

Data Representation and Efficient Solution: A Decision Diagram Approach^{*}

Gianfranco Ciardo

Dept. of Computer Science and Engineering, UC Riverside, CA 92521, USA
ciardo@cs.ucr.edu

Abstract. Decision diagrams are a family of data structures that can compactly encode many functions on discrete structured domains, that is, domains that are the cross-product of finite sets. We present some important classes of decision diagrams and show how they can be effectively employed to derive “symbolic” algorithms for the analysis of large discrete-state models. In particular, we discuss both explicit and symbolic algorithms for state-space generation, CTL model-checking, and continuous-time Markov chain solution. We conclude with some suggestions for future research directions.

Key words: binary decision diagrams, multi-valued decision diagrams, edge-valued decision diagrams, state-space generation, symbolic model checking, Kronecker algebra.

1 Introduction

Discrete-state models are useful to describe and analyze computer-based and communication systems, or distributed software, and many other man-made artifacts. Due to the inherent combinatorial nature of their interleaving behavior, such models can easily have enormous state spaces, which can make their computer-supported analysis very difficult. Of course, high-level formalisms such as Petri nets, queueing networks, communicating sequential processes, and specialized languages can be effectively used to describe these enormous underlying state space. However, when a model described in one of these formalisms needs to be analyzed, the size of the state space remains a major obstacle.

In this contribution, we present *decision diagrams*, a class of data structures that can compactly encode functions (or set and relations, or vectors and matrices) on very large but structured domains. Then, we briefly summarize some of the main computational tasks involved in the traditional “explicit” analysis of systems, both in a strictly logical setting and in a Markovian setting. Putting the two together, we then show how these tasks can be effectively performed “symbolically” using decision diagrams. We conclude by listing some research challenges lying ahead.

^{*} Work supported in part by the National Science Foundation under grants CNS-0501747 and ATM-0428880

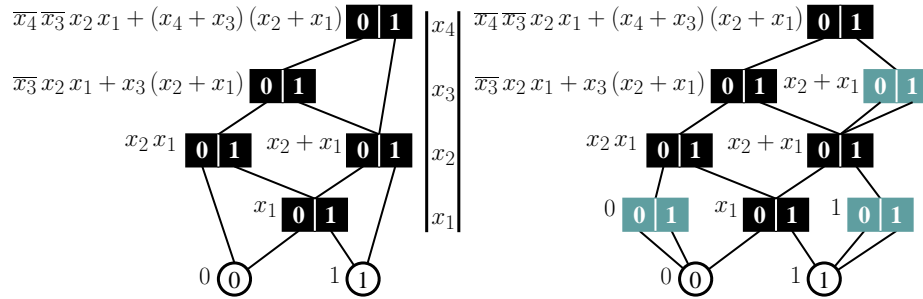


Fig. 1. Fully-reduced vs. quasi-reduced BDDs.

In the following, calligraphic letters (e.g., \mathcal{A} , \mathcal{X}) denote sets or relations, lowercase bold letters (e.g., \mathbf{i} , \mathbf{j}) indicate global state of the system, and lowercase italic letters (e.g., i , x_k) indicate local components of the state. Also, \mathbb{B} , \mathbb{N} , \mathbb{Z} , and \mathbb{R} indicate the set $\{0, 1\}$ of boolean values, the natural numbers, the integers, and the real numbers, respectively.

2 Decision diagrams: a data structure for structured data

Many classes of decision diagrams have been defined in the literature. This section presents some of the most common ones, which will be used in Section 4 to improve the efficiency of discrete-state system analysis.

2.1 Binary decision diagrams

The most widely known and used class of decision diagrams is by far the **binary decision diagrams (BDDs)** [3], which provide a compact representation for functions of the form $f : \mathbb{B}^L \rightarrow \mathbb{B}$, for some finite $L \in \mathbb{N}$. In particular, if the BDDs are *reduced* and *ordered* [3, 4], properties we assume from now on for all classes of decision diagrams we discuss, they enjoy several important advantages. Such BDDs are *canonical*, thus testing for satisfiability, i.e.,

“is there an $\mathbf{i} \in \mathbb{B}^L$ such that $f(\mathbf{i}) = 1$?”

or equivalence, i.e.,

“given functions f and g , is $f(\mathbf{i}) = g(\mathbf{i})$ for all $\mathbf{i} \in \mathbb{B}^L$?”

can be done in constant time, while important binary operations such as conjunction, disjunction, and relational product (described in detail later) require time and memory proportional to the product of the size (number of nodes) of the argument BDDs, in the worst case.

Formally, an L -variable BDD is an acyclic directed edge-labeled graph where each of its nodes encodes a function of the form $f : \mathbb{B}^L \rightarrow \mathbb{B}$. The nodes of

the graph are assigned to a level, we let $p.lvl$ denote the level of node p . A non-terminal node at level k , with $L \geq k \geq 1$, corresponds to a choice for the value of the boolean variable x_k , the k^{th} argument to the function, while the two terminal nodes 0 and 1 correspond to the two possible values of the function. A node p at level k has two edges, labeled with the possible values of x_k , 0 and 1. The edge labeled 0, or 0-edge, points to node $p[0]$ while the 1-edge point to node $p[1]$, where both nodes are at levels below k (this is the “ordered” property: nodes are found along any path in an order consistent with the order x_L, \dots, x_1 of the argument variables). Also, nodes must be *unique* (thus, no node can be a *duplicate* of another node at the same level, i.e., have have the same 0-child and the same 1-child) and *non-redundant*, i.e., $p[0]$ and $p[1]$ must differ (this is the “fully reduced” property). A node p at level k encodes the function $v_p : \mathbb{B}^L \rightarrow \mathbb{B}$ defined recursively by

$$v_p(x_L, \dots, x_1) = \begin{cases} p & \text{if } k = 0 \\ \overline{x_k} \wedge v_{p[0]}(x_L, \dots, x_1) \vee x_k \wedge v_{p[1]}(x_L, \dots, x_1) & \text{if } k > 0. \end{cases}$$

Thus, given a constant vector $\mathbf{i} = (i_L, \dots, i_1) \in \mathbb{B}^L$, we can evaluate $v_p(\mathbf{i})$ in $O(L)$ time. Fig. 1 on the left shows an example of BDD and the boolean functions encoded by its nodes. On the right, the same set of functions are encoded using a “quasi-reduced” version of BDDs, where duplicate nodes are not allowed, but redundant nodes (shown in gray in the figure) may have to be present, since edges can only span one level. Such BDDs are still canonical, and, while they may use more nodes, all their edges connect nodes at adjacent levels, resulting in simpler manipulation algorithms.

Strictly speaking, a BDD encoding a given function f has a specific root p such that $f = v_p$. In practice, BDD algorithms need to manage multiple functions on the same domain \mathbb{B}^L , and this is done by storing (without duplicates) all the roots of these functions, as well as the nodes reached by them, in a single BDD, often referred to as a BDD *forest*.

2.2 Multi-valued, multi-terminal, and multi-dimensional extensions

Many variants of BDDs have been defined to extend their applicability or to target specific applications. This section discusses several “terminal-valued” variants, that is, decision diagrams where the value of function f evaluated on argument \mathbf{i} is given by the the terminal node reached when following the path corresponding to \mathbf{i} , just as is the case for BDDs.

Multi-valued decision diagrams (MDDs) [24] encode functions of the form $\hat{\mathcal{X}} \rightarrow \mathbb{B}$, where the domain $\hat{\mathcal{X}}$ is the cross-product $\hat{\mathcal{X}} = \mathcal{X}_L \times \dots \times \mathcal{X}_1$ of L finite sets and each \mathcal{X}_k , for $L \geq k \geq 1$, is of the form $\mathcal{X}_k = \{0, 1, \dots, n_k - 1\}$, for some $n_k \in \mathbb{N}$. Thus, a non-terminal node at level k corresponds to a multi-way choice for the argument variable x_k . The top of Fig. 2 shows, on the left, a quasi-reduced MDD where redundant nodes p with $p[0] = \dots = p[n_k - 1]$ are kept and, on the right, the same MDD in a fully-reduced version, where its “long edges”

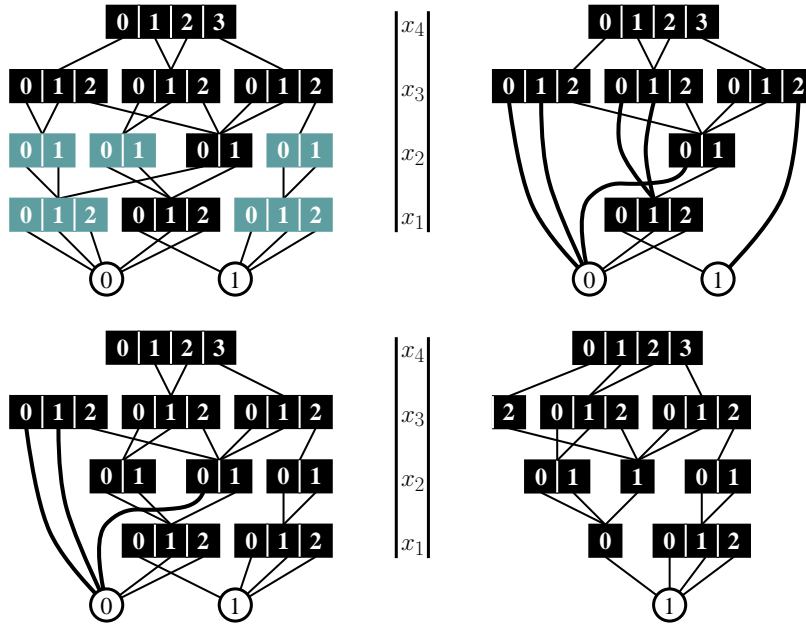


Fig. 2. Fully-reduced vs. quasi-reduced, full vs. sparse representation of MDDs.

(spanning multiple levels) are shown with thicker lines. The bottom of the same figure shows, on the left a *default-reduced* version of the same MDD where only edges to the default terminal node 0 can span multiple levels (a still canonical compromise between the fully-reduced and quasi-reduced versions) and, on the right, its *sparse* representation where paths leading to node 0 are not shown. Graphically, this last representation is quite compact, and it also reflects quite closely how MDDs are implemented in the tool SMART [9].

Multi-terminal BDDs (MTBDDs) [21] can encode functions of the form $\mathbb{B}^L \rightarrow \mathbb{R}$, by attaching arbitrary values from \mathbb{R} to the terminal nodes of a binary decision diagram. The **algebraic decision diagrams (ADDs)** [1] are exactly analogous, except were defined to encode function on arbitrary ranges, not just the reals. The **multi-terminal multi-valued decision diagrams (MT-MDDs)**, an example of which is shown in Fig. 3 on the left in fully-reduced version and on the right in quasi-reduced version, naturally extend MTBDDs by additionally allowing multi-way choices at each node.

A function $f : \hat{\mathcal{X}} \rightarrow \mathcal{S}$ can of course also be thought of as an \mathcal{S} -valued one-dimensional vector of size $|\hat{\mathcal{X}}|$. Many applications also need to encode functions of the form $\hat{\mathcal{X}} \times \hat{\mathcal{X}} \rightarrow \mathcal{S}$, or two-dimensional matrices. An obvious way to accomplish this is to use a BDD, MDD, MTBDD, or MTMDD with twice as many levels. The traditional notation uses an “unprimed” x_k for the rows, or “from”, variables, and a “primed” x'_k for columns, or “to” variables. Furthermore, the manipulation

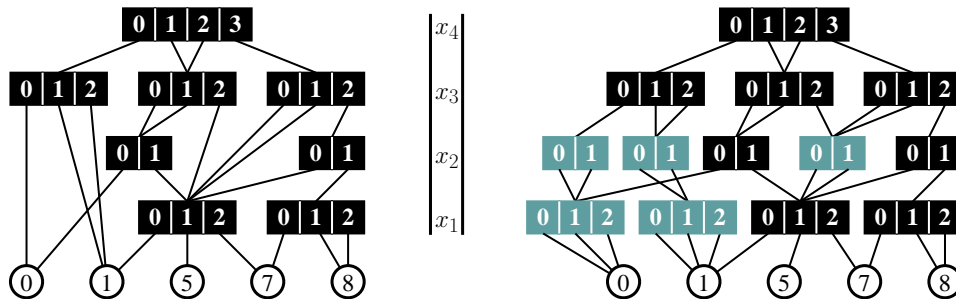


Fig. 3. Fully-reduced vs. quasi-reduced MTMDD.

algorithms are more easily written, and usually much more efficient, if the levels are *interleaved*, that is, the order of the function parameters is $(x_L, x'_L, \dots, x_1, x'_1)$. A similar, but more direct, way to encode such matrices is to use a **(boolean-valued) matrix diagrams (MxDs)** [13, 27], where a non-terminal node P at level k , for $L \geq k \geq 1$, has $n_k \times n_k$ edges, so that $P[i_k, i'_k]$ points to the node to be reached when $x_k = i_k$ and $x'_k = i'_k$. The top left of Fig. 4 shows a $2L$ -level MDD, where $L = 2$, in sparse format. The encoded $(3 \cdot 2) \times (3 \cdot 2)$ matrix has zero entries in all but seven positions (the rows and columns of the resulting matrix are indexed by $x_2 \cdot 2 + x_1$ and $x'_2 \cdot 2 + x'_1$, respectively):

<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>3</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>4</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>5</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table>		0	1	2	3	4	5	0	0	0	0	0	0	0	1	0	0	0	0	1	0	2	0	0	0	0	0	0	3	1	1	1	1	0	0	4	0	0	1	0	0	0	5	0	0	0	1	0	0	where	<table style="border-collapse: collapse;"> <tr><td>0</td><td>$\equiv (x_2 = 0, x_1 = 0)$</td></tr> <tr><td>1</td><td>$\equiv (x_2 = 0, x_1 = 1)$</td></tr> <tr><td>2</td><td>$\equiv (x_2 = 1, x_1 = 0)$</td></tr> <tr><td>3</td><td>$\equiv (x_2 = 1, x_1 = 1)$</td></tr> <tr><td>4</td><td>$\equiv (x_2 = 2, x_1 = 0)$</td></tr> <tr><td>5</td><td>$\equiv (x_2 = 2, x_1 = 1)$</td></tr> </table>	0	$\equiv (x_2 = 0, x_1 = 0)$	1	$\equiv (x_2 = 0, x_1 = 1)$	2	$\equiv (x_2 = 1, x_1 = 0)$	3	$\equiv (x_2 = 1, x_1 = 1)$	4	$\equiv (x_2 = 2, x_1 = 0)$	5	$\equiv (x_2 = 2, x_1 = 1)$
	0	1	2	3	4	5																																																									
0	0	0	0	0	0	0																																																									
1	0	0	0	0	1	0																																																									
2	0	0	0	0	0	0																																																									
3	1	1	1	1	0	0																																																									
4	0	0	1	0	0	0																																																									
5	0	0	0	1	0	0																																																									
0	$\equiv (x_2 = 0, x_1 = 0)$																																																														
1	$\equiv (x_2 = 0, x_1 = 1)$																																																														
2	$\equiv (x_2 = 1, x_1 = 0)$																																																														
3	$\equiv (x_2 = 1, x_1 = 1)$																																																														
4	$\equiv (x_2 = 2, x_1 = 0)$																																																														
5	$\equiv (x_2 = 2, x_1 = 1)$																																																														

MxDs, however, were not introduced just to stress the natural interleaving of unprimed and primed variables, but to exploit a much more fundamental property often present in large asynchronous systems: the large number of *identity patterns*. The top right of Fig. 4 shows the MxD encoding the same matrix, and the gray node in it is an example of an identity: its diagonal edges point to the same node, the terminal node 1 in this case, while its off-diagonal entries point to node 0. The bottom left of Fig. 4 shows the *identity-reduced* version of MxDs which is commonly employed, where long edges signify skipped identity nodes; on the right is the sparse format representation, which just lists explicitly the row-column pairs of indices corresponding to non-zero node entries.

2.3 Edge-valued extensions

Further “edge-valued” variants of decision diagrams have been defined to represent functions with a non-boolean range, as MTMDDs and ADDs can, but

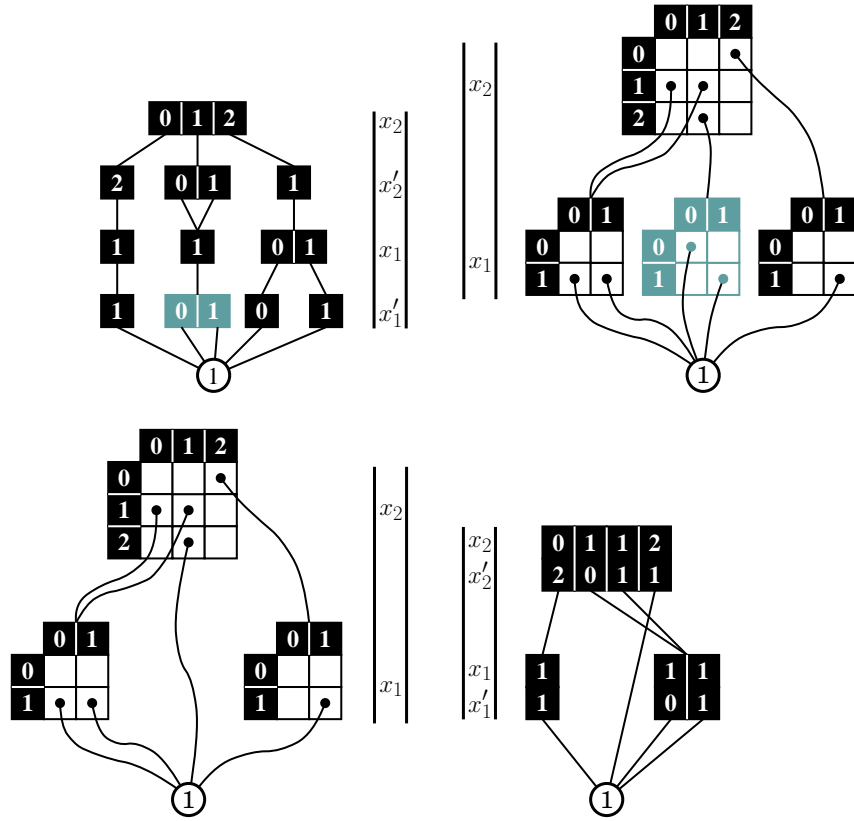


Fig. 4. Representing a boolean matrix with $2L$ -level MDDs or with MxDs.

in such a way that the value of the function is not found in a terminal node, but is distributed over the edges found along a path. The manipulation algorithms are generally more complex, but these classes of decision diagrams can be exponentially more compact than their respective terminal-valued versions.

Edge-valued BDDs (EVBDDs) [26] encode functions of the form $\mathbb{B}^L \rightarrow \mathbb{Z}$, by having a single terminal node “ Ω ”, which carries no value, and associating integer values to the edges of the diagram. The value of the function is obtained by *adding* the values encountered along the path to Ω corresponding to the function’s argument; the result is a possibly exponentially smaller diagram than with an MTBDD. Nodes are normalized by scaling their edge values so that the 0-edge has an associated value of 0 (this fact can be used to save storage in a practical implementation, but is also one way to enforce canonicity), and the root node has a “dangling arc” whose associated value is summed to the path value when evaluating the function, thus it is the value of the function being encoded when the argument is $(0, \dots, 0)$. The **EVMDD** shown in Fig. 5 on the left is

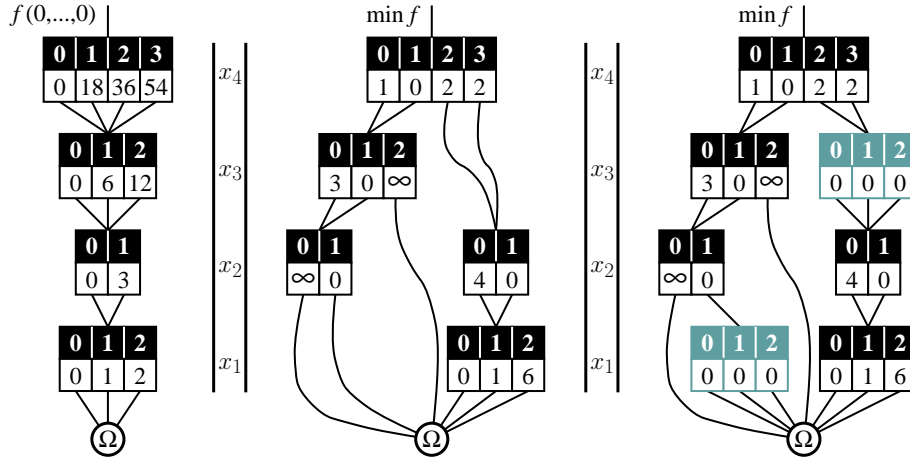


Fig. 5. EVMDDs, fully-reduced EV⁺MDDs, and quasi-reduced EV⁺MDDs.

the corresponding version with multi-way choices, it encodes $f(x_4, x_3, x_2, x_1) = \sum_{L \geq k \geq 1} x_k \cdot \prod_{k > h \geq 1} n_h$, i.e., the value of (x_4, x_3, x_2, x_1) interpreted as a *mixed-base* integer; the same function would require a full tree, thus exponential space, if encoded with an MTMDD. Formally, the function $v_{(\sigma,p)} : \hat{\mathcal{X}} \rightarrow \mathbb{Z}$ encoded by edge (σ, p) , where $\sigma \in \mathbb{Z}$ and p is a node at level k , is defined recursively as

$$v_{(\sigma,p)}(x_L, \dots, x_1) = \begin{cases} \sigma & \text{if } k = 0 \\ \sigma + v_{p[x_k]}(x_L, \dots, x_1) & \text{if } k > 0. \end{cases}$$

The **positive edge-valued MDDs (EV⁺MDDs)** [14] use a different normalization rule where all edge values leaving a node are non-negative or (positive) ∞ , but at least one of them is zero, so that the value associated with the dangling edge is the minimum of the function. To ensure canonicity, an edge with an associated value of ∞ can only point to Ω . EV⁺MDDs can encode arbitrary partial functions of the form $\hat{\mathcal{X}} \rightarrow \mathbb{N} \cup \{\infty\}$. For example, the function encoded by the EV⁺MDD in the middle of Fig. 5 cannot be encoded by an EVMDD because $f(0, \dots, 0) = \infty$, but f is not identically ∞ . Furthermore, if the dangling arc of the EV⁺MDD is allowed to be an arbitrary integer, then, arbitrary partial functions of the form $\hat{\mathcal{X}} \rightarrow \mathbb{Z} \cup \{\infty\}$ can be encoded. The EV⁺MDD shown on the right of Fig. 5 shows the equivalent quasi-reduced version: the condition for a node to be redundant (as the additional gray nodes in the figure are) is now that all of its edges point to the same node and have the same associated value, which must then be 0, given the normalization requirement. In the figure, long edges with an associated value of ∞ can still exist; alternatively, even those could be required to span only one level through the introduction of further redundant node, but this would not further simplify the manipulation algorithms.

We already saw boolean MxDs, but, originally, **(real-valued) matrix diagrams (MxDs)** [13, 27] were introduced to overcome some of the applicability

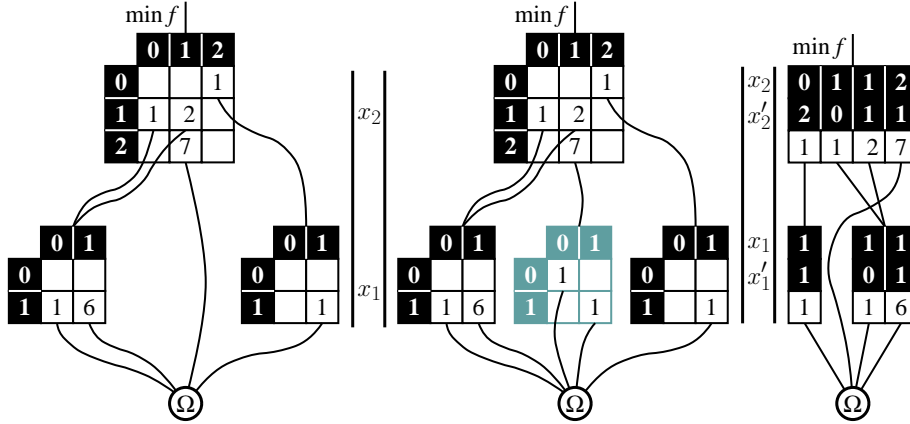


Fig. 6. Fully-reduced vs. quasi-reduced real-valued MxDs, and sparse representation.

and efficiency limitations encountered when using Kronecker algebra [18] to encode the transition rate matrix of large structured Markov chains [5, 7, 20]. These MxDs can encode functions of the form $\hat{\mathcal{X}}^2 \rightarrow \mathbb{R}^{\geq 0}$, that is, non-negative matrices, by having a row and column choice at each node and multiplying (instead of summing, as for EVMDDs) the values encountered along a path to the single terminal node Ω . Canonicity is enforced by requiring that the minimum non-zero edge value in each node be 1, so that the value associated to the dangling arc is, again, the minimum of the function (alternatively, one can require that the maximum edge value be 1, so that the value associated to the dangling is the maximum of the function). Fig. 6 on the left shows a two-level MxD; when an edge value is zero, the edge goes to Ω and is not shown for clarity, while the semantic of an edge skipping a level is that a redundant node with an identity pattern is assumed. Thus, in the figure, $f(x_2, x'_2, x_1, x'_1)$ is 7 when $x_2 = 2$, $x'_2 = 1$, $x_1 = x'_1$, and it is 0 when $x_2 = 2$, $x'_2 = 1$, $x_1 \neq x'_1$. The center of the same figure shows the quasi-reduced version of this MxD, where the only long edges are those with an associated value of 0, thus are not shown, while the right shows its sparse representation.

2.4 Decision diagram manipulation algorithms

So far we have discussed static aspects of various classes of decision diagrams, i.e., we have seen how decision diagrams can compactly encode functions over large structured domains. However, their time efficiency is just as important as their ability to save memory. Thus, we now turn to dynamic aspects of their manipulation. Decision diagram algorithms can usually be elegantly expressed in a recursive style. Two data structures are essential to achieve the desired efficiency. First, a *unique table* is required to enforce canonicity. This is a hash table that can be searched based on the level and the pattern of edges (and values, in the edge-valued case) of a node, to avoid creating duplicate nodes.

Second, an *operation cache* is used to look up whether a needed result of some operation on some nodes has been previously computed. Also this is a hash table, this time searched on the argument nodes' unique identifier (e.g., their memory address) and the operation code.

For example, Fig. 7 shows two BDD algorithms. The first one, *Or*, computes the disjunction of its arguments, i.e., given two functions $v_a, v_b : \mathbb{B}^L \rightarrow \mathbb{B}$, encoded by the two nodes a and b , it computes the node r encoding function v_r satisfying $v_r(\mathbf{i}) = v_a(\mathbf{i}) \vee v_b(\mathbf{i})$, for all $\mathbf{i} \in \mathbb{B}^L$. This algorithm assumes that these function are encoded in a fully-reduced BDD forest, thus, after testing for trivial cases that can be directly answered without recursion, it must test the level of a and b and proceed accordingly. Note that checking whether *Cache* contains already a result r for the *Or* of a and b is essential to efficiency; without it, the complexity would be proportional to the number of *paths* in the BDDs, not to the number of *nodes* in them.

The second algorithm in Fig. 7 computes the so-called *relational product* r of an L -level BDD x with a $2L$ -level BDD t , i.e., $v_r(\mathbf{j}) = 1 \Leftrightarrow \exists \mathbf{i}, v_x(\mathbf{i}) = 1 \wedge v_t(\mathbf{i}, \mathbf{j}) = 1$. Note that, as the BDDs are assumed to be quasi-reduced, the recursion on x and t proceeds in lockstep, i.e., level k of x is processed with levels k and k' of t to compute level k (conceptually k') of the result, thus there is no need to check for the levels of the nodes, as in the fully-reduced case.

Edge-valued decision diagram algorithms also operate recursively, but the arguments passed and returned in the recursions are *edges*, i.e., (value,node) pairs, rather than just nodes. For example, Fig. 8 shows an algorithm to compute the EV⁺MDD (μ, r) encoding the minimum of the function encoded by the two EV⁺MDDs (α, a) and (β, b) , i.e., $v_{(\mu, r)}(\mathbf{i}) = \min\{v_{(\alpha, a)}(\mathbf{i}), v_{(\beta, b)}(\mathbf{i})\}$, or, in other words, $\mu + v_{(0, r)}(\mathbf{i}) = \min\{\alpha + v_{(0, a)}(\mathbf{i}), \beta + v_{(0, b)}(\mathbf{i})\}$, for all $\mathbf{i} \in \widehat{\mathcal{X}}$. The notation $p[i].child$ and $p[i].val$ is used to denote the node pointed by the i^{th} edge, and the associate value, respectively.

Finally, it should be stressed that the efficiency of computing $f \otimes g$ for some arbitrary operator \otimes when the functions f and g are encoded by edge-valued decision diagrams depends on the relative properties of both the operator \otimes and the operator \odot used to combine the values along the paths of the decision diagram. For example, computing the elementwise minimum or sum of two functions encoded using EV⁺MDDs is quite efficient, but computing their elementwise product is obviously more difficult.

3 Discrete state models

We now describe the setting for the classes of system analysis we intend to perform symbolically, using decision diagrams. First we discuss logical analysis, then Markov analysis.

3.1 Logic system description and analysis

We consider a discrete-state model $(\widehat{\mathcal{X}}, \mathcal{X}_{init}, \mathcal{T})$, where $\widehat{\mathcal{X}}$ is a finite set of states, $\mathcal{X}_{init} \subseteq \widehat{\mathcal{X}}$ is the set of initial states, and $\mathcal{T} \subseteq \widehat{\mathcal{X}} \times \widehat{\mathcal{X}}$ is a transition relation. We

```

bdd Or(bdd a, bdd b) is • fully-reduced version
local bdd r, r0, r1;
local level k;
1 if a = 0 or b = 1 then return b;
2 if b = 0 or a = 1 then return a;
3 if a = b then return a;
4 if Cache contains entry ⟨ORcode, {a, b} : r⟩ then return r;
5 if a.lvl < b.lvl then • compare the levels of nodes a and b
6   k ← b.lvl;
7   r0 ← Or(a, b[0]);
8   r1 ← Or(a, b[1]);
9 else if a.lvl > b.lvl then
10  k ← a.lvl;
11  r0 ← Or(a[0], b);
12  r1 ← Or(a[1], b);
13 else • a.lvl = b.lvl
14  k ← a.lvl;
15  r0 ← Or(a[0], b[0]);
16  r1 ← Or(a[1], b[1]);
17 r ← UniqueTableInsert(k, r0, r1);
18 enter ⟨ORcode, {a, b} : r⟩ in Cache;
19 return r;

```

```

bdd RelProd(bdd x, bdd2 t) is • quasi-reduced version
local bdd r, r0, r1;
1 if x = 0 or t = 0 then return 0;
2 if x = 1 and t = 1 then return 1;
3 if Cache contains entry ⟨RELPRODcode, x, t : r⟩ then return r;
4 r0 ← Or(RelProd(x[0], t[0][0]), RelProd(x[1], t[1][0]));
5 r1 ← Or(RelProd(x[0], t[0][1]), RelProd(x[1], t[1][1]));
6 r ← UniqueTableInsert(x.lvl, r0, r1);
7 enter ⟨RELPRODcode, x, t : r⟩ in Cache;

```

Fig. 7. Examples of recursive algorithms on fully-reduced and quasi-reduced BDDs.

assume the *global* model state to be of the form (x_L, \dots, x_1) , where, for $L \geq k \geq 1$, each *local* state variable x_k takes value from a set $\mathcal{X}_k = \{0, 1, \dots, n_k - 1\}$, with $n_k > 0$. Thus, $\hat{\mathcal{X}} = \mathcal{X}_L \times \dots \times \mathcal{X}_1$ and we write $\mathcal{T}(i_L, \dots, i_1, i'_L, \dots, i'_1)$, or $\mathcal{T}(\mathbf{i}, \mathbf{i}')$, if the model can move from the *current state* \mathbf{i} to a *next state* \mathbf{i}' in one step.

The first step in system analysis is often the computation of the *reachable state space*. The goal is to find the set \mathcal{X}_{reach} of states reachable from the initial set of states \mathcal{X}_{init} according to the transition relation \mathcal{T} . Let $\mathbf{i} \rightarrow \mathbf{i}'$ mean that state \mathbf{i} can reach state \mathbf{i}' in one step, i.e., $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}$, which we also write as $\mathbf{i}' \in \mathcal{T}(\mathbf{i})$ with a slight abuse of notation. Then, the reachable state space is

$$\mathcal{X}_{reach} = \{\mathbf{j} : \exists d > 0, \exists \mathbf{i}^{(1)} \rightarrow \mathbf{i}^{(2)} \rightarrow \dots \rightarrow \mathbf{i}^{(d)} \wedge \mathbf{i}^{(1)} \in \mathcal{X}_{init} \wedge \mathbf{j} = \mathbf{i}^{(d)}\}.$$

Further logic analysis might involve searching for *deadlocks* or proving certain *liveness* properties. In general, such questions can be expressed in some *temporal*

<pre> evmdd $Min(\text{level } k, \text{evmdd } (\alpha, a), \text{evmdd } (\beta, b))$ is local evmdd $(\mu, r), r_0, \dots, r_{n_k-1}, (\alpha', a'), (\beta', b')$; 1 if $\alpha = \infty$ then return (β, b); 2 if $\beta = \infty$ then return (α, a); 3 $\mu \leftarrow \min(\alpha, \beta)$; 4 if $k = 0$ then return (μ, Ω); 5 if <i>Cache</i> contains entry $\langle MINcode, a, b, \alpha - \beta : (\gamma, r) \rangle$ then return $(\gamma + \mu, r)$; 6 for $i = 0$ to $n_k - 1$ do 7 $a' \leftarrow a[i].child$; 8 $\alpha' \leftarrow \alpha - \mu + a[i].val$; 9 $b' \leftarrow b[i].child$; 10 $\beta' \leftarrow \beta - \mu + b[i].val$; 11 $r_i \leftarrow Min(k-1, (\alpha', a'), (\beta', b'))$; 12 $r \leftarrow UniqueTableInsert(k, r_0, \dots, r_{n_k-1})$; 13 enter $\langle MINcode, a, b, \alpha - \beta : (\mu, r) \rangle$ in <i>Cache</i>; 14 return (μ, r); </pre>	<ul style="list-style-type: none"> • <i>quasi-reduced version</i> • <i>the only node at level 0 is Ω</i> • <i>continue downstream</i>
--	--

Fig. 8. A recursive algorithms for EV⁺MDDs.

logic. Here, we assume the use of *computation tree logic (CTL)* [17, 23]. This requires a *Kripke structure*, i.e., augmenting our discrete state model with a set of *atomic propositions* \mathcal{A} and a *labeling function* $\mathcal{L} : \hat{\mathcal{X}} \rightarrow 2^{\mathcal{A}}$ giving, for each state $\mathbf{i} \in \hat{\mathcal{X}}$, the set of atomic propositions $\mathcal{L}(\mathbf{i}) \subseteq \mathcal{A}$ that hold in \mathbf{i} . Then, the syntax of CTL is as follows:

- if $a \in \mathcal{A}$, a is a *state formula*;
- if p and p' are state formulas, $\neg p$, $p \vee p'$, and $p \wedge p'$ are state formulas;
- if p and p' are state formulas, Xp , Fp , Gp , pUp' , pRp' are *path formulas*;
- if q is a path formula, Eq and Aq are state formulas.

The semantic of CTL assigns a set of model states to each state formula p , thus, CTL operators must occur in pairs: a *path quantifier*, E or A, must always immediately precede a *temporal operator*, X, F, G, U, R. For brevity, we discuss only the semantics of the operator pairs EX, EU, and EG, since these are *complete*, meaning that they can be used to express any of the other seven CTL operators through complementation, conjunction, and disjunction:

- $AXp = \neg EX\neg p$,
- $EFp = E[true \ U \ p]$,
- $E[pRq] = \neg A[\neg pU\neg q]$,
- $AFp = \neg EG\neg p$,
- $A[p \ U \ q] = \neg E[\neg q \ U \ \neg p \ \wedge \ \neg q] \ \wedge \ \neg EG\neg q$,
- $A[pRq] = \neg E[\neg pU\neg q]$, and
- $AGp = \neg EF\neg p$,

where *true* is a predicate that holds in any state.

Let \mathcal{P} and \mathcal{Q} be the sets of states satisfying two CTL formulas p and q , respectively. Then, the sets of states satisfying $\text{EX}p$, $\text{EpU}q$, and $\text{EG}p$ are:

$$\begin{aligned}\mathcal{X}_{\text{EX}p} &= \{\mathbf{i} : \exists \mathbf{i}', \mathbf{i} \rightarrow \mathbf{i}' \wedge \mathbf{i}' \in \mathcal{P}\}, \\ \mathcal{X}_{\text{EpU}q} &= \{\mathbf{i} : \exists d > 0, \exists \mathbf{i}^{(1)} \rightarrow \dots \rightarrow \mathbf{i}^{(d)} \wedge \mathbf{i} = \mathbf{i}^{(1)} \wedge \mathbf{i}^{(d)} \in \mathcal{Q} \wedge \forall c, 1 \leq c < d, \mathbf{i}^{(c)} \in \mathcal{P}\}, \\ \mathcal{X}_{\text{EG}p} &= \{\mathbf{i} : \forall d > 0, \exists \mathbf{i}^{(1)} \rightarrow \dots \rightarrow \mathbf{i}^{(d)} \wedge \mathbf{i} = \mathbf{i}^{(1)} \wedge \forall c, 1 \leq c \leq d, \mathbf{i}^{(c)} \in \mathcal{P}\}.\end{aligned}$$

Fig. 9 shows the pseudocode to compute the set of states satisfying $\text{EX}p$, $\text{EpU}q$, and $\text{EG}p$, or, more precisely, to “label” these states, assuming that the states satisfying the CTL formulas p and q have already been labeled. In other words, the labels corresponding to the subformulas of a CTL formulas are assigned first; of course, at the innermost level, the labeling corresponding to atomic propositions is just given by the labeling function \mathcal{L} of the Kripke structure. Unlike state-space generation, these algorithms “walk backwards” in the transition relation, thus use the *inverse transition relation* \mathcal{T}^{-1} instead of \mathcal{T} .

3.2 Markov system description and analysis

A discrete-state model or a Kripke structure can be extended by associating timing information to each state-to-state transition. If the time required for each transition is an exponentially distributed random variable independently sampled every time the state is entered and if all the transitions out of each state are “racing” concurrently, this defines an underlying *continuous-time Markov chain (CTMC)*. Formally, a CTMC is a stochastic process $\{X_t : t \in \mathbb{R}\}$ with a discrete state space $\mathcal{X}_{\text{reach}}$ satisfying the memoryless property, that is:

$$\forall r \geq 1, \forall t > t^{(r)} > \dots > t^{(1)}, \forall \mathbf{i}, \mathbf{i}^{(1)}, \dots, \mathbf{i}^{(r)} \in \mathcal{X}_{\text{reach}},$$

$$\Pr\{X_t = \mathbf{i} \mid X_{t^{(r)}} = \mathbf{i}^{(r)}, \dots, X_{t^{(1)}} = \mathbf{i}^{(1)}\} = \Pr\{X_t = \mathbf{i} \mid X_{t^{(r)}} = \mathbf{i}^{(r)}\}.$$

We limit our discussion to *homogeneous* CTMCs, where the above probability depends on t and $t^{(r)}$ only through the difference $t - t^{(r)}$, i.e.,

$$\Pr\{X_t = \mathbf{i} \mid X_{t^{(r)}} = \mathbf{i}^{(r)}\} = \Pr\{X_{t-t^{(r)}} = \mathbf{i} \mid X_0 = \mathbf{i}^{(r)}\}.$$

Let $\boldsymbol{\pi}_t$ be the *probability vector* denoting the probability $\boldsymbol{\pi}_t[\mathbf{i}] = \Pr\{X_t = \mathbf{i}\}$ of each state $\mathbf{i} \in \mathcal{X}_{\text{reach}}$ at time $t \geq 0$. A homogeneous CTMC is then described by its *initial probability vector* $\boldsymbol{\pi}_0$, satisfying

$$\boldsymbol{\pi}_0[\mathbf{i}] > 0 \Leftrightarrow \mathbf{i} \in \mathcal{X}_{\text{init}}$$

and by its *transition rate matrix* \mathbf{R} , defined by

$$\forall \mathbf{i}, \mathbf{j} \in \mathcal{X}_{\text{reach}}, \mathbf{R}[\mathbf{i}, \mathbf{j}] = \begin{cases} 0 & \text{if } \mathbf{i} = \mathbf{j} \\ \lim_{h \rightarrow 0} \Pr\{X_h = \mathbf{j} \mid X_0 = \mathbf{i}\} / h & \text{if } \mathbf{i} \neq \mathbf{j} \end{cases}$$

<i>ExplicitBuildEX</i> (p) is	
1 $\mathcal{X} \leftarrow \{\mathbf{i} \in \mathcal{X}_{reach} : p \in labels(\mathbf{i})\};$	• initialize \mathcal{X} with the states satisfying p
2 while $\mathcal{X} \neq \emptyset$ do	
3 pick and remove a state \mathbf{i}' from \mathcal{X} ;	
4 for each $\mathbf{i} \in \mathcal{T}^{-1}(\mathbf{i}')$ do	• state \mathbf{i} can transition to state \mathbf{i}'
5 $labels(\mathbf{i}) \leftarrow labels(\mathbf{i}) \cup \{EXp\};$	
<i>ExplicitBuildEU</i> (p, q) is	
1 $\mathcal{X} \leftarrow \{\mathbf{i} \in \mathcal{X}_{reach} : q \in labels(\mathbf{i})\};$	• initialize \mathcal{X} with the states satisfying q
2 for each $\mathbf{i} \in \mathcal{X}$ do	
3 $labels(\mathbf{i}) \leftarrow labels(\mathbf{i}) \cup \{E[pUq]\};$	
4 while $\mathcal{X} \neq \emptyset$ do	
5 pick and remove a state \mathbf{i}' from \mathcal{X} ;	
6 for each $\mathbf{i} \in \mathcal{T}^{-1}(\mathbf{i}')$ do	• state \mathbf{i} can transition to state \mathbf{i}'
7 if $E[pUq] \notin labels(\mathbf{i})$ and $p \in labels(\mathbf{i})$ then	
8 $labels(\mathbf{i}) \leftarrow labels(\mathbf{i}) \cup \{E[pUq]\};$	
9 $\mathcal{X} \leftarrow \mathcal{X} \cup \{\mathbf{i}\};$	
<i>ExplicitBuildEG</i> (p) is	
1 $\mathcal{X} \leftarrow \{\mathbf{i} \in \mathcal{X}_{reach} : p \in labels(\mathbf{i})\};$	• initialize \mathcal{X} with the states satisfying p
2 build the strongly connected components in the subgraph of \mathcal{T} induced by \mathcal{X} ;	
3 $\mathcal{Y} \leftarrow \{\mathbf{i} : \mathbf{i} \text{ is in one of these strongly connected component}\};$	
4 for each $\mathbf{i} \in \mathcal{Y}$ do	
5 $labels(\mathbf{i}) \leftarrow labels(\mathbf{i}) \cup \{EGp\};$	
6 while $\mathcal{Y} \neq \emptyset$ do	
7 pick and remove a state \mathbf{i}' from \mathcal{Y} ;	
8 for each $\mathbf{i} \in \mathcal{T}^{-1}(\mathbf{i}')$ do	• state \mathbf{i} can transition to state \mathbf{i}'
9 if $EGp \notin labels(\mathbf{i})$ and $p \in labels(\mathbf{i})$ then	
10 $labels(\mathbf{i}) \leftarrow labels(\mathbf{i}) \cup \{EGp\};$	
11 $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{\mathbf{i}\};$	

Fig. 9. Explicit CTL model checking algorithms.

or its *infinitesimal generator matrix* \mathbf{Q} , defined by,

$$\forall \mathbf{i}, \mathbf{j} \in \mathcal{X}_{reach}, \mathbf{Q}[\mathbf{i}, \mathbf{j}] = \begin{cases} -\sum_{\mathbf{l} \neq \mathbf{i}} \mathbf{R}[\mathbf{i}, \mathbf{l}] & \text{if } \mathbf{i} = \mathbf{j} \\ \mathbf{R}[\mathbf{i}, \mathbf{j}] & \text{if } \mathbf{i} \neq \mathbf{j}. \end{cases}$$

The short-term, or *transient*, behavior of the CTMC is found by computing the transient probability vector π_t , which is the solution of the ordinary differential equation $d\pi_t/dt = \pi_t \mathbf{Q}$ with initial condition π_0 , thus it is given by the *matrix exponential* expression $\pi_t = \pi_0 e^{\mathbf{Q}t}$. The long-term, or *steady-state*, behavior is found by computing the steady-state probability vector $\pi = \lim_{t \rightarrow \infty} \pi_t$; if the CTMC is *irreducible* (since we assume that \mathcal{X}_{reach} is finite, this implies that \mathcal{X}_{reach} is a single strongly-connected component), π is independent of π_0 and is the unique solution of the homogeneous linear system $\pi \mathbf{Q} = \mathbf{0}$ subject to $\sum_{\mathbf{i} \in \mathcal{X}_{reach}} \pi[\mathbf{i}] = 1$. The probability vectors π and π_t are typically used to eval-

```

real[n] Jacobi(real[n]  $\pi^{(old)}$ ,  $\mathbf{h}$ , real[n, n]  $\mathbf{R}$ ) is
local real[n]  $\pi^{(new)}$ ;
1 repeat
2   for  $j = 1$  to  $|\mathcal{X}_{reach}|$ 
3      $\pi^{(new)}[j] \leftarrow \mathbf{h}[j] \cdot \sum_{i:\mathbf{R}[i,j]>0} \pi^{(old)}[i] \cdot \mathbf{R}[i, j]$ ;
4      $\pi^{(new)} \leftarrow \pi^{(new)} / (\pi^{(new)} \cdot \mathbf{1})$ ;
5      $\pi^{(old)} \leftarrow \pi^{(new)}$ ;
6 until "converged";
7 return  $\pi^{(new)}$ ;

```

```

real[n] GaussSeidel(real[n]  $\pi$ ,  $\mathbf{h}$ , real[n, n]  $\mathbf{R}$ ) is
1 repeat
2   for  $j = 1$  to  $|\mathcal{X}_{reach}|$ 
3      $\pi[j] \leftarrow \mathbf{h}[j] \cdot \sum_{i:\mathbf{R}[i,j]>0} \pi[i] \cdot \mathbf{R}[i, j]$ ;
4    $\pi \leftarrow \pi / (\pi \cdot \mathbf{1})$ ;
5 until "converged";
6 return  $\pi$ ;

```

```

real[n] Uniformization(real[n]  $\pi_0$ , real[n, n]  $\mathbf{P}$ , real  $q, t$ , natural  $M$ ) is
local real[n]  $\pi_t$ ;
1  $\pi_t \leftarrow \mathbf{0}$ ;
2  $\gamma \leftarrow \pi_0$ ;
3  $Poisson \leftarrow e^{-qt}$ ;
4 for  $k = 1$  to  $M$  do
5    $\pi_t \leftarrow \pi_t + \gamma \cdot Poisson$ ;
6    $\gamma \leftarrow \gamma \cdot \mathbf{P}$ ;
7    $Poisson \leftarrow Poisson \cdot q \cdot t/k$ ;
8 return  $\pi_t$ ;

```

Fig. 10. Numerical solution algorithms for CTMCs ($n = |\mathcal{X}_{reach}|$).

uate expected *instantaneous reward measures*. For example, a reward function $r : \mathcal{X}_{reach} \rightarrow \mathbb{R}$ specifies the rate at which a “reward” is generated in each state, and its expected value in steady state is computed as $\sum_{\mathbf{i} \in \mathcal{X}_{reach}} \pi[\mathbf{i}]r(\mathbf{i})$.

It is also possible to evaluate *accumulated reward measures* over a time interval $[t_1, t_2]$, in either the transient ($t_1 < t_2 < \infty$) or the long term ($t_1 < t_2 = \infty$). The numerical algorithms and the issues they raise are similar to those for instantaneous rewards discussed above, thus we omit them for brevity.

In practice, for exact steady-state analysis, the linear system $\pi\mathbf{Q} = \mathbf{0}$ is solved using iterative methods such as Jacobi or Gauss-Seidel, since the matrix \mathbf{Q} is typically extremely large and quite sparse. If \mathbf{Q} is stored by storing matrix \mathbf{R} , in sparse row-wise or column-wise format, and the diagonal of \mathbf{Q} , as a full vector, the operations required for these iterative solution methods are vector-matrix multiplications, i.e., vector-column (of a matrix) dot products. In addition to \mathbf{Q} , the solution vector π must be stored, and most iterative methods (such as Jacobi) require one or more auxiliary vectors of the same dimension as π , $|\mathcal{X}_{reach}|$. For extremely large CTMCs, these auxiliary vectors may impose excessive memory requirements.

If matrix \mathbf{R} is stored in sparse column-wise format, and vector \mathbf{h} contains the expected “holding time” for each state, where $\mathbf{h}[\mathbf{i}] = -1/\mathbf{Q}[\mathbf{i}, \mathbf{i}]$, then the Jacobi method can be written as in Fig. 10, where $\boldsymbol{\pi}^{(new)}$ is an auxiliary vector and the matrix \mathbf{R} is accessed by columns, although the algorithm can be rewritten to access matrix \mathbf{R} by rows instead. The method of Gauss-Seidel, also shown in Fig. 10, is similar to Jacobi, except the newly computed vector entries are used immediately; thus only the solution vector $\boldsymbol{\pi}$ is stored, with newly computed entries overwriting the old ones. The Gauss-Seidel method can also be rewritten to access \mathbf{R} by rows, but this is not straightforward, requires an auxiliary vector, and adds more computational overhead [19].

For transient analysis, the *uniformization* method is most often used, as it is numerically stable and uses straightforward vector-matrix multiplications as its primary operations (Fig. 10). The idea is to *uniformize* the CTMC with a rate $q \geq \max_{\mathbf{i} \in \mathcal{X}_{reach}} \{\mathbf{Q}[\mathbf{i}, \mathbf{i}]\}$ and obtain a discrete-time Markov chain with transition probability matrix $\mathbf{P} = \mathbf{Q}/q + \mathbf{I}$. The number of iterations M must be large enough to ensure that $\sum_{k=0}^M e^{-qt} (qt)^k / k!$ is very close to 1.

4 Putting it all together: structured system analysis

We are now ready to show how the logical and numerical analysis algorithms of the previous section can be implemented *symbolically* using appropriate classes of decision diagrams. Most of these algorithms compute the *fixpoint* of some *functional*, i.e., a function transformer, where the fixpoint is a function encoded as a decision diagram.

4.1 Symbolic state space generation

State space generation is one of the simplest examples of symbolic fixpoint computation, and arguably the most important one. The reachable state space \mathcal{X}_{reach} can be characterized as the *smallest* solution of the fixpoint equation

$$\mathcal{X} \subseteq \mathcal{X}_{init} \cup \mathcal{T}(\mathcal{X}).$$

Algorithm *Bfs* in Fig. 11 implements exactly this fixpoint computation, where sets and relations are stored using L -level and $2L$ -level MDDs, respectively, i.e., node p encodes the set \mathcal{X}_p having characteristic function v_p satisfying

$$v_p(i_L, \dots, i_1) = 1 \Leftrightarrow (i_L, \dots, i_1) \in \mathcal{X}_p.$$

The union of sets is simply implemented by applying the *Or* operator of Fig. 7 to their characteristic functions, and the computation of the states reachable in one step is implemented by using function *RelProd*, also from Fig. 7 (of course, the MDD version of these functions must be employed if MDDs are used instead of BDDs). Since it performs a breadth-first symbolic search, algorithm *Bfs* halts in exactly as many iterations as the maximum distance of any reachable state from the initial states.

Many high-level formalisms can be used to implicitly describe the state space by specifying the initial state or states, thus \mathcal{X}_{init} , and a rule to generate the states reachable in one step from each state, thus \mathcal{T} . Most formalisms are not only “structured” in the sense that they define the model state through L variables (x_L, \dots, x_1) , which is of course required for any symbolic approach, but also “asynchronous”, in the sense that they *disjunctively partition* [8] the transition relation \mathcal{T} according to a set \mathcal{E} of *events*. When \mathcal{T} is expressed as $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$, it is usually more efficient to deviate from a strict breadth-first approach. Algorithm *BfsChaining* in Fig. 11 implements a *chaining* approach [31] where the effect of applying each event is immediately accumulated as soon as it is computed. Chaining is based on the observation that the number of symbolic iterations might be reduced if the application of asynchronous events is compounded sequentially. While the search order is not strictly breadth-first anymore when chaining is used, the number of iterations of the **repeat** loop is at most as large as for breadth-first search, and usually much smaller. However, the efficiency of symbolic state-space generation is determined not just by the *number* of iterations but also by their *cost*, i.e., by the size of the MDDs involved. In practice, chaining has been shown to be quite effective in many models, but its effectiveness can be greatly affected by the order in which events are applied.

Much larger efficiency improvements, however, can be usually achieved with the *Saturation* algorithm [12, 16]. Saturation is motivated by the observation that, in many distributed systems, *interleaving semantic* implies that multiple *events* may occur, each exhibiting a strong *locality*, i.e., affecting only a few state variables. We associate two sets of state variables with each event e :

$$\begin{aligned} \mathcal{V}_M(e) &= \{x_k : \exists \mathbf{i} = (i_L, \dots, i_1), \exists \mathbf{i}' = (i'_L, \dots, i'_1), \mathbf{i}' \in \mathcal{T}_e(\mathbf{i}) \wedge i_k \neq i'_k\} \quad \text{and} \\ \mathcal{V}_D(e) &= \{x_k : \exists \mathbf{i} = (i_L, \dots, i_1), \exists \mathbf{j} = (j_L, \dots, j_1), \forall h \neq k, i_h = j_h \wedge \mathcal{T}_e(\mathbf{i}) \neq \emptyset \wedge \mathcal{T}_e(\mathbf{j}) = \emptyset\}, \end{aligned}$$

the state variables that can be modified by e or that can disable e , respectively. Then, we let

$$Top(e) = \max\{k : x_k \in \mathcal{V}_M(e) \cup \mathcal{V}_D(e)\}$$

be the highest state variable, thus MDD level, affected by event e , and we partition the event set \mathcal{E} into $\mathcal{E}_k = \{e : Top(e) = k\}$, for $L \geq k \geq 1$.

Saturation computes multiple “lightweight” nested fixpoints. Starting from the quasi-reduced MDD encoding \mathcal{X}_{init} , Saturation begins by exhaustively applying events $e \in \mathcal{E}_1$ to each node p at level 1, until it is *saturated*, i.e., until $\bigcup_{e \in \mathcal{E}_1} \mathcal{T}_e(\mathcal{X}_p) \subseteq \mathcal{X}_p$. Then, it saturates nodes at level 2 by exhaustively applying to them all the events $e \in \mathcal{E}_2$, with the proviso that, if any new node at level 1 is created in the process, it is immediately saturated (by firing the events in \mathcal{E}_1 on it). The approach proceeds in bottom-up order, until the events in \mathcal{E}_L have been applied exhaustively to the topmost node r . At this point, the MDD is saturated, $\bigcup_{e \in \mathcal{E}} \mathcal{T}_e(\mathcal{X}_r) \subseteq \mathcal{X}_r$, and r encodes the reachable state space \mathcal{X}_{reach} .

Experimentally, Saturation has been shown to be often several orders of magnitude more efficient than symbolic breadth-first iterations in both memory and time, when employed on asynchronous systems. Extensions of the Saturation algorithm have also been presented, where the size and composition of the local

<pre> mdd <i>Bfs</i>(mdd \mathcal{X}_{init}) is local mdd p; 1 $p \leftarrow \mathcal{X}_{init}$; 2 repeat 3 $p \leftarrow Or(p, RelProd(p, \mathcal{T}))$; 4 until p does not change; 5 return p; </pre>	
<pre> mdd <i>BfsChaining</i>(mdd \mathcal{X}_{init}) is local mdd p; 1 $p \leftarrow \mathcal{X}_{init}$; 2 repeat 3 for each $e \in \mathcal{E}$ do 4 $p \leftarrow Or(p, RelProd(p, \mathcal{T}_e))$; 5 until p does not change; 6 return p; </pre>	
<pre> mdd <i>Saturation</i>(mdd \mathcal{X}_{init}) is 1 return <i>Saturate</i>(L, \mathcal{X}_{init}); </pre>	• <i>assumes quasi-reduced MDDs</i>
<pre> mdd <i>Saturate</i>(level k, mdd p) is local mdd r, r_0, \dots, r_{n_k-1}; 1 if $p = 0$ then return 0; 2 if $p = 1$ then return 1; 3 if <i>Cache</i> contains entry $\langle SATcode, p : r \rangle$ then return r; 4 for $i =$ to $n_k - 1$ do 5 $r_i \leftarrow Saturate(k - 1, p[i])$; • <i>first, be sure that the children are saturated</i> 6 repeat 7 choose $e \in \mathcal{E}_k, i, j \in \mathcal{X}_k$, such that $r_i \neq 0$ and $\mathcal{T}_e[i][j] \neq 0$; 8 $r_j \leftarrow Or(r_j, RelProdSat(k - 1, r_i, \mathcal{T}_e[i][j]))$; 9 until r_0, \dots, r_{n_k-1} do not change; 10 $r \leftarrow UniqueTableInsert(k, r_0, \dots, r_{n_k-1})$; 11 enter $\langle SATcode, p : r \rangle$ in <i>Cache</i>; 12 return r; </pre>	
<pre> mdd <i>RelProdSat</i>(level k, mdd q, mdd2 f) is local mdd r, r_0, \dots, r_{n_k-1}; 1 if $q = 0$ or $f = 0$ then return 0; 2 if <i>Cache</i> contains entry $\langle RELPRODSATcode, q, f : r \rangle$ then return r; 3 for each $i, j \in \mathcal{X}_k$ such that $q[i] \neq 0$ and $f[i][j] \neq 0$ do 4 $r_j \leftarrow Or(r_j, RelProdSat(k - 1, q[i], f[i][j]))$; 5 $r \leftarrow Saturate(k, UniqueTableInsert(k, r_0, \dots, r_{n_k-1}))$; 6 enter $\langle RELPRODSATcode, q, f : r \rangle$ in <i>Cache</i>; 7 return r. </pre>	

Fig. 11. Symbolic breadth-first, chaining, and Saturation state-space generation.

state spaces \mathcal{X}_k is not known prior to generating the state space; rather, the local state spaces are built “on-the-fly” alongside the (global) reachable state space during the symbolic iterations [11, 16].

4.2 Symbolic CTL model checking

Moving now to symbolic CTL model checking, we need MDD-based algorithms for the EX, EU, and EG operators. The first requires no fixpoint computation, as it can be computed in one step:

$$\mathcal{X}_{\text{Exp}} = \mathcal{T}^{-1}(\mathcal{P})$$

The set of states satisfying $\text{EpU}q$ can instead be characterized as the *smallest* solution of the fixpoint equation

$$\mathcal{X} \subseteq \mathcal{Q} \cup (\mathcal{P} \cap \mathcal{T}^{-1}(\mathcal{X})),$$

while the set of states satisfying $\text{EG}p$ can be characterized as the *largest* solution of the fixpoint equation

$$\mathcal{X} \supseteq \mathcal{P} \cap \mathcal{T}^{-1}(\mathcal{X}).$$

Fig. 12 shows the pseudocode for a breadth-first-style symbolic implementation of these three operators. Again, all sets and relations are encoded using L -level and $2L$ -level MDDs, respectively. Chaining and Saturation versions of the symbolic EU algorithm are possible when \mathcal{T} is disjunctively partitioned. These are usually much more efficient in terms of both memory and time [15].

4.3 Symbolic Markov analysis

Approaches that exploit the structured representation of the transition rate matrix \mathbf{R} of a CTMC have been in use for over two decades, based on *Kronecker algebra* [18]. Formally, such approaches require \mathbf{R} to be the submatrix corresponding to the reachable states of a matrix $\widehat{\mathbf{R}} \in \mathbb{R}^{|\widehat{\mathcal{X}} \times \widehat{\mathcal{X}}|}$ expressed as a sum of *Kronecker products*:

$$\mathbf{R} = \widehat{\mathbf{R}}[\mathcal{X}_{\text{reach}}, \mathcal{X}_{\text{reach}}] \quad \text{where} \quad \widehat{\mathbf{R}} = \sum_{e \in \mathcal{E}} \widehat{\mathbf{R}}_e \quad \text{and} \quad \widehat{\mathbf{R}}_e = \bigotimes_{L \geq k \geq 1} \mathbf{R}_{e,k}.$$

$\widehat{\mathbf{R}}_e$ is the transition rate matrix due to event e and can be decomposed as the Kronecker product of L matrices $\mathbf{R}_{e,k} \in \mathbb{R}^{|\widehat{\mathcal{X}} \times \widehat{\mathcal{X}}|}$, each expressing the contribution of the local state x_k to the rate of event e [20, 30]. Recall that the Kronecker product of two matrices $\mathbf{A} \in \mathbb{R}^{n_r \times n_c}$ and $\mathbf{B} \in \mathbb{R}^{m_r \times m_c}$ is a matrix $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{n_r \cdot m_r \times n_c \cdot m_c}$ satisfying

$$\forall i_a \in \{0, \dots, n_r - 1\}, \forall j_a \in \{0, \dots, n_c - 1\}, \forall i_b \in \{0, \dots, m_r - 1\}, \forall j_b \in \{0, \dots, m_c - 1\},$$

$$(\mathbf{A} \otimes \mathbf{B})[i_a \cdot m_r + i_b, j_a \cdot m_c + j_b] = \mathbf{A}[i_a, j_a] \cdot \mathbf{B}[i_b, j_b].$$

The algorithms for the numerical solution of CTMCs shown in Fig. 10 essentially perform a sequence of vector-matrix multiplications at their core. Thus, the efficient computation of $\mathbf{y} \leftarrow \mathbf{x} \cdot \mathbf{A}$ or $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{x} \cdot \mathbf{A}$, when \mathbf{A} , or, rather, $\widehat{\mathbf{A}}$, is encoded as a Kronecker product $\bigotimes_{L \geq k \geq 1} \mathbf{A}_k$, has been studied at length [6]. The

```

mdd SymbolicBuildEX(mdd  $\mathcal{P}$ , mdd2  $T^{-1}$ ) is
local mdd  $\mathcal{X}$ ;
1  $\mathcal{X} \leftarrow \text{RelProd}(\mathcal{P}, T^{-1})$ ;    • perform one backward step in the transition relation
2 return  $\mathcal{X}$ ;

```

```

mdd SymbolicBuildEU(mdd  $\mathcal{P}$ , mdd  $\mathcal{Q}$ , mdd2  $T^{-1}$ ) is
local mdd  $\mathcal{O}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$ ;
1  $\mathcal{X} \leftarrow \mathcal{Q}$ ;    • initialize the currently known result with the states satisfying  $q$ 
2 repeat
3    $\mathcal{O} \leftarrow \mathcal{X}$ ;    • save the old set of states
4    $\mathcal{Y} \leftarrow \text{RelProd}(\mathcal{X}, T^{-1})$ ; • perform one backward step in the transition relation
5    $\mathcal{Z} \leftarrow \text{And}(\mathcal{Y}, \mathcal{P})$ ; • perform set intersection to discard states not satisfying  $p$ 
6    $\mathcal{X} \leftarrow \text{Or}(\mathcal{Z}, \mathcal{X})$ ;    • add to the currently known result
7 until  $\mathcal{O} = \mathcal{X}$ ;
8 return  $\mathcal{X}$ ;

```

```

mdd SymbolicBuildEG(mdd  $\mathcal{P}$ , mdd2  $T^{-1}$ ) is
local mdd  $\mathcal{O}, \mathcal{X}, \mathcal{Y}$ ;
1  $\mathcal{X} \leftarrow \mathcal{P}$ ;    • initialize  $\mathcal{X}$  with the states satisfying  $p$ 
2 repeat
3    $\mathcal{O} \leftarrow \mathcal{X}$ ;    • save the old set of states
4    $\mathcal{Y} \leftarrow \text{RelProd}(\mathcal{X}, T^{-1})$ ; • perform one backward step in the transition relation
5    $\mathcal{X} \leftarrow \text{And}(\mathcal{X}, \mathcal{Y})$ ;
6 until  $\mathcal{O} = \mathcal{X}$ ;
7 return  $\mathcal{X}$ ;

```

Fig. 12. Symbolic CTL model checking algorithms for the EX, EU, and EG operators.

well-known *shuffle algorithm* [18, 20] or other algorithms presented in [6] can be employed to compute this product, but their efficiency stems from two properties common to all algorithms for decision diagram manipulation. First, the object being encoded (\mathbf{A} , in this case) can be structured into L levels (the matrices \mathbf{A}_k , in this case). Second, some kind of caching is used to avoid recomputing the result of operations already performed.

One disadvantage of earlier Kronecker-based approaches for the steady-state or transient solution of a CTMC was that the probability vector $\boldsymbol{\pi} \in \mathbb{R}^{|\mathcal{X}_{reach}|}$ was actually stored using a possibly much larger probability vector $\hat{\boldsymbol{\pi}} \in \mathbb{R}^{|\hat{\mathcal{X}}|}$, to simplify state indexing. Correctness was achieved by ensuring that $\hat{\boldsymbol{\pi}}[\mathbf{i}] = 0$ for all $\mathbf{i} \in \hat{\mathcal{X}} \setminus \mathcal{X}_{reach}$, but the additional computational and, especially, memory overhead was substantial. Decision diagrams helped first by providing an efficient encoding for state indices. Given the MDD for $\mathcal{X}_{reach} \subseteq \hat{\mathcal{X}}$, an EV⁺MDD encoding the *lexicographic state indexing* function $\psi : \hat{\mathcal{X}} \rightarrow \mathbb{N} \cup \{\infty\}$ can be easily built. Its definition is such that $\psi(\mathbf{i})$ is the number of reachable states preceding \mathbf{i} in lexicographic order, if $\mathbf{i} \in \mathcal{X}_{reach}$, and is ∞ otherwise. This EV⁺MDD has exactly the same number of nodes and edges as the MDD encoding \mathcal{X}_{reach} . For example, Fig. 13 shows a reachable state space \mathcal{X}_{reach} , the MDD encoding it, and the EV⁺MDD encoding ψ . To compute the index of a state, sum the

$$\mathcal{X}_{reach} = \left\{ \begin{array}{cccccccccccccccccccc} 0 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 2 & 0 & 0 & 0 & 1 & 1 & 2 & 0 & 0 & 1 & 1 & 2 & 0 & 1 & 2 & 0 & 2 & 2 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 1 & 2 \end{array} \right\}$$

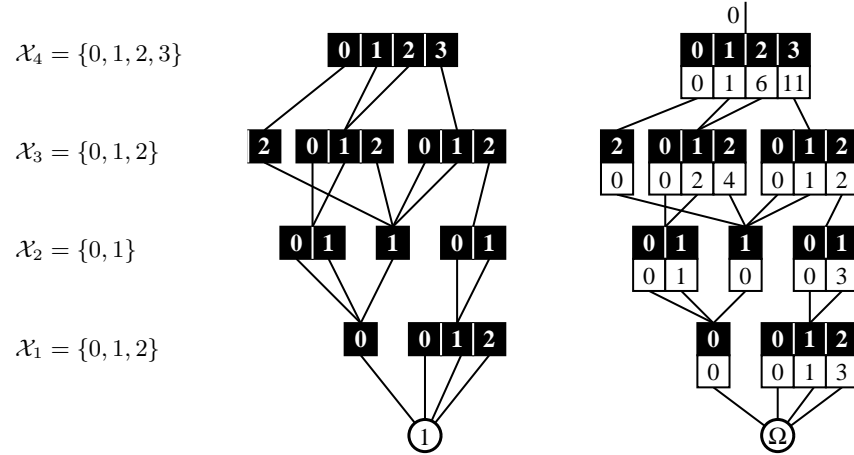


Fig. 13. Encoding the lexicographic state indexing function ψ using EV⁺MDDs.

values found on the corresponding path: $\psi(2, 1, 1, 0) = 6 + 2 + 1 + 0 = 9$; if a state is unreachable, the path is not complete: $\psi(0, 2, 0, 0) = 0 + 0 + \infty = \infty$. With this encoding of ψ , the probability vector π can be stored in a full array of size $|\mathcal{X}_{reach}|$, instead of $|\hat{\mathcal{X}}|$, with little indexing overhead, making the Kronecker approach much more memory efficient.

However, at their best, the Kronecker algorithms that have been presented in the literature can barely match the efficiency of good decision diagram algorithms. Thus, we limit our discussion to the latter, and, in particular, to MxDs, which were indeed defined as a more general and efficient alternative to a Kronecker encoding. In this context, an MxD can be seen as a generalization of a Kronecker encoding, as both multiply L elements, corresponding to the L local states, to obtain an entry of the encoded matrix, and both exploit the likely occurrence of an identity matrix at any level to avoid useless “multiplications by 1”. In addition, MxD can encode arbitrary matrices $\hat{\mathbf{R}}_e$, even those that are cannot be expressed as a Kronecker product of L local matrices, and can reflect the actual reachability of states. This means that, given the MDD encoding \mathcal{X}_{reach} and an MxD encoding a matrix $\hat{\mathbf{R}}$ such that $\hat{\mathbf{R}}[\mathcal{X}_{reach}, \mathcal{X}_{reach}] = \mathbf{R}$, we can enforce the fact that the entries of $\hat{\mathbf{R}}$ should be 0 for any unreachable row $\mathbf{i} \in \hat{\mathcal{X}} \setminus \mathcal{X}_{reach}$. Alternatively, these *spurious* nonzero entries can be dealt with explicitly, by testing for reachability, when using Gauss-Seidel iterations, which are best implemented with access-by-column to \mathbf{R} (with the Kronecker approach, only this second explicit filtering of the spurious entries is possible).

```

real[n] VectorMatrixMult(real[n] x, mxd_node A, evmdd_node ψ) is
local natural s;                                • state index in x
local real[n] y;
local sparse_real c;
1 s ← 0;
2 for each j = (jL, ..., j1) ∈ Xreach in lexicographic order do      • s = ψ(j)
3   c ← GetCol(L, A, ψ, jL, ..., j1);    • build column j of A using sparse storage
4   y[s] ← ElementWiseMult(x, c);    • x uses full storage, c uses sparse storage
5   s ← s + 1;
6 return y;

sparse_real GetCol(level k, mxd_node M, evmdd_node φ, natural jk, ..., j1) is
local sparse_real c, d;
1 if k = 0 then return [1];                • a vector of size one, with its entry set to 1
2 if Cache contains entry ⟨COLcode, M, φ, jk, ..., j1 : c⟩ then return c;
3 c ← 0;                                    • initialize the results to all zero entries
4 for each ik ∈ Xk such that M[ik, jk].val ≠ 0 and φ[ik].val ≠ ∞ do
5   d ← GetCol(k - 1, M[ik, jk].child, φ[ik].child, jk-1, ..., j1);
6   for each i such that d[i] ≠ 0 do
7     c[i + φ[ik].val] ← c[i + φ[ik].val] + M[ik, jk].val · d[i];
8 enter ⟨COLcode, M, φ, jk, ..., j1 : c⟩ in Cache;
9 return c;

```

Fig. 14. MxD-based vector-matrix multiplication algorithm ($n = |\mathcal{X}_{reach}|$).

We now discuss algorithm *VectorMatrixMult* of Fig. 14, which multiplies a full real vector \mathbf{x} of size $|\mathcal{X}_{reach}|$ by the submatrix $\hat{\mathbf{A}}[\mathcal{X}_{reach}, \mathcal{X}_{reach}]$, where $\hat{\mathbf{A}}$ is encoded by an MxD rooted at node A . For simplicity, we ignore the value ρ of the incoming dangling edge, i.e., we assume that $\rho = 1$. In practice, we could enforce this assumption by allowing the root node A , and only it, to be unnormalized, so that its entries are multiplied by ρ . Also for simplicity, we assume that the MxD is quasi-reduced, thus $A.lvl = L$. The correctness of this algorithm does not depend on the absence of spurious entries in $\hat{\mathbf{A}}$, since the lexicographic state indexing function, encoded by an EV⁺MDD with root edge $(0, \psi)$, is used to select only the rows corresponding to reachable states.

Algorithm *VectorMatrixMult* operates by building the column of $\hat{\mathbf{A}}$ corresponding to each reachable state \mathbf{j} . As such column is usually very sparse, it is stored in a sparse data structure, not as a full array. The key procedure is *GetCol* [29], which recursively builds the required column, filtering out entries corresponding to unreachable rows. Once the column \mathbf{c} is returned, *VectorMatrixMult* can perform an efficient multiplication of $\mathbf{x}^T \cdot \mathbf{c}$, by simply examining the nonzero entries of \mathbf{c} and using direct access to the corresponding entries of \mathbf{x} . Note that, since the columns \mathbf{j} are built in lexicographic order, the state index s can be simply incremented for each new column.

The algorithm just presented is considered the current state-of-the-art, but it nevertheless shares an important limitation with all *hybrid* approaches (including Kronecker-based ones): the vector \mathbf{x} , thus the probability vector $\boldsymbol{\pi}$, uses full storage, thus requires $O(|\mathcal{X}_{reach}|)$ memory. Sparse storage does not help (since

none of its entries is zero if the CTMC is ergodic) and symbolic storage usually requires even more memory, since the MTMDD or the EV⁺MDD (or its multiplicative analogue where edge values are multiplied along the path) end up being close to a tree with as many leaves as reachable states. Strictly symbolic approaches where not only \mathbf{R} but also $\boldsymbol{\pi}$ is stored using decision diagrams have been successful so far only when many states share the same probability, usually a clear indication that the CTMC exhibits strong symmetries, and is thus *lumpable* [25].

5 Conclusion and future areas of research

Major advances in our capability of analyzing large and complex systems have been already achieved through the use of decision diagram techniques. Nevertheless, several research challenges still lie ahead.

Variable ordering heuristics. It is well known that the ordering of the L state variables can exponentially affect the size of the decision diagrams, thus the efficiency of their manipulation. Unfortunately, finding an optimal variable order is known to be a hard problem [2], thus, heuristics for either the static ordering (at the beginning of the analysis, prior to building any decision diagram) or the dynamic ordering (during the analysis, if the decision diagrams become too large) of state variables have been proposed with varying degree of success [22].

Most approaches have been targeted at BDDs and breadth-first iterations. We have begun considering the special requirements of Saturation and MDDs in [32, 10], but much more work is required to explore heuristics for arbitrary classes of decision diagrams and their manipulations.

Strictly symbolic numerical CTMC solution. As mentioned, hybrid solution approaches for CTMCs are probably as efficient as they can be, at least in a general setting, i.e., unless the model belongs to a special class whose properties can be exploited to gain some efficiency on a case-by-case basis. A strictly symbolic approach, possibly allowing for a controlled level of approximation, might then be a very valuable contribution. Along these lines, an approach that uses exact symbolic representations of \mathcal{X}_{reach} and \mathbf{R} but an approximate representation for $\boldsymbol{\pi}$ has been proposed [28], but much more work is required.

References

1. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206, Apr. 1997.
2. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
4. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):293–318, 1992.

5. P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.
6. 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, 2000.
7. P. Buchholz, J. P. Katoen, P. Kemper, and C. Tepper. Model-checking large structured Markov chains. *J. Logic & Algebraic Progr.*, 56(1/2):69–97, 2003.
8. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *Int. Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.
9. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
10. G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In *Proc. 28th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*, Siedlce, Poland, June 2007. Springer-Verlag. To appear.
11. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.
12. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, Feb. 2006.
13. G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.
14. G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In M. D. Aagaard and J. W. O’Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.
15. G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In W. Hunt, Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, LNCS 2725, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.
16. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In D. Borriore and W. Paul, editors, *Proc. CHARME*, LNCS 3725, pages 146–161, Saarbrücken, Germany, Oct. 2005. Springer-Verlag.
17. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71, London, UK, 1981. Springer-Verlag.
18. M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30:116–125, Feb. 1981.
19. D. D. Deavours and W. H. Sanders. “On-the-fly” solution techniques for stochastic Petri nets and extensions. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 132–141, Saint-Malo, France, June 1997. IEEE Comp. Soc. Press.

20. P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.
21. M. Fujita, P. C. McGeer, , and J. C.-Y. Yang. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.
22. O. Grumberg, S. Livne, and S. Markovitch. Learning to order BDD variables in verification. *J. Art. Int. Res.*, 18:83–116, 2003.
23. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
24. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
25. J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand-Reinhold, New York, NY, 1960.
26. Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proceedings of the 29th Conference on Design Automation*, pages 608–613, Los Alamitos, CA, USA, June 1992. IEEE Computer Society Press.
27. A. S. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In R. German and B. Haverkort, editors, *Proc. 9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, Sept. 2001. IEEE Comp. Soc. Press.
28. A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In J. Kurose and P. Nain, editors, *Proc. ACM SIGMETRICS*, pages 207–216, Santa Clara, CA, USA, June 2000. ACM Press.
29. A. S. Miner and D. Parker. Symbolic representations and analysis of large state spaces. In C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, editors, *Validation of Stochastic Systems*, LNCS 2925, pages 296–338. Springer-Verlag, 2004.
30. B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. ACM SIGMETRICS*, pages 147–153, Austin, TX, USA, May 1985.
31. O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In G. De Michelis and M. Diaz, editors, *Proc. 16th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 935, pages 374–391, Turin, Italy, June 1995. Springer-Verlag.
32. R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In H. Hermanns and J. Palsberg, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 3920, pages 90–104, Vienna, Austria, Mar. 2006. Springer-Verlag.