Symbolic Representations and Analysis of Large Probabilistic Systems

Andrew Miner[1]* and David Parker[2]**

[1] Dept. of Computer Science, Iowa State University, Ames, Iowa, 50011
[2] School of Computer Science, University of Birmingham, UK

Abstract. This paper describes symbolic techniques for the construction, representation and analysis of large, probabilistic systems. Symbolic approaches derive their efficiency by exploiting high-level structure and regularity in the models to which they are applied, increasing the size of the state spaces which can be tackled. In general, this is done by using data structures which provide compact storage but which are still efficient to manipulate, usually based on binary decision diagrams (BDDs) or their extensions. In this paper we focus on BDDs, multi-valued decision diagrams (MDDs), multi-terminal binary decision diagrams (MTBDDs) and matrix diagrams.

1 Introduction

This paper provides an overview of *symbolic* approaches to the validation of stochastic systems. We focus on those techniques which are based on the construction and analysis of finite-state probabilistic models. These models comprise a set of states, corresponding to the possible configurations of the system being considered, and a set of transitions which can occur between these states, labeled in some way to indicate the likelihood that they will occur. In particular, this class of models includes discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs).

Analysis of these models typically centers around either computation of transient or steady-state probabilities, which describe the system at a particular time instant or in the long-run, respectively, or computation of the probability that the system will reach a particular state or class of states. These are the key constituents of several approaches to analyzing these models, including both traditional performance or dependability evaluation and more recent, model checking based approaches using temporal logics such as PCTL or CSL.

As is well known, one of the chief practical problems plaguing the implementations of such techniques is the tendency of models to become unmanageably large, particularly when they comprise several parallel components, operating concurrently. This phenomenon is often referred to as 'the state space explosion problem', 'largeness' or 'the curse of dimensionality'. A great deal of work

* Supported in part by fellowships from the NASA Graduate Student Researchers Program (NGT-1-52195) and the Virginia Space Grant Consortium.
** Supported in part by QinetiQ and EPSRC grant GR/S11107.

has been put into developing space and time efficient techniques for the storage and analysis of probabilistic models. Many of the recent approaches that have achieved notable success are *symbolic* techniques, by which we mean those using data structures based on binary decision diagrams (BDDs). Also known as *implicit* or *structured* methods, these approaches focus on generating compact model representations by exploiting structure and regularity, usually derived from the high-level description of the system. This is possible in practice because systems are typically modeled using structured, high-level specification formalisms such as Petri nets and process algebras. In contrast, *explicit* or *enumerative* techniques are those where the entire model is stored and manipulated explicitly. In the context of probabilistic models, sparse matrices are the most obvious and popular explicit storage method.

The tasks required to perform analysis and verification of probabilistic models can be broken down into a number of areas, and we cover each one separately in this paper. In Section 2, we consider the storage of sets of states, such as the reachable state space of a model. In Section 3, we look at the storage of the probabilistic model itself, usually represented by a real-valued transition matrix. We also discuss the generation of each of these two entities. In Section 4, we describe how the processes of analysis, which usually reduce to numerical computation, can be performed in this context. Finally, in Section 5, we discuss the relative strengths and weaknesses of the various symbolic approaches, and suggest some areas for future work.

2 Representing Sets of States

We begin by focusing on the problem of representing a set of states. The most obvious task here is to represent the entire set of states which the system can possibly be in. Usually, some proportion of the potential configurations of the system are impossible. These are removed by computing the set of *reachable* states, from some initial state(s), and removing all others. Note that in some verification technologies, non-probabilistic variants in particular, this *reachability* computation may actually constitute the verification itself. Here, though, it is seen as part of the initial construction of the model since probabilistic analysis cannot be carried out until afterwards.

In addition to storing the set of reachable states of the model, it may often be necessary to represent a particular class of states, e.g. those which satisfy a given specification. For model checking, in particular, this is a fundamental part of any algorithm. Clearly, in both cases, there is a need to store sets of states compactly and in such a way that they can be manipulated efficiently. In this section, we consider two symbolic data structures for this purpose: *binary decision diagrams* (BDDs) and *multi-valued decision diagrams* (MDDs).

2.1 Binary Decision Diagrams

Binary decision diagrams (BDDs) are rooted, directed, acyclic graphs. They were originally proposed by Lee [55] and Akers [3], but were later popularized by Bryant [13], who refined the data structure and presented a number of algorithms for their efficient manipulation. A BDD is associated with a finite set of Boolean variables and represents a Boolean function over these variables. We denote the function represented by the BDD B over K variables $x_K, \ldots, x_1$ as $f_{\mathsf{B}} : \mathbb{B}^K \rightarrow \mathbb{B}$.

The vertices of a BDD are usually referred to as *nodes*. A node m is either *non-terminal*, in which case it is labeled with a Boolean variable $var(\mathsf{m}) \in \{x_K, \ldots, x_1\}$, or *terminal*, in which case it is labeled with either 0 or 1. Each non-terminal node m has exactly two children, $then(\mathsf{m})$ and $else(\mathsf{m})$. A terminal node has no children. The value of the Boolean function f_{B}, represented by BDD B, for a given valuation of its Boolean variables can be determined by tracing a path from its root node to one of the two terminal nodes. At each node m, the choice between $then(\mathsf{m})$ and $else(\mathsf{m})$ is determined by the value of $var(\mathsf{m})$: if $var(\mathsf{m}) = 1$, $then(\mathsf{m})$ is taken, if $var(\mathsf{m}) = 0$, $else(\mathsf{m})$ is taken. Every BDD node m corresponds to some Boolean function f_{m}. The terminal nodes correspond to the trivial functions $f_{\mathsf{0}} = 0$, $f_{\mathsf{1}} = 1$.

For a function f, variable x_k, and Boolean value b, the *cofactor* $f|_{x_k=b}$ is found by substituting the value b for variable x_k:

$$f|_{x_k=b} = f(x_K, \ldots, x_{k+1}, b, x_{k-1}, \ldots, x_1).$$

An important property of BDDs is that the children of a non-terminal node m correspond to cofactors of function f_{m}. That is, for every non-terminal node m, $f_{then(\mathsf{m})} = f_{\mathsf{m}}|_{var(\mathsf{m})=1}$, and $f_{else(\mathsf{m})} = f_{\mathsf{m}}|_{var(\mathsf{m})=0}$. We will also refer to the cofactor of a BDD node m, with the understanding that we mean the BDD node representing the cofactor of the function represented by node m.

A BDD is said to be *ordered* if there is a total ordering of the variables such that every path through the BDD visits nodes according to the ordering. In an ordered BDD (OBDD), each child m' of a non-terminal node m must therefore either be terminal, or non-terminal with $var(\mathsf{m}) > var(\mathsf{m}')$.

A *reduced* OBDD (ROBDD) is one which contains no *duplicate* nodes, i.e. non-terminal nodes labeled with the same variable and with identical children, or terminal nodes labeled with the same value; and which contains no *redundant* nodes, i.e., non-terminal nodes having two identical children. Since the function represented by a redundant node m does not depend on the value of variable $var(\mathsf{m})$, redundant nodes are sometimes referred to as *don't care* nodes.

Any OBDD can be reduced to an ROBDD by repeatedly eliminating, in a bottom-up fashion, any instances of duplicate and redundant nodes. If two nodes are duplicates, one of them is removed and all of its incoming pointers are redirected to its duplicate. If a node is redundant, it is removed and all incoming pointers are redirected to its unique child.

A common variant of the above reduction scheme is to allow redundant nodes but not duplicate nodes. In particular, an OBDD is said to be *quasi-reduced* if it contains no duplicate nodes and if all paths from the root node (which must have label x_K) to a terminal node visit exactly one node for each variable. Note that a quasi-reduced OBDD (QROBDD) can be obtained from any ROBDD by inserting redundant nodes as necessary.

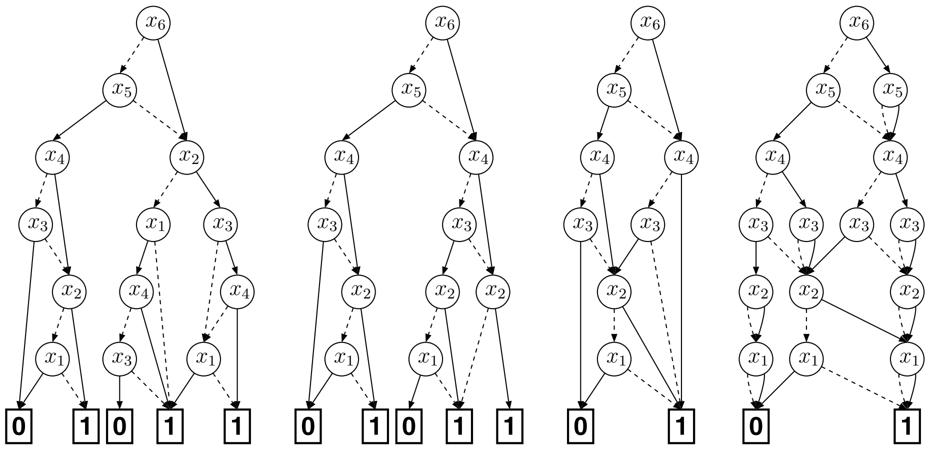


Fig. 1. Example BDDs for the same Boolean function

Figure 1 shows four equivalent data structures, a BDD, an OBDD, an ROBDD and a QROBDD, each representing the same Boolean function, f. Tracing paths from the root node to the terminal nodes of the data structures, we can see, for example, that $f(0,0,1,0,0,1) = 1$ and $f(0,1,0,1,1,1) = 0$. The most commonly used of these four variants is the ROBDD and this will also be the case in this paper. For simplicity, and by convention, from this point on we shall refer to ROBDDs simply as BDDs. On the occasions where we require QROBDDs, this will be stated explicitly.

It is important to note that the reduction rules for BDDs described in the previous paragraphs have no effect on the function being represented. They do, however, typically result in a significant decrease in the number of BDD nodes. More importantly still, as shown by Bryant [13], for a fixed ordering of the Boolean variables, BDDs are a *canonical* representation. This means that there is a one-to-one correspondence between BDDs and the Boolean functions they represent. Similarly, it can also be shown that QROBDDs are a canonical representation for a fixed variable ordering.

The canonical nature of BDDs has important implications for efficiency. For example, it makes checking whether or not two BDDs represent the same function very easy. This is an important operation in many situations, such as the implementation of iterative fixed-point computations. In practice, these reductions are taken one step further. Many BDD packages will actually store all BDDs in a single, multi-rooted graph structure, known as the *unique-table*, where no two nodes are duplicated. This means that comparing two BDDs for equality is as simple as checking whether they are stored in the same place in memory.

It is also important to note that the choice of an ordering for the Boolean variables of a BDD can have a tremendous effect on the size of the data structure, i.e. its number of nodes. Finding the optimal variable ordering, however, is known to be computationally expensive [10]. For this reason, the efficiency of

BDDs in practice is largely reliant on the development of application-dependent *heuristics* to select an appropriate ordering, e.g. [38]. There also exist techniques such as *dynamic variable reordering*, which can be used to change the ordering for an existing BDD in an attempt to reduce its size, see e.g. [66].

BDD Operations. One of the main appeals of BDDs is the efficient algorithms for their manipulation which have been developed, e.g. [13, 14, 12]. A common BDD operation is the ITE (IfThenElse) operator, which takes three BDDs, B_1, B_2 and B_3, and returns the BDD representing the function "if f_{B_1} then f_{B_2} else f_{B_3}". The ITE operator can be implemented recursively, based on the property $\mathsf{ITE}(\mathsf{B}_1, \mathsf{B}_2, \mathsf{B}_3)|_{x_k=b} = \mathsf{ITE}(\mathsf{B}_1|_{x_k=b}, \mathsf{B}_2|_{x_k=b}, \mathsf{B}_3|_{x_k=b})$.

The algorithm to perform this is shown in Figure 2. Lines 1–8 cover the trivial base cases. In lines 14 and 15, the algorithm splits recursively, based on the cofactors of the three BDD operands. The variable, x_k, used to generate these cofactors is the top-most variable between the three BDDs. The resulting BDD, H, is generated by attaching the BDDs from the two recursive calls to a node labeled with x_k. In line 16, reduction of H is performed. Assuming that the operands to the algorithm were already reduced, the only two checks required here are that: (a) *then*(H) and *else*(H) are distinct; and (b) the root node of H does not already exist.

```
ITE(in: $B_1, B_2, B_3$)
  • $B_1, B_2, B_3$ are BDD nodes
 1: if $B_1 = 0$ then
 2:     Return $B_3$;
 3: else if $B_1 = 1$ then
 4:     Return $B_2$;
 5: else if $(B_2 = 1) \wedge (B_3 = 0)$ then
 6:     Return $B_1$;
 7: else if $B_2 = B_3$ then
 8:     Return $B_2$;
 9: else if $\exists$ computed-table entry $(B_1, B_2, B_3, H)$ then
10:     Return H;
11: end if
12: $x_k \leftarrow$top variable of $B_1, B_2, B_3$;
13: $H \leftarrow$ new non-terminal node with label $x_k$;
14: $then(H) \leftarrow ITE(B_1|_{x_k=1}, B_2|_{x_k=1}, B_3|_{x_k=1})$;
15: $else(H) \leftarrow ITE(B_1|_{x_k=0}, B_2|_{x_k=0}, B_3|_{x_k=0})$;
16: Reduce(H);
17: Add entry $(B_1, B_2, B_3, H)$ to computed-table;
18: Return H;
```

Fig. 2. Algorithm for the BDD operator ITE

A crucial factor in the efficiency of the ITE algorithm is the *computed-table*, which is used to cache the result of each intermediate call to the algorithm. Notice how, in line 9, entries in the cache are checked and reused if possible. In practice, many recursive calls would typically be repeated without this step. Furthermore, this means that the computational complexity of the algorithm is bounded by the maximum number of distinct recursive calls to ITE, $\mathcal{O}(|\mathsf{B}_1|\cdot|\mathsf{B}_2|\cdot|\mathsf{B}_3|)$, where $|\mathsf{B}|$ denotes the number of nodes in BDD B.

Another common operation is Apply, which takes two BDDs, B_1 and B_2, plus a binary Boolean operator *op*, such as $\wedge$ or $\vee$, and produces the BDD which represents the function $f_{\mathsf{B}_1}\, op\, f_{\mathsf{B}_2}$. For convenience we often express such operations in infix notation, for example, $\mathsf{B}_1 \vee \mathsf{B}_2 \equiv \mathsf{Apply}(\vee, \mathsf{B}_1, \mathsf{B}_2)$. The Apply operator can be implemented using a recursive algorithm similar to the one for ITE, described above, again making use of the computed-table. Alternatively, any Apply operator can be expressed using ITE; e.g. $\mathsf{Apply}(\wedge, \mathsf{B}_1, \mathsf{B}_2) \equiv \mathsf{ITE}(\mathsf{B}_1, \mathsf{B}_2, \mathsf{0})$. The latter has the advantage that it is likely to increase the hit-rate in the computed-table cache since only one operation is required, as opposed to several operations with their own computed-table caches (one for each binary Boolean operator *op*).

2.2 Multi-Valued Decision Diagrams

Multi-valued decision diagrams (MDDs) are also rooted, directed, acyclic graphs [47]. An MDD is associated with a set of K variables, $x_K, \ldots, x_1$, and an MDD M represents a function $f_{\mathsf{M}} : \mathbb{N}_K \times \cdots \times \mathbb{N}_1 \rightarrow \mathbb{M}$, where $\mathbb{N}_k$ is the finite set of values that variable x_k can assume, and $\mathbb{M}$ is the finite set of possible function values. It is usually assumed that $\mathbb{N}_k = \{0, \ldots, N_k - 1\}$ and $\mathbb{M} = \{0, \ldots, M - 1\}$ for simplicity. Note that BDDs are the special case of MDDs where $\mathbb{M} = \mathbb{B}$ and $\mathbb{N}_k = \mathbb{B}$ for all k. MDDs are similar to the "shared tree" data structure described in [71].

Like BDDs, MDDs consist of *terminal* nodes and *non-terminal* nodes. The terminal nodes are labeled with an integer from the set $\mathbb{M}$. A non-terminal node m is labeled with a variable $var(\mathsf{m}) \in \{x_K, \ldots, x_1\}$. Since variable x_k can assume values from the set $\mathbb{N}_k$, a non-terminal node m labeled with variable x_k has N_k children, each corresponding to a cofactor $\mathsf{m}|_{x_k = c}$. We refer to child c of node m as $child(\mathsf{m}, c)$, where $f_{child(\mathsf{m},c)} = f_{\mathsf{m}}|_{var(\mathsf{m}) = c}$. Every MDD node corresponds to some integer function.

The BDD notion of ordering can also be applied to MDDs, to produce ordered MDDs (OMDDs). A non-terminal MDD node m is redundant if *all* of its children are identical, i.e., $child(\mathsf{m}, i) = child(\mathsf{m}, j)$ for all $i, j \in \mathbb{N}_{var(\mathsf{m})}$. Two non-terminal MDD nodes m_1 and m_2 are duplicates if $var(\mathsf{m}_1) = var(\mathsf{m}_2)$ and $child(\mathsf{m}_1, i) = child(\mathsf{m}_2, i)$ for all $i \in \mathbb{N}_{var(\mathsf{m})}$. Based on the above definitions, we can extend the notion of reduced and quasi-reduced BDDs to apply also to MDDs. It can be shown [47] that reduced OMDDs (ROMDDs) are a canonical representation for a fixed variable ordering. The same can also be shown for quasi-reduced OMDDs (QROMDDs). Finally, like BDDs, the number of ROMDD nodes required to represent a function may be sensitive to the chosen variable ordering.

Example MDDs are shown in Figure 3, all representing the same function over three variables, x_3, x_2, x_1 with $N_3 = N_2 = N_1 = 4$ and $M = 3$. The value

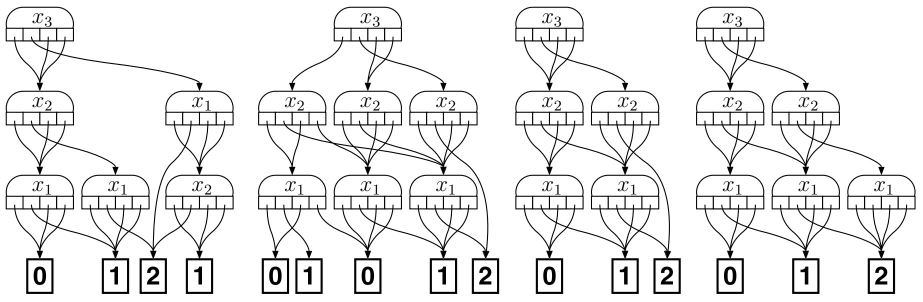


Fig. 3. Example MDDs for the same function

of the function is zero if none of the variables has value 1, one if exactly one of the variables has value 1, and two if two or more of the variables have value 1. Figure 3(a) shows an MDD that is not ordered, Figure 3(b) shows an OMDD that is not reduced, and Figure 3(c) and Figure 3(d) show the ROMDD and QROMDD for the function, for the given variable ordering. Unless otherwise stated, the remainder of the paper will assume that all MDDs are ROMDDs.

MDD Operations. Like BDDs, MDDs can be manipulated using various operators. One such operator is the Case operator, defined in [47] as:

$$\mathsf{Case}(\mathsf{F}, \mathsf{G}^0, \ldots, \mathsf{G}^{M-1})(x_K, \ldots, x_1) = \mathsf{G}^{f_{\mathsf{F}}(x_K,\ldots,x_1)}(x_K, \ldots, x_1).$$

Thus, Case selects the appropriate MDD G^i based on the value of F. Note that Case is a generalization of ITE for BDDs: $\mathsf{ITE}(\mathsf{A}, \mathsf{B}, \mathsf{C}) \equiv \mathsf{Case}(\mathsf{A}, \mathsf{C}, \mathsf{B})$. A recursive algorithm to implement Case for MDDs is presented in [47], and is shown in Figure 4(a). It is based on the recurrence:

$$\mathsf{Case}(\mathsf{F}, \mathsf{G}^0, \ldots, \mathsf{G}^{M-1})|_{x_k=i} = \mathsf{Case}(\mathsf{F}|_{x_k=i}, \mathsf{G}^0|_{x_k=i}, \ldots, \mathsf{G}^{M-1}|_{x_k=i})$$

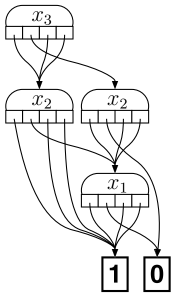
with appropriate terminal conditions (e.g. if F is a constant). The Case algorithm is quite similar to the ITE algorithm: both algorithms first handle the trivial cases, then handle the already-computed cases by checking entries in the computed-table. The computation is performed for the other cases, and the resulting node is reduced. The main difference is that, since Case operates on MDDs, a loop is required to generate the recursive calls to Case, one for each possible value of top variable x_k.

As a simple example, the MDD produced by $\mathsf{Case}(\mathsf{F}, 1, 1, 0)$ is shown in Figure 4(b), where F is the MDD shown in Figure 3(c).

As with ITE, a simple recursive implementation of Case without using a computed-table can be computationally expensive: each call to Case with top variable x_k will generate N_k recursive calls to Case. This leads to a computational complexity of $\mathcal{O}(\prod_{k=1}^{K} N_k)$ (assuming constant time for node reduction), which is often intractable. Again, the number of *distinct* recursive calls to Case

```
Case(in: $F, G^0, \ldots, G^{M-1}$)
  • $F, G^0, \ldots, G^{M-1}$ are MDD nodes
1: if $F = c, c \in \mathbb{M}$ then
2:   Return $G^c$;
3: else if $(G^0 = 0) \wedge \cdots \wedge (G^{M-1} = M-1)$ then
4:   Return F;
5: else if $G^0 = G^1 = \cdots = G^{M-1}$ then
6:   Return $G^0$;
7: else if $\exists$ computed-table entry $(F, G^0, \ldots, G^{M-1}, H)$ then
8:   Return H;
9: end if
10: $x_k \leftarrow$Top variable of $F, G^0, \ldots, G^{M-1}$;
11: $H \leftarrow$ new non-terminal node with label $x_k$;
12: for each $i \in \mathbb{N}_k$ do
13:   $child(H, i) \leftarrow$ Case($F|_{x_k=i}, G^0|_{x_k=i}, \ldots, G^{M-1}|_{x_k=i}$);
14: end for
15: Reduce(H);
16: Add entry $(F, G^0, \ldots, G^{M-1}, H)$ to computed-table;
17: Return H;
```

(a) Algorithm



(b) Example

Fig. 4. MDD operator Case

is bounded by $|F| \cdot |G^0| \cdots |G^{M-1}|$; thus, the use of the computed-table bounds the worst-case computational complexity of Case by:

$$\mathcal{O}(|F| \cdot |G^0| \cdots |G^{M-1}| \cdot \max\{N_K, \ldots, N_1\}),$$

since each non-trivial call to Case with top variable x_k has computational cost of $\mathcal{O}(N_k)$. Note that the resulting MDD will have at most $|F| \cdot |G^0| \cdots |G^{M-1}|$ nodes, regardless of implementation.

2.3 State Set Representation and Generation

We now describe how the data structures introduced in the previous two sections, BDDs and MDDs, can be used to represent and manipulate sets of states of a probabilistic model. We also consider the problem of generating the set of reachable states for a model using these data structures.

Representing Sets of States. To represent a set of states using BDDs or MDDs, each state $\mathbf{s}$ must be expressible as a collection of K state variables, $\mathbf{s} = (x_K, \ldots, x_1)$, where each state variable can assume a finite number of values. If $\mathbb{N}_k$ is the set of possible values for state variable x_k, then the set of all possible states is $\mathcal{S} = \mathbb{N}_K \times \cdots \times \mathbb{N}_1$. Thus, any set of states will be a subset of $\mathcal{S}$.

The basic idea behind BDD and MDD state set representations is to encode the *characteristic function* of a subset $\mathcal{S}'$ of the set of states $\mathcal{S}$, i.e. the function

$\chi_{\mathcal{S}'} : \mathcal{S} \to \{0,1\}$ where $\chi_{\mathcal{S}'}(x_K, \ldots, x_1) = 1$ if and only if $(x_K, \ldots, x_1) \in \mathcal{S}'$. If states are encoded as Boolean vectors, i.e. a state is an element of $\mathbb{B}^K$, then $\mathbb{N}_k = \mathbb{B}$ and the characteristic function can be encoded using a BDD in the usual way. To use BDDs when $\mathbb{N}_k \neq \mathbb{B}$, it becomes necessary to derive some encoding of the state space into Boolean variables. This process must be performed with care since it can have a dramatic effect on the efficiency of the representation. We will return to this issue later. Alternatively, MDDs can encode the characteristic function in a straightforward way as long as each set $\mathbb{N}_k$ is finite.

Example. We now introduce a small example which will be reused in subsequent sections. Consider an extremely simple system consisting of three cooperating processes, P_3, P_2 and P_1, each of which can be in one of four local states, 0, 1, 2 or 3. The (global) state space $\mathcal{S}$ is $\{0,1,2,3\} \times \{0,1,2,3\} \times \{0,1,2,3\}$.

To represent sets of states using MDDs, we can use $K = 3$ state variables, x_3, x_2 and x_1, with $N_3 = N_2 = N_1 = 4$. To represent sets of states using BDDs, we must adopt a Boolean encoding. In this simple example, we can allocate two Boolean variables to represent each local state, and use the standard binary encoding for the integers $\{0,1,2,3\}$. Hence, we need $K = 6$ variables, where x_{2n}, x_{2n-1} represent the state of process P_n.

Let us suppose that when a process in local state 1, it is modifying shared data and requires exclusive access to do so safely. Consider the set of states $\mathcal{S}'$, in which at most one process is in local state 1. The MDD and BDD representing $\mathcal{S}'$ can be seen in Figure 4(b) and Figure 1(c), respectively.

Manipulating and Generating Sets of States. Basic manipulation of state sets can easily be translated into BDD or MDD operations. For example, the union and intersection of two sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, represented by BDDs, S_1 and S_2, can be computed using $\mathsf{Apply}(\vee, \mathsf{S}_1, \mathsf{S}_2)$ and $\mathsf{Apply}(\wedge, \mathsf{S}_1, \mathsf{S}_2)$, respectively. Similarly, sets $\mathcal{S}_1$ and $\mathcal{S}_2$, represented by MDDs S_1 and S_2, can be manipulated using Case; for example, $\mathcal{S}_1 \setminus \mathcal{S}_2$ and $\mathcal{S}_1 \cup \mathcal{S}_2$ can be computed using $\mathsf{Case}(\mathsf{S}_2, \mathsf{S}_1, 0)$ and $\mathsf{Case}(\mathsf{S}_1, \mathsf{S}_2, 1)$, respectively.

Of particular interest are the operations required to compute the BDD or MDD representing the set of reachable states of a model. To do so effectively requires a *next-state* function, which reports the states that are reachable from a given state in a single step. The next-state function must be represented in a format that is suitable for BDD and MDD manipulation algorithms.

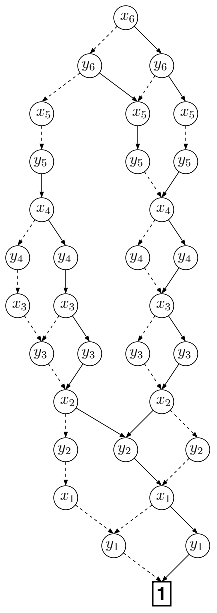
One approach is to represent the next-state function using another BDD or MDD. Since the next-state function is a relation R over pairs of states from a set $\mathcal{S}$, a BDD or MDD can encode its characteristic function $\chi_R : \mathcal{S} \times \mathcal{S} \to \{0,1\}$ where $\chi_R(\mathbf{s}, \mathbf{s}') = 1$ if and only if $(\mathbf{s}, \mathbf{s}') \in R$ for all $\mathbf{s}, \mathbf{s}' \in \mathcal{S}$. Note that this BDD or MDD R requires two sets of variables, $\{x_K, \ldots, x_1\}$ and $\{y_K, \ldots, y_1\}$, where $f_{\mathsf{R}}(x_K, \ldots, x_1, y_K, \ldots, y_1) = \chi_R(\mathbf{s}, \mathbf{s}')$; variables $\{x_K, \ldots, x_1\}$ represent the "source" states, while variables $\{y_K, \ldots, y_1\}$ represent the "target" states.

Given a BDD or MDD S' in variables $\{x_K, \ldots, x_1\}$, representing a set of states $\mathcal{S}'$, the BDD or MDD S'' for the set of states $\mathcal{S}''$ reachable in exactly one step from any state in $\mathcal{S}'$ can be computed relatively easily, a process sometimes known as *image computation*. With BDDs, for example, we can use the following:

$$\mathsf{S}'' = \exists\{x_K \ldots, x_1\}(\mathsf{S}' \wedge \mathsf{R})$$

where $\exists x_k.\mathsf{B} = \mathsf{B}|_{x_k=0} \vee \mathsf{B}|_{x_k=1}$ and $\exists\{x_K, \ldots, x_1\}.\mathsf{B} = \exists x_K \ldots \exists x_1.\mathsf{B}$. Note that S'', the resulting BDD, will be in variables $\{y_K, \ldots, y_1\}$ but can easily be converted back to $\{x_K, \ldots, x_1\}$ if required.

Returning to our running example from the previous section, suppose that a process can asynchronously change from local states 1 to 2, from 2 to 3, or from 3 to 0. Processes can also change from local state 0 to 1, but only if no other process is already in its local state 1. The portion of the next-state function that characterizes the possible local state changes of process P_3, represented by a BDD over variables $\{x_6, \ldots, x_1\}, \{y_6, \ldots, y_1\}$, is shown in Figure 5(a), where terminal node 0 and all of its incoming arcs are omitted for clarity. The overall next-state function can be obtained by constructing similar BDDs for processes P_2 and P_1, and combining them using $\mathsf{Apply}(\vee, \cdot, \cdot)$. Note that the variable ordering of the BDD in Figure 5(a) interleaves the variables for "source" and "target" states. This is a well-known heuristic for reducing the size of BDDs which represent transition relations [36].



(a) BDD

Relation	**Table**
$R_3^{e3} = \{(0,1)\}$	$\begin{bmatrix} 0&1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{bmatrix}$
$R_2^{e3} = R_1^{e3} = \{(0,0),(2,2),(3,3)\}$	$\begin{bmatrix} 1&0&0&0 \\ 0&0&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{bmatrix}$
$R_3^{e6} = \{(1,2),(2,3),(3,0)\}$	$\begin{bmatrix} 0&0&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \\ 1&0&0&0 \end{bmatrix}$
$R_2^{e6} = R_1^{e6} = \{(0,0),(1,1),(2,2),(3,3)\}$	$\begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{bmatrix}$

(b) Tabular

Fig. 5. Next-state function representations

Another approach is to decompose the next-state function and represent its parts using tables. In practice, models are usually described using some high-level formalism; this approach requires to represent a separate next-state function for each model "event" (e.g. a Petri net transition) [60]. This implies that the next-state relation R can be expressed as $\chi_R(\mathbf{s}, \mathbf{s}') = \bigvee_{e \in \mathcal{E}} \chi_{R^e}(\mathbf{s}, \mathbf{s}')$, where R^e represents the next-state relation due to event e, and $\mathcal{E}$ is the (finite) set of all possible events. To represent the relation R^e efficiently, it is required that R^e can be expressed in a "product-form"

$$\chi_{R^e}((x_K, \ldots, x_1), (y_K, \ldots, y_1)) = \chi_{R^e_K}(x_K, y_K) \wedge \cdots \wedge \chi_{R^e_1}(x_1, y_1)$$

of "local" relations R^e_k. Each relation R^e_k can be stored using a Boolean table of size $N_k \times N_k$ (or in a sparse format).

Returning again to the running example, the next-state relation can be divided into events $e_1, \ldots, e_6$, where events e_k correspond to process P_k changing from local state 0 to 1, and events e_{3+k} correspond to process P_k changing from local state 1 to 2, 2 to 3, or 3 to 0. The local relations required to represent R^{e_3} and R^{e_6}, and their corresponding Boolean tables, are shown in Figure 5(b). Representations for relations R^{e_1} and R^{e_2} are similar to R^{e_3}, and the representations for R^{e_4} and R^{e_5} are similar to R^{e_6}.

Techniques based on tabular representation of the next-state function are typically applied to MDD-based storage of sets, rather than BDD-based storage (often the product-form requirement does not allow decomposition of the next-state function into Boolean components). Given an MDD S' representing a set of states $\mathcal{S}'$, and the "product-form" representation for relation R^e, the set of states reachable in exactly one step via event e from any state in $\mathcal{S}'$ can be determined using recursive algorithm Next, shown in Figure 6. Note that the algorithm should be invoked with $k = K$ at the top level. The overall set of states reachable in one step from $\mathcal{S}'$ can be determined by calling Next once for each event and taking the union of the obtained sets.

Using either of the two approaches outlined above for determining the states reachable in one step from a given set of states, it is relatively trivial to compute the set of all reachable states of a model. Starting with the BDD or MDD for some set of initial states, we iteratively compute all the states reachable in one step from the set of reachable states already discovered. These newly explored states are then added to the set of discovered states using set union. The algorithm can be terminated when the latter set ceases to increase.

This process equates to performing a breadth-first search of the model's state space. BDDs and MDDs are often well suited to this purpose for several reasons. Firstly, they can be extremely good at storing and manipulating *sets* of states, and exploiting regularity in these sets. Algorithms which require manipulation of *individual* states, such as depth-first search are less well suited. Secondly, checking for convergence of the algorithm, which requires verification that two sets are identical, reduces to testing two BDDs or MDDs for equality, a process which is efficient due to their canonicity.

It is important to emphasize that the state space generation techniques presented here are actually quite simplistic. Since the bulk of this paper focuses on issues specific to *probabilistic* models, we provide only an introductory presentation to this area. A wide range of much more sophisticated approaches,

```
Next(in: $k$, $R^e$, $\mathsf{S}'$)
  • $R^e$ is the product of local relations, $R^e_K \times \cdots \times R^e_1$.
  • Returns states reachable from $\mathsf{S}'$ in one step via event $e$.
 1: if $\mathsf{S}' = 0$ then
 2:     Return 0;
 3: else if $k = 0$ then
 4:     Return 1;
 5: else if computed-table has entry $(k, R^e, \mathsf{S}', \mathsf{H})$ then
 6:     Return H;
 7: end if
 8: H $\leftarrow$ new non-terminal node with label $x_k$;
 9: for each $i \in \mathbb{N}_k$ do
10:     $child(\mathsf{H}, i) \leftarrow 0$;
11: end for
12: for each $(i, j) \in R^e_k$ do
13:     D $\leftarrow$ Next$(k-1, R^e, \mathsf{S}'|_{x_k=i})$;
14:     $child(\mathsf{H}, j) \leftarrow$ Case$(child(\mathsf{H}, j), \mathsf{D}, 1)$;
15: end for
16: Reduce(H);
17: Add entry $(k, R^e, \mathsf{S}', \mathsf{H})$ to computed-table;
18: Return H;
```

Fig. 6. Image computation using a decomposed next-state relation

particularly those based on BDDs, have been presented in the literature. For instance, more sophisticated BDD-based state space generation techniques for Petri net models are described in [63, 64, 32]. Furthermore, BDDs have proved extremely successful in more complex applications than simple state space generation, such as in formal verification [19, 30]. The well-known (non-probabilistic) symbolic model checker SMV [56], for example, is based on BDD technology.

MDD-based approaches have also proved very successful in practice and, again, a number of enhanced techniques have been developed. Examples are those that exploit the concept of *event locality*: the fact that some events will only affect certain state variables [23, 57, 20]. In particular, [21] introduces the concept of *node saturation*: when a new MDD node is created corresponding to submodel k, it is immediately *saturated* by repeatedly applying all events that affect only submodels k through 1. Recently, more flexible next-state representations have been investigated that allow MDD-based reachability set generation algorithms to handle complex priority structure and immediate events [59], allowing for elimination of vanishing states both on-the-fly and after generation.

2.4 Indexing and Enumerating States

Once the set of reachable states of a model has been determined, it may be used for a variety of purposes. Of particular interest in the case of probabilistic models is the fact that it is usually required when performing numerical computation to analyze the model. Three operations that are commonly required are: (a) determining if a particular state is reachable or not; (b) enumerating all reachable states in order; and (c) determining the *index* of a given state in the set of all states. The latter, for example, is needed when a vector of numerical values, one for each state, is stored explicitly in an array and the index of a state is needed to access its corresponding value.

To perform these three operations efficiently, the BDD or MDD representation of the state space needs to be augmented with additional information. This process is described in [24, 57] for MDDs and in [62] for BDDs. In this section, we will describe how it can be done using the quasi-reduced variants of the data structures (i.e. QROBDDs and QROMDDs). Similar techniques can also be applied to fully reduced decision diagrams, but these require more complex algorithms to correctly handle cases where levels are "skipped" due to redundant nodes. In the following, we will assume that $\mathcal{S}$ denotes the set of all possible states of the model and that $\mathcal{S}' \subseteq \mathcal{S}$ is the set of all reachable states.

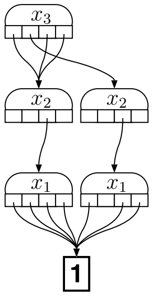
Given a K-variable BDD or MDD representation for $\mathcal{S}'$, determining if an arbitrary state $(s_K, \ldots, s_1)$ is contained in the set $\mathcal{S}'$ can be done by traversing the data structure, as described earlier: if the given variable assignments produce a path leading to terminal node 1, then the state is contained in the set. Note that, if each node has direct access to the downward pointers (i.e., an arbitrary child can be found in constant time), then the computational cost to determine if a state is reachable is exactly $\mathcal{O}(K)$ for reachable states, and at worst $\mathcal{O}(K)$ for unreachable states.

In order to enumerate the set of states $\mathcal{S}'$, one possibility is to go through all the states in $\mathcal{S}$, and for each one, determine if it is in the set $\mathcal{S}'$ as described above. However, this has a computational cost of $\mathcal{O}(K \cdot |\mathcal{S}|)$, which is often unacceptable since it is possible for the set $\mathcal{S}$ to be several orders of magnitude larger than $\mathcal{S}'$. A more efficient approach is to follow all paths in the BDD or MDD for $\mathcal{S}'$ that can lead to terminal node 1. That is, for each node, we visit all children except those whose paths lead only to terminal node 0. Note that nodes whose paths lead only to terminal node 0 represent the constant function 0, and thus can be detected (and passed over) thanks to the canonicity property.

Figure 7(a) shows the recursive algorithm Enumerate, which enumerates all the states represented by QROMDD S', as just described. For the enumeration to be efficient, we must be able to visit only the non-zero children of each node; this can be done quickly if sparse storage is used for the children in each node, or if a list of the non-zero children is kept for each node. Assuming constant time access to the non-zero children (i.e. the ability to find the first or next non-zero child in constant time), it can be shown that algorithm Enumerate has a computational cost of $\mathcal{O}(|\mathcal{S}'|)$ in the best case (when each non-terminal node has several non-zero children) and $\mathcal{O}(K \cdot |\mathcal{S}'|)$ in the worst case (when most non-terminal nodes have a single non-zero child).

Note that, if the values s_k are selected in order in line 5 of Figure 7(a), then the states will be visited in lexicographical order. An example QROMDD, corresponding to the set of reachable states with process 2 in its local state 2 for our running example, is shown in Figure 7(b). Nodes corresponding to the

```
Enumerate(in: $\mathsf{S}'$)
1: if $\mathsf{S}' = 1$ then
2:     Visit($s_K, \ldots, s_1$);
3: else if $\mathsf{S}' \neq 0$ then
4:     $x_k \leftarrow var(\mathsf{S}')$;
5:     for each $s_k : f_{\mathsf{S}'}|_{x_k=s_k} \neq 0$ do
6:         Enumerate($child(\mathsf{S}', s_k)$);
7:     end for
8: end if
```



0: $(0,2,0)$	7: $(2,2,0)$
1: $(0,2,1)$	8: $(2,2,1)$
2: $(0,2,2)$	9: $(2,2,2)$
3: $(0,2,3)$	10: $(2,2,3)$
4: $(1,2,0)$	11: $(3,2,0)$
5: $(1,2,2)$	12: $(3,2,1)$
6: $(1,2,3)$	13: $(3,2,2)$
	14: $(3,2,3)$

(a) Enumeration algorithm (b) QROMDD (c) Visited states

Fig. 7. Enumerating all states in a QROMDD-encoded set

constant function 0 are omitted from the figure for clarity. The states visited by Enumerate for this QROMDD are shown in Figure 7(c), where the integer to the left of the state indicates the order in which states are visited.

For numerical solution, it is often necessary to assign a unique index to each state using some one-to-one indexing function $\psi : \mathcal{S}' \to \{0, \ldots, |\mathcal{S}'| - 1\}$. A commonly-used function ψ is to assign indices in lexicographical order; i.e., $\psi(\mathbf{s})$ is the number of states in $\mathcal{S}'$ that precede $\mathbf{s}$ in lexicographical order. Using an MDD to encode ψ will lead to a rather large MDD, one with $|\mathcal{S}'|$ terminal nodes (this is essentially the multi-level structure described in [23]). An alternative is to use an edge-valued MDD (EV$^+$MDD), a data structure described in [25], to represent the function ψ. An EV$^+$MDD is an MDD where each edge in the MDD from a non-terminal node m to child c has a corresponding value, $value(\mathsf{m}, c)$. The edge values are summed along a path to obtain the function value. With certain restrictions, EV$^+$MDDs are also a canonical representation for functions [25]. The edge values are sometimes referred to as *offsets* [24, 57, 62].

Given a QROMDD representing a set $\mathcal{S}'$, the EV$^+$MDD for ψ, corresponding to the lexicographical indexing function for states, can be constructed by assigning appropriate edge values in a bottom-up fashion, using algorithm BuildOffsets shown in Figure 8. The algorithm is based on the property that the index of a state is equal to the number of paths "before" the path of that state (in lexicographical order), which can be counted by enumerating the paths. Indeed, algorithm BuildOffsets is quite similar to algorithm Enumerate, except that once the paths from a given node to terminal node 1 have been counted, they do not need to be counted again. A computed-table is used to keep track of the number of paths from each non-terminal node to node 1. For the QROMDD shown in Figure 7(b), the resulting MDD with offset values (i.e., the EV$^+$MDD) produced by algorithm Enumerate is shown in Figure 9(a), where the state indices match those shown in Figure 7(c). For instance, the index for state $(3, 2, 1)$ is found by following the appropriate path through the EV$^+$MDD and summing the edge values $11 + 0 + 1 = 12$. Note that the edge values are unnecessary for children corresponding to the constant function 0, since it is impossible to reach terminal node 1 following these paths. As such, edge values for these children can be set

```
BuildOffsets(in: S')
  • Returns the number of paths from S' to terminal 1.
 1: if $\mathsf{S}' = 0$ then
 2:     Return 0;
 3: else if $\mathsf{S}' = 1$ then
 4:     Return 1;
 5: else if computed-table contains entry $(\mathsf{S}', n)$ then
 6:     Return $n$;
 7: end if
 8: $x_k \leftarrow var(\mathsf{S}')$;
 9: $n \leftarrow 0$;
10: for each $s_k : f_{\mathsf{S}'}|_{x_k=s_k} \neq 0$ do
11:     $value(\mathsf{S}', s_k) \leftarrow n$;
12:     $n \leftarrow n +$ BuildOffsets($child(\mathsf{S}', s_k)$);
13: end for
14: Add $(\mathsf{S}', n)$ to computed-table;
15: Return $n$;
```

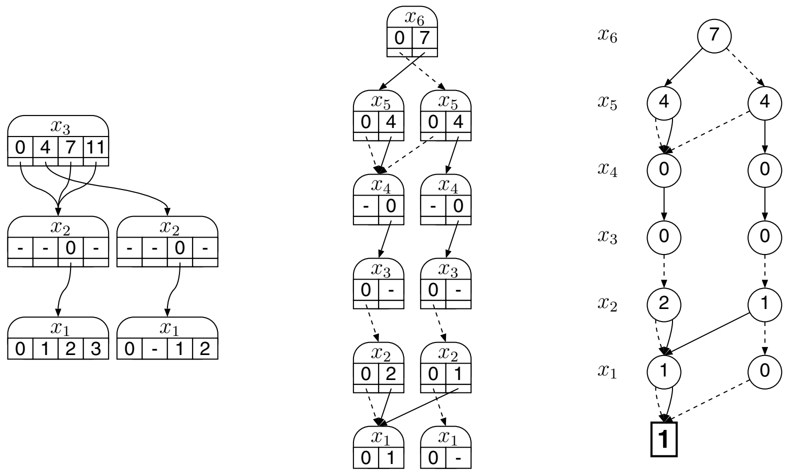
Fig. 8. Constructing "offset" edge-values

to any value, including "undefined", since the values will never be used; these are represented by dashes in Figure 9(a). Note that the pointers in the bottom-level nodes become unnecessary in this case: if the edge value is undefined, then the child must be terminal node 0, otherwise the child is terminal node 1. Other implementation tricks, including how to combine offsets with direct access and non-zero-only access, are described in [24, 57].

The QROBDD version of the set $\mathcal{S}'$ and indexing function ψ is shown in Figure 9(b). In this case, the MDD-encoded state $(3, 2, 1)$ is encoded as the binary state $(1, 1, 1, 0, 0, 1)$, and the state index is given by $7+4+0+0+0+1 = 12$. However, in the special case of BDDs, a simplification is possible. Since child 0 will always have either an edge value of zero or an undefined edge value (this is true for both MDDs and BDDs), the edge value for the *else* child in a BDD does not need to be explicitly stored in the node. Instead, it is sufficient to store only the edge value for the *then* child. This produces the QROBDD shown in Figure 9(c), where the label for each non-terminal node is the edge value for the *then* child (the variable for each node is shown to the left, for all nodes with the same variable). Note that the dashed values of Figure 9(b) have been changed to zeroes in Figure 9(c) by choice [62]. In this case, when tracing a path through the BDD corresponding to a particular state, the index can be computed by summing the offsets on nodes from which the *then* child was taken. For the example, the index for state $(1, 1, 1, 0, 0, 1)$ is given by $7 + 4 + 0 + 1 = 12$.

3 Model Representation

In the previous section, we have seen ways of storing and manipulating sets of states of a probabilistic model, in particular, the set of all of its reachable states.



(a) MDD with offset values (b) BDD with offset values (c) Offset-labeled BDD

Fig. 9. Decision diagrams with offset information

The latter can provide some useful information about the model, for example, whether or not it is possible to reach a state which constitutes an error or failure. Typically, though, we are interested in more involved properties, for example, the *probability* of reaching such a state. Studying properties of this kind will generally require numerical calculation, which needs access to the model itself, i.e. not just the set of reachable states, but the transitions between these states and the probabilistic information assigned to them.

In this section, we concern ourselves with the storage of the model. For two of the most common types of probabilistic models, DTMCs and CTMCs, this representation takes the form of a real-valued square matrix. In the following sections we will see how symbolic data structures such as BDDs and MDDs can be extended to achieve this task. We will also mention extensions for more complex models such as MDPs. The two principal data structures covered are *multi-terminal binary decision diagrams* (MTBDDs) and *matrix diagrams*. The latter is based on the *Kronecker representation*. We also provide an introduction to this area. In addition, we will discuss two further data structures: decision node binary decision diagrams (DNBDDs) and probabilistic decision graphs (PDGs).

3.1 Multi-Terminal Binary Decision Diagrams (MTBDDs)

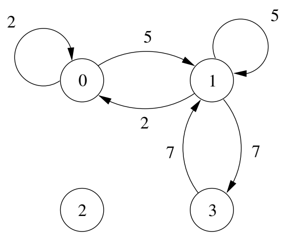
Multi-terminal binary decision diagrams (MTBDDs) are an extension of BDDs. The difference is that terminal nodes in an MTBDD can be labeled with arbitrary values, rather than just 0 and 1 as in a BDD. In typical usage, these values are real (or in practice, floating point). Hence, an MTBDD M over K Boolean variables $x_K, \ldots, x_1$ now represents a function of the form $f_{\mathsf{M}} : \mathbb{B}^K \to \mathbb{R}$.

The basic idea behind MTBDDs was originally presented by Clarke et al. in [28]. They were developed further, independently, by Clarke et al. [27] and

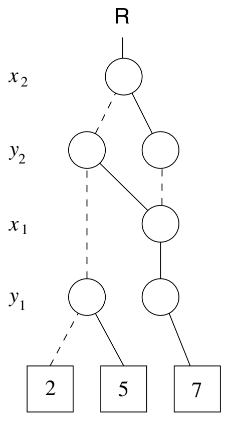
Bahar et al. [5], although in the latter they were christened algebraic decision diagrams (ADDs).

These papers also proposed the idea of using MTBDDs to represent vectors and matrices. Consider a real-valued vector $\underline{v}$ of size 2^K. This can be represented by a mapping from integer indices to the reals, i.e. $f_{\underline{v}} : \{0, \ldots, 2^K - 1\} \to \mathbb{R}$. Given a suitable encoding of these integer indices into K Boolean variables, this can instead be expressed as a function of the form $f_{\mathsf{V}} : \mathbb{B}^K \to \mathbb{R}$, which is exactly what is represented by an MTBDD V over K Boolean variables. Similarly, a real-valued matrix $\mathbf{M}$ of size $2^K \times 2^K$ can be interpreted as a mapping from pairs of integer indices to the reals, i.e. $f_{\mathbf{M}} : \{0, \ldots, 2^K - 1\} \times \{0, \ldots, 2^K - 1\} \to \mathbb{R}$, and hence also as $f_{\mathsf{M}} : \mathbb{B}^K \times \mathbb{B}^K \to \mathbb{R}$, which can represented by an MTBDD M over $2K$ Boolean variables, K of which encode row indices and K of which encode column indices.

Figure 10 shows an example of an MTBDD R which represents a 4×4 matrix $\mathbf{R}$. Since we are introducing MTBDDs in the context of a representation for probabilistic models, the matrix $\mathbf{R}$ is actually the transition rate matrix for a 4 state CTMC. This CTMC is also shown in Figure 10. Note that structure of an MTBDD is identical to a BDD, except for the presence of multiple terminal nodes labeled with real values. The function f_{R} represented by the MTBDD R can also be read off in identical fashion to a BDD. This process is illustrated by the rightmost five columns of the table in Figure 10.



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



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

Fig. 10. A CTMC, its transition rate matrix $\mathbf{R}$ and an MTBDD R representing it

The table also demonstrates how the matrix $\mathbf{R}$ is represented by the MTBDD. We use four Boolean variables: x_2 and x_1, which are used to encode row indices; and y_2 and y_1, which are used to encode column indices. In both cases, we have used the standard binary encoding of integers. For the matrix entry $(3,1)$, for example, $x_2 = 1$ and $x_1 = 1$ encode the row index 3 and $y_2 = 0$ and $y_1 = 1$ encode the column index 1. Note that, in the MTBDD, the variables for rows and columns are interleaved, i.e. $x_2 > y_2 > x_1 > y_1$. This is a well-known heuristic for MTBDDs which represent transition matrices (or analogously, as mentioned previously in Section 2.3, for BDDs which represent transition relations) and typically gives a significant improvement in the size of the data structure. Therefore, to establish the value of entry $(3,1)$ in $\mathbf{R}$, we trace the path $1,0,1,1$ through R and determine that it is equal to 7.

MTBDD Operations. Operations on MTBDDs can be defined and implemented on MTBDDs in a similar way to BDDs. The Apply operator, for example, extends naturally to this data structure. In this case, the operation applied can be a real-valued function, such as addition or multiplication, for example $\mathsf{Apply}(+, \mathsf{M}_1, \mathsf{M}_2)$ produces the MTBDD representing the function $f_{\mathsf{M}_2} + f_{\mathsf{M}_2}$. The MTBDD version of Apply can be implemented in essentially identical fashion to the BDD version, based on a recursive descent of the data structures and using a computed-table to cache intermediate results. The only difference is the handling of the terminal cases. Its time complexity is $\mathcal{O}(|\mathsf{M}_1| \cdot |\mathsf{M}_2|)$, where $|\mathsf{M}|$ denotes the number of nodes in MTBDD M.

We are particularly interested in operations which can be applied to MTBDDs representing matrices and vectors. The Apply operation can be used to perform point-wise operations such as addition of two matrices or scalar multiplication of a matrix by a real constant. Of more interest are operations specifically tailored to matrices and vectors. The most obvious example is the matrix-vector or matrix-matrix multiplication operation. Crucially, because matrix-based multiplications can be expressed in a recursive fashion, they can be efficiently implemented using a similar approach as for the Apply operator. Three slight variants have been presented in [28, 27, 5]. In [5], empirical results are presented to compare the efficiency of the three algorithms.

Model Representation with MTBDDs. Some classes of probabilistic models, such as DTMCs and CTMCs, are described simply by a real valued matrix and can hence be represented by an MTBDD using the techniques described above. These observations have been made in numerous places, e.g. [39, 40, 8, 44]. We have already seen an example of the representation for a CTMC in Figure 10. In addition, we describe here how a third type of model, Markov decision processes (MDPs), can be represented as MTBDDs. This was initially proposed in [7, 6], the first concrete implementation of the idea was presented in [34] and the ideas have since been extended in [62].

The chief difference between MDPs and DTMCs or CTMCs is the presence of nondeterminism. A DTMC or CTMC is specified by the likelihood of making a transition from each state to any other state. This is a real value, a discrete probability for a DTMC or a parameter of an exponential distribution for a CTMC. In either case, the values for all pairs of states can be represented by a square, real-valued matrix. In an MDP, each state contains several nondeterministic choices, each of which specifies a discrete probability for every state. In terms of a matrix, this equates to each state being represented by several different rows. This can be thought of either as a non-square matrix or as a three-dimensional matrix. In any case, we can treat non-determinism as a third index, meaning that an MDP is effectively a function of the form $f : \mathcal{S} \times \{1, \ldots, n\} \times \mathcal{S} \rightarrow [0, 1]$, where $\mathcal{S}$ is the set of states and n is the maximum number of nondeterministic choices in any state. By encoding the set $\{1, \ldots, n\}$ with Boolean variables, we can store the MDP as an MTBDD over three sets of variables, one for source states, one for destination states and one to distinguish between nondeterministic choices.

Model Generation with MTBDDs. The process of generating the MTBDD which represents a given probabilistic model is particularly important as it can have a huge impact on the efficiency of the MTBDD as a representation. This is an issue considered in [44], which presents a number of heuristics for this purpose. The most important of these is that one should try to exploit *structure* and *regularity*. In practice, a probabilistic model will be described in some high-level specification formalism, such as a stochastic process algebra, stochastic Petri nets or some other, custom language. Typically, this high-level description is inherently structured. Therefore, the most efficient way to construct the MTBDD representing the model is via a direct translation from the high-level description. This is demonstrated in [44] on some simple examples using formalisms such as process algebras and queuing networks. These findings have been confirmed by others, e.g. [34, 49, 62, 43] on a range of formalisms. Like BDDs, attention should also be paid to the ordering of the Boolean variables in the MTBDD. Discussions can be found in [44, 62, 43].

Direct translation of a model from its high-level description usually results in the introduction of unreachable states. This necessitates the process of reachability: computing all reachable states. This can be done with BDDs, as discussed in Section 2, and then the unreachable states can be removed from the MTBDD with a simple Apply operation. This process is facilitated by the close relation between BDDs and MTBDDs.

In practice, it has been shown that MTBDDs can be used to construct and store extremely large probabilistic models [39, 44, 34]. This is possible by exploitation of high-level structure in the description of the model. Often, the construction process itself is also found to be efficient. This is partly due to the fact that the symbolic representation is constructed directly from the high-level description and partly because the process can integrate existing efficient techniques for BDD-based reachability.

3.2 Kronecker Algebra

A well-accepted compact representation for certain types of models (usually CTMCs) is based on Kronecker algebra. Relevant well-known properties of Kronecker products are reviewed here; details can be found (for example) in [33]. The Kronecker product of two matrices multiplies every element of the first matrix by every element of the second matrix. Given square matrices $\mathbf{M}_2$ of dimension N_2 and $\mathbf{M}_1$ of dimension N_1, their Kronecker product

$$\mathbf{M}_2 \otimes \mathbf{M}_1 = \begin{bmatrix} \mathbf{M}_2[0,0] \cdot \mathbf{M}_1 & \cdots & \mathbf{M}_2[0,N_2-1] \cdot \mathbf{M}_1 \\ \vdots & \ddots & \vdots \\ \mathbf{M}_2[N_2-1,0] \cdot \mathbf{M}_1 & \cdots & \mathbf{M}_2[N_2-1,N_2-1] \cdot \mathbf{M}_1 \end{bmatrix}$$

is a square matrix of dimension $N_2 N_1$. This can be generalized to the Kronecker product of K matrices $\mathbf{M} = \mathbf{M}_K \otimes \cdots \otimes \mathbf{M}_1$ since the Kronecker product is associative. In this case, the square matrix $\mathbf{M}$ has dimension $\prod_{k=1}^{K} N_k$, where N_k is the dimension of square matrix $\mathbf{M}_k$.

Kronecker algebra has been fairly successful in representing large CTMCs generated from various high-level formalisms [16, 35, 37, 46, 65]. The key idea behind Kronecker-based approaches is to represent the portion of the transition rate matrix for the CTMC due to a single event e, denoted as $\mathbf{R}^e$, as the Kronecker product of matrices $\mathbf{W}_k^e$, which describe the contribution to event e due to each model component k. The overall transition rate matrix is the sum of each $\mathbf{R}^e$ over all events e. Note that each matrix $\mathbf{W}_k^e$ is a square matrix of dimension N_k, and the dimension of the represented matrix is $N_K \cdots N_1$. Thus, just like MDDs, to use a Kronecker representation, a model state must be expressible as a collection of K state variables $(s_K, \ldots, s_1)$, where $s_k \in \mathbb{N}_k$. Unless matrix elements are allowed to be functions [37], the state variables and events must be chosen so that the rate of transition from state $(s_K, \ldots, s_1)$ to state $(s'_K, \ldots, s'_1)$ due to event e can be expressed as the product

$$\lambda^e((s_K, \ldots, s_1), (s'_K, \ldots, s_1)) = \lambda_K^e(s_K, s'_K) \cdots \lambda_1^e(s_1, s'_1)$$

which means that the transition rates cannot arbitrarily depend on the global state. When the above product-form requirement is met, all matrix elements are constants: $\mathbf{W}_k^e[s_k, s'_k] = \lambda_k^e(s_k, s'_k)$.

We now return to the running example. Let us assume that the model is a CTMC, i.e. each transition of the model is associated with a rate. Let the rate of transition from local state 2 to local state 3 be 2.3, from local state 3 to local state 0 be 3.0, from local state 0 to local state 1 be $5.0 + 0.1 \cdot k$ for process k, and from local state 1 to local state 2 be $1.5 + 0.1 \cdot k$ for process k. The corresponding Kronecker matrices for $\mathbf{R}^{e_3}$ and $\mathbf{R}^{e_6}$ are shown in Figure 11. For this example, according to $\mathbf{R}^{e_6}$, the rate of transition from state $(1, 2, 1)$ to state $(2, 2, 1)$ is $1.8 \cdot 1.0 \cdot 1.0 = 1.8$. Note that the matrices $\mathbf{W}_2^{e_6}$ and $\mathbf{W}_1^{e_6}$ are identity matrices; this occurs whenever an event does not affect, and is not affected by, a particular component. Also note the similarities between the matrices of Figure 11 and the tables in Figure 5(b). The tabular next-state representation is in fact a Kronecker representation, where the table for relation R_k^e corresponds to a Boolean matrix $\mathbf{W}_k^e$.

$$\mathbf{W}_3^{e3} = \begin{bmatrix} 0 & 5.3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{W}_2^{e3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{W}_1^{e3} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{W}_3^{e6} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1.8 & 0 \\ 0 & 0 & 0 & 2.3 \\ 3.0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{W}_2^{e6} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{W}_1^{e6} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 11. Kronecker matrices for the running example

For in-depth discussion of Kronecker representations and algorithms for efficient numerical solution of Markov chains expressed using Kronecker algebra, see for example [15, 18, 69].

3.3 Matrix Diagrams

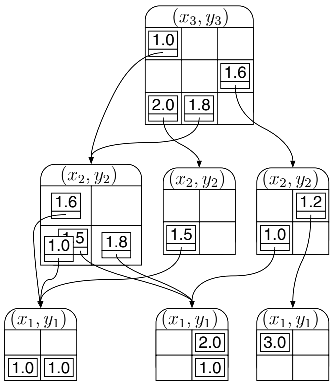
Matrix diagrams (abbreviated as MDs or MxDs) are, like BDDs, MDDs and MTBDDs, rooted directed acyclic graphs. A matrix diagram is associated with K pairs of variables, $(x_K, y_K), \ldots, (x_1, y_1)$, and a matrix diagram M represents a matrix (or function) $f_{\mathsf{M}} : (\mathbb{N}_K \times \cdots \times \mathbb{N}_1)^2 \rightarrow \mathbb{R}$. Note that both variables in a pair have the same possible set of values: $x_k, y_k \in \mathbb{N}_k$. Matrix diagrams consist of terminal nodes labeled with 1 and 0, and non-terminal nodes m labeled with variable pairs $var(\mathsf{m}) \in \{(x_K, y_K), \ldots, (x_1, y_1)\}$. A non-terminal node with label (x_k, y_k) is sometimes called a *level-k* node. According to the original definition [24], for a given node and pair of variable assignments, there is an associated *set* of children nodes, and each child node has a corresponding real edge value. We denote this as a set of pairs, $pairs(\mathsf{m}, i, j) \subset \mathbb{R} \times \mathcal{M}$, where (i, j) are the variable assignments for $var(\mathsf{m})$, and $\mathcal{M}$ is the set of matrix diagram nodes. If a set contains more than one pair, then a given group of variable assignments can lead to multiple paths through the matrix diagram. The value of a matrix element corresponding to the appropriate variable assignments is found by taking the product of the real values encountered along each path, and summing those products over all paths.

Like decision diagrams, the concept of ordering can be applied to obtain ordered MDs (OMDs). Two non-terminal matrix diagram nodes m_1 and m_2 are duplicates if $var(\mathsf{m}_1) = var(\mathsf{m}_2)$ and if $pairs(\mathsf{m}_1, i, j) = pairs(\mathsf{m}_2, i, j)$, for all possible $i, j \in \mathbb{N}_{var(\mathsf{m}_1)}$. The usual notion of quasi-reduction can be applied (using the above definition for duplicates) to obtain quasi-reduced OMDs (QROMDs). An element of the matrix encoded by a QROMD m can be computed using the recurrence:

$$f_{\mathsf{m}}(x_k, \ldots, x_1, y_k, \ldots, y_1) = \sum_{\forall (r, \mathsf{m}') \in pairs(\mathsf{m}, x_k, y_k)} r \cdot f_{\mathsf{m}'}(x_{k-1}, \ldots, x_1, y_{k-1}, \ldots, y_1)$$

with terminating conditions $f_1 = 1$ and $f_0 = 0$. However, QROMDs are *not* a canonical representation. Additional rules can be applied to matrix diagrams to obtain a canonical form, as described in [58]: sets of pairs are eliminated, so that each group of variable assignments corresponds to exactly one path through the matrix diagram, and the possible edge values are restricted. For the remainder of the paper, we assume that all matrix diagrams are QROMDs (with sets of pairs).

A simple example of a matrix diagram is shown in Figure 12(a). Each matrix element within a non-terminal node is either the empty set (represented by blank space) or a set of pairs (represented by stacked boxes). At the bottom level, the only possible child is terminal node 1 (pairs with terminal 0 children or zero edge values can be removed); thus, the pointers can be omitted. The matrix encoded by the matrix diagram is shown in Figure 12(b). The large blocks of zeroes are due to the blank spaces in the (x_3, y_3) matrix diagram node.



(a) Matrix diagram

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.6 & 1.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3.0 & 0 & 3.6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1.0 & 2.5 & 0 & 1.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5.7 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 2.8 & 2.8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 5.4 & 0 & 6.4 & 0 & 0 & 0 & 0 \\
3.0 & 3.0 & 0 & 0 & 1.8 & 4.5 & 0 & 3.2 & 0 & 0 & 0 & 0
\end{bmatrix}$$

(b) Encoded matrix

Fig. 12. Example matrix diagram

Matrix Diagram Operations. In this section, we describe three useful operations for matrix diagrams: constructing a matrix diagram from a Kronecker product, addition of matrix diagrams and selection of a submatrix from a matrix diagram.

Building a Kronecker product: Given matrices $\mathbf{M}_K, \ldots, \mathbf{M}_1$, we can easily construct a matrix diagram representation of their Kronecker product $\mathbf{M}_K \otimes \cdots \otimes \mathbf{M}_1$. The matrix diagram will have one non-terminal node for each of the K matrices, where the level-k node m_k corresponds to matrix $\mathbf{M}_k$, and the entries of node m_k are determined as follows. If $\mathbf{M}_k[i, j]$ is zero, then $\textit{pairs}(\mathsf{m}_k, i, j)$ is the empty set; otherwise, $\textit{pairs}(\mathsf{m}_k, i, j)$ is the set containing only the pair $(\mathbf{M}_k[i, j], \mathsf{m}_{k-1})$. The level-0 node m_0 is terminal node 1. The matrix diagrams obtained for the Kronecker products for the running example are shown in Figure 13. Since the matrix diagrams are QROMDs, the variable pairs for each node are omitted from the figure to save space.

Addition of two matrix diagrams: Given two level-k matrix diagram nodes m_1 and m_2, the level-k matrix diagram node m encoding their sum, i.e., $f_{\mathsf{m}} = f_{\mathsf{m}_1} + f_{\mathsf{m}_2}$, can be easily constructed by set union

$$pairs(\mathsf{m}, i, j) = pairs(\mathsf{m}_1, i, j) \cup pairs(\mathsf{m}_2, i, j)$$

for each possible $i, j \in \mathbb{N}_k$. For example, the sum of matrix diagrams for $\mathbf{R}^{e_1}$ and $\mathbf{R}^{e_2}$ from Figure 13 is shown in Figure 14(a). While this produces a correct matrix diagram, a more compact representation can sometimes be obtained. For instance, if a set contains pairs (r_1, m) and (r_2, m) with the same child m, the two pairs can be replaced with the single pair $(r_1 + r_2, \mathsf{m})$, since $r_1 \cdot \mathbf{M} + r_2 \cdot \mathbf{M} = (r_1 + r_2) \cdot \mathbf{M}$. Note that this implies that level-1 nodes never need to have sets with more than one pair. Reduction may be possible when a set contains pairs (r, m_1) and (r, m_2) with the same edge value r, by replacing the two pairs with the pair (r, m), where m encodes the sum of m_1 and m_2 (computed recursively). When this is done for the sum of $\mathbf{R}^{e_1}$ and $\mathbf{R}^{e_2}$, we obtain the matrix diagram shown in Figure 14(b). Note that this replacement rule does not always reduce the size of the matrix diagram. For canonical matrix diagrams that do not allow sets of pairs [58], addition is still possible but requires more complex algorithms.

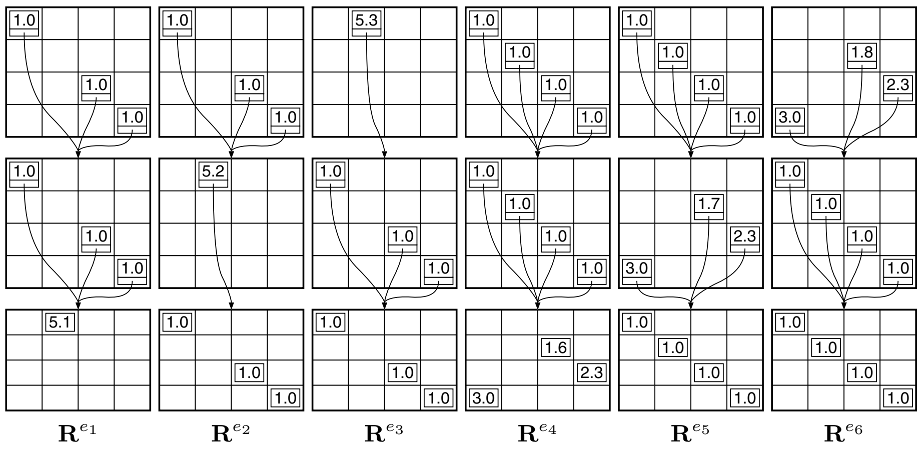


Fig. 13. Matrix diagrams built from Kronecker products for running example

Selecting a submatrix from a matrix diagram: Given a matrix diagram representation of a matrix, we can select a submatrix with specified rows and columns by using MDD representations for the sets of desired rows and columns. Using a QROMDD R over variables $x_K, \ldots, x_1$ for the set of rows, a QROMDD C over variables $y_K, \ldots, y_1$ for the set of columns, and a QROMD M over variable pairs $(x_K, y_K), \ldots, (x_1, y_1)$ allows for a straightforward recursive algorithm, shown in Figure 15(a), to construct $\mathsf{M}' = \mathsf{Submatrix}(\mathsf{M}, \mathsf{R}, \mathsf{C})$. Note that, in addition to checking for duplicate nodes, Reduce also replaces nodes containing no

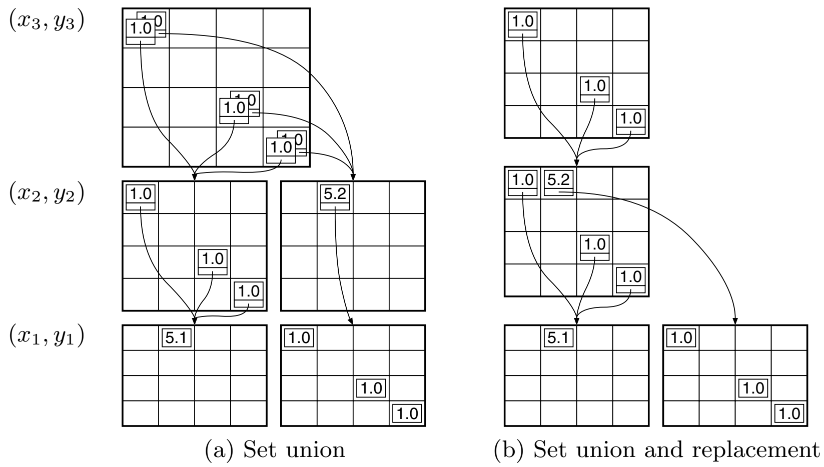


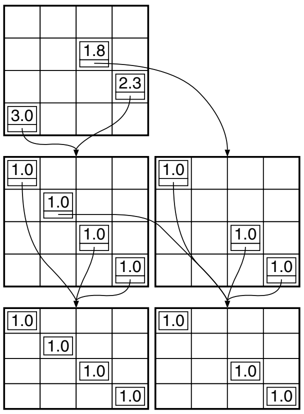
Fig. 14. Addition of $\mathbf{R}^{e_1}$ and $\mathbf{R}^{e_2}$ from Figure 13

pairs (i.e., $pairs(\mathsf{M}, i, j)$ is the empty set for all i, j) with terminal node 0. The Submatrix operator can also be implemented using ROMDD representations for R and C, using a slightly more complex algorithm. Note that it is also possible to physically remove the undesired rows and columns (rather than setting them to zero, as done by Submatrix); this may produce matrices with different dimensions within matrix diagram nodes with the same labels, and leads to more complex implementation [24, 57]. Figure 15(b) shows the matrix diagram obtained using $\mathbf{R}^{e_6}$ (from Figure 13) for the input matrix and the MDD encoding of the set of reachable states (from Figure 4(b)) for the desired rows and columns. Note that the rate of transition from state $(1, 2, 1)$ to state $(2, 2, 1)$ is $1.8 \cdot 1.0 \cdot 0 = 0$ in the obtained matrix diagram, since state $(1, 2, 1)$ is unreachable according to the MDD shown in Figure 4(b).

Model Representation and Construction with Matrix Diagrams. Given a high-level formalism that supports construction of the low-level model (e.g., CTMC) as a sum of Kronecker products, as described in Section 3.2, it is straightforward to construct the corresponding matrix diagram representation. Matrix diagrams can be constructed for each individual Kronecker product; these can then be summed to obtain the overall model. Note that both the overall matrix diagram and the Kronecker representation describe a model with states $\mathcal{S}$. The matrix diagram corresponding to the "reachable" portion of the model can be obtained using the Submatrix operator, using the set of reachable states $\mathcal{S}'$ as the set of desired rows and columns. This operation is important particularly for numerical solution, especially when $|\mathcal{S}| \gg |\mathcal{S}'|$. Note that this requires construction of the MDD representing the reachability set $\mathcal{S}'$, as described in Section 2.3. Summing the matrix diagrams in Figure 13 (using the replacement rules) and selecting the reachable portion of the matrix using Submatrix produces the matrix diagram shown in Figure 16.

```
Submatrix(in: M, R, C)
 1: if $\mathsf{R} = 1 \wedge \mathsf{C} = 1$ then
 2:    Return M;
 3: else if $\mathsf{R} = 0 \vee \mathsf{C} = 0$ then
 4:    Return 0;
 5: else if $\exists$ computed-table entry $(\mathsf{M}, \mathsf{R}, \mathsf{C}, \mathsf{M}')$ then
 6:    Return M′;
 7: end if
 8: $(x_k, y_k) \leftarrow var(\mathsf{M})$;
 9: $\mathsf{M}' \leftarrow$ new non-terminal node with label $(x_k, y_k)$;
10: for each $i, j : f_{\mathsf{R}}|_{x_k=i} \neq 0 \wedge f_{\mathsf{C}}|_{y_k=j} \neq 0$ do
11:    for each $(r, \mathsf{m}) \in pairs(\mathsf{M}, i, j)$ do
12:       $\mathsf{m}' \leftarrow$ Submatrix(m, $child(\mathsf{R}, i)$, $child(\mathsf{C}, j)$);
13:       if $\mathsf{m}' \neq 0$ then
14:          $pairs(\mathsf{M}', i, j) \leftarrow pairs(\mathsf{M}', i, j) \cup \{(r, \mathsf{m}')\}$;
15:       end if
16:    end for
17: end for
18: Reduce(M′);
19: Add entry (M, R, C, M′) to compute table;
20: Return M′;
```

(a) Selection algorithm



(b) Select($\mathbf{R}^{e_6}, \mathcal{S}', \mathcal{S}'$)

Fig. 15. Submatrix selection for matrix diagrams

Another method to construct a matrix diagram is to do so explicitly, by adding each non-zero entry of the matrix to be encoded into the matrix diagram, one at a time [58]. This approach essentially visits each non-zero entry, constructs a matrix diagram representing a matrix containing only the desired non-zero entry, and sums all the matrix diagrams to obtain the representation for the overall matrix. Overhead is kept manageable by mixing unreduced and reduced nodes in the same structure: in-place updates are possible for the unreduced nodes, but may contain duplicates. The unreduced portion is periodically merged with the reduced portion.

3.4 Other Data Structures

MTBDDs and matrix diagrams are not the only symbolic representations proposed for probabilistic models. There are a number of others, typically all extensions in some fashion of the basic BDD data structure. We describe two notable examples here: decision node binary decision diagrams (DNBDDs) and probabilistic decision graphs (PDGs).

Decision node binary decision diagrams (DNBDDs) were proposed by Siegle [67, 68] to represent CTMCs. While MTBDDs extend BDDs by allowing multiple terminal nodes, DNBDDs do so by adding information to certain *then* and *else* edges. Again, the purpose of this information is to encode the real values which the function being represented can take. This approach has the advantage that

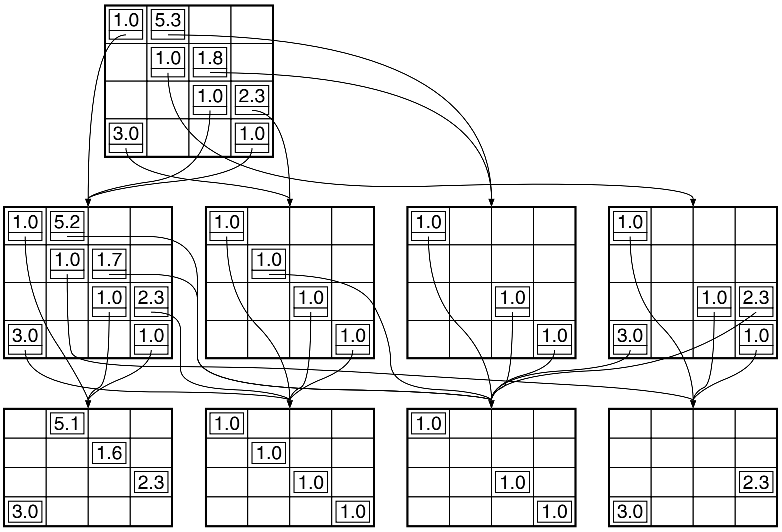


Fig. 16. Matrix diagram for the rate matrix of the running example

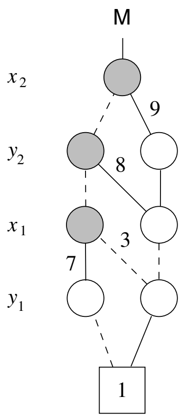
the DNBDD for a CTMC is identical in structure to the BDD which represents its transition relation, as might be used for example during reachability computation.

Figure 17 shows a simple example of a 4×4 matrix and the DNBDD which represents it. As for our MTBDD representation of a matrix of this size in Figure 10, the DNBDD uses two (interleaved) pairs of Boolean variables: x_2, x_1 to encode row indices and y_2, y_1 to encode column indices. The path through the DNBDD M which corresponds to each entry of matrix $\mathbf{M}$ is shown in the table in Figure 17. The nodes of the DNBDD which are shaded are *decision nodes*. Each node which has both *then* and *else* edges eventually leading to the 1 terminal node is a decision node. The value corresponding to a given path through the DNBDD is equal to the value which labels the edge from the last decision node along that path. For the matrix entry $(2, 3)$, for example, the row index is encoded as $1, 0$ and the column index as $1, 1$. Tracing the path $1, 1, 0, 1$ through the DNBDD, we see that the last labeled edge observed is 9, which is the value of the entry. In cases, where a single path corresponds to more than one matrix entry, the distinct values are differentiated by labeling the edge with an ordered list of values, rather than a single value.

In [45], DNBDDs were used to implement bisimulation algorithms for CTMCs. They proved to be well suited to this application, since it can be seen as an extension of the non-probabilistic case, for which BDD-based algorithms had already been developed. It has not been shown, though, how DNBDDs could be used to perform analysis of CTMCs requiring numerical computation, as we consider in the next section.

Probabilistic decision graphs (PDGs) were proposed by Bozga and Maler in [11]. A PDG is a BDD-like data structure, designed specifically for storing vec-

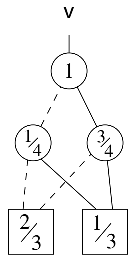
$$\mathbf{M} = \begin{pmatrix} 0 & 3 & 0 & 8 \\ 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$



Entry in $\mathbf{M}$	Path in M			
	x_2	y_2	x_1	y_1
$(0,1)=3$	0	0	0	1
$(1,0)=7$	0	0	1	0
$(0,3)=8$	0	1	0	1
$(2,3)=9$	1	1	0	1

Fig. 17. A matrix and a DNBDD representing it

$$\underline{v} = \left[\, \tfrac{1}{6} \;\; \tfrac{1}{12} \;\; \tfrac{1}{2} \;\; \tfrac{1}{4} \,\right]$$



$$\underline{v}(0) = 1 \cdot \tfrac{1}{4} \cdot \tfrac{2}{3} = \tfrac{1}{6}$$

$$\underline{v}(1) = 1 \cdot \tfrac{1}{4} \cdot \tfrac{1}{3} = \tfrac{1}{12}$$

$$\underline{v}(2) = 1 \cdot \tfrac{3}{4} \cdot \tfrac{2}{3} = \tfrac{1}{2}$$

$$\underline{v}(3) = 1 \cdot \tfrac{3}{4} \cdot \tfrac{1}{3} = \tfrac{1}{4}$$

Fig. 18. A vector and the PDG representing it

tors and matrices of probabilities. Figure 18 shows a simple example of a PDG with two levels of nodes representing a vector of length 4. Nodes are labeled with probabilities and the values for all the nodes on each level must sum to one. Intuitively, these represent conditional probabilities: the actual value of each vector element can be determined by multiplying the conditional probabilities along the corresponding path, as illustrated in the figure. Like in a BDD, duplicate nodes are merged to save space. The hope is that some functions which have no compact representation as an MTBDD, perhaps because of an excessive number of distinct values, can be stored more efficiently as a PDG.

Buchholz and Kemper extended this work in [17]. They modified the PDG data structure to allow more than two children in each node, like in an MDD. They then used PDGs to store vectors and Kronecker representations to store matrices, combining the two to perform numerical solution of CTMCs. It was found, though, that the additional work required to manipulate the PDG data structure slowed the process of numerical computation.

4 Numerical Solution

In the preceding two sections, we have seen how symbolic data structures can be used to construct and store probabilistic models and their state spaces. In this section, we discuss how to actually perform analysis of the models when they are represented in this way. In the probabilistic setting, the most commonly required types of analysis will be those that require some form of numerical solution to be performed. Hence, this is the problem which we shall focus upon.

The exact nature of the work that needs to be performed will depend on several factors, for example, the type of probabilistic model being studied and the high-level formalisms which are used to specify both the model and the properties of the model which are to be analyzed. In *probabilistic model checking*, for example, properties are typically expressed in probabilistic extensions of temporal logics, such as PCTL [41] and CSL [4, 9]. This allows concise specifications such as "the probability that the system fails within 60s is at most 0.01" to be expressed. Model checking algorithms to verify whether or not such specifications are satisfied can be found in the literature (for surveys, see for example [26, 29]). In more traditional *performance analysis* based approaches, one might be interested in computing properties such as throughput, the average number of jobs in a queue, or average time until failure. In this presentation, we will avoid discussing the finer details of these various approaches. Instead, we will focus on some of the lower level numerical computations that are often required, regardless of the approach or formalism used. One of the most common tasks likely to be performed is the computation of a vector of probabilities, where one value corresponds to each state of the model. If the model is a continuous-time Markov chain (CTMC), this vector might, for example, contain *transient* or *steady-state* probabilities, which describe the model at a particular time instant or in the long-run, respectively. For the latter, the main operation required is the solution of a linear system of equations. For the former, a technique called uniformization provides an efficient and numerically stable approach. Another common case is where the values to be computed are the probabilities of, from each state in the model, reaching a particular class of states. If the model is a discrete-time Markov chain (DTMC), the principal operation is again solution of a linear system of equations; if the model is a Markov decision process (MDP), it is the solution of a linear optimization problem.

An important factor that all these types of computation have in common is that they can be (and often are) performed *iteratively*. Consider, for example, the problem of solving a linear equation system. This is, of course, a well-studied problem for which a range of techniques exist. For analysis of very large models, however, which is our aim, common *direct* methods such as Gaussian elimination are impractical because they do not scale up well. Fortunately, iterative techniques, such as the Power, Jacobi and Gauss-Seidel methods, provide efficient alternatives. There is a similar situation for the linear optimization problems required for analysis of MDPs. One option would be to use classic linear programming techniques such as the Simplex algorithm. Again, though, these are poorly suited to the large problems which frequently occur in these applications.

Fortunately, in this instance, there exist alternative, iterative algorithms to solve the problem. Lastly, we note that the aforementioned uniformization method for transient analysis of CTMCs is also an iterative method.

There is an important feature common to the set of iterative solution techniques listed in the previous paragraph. The computation performed depends heavily, of course, on the probabilistic model: a real matrix for DTMCs and CTMCs or a non-square matrix for MDPs. However, in all of these techniques, with the possible exception of some initialization steps, the model or matrix does not need to be modified during computation. Each iteration involves extracting the matrix entries, performing some computation and updating the solution vector. This is in contrast to some of the alternatives such as Gaussian elimination or the Simplex method which are based entirely on modifications to the matrix.

In the remainder of this section, we consider the issue of performing such numerical solution techniques, when the model is being represented either as an MTBDD or as a matrix diagram.

4.1 MTBDDs

Numerical solution using MTBDDs was considered in some of the earliest papers describing the data structure: [27, 5] gave algorithms for L/U decomposition and Gaussian elimination. The experimental results presented in the latter paper illustrated that, in comparison to more conventional, explicit approaches, MTBDDs were relatively poor for these tasks. They also identified the reasons for this. Methods such as Gaussian elimination are based on access to and modification of individual elements, rows or columns of matrices. MTBDDs, though are an inherently recursive data structure, poorly suited to such operations. Moreover, compact MTBDD representations rely on high-level structure in the matrix. This is typically destroyed by many such operations, increasing MTBDD sizes and time requirements.

Subsequently, Hachtel at al. [39, 40] and Xie and Beerel [70] presented MTBDD-based algorithms for iterative solution of linear equation systems, finding them better suited to symbolic implementation than direct methods. We have already observed that such techniques are preferable in our circumstances anyway because of the size of problems involved.

The basic idea of an iterative method is that a vector containing an estimate to the solution is repeatedly updated. This update is based on operations which use the matrix. Often, the main operation required is matrix-vector multiplication. For example, when solving the linear equation system $\mathbf{A} \cdot \underline{x} = \underline{b}$ using the Jacobi method, the kth iteration, which computes the solution vector $\underline{x}^{(k)}$ from the previous approximation $\underline{x}^{(k-1)}$ is as follows:

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

where $\mathbf{D}$, $\mathbf{L}$ and $\mathbf{U}$ are diagonal, lower- and upper-triangular matrices, respectively, such that $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$. As we saw earlier, matrix multiplication is well suited to MTBDDs because it can be implemented in a recursive fashion. The other operations required here, i.e. matrix addition, vector addition and inversion of a diagonal matrix can all be implemented using the Apply operator.

Hachtel et al. applied their implementation to the problem of computing steady-state probabilities for DTMCs derived from a range of large, benchmark circuits. The largest of these DTMCs handled had more than 10^{27} states. MTBDDs proved invaluable in that an explicit representation of a matrix this size would be impossible on the same hardware. The MTBDDs were able to exploit a significant amount of structure and regularity in the DTMCs. However, it was observed that the MTBDD representation of the solution vector was more problematic. Hachtel et al. were forced to adopt techniques such as rounding all values below a certain threshold to zero.

Following the discovery that MTBDDs could be applied to iterative numerical solution, and inspired by the success of BDD-based model checking implementations, numerous researchers proposed symbolic approaches to the verification and analysis of probabilistic models. These included PCTL model checking of DTMCs [8, 42, 7, 6], PCTL model checking of MDPs [6, 53, 34, 31], computation of steady-state probabilities for CTMCs [44] and CSL model checking for CTMCs [9, 48]. While some of these papers simply presented algorithms, others implemented the techniques and presented experimental results. Their conclusions can be summarized as follows.

Firstly, MTBDDs can be used, as mentioned earlier, to construct and store extremely large, probabilistic models, where high-level structure can be exploited. Furthermore, in some instances, MTBDD-based numerical solution is also very successful. For example, [54] presented results for symbolic model checking of MDPs with more than 10^{10} states. This would be impossible with explicit techniques, such as sparse matrices, on the same hardware. However, in general, it was found that MTBDD-based numerical computation performed poorly in comparison to explicit alternatives, in terms of both time and space efficiency. The simple reason for this is that, despite compact MTBDD-based storage for probabilistic models, the same representation for solution vectors is usually inefficient. This is unsurprising since these vectors will usually be unstructured and contain many distinct values. Both of these factors generally result in greatly increased MTBDD sizes which are not only more expensive to store, but are slower to manipulate. By contrast, in an explicit (e.g. sparse matrix based) implementation, solution vectors are typically stored in arrays which are fast to access and manipulate, regardless of structure.

4.2 Offset-Labeled MTBDDs

To combat the problems described above, modifications to the MTBDD-based approach have been proposed [52, 62]. The basic idea is to combine the compact model representation afforded by symbolic techniques with the fast and efficient numerical solution provided by explicit approaches. This is done by performing numerical computation using an MTBDD for matrix storage but an array for the solution vector.

As above, the most important operation to be implemented is matrix-vector multiplication. A crucial observation is that this requires access to each of the non-zero entries in the matrix exactly once and that the order in which they are obtained is unimportant. The non-zero entries of a matrix stored by an MTBDD can be extracted via a recursive depth-first traversal of the data structure. Essentially, this traces every path from the root node of the MTBDD to a terminal node, each path corresponding to a single non-zero matrix entry.

During this process, we need to be able to compute the row and column index of each matrix entry. This is done by augmenting the MTBDD with additional information: we convert the MTBDD into an *offset-labeled MTBDD*, each node of which is assigned an integer offset. This is essentially the same as the indexing scheme we described for BDDs and MDDs in Section 2.4. Like in the simpler case, when tracing a path through an MTBDD, the required indices can be computed by summing the values of the offsets. More specifically, the row and column index are computed independently: the row index by summing offsets on nodes labeled with x_k variables from which the *then* edge was taken; and the column index by summing those on nodes labeled with y_k variables where the *then* edge was taken.

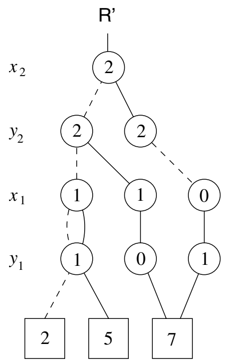
Figure 19 shows an example of an offset-labeled MTBDD R', the transition rate matrix $\mathbf{R}$ it represents, and a table explaining the way in which the information is represented. The matrix $\mathbf{R}$ is the same as used for the earlier example in Figure 10. State 2 of the CTMC which the matrix represents (see Figure 10) is unreachable. All entries of the corresponding row and column of $\mathbf{R}$ are zero. In Figure 19, this is emphasized by marking these entries as '–'.

Let us consider a path through the offset-labeled MTBDD: 1, 0, 1, 1. This is the same as used for the MTBDD example on page 313. The path leads to the 7 terminal, revealing that the corresponding matrix entry has value 7. To compute the row index of the entry, we sum the offsets on x_k nodes from which the *then* edge was taken. For this path, the sum is $2 + 0 = 2$. For the column index, we perform the same calculation but for y_k nodes. The *then* edge was only taken from the y_1 node, so the index is 1, the offset on this node. Hence, our matrix entry is $(2, 1) = 7$. Notice that the corresponding matrix entry last time was $(3, 1) = 7$, i.e. the offsets encode the fact that the unreachable state is not included in the indexing.

Compare the MTBDD R from Figure 10 with the offset-labeled variant R' in Figure 19, both of which represent the same matrix. Firstly, R' has been converted to its quasi-reduced form so that offsets can be added on each level (as was the case for BDDs in Section 2.4). Secondly, note that two additional nodes have been added (rightmost x_1 and y_1 nodes). These nodes would be duplicates (and hence removed) if not for the presence of offsets. This situation occurs when there are two paths through the MTBDD passing through a common node and the offset required to label that node is different for each path. Empirical results show that, in practice, this represents only a small increase in MTBDD size. For further details in this area, see [62].

Each row of the table in Figure 19 corresponds to a single path through R' which equates to a single non-zero entry of $\mathbf{R}$. The entries are listed in the order

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



Path					Offsets				Entry of $\mathbf{R}$
x_2	y_2	x_1	y_1	$f_{\mathsf{R}'}$	x_2	y_2	x_1	y_1	
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. 19. An offset-labeled MTBDD R' representing a matrix $\mathbf{R}$

```
TraverseRec(m, $row$, $col$)
  • m is an MTBDD node
  • $row$ is the current row index
  • $col$ is the current column index
 1: if m is a non-zero terminal node then                  • Terminal case
 2:    Found matrix element $(row, col) = val(\mathsf{m})$
 3: else if m is the zero terminal node then               • Terminal case
 4:    Return
 5: else if m is an $x_k$ variable then                    • Recurse
 6:    TraverseRec($else$(m), $row$, $col$)
 7:    TraverseRec($then$(m), $row + offset(\mathsf{m})$, $col$)
 8: else if m is an $y_k$ variable then                    • Recurse
 9:    TraverseRec($else$(m), $row$, $col$)
10:    TraverseRec($then$(m), $row$, $col + offset(\mathsf{m})$)
11: end if
```

Fig. 20. Offset-labeled MTBDD traversal algorithm

in which they would be extracted by a recursive traversal of the offset-labeled MTBDD. Figure 20 shows the recursive algorithm, TraverseRec, used to perform such a traversal. Its first argument is the current MTBDD node m. Initially, this is the root node of the MTBDD being traversed. The other two arguments, *row* and *col*, keep track of the row and column indices respectively. Initially, these are both zero. Line 2 is where each matrix entry is found. In practice, this would actually use the matrix entry to perform a matrix-vector multiplication. We denote by *offset*(m) the offset labeling the node m. Note that, in essence, this algorithm is very similar to Enumerate in Figure 7(a).

Empirical results for the application of offset-labeled MTBDDs to several numerical solution problems can be found in [52, 62]. In practice, a number of optimizations are applied. For example, since TraverseRec is typically performed many times (once for every iteration of numerical solution), it is beneficial to cache some of the results it produces and reuse them each time. See [52, 62] for the details of these techniques. With these optimizations in place, that the speed of numerical solution using offset-labeled MTBDDs almost matches that of explicit methods based on sparse matrices. More importantly, because of the significant savings in memory made by storing the matrix symbolically, offset-labeled MTBDDs can usually handle much larger problems. An increase of an order of magnitude is typical.

4.3 Matrix Diagrams

In [24, 57], algorithms were given to compute the steady-state probabilities of a CTMC using the Gauss-Seidel iterative method, when the CTMC is represented with matrix diagrams and the steady-state probabilities are stored explicitly. To perform an iteration of Gauss-Seidel, we must be able to efficiently enumerate the columns of the transition rate matrix $\mathbf{R}$. This can be done using matrix diagrams, provided the matrix for each node is stored using a sparse format that allows for efficient column access [24, 57]. Column construction proceeds from the bottom level nodes of the matrix diagram to the top node. A recursive algorithm to perform the computation is shown in Figure 21. The algorithm parameters include the current "level", k, a level-k matrix diagram node M which encodes a matrix $\mathbf{M}$, a QROMDD node R with offsets which encodes the desired set of rows $\mathcal{R}$ (i.e., an EV$^+$MDD encoding the lexicographical indexing function ψ for $\mathcal{R}$), and the specification of the desired column $(y_k, \ldots, y_1)$. Using an ROMDD with offsets is also possible for $\mathcal{R}$, but leads to a slightly more complex algorithm to handle removed redundant nodes. Algorithm GetColumn returns the desired column of matrix $\mathbf{M}$ over the desired rows only, which is a vector of dimension $|\mathcal{R}|$. The offset information of R is used to ensure that an element at row $(x_K, \ldots, x_1)$ in the desired column is returned in position $\psi(x_K, \ldots, x_1)$ of the column vector. The algorithm assumes that the undesired rows have been removed from the matrix already using the Submatrix operator.

Since probabilistic systems tend to produce sparse matrices, the algorithm assumes that the desired column will also be sparse, i.e. that the number of non-zero elements it contains will be much less than $|\mathcal{R}|$. Hence, a sparse structure is used for the columns. Since the maximum number of non-zeros present in any column can be determined in a preprocessing step, we can bound the size of the sparse vector that is necessary for any call to GetColumn, and use an efficient array-based structure to represent the sparse columns. This allows for efficient

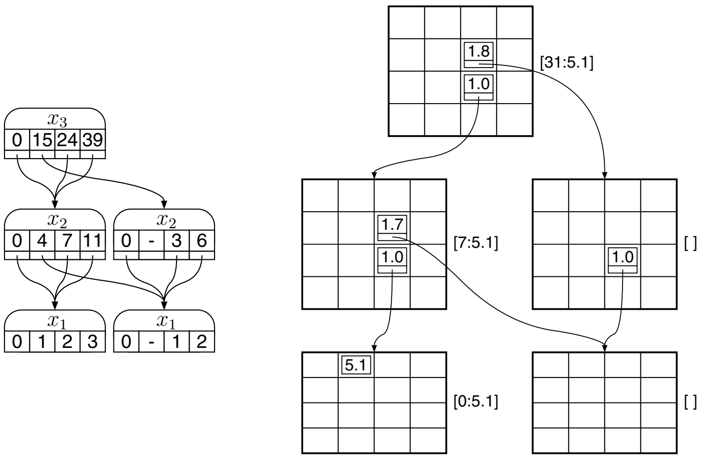
```
GetColumn($k, \mathsf{M}, \mathsf{R}, y_k, \ldots, y_1$)
 • R is an MDD with offsets, encoding the set of rows $\mathcal{R}$ to consider.
 • M is a matrix diagram, where rows $\notin \mathcal{R}$ of the encoded matrix are zero.
 • $(y_k, \ldots, y_1)$ is the desired column to obtain.
 1: if $k = 0$ then                                                    • Terminal case
 2:    Return $[1]$;
 3: else if $\exists$ computed-table entry $(k, \mathsf{M}, \mathsf{R}, y_k, \ldots, y_1, \underline{col})$ then
 4:    Return $\underline{col}$;
 5: end if
 6: $\underline{col} \leftarrow \mathbf{0}$;                            • $\underline{col}$ is a sparse vector
 7: for each $x_k : pairs(\mathsf{M}, x_k, y_k) \neq \emptyset$ do
 8:    for each $(r, \mathsf{M}') \in pairs(\mathsf{M}, x_k, y_k)$ do
 9:       $\underline{d} \leftarrow$ GetColumn$(k-1, \mathsf{M}', child(\mathsf{R}, x_k), y_{k-1}, \ldots, y_1)$;
10:       for each $i : \underline{d}[i] \neq 0$ do
11:          $\underline{col}[i + value(\mathsf{R}, x_k)] \leftarrow \underline{col}[i + value(\mathsf{R}, x_k)] + r \cdot \underline{d}[i]$;
12:       end for
13:    end for
14: end for
15: Add entry $(k, \mathsf{M}, \mathsf{R}, y_k, \ldots, y_1, \underline{col})$ to computed-table;
16: Return $\underline{col}$;
```

Fig. 21. Recursive algorithm to select a matrix diagram column

implementation of the loop in lines 10 through 12. Note that the algorithm works correctly for dense matrices, but may not be the most efficient approach.

As with most recursive algorithms for decision diagrams, it is possible (and in fact likely) that a single top-level call to GetColumn will produce several *identical* recursive calls to GetColumn. Thus, GetColumn utilizes a *computed-table*, like many of the algorithms for decision diagram operators. The computed-table is also useful when performing a sequence of column operations, such as during numerical solution, which requires obtaining each reachable column in turn during a single iteration. However, results cannot be stored in the computed-table indefinitely; doing so would require storage space comparable to explicit sparse storage of the entire transition rate matrix once all columns have been computed. The solution to this problem adopted in [24, 57] is to save only the most recent column result for each node. Note that this allows simplification to lines 3 and 15 in Figure 21, since each matrix diagram node will have at most one computed-table entry. Other benefits of this approach are that memory usage is not only relatively small, but fixed: by traversing the matrix diagram once in a preprocessing step, we can determine the largest column that will be produced by each node, and allocate a fixed-size computed-table of the appropriate size. Also, the algorithm GetColumn can be implemented to use the computed-table space directly, thus avoiding the need to copy vectors into the table.

As an example, the computed-table entries after obtaining column $(2, 2, 1)$ for the running example are shown in Figure 22(b). Note that the matrix diagram



(a) MDD plus offsets for $\mathcal{S}'$ (b) Entries accessed by GetColumn

Fig. 22. Obtaining column $(2, 2, 1)$ for the running example

shown corresponds to the relevant portion of the matrix diagram of Figure 16. The reachable rows are encoded using the MDD with offset values shown in Figure 22(a). Note that column 1 of the rightmost level-1 matrix diagram node is empty; thus, the resulting column has no non-zero entries. The only transition to state $(2, 2, 1)$ is from state $(2, 2, 0)$, with rate $5.1 \cdot 1.0 \cdot 1.0 = 5.1$. Since state $(2, 2, 0)$ has index $24 + 7 + 0 = 31$ according to the MDD, the resulting column has a single non-zero entry at position 31, with value 5.1.

The computed-table must be invalidated at level k whenever the specified column changes for any component less or equal to k, since columns are constructed in a bottom-up fashion. For instance, if we just computed the column $(y_K, \ldots, y_1)$ and we want to compute the column $(y_K, \ldots, y_{k+1}, y'_k, y_{k-1}, \ldots, y_1)$ then the computed-table columns at levels K through k must be cleared. Thus, to improve our chances of hits in the computed-table (by reducing the number of times we must clear columns), after visiting the reachable column for state $(y_K, \ldots, y_1)$, we should next visit the reachable column for the state that follows $(y_K, \ldots, y_1)$ in lexicographic order *reading the strings in reverse*; that is, the order obtained when component K is the least significant (fastest changing) and component 1 is the most significant (slowest changing). Note that this order is *not* reverse lexicographic order. As described in [24], this order can be quickly traversed by storing the set of columns using an "upside-down" variable order. That is, if the reachable rows are stored using an MDD R with variable ordering $x_K, \ldots, x_1$, and the matrix diagram M is stored using ordering $(x_K, y_K), \ldots, (x_1, y_1)$, then the reachable columns should be stored in an MDD C with variable ordering $y_1, \ldots, y_K$. Enumerating the reachable columns of C as described in Section 2.4 will treat component K as the fastest changing component.

Numerical solution can then proceed, using explicit storage for solution vectors (with dimension $|\mathcal{R}|$), MDDs to represent the reachable states for the rows and columns, and a matrix diagram to represent the transition rate matrix $\mathbf{R}$. Often, the diagonal elements of $\mathbf{Q}$ must be easily accessible during numerical solution; these can be stored either explicitly in a vector (at a cost of increased storage requirements), or implicitly using a second matrix diagram (at a cost of increased computational requirements) [24].

Using matrix diagrams allows for the numerical solution of large models [24, 57]. Storage requirements for the required MDDs and for the matrix diagram itself are often negligible compared to the requirements for the solution vector; essentially, all available memory can be used for the solution vector. In practice, using matrix diagrams instead of explicit sparse matrix storage allows for solution of models approximately one order of magnitude larger.

5 Discussion and Conclusion

In this paper, we have surveyed a range of symbolic approaches for the generation, representation and analysis of probabilistic models. We conclude by summarizing some of their relative strengths and weaknesses and highlighting areas for future work.

In Section 2, we presented techniques for the representation and manipulation of state sets using BDDs and MDDs. These are at the heart of non-probabilistic symbolic model checking approaches and have proved to be extremely successful. They have also been examined at great length in the literature. Of more interest here are the extensions of these techniques to handle problems which are specific to the validation of probabilistic models.

In Sections 3 and 4, we focused on methods to construct, store and manipulate probabilistic models. We have seen that symbolic data structures such as MTBDDs and matrix diagrams can be successfully applied to this problem, allowing large, structured models to be constructed quickly and stored compactly. Furthermore, we have seen that, with suitable adaptations and algorithms, these data structures can be used to perform efficient numerical computation, as required for analysis of the probabilistic models. It is important to note that such results are reliant on high-level structure and regularity in these models. Furthermore, heuristics for state space encoding and variable ordering may need to be employed, especially where a binary encoding must be chosen. Implementations of both techniques are available in the tools PRISM [51, 1] and SMART [22, 2], respectively. These tools have been applied to a wide range of case studies. Overall, the conclusion is that we can increase by approximately an order of magnitude the size of problems which can be handled.

While the symbolic techniques discussed here have been successful in this respect, there remain important challenges for future work. One interesting area would be to investigate more closely the similarities and differences between the various data structures. This paper was broken into two distinct threads, describing the use of first BDDs and MTBDDs and then MDDs and matrix diagrams. It is evident that there are many similarities between the two approaches. There are also important differences though, leaving room for experimentation and exchanges of ideas.

The two principal issues are as follows. Firstly, the variables in BDDs and MTBDDs are Boolean, whereas in MDDs and matrix diagrams, they are multi-valued. For BDDs and MTBDDs, this simplifies both the storage of the data structure and the operations which are performed on it. However, it is not clear whether forcing everything to be encoded as Boolean variables is a good idea. It may be more intuitive or even more efficient to treat models as a small number of fixed-size submodels, as is the case for MDDs and matrix diagrams. Furthermore, larger variables will generally imply fewer levels to encode a given model, possibly implying less overhead during manipulation.

The second key difference is that matrix diagrams are implicitly tied to the Kronecker representation, where large matrices are stored as Kronecker-algebraic expressions of smaller matrices. This may well produce a more compact representation than MTBDDs, particularly where it results in many distinct values being stored. However, this may lead to more overhead in terms of actually computing each matrix entry than is necessary for MTBDDs, since floating-point multiplications are required.

Given these points, it is tempting to compare the two approaches experimentally. However, we feel that including experimental results here is inappropriate for the following reasons. First, since we are unaware of any single software tool that incorporates both approaches, it would be necessary to use different tools for each technique, introducing numerous uncertainties. For example, observed differences in performance could be due to the implementation quality of the respective tools, or to the fact that one tool handles a more general (but more computationally expensive) class of models. Second, without a thorough study of the differences between the two approaches, it would be unclear whether experimental differences apply in general, for a particular type of model, or simply for the models used in the experiments. Finally, results presented here would essentially duplicate those already published elsewhere. Clearly, further work is necessary to study the differences between the approaches in detail, in order to understand how the techniques can best be used in practice (to date, we are unaware of *any* work in this area).

There is one issue which unites the two schemes. We have seen that both usually allow an increase of approximately an order of magnitude in model size over explicit alternatives. While this is useful, improvement beyond this range is difficult. This is because, despite compact storage of the state space using BDDs or MDDs and of the model itself using MTBDDs or matrix diagrams, solution vectors proportional to the size of the model's state space must still be stored explicitly to allow acceptable speed of solution. This generally represents the limiting factor in terms of memory requirements. Hence, MTBDDs and matrix diagrams can be used to construct and store much larger models than can at present feasibly be solved. Attempts to store vectors symbolically using either

MTBDDs or PDGs have met with only limited success. It appears that the efficient representation for these cases is a much more difficult problem. In fact, it may be asked whether there is even structure there to exploit. Further research is needed to clarify this issue.

One possibility for alleviating the above difficulties, though, is by using parallel, distributed or out-of-core approaches, as in e.g. [50]. Also, it is also worth mentioning that these problems apply only to the computation of an *exact* solution. The ability to represent and manipulate a large model exactly using symbolic techniques may lead to new and intriguing *approximation* algorithms. Indeed, initial efforts in this area have been promising [61]. Much work is needed in this new area.

References

1. PRISM web site. www.cs.bham.ac.uk/˜dxp/prism.
2. SMART web site. http://www.cs.wm.edu/˜ciardo/SMART/.
3. S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
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 and E. Clarke. The algebraic mu-calculus and MTBDDs. In *Proc. 5th Workshop on Logic, Language, Information and Computation (WOLLIC'98)*, pages 27–38, 1998.
8. 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.
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. B. Bollig and I. Wegner. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1006, 1996.
11. 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.

12. K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automation Conference (DAC'90)*, pages 40–45. IEEE Computer Society Press, 1990.
13. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
14. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
15. P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, SUMMER 2000.
16. P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, October 1995. IEEE Comp. Soc. Press.
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. P. Buchholz and P. Kemper. Kronecker based matrix representations for large Markov models. This proceedings, 2003.
19. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990.
20. G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000 (Proc. 21st Int. Conf. on Applications and Theory of Petri Nets)*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.
21. G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.
22. 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.
23. G. Ciardo and A. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *LNCS*, pages 44–57. Springer, 1997.
24. 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.
25. G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In Mark D. Aagaard and John W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, November 2002. Springer.

26. F. Ciesinski and F. Grössner. On probabilistic computation tree logic. This proceedings, 2003.
27. 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.
28. 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.
29. L. Cloth. Specification and verification of Markov reward models. This proceedings, 2003.
30. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proc. International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373. Springer, 1989.
31. 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. 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.
32. I. Davies, W. Knottenbelt, and P. Kritzinger. Symbolic methods for the state space exploration of GSPN models. 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 188–199. Springer, 2002.
33. M. Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, C-30(2):116–125, February 1981.
34. 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.
35. S. Donatelli. Superposed stochastic automata: A class of stochastic Petri nets amenable to parallel solution. *Performance Evaluation*, 18:21–36, 1993.
36. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In K. Larsen and A. Skou, editors, *Proc. 3rd International Workshop on Computer Aided Verification (CAV'91)*, volume 575 of *LNCS*, pages 203–213. Springer, 1991.
37. P. Fernandes, B. Plateau, and W. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.
38. M. Fujita, Y. Matsunaga, and T. Kakuda. On the variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European conference on Design automation*, pages 50–54, 1991.
39. 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.
40. G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Trans. on CAD*, 15(12):1479–1493, 1996.
41. H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

42. V. Hartonas-Garmhausen. *Probabilistic Symbolic Model Checking with Engineering Models and Applications*. PhD thesis, Carnegie Mellon University, 1998.
43. 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.
44. 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.
45. H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In J.-P. Katoen, editor, *Proc. 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 244–264. Springer, 1999.
46. J. Hillston and L. Kloul. An efficient Kronecker representation for PEPA models. In *Proc. Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01)*, pages 120–135. Springer-Verlag, September 2001.
47. T. Kam, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
48. 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. 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.
49. 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.
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, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.
53. 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.

54. 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.
55. C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.
56. K. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
57. A. Miner. *Data Structures for the Analysis of Large Structured Markov Chains.* PhD thesis, Department of Computer Science, College of William & Mary, Virginia, 2000.
58. A. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In R. German and B. Haverkort, editors, *Proc. 9th International Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110. IEEE Computer Society Press, 2001.
59. A. Miner. Efficient state space generation of GSPNs using decision diagrams. In *Proc. 2002 Int. Conf. on Dependable Systems and Networks (DSN 2002)*, pages 637–646, Washington, DC, June 2002.
60. A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In S. Donatelli and J. Kleijn, editors, *Proc. 20th International Conference on Application and Theory of Petri Nets (ICATPN'99)*, volume 1639 of *LNCS*, pages 6–25. Springer, 1999.
61. A. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. 2000 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 207–216, Santa Clara, CA, June 2000.
62. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.
63. E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Design Automation and Test in Europe (DATE'98)*, Paris, February 1998.
64. E. Pastor, J. Cortadella, and M. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *20th International Conference on Application and Theory of Petri Nets*, Williamsburg, June 1999.
65. 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.
66. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conference on Computer-Aided Design (ICCAD'93)*, pages 42–47, 1993.
67. M. Siegle. Compositional representation and reduction of stochastic labelled transition systems based on decision node BDDs. In D. Baum, N. Mueller, and R. Roedler, editors, *Proc. Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen (MMB'99)*, pages 173–185. VDE Verlag, 1999.
68. M. Siegle. Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties. Habilitation thesis, Universität Erlangen-Nürnberg, 2002.
69. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton, 1994.

70. 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.
71. D. Zampunièris. *The Sharing Tree Data Structure, Theory and Applications in Formal Verification.* PhD thesis, Department of Computer Science, University of Namur, Belgium, 1997.