Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach[⋆]

Marta Kwiatkowska, Gethin Norman, and David Parker

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

Abstract. In this paper we introduce PRISM, a probabilistic model checker, and describe the efficient symbolic techniques we have developed during its implementation. PRISM is a tool for analysing probabilistic systems. It supports three models: discrete-time Markov chains, continuous-time Markov chains and Markov decision processes. Analysis is performed through model checking 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. We present a novel hybrid technique which combines aspects of symbolic and explicit approaches to overcome these performance problems. For typical examples, we achieve orders of magnitude speed-up compared to MTBDDs and are able to almost match the speed of sparse matrices whilst maintaining considerable space savings.

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 probabilistic model of the system. Properties 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. Model checking algorithms

[⋆] Supported in part by EPSRC grant GR/M04617 and MathFIT studentship for David Parker.

J.-P. Katoen and P. Stevens (Eds.), 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), volume 2280 of LNCS, pages 52–66, 2002.

have been developed which then automatically verify whether the model satisfies these properties.

Motivated by the success of symbolic model checkers, such as SMV [28] which use BDDs (binary decision diagrams) [11], we have developed a symbolic *probabilistic* model checker. In the non-probabilistic setting, model checking involves analysing properties of state transition systems and the manipulation of sets of states. Both these entities can be represented naturally as BDDs, often very compactly [13]. In the probabilistic case, since probability transition matrices and probability vectors are required, BDDs alone are not sufficient, and hence we also use MTBDDs (multi-terminal binary decision diagrams) [17,3], a natural extension of BDDs for representing real-valued functions.

Symbolic probabilistic model checking has been considered by a number of people [5,21,4,26,19,23,7,25,27] and it has been shown that it is feasible to use MTBDDs to construct and compute the reachable state space of extremely large, structured, probabilistic models. In these cases, it is often also possible to verify *qualitative* properties, where model checking reduces to reachability-based analysis. For example, in [19], systems with over 10^{30} states have been verified.

Model checking *quantitative* properties, on the other hand, involves numerical computation. In some cases, such as in [27], 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, we have also implemented the equivalent numerical computation routines explicitly, using sparse matrices. In these cases, we find that sparse matrices are orders of magnitude faster. Here, we present a novel hybrid approach which uses extensions of the MTBDD data structure and borrows ideas from the sparse matrix 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, and in fact almost match the speed of sparse matrices, 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 this model checking. We then move on to discuss the implementation. Section 4 introduces the MTBDD data structure and explains how it is 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 and analyse the success of our technique. Section 7 concludes the paper.

2 Probabilistic Model Checking

In this section we briefly summarise the three probabilistic models and two temporal logics that PRISM supports. The simplest probabilistic model is the *discrete-time Markov chain* (DTMC), which specifies the probability $\mathbf{P}(s, s')$ of making a transition from state s to some target state s', where the probabilities of reaching the target states from a given state 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 non-deterministic behaviour. More formally, in any state there is a non-deterministic choice between a number of discrete probability distributions over states. Non-determinism enables the modelling of asynchronous parallel composition of probabilistic systems, and permits the under-specification of certain aspects of a system. A *continuous-time Markov chain* (CTMC), on the other hand, specifies the rates $\mathbf{R}(s, s')$ of making a transition from state s to s', with the interpretation that the probability of moving from s to s' within t time units (for positive real valued t) is $1 - e^{-\mathbf{R}(s,s')\cdot t}$.

Probabilistic specification formalisms include PCTL [20,10,8], a probabilistic extension of the temporal logic CTL applicable in the context of MDPs and DTMCs, and the logic CSL [7], a specification language for CTMCs based on CTL and PCTL.

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 randomized leader election protocol of Itai and Rodeh [24] 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 fair scheduling, a leader is eventually elected with probability 1".
– $\mathcal{P}_{\leq 0.5}[\Diamond^{\leq k}\, \mathit{leader}]$ - "under any fair scheduling, the probability of electing a leader within k discrete time steps is at most 0.5".

The specification language 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 chance that the queue is not full is at least 0.98".

Model checking algorithms for PCTL have been introduced in [20,10] and extended in [8,4] to include fairness. An algorithm for CSL was first proposed in [7] and has since been improved in [6,25]. The model checking algorithms for both logics reduce to a combination of reachability-based computation (manipulation of sets of states) and numerical computation. The former corresponds to finding all those states that satisfy the formula under study with probability exactly 0 or 1. The latter corresponds to calculating the probabilities for the remaining states. For DTMCs, this entails solution of a linear equation system, for

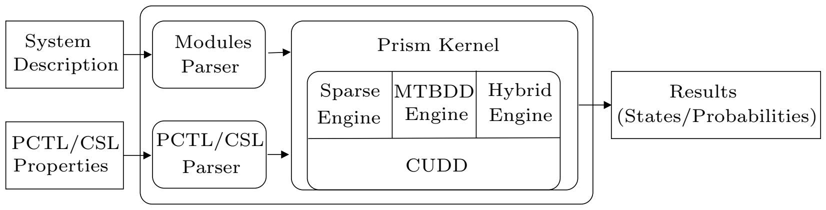


Fig. 1. PRISM System Architecture

MDPs, solving a linear optimisation problem, and for CTMCs, either solution of a linear equation system or a standard technique known as uniformisation. These numerical problems are typically too large for the application of direct methods, and instead iterative techniques which approximate the solution up to some specified accuracy are used.

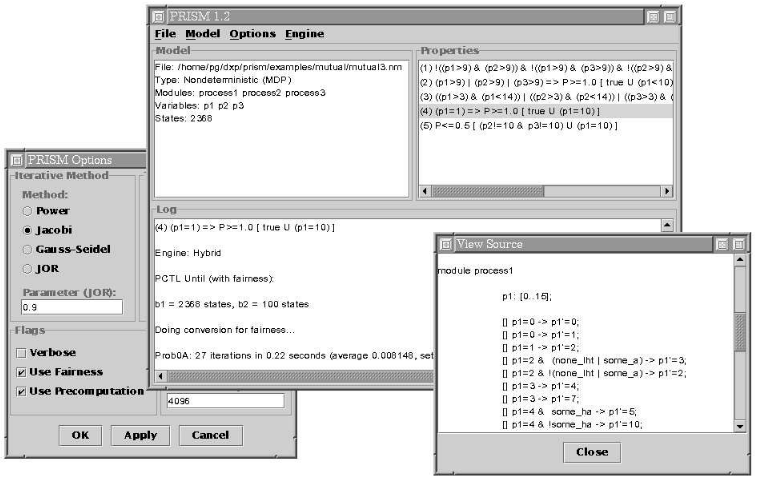
3 The Tool

PRISM is a tool developed at the University of Birmingham which supports the model checking described in the previous section. The tool takes as input a description of a system written in PRISM's system description language, a probabilistic variant of Reactive Modules [1]. It first constructs the model from this description and computes the set of reachable 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 system satisfy each specification. All reachability-based computation is performed using BDDs. For numerical analysis, however, there is a choice of three engines: one using pure MTBDDs, one based on conventional sparse matrices, and a third using the hybrid approach we present in this paper. Figure 1 shows the structure of the tool and Figure 2 shows a screen-shot of the graphical user interface.

PRISM is written in a combination of Java and C++ and uses CUDD [32], 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 engines and libraries are written in C++. The tool and its source code, along with further information about the system description language and case studies, is available from the PRISM web page [31].

4 An MTBDD Implementation

We now describe the implementation of the tool. The fundamental data structures in PRISM are MTBDDs and BDDs. MTBDDs can be used to represent all three of the supported models: DTMCs, MDPs and CTMCs. Furthermore, all algorithms for the construction and analysis of these models can be implemented



```
File: /home/pg/dxp/prism/examples/mutual/mutual3.nm
Type: Nondeterministic (MDP)
Modules: process1 process2 process3
Variables: p1 p2 p3
States: 2368
```

```
(1) !((p1>9) & (p2>9)) & !((p1>9) & (p3>9)) & !((p2>9) &
(2) (p1>9) | (p2>9) | (p3>9) => P>=1.0 [ true U (p1<10)
(3) ((p1>3) & (p1<14)) | ((p2>3) & (p2<14)) | ((p3>3) & (
(4) (p1=1) => P>=1.0 [ true U (p1=10) ]
(5) P<=0.5 [ (p2!=10 & p3!=10) U (p1=10) ]
```

```
(4) (p1=1) => P>=1.0 [ true U (p1=10) ]

Engine: Hybrid

PCTL Until (with fairness):

b1 = 2368 states, b2 = 100 states

Doing conversion for fairness...

Prob0A: 27 iterations in 0.22 seconds (average 0.008148, set
```

Power
[✓ Jacobi]
Gauss-Seidel
JOR

Parameter (JOR): 0.9

Verbose
[✓ Use Fairness]
[✓ Use Precomputation]

4096

```
module process1

            p1: [0..15];

            [] p1=0 -> p1'=0;
            [] p1=0 -> p1'=1;
            [] p1=1 -> p1'=2;
            [] p1=2 &  (none_lht | some_a) -> p1'=3;
            [] p1=2 & !(none_lht | some_a) -> p1'=2;
            [] p1=3 -> p1'=4;
            [] p1=3 -> p1'=7;
            [] p1=4 &  some_ha -> p1'=5;
            [] p1=4 & !some_ha -> p1'=10;
```

Fig. 2. The PRISM Graphical User Interface

using these data structures. In this section, we summarise how this is done and discuss its performance.

Introduction to MTBDDs

Let $x_1 < \cdots < x_n$ be a set of distinct, totally ordered, Boolean variables. An MTBDD M over $(x_1, \ldots, x_n)$ is a rooted, directed acyclic graph with vertex set $V = V_n \cup V_t$ partitioned into *non-terminal* and *terminal* vertices. A non-terminal vertex $v \in V_n$ is labelled by a variable $var(v) \in \{x_1, \ldots, x_n\}$ and has two children $then(v), else(v) \in V$. A terminal vertex $v \in V_t$ is labelled by a real number $val(v)$.

We impose the Boolean variable ordering $<$ onto the graph by requiring that a child w of a non-terminal vertex v is either terminal or is non-terminal and satisfies $var(v) < var(w)$. The MTBDD represents a function $f_{\mathsf{M}}(x_1, \ldots, x_n) : \mathbb{B}^n \rightarrow \mathbb{R}$. The value of $f_{\mathsf{M}}(x_1, \ldots, x_n)$ is determined by traversing M from the root vertex, following the edge from vertex v to $then(v)$ or $else(v)$ if $var(v)$ is 1 or 0 respectively. Note that a BDD is merely an MTBDD with the restriction that the labels on terminal vertices can only be 1 or 0.

MTBDDs are efficient because they are stored in reduced form. If vertices v and w are identical (i.e. $var(v) = var(w)$, $then(v) = then(w)$ and $else(v) = else(w)$), then only one is stored. Furthermore, if a vertex v satisifes $then(v) = else(v)$, it is removed and any incoming edges are redirected to its unique child.

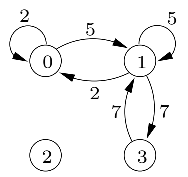
One of the most important factors about MTBDDs from a practical point of view is that their size (number of vertices) is heavily dependent on the ordering of the Boolean variables. Although, in the worst case, the size of an MTBDD representation is exponential and the problem of deriving the optimal ordering for a given MTBDD is an NP-complete problem, by applying heuristics to minimise graph size, MTBDDs can provide extremely compact storage for real-valued functions.

MTBDD Represention of Probabilistic Models

From their inception in [17,3], MTBDDs have been used to encode real-valued vectors and matrices. An MTBDD M over variables $(x_1, \ldots, x_n)$ represents a function $f_{\mathsf{M}} : \mathbb{B}^n \to \mathbb{R}$. 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 M 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. The case for MDPs is more complex since we need to encode the non-deterministic choices. If the maximum number of non-deterministic choices in any state is bounded by 2^k for some integer k, we can view the MDP as a function from $\{1, \ldots, 2^k\} \times \{1, \ldots, 2^n\} \times \{1, \ldots, 2^n\}$ to $\mathbb{R}$. By adding k extra Boolean variables to encode this third index, we can represent the MDP as an MTBDD.

Figure 3 shows an example of a CTMC and its rate matrix. The CTMC includes one state which is unreachable. This is explained in Section 5. Figure 4 gives the MTBDD which represents this CTMC and a table explaining its construction. For clarity, in our notation for MTBDDs, we omit edges which lead directly to the zero terminal vertex.



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

Fig. 3. A CTMC and its rate matrix

Transition	x_1	x_2	y_1	y_2	x_1	y_1	x_2	y_2	$\mathbf{R}$
$0 \xrightarrow{2} 0$	0	0	0	0	0	0	0	0	2
$0 \xrightarrow{5} 1$	0	0	0	1	0	0	0	1	5
$1 \xrightarrow{2} 0$	0	1	0	0	0	0	1	0	2
$1 \xrightarrow{5} 1$	0	1	0	1	0	0	1	1	5
$1 \xrightarrow{7} 3$	0	1	1	1	0	1	1	1	7
$3 \xrightarrow{7} 1$	1	1	0	1	1	0	1	1	7

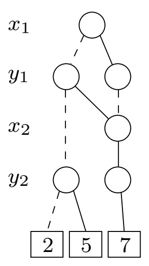


Fig. 4. An MTBDD representing the CTMC in Figure 3

Observe that, in Figure 4, row and column variables are ordered alternately. This is one of the most common variable ordering heuristics to minimise MTBDD size. To achieve compact MTBDD representations of probabilistic systems, however, we must also consider the actual encoding of row and column indices to Boolean variables. A well known rule of thumb is to try and preserve structure in the entity being represented [23]. In practice, this can be accomplished by performing a direct translation from a high-level description of the model (in our case the PRISM system description language) to MTBDDs. We presented such a scheme in [19] which is not only fast but can lead to a very compact encoding of probabilistic systems. The resulting variable ordering encodes unreachable states, as well as reachable states, and hence reachability analysis (via a simple BDD fixpoint calculation) must be performed to identify them.

Our experimental data is presented in Section 6. For reasons of space we only include statistics for two typical examples: firstly, an MDP model of the coin protocol from Aspnes and Herlihy's randomized consensus algorithm [2], parameterised by N (the number of processes) and an additional parameter K fixed at 4; secondly, a CTMC model of a Kanban manufacturing system [16] parameterised by N (the number of pallets in the system). Figure 8 gives the memory requirements for storing these models. Compare the 'MTBDD' and 'Sparse' columns: significant savings in memory can be achieved using the symbolic scheme described above over an explicit storage method. For other examples which demonstrate this result, see the PRISM web page [31].

Probabilistic Model Checking with MTBDDs

We have implemented the entire model checking procedure for PCTL and CSL in MTBDDs and BDDs. As we saw in Section 2, essentially this reduces to a combination of reachability-based computation and numerical calculation. The former can be performed with BDDs and forms the basis of non-probabilistic symbolic model checking which has been proven to be very successful [13,28]. The latter involves iterative numerical methods, based on matrix-vector multiplication, an operation for which efficient MTBDD algorithms have been introduced [3,17]. In fact, alternative, direct, methods such as Gaussian elimination and Simplex could be applied to some of these problems, but have been shown to be unsuitable for an MTBDD implementation [3,26]. 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 this implementation in MTBDDs can be summarised as follows. There is a clear distinction between the two different aspects of model checking. Reachability-based computation, which is sufficient for model checking qualitative properties, can be implemented efficiently with BDDs, as is shown in [19,27]. Numerical computation, which is required for checking of quantitative properties, is more unpredictable. This is the problem we focus on here.

We have found a number of case studies for which MTBBDs outperform explicit techniques on numerical computation. One such example is the coin protocol, introduced previously. We include results for this model in the top half of Figure 9. Compare the columns for 'MTBDD' and 'Sparse': it would be impossible to even store the sparse matrix for the larger examples, assuming a reasonable amount of memory. We have found this pattern to hold for several of the other MDP case studies we have considered. Other examples which illustrate this can be found on the PRISM web site [31], and include a Byzantine agreement protocol and the IEEE 1394 FireWire root contention protocol.

For a second class of models, namely CTMCs, we find that the symbolic implementation of numerical iterative methods is far from efficient. Despite a compact MTBDD representation of the model, the process is generally very slow or infeasible. This inefficiency is caused by the MTBDD representation of the iteration vectors becoming too large. For vectors to be represented compactly by MTBDDs the main requirement is a limited number of distinct elements. However, in general, when performing numerical analysis, the iteration vector quickly acquires almost as many distinct values as there are states in the system under study. Figure 9 shows the contrast in performance of MTBDDs between the Kanban CTMC and the coin protocol MDP. The sparse matrix based implementation is much faster.

5 A Hybrid Approach

We now present a method to overcome the inefficiencies with MTBDDs outlined in the previous section. Recall that sparse matrix techniques can yield extremely fast numerical computation. Since the iteration vector is stored in a full array, it remains a constant size. A single matrix-vector multiplication is carried out by traversing the sparse matrix and extracting all the non-zero entries, each of which is needed exactly once to compute the new iteration vector. Unfortunately, since the probabilistic model is also stored explicitly, application to large examples is often limited by memory constraints.

The approach taken here is to a use a hybrid of the two techniques: MTBDDs and sparse matrices. We store the transition matrix in an MTBDD-like data structure but use a full array for the iteration vector. We can then perform matrix-vector multiplication, and hence iterative methods, using these two data structures. The key difference in this hybrid approach is that we need to extract the non-zero matrix entries from an MTBDD rather than a sparse matrix. For clarity, this presentation focuses on the case of CTMCs, where we solve a linear equation system by iterative methods to compute the steady-state probabilities. We have also applied these techniques to DTMC and MDP models.

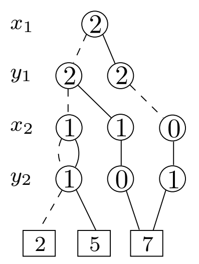
If we restrict ourselves to certain iterative methods, namely Power, Jacobi and JOR, then the matrix entries can be extracted in any order to perform an iteration. This means that we can can proceed via a recursive traversal of the MTBDD: it does not that matter that this will enumerate the entries in an essentially random order, rather than row-by-row (or column-by-column) as with a sparse matrix.

Since the matrix indices are encoded with the standard binary representation of integers, it is trivial to keep track of the row and column index during traversal by noting whether a *then* or *else* edge is taken at each point and summing the appropriate powers of 2. Unfortunately, there are a number of problems with this naive approach. To resolve these, we make a number of modifications to the MTBDD data structure.

Modifying the MTBDD Data Structure

First, recall from Section 4, that to get an efficient MTBDD representation of our transition matrix, it must contain unreachable states. 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 augment the MTBDD with vertex labels: integer offsets which can be used to compute the actual indices of the matrix elements (in terms of reachable states only) during our recursive traversal. Figure 5 illustrates this idea on the example from Section 4, which included an unreachable state. On the left is the modified MTBDD representing the transition matrix **R** of the CTMC. The table on the right explains how the traversal process works. Each row corresponds to a single matrix entry (transition). The first five columns describe the path taken through the MTBDD. The next four columns give the vertex offsets along this path. The last column gives the resulting matrix entry. In Figure 6, we give the actual traversal algorithm. This would be called as follows: TraverseRec(*root*, 0, 0), where *root* is the top-level vertex in the MTBDD.



Path					Offsets				Transition
x_1	y_1	x_2	y_2	**R**	x_1	y_1	x_2	y_2	
0	0	0	0	2	-	-	-	-	$0 \xrightarrow{2} 0$
0	0	0	1	5	-	-	-	1	$0 \xrightarrow{5} 1$
0	0	1	0	2	-	-	1	-	$1 \xrightarrow{2} 0$
0	0	1	1	5	-	-	1	1	$1 \xrightarrow{5} 1$
0	1	1	1	7	-	2	1	0	$1 \xrightarrow{7} 2$
1	0	1	1	7	2	-	0	1	$2 \xrightarrow{7} 1$

Fig. 5. The modified MTBDD representing the CTMC in Figure 5

The key idea is that indices are computed by summing offsets. A vertex's offset is only added when leaving the *then* edge of that vertex. Note that row and column indices are computed independently, rows from offsets on x_i vertices, and columns from offsets on y_i vertices. As an example, consider the last line of the table. We take the path $1, 0, 1, 1$ through the MTBDD which leads to the 7 terminal vertex. We only use the offsets at levels x_1, x_2 and y_2 where we exited via *then* edges. The row index is $2 + 0 = 2$, the column index is 1 and we obtain

```
TraverseRec($v$, $row$, $col$)
    if ($v$ is a non-zero terminal vertex) then
        found matrix element $(row, col) = val(v)$
    elseif ($v$ is a row vertex) then
        TraverseRec($else(v)$, $row$, $col$)
        TraverseRec($then(v)$, $row + offset(v)$, $col$)
    elseif ($v$ is a row vertex) then
        TraverseRec($else(v)$, $row$, $col$)
        TraverseRec($then(v)$, $row$, $col + offset(v)$)
    endif
end
```

Fig. 6. Hybrid Traversal Algorithm

the matrix entry $(2,1) = 7$. Note that references to state 3 in Figure 4 have changed to state 2 in Figure 5, since we only have three *reachable* states.

There are two further points to consider about the conversion of the MTBDD (Figure 4) to its new form (Figure 5). First, note that some vertices in an MTBDD can be reached along several different paths. These shared vertices correspond to repeated sub-matrices in the overall matrix. Consider the matrix in Figure 3 and its MTBDD representation in Figure 4. 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 vertex 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 potential clash as to which offset should label the vertex.

We resolve this by adding extra copies of the vertex where necessary, labelled with different offsets. Note the additional two vertices on the right hand side in Figure 5. Effectively, we have modified the condition under which two MTBDD vertices are merged, requiring not only that are they are on the same level and have identical children, but also that they have the same offset label. It should be noted here that we transform the MTBBD once, use it for as many iterations are required, and then discard it. Hence, we only need to be able to traverse the data structure, not manipulate it in any way.

The second point to make about the conversion involves skipped levels. In an MTBDD, if a vertex has identical children, it is omitted to save space. This causes potential problems, because we must be careful to detect this during traversal. In fact, the solution we adopt is to perform this check only once, during the initial conversion, and replace skips with extra vertices. This allows us to ignore the issue entirely during traversal and makes the process faster. There is an example of this in Figure 5 – note the extra x_2 vertex on the left hand side. The exception to this rule is that we do allow edges to skip from any vertex directly to the zero terminal vertex, since we are only interested in the non-zero entries.

Optimising the Approach

We can optimise our method considerably via a form of caching. MTBDDs exploit structure in the model being analysed giving a significant space saving. This is achieved by identical vertices (representing identical sub-matrices) being merged and stored only once. During traversal, however, each of these shared vertices 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 computed every time. By storing and reusing the results of this computation, 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 vertices. We select some subset of the vertices, build explicit (sparse matrix) representations of their associated sub-matrices and attach them to the MTBDD. 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 vertices (and which ones) we attach matrices to. From our experiences, a good policy is to replace all the vertices in one (fairly low) level of the MTBDD. In Figure 7, we demonstrate this technique on the running example, replacing all vertices on the x_2 level with the matrices they represent.

In practice, we find that caching can improve traversal speed by an order of magnitude. In the next section, we give experimental results from our implementation which includes all the techniques described here.

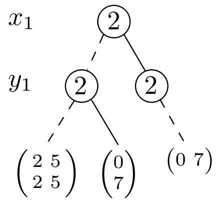

x_1
y_1
2
2
2
$\begin{pmatrix} 2 & 5 \\ 2 & 5 \end{pmatrix}$
$\begin{pmatrix} 0 \\ 7 \end{pmatrix}$
$\begin{pmatrix} 0 & 7 \end{pmatrix}$

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

6 Results

In this section, we present our experimental results, obtained from the PRISM tool. We compare the performance of the three implementations discussed in this paper: pure MTBDDs, sparse matrices, and our hybrid approach, focusing on the problem of iterative numerical computation.

In Figure 8, we give storage requirements for the coin protocol and Kanban models introduced earlier. We compare the size of the MTBDD, the sparse matrix and the modified MTBDD used in the hybrid approach, with and without optimisation. In Section 4, we observed the significant advantage of MTBDDs over sparse matrices. Note that, even when storing offset information, extra vertices and explicit sub-matrices, the hybrid approach remains memory efficient.

Model	N	States	Memory (KB)			
			MTBDD	Sparse	Hybrid	Hybrid Opt.
Coin protocol	2	528	6.4	16.5	10.4	12.3
	4	43,136	28.5	2,265	56.7	69.8
	6	2,376,448	61.2	173,424	93.3	314
	8	114,757,632	109	10,673,340	171	1,669
	10	5,179,854,848	170	584,181,500	275	3,600
Kanban system	3	58,400	48.3	5,459	86.0	99.9
	4	454,475	95.7	48,414	171	231
	5	2,546,432	123	296,588	219	337
	6	11,261,376	154	1,399,955	272	486
	7	41,644,800	186	5,441,445	327	685

Fig. 8. Storage requirements for the coin protocol and Kanban examples

Model	N	States	Iter.s	Time per iteration (sec.)			
				MTBDD	Sparse	Hybrid	Hybrid Opt.
Coin protocol	2	528	1,740	0.008	0.0002	0.001	0.0006
	4	43,136	6,133	0.173	0.034	0.07	0.039
	6	2,376,448	12,679	1.01	1.741	5.58	3.02
	8	114,757,632	21,110	3.17	-	-	-
	10	5,179,854,848	31,255	8.38	-	-	-
Kanban system	3	58,400	300	41.7	0.044	0.451	0.052
	4	454,475	466	-	0.436	6.09	0.502
	5	2,546,432	663	-	2.76	33.4	3.150
	6	11,261,376	891	-	-	146	14.76
	7	41,644,800	1,148	-	-	558	58.87

Fig. 9. Model checking times for the coin protocol and Kanban examples

Furthermore, the time for adding this information to the MTBDD was in all cases negligle compared to that for model checking.

In Figure 9, we present model checking times for the same two case studies. For the coin protocol, we verify a quantitative PCTL property which requires solution of a linear optimisation problem. For the Kanban system, we model check a quantitative CSL property which requires computation of the steady-state probabilities via the solution of a linear equation system. We use the JOR iterative method. All experiments were run on a 440 MHz Sun Ultra 10 workstation with 1 GB memory. The iterative methods were terminated when the relative error between subsequent iteration vectors was less than 10^{-6}.

As we remarked in Section 4, the coin protocol model, and many of our other MDP models, are efficient for MTBDDs. The problem we try to address with our hybrid approach is typified by the Kanban example, where MTBDDs alone are inefficient. By using the techniques presented in this paper, we were able to consider larger models than with sparse matrices. Furthermore, using the optimised version, we can almost match the speed of sparse matrices. Other CTMC case studies we have considered, such as queueing networks and workstation clusters, confirm these results. Details can be found on the PRISM web page [31].

Related Work: We are aware of three other probabilistic model checking tools. ProbVerus [21] is an MTBDD-based model checker which only supports DTMCs and a subset of PCTL. The tool $\mathsf{E}\vdash\mathsf{MC}^2$ [22] supports model checking of CTMCs against CSL specifications using sparse matrices. The tool described in [18] uses abstraction and refinement to perform model checking for a subset of PCTL over MDPs. There are a number of sparse-matrix based DTMC and CTMC tools, such as MARCA [33], which do not allow logic specifications but support steady-state and transient analysis.

An area of research which has close links with our work is the Kronecker approach [30], a technique for the analysis of very large, structured CTMCs and DTMCs. The basic idea is that the matrix of the full system is defined as a Kronecker algebraic expression of smaller matrices, which correspond to subcomponents of the overall system. It is only necessary to store these small matrices and the structure of the Kronecker expression. Iterative solution methods can be applied to the matrix while in this form. As with our approach, storage requirements for the matrix are relatively small, but ingenious techniques must be developed to minimise the time overhead required for numerical solution. Tools which support Kronecker based methods include APNN [9] and SMART [14].

In particular, SMART incorporates matrix diagrams [15], a data structure developed as an efficient implementation of the Kronecker techniques. The matrix diagram approach has much in common with the hybrid method we present in this paper. In particular, both methods use a decision-diagram like data structure for storing matrices and full array to store vectors. The key difference is that matrix diagrams are tied to the Kronecker representation and as such require more work to extract the transition matrix entries. In addition to traversing the data structure, as we do, computation of matrix elements requires multiplication of entries from the smaller matrices.

Another important difference is that Kronecker and matrix diagram approaches permit the use of more efficient iterative methods, such as Gauss-Seidel. Our approach does not presently support these. Hence, although we have less work to do per iteration, we may require more iterations using our methods. In addition, Gauss-Seidel can be implemented with a single iteration vector, whereas methods such as Jacobi and JOR require two.

One issue that unites the Kronecker approach, matrix diagrams and our method is that their limiting factor is the space required to store the iteration vector. However compact the matrix representation is, memory proportional to the number of states is required for numerical solution. Buchholz and Kemper consider an interesting technique in [12] using PDGs (Probabilistic Decision Graphs). This attempts to store the iteration vector in a structured way, as is done with the matrix. More investigation is required to discover the potential of this approach.

7 Conclusion

We have introduced PRISM, a tool to build and analyse probabilistic systems which supports three types of models (DTMCs, MDPs and CTMCs) and two probabilistic logics (PCTL and CSL). As well as MTBDD and sparse matrix based model checking engines, PRISM includes a novel, hybrid engine which combines symbolic and explicit approaches. We have shown that very large probabilistic systems can be constructed and analysed using MTBDDs, but that, often, numerical computation is very inefficient. Our hybrid approach addresses these performance problems, allowing verification, at an acceptable speed, of much larger systems than would be feasible using sparse matrices. Further details of this will be available in [29].

One problem with our current techniques is that they presently only support the Power, Jacobi and JOR iterative methods. We plan to extend the work to allow more rapidly converging alternatives such as Gauss-Seidel or Krylov methods to be used.

The development of PRISM is an ongoing activity. In the near future we intend to consider extensions of PCTL for expressing expected time and long run average properties and of CSL to include rewards, expand the PRISM input language to allow process algebra terms, and develop model checking engines for PRISM which work in a parallel or distributed setting.

References

1. R. Alur and T. Henzinger. Reactive modules. In *Proc. LICS'96*, 1996.
2. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1), 1990.
3. I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E.Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. ICCAD'93*, 1993.
4. C. Baier. On algorithmic verification methods for probabilistic systems. Habilitation thesis, Universität Mannheim, 1998.
5. C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *Proc. ICALP'97*, 1997.
6. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Proc. CAV'00*, 2000.
7. C. Baier, J.-P. Katoen, and H. Hermanns. Approximative symbolic model checking of continuous-time Markov chains. In *Proc. CONCUR'99*, 1999.
8. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3), 1998.
9. F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In *Computer Performance Evaluation (Tools)*, 1998.
10. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. FST & TCS*, 1995.
11. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
12. P. Buchholz and P. Kemper. Compact representations of probability distributions in the analysis of superposed GSPNs. In *Proc. PNPM'01*, 2001.

13. 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. LICS'90*, 1990.
14. G. Ciardo and A. Miner. SMART: Simulation and Markovian analyser for reliability and timing. In *Tool Descriptions from PNPM'97*, 1997.
15. G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. PNPM'99*, 1999.
16. 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.
17. E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. IWLS'93*, 1993.
18. P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Proc. PAPM/PROBMIV'01*, 2001.
19. 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 *Proc. TACAS'00*, 2000.
20. H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6, 1994.
21. V. Hartonas-Garmhausen, S. Campos, and E. Clarke. ProbVerus: Probabilistic symbolic model checking. In *Proc. ARTS'99*, 1999.
22. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A Markov chain model checker. In *Proc. TACAS'00*, 2000.
23. H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Proc. NSMC'99*, 1999.
24. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.
25. J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *Proc. PAPM-PROBMIV'01*, 2001.
26. 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.
27. M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *Proc. CAV'01*, 2001.
28. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
29. D. Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, University of Birmingham, 2002. To appear.
30. 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*, 1985.
31. PRISM web page. http://www.cs.bham.ac.uk/~dxp/prism/.
32. F. Somenzi. CUDD: Colorado University decision diagram package. Public software, Colorado Univeristy, Boulder, 1997.
33. W. Stewart. MARCA: Markov chain analyser, a software package for Markov modelling. In *Proc. NSMC'91*, 1991.