Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach*

Marta Kwiatkowska, Gethin Norman, David Parker

School of Computer Science, University of Birmingham,
Birmingham, B15 2TT, United Kingdom
e-mail: {mzk,gxn,dxp}@cs.bham.ac.uk

The date of receipt and acceptance will be inserted by the editor

Abstract. In this paper we present efficient symbolic techniques for probabilistic model checking. These have been implemented in PRISM, a tool for the analysis of probabilistic models such as discrete-time Markov chains, continuous-time Markov chains and Markov decision processes using specifications in the probabilistic temporal logics PCTL and CSL. Motivated by the success of model checkers such as SMV, which use BDDs (binary decision diagrams), we have developed an implementation of PCTL and CSL model checking based on MTBDDs (multi-terminal BDDs) and BDDs. Existing work in this direction has been hindered by the generally poor performance of MTBDD-based numerical computation, which is often substantially slower than explicit methods using sparse matrices. The focus of this paper is a novel hybrid technique which combines aspects of symbolic and explicit approaches to overcome these performance problems. For typical examples, we achieve a dramatic improvement over the purely symbolic approach. In addition, thanks to the compact model representation using MTBDDs, we can verify systems an order of magnitude larger than with sparse matrices, whilst almost matching or even beating them for speed.

1 Introduction

In the design and analysis of software and hardware systems it is often desirable or even necessary to include probabilistic aspects of a system's behaviour. Examples include representing unreliable or unpredictable behaviour in fault-tolerant systems; deriving efficient algorithms by using electronic coin flipping in decision making; and modelling the arrivals and departures of calls in a wireless cell.

Probabilistic model checking refers to a range of techniques for calculating the likelihood of the occurrence of certain events during the execution of systems which exhibit such behaviour. One first constructs a model of the system, defining the set of possible states that it can be in and the likelihood that transitions will occur between these states. Desirable or required properties of the system such as "shutdown occurs with probability 0.01 or less" and "the video frame will be delivered within 5ms with probability 0.97 or greater" can be expressed in probabilistic temporal logics. These specifications can then be automatically verified by a probabilistic model checker.

Motivated by the success of *symbolic* model checkers, such as SMV [55] which use BDDs (binary decision diagrams) [15], we have developed a symbolic probabilistic model checker, PRISM [51,1]. In the non-probabilistic setting, model checking involves manipulation of state transition systems and sets of states, both of which can be represented naturally as BDDs, often very compactly [18]. In the probabilistic case, since real-valued matrices and vectors are required, BDDs alone are insufficient, and hence we also use MTBDDs (multi-terminal binary decision diagrams) [23,5], a natural extension of BDDs for representing real-valued functions.

The use of MTBDDs for the analysis of probabilistic models has been studied extensively in the literature [35,36,63,7,6,43,9,52,31,26,46,53] and it has been demonstrated that it is feasible to construct and compute the reachable state space of extremely large, structured, probabilistic models in this way. In these cases, it is often also possible to verify *qualitative* properties, where model checking reduces to reachability-based analysis. For example, in [31], systems with over 10^{30} states have been verified.

* Supported in part by EPSRC grants GR/M04617, GR/N22960 and GR/S11107 and MathFIT studentship for David Parker.

Model checking *quantitative* properties, on the other hand, involves numerical computation. In some cases, such as in [53], MTBDDs have been very successful, being applied to systems with over 10^{10} states. Often, however, it turns out that such computation is slow or infeasible. By way of comparison, the equivalent numerical computation routines implemented explicitly using sparse matrices are often orders of magnitude faster.

Here, we present a novel *hybrid* approach which uses an extension of the MTBDD data structure and borrows ideas from explicit techniques to overcome these performance problems. We include experimental data which demonstrates that, using this hybrid approach, we can achieve speeds which are orders of magnitude faster than MTBDDs. It is possible to, in general, almost match the speed of sparse matrices and, in some cases, outperform them, whilst maintaining considerable space savings.

The outline of this paper is as follows. Section 2 gives an overview of probabilistic model checking, introducing the probabilistic models and temporal logics we consider. In Section 3, we describe our tool, PRISM, which implements the model checking of these models and logics. We then move on to discuss the tool's implementation. Section 4 introduces the MTBDD data structure and explains how it can be used to represent and analyse probabilistic models. We identify a number of performance problems in this implementation and, in Section 5, describe how we overcome these limitations. In Section 6, we present experimental results to judge the performance of our technique and in Section 7, we discuss how it relates to existing work. Section 8 concludes the paper.

2 Probabilistic Model Checking

In this section we briefly summarise the models and temporal logics that our implementation of probabilistic model checking supports. The simplest probabilistic model is the *discrete-time Markov chain* (DTMC), defined by a set of states S and a transition probability matrix $\mathbf{P} : S \times S \rightarrow [0,1]$, where $\mathbf{P}(s,s')$ is the probability of making a transition from one state s to another state s'. The probabilities from state s must sum up to 1, i.e. $\sum_{s'} \mathbf{P}(s,s') = 1$.

Markov decision processes (MDPs) extend DTMCs by allowing both probabilistic and nondeterministic behaviour. More formally, in any state there is a nondeterministic choice between a number of discrete probability distributions over states. Nondeterminism enables the modelling of asynchronous parallel composition of probabilistic systems. It also permits under-specification of certain aspects of a system.

A *continuous-time Markov chain* (CTMC), on the other hand, is defined by a set of states S and a transition rate matrix $\mathbf{R} : S \times S \rightarrow \mathbb{R}_{\geq 0}$, where $\mathbf{R}(s,s')$ is the rate of making a transition from state s to s'. The interpretation is that the probability of moving from s to s' within t time units (for positive, real-valued t) is $1 - e^{-\mathbf{R}(s,s')\cdot t}$.

As specification formalisms, we use probabilistic extensions of the temporal logic CTL. In particular, we use PCTL [37,12,10] in the context of DTMCs and MDPs and CSL [4,9] in the context of CTMCs.

PCTL allows us to express properties of the form "under any scheduling of processes, the probability that event A occurs is at least p (at most p)". By way of illustration, we consider the asynchronous randomised leader election protocol of Itai and Rodeh [45] which gives rise to an MDP. In this algorithm, the processors of an asynchronous ring make random choices based on coin tosses in an attempt to elect a leader. We use the atomic proposition *leader* to label states in which a leader has been elected. Examples of properties we would wish to verify can be expressed in PCTL as follows:

- $\mathcal{P}_{\geq 1}[\Diamond\, \mathit{leader}]$ - "under any scheduling, a leader is eventually elected with probability 1".
- $\mathcal{P}_{\leq 0.5}[\Diamond^{\leq k}\, \mathit{leader}]$ - "under any scheduling, the probability of electing a leader within k discrete time-steps is at most 0.5".

In [27,28], an extension of PCTL, called pTL, is introduced which additionally allows the specification of *expected time* and *average time* properties. Returning to the leader election protocol given above, an example of such a property is:

- $\mathcal{D}_{\leq 10}[\mathit{leader}]$ - "under any scheduling, the expected number of steps until leader election is at most 10".

The logic CSL includes the means to express both transient and steady-state performance measures of CTMCs. Transient properties describe the system at a fixed, real-valued time-instant t, whereas steady-state properties refer to the behaviour of a system in the "long run". For example, consider a queueing system where the atomic proposition *full* labels states where the queue is full. CSL then allows us to express properties such as:

- $\mathcal{P}_{\leq 0.01}[\Diamond^{\leq t}\, \mathit{full}]$ - "the probability that the queue becomes full within t time units is at most 0.01"
- $\mathcal{S}_{\geq 0.98}[\neg \mathit{full}]$ - "in the long run, the probability that the queue is not full is at least 0.98".

Model checking algorithms for PCTL have been introduced in [37,12] and extended in [10,6] to include fairness. The case for pTL is dealt with in [27,29,30]. An algorithm for CSL was first proposed in [9] and has since been improved in [8,46]. The model checking algorithms for all logics reduce to a combination of reachability-based computation and numerical calculation. The former may be used, for example, to determine states which satisfy a temporal logic formula with probability exactly 0 or 1. The latter is needed where exact probabilities must be determined. Here, the computation required varies. For DTMCs, this usually entails solution of a

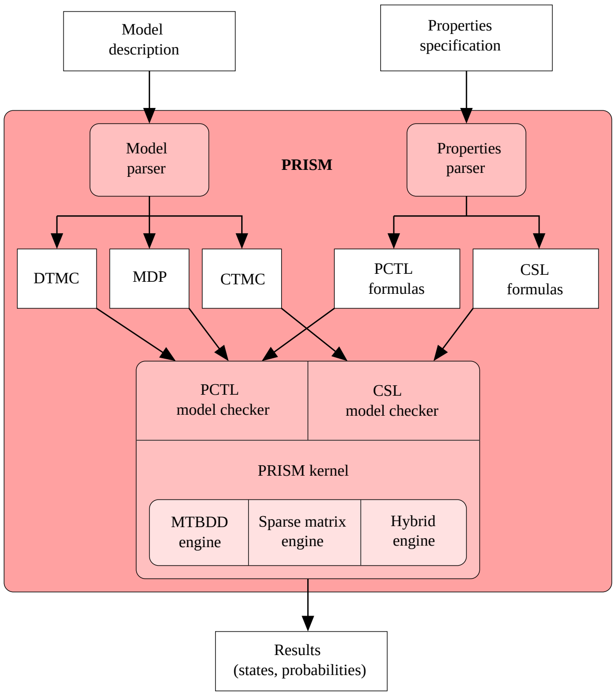


Fig. 1. PRISM system architecture

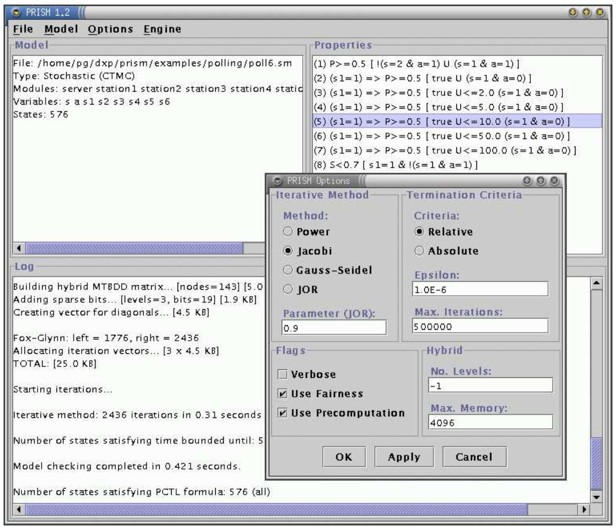


Fig. 2. The PRISM graphical user interface

linear equation system, whereas for MDPs a linear optimisation problem must be solved. For CTMCs, either solution of a linear equation system or an iterative technique known as uniformisation is performed. Since the size of the problem to be solved is often large, *direct methods*, such as Gaussian elimination (for linear equation systems) or Simplex (for linear optimisation problems), are usually impractical. Instead, we opt to use *iterative methods* which approximate the solution up to some specified accuracy.

3 PRISM

PRISM [51,1] is a model checking tool developed at the University of Birmingham which supports verification of the models and logics described in the previous section. The tool takes as input a description of a probabilistic system written in the PRISM language, a variant of the Reactive Modules formalism of Alur and Henzinger [2]. It first constructs the model from this description (either a DTMC, an MDP or a CTMC), computes the set of reachable states, and identifies any deadlock states. PRISM accepts specifications in either the logic PCTL or CSL depending on the model type. It then performs model checking to determine which states of the model satisfy each specification. Figure 1 illustrates the structure of the tool and Figure 2 shows a screen-shot of the graphical user interface. A text-based, command line version is also available.

The underlying data structures in PRISM are BDDs and MTBDDs. For numerical computation, however, the tool provides three distinct engines which can be used interchangeably. The first is a pure MTBDD-based implementation, as described in Section 4; the second is a conventional explicit version using sparse matrices, implemented for comparison purposes; the third uses the hybrid approach presented in this paper.

PRISM is written in a combination of Java and C++ and uses CUDD [60], a publicly available BDD/MTBDD library developed at the University of Colorado at Boulder. The high-level parts of the tool, such as the user interface and parsers, are written in Java. The low-level libraries are written in C++. The tool and its source code are available for download from the PRISM web site [1]. Further information about the tool and a large number of case studies to which it has been applied are also available here.

4 An MTBDD Implementation

This section describes how probabilistic model checking can be implemented using MTBDDs. We begin by introducing the data structure, then explain how it can be used to produce a compact representation for probabilistic models, and finally show how techniques to analyse these models can be implemented. We also summarise the performance of these symbolic techniques.

4.1 Introduction to MTBDDs

An MTBDD M is a rooted, directed acyclic graph associated with a set of ordered, Boolean variables $x_1 < \cdots < x_n$. It represents a function $f_{\mathsf{M}}(x_1, \ldots, x_n) : \mathbb{B}^n \to \mathbb{R}$ over these variables. The graph contains two types of nodes: *non-terminal* and *terminal*. A non-terminal node m is labelled by a variable $var(\mathsf{m}) \in \{x_1, \ldots, x_n\}$ and has two children, *then*(m) and *else*(m). A terminal node m is labelled by a real number *val*(m).

We impose the Boolean variable ordering $<$ onto the graph by requiring that a child m′ of a non-terminal node m is either terminal or is non-terminal and satisfies $var(\mathsf{m}) < var(\mathsf{m}')$. The value of $f_{\mathsf{M}}(x_1, \ldots, x_n)$, the function which the MTBDD represents, is determined by traversing M from the root node, and at each subsequent node m taking the edge to *then*(m) or *else*(m) if *var*(m) is 1 or 0 respectively. Note that a BDD is merely an MTBDD with the restriction that the labels on terminal nodes can only be 1 or 0.

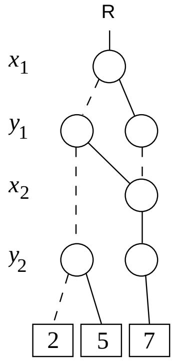
MTBDDs are efficient because they are stored in reduced form. If nodes m and m′ are identical (that is $var(\mathsf{m}) = var(\mathsf{m}')$, $then(\mathsf{m}) = then(\mathsf{m}')$ and $else(\mathsf{m}) = else(\mathsf{m}')$ for non-terminals or $val(\mathsf{m}) = val(\mathsf{m}')$ for terminals), then only one copy is stored. Furthermore, if a node m is *redundant*, i.e. satisfies $then(\mathsf{m}) = else(\mathsf{m})$, it is removed and any incoming edges are redirected to its unique child. These reductions mean that MTBDD representations of functions which exhibit regularity or redundancy can be extremely compact. There is, however, another important advantage. With these two rules in place, MTBDDs can be shown to be *canonical*, meaning that for a given variable ordering, there is a one-to-one correspondence between MTBDDs and the functions which they represent. One important implication of this is that it is very efficient to compare two MTBDDs for equality. This is useful, for example, when implementing a cache of previously performed MTBDD operations.

Another important characteristic of MTBDDs, from a practical point of view, is that their size (number of nodes) is heavily dependent on the ordering of their Boolean variables. Although in the worst case the size of an MTBDD representation is exponential and the problem of deriving the optimal ordering for a given MTBDD is an NP-hard problem [62,13], through application of heuristics, MTBDDs can provide extremely compact storage for structured, real-valued functions.

4.2 MTBDD Representation of Probabilistic Models

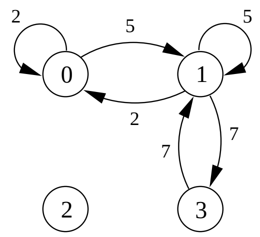
From their inception in [23,5], MTBDDs have been used to encode real-valued vectors and matrices. An MTBDD v over variables $(x_1, \ldots, x_n)$ represents a function $f_{\mathsf{v}} : \mathbb{B}^n \to \mathbb{R}$. Observe that a real vector $\mathbf{v}$ of length 2^n is simply a mapping from $\{1, \ldots, 2^n\}$ to the reals $\mathbb{R}$. Hence, if we decide upon an encoding of $\{1, \ldots, 2^n\}$ in terms of $\{x_1, \ldots, x_n\}$ (for example the standard binary encoding), then an MTBDD v can represent $\mathbf{v}$.

In a similar fashion, we can consider a square matrix $\mathbf{M}$ of size 2^n by 2^n to be a mapping from $\{1, \ldots, 2^n\} \times \{1, \ldots, 2^n\}$ to $\mathbb{R}$. Taking Boolean variables $\{x_1, \ldots, x_n\}$ to range over row indices and $\{y_1, \ldots, y_n\}$ to range over column indices, we can represent $\mathbf{M}$ by an MTBDD over $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$. DTMCs and CTMCs are described by such matrices, and hence are also straightforward to represent as MTBDDs.



x_1	y_1	x_2	y_2	f_{R}	Entry of $\mathbf{R}$
0	0	0	0	2	$(0,0) = 2$
0	0	0	1	5	$(0,1) = 5$
0	0	1	0	2	$(1,0) = 2$
0	0	1	1	5	$(1,1) = 5$
0	1	1	1	7	$(1,3) = 7$
1	0	1	1	7	$(3,1) = 7$

Fig. 3. An MTBDD R and the function it represents



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

Fig. 4. A CTMC and its rate matrix $\mathbf{R}$

Figure 3 gives an example of an MTBDD R over four Boolean variables, x_1, y_1, x_2 and y_2. In our notation, non-terminal nodes are drawn as circles and terminal nodes as squares. Non-terminal nodes are grouped into levels according to their variable labelling, which is displayed at the left-hand end of each level. The downward *then* and *else* edges from each node are drawn as solid and dashed lines respectively. For clarity, we omit the zero terminal node and any edges which lead directly to it. Figure 3 also includes a table showing the function f_{R} which the MTBDD R represents. Consider, for example, the valuation $(x_1, y_1, x_2, y_2) = (0, 1, 1, 1)$. Tracing the appropriate path through the MTBDD, we can see that the resulting value is 7.

In fact, the MTBDD R in Figure 3 represents the transition rate matrix $\mathbf{R}$ of a CTMC. This matrix and the associated CTMC are shown in Figure 4. Note that the CTMC includes one state which is unreachable. This will be important when we reuse the example later in the paper. The corresponding row and column of $\mathbf{R}$ are filled with zeros, but to emphasise that these are unreachable, the entries are marked as '$-$' rather than '0'.

The right-most column of the table in Figure 3 illustrates exactly how R corresponds to $\mathbf{R}$. As described above, we use variables x_1 and x_2 to encode row indices and y_1 and y_2 to encode column indices. In both cases, we use the standard binary representation of integers. By way of example, for entry $(1,3)$ of $\mathbf{R}$, the row index 1 is encoded as $(x_1, x_2) = (0, 1)$ and the column index 3 is encoded as $(y_1, y_2) = (1, 1)$. Tracing a path down the

MTBDD using these values leads to the terminal node 7 which is the value of the entry.

Observe that, in R, the variables for rows and columns are ordered alternately. This is a common variable ordering heuristic for minimising the size of the MTBDD representation of a transition matrix. Note also that the MTBDD is an inherently recursive data structure: each node of the data structure is itself an MTBDD. In terms of matrices, this means that each node of an MTBDD represents a submatrix of the overall matrix being represented. Intuitively, this is how MTBDDs can provide a compact representation: in cases where submatrices are repeated, this can be exploited by the reduced nature of the data structure.

The process for the representation of MDPs is more complex since the nondeterminism must also be encoded. However, if the maximum number of nondeterministic choices in any state is bounded by 2^k for some integer k, we can view the MDP as a function from $\{1,\ldots,2^n\}\times\{1,\ldots,2^k\}\times\{1,\ldots,2^n\}$ to $\mathbb{R}$. By adding k extra Boolean variables to encode this third index, we can represent the MDP as an MTBDD. In this way, an MTBDD can be considered to be represented by a *set* of matrices.

In order to produce compact MTBDD representations of probabilistic models, it is important to consider *how* they are encoded. This issue was first addressed by Hermanns et al. in [43], which presents a number of heuristics for this purpose. The most important conclusion is that it is essential to exploit *structure* and *regularity* in the model, typically that derived from its high-level description. For example, if the formalism used describes the model in a compositional fashion, it is advisable to first encode individual components of the model as MTBDDs and then combine them in a structured fashion. In practice, this can be accomplished by performing a direct translation from the high-level description of the model into an MTBDD.

Examples of this for queueing network and process algebra descriptions can be found in [43,48,42]. The translation of the PRISM language, used to describe models in our tool, is considered in [31,57]. Heuristics have also been developed to select efficient variable orderings in these cases [43,42,57]. In practice, these techniques have been used to construct and represent extremely large probabilistic models, e.g. [43,53,54]. We include some experimental data in Section 6 which illustrates this. Often, a direct translation from a high-level formalism also results in the construction process being relatively fast. It should be noted, though, that it also often introduces unreachable states. These must be determined through reachability analysis (via a simple BDD fixpoint calculation) and then removed.

4.3 Probabilistic Model Checking with MTBDDs

Once a model's MTBDD representation has been constructed, it can be analysed, for example using PCTL or CSL model checking. This analysis can be carried out symbolically, using MTBDDs. As described in Section 2, the process usually comprises two types of computation: graph-based analysis using reachability techniques and numerical calculation. The former can be performed with BDDs and is at the heart of non-probabilistic symbolic model checking which has been proven to be very successful [18,55]. The latter can be implemented with MTBDDs, as we will now see.

Fortunately, all the numerical problems we need to solve can be implemented as iterative methods. For solution of a linear equation system, standard techniques such as the Jacobi and Gauss-Seidel methods are available. For the linear optimisation problems required for PCTL model checking of MDPs, iterative methods based on dynamic programming can be used. Other techniques needed for transient analysis of CTMCs and the computation of expected costs or rewards can also be performed through iterative methods.

The typical structure of an iterative solution method is as follows. A solution vector, containing an approximation to the values being computed, is repeatedly updated until it us judged to have converged. An example of a check for convergence is to note when the maximum difference between vector elements of successive iterations falls below some threshold. At each iteration, the operations performed to compute the updated vector use both the vector from the previous iteration and the matrix representing the probabilistic model. In the majority of cases, the bulk of this work reduces to performing a matrix-vector multiplication. As seen above, MTBDDs can easily represent both matrices and vectors. Furthermore, efficient algorithms to perform matrix multiplications using the MTBDD data structure have been developed [24,5,23]. This constitutes the basis for a wide range of MTBDD implementations of numerical iterative methods [35,36,63,7,6,43,31,46,53].

Although we have already dismissed the use of direct methods because they do not scale well to large problems, it is interesting to note that they are poorly suited to symbolic implementation anyway. MTBDD-based versions of Gaussian elimination and Simplex have been presented in [5] and [52], respectively, and found to perform badly. The reason for this is that they rely on modifying the model representation through operations on individual rows, columns or elements. This is not only slow, but leads to a loss in regularity and a subsequent explosion in MTBDD size.

The results of the implementation of probabilistic model checking with MTBDDs can be summarised as follows. Firstly, there is a clear distinction between the two different types of computations required for the process. Those based on reachability, which are sufficient for model checking qualitative properties, can be implemented efficiently with BDDs, as shown for example in [31]. On the other hand, numerical computation, which is required for checking of quantitative properties, is more unpredictable. This is the problem we focus on in this paper.

There have been instances where MTBDD-based numerical computation proves to be extremely efficient. For example, [53] presents results for the analysis of the coin protocol from Aspnes and Herlihy's randomised consensus algorithm [3]. This includes model checking of MDPs with more than 10^{10} states. In [54], an analysis of the IEEE 1394 FireWire root contention protocol involved MDPs with more than 170 million states. In both cases, it would be impossible even to construct and store an explicit representation of the models, given the same hardware constraints.

In general though, the symbolic implementation of numerical iterative methods is far from efficient. The problem is that, despite a compact MTBDD representation of the model, the MTBDD representation of the solution vector tends to grow extremely large. This is due to a lack of regularity in the vector as the computation progresses and is worsened by an accompanying increase in the number of distinct values it contains. These results have been observed in e.g. [35,36,31,46]. By contrast, in explicit implementations, such as those based on sparse matrices, solution vectors are stored in arrays. These remain a fixed size and it is quick and easy to access and modify their contents. Hence, sparse matrix based techniques are usually much faster than their symbolic counterparts. However, since the probabilistic model is also stored explicitly, application to large examples is often limited by memory constraints.

5 A Hybrid Approach

We now present a method to overcome the inefficiencies with MTBDDs outlined in the previous section. The approach taken here is to a use a hybrid of the two techniques: symbolic and explicit. We store the transition matrix in an MTBDD but use a full array for the iteration vector. We will consider the problem of performing a matrix-vector multiplication using these two contrasting data structures. As seen previously, this is then sufficient to allow us to implement a range of numerical computation techniques, as required for probabilistic model checking. In particular, this is directly applicable to PCTL model checking of DTMCs and CSL model checking of CTMCs, provided that linear equation systems are solved using the Jacobi or JOR (Jacobi with over-relaxation) methods. In fact, the techniques can also be extended to MDP-based model checking [57].

5.1 The Basic Algorithm

In the remainder of this section, we will describe how a matrix-vector multiplication can be carried out when the matrix is stored in an MTBDD and the vector in an array. Essentially, we will emulate the operations that would be carried out in an explicit approach, e.g. using a sparse matrix data structure. The overall process reduces to the extraction of all the entries of the matrix, each of which is needed exactly once to compute the multiplication. The key difference in our approach is that we need to extract them from an MTBDD rather than a sparse matrix. Crucially, we note that, for matrix-vector multiplication, the order in which the entries are extracted is not important. This means that we can can proceed via a recursive traversal of the MTBDD: it does not matter that the entries will be in an essentially random order, rather than row-by-row (or column-by-column) as with a sparse matrix.

A single matrix entry comprises three pieces of information: its row index, column index and value. A recursive traversal of an MTBDD essentially enumerates every possible path from the root node of the data structure to a terminal node. Each of these paths corresponds to a matrix entry. The value of this entry is equal to the labelling of the terminal node at the end of the path. The row and column index can be determined as follows. Let us assume that the matrix indices were encoded using the standard binary representation of integers. Since we know the path that was taken, i.e. whether the *then* or *else* edge was taken at each step, we can deduce the binary representation of the indices and hence convert them to decimal.

In terms of the recursive traversal algorithm, this is achieved by maintaining a running total for both the row and column indices. At each recursive call, if the *then* edge is taken from the current node m, the appropriate power of 2 is added to the corresponding index: the row index if $var(\mathsf{m})$ is an x_i variable and the column index if $var(\mathsf{m})$ is a y_i variable. Alternatively, we can view this as a truly recursive problem. Each call to the algorithm computes the local entries for the submatrix which that node represents. The actual indices are computed correctly by adding the appropriate offsets at each level. This offset will be a power of 2 corresponding to the size of the submatrices at the level of recursion below.

5.2 Offset-Labelled MTBDDs

Unfortunately there is a drawback to using the simple algorithm outlined in the previous section. Recall from Section 4.2 that generating an efficient, structured MTBDD representation of a probabilistic model typically results in the inclusion of a number of unreachable states in the encoding. Performing matrix-vector multiplication as just described on such an MTBDD would require the vector array to store entries for all states, including those that are unreachable. The number of unreachable states is potentially very large, in some cases orders of magnitude larger than the reachable portion.

This puts unacceptable limits on the size of problem which we can handle.

The solution we adopt is to compute the row and column index in terms of reachable states only. This can be integrated into our existing recursive MTBDD traversal algorithm described above. As before, we will keep track of the indices during traversal by adding offsets at each node. Here, though, the offset will be equal to the number of rows or columns in the next level of recursion's submatrix which correspond to reachable states, rather than the total number (a power of 2) as before. To facilitate this process, we a use a modified version of the MTBDD data structure, which we call *offset-labelled MTBDDs*.

An offset-labelled MTBDD is essentially an MTBDD but with two important differences. Firstly, each node of the data structure is labelled with an integer offset. These will be the values added to the row and column indices during traversal. Secondly, we modify the reduction rules that are applied to the data structure. Recall from Section 4.1 that there are two types of reduction: merging of identical nodes and removal of redundant nodes. In an offset-labelled MTBDD, the second type of reduction is not performed. In the context of BDDs, of which MTBDDs are an extension, this variant is known as a *quasi-reduced* BDD. The reason we do this is that, during the computation of indices, offsets may need to be added at any level of the MTBDD. Since the offsets are stored on the nodes, we need nodes to be present at every level. The exception to this rule is that we *do* allow edges to skip levels if they lead directly to the zero terminal node. This is because we are only ever interested in extracting the non-zero entries of the matrix.

We also relax the constraints on the first type of reduction (merging of identical nodes), making it optional. In most cases, this reduction will still be used in order to minimise the size of the data structure. There are some cases, though, as we will see, where it is important not to do this. Removing instances of this type of reduction means that the data structure is no longer canonical. Fortunately, for our usage of offset-labelled MTBDDs, this is not a problem. Usually, the reason for maintaining canonicity is for the efficient implementation of lookups in a cache of previously performed operations. This is crucial for efficient manipulation of MTBDDs. Here though, we will never actually need to manipulate the data structure: we will construct an instance of it (from an existing MTBDD), use it for numerical computations (via traversals which do not involve manipulating the graph), and then discard it.

Figure 5 gives the recursive algorithm for traversal of an offset-labelled MTBDD. At the top-level, this would be called as follows:

$$\textsc{TraverseRec}(\mathsf{root}, 0, 0)$$

where root is the root node of the MTBDD. The base cases are when the current MTBDD node m is a terminal node. If m is the zero terminal, there are no more non-zero entries in this portion of the matrix. If m is a non-zero terminal, a matrix entry has been identified. The variables *row* and *col* are used to keep track of row and column indices. The offset label on each node m, used to compute these indices, is denoted *offset*(m). The row and column index of an entry are computed (independently) by summing the offsets on x_i and y_i labelled nodes, respectively, from which the *then* edge was taken.

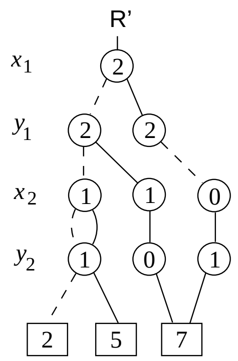
```
TraverseRec(m, row, col)
if (m is the zero terminal node) then return
elseif (m is a non-zero terminal node) then
    found matrix element (row, col) = val(m)
elseif (m is a row node) then
    TraverseRec(else(m), row, col)
    TraverseRec(then(m), row + offset(m), col)
elseif (m is a column node) then
    TraverseRec(else(m), row, col)
    TraverseRec(then(m), row, col + offset(m))
endif
```

Fig. 5. Offset-labelled MTBDD traversal algorithm

To illustrate the whole process more clearly, we now present a simple example. Figure 6 shows the offset-labelled MTBDD representing the CTMC from Figure 4. The table to its right explains how the information is encoded. Each row corresponds to a single matrix entry (i.e. a transition of the CTMC). These are given in the order that they would be extracted by the recursive traversal algorithm. The first five columns describe the path taken through the MTBDD. The next four columns give the node offsets along this path. The last column gives the resulting matrix entry. For example, the path 0111, i.e. where $(x_1, y_1, x_2, y_2) = (0, 1, 1, 1)$, leads to the 7 terminal node. Since the *then* edge was taken at the x_2 level but not the x_1 level, the row index is equal to the offset on the x_2 node, 1. For the y_1 and y_2 levels, the *then* edge was taken in both cases. Hence, the column index is the sum of the two offsets, $2+0 = 2$. The matrix entry is therefore $(1, 2) = 7$. Note how in the previous case, in Figure 3, this entry was computed as $(1, 3)$. Here, the unreachable state has been taken into account.

A closer comparison of the offset-labelled MTBDD in Figure 6 and the MTBDD in Figure 3 highlights the other differences between the two data structures. Note the extra x_2 node on the left hand side in Figure 6. This is because we do not allow skipped levels in offset-labelled MTBDDs. Secondly, observe that the bottom two nodes of the two paths to the 7 terminal node in Figure 6 are identical, except for the offset-labelling. Hence, in the original MTBDD, these nodes were merged together. This is why we are less stringent about the first reduction rule.

The explanation is that some nodes in an MTBDD can be reached along several different paths. These shared nodes correspond to repeated sub-matrices in the overall



Path					Offsets				Entry of **R**
x_1	y_1	x_2	y_2	$f_{R'}$	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	$(2,1)=7$

Fig. 6. The offset-labelled MTBDD representing the CTMC in Figure 4

matrix. Consider the matrix in Figure 4 and its MTBDD representation in Figure 3. The bottom-left and top-right quadrants of the matrix are identical (since rows and columns of unreachable states are filled with zeros). This is reflected by the fact that the x_2 node in the MTBDD has two incoming edges. The two identical sub-matrices do not, however, share the same pattern of reachable states. This means that there is a clash as to which offset should label the node. In Figure 6, this has been resolved by duplicating the node, labelling each copy with a different offset.

Finally, we briefly describe how offset-labelled MTBDDs are constructed. A more detailed presentation of the process, including full algorithms, can be found in [57]. An offset-labelled MTBDD is only needed for the numerical solution phase of model checking. It is constructed from an existing MTBDD before this phase begins. The first step is to remove instances of skipped levels in the original MTBDD. This is done simply by traversing the data structure, comparing the variables on consecutive pairs of nodes and inserting additional nodes where required. Secondly, we traverse the new MTBDD and add offset labels onto the nodes. These offsets are computed beforehand using a BDD representation of the reachable state space. During the labelling process, possible clashes of label are checked for and, where necessary, additional nodes with the correct offset are inserted. In Section 6, we include empirical results giving the time required for the construction process and the amount of additional memory usage which it incurs.

5.3 Optimising the Hybrid Approach

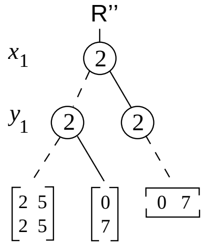
We can optimise the method described in the previous section considerably via a form of *caching*. MTBDDs exploit structure and regularity in the probabilistic model being analysed, often resulting in a significant space saving. This is achieved by merging identical nodes, representing identical sub-matrices. During a single traversal of the data structure, however, each of these shared nodes will be visited several times (as many times as the sub-matrix occurs in the overall matrix) and the entries of the sub-matrix will be extracted separately every time. Furthermore, a typical instance of numerical computation will comprise many iterations, each of which requires one such traversal. Hence, by storing and reusing the results of the traversal process, we can achieve a significant speed-up in traversal time.

Rather than store these results in a cache, which would need to be searched through frequently, we simply attach the information directly to MTBDD nodes. We select some subset of the nodes, build explicit (sparse matrix) representations of their associated sub-matrices and attach them to the MTBDD. When these nodes are reached during each traversal, the entries of their associated submatrices can be extracted extremely quickly.

There is an obvious trade-off here between the additional space required to store the data and the resulting improvement in speed. The space required and time improvement both depend on how many nodes (and which ones) we attach matrices to. A simple yet effective policy is to select a single layer of the MTBDD and compute matrices for all nodes on this level.

In Figure 7, we demonstrate this technique on the running example, replacing all nodes on the x_2 level with the matrices they represent. The table illustrates how the recursive traversal process now functions. At the bottom of the recursion, small, explicit submatrices are obtained. The *local entries* of these matrices are converted to *global entries* of the overall matrix by adding row and column index offsets as before.

In general, selecting a higher level of the MTBDD for which to compute submatrices means that entries of the whole matrix can be extracted in less time but that more memory is required for storage. Our approach is to precompute, for each level, the amount of storage that would be required and then select the highest level possible for which some given memory threshold is not exceeded. In this work, we take this threshold to be 1024 KB. In practice, we find that this optimisation technique produces a marked improvement in traversal speed. In Section 6, we give experimental results to illustrate this.



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

Fig. 7. The modified MTBDD labelled with explicit sub-matrices

5.4 Additional Extensions

More detailed information about the techniques given in this paper can be found in [57]. In addition, two extensions of the basic algorithms described here are included. Firstly, the process is extended to cover model checking of MDPs. In the sections above, we have focused on the problem of extracting entries from a real-valued matrix. This is directly applicable to the numerical computation required for model checking of DTMCs and CTMCs, which are represented by such matrices. Computations on MDPs are complicated by the inclusion of nondeterminism. However, as explained in Section 4.2, an MDP can actually be considered as a set of matrices. Each iteration of the numerical solution required for model checking can then be performed using several matrix-vector multiplications, as opposed to just one for DTMCs and CTMCs.

Secondly, [57] addresses an important limitation of our approach: that the algorithms we have given can only be used for iterative methods based on matrix-vector multiplication. For some tasks, such as transient analysis of CTMCs and PCTL model checking of MDPs, this is not restrictive. In cases where solution of a linear equation system is required, though, iterative numerical methods such as Gauss-Seidel, which generally converge faster, can not be used. This is because such methods require access to individual rows or columns of the matrix. Our approach of recursively traversing the MTBDD in a single pass does not allow this kind of access. One way this situation can be improved is to use a block-based variant of Gauss-Seidel which also provides an improvement in convergence speed, but which is amenable to implementation with MTBDDs. Intuitively, this is because blocks of a matrix which is represented by an MTBDD are easy to access, unlike individual rows or columns.

6 Results

In this section, we present experimental results to compare the performance of the three implementations discussed in this paper: the purely MTBDD-based approach described in Section 4; an equivalent explicit implementation using sparse matrices; and the hybrid approach described in Section 5. All three are available in the PRISM model checker. We also made use of a prototype extension of PRISM which is being developed for the analysis of expected-time properties. All experiments were run on a 450MHz workstation with 1GB of main memory.

We focus on the problem of iterative numerical computation, as required for quantitative probabilistic model checking. The results are for three case studies: a CTMC model of a Kanban manufacturing system [22]; a DTMC model of a randomised self-stabilising algorithm due to Herman [39]; and an MDP model of the coin protocol from Aspnes and Herlihy's randomised consensus algorithm [3]. In each case, we construct models of different sizes by varying a parameter N. This equates to the maximum number of jobs in the Kanban system, and the number of processes in the other two examples. Table 1 shows, for each of the above examples, the model size in terms of number of states and number of transitions as the parameter N varies.

Table 2 gives the memory requirements for storing these models, using four different implementations: MTBDDs, sparse matrices and the hybrid approach, with and without the optimisation described in Section 5.3. An MTBDD requires 20 bytes for each node of the data structure. A sparse matrix requires $4a + 12b$ bytes where a is the number of states and b is the number of transitions. For an MDP, extra memory is required to store information about nondeterministic choices. See [57] for more details. Note that entries with an asterisk in the sparse matrix column were not actually generated because of physical memory limitations. In our implementation, an offset-labelled MTBDD can also be stored using 20 bytes per node. This is because certain information normally present in MTBDDs can be omitted since the data structure is never modified. In the optimised case, we require an extra 4 bytes per node to store pointers to the explicit submatrices. Furthermore, memory for the matrices themselves is needed. As described earlier, this is capped at 1024 KB.

It is easy to see that, by exploiting the structure in these examples, MTBDDs can result in a colossal saving in memory. Furthermore, the memory overhead associated with our hybrid approach remains relatively low. This is despite the fact that we potentially add more nodes to the data structure and store some of the submatrices explicitly.

Model	N	States	Transitions
Kanban system (CTMC)	3	58,400	446,400
	4	454,475	3,979,850
	5	2,546,432	24,460,016
	6	11,261,376	115,708,992
	7	41,644,800	450,455,040
Herman's self-stabilisation (DTMC)	13	8,192	1,594,324
	15	32,768	14,348,908
	17	131,072	129,140,164
	19	524,288	1,162,261,468
	21	2,097,152	10,460,353,204
	23	8,388,608	125,524,238,436
Coin protocol (MDP)	4	43,136	201,056
	6	2,376,448	16,607,040
	8	114,757,632	1,069,599,744
	10	5,179,854,848	60,364,590,080

Table 1. Model statistics

Model	N	Matrix storage (KB)			
		MTBDD	Sparse	Hybrid	Hybrid (optimised)
Kanban system (CTMC)	3	48.3	5,459	58.0	866
	4	95.7	48,414	115	858
	5	123	296,588	148	671
	6	154	1,399,955*	185	944
	7	186	5,441,445*	223	1,033
Herman's self-stabilisation (DTMC)	13	11.7	18,715	19.9	493
	15	15.8	168,279	27.2	658
	17	20.6	1,513,873*	35.6	825
	19	25.9	13,622,300*	45.2	992
	21	31.9	122,590,456*	55.9	456
	23	34.6	1,471,019,937*	63.3	577
Coin protocol (MDP)	4	32.7	1,651	97.0	695
	6	84.2	93,948	348	933
	8	168	4,236,739*	902	1,276
	10	291	37,011,024*	1,922	2,387

* Not actually constructed due to memory limitations

Table 2. Memory requirements

Model	N	Construction time (sec.)		Average time per iteration (sec.)			
		Hybrid	Hybrid (optimised)	MTBDD	Sparse	Hybrid	Hybrid (optimised)
Kanban system (CTMC)	3	0.01	0.03	45.5	0.04	0.33	0.04
	4	0.01	0.10	-	0.44	4.32	0.48
	5	0.02	0.21	-	2.85	24.41	3.09
	6	0.04	0.49	-	-	113	15.6
	7	0.06	0.88	-	-	437	61.9
Herman's self-stabilisation (DTMC)	13	0.01	0.03	2.02	0.15	0.25	0.09
	15	0.01	0.05	13.2	1.40	2.22	0.80
	17	0.05	0.10	78.1	-	20.6	7.26
	19	0.18	0.33	803	-	187	66.2
	21	0.71	3.27	-	-	1,644	525
	23	2.68	35.1	-	-	20,751	7,161
Coin protocol (MDP)	4	0.05	0.08	0.17	0.03	0.07	0.04
	6	1.10	1.16	1.01	1.04	5.58	3.02
	8	-	-	3.17	-	-	-
	10	-	-	8.38	-	-	-

Table 3. Timing statistics

In Table 3, we present timing statistics for model checking on the same three case studies. For the Kanban system, we model check a quantitative CSL property which requires computation of steady-state probabilities. This is done via the solution of a linear equation system using the JOR (Jacobi with over-relaxation) iterative method. For the self-stabilisation protocol, we verify an expected-time property, computing the solution iteratively. For the coin protocol, we verify a quantitative PCTL property which requires solution of a linear optimisation problem. In each case the times presented in the table are the average time per iteration of the numerical method. We also give the time required for construction of the offset-labelled MTBDD for the hybrid approach, with and without optimisation.

The relative performances of the four implementations can be summarised as follows. Concentrating first on the DTMC and CTMC examples, we see that our hybrid approach represents a significant improvement over the conventional MTBDD-based approach. We note also that applying the optimisation described in Section 5.3 makes our approach far more efficient. In comparison to the explicit approach using sparse matrices, we find that we can in one case almost match and in another case beat the time per iteration using our approach. More importantly, thanks to the memory savings afforded by the MTBDD-based model representation we can also handle models an order of magnitude larger.

For the coin protocol model, it is the pure MTBDD approach which is most effective, successfully coping with state spaces of 5 billion states. For the other three implementations, storing even the solution vector is impossible here. In this case, both the model and vectors to be stored exhibit a large amount of structure which can be exploited. This is helped by the fact that the model, like many taken from randomised distributed algorithms, contains only a small number of distinct probabilities.

Finally, we note that the time required to construct offset-labelled MTBDDs is small in all cases, and is negligible in comparison to the overall solution process. For details of further case studies to which these techniques have been applied, see the PRISM web site [1].

7 Related Work

We are aware of three other probabilistic model checkers: ProbVerus [38], a prototype tool which supports model checking of DTMCs using a subset of PCTL; $\mathsf{E}\vdash\mathsf{MC}^2$ [41], which allows verification of CTMCs and DTMCs against CSL and PCTL specifications, respectively; and RAPTURE [26], which uses abstraction and refinement to perform model checking for a subset of PCTL over MDPs. Similarly to the tool PRISM, both ProbVerus and RAPTURE use MTBDDs. $\mathsf{E}\vdash\mathsf{MC}^2$, on the other hand, is an explicit implementation based on sparse matrices. The APNN-Toolbox [11], a Petri net based tool which supports analysis of CTMCs, has recently added support for CSL model checking. There are also a wide range of other CTMC-based tools available which, despite not offering probabilistic model checking explicitly, typically perform steady-state and transient analysis of CTMCs, the numerical computation for which is very similar. These include GreatSPN [19], MARCA [61], Möbius [25], the PEPA workbench [34], SMART [20], TIPP-tool [40] and UltraSAN [59].

Of the implementations described above, SMART and the APNN-Toolbox are of particular interest because, like PRISM, they incorporate sophisticated data structures designed to exploit large, structured models. Both tools include support for *Kronecker* representations, the basic idea of which is that the transition matrix of a CTMC is defined as a Kronecker (tensor) algebraic expression of smaller matrices, corresponding to sub-components of the model. Like MTBDD-based approaches, the goal is to derive a compact representation of large models by exploiting high-level structure and regularity. Methods have been developed which allow numerical solution of a CTMC to be performed directly on its Kronecker representation (see for example [58,16]). This process has close similarities to the techniques described in this paper in that a compact, symbolic representation of the CTMC is used alongside explicit (array-based) storage of the solution vector.

Furthermore, SMART uses a variety of BDD-based data structures. Most relevant to this paper are *matrix diagrams* [21,56], a data structure developed as an efficient implementation of the Kronecker techniques. There are several further similarities between this approach and ours. Firstly, it uses decision diagrams, i.e. reduced directed acyclic graphs, like BDDs and MTBDDs. In the case of matrix diagrams, the nodes of these graphs are the small matrices which make up the model's Kronecker representation. In fact, SMART uses a combination of matrix diagrams, to represent and manipulate CTMC transition rate matrices, and MDDs (multi-valued decision diagrams), to generate and store the CTMC's reachable state space. This is analogous with the way we use MTBDDs and BDDs, respectively. Furthermore, matrix diagrams and MDDs are labelled with offsets which are used to compute row and column offsets. These serve the same purpose as the values added to offset-labelled MTBDDs. Finally, the implementations of both MTBDD and matrix diagram based numerical computation require consideration of a number of factors common to the majority of BDD-based data structures. These include the encoding of the model's state space into variables, the ordering chosen for these variables, and the use of caches to minimise duplicated operations.

There are also important differences between the matrix diagram approach, used in SMART, and the offset-labelled MTBDD approach, described in this paper and implemented in PRISM. Below, we outline these differences and consider the effect that they have on the time

Model	N	States	Average time/iter (sec.)			Matrix storage (KB)		
			Hybrid	MDs	Sparse	Hybrid	MDs	Sparse
Kanban system (CTMC)	2	4,600	0.0004	0.0004	0.0002	39.5	1.2	348
	3	58,400	0.01	0.007	0.005	58.0	2.6	5,459
	4	454,475	0.08	0.07	0.05	115	4.9	48,414
	5	2,546,432	0.46	0.45	0.32	148	8.3	296,588
	6	11,261,376	2.28	2.00	-	185	12.9	-
Cyclic server polling system (CTMC)	10	15,360	0.002	0.002	0.001	22.2	1.4	1,110
	12	73,728	0.01	0.01	0.007	30.8	1.7	6,192
	14	344,064	0.04	0.08	0.04	40.9	2.0	32,928
	15	737,280	0.10	0.16	0.08	46.5	2.2	74,880
	16	1,572,864	0.24	0.35	0.19	52.3	2.3	168,960
	17	3,342,336	0.52	0.79	-	59.0	2.5	-
	18	7,077,888	1.16	1.66	-	65.5	2.6	-

Table 4. Comparison of offset-labelled MTBDDs, matrix diagrams and sparse matrices

and space efficiency of the two techniques. In this respect, we are chiefly interested in two main aspects of the implementation: the amount of memory taken up by the data structure and the speed with which information can be extracted from it for numerical solution.

The most obvious difference is that matrix diagrams are based on the Kronecker representation. This means that to extract a single matrix entry from a matrix diagram, it is necessary not only to trace a path through the data structure (like with offset-labelled MTBDDs), but also to perform a number of numerical operations (multiplications and possibly additions). In practice, when all matrix entries are extracted at once, a reduction in the number of operations performed is possible, in the same way that all matrix entries are extracted from an offset-labelled MTBDD via a single traversal of the data structure, as opposed to tracing a separate path for each matrix entry. The advantage of the Kronecker approach, however, is that it will often result in a more compact representation. In particular, in cases where the matrix contains many distinct values, MTBDDs will generally be more expensive by comparison, since this results in a larger number of terminal nodes, reducing the capacity for sharing between nodes, and hence increasing the size of the data structure.

Another important factor to consider is the encoding of row and column indices. For MTBDDs, the encoding is into Boolean variables; while for matrix diagrams, it is into integer-valued variables. This makes the implementation of MTBDDs simpler and reduces the storage required for each node. However, the number of levels in a matrix diagram (i.e. its height) will generally be smaller. Of course, this makes traversing the data structure during numerical solution faster. Integer-valued variables may also result in a more intuitive encoding of a model in terms of its high-level description.

All in all, a comparison of the relative merits of the two structures is fairly complex and it is likely that each approach will fare better in some circumstances. In Table 4, we present some preliminary, empirical results which compare the two. We performed computation of the steady-state probabilities for two CTMC case studies: the Kanban manufacturing system, as used earlier in the paper, and the cyclic server polling system of [44]. We used the JOR and Jacobi methods, respectively, and ran experiments on a 2GHz PC with 512KB of main memory. Equivalent computations were performed for offset-labelled MTBDDs ("Hybrid"), matrix diagrams ("MDs") and sparse matrices ("Sparse"). For matrix diagrams, we used SMART (version 1.1); for the other two approaches, we used PRISM (version 1.3.1). For all three implementations, the table gives the average time per iteration of numerical computation and the memory required to store the CTMC transition rate matrix.

The statistics in Table 4 back up our discussion of the various points above. Considering first memory usage, we see that matrix diagrams do indeed provide a more compact representation than offset-labelled MTBDDs on these examples. However, both are negligible compared to the storage requirements of a sparse matrix. In addition, as we will discuss shortly, they are insignificant with respect to the amount of memory needed to store the solution vector. In terms of speed for numerical solution, we observe that offset-labelled MTBDDs and matrix diagrams are essentially comparable, each outperforming the other on one of the examples. Both approaches are slightly slower than sparse, but are still competitive in this respect.

Another issue to bear in mind which is not included in the above discussion or experimental results is the choice of numerical solution method. For solution of linear equation systems, it is often preferable to use the Gauss Seidel method, rather than the Jacobi method, since it generally converges faster and can be implemented with a single solution vector. An efficient implementation of Gauss Seidel has been developed for matrix diagrams. A version for offset-labelled MTBDDs, while certainly possible in principle, has yet to be realised in practice. See [57] for an alternative, where offset-labelled MTBDDs are applied to a block-based variant of Gauss-

Seidel. Note also that, for several numerical solution problems, including transient analysis of CTMCs and PCTL model checking of MDPs, this issue does not arise.

We conclude our discussion of related work by observing that there is one vital issue which unites offset-labelled MTBDDs, matrix diagrams and other implementations of the Kronecker approach, namely the storage of solution vectors. While all these techniques typically achieve compact matrix representation, they also require explicit storage of at least one vector of size proportional to the number of states in the model being analysed. This constitutes the limiting factor in terms of the size of model which can be handled.

Removal of this bottleneck remains an important challenge for the development of these techniques. Attempts to harness any regularity in the solution vectors by storing them symbolically, either with MTBDDs or PDGs (Probabilistic Decision Graphs) [14,17], have generally failed to resolve the problem. An alternative direction which we are investigating [49,50] is the development of parallel or distributed techniques, where storage and workload is split across several computers or processors, and out-of-core techniques, where some data structures are stored on disk instead of in main memory.

8 Conclusion

In this paper, we have presented a novel approach to the symbolic implementation of probabilistic model checking for three types of models (DTMCs, MDPs and CTMCs) and two probabilistic logics (PCTL and CSL). In particular, we have focused on the problem of numerical computation. Our techniques overcome the limitations of existing MTBDD implementations by using explicit storage for solution vectors. This has increased the range of models to which MTBDD-based model checking can be applied and significantly improved the speed of the implementation. We also find that we can come close to, and in some cases beat, explicit sparse matrix based approaches in terms of solution speed. Furthermore, thanks to the compact model storage provided by MTBDDs, we can handle models an order of magnitude larger.

Our techniques have been implemented in the probabilistic model checker PRISM [51]. The tool, its source code and information for a range of case studies to which it has been applied can be found on the PRISM web site [1]. More detailed information about the techniques presented in this paper can be found in [57].

The development of PRISM is an ongoing project. Research directions to improve its implementation include, as described in the previous chapter, development of parallel, distributed or out-of-core implementations and investigation into more in-depth comparisons between our techniques and Kronecker-based implementations such as matrix diagrams. It would also be interesting to compare the effectiveness of our symbolic approach with that of alternative ideas for handling large probabilistic models, such as using disk-based storage [32,47] or "on-the-fly" techniques [33].

References

1. PRISM web site. www.cs.bham.ac.uk/~dxp/prism.
2. R. Alur and T. Henzinger. Reactive modules. In *Proc. 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 207–218. IEEE Computer Society Press, July 1996.
3. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, 1990.
4. 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.
5. 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.
6. C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim, 1998.
7. 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.
8. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In A. Emerson and A. Sistla, editors, *Proc. 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.
9. 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.
10. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
11. F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In R. Puigjaner, N. Savino, and B. Serra, editors, *Proc. Computer Performance Evaluation (TOOLS'98)*, volume 1469 of *LNCS*, pages 356–359. Springer, 1998.
12. 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.

13. B. Bollig and I. Wegner. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.
14. 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.
15. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
16. 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.
17. 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.
18. J. Burch, E. Clarke, K. McMillan, D. 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.
19. G. Chiola. GreatSPN 1.5 software architecture. *Computer Performance Evaluation*, pages 121–136, 1992.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. D. Daly, D. Deavours, J. Doyle, P. Webster, and W. Sanders. Möbius: An extensible tool for performance and dependability modeling. In B. Haverkort, H. Bohnenkamp, and C. Smith, editors, *Proc. 11th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'00)*, volume 1786 of *LNCS*, pages 332–336. Springer, 2000.
26. 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.
27. L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
28. L. de Alfaro. Temporal logics for the specification of performance and reliability. In R. Reischuk and M. Morvan, editors, *Proc. Symposium on Theoretical Aspects of Computer Science (STACS'97)*, volume 1200 of *LNCS*, pages 165–176. Springer, 1997.
29. L. de Alfaro. How to specify and verify the long-run average behavior of probabilistic systems. In *Proc. 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 454–465, 1998.
30. L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In J. Baeten and S. Mauw, editors, *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 66–81. Springer, 1999.
31. 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.
32. D. Deavours and W. Sanders. An efficient disk-based tool for solving very large Markov models. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th International Conference on Modelling Techniques and Tools (TOOLS'97)*, volume 1245 of *LNCS*, pages 58–71. Springer, 1997.
33. D. Deavours and W. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.
34. 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.
35. 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.
36. 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.
37. H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
38. 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.
39. T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.
40. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39(1-4):5–35, January 2000.
41. 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.
42. 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*, 56(1-2):23–67, 2003.
43. 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.
44. O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.
45. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.
46. 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.
47. W. Knottenbelt and P. Harrison. Distributed disk-based solution techniques for large Markov models. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 58–75. Prensas Universitarias de Zaragoza, 1999.
48. M. Kuntz and M. Siegle. Deriving symbolic representations from stochastic process algebras. In H. Hermanns and R. Segala, editors, *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, volume 2399 of *LNCS*, pages 188–206. Springer, 2002.
49. M. Kwiatkowska and R. Mehmood. Out-of-core solution of large linear systems of equations arising from stochastic modelling. In H. Hermanns and R. Segala, editors, *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, volume 2399 of *LNCS*, pages 135–151. Springer, 2002.
50. 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.
51. 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.
52. 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.
53. 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.
54. 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*, 14:295–318, 2003.
55. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
56. A. Miner. *Data Structures for the Analysis of Large Structured Markov Chains*. PhD thesis, Department of Computer Science, College of William & Mary, Virginia, 2000.
57. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
58. 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.
59. W. Sanders, W. Obal, M. Qureshi, and F. Widjanarko. The UltraSAN modeling environment. *Performance Evaluation*, 24(1):89–115, 1995.
60. F. Somenzi. CUDD: Colorado University decision diagram package. Public software, Colorado Univeristy, Boulder, http://vlsi.colorado.edu/~fabio/, 1997.
61. W. Stewart. MARCA: Markov chain analyser, a software package for Markov modelling. In *Proc. NSMC'91*, 1991.
62. 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.
63. 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.