states is then REACH(init):

```
REACH(init)
1.  sol := REPLACEVARS(init, x, y)
2.  done := false
3.  while (done = false)
4.      sol' := sol ∨ THEREEXISTS(x, T ∧ REPLACEVARS(sol, y, x))
5.      if (sol' = sol) then done := true
6.      sol := sol'
7.  endwhile
8.  return REPLACEVARS(sol, y, x)
```

This algorithm returns a BDD over variables $\underline{\mathsf{x}}$, which we will denote reach, representing the set of reachable states. We then use this BDD to remove unreachable states from the model. In fact, rather than sacrifice our efficient state space encoding, we actually just remove any non-zero entries from rows and columns corresponding to these states. Since, by definition, we cannot reach an unreachable state from a reachable one, we only need to do this for rows, not columns. This is performed as follows:

- $\mathsf{P}' := \mathsf{reach} \times \mathsf{P}$
- $\mathsf{R}' := \mathsf{reach} \times \mathsf{R}$
- $\mathsf{Steps}' := \mathsf{reach} \times \mathsf{Steps}$

Finally, we identify any deadlock states in the model, i.e. reachable states with no outgoing transitions. This is done by computing a BDD T', representing the transition relation of the new model with no unreachable states:

- $\mathsf{T}' := \text{THRESHOLD}(\mathsf{P}', >, 0)$
- $\mathsf{T}' := \text{THRESHOLD}(\mathsf{R}', >, 0)$
- $\mathsf{T}' := \text{THEREEXISTS}(\underline{\mathsf{z}}, \text{THRESHOLD}(\mathsf{Steps}', >, 0))$

We then use the THEREEXISTS operator to identify states which have at least one outgoing transition and negate:

- $\mathsf{deadlocks} := \mathsf{reach} \wedge \neg\, \text{THEREEXISTS}(\underline{\mathsf{y}}, \mathsf{T}')$

after which, $\mathsf{deadlocks}$ is a BDD over the variables $\underline{\mathsf{x}}$ identifying all deadlock states.

Appendix D

MTBDD Model Checking Algorithms

This appendix contains the algorithms for implementing PCTL and CSL model checking with MTBDDs. The original algorithms were presented in Chapter 3 and the issues involved in developing a symbolic implementation were discussed in Chapter 5.

The bulk of the work for model checking constitutes calculation of the probabilities for the $\mathcal{P}$ operator of PCTL and the $\mathcal{P}$ and $\mathcal{S}$ operators of CSL. Typically, this requires two phases: application of precomputation algorithms to quickly identify states where the probability is exactly 0 or 1; and numerical computation to determine the probabilities for the remaining states. The algorithms for each phase are covered in Sections D.1 and D.2, respectively. The complete list of algorithms described here is:

Precomputation	Numerical computation		
	DTMCs	MDPs	CTMCs
Prob0	PctlNext	PctlNextMax	CslNext
Prob1	PctlBoundedUntil	PctlNextMin	CslUntil
Prob0A	PctlUntil	PctlBoundedUntilMax	CslBoundedUntil
Prob1E		PctlUntilMax	CslSteadyState
Prob0E		PctlUntilMin	
		PctlUntilMinFair	

In the following, we assume that $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$ and $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$ are sets of row and column variables, respectively. A DTMC is represented by an MTBDD P over $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$ and a CTMC is represented by an MTBDD R over $\underline{\mathsf{x}}$ and $\underline{\mathsf{y}}$. For an MDP, we also use the set of nondeterministic variables $\underline{\mathsf{z}} = (\mathsf{z}_1, \ldots, \mathsf{z}_k)$ and represent the model by an MTBDD Steps over $\underline{\mathsf{x}}$, $\underline{\mathsf{y}}$ and $\underline{\mathsf{z}}$. We use a BDD reach over variables $\underline{\mathsf{x}}$ to denote the set of reachable states of the model.

D.1 Precomputation Algorithms

We use five precomputation algorithms. These are: PROB0, PROB1, PROB0A, PROB1E and PROB0E. The algorithms were described in Sections 3.3.1 and 3.3.2 and their symbolic implementation was discussed in Section 5.2.

All five algorithms are entirely BDD-based. As in the algorithm for reachability in Appendix C, only the existence of a transition is relevant, not the actual probabilities or rates associated with it. This information is stored in a BDD T which, depending on the model type, this is computed as follows:

- $\mathsf{T} := \textsc{Threshold}(\mathsf{P}, >, 0)$ (for DTMCs)
- $\mathsf{T} := \textsc{Threshold}(\mathsf{R}, >, 0)$ (for CTMCs)
- $\mathsf{T} := \textsc{Threshold}(\mathsf{Steps}, >, 0)$ (for MDPs)

Note that in the case for MDPs, the BDD T is different from the one used to perform reachability in Appendix C; for these algorithms, we have to preserve information about which transitions correspond to which nondeterministic choice.

All five algorithms take the sets $Sat(\phi_1)$ and $Sat(\phi_2)$ as arguments. These are represented by the BDDs phi_1 and phi_2, respectively. The algorithm PROB1 takes the result of PROB0 as an additional argument. This is represented by the BDD $\mathsf{prob0}$.

PROB0($\mathsf{phi}_1, \mathsf{phi}_2$)

1. $\mathsf{sol} := \mathsf{phi}_2$
2. $done :=$ **false**
3. **while** ($done =$ **false**)
4. $\quad \mathsf{sol}' := \mathsf{sol} \vee (\mathsf{phi}_1 \wedge \textsc{ThereExists}(\underline{\mathsf{y}}, \mathsf{T} \wedge \textsc{ReplaceVars}(\mathsf{sol}, \underline{\mathsf{x}}, \underline{\mathsf{y}})))$
5. $\quad$ **if** ($\mathsf{sol}' = \mathsf{sol}$) **then** $done :=$ **true**
6. $\quad \mathsf{sol} := \mathsf{sol}'$
7. **endwhile**
8. **return** $\neg\,\mathsf{sol}$

PROB1($\mathsf{phi}_1, \mathsf{phi}_2, \mathsf{prob0}$)

1. $\mathsf{sol} := \mathsf{prob0}$
2. $done :=$ **false**
3. **while** ($done =$ **false**)
4. $\quad \mathsf{sol}' := \mathsf{sol} \vee (\mathsf{phi}_1 \wedge \neg\,\mathsf{phi}_2 \wedge \textsc{ThereExists}(\underline{\mathsf{y}}, \mathsf{T} \wedge \textsc{ReplaceVars}(\mathsf{sol}, \underline{\mathsf{x}}, \underline{\mathsf{y}})))$
5. $\quad$ **if** ($\mathsf{sol}' = \mathsf{sol}$) **then** $done :=$ **true**
6. $\quad \mathsf{sol} := \mathsf{sol}'$
7. **endwhile**
8. **return** $\neg\,\mathsf{sol}$

```
PROB0A(phi_1, phi_2)
1.  sol := phi_2
2.  done := false
3.  while (done = false)
4.      tmp := THEREEXISTS(y, T ∧ REPLACEVARS(sol, x, y))
5.      sol' := sol ∨ (phi_1 ∧ THEREEXISTS(z, tmp))
6.      if (sol' = sol) then done := true
7.      sol := sol'
8.  endwhile
9.  return ¬ sol
```

```
PROB1E(phi_1, phi_2)
1.  sol := reach
2.  done := false
3.  while (done = false)
4.      sol' := phi_2
5.      done' := false
6.      while (done' = false)
7.          tmp_1 := FORALL(y, T → REPLACEVARS(sol, x, y))
8.          tmp_2 := THEREEXISTS(y, T ∧ REPLACEVARS(sol', x, y))
9.          sol'' := sol' ∨ (phi_1 ∧ THEREEXISTS(z, tmp_1 ∧ tmp_2))
10.         if (sol'' = sol') then done' := true
11.         sol' := sol''
12.     endwhile
13.     if (sol' = sol) then done := true
14.     sol := sol'
15. endwhile
16. return sol
```

```
PROB0E(phi_1, phi_2)
1.  sol := phi_2
2.  done := false
3.  while (done = false)
4.      tmp := THEREEXISTS(y, T ∧ REPLACEVARS(sol, x, y))
5.      sol' := sol ∨ (phi_1 ∧ FORALL(z, tmp))
6.      if (sol' = sol) then done := true
7.      sol := sol'
8.  endwhile
9.  return ¬ sol
```

D.2 Numerical Computation

PCTL over DTMCs

The algorithms to compute probabilities for the PCTL next ($X\,\phi$), PCTL bounded until ($\phi_1\,\mathcal{U}^{\leq k}\,\phi_2$) and PCTL until ($\phi_1\,\mathcal{U}\,\phi_2$) operators over DTMCs were described in Section 3.3.1. The corresponding MTBDD algorithms are PCTLNEXT, PCTLBOUNDEDUNTIL and PCTLUNTIL. The arguments of these algorithms, $Sat(\phi)$, $Sat(\phi_1)$ or $Sat(\phi_2)$, are represented by BDDs phi, phi_1 and phi_2, respectively. PCTLBOUNDEDUNTIL has a additional argument: the bound k. All three return the vector of probabilities as an MTBDD.

```
PCTLNEXT(phi)
1.  probs := MVMULT(P, phi)
2.  return probs
```

```
PCTLBOUNDEDUNTIL(phi_1, phi_2), k
1.  s_no := ¬(phi_1 ∨ phi_2)
2.  s_yes := phi_2
3.  s_? := ¬(s_no ∨ s_yes)
4.  P' := s_? × P
5.  probs = s_yes
6.  for (i = 1...k)
7.      probs := MVMULT(P', probs)
8.      probs := probs + yes
9.  endfor
10. return probs
```

```
PCTLUNTIL(phi_1, phi_2)
1.  s_no := PROB0(phi_1, phi_2)
2.  s_yes := PROB1(phi_1, phi_2, s_no)
3.  s_? := ¬(s_no ∨ s_yes)
4.  P' := s_? × P
5.  A := IDENTITY(x, y) − P'
6.  b := s_yes
7.  probs := SOLVEJACOBI(A, b, b)
8.  return probs
```

In the above, the function SOLVEJACOBI($\mathsf{A}, \mathsf{b}, \mathsf{init}$) takes MTBDDs representing a matrix $\mathbf{A}$ and vectors $\underline{b}$ and $\underline{init}$, respectively. It returns the solution to the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$, computed using the Jacobi method with initial approximation $\underline{init}$. As discussed in Section 5.3.1, this and several other iterative algorithms presented here are terminated when the maximum relative difference between elements of consecutive vectors drops below some threshold ε. This is checked using the function MAXDIFF.

```
SOLVEJACOBI(A, b, init)
1.  d := ABSTRACT(max, y, A × IDENTITY(x, y))
2.  A' := A × CONST(−1) × ¬IDENTITY(x, y)
3.  A' := A' ÷ d
4.  b' := b ÷ d
5.  sol := init
6.  done := false
7.  while (done = false)
8.      sol' := MVMULT(A', sol)
9.      sol' := sol' + b'
10.     if (MAXDIFF(sol, sol') < ε) then
11.         done := true
12.     endif
13.     sol := sol'
14. endwhile
15. return sol
```

The algorithm above can trivially be converted to the JOR method, with over-relaxation parameter ω, by adding the following statement between lines 9 and 10:

$$\mathsf{sol}' := (\mathsf{sol}' \times \textsc{Const}(\omega)) + (\mathsf{sol} \times \textsc{Const}(1-\omega))$$

PCTL over MDPs

The algorithms to compute probabilities for the PCTL next ($X\,\phi$), PCTL bounded until ($\phi_1\,\mathcal{U}^{\leq k}\,\phi_2$) and PCTL until ($\phi_1\,\mathcal{U}\,\phi_2$) operators over MDPs were described in Section 3.3.2. For each, we must consider two cases: one for computing maximum probabilities, $p_s^{max}(\cdot)$, and one for computing minimum probabilities, $p_s^{min}(\cdot)$.

As above, the arguments of the algorithms, $Sat(\phi)$, $Sat(\phi_1)$ or $Sat(\phi_2)$, are represented by BDDs phi, phi_1 and phi_2, respectively and the vector of probabilities is returned as an MTBDD. For the PCTL next operator, the required algorithms are PCTLNEXTMAX and PCTLNEXTMIN.

```
PCTLNEXTMAX(phi)
1.  probs := MVMULT(Steps, phi)
2.  probs := ABSTRACT(max, z, probs)
3.  return probs
```

```
PCTLNEXTMIN(phi)
1.  probs := MVMULT(Steps, phi)
2.  probs := ABSTRACT(min, z, probs)
3.  return probs
```

For computing $p_s^{max}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_1)$, we use PCTLBOUNDEDUNTILMAX. This has an additional argument: the bound k.

```
PCTLBOUNDEDUNTILMAX(phi_1, phi_2, k)
1.   s_no := ¬(phi_1 ∨ phi_2)
2.   s_yes := phi_2
3.   s_? := ¬(s_no ∨ s_yes)
4.   Steps' := s_? × Steps
5.   probs = s_yes
6.   for (i = 1...k)
7.        probs := MVMULT(P', probs)
8.        probs := ABSTRACT(max, z, probs)
9.        probs := probs + yes
10.  endfor
11.  return probs
```

The dual function for computing $p_s^{min}(\phi_1 \, \mathcal{U}^{\leq k} \, \phi_1)$ is PCTLBOUNDEDUNTILMIN. This can be obtained from the above by replacing the 'max' in line 8 with 'min'.

For PCTL until formulas, we must also distinguish between the case for all adversaries and the case for fair adversaries only. The algorithm PCTLUNTILMAX below computes $p_s^{max}(\phi_1 \, \mathcal{U} \, \phi_1)$ over all adversaries.

```
PCTLUNTILMAX(phi_1, phi_2)
1.   s_no := PROB0A(phi_1, phi_2)
2.   s_yes := PROB1E(phi_1, phi_2)
3.   s_? := ¬(s_no ∨ s_yes)
4.   Steps' := s_? × Steps
5.   probs := s_yes
6.   done := false
7.   while (done = false)
8.        probs' := MVMULT(Steps', probs)
9.        probs' := ABSTRACT(max, z, probs')
10.       probs' := probs' + yes
11.       if (MAXDIFF(probs, probs') < ε) then
12.            done := true
13.       endif
14.       probs := probs'
15.  endwhile
16.  return probs
```

The dual function PCTLUNTILMIN, which computes $p_s^{min}(\phi_1 \, \mathcal{U} \, \phi_1)$ over all adversaries, can be seen below. There are only two differences: the substitution of 'max' for 'min' in line 9; and the use of alternative precomputation algorithms.

```
PctlUntilMin(phi_1, phi_2)
 1.  s_no := Prob0E(phi_1, phi_2)
 2.  s_yes := phi_2
 3.  s_? := ¬ (s_no ∨ s_yes)
 4.  Steps' := s_? × Steps
 5.  probs := s_yes
 6.  done := false
 7.  while (done = false)
 8.        probs' := MVMult(Steps', probs)
 9.        probs' := Abstract(min, z, probs')
10.        probs' := probs' + yes
11.        if (MaxDiff(probs, probs') < ε) then
12.              done := true
13.        endif
14.        probs := probs'
15.  endwhile
16.  return probs
```

As described in Section 3.3.2, the calculation of $p_s^{max}(\phi_1 \; \mathcal{U} \; \phi_1)$ remains unchanged when considering fair adversaries only. Hence, we can reuse PctlUntilMax from above. For $p_s^{min}(\phi_1 \; \mathcal{U} \; \phi_1)$ over fair adversaries, the computation is different, requiring conversion to a dual problem. This is handled by the algorithm PctlUntilMinFair below.

```
PctlUntilMinFair(phi_1, phi_2)
1.  a_+ := ¬Prob0A(phi_1, phi_2)
2.  a_# := a_+ ∧ ¬phi_2
3.  probs := PctlUntilMax(a_#, ¬a_+)
4.  probs := Const(1) − probs
5.  return probs
```

CSL over CTMCs

The probabilities required for model checking the CSL next, time-bounded until, until and steady-state operators over CTMCs are computed as described in Section 3.3.3. As above, in the corresponding MTBDD algorithms, the arguments $Sat(\phi)$, $Sat(\phi_1)$ and $Sat(\phi_2)$ are represented by BDDs phi, phi_1 and phi_2, respectively. The time-bounded until model checking algorithm has an extra argument t. In all cases, the vector of probabilities is returned as an MTBDD.

For the CSL next and until operators, we compute the embedded DTMC $\mathbf{P}$, represented by MTBDD P, of the CTMC, and reuse the algorithms for PCTL over DTMCs given above. This is done by the algorithms CslNext and CslUntil.

```
CSLNEXT(phi)
1.  diags := ABSTRACT(+, y, R)
2.  P := R ÷ diags
3.  return PCTLNEXT(phi)
```

```
CSLUNTIL(phi_1, phi_2)
1.  diags := ABSTRACT(+, y, R)
2.  P := R ÷ diags
3.  return PCTLUNTIL(phi_1, phi_2)
```

For the time-bounded until operator, we use the algorithm CSLBOUNDEDUNTIL.

```
CSLBOUNDEDUNTIL(phi_1, phi_2, t)
1.  s_no := PROB0(phi_1, phi_2)
2.  s_yes := phi_2
3.  s_? := ¬(s_no ∨ s_yes)
4.  diags := ABSTRACT(+, y, R) × CONST(−1)
5.  Q := R + (diags × IDENTITY(x, y))
6.  Q := Q × s_?
7.  q := −1.02 × FINDMIN(diags × s_?)
8.  P := IDENTITY(x, y) + CONST(1/q) × Q
9.  (L_ε, R_ε, {γ_i}_{i=L_ε...R_ε}) := FOXGLYNN(q · t, ε)
10. sol := yes
11. for (k := 1 ... L_ε − 1)
12.       sol' := MVMULT(P, sol)
13.       if (MAXDIFF(sol, sol') < ε) then
14.             return sol'
15.       endif
16.       sol := sol'
17. endfor
18. probs := CONST(0)
19. for (k := L_ε ... R_ε)
20.       if (k > 0) then
21.             sol' := MVMULT(P, sol)
22.             if (MAXDIFF(sol, sol') < ε) then
23.                   return probs + CONST(Σ_{i=k}^{R_ε} γ_i) × sol'
24.             endif
25.             sol := sol'
26.       endif
27.       probs := probs + (sol × CONST(γ_k))
28. endfor
29. return probs
```

Note that the above includes an additional optimisation proposed in [BHHK00a]. We check to see if the iteration vector converges before reaching the upper bound R_ε. If

so, the iterative method can be terminated prematurely. Intuitively, this means that the time bound t is high enough that the steady-state has already been reached. The function FOXGLYNN in line 9 implements the method of [FG88] to compute Poisson probabilities. This was discussed in Section 3.3.3.

Finally, we give the algorithm CSLSTEADYSTATE, which computes the steady-state probabilities of the CTMC, as required for model checking the $\mathcal{S}$ operator.

CSLSTEADYSTATE(phi, n)
1. diags := ABSTRACT($+$, y, R) $\times$ CONST(-1)
2. Q := R + (diags $\times$ IDENTITY(x, y))
3. probs := SOLVEJACOBITRANS(Q, CONST(0), CONST($1/n$))
4. sol := ABSTRACT($+$, x, probs $\times$ phi)
5. **return** sol

The bulk of the work is done by the SOLVEJACOBITRANS function. Given the MTBDD A representing a matrix $\mathbf{A}$, and MTBDDs b and init, representing vectors $\underline{b}$ and $\underline{init}$, SOLVEJACOBITRANS(A, b, init) solves the linear equation system $\underline{x} \cdot \mathbf{A} = \underline{b}$, using the Jacobi method with initial approximation $\underline{init}$.

In this case, $\underline{init}$ is set to an equiprobable distribution over all states. The number of states n is passed in as a parameter. SOLVEJACOBITRANS can be implemented either by transposing the matrix $\mathbf{A}$ and using the algorithm SOLVEJACOBI given previously or, more efficiently, by rewriting SOLVEJACOBI to solve the transposed system directly. The only significant difference is that it is based on vector-matrix multiplications, rather than matrix-vector multiplications.

Appendix E

Case Studies

In order to test our implementation of probabilistic model checking, we applied the techniques described in this thesis to a large number of case studies from the literature. The selection of these for which we have presented results in the preceding chapters was chosen to demonstrate the applicability and effectiveness of our techniques on as wide a range as possible of models, logics and model checking algorithms. This appendix gives a brief description of each of these case studies and the properties for which verification results were presented in the thesis. We also provide the PRISM language description for one of the case studies. Information about further properties considered for each example and additional case studies can be found on the PRISM web site [Pri]. Particular thanks go to Gethin Norman, who was responsible for modelling many of these examples in PRISM.

E.1 DTMC Case Studies

E.1.1 Bounded Retransmission Protocol (BRP)

The bounded retransmission protocol (BRP) of Helmink, Sellink and Vaandrager [HSV94] is a simplified version of a Philips telecommunication protocol. Its purpose is to communicate messages over unreliable channels. The messages can be of arbitrary size and are transmitted as a number of packets. Packets which are not successfully sent are retransmitted, but only a fixed number of times, hence the name of the protocol. The model has two parameters: N, the number of packets to be sent, and K, the maximum number of times that each packet can be retransmitted. In the examples used in this thesis, we fix the value of K as 3. We verify that the probability of the sender eventually not reporting a successful transmission is sufficiently low. This can be expressed as a PCTL property of the form $\mathcal{P}_{<p}[\Diamond\, error]$.

E.2 MDP Case Studies

E.2.1 Rabin's Randomised Mutual Exclusion Algorithm

Rabin's randomised algorithm for mutual exclusion [Rab82] presents a probabilistic solution to the well known problem of controlling a set of processes which periodically require exclusive access to some shared resource. The parameter N of the model represents the number of processes present. We model check the liveness property: "if the shared resource is free and there is a process requesting access to it, then eventually, with probability 1, some process is given access to the resource". This can be expressed as a PCTL property of the form: $\mathit{request} \rightarrow \mathcal{P}_{\geq 1}[\Diamond\, \mathit{access}]$.

E.2.2 Lehmann and Rabin's Randomised Dining Philosophers

In [LR81], Lehmann and Rabin proposed a randomised algorithm as the solution to the classic, distributed resource-allocation problem: "the dining philosophers". The problem consists of N philosophers sat round a circular table who do nothing except think and eat. There is a bowl of food in the centre and N forks, placed between the philosophers. A philosopher can only eat if he has both his left and right forks. The task is to devise a protocol whereby the philosophers do not starve.

This case study is based on Lehmann and Rabin's algorithm and its analysis by Lynch, Saias and Segala [LSS94]. We build an additional constraint into the model that when a philosopher is hungry, at most K time steps can pass before it makes a move, where K is a parameter of the algorithm. This allows us to model a similar notion of fairness to that used in [LSS94].

We consider two model checking problems. Firstly, we verify that, "if any philosopher is hungry, then with probability at least $\frac{1}{8}$, some philosopher eats within t time steps". In PCTL, this becomes $\mathit{hungry} \rightarrow \mathcal{P}_{\geq \frac{1}{8}}[\Diamond^{\leq t}\, \mathit{eat}]$. This is checked for several values of t. Secondly, we verify that the algorithm is *livelock-free*, meaning that if some philosophers are hungry, then eventually, with probability 1, some philosopher will eat. This is expressed as a PCTL property of the form $\mathit{hungry} \rightarrow \mathcal{P}_{\geq 1}[\Diamond\, \mathit{eat}]$. This second property can be model checked with precomputation algorithms alone.

E.2.3 Randomised Consensus Coin Protocol

A distributed consensus protocol is an algorithm for ensuring that a set of distributed processes can all agree on the outcome of some decision. Here, we assume that there are only two possible outcomes: 1 and 2. In [AH90], Aspnes and Herlihy presented a

randomised solution to this problem. A crucial component of their algorithm is its *shared coin protocol*, which controls a counter, shared between the processes. Each process periodically increments or decrements the value of the counter, based on the outcome of a random coin toss. There are fixed lower and upper bounds for the value of the counter. The choice between outcomes 1 and 2 is made on the basis of which of these bounds is exceeded first. Our model of this coin protocol has two parameters: N, the number of processes; and K, which is used to determine the counter's bounds.

We model check two properties, the first of which is "with probability 1, all processes that enter the shared counter phase eventually leave it". This is expressed by a PCTL property of the form $\mathcal{P}_{\geq 1}[\Diamond\, \mathit{leave}]$. Since the bound attached to the $\mathcal{P}$ operator is 1, this can be verified using precomputation algorithms alone. Secondly, we compute, for each value $v \in \{1, 2\}$, the probability that processes entering the shared counter phase leave it having selected v. This is then compared to a bound p, giving a PCTL formula such as $\mathcal{P}_{\geq p}[\Diamond\, \mathit{decide}_v]$. Model checking this formula requires numerical computation. For full details of the analysis, see [KNS01].

E.2.4 FireWire Root Contention Protocol

IEEE 1394 (also known as FireWire) is a standard defining a high performance serial bus. It is designed to allow networking of a wide range of electronic devices in a flexible and scalable fashion. It is also 'hot-pluggable', meaning that devices can be added and removed from the bus while it is active. When the bus is reset, the network has to be reconfigured and a *root node* (the device which will act as manager of the bus) elected. Conflicts can occur when two nodes compete to become the root. The FireWire standard includes a randomised algorithm, called the *root contention protocol*, which resolves this.

This case study is based on Stoelinga and Vaandrager's model [SV99] of this randomised root contention protocol. We compute the probability that a root node is elected before some time deadline passes. The deadline is built into the model and is represented by a parameter N. To analyse this, we model check a PCTL property of the form $\mathcal{P}_{\geq p}[\Diamond\, \mathit{deadline}]$ where the atomic proposition *deadline* denotes states where the deadline has passed. In fact, we do not verify the property for specific values of the probability bound p, but instead determine the actual probability. For full details of the analysis of this protocol, see [KNS02, DKN02].

E.3 CTMC Case Studies

E.3.1 Kanban Manufacturing System

In [CT96], Ciardo and Tilgner present a model of a Kanban manufacturing system, which uses 'Kanban' tickets to control the flow of jobs between different machines in a production network. This model consists of four machines. The parameter N denotes the maximum number of jobs in each machine at any one time. We compute the steady-state probability distribution of the model. This can be used to model check any CSL formula which is based on the $\mathcal{S}$ operator.

E.3.2 Cyclic Server Polling System

This case study is based on Ibe and Trivedi's model of a cyclic server polling system from [IT90]. The system comprises N queues which are serviced by a single server. The server polls the queues in a cyclic fashion, determining whether or not there are jobs in the queue to be processed. We model check a CSL property of the form $\mathcal{S}_{<p}[busy_i \wedge \neg serve_i]$ which states that, in the long-run, the probability that queue i is waiting to be served is less than some bound p.

E.3.3 Tandem Queueing Network

In [HMKS99], Hermanns et al. present a CTMC model of a tandem queueing network. It consists of an M/Cox_2/1-queue sequentially composed with an M/M/1-queue. Both queues have the same capacity, determined by a parameter N. We model check a CSL property of the form $\mathcal{P}_{<p}[\Diamond^{\leq t}\, full_1]$ which states that the probability of the first queue becoming full within t time units is less than some bound p.

E.3.4 Flexible Manufacturing System (FMS)

We use Ciardo and Trivedi's [CT93] model of a flexible manufacturing system. It consists of three separate machines which collaborate to process and then assemble several different types of part. A parameter N of the model determines the maximum number of parts which each machine can handle. For some time bound t and probability bound p, we verify that the probability of machine 1 reaching its full capacity within t time units is less than p. This can be expressed by the CSL formula $\mathcal{P}_{<p}[\Diamond^{\leq t}\, full_1]$.

E.4 PRISM Language Description

Finally, by way of example, we include the PRISM language description for one of our case studies: the cyclic server polling system with parameter N equal to 2.

ctmc

rate $mu = 1$;
rate $gamma = 200$;
rate $lambda = 0.5$; *// mu/N*

module *server*

$s : [1..2]$ **init** 1; *// station*
$a : [0..1]$ **init** 0; *// action: 0=polling, 1=serving*

$[loop1a]$	$(s = 1) \wedge (a = 0)$	$\rightarrow$	$gamma :$	$(s' = s + 1);$
$[loop1b]$	$(s = 1) \wedge (a = 0)$	$\rightarrow$	$gamma :$	$(a' = 1);$
$[serve1]$	$(s = 1) \wedge (a = 1)$	$\rightarrow$	$mu :$	$(s' = s + 1) \wedge (a' = 0);$
$[loop2a]$	$(s = 2) \wedge (a = 0)$	$\rightarrow$	$gamma :$	$(s' = 1);$
$[loop2b]$	$(s = 2) \wedge (a = 0)$	$\rightarrow$	$gamma :$	$(a' = 1);$
$[serve2]$	$(s = 2) \wedge (a = 1)$	$\rightarrow$	$mu :$	$(s' = 1) \wedge (a' = 0);$

endmodule

module *station*1

$s1 : [0..1]$ **init** 0; *// state of station: 0=empty, 1=full*

$[loop1a]$	$(s1 = 0)$	$\rightarrow$	$1 :$	$(s1' = 0);$
$[]$	$(s1 = 0)$	$\rightarrow$	$lambda :$	$(s1' = 1);$
$[loop1b]$	$(s1 = 1)$	$\rightarrow$	$1 :$	$(s1' = 1);$
$[serve1]$	$(s1 = 1)$	$\rightarrow$	$1 :$	$(s1' = 0);$

endmodule

// construct second module through renaming

module *station*2 =
$station1$ [$s1 = s2$, $loop1a = loop2a$, $loop1b = loop2b$, $serve1 = serve2$]
endmodule

Bibliography

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, 1990.

[AH99] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[Ake78] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[ASB+95] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, *Proc. 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *LNCS*, pages 155–165. Springer, 1995.

[ASSB96] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[Bai98] C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.

[BBO+02] R. Batsell, L. Brenner, D. Osherson, S. Tsavachidis, and M. Vardi. Eliminating incoherence from subjective estimates of chance. In *Proc. 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02)*, 2002.

[BC98] C. Baier and E. Clarke. The algebraic mu-calculus and MTBDDs. In *Proc. 5th Workshop on Logic, Language, Information and Computation (WOLLIC'98)*, pages 27–38, 1998.

[BCDK97] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of Kronecker operations on sparse matrices with applications to the solution of Markov models. ICASE Report 97-66, Institute for Computer Applications in Science and Engineering, 1997.

[BCHG+97] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 430–440. Springer, 1997.

[BCL91] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. Denyer, editors, *Proc. International Conference on Very Large Scale Integration (VLSI'91)*, volume A-1 of *IFIP Transactions*, pages 49–58. North-Holland, 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990.

[BCSS98] M. Bernardo, W. Cleaveland, S. Sims, and W. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proc. Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV'98)*, volume 135 of *IFIP Conference Proceedings*, pages 457–467. Kluwer, 1998.

[BdA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[BFG+93] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conference on Computer-Aided Design (ICCAD'93)*, pages 188–191, 1993. Also available in *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[BFG+97] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, 1997.

[BHHK00a] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Proc. 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.

[BHHK00b] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *LNCS*, pages 780–792. Springer, 2000.

[BK98] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[BK01] P. Buchholz and P. Kemper. Compact representations of probability distributions in the analysis of superposed GSPNs. In R. German and B. Haverkort, editors, *Proc. 9th International Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 81–90. IEEE Computer Society Press, 2001.

[BKH99] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In J. Baeten and S. Mauw, editors, *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.

[BM99] M. Bozga and O. Maler. On the representation of probabilities over structured domains. In N. Halbwachs and D. Peled, editors, *Proc. 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 261–273. Springer, 1999.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automation Conference (DAC'90)*, pages 40–45. IEEE Computer Society Press, 1990.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BT91] D. Bertsekas and J. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991.

[BW96] B. Bollig and I. Wegner. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.

[Cam96] S. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[CE81] E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *LNCS*. Springer, 1981.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logics. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, pages 1–15, 1993. Also available in *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[CGH+93] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proc. 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL'93)*, volume A-32 of *IFIP Transactions*, pages 15–30. North-Holland, 1993.

[CM96] G. Ciardo and A. Miner. SMART: Simulation and Markovian analyser for reliability and timing. In *Proc. 2nd International Computer Performance and Dependability Symposium (IPDS'96)*, page 60. IEEE Computer Society Press, 1996.

[CM97] G. Ciardo and A. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *LNCS*, pages 44–57. Springer, 1997.

[CM99] G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz and M. Silva, editors, *Proc. 8th International Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31. IEEE Computer Society Press, 1999.

[CMZ+93] E. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. 30th Design Automation Conference (DAC'93)*, pages 54–60. ACM Press, 1993. Also available in *Formal Methods in System Design*, 10(2/3):137–148, 1997.

[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[CT96] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stocastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite state probabilistic programs. In *Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS'88)*, pages 338–345. IEEE Computer Society Press, 1988.

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In M. Paterson, editor, *Proc. 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *LNCS*, pages 336–349. Springer, 1990.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[dA97] L. de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford University, 1997.

[dA98] L. de Alfaro. From fairness to chance. In C. Baier, M. Huth, M. Kwiatkowska, and M. Ryan, editors, *Proc. 1st International Workshop on Probabilistic Methods in Verification (PROBMIV'98)*, volume 22 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[dAKN+00] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs

and the Kronecker representation. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.

[DB95] A. Dsouza and B. Bloom. Generating BDDs for symbolic model checking in CCS. In P. Wolper, editor, *Proc. 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *LNCS*, pages 16–30. Springer, 1995.

[DJJL01] P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In L. de Alfaro and S. Gilmore, editors, *Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01)*, volume 2165 of *LNCS*, pages 39–56. Springer, 2001.

[DKN02] C. Daws, M. Kwiatkowska, and G. Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. In R. Cleaveland and H. Garavel, editors, *Proc. 7th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66.2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[Don94] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In R. Valette, editor, *Proc. 15th International Conference on Application and Theory of Petri Nets (APN'94)*, volume 815 of *LNCS*, pages 258–277. Springer, 1994.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer, 1996.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In K. Larsen and A. Skou, editors, *Proc. 3rd International Workshop on Computer Aided Verification (CAV'91)*, volume 575 of *LNCS*, pages 203–213. Springer, 1991.

[FG88] B. Fox and P. Glynn. Computing Poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988.

[Fre94] L.-å. Fredlund. The timing and probability workbench: A tool for analysing timed processes. Technical Report DoCS 94/49, Department of Computer Systems, Uppsala University, 1994.

[GH94] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation)*, volume 794 of *LNCS*, pages 353–368. Springer, 1994.

[Han94] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1994.

[HCH+02] B. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier. Model checking performability properties. In *Proc. International Conference on Dependable Systems and Networks (DSN'02)*. IEEE Computer Society Press, 2002.

[HG98] V. Hartonas-Garmhausen. *Probabilistic Symbolic Model Checking with Engineering Models and Applications*. PhD thesis, Carnegie Mellon University, 1998.

[HGCC99] V. Hartonas-Garmhausen, S. Campos, and E. Clarke. ProbVerus: Probabilistic symbolic model checking. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 96–110. Springer, 1999.

[HHK+98] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. In R. Puigjaner, N. Savino, and B. Serra, editors, *Proc. 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS '98)*, volume 1469 of *LNCS*, pages 51–62. Springer, 1998.

[Hil94] J. Hillston. The nature of synchronisation. In U. Herzog and M. Rettelbach, editors, *Proc. 2nd Workshop on Process Algebras and Performance Modelling (PAPM '94)*, pages 51–70, 1994.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HKMKS00] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In S. Graf and M. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 347–362. Springer, 2000.

[HKN+02] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, 2002. To appear.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[HMPS94] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Probabilistic analysis of large finite state machines. In *Proc. 31st Design Automation Conference (DAC'94)*, pages 270–275. ACM Press, 1994.

[HMPS96] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on CAD*, 15(12):1479–1493, 1996.

[Hoa85] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HS84] S. Hart and M. Sharir. Probabilistic temporal logics for finite and bounded models. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 1–13, 1984. The full version "Probabilistic Propositional Temporal Logic" is available in *Information and Control*, 70:(2/3):97–155, 1986.

[HS99] H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 244–264. Springer, 1999.

[HSAHB99] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proc. 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 279–288, 1999.

[HSP83] S. Hart, M. Sharir, and A. Pnueli. Termination of probabilistic concurrent programs. *ACM Transactions on Programming Languages and Systems*, 5(3):356–380, 1983.

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proc. International Workshop on Types for Proofs and Programs (TYPES'93)*, volume 806 of *LNCS*, pages 127–165. Springer, 1994.

[IT90] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[Kem96] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(9):615–628, 1996.

[KKNP01] J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In L. de Alfaro and S. Gilmore, editors, *Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01)*, volume 2165 of *LNCS*, pages 23–38. Springer, 2001.

[KMNP02] M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A symbolic out-of-core solution method for Markov models. In *Proc. Workshop on Parallel and Distributed Model Checking (PDMC'02)*, volume 68.4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[KNP02a] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[KNP02b] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[KNPS99] M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic systems using MTBDDs and Simplex.

Technical Report CSR-99-1, School of Computer Science, University of Birmingham, 1999.

[KNS01] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. Springer, 2001.

[KNS02] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Special Issue of Formal Aspects of Computing*, 2002. To appear.

[KR97] T. Kropf and J. Ruf. Using MTBDDs for discrete timed symbolic model checking. In *Proc. 1997 European Design and Test Conference (ED&TC'97)*, pages 182–187. IEEE Computer Society Press, 1997.

[KSK66] J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. D. Van Nostrand Company, 1966.

[Lee59] C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[LPV96] Y.-T. Lai, M. Pedram, and S. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Transactions on Computers*, 45(2):247–255, 1996.

[LR81] D. Lehmann and M. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.

[LS82] D. Lehmann and S. Shelah. Reasoning with time and chance. *Information and Control*, 53(3):165–198, 1982.

[LSS94] N. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proc. 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, pages 314–323. ACM Press, 1994.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Min00] A. Miner. *Data Structures for the Analysis of Large Structured Markov Chains*. PhD thesis, Department of Computer Science, College of William & Mary, Virginia, 2000.

[Min01] A. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In R. German and B. Haverkort, editors, *Proc. 9th International Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110. IEEE Computer Society Press, 2001.

[NPK+02] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Formal analysis and validation of continuous time Markov chain based system level power management strategies. In W. Rosenstiel, editor, *Proc. 7th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02)*, pages 45–50. OmniPress, 2002.

[Pla85] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, volume 13(2) of *Performance Evaluation Review*, pages 147–153, 1985.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society Press, 1977.

[Pnu83] A. Pnueli. On the extremely fair treatment of probabilstic algorithms. In *Proc. 15th Annual ACM Symposium on Theory of Computing*, pages 278–290, 1983.

[Pri] PRISM web site. www.cs.bham.ac.uk/~dxp/prism.

[PZ86] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[Rab82] M. Rabin. N-process mutual exclusion with bounded waiting by $4 \log_2 N$-valued shared variable. *Journal of Computer and System Sciences*, 25(1):66–75, 1982.

[RK98] J. Ruf and T. Kropf. Using MTBDDs for composition and model checking of real-time systems. In G. Gopalakrishnan and P. Windley, editors, *Proc. Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 185–202. Springer, 1998.

[Ros98] W. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall, 1998.

[SAHB00] R. St-Aubin, J. Hoey, and C. Boutilier. APRICODD: Approximate policy construction using decision diagrams. In T. Leen, T. Dietterich, and V. Tresp, editors, *Proc. Advances in Neural Information Processing Systems (NIPS 2000)*, pages 1089–1095. MIT Press, 2000.

[Shm02] V. Shmatikov. Probabilistic analysis of anonymity. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 119–128. IEEE Computer Society Press, 2002.

[Sie99] M. Siegle. Compositional representation and reduction of stochastic labelled transition systems based on decision node BDDs. In D. Baum, N. Mueller, and R. Roedler, editors, *Proc. Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen (MMB'99)*, pages 173–185. VDE Verlag, 1999.

[SL94] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In B. Jonsson and J. Parrow, editors, *Proc. 5th International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 481–496. Springer, 1994.

[Som97] F. Somenzi. CUDD: Colorado University decision diagram package. Public software, Colorado Univeristy, Boulder, http://vlsi.colorado.edu/~fabio/, 1997.

[Ste94] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton, 1994.

[SV99] M. Stoelinga and F. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 53–74. Springer, 1999.

[Taf94] P. Tafertshofer. Factored edge-valued binary decision diagrams and their application to matrix representation and manipulation. Master's thesis, Institute of Electronic Design Automation, Technical University of Munich, 1994.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In K. Ng, P. Raghavan, N. Balasubramanian, and F. Chin, editors, *Proc. 4th International Symposium on Algorithms and Computation (ISAAC'93)*, volume 762 of *LNCS*, pages 389–398. Springer, 1993.

[TP97] P. Tafertshofer and M. Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2/3):137–164, 1997.

[Var85] M. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 327–338. IEEE Computer Society Press, 1985.

[VPL96] S. Vrudhula, M. Pedram, and Y.-T. Lai. Edge-valued binary decision diagrams. In T. Sasao and M. Fujita, editors, *Representations of Discrete Functions*, pages 109–132. Kluwer Academic Publishers, 1996.

[XB97] A. Xie and A. Beerel. Symbolic techniques for performance analysis of timed circuits based on average time separation of events. In *Proc. 3rd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'97)*, pages 64–75. IEEE Computer Society Press, 1997.

[YBO+98] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In G. Gopalakrishnan and P. Windley, editors, *Proc. Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 255–289. Springer, 1998.