Research Directions in Decision Diagram Technology

Gianfranco Ciardo Andrew S. Miner

Department of Computer Science
Iowa State University
{ciardo,asminer}@iastate.edu

***Abstract*—Binary decision diagrams (BDDs) have been a huge success story in hardware and software verification and are increasingly applied to a wide range of combinatorial problems.**

While BDDs can encode boolean-valued functions of boolean-valued variables, many BDD variants have been proposed, not just to improve their efficiency, but to manage multivalued domains (a straightforward extension), multivalued ranges (using several competitive alternatives), and two-dimensional data (relations and matrices instead of sets or vectors).

Orthogonally to these extensions, much effort has been spent on variable order heuristics, an essential aspect that can affect memory and time requirements by up to an exponential factor.

We survey some of these exciting results and discuss some fruitful research directions for further work.

***Index Terms*—Binary decision diagrams, canonicity, discrete function encoding, variable order heuristics, Markov chains**

I. Introduction

Binary decision diagrams (BDDs) were introduced in 1959 by Lee [28] as a way to encode boolean functions of boolean variables, but truly had a major impact only after Bryant's seminal 1986 paper [9]. Since then, many variants (notably, *zero-suppressed BDDs* [40], to encode "sparse" boolean functions, i.e., functions that often evaluate to 0 when some input variables have value 1) and extensions (*multivalued decision diagrams* [24] to manage non-boolean domains; *multiterminal* [22] and *edge-valued* [27] BDDs, to manage non-boolean ranges) have been defined and applied to fields from VLSI verification of both combinatorial and sequential circuits [12] and general symbolic model checking [23], to various combinatorial problems involving enumeration of, or search in, very large but finite structured domains [39].

We survey some of the most common BDD variants, describe our contributions to improve their efficiency and extend their applicability, and outline several research questions whose answers, we believe, could further strengthen the impact of this fundamental data structure.

II. Background: Binary Decision Diagrams

BDDs are directed acyclic graphs that encode boolean functions of boolean variables. In the following, we limit ourselves to *ordered* BDDs, and in particular we recall three widely used variants, each of them *canonical*, i.e., able to uniquely encode any function $f:\mathbb{B}^L\to\mathbb{B}$, once we impose an *order* on the (input) variables $\{x_1,...,x_L\}$.

A. Definitions of QBDDS, FBDDs, and ZBDDs.

We first introduce the concept of a generic ordered BDD.

Ordered BDDs. An L-level *ordered BDD* is an acyclic directed edge-labeled graph with terminal nodes $\mathbf{0}$ and $\mathbf{1}$, at level 0 (corresponding to the range of the functions encoded by BDD nodes), while a nonterminal node p belongs to level $p.lvl = k \in \{1,...,L\}$ (corresponding to boolean variable x_k) and has a 0-child $p[0]$ and a 1-child $p[1]$ satisfying the *ordered property*: $k > p[0].lvl$ and $k > p[1].lvl$. □

The edge to the 0-child (also known as the "low" child) is drawn with a dashed line, while the edge to the 1-child (or "high" child) is drawn with a solid line. Without providing a semantics for *long edges* (from a node at level k to a node at level $k-2$ or below, i.e., skipping one or more levels), the function encoded by an ordered BDD cannot be defined.

QBDDs [26]. An L-level ordered BDD is a *quasi-reduced BDD* (QBDD) if it has no *duplicates*, i.e., no nodes p and q at level $k > 0$ satisfy $p[0] = q[0]$ and $p[1] = q[1]$ (required of all DD forms we consider) and no long edges, i.e., if node p is at level $k > 0$, $p[0].lvl = p[1].lvl = k-1$. Then, the function $f_p : \mathbb{B}^l \to \mathbb{B}$ encoded by a QBDD node p at level k is defined recursively as (let $i_{h:k}$ denote $i_h,...,i_k$):

$$f_p(i_{1:k}) = \begin{cases} f_{p[i_k]}(i_{1:k-1}) & \text{if } p.lvl > 0 \\ p & \text{if } p \in \{\mathbf{0},\mathbf{1}\}. \end{cases}$$ □

Thus, only QBDD nodes at level L (called "roots") encode functions of all L variables.

FBDDs (Bryant's BDDs) [9]. An L-level ordered BDD is a *fully-reduced BDD* (FBDD) if it has no *duplicates* and no *redundant* nodes, i.e., no nonterminal node p s.t. $p[0] = p[1]$. The function $f_p : \mathbb{B}^L \to \mathbb{B}$ encoded by a FBDD node p at level k is defined recursively as:

$$f_p(i_{1:L}) = \begin{cases} f_{p[i_k]}(i_{1:L}) & \text{if } p.lvl > 0 \\ p & \text{if } p \in \{\mathbf{0},\mathbf{1}\}. \end{cases}$$ □

Thus, any FBDD node encodes a function of all L variables regardless of its level, and the meaning of a long edge is that the skipped variables, including those above the level of the node, are *don't-cares*. This reduction is effective if it is often the case that the value of the encoded function does not change if we flip the value of some variable, as this corresponds to a redundant QBDD node.

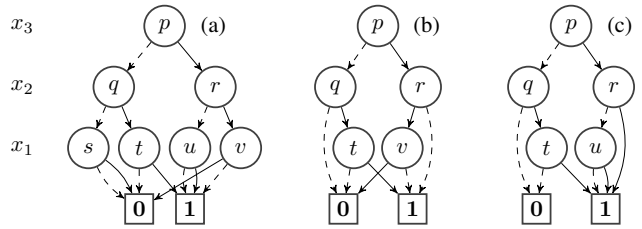

Fig. 1. (a) QBDD, (b) FBDD, (c) ZBDD encoding $\overline{x_3}x_2x_1 \vee x_3\overline{x_2x_1}$.

ZBDDs [30]. An L-level ordered BDD is a *zero-suppressed BDD* (ZBDD) if it has no *duplicates* and no *high-zero* nodes, i.e., no nonterminal node p satisfies $p[1] = \mathbf{0}$. The function $f_p : \mathbb{B}^l \to \mathbb{B}$ encoded by a ZBDD node p at level k must be defined recursively with respect to a level $l \geq k$:

$$f_p^l(i_{1:l}) = \begin{cases} \mathbf{0} & \text{if } l > k \text{ and } (i_{k+1} \vee \cdots \vee i_l) = 1 \\ f_p^k(i_{1:k}) & \text{if } l > k \text{ and } (i_{k+1} \vee \cdots \vee i_l) = 0 \\ f_{p[i_k]}^{k-1}(i_{1:k-1}) & \text{if } l = k > 0 \\ p & \text{if } l = k = 0. \end{cases} \quad \square$$

This reduction is effective when the function to be encoded is sparse, i.e., it tends to be $\mathbf{0}$ when some variable has value 1, as this corresponds to a high-zero QBDD node.

Figure 1 illustrates a specific function, $\overline{x_3}x_2x_1 \vee x_3\overline{x_2x_1}$, encoded as a QBDD, FBDD, or ZBDD. In the following, we might omit the "Q", "F", or "Z" reduction indicator when we mean that what is being said holds true regardless of the specific version of BDD.

We observe that *multivalued DDs* (MDDs) [24] have also been defined, to encode and manipulate functions of the form $f : \mathcal{X}_1 \times \cdots \times \mathcal{X}_L \to \mathbb{B}$, where each $\mathcal{X}_k$ is a finite set, which can be assumed to be $\{0, ..., n_k\}$ for some $n_k \in \mathbb{N}$. This extension is straightforward (essentially we allow a multi-way choice at each node of the diagram, instead of just a binary choice), and can be simulated using multiple boolean variables to encode each multivalued variable ($n_k + 1$ for a *one-hot* encoding, or $\lceil \log_2(n_k+1) \rceil$ for a *binary* encoding), thus we consider it only occasionally in the remainder of the paper, and stress that, for most BDD/MDD variants, we can exchange "B" with "M" and modify the definition accordingly.

B. Canonicity and efficiency

A fundamental property enjoyed by almost all decision diagrams used in practical applications, and by all the ones we consider, is that each function that can be encoded by a decision diagram variant has a *unique* representation in it, once we decide a *variable order* (i.e., a mapping of the L input boolean variables to the L levels). If all nodes are stored in a *forest* using a *unique table* to avoid duplicates, then function equality can be decided in $O(1)$ time, since functions are equal iff they are encoded by the same node (or edge, depending on the decision diagram variant).

This *canonicity* property has profound implications not only for memory efficiency, since multiple nodes encoding the same function are never stored, but also for time efficiency. This is because any binary operation on decision diagrams recurses down by levels and, thanks to the use of a *cache*, computing $f = f' \odot f''$, where f' and f'' are encoded as decision diagram nodes and $\odot$ is an elementwise operation (such as $\wedge$ or $\vee$ in the boolean case), requires $O(||f'|| \cdot ||f''||)$ time, where $||g||$ is the number of nodes in the decision diagram encoding function g. Canonicity ensures that $||g||$ is minimal (for the given variable order and the particular decision diagram variant), since there are no duplicate nodes.

III. Improving Binary Decision Diagrams

Each of the three BDD variants we recalled has distinct advantages: QBDDs have smaller nodes and simpler algorithms, FBDDs use fewer nodes when encoding functions with many don't cares, and ZBDDs use fewer nodes when encoding functions that mostly evaluate to 0 if variables have value 1. In practice, researchers rarely choose QBDDs, assuming that the node savings of FBDDs or ZBDDs more than offset the need for slightly larger nodes and more complex algorithms (although we discuss in Section VI how, in certain classes of applications, QBDDs and FBDDs always coincide). However, having to choose between FBDDs and ZBDDs is unsatisfying for two main reasons. First, users may not have a good a priori intuition of which one will work better for their specific application. Second, and most importantly, just because the number of (redundant) nodes removed by using FBDDs may be greater than the number of (high-zero) nodes removed by using ZBDDs, this does not mean that we should not try to remove *both* kind of nodes.

Two recent approaches achieved this goal. In 2017, Van Dijk introduced *tagged BDDs* (TBDDs) [37], where either FBDD reductions are implied from the source node to a *tag* level associated to an edge, then ZBDD reductions are implied below that, or ZBDD reductions are implied first and FBDD reductions second. In 2018, Bryant introduced *chain-reduced FBDDs* (CFBDDs) and *chain-reduced ZBDDs* (CZBDDs) [10], which respectively allow FBDD nodes to encode chains of high-zero nodes, or ZBDD nodes to encode chains of redundant nodes. In the CFBDD variant, each node specifies two levels, the first one indicating the level at which a chain of high-zero nodes would have started (analogous to the level of an ordinary BDD node), the second one indicating where the chain would have ended, so that edges leaving a CFBDD node have again the fully-reduced interpretation. In the CZBDD variant, the meaning of edges is instead as in ZBDDs, and nodes encode chains of redundant nodes.

However, both tagged and chained proposals require larger nodes (compared to FBDDs and ZBDDs, Van Dijk's nodes store three levels instead of one, Bryant's nodes store two levels instead of one). More importantly, the resulting BDD size depends on the choice (FBDD-first vs. ZBDD-first for tagged, CFBDDs vs. CZBDDs for chained), and choosing between them is arguably even harder for a user than choosing between FBDDs and ZBDDs.

We have more recently addressed this problem in a more efficient and general way by explicitly attaching a "reduction rule" to each edge, to specify how to interpret skipped levels.

We did so first in 2019 with the *edge-specified-reduction* BDDs (ESRBDDs) [6], which allow one of three reductions: an edge with reduction $\mathtt{X}$ means that the skipped variables are don't cares along that path, it arises from eliminating a series of redundant nodes (FBDD semantic); an edge with reduction $\mathtt{H}_0$ means that the function has value 0 if any of the skipped variables has value 1, it arises from eliminating a series of high-zero nodes (ZBDD semantic); finally, an edge with reduction $\mathtt{L}_0$ means that the function has value 0 if any of the skipped variables has value 0 (this is a natural alternative to the $\mathtt{H}_0$ reduction, to eliminate *low-zero* nodes, thus is effective when encoding functions that mostly evaluate to 0 if variables have value 0). As short edges do not need a reduction, we say they have reduction $\mathtt{N}$, which is really a shorthand for the set $\{\mathtt{X}, \mathtt{H}_0, \mathtt{L}_0\}$, since these three reductions are equivalent for short edges. In an implementation, ESRBDDs require only four additional bits per node, yet allow more and more flexible reductions than either tagged or chained BDDs. Theoretically, ESRBDDs never require more nodes than FBDDs or ZBDDs, never more than twice the nodes of CFBDDs or CZBDDs, and never more than thrice the nodes of TBDDs; on the other hand, all these other variants can require $O(L)$ times as many nodes as ESRBDDs. Experimentally, ESRBDDs were always either the smallest (often by a factor of 2) or the second smallest (and less than 1% larger than the smallest) among FBDDs, ZBDDs, tagged BDDs, or chained BDDs, on a variety of benchmarks: encoding dictionaries of English words or passwords, combinatorial circuits, or the state space of *safe* (1-bounded) Petri net models taken from the MCC competition [1]. Experiments indicate that ESRBDDs have lower computation times as well, confirming that reducing BDD sizes leads to savings in both memory and time.

Then, in 2022, we extended ESRBDDs by adding (output) *complement flags* and further reduction options, resulting in the CESRBDDs [5]. The additional reductions are $\mathtt{L}_1$ and $\mathtt{H}_1$, which are the complement of $\mathtt{L}_0$ and $\mathtt{H}_0$, in the sense that they indicate that the value of the function is 1 (instead of 0) if any of the skipped variables are low or high, respectively. These additional reductions are not only naturally inspired by symmetry considerations, but are also required to achieve memory and time efficient BDD operations in conjunction with the use of complement flags. A complement flag set to 1 associated with an edge pointing to node q indicates that the function f_q should be complemented. This avoids the need to store node $\overline{q}$ encoding the complement $\overline{f_q}$ of a function f_q, if node q is already present, and potentially halves the number of required nodes. However, these savings can only be realized with canonicity and, while complement flags were introduced in 1978 [2], rules to achieve canonicity for QBDDs or FBDDs with complement flags were discovered only ten years later [25], [29]. One way to ensure canonicity is to require a complement flag set to 0 on 0-edges (thus complement flags require only one additional bit per node, to store the complement flag of the 1-edge, and $f_q(0, ..., 0) = 0$ for any node q), and to eliminate terminal node $\mathbf{1}$ (edges to terminal $\mathbf{1}$ become complemented edges to terminal $\mathbf{0}$).

In addition to reducing the number of nodes (thus memory requirements), complemented edges also have the potential to decrease the runtime requirements, both because fewer nodes mean more potential cache hits, and because computing the BDD encoding $\overline{f}$ can be done in $O(1)$ time (by simply flipping a complement flag) instead of in $O(||f||)$ time (by traversing the BDD and building a second BDD exactly equal to the one for f, except that the terminals $\mathbf{0}$ and $\mathbf{1}$ are exchanged).

Since CESRBDDs appear to be even more competitive (from slightly better to much better) than ESRBDDs and all other BDD variants discussed so far, on all the benchmarks we tried [5], we are currently investigating and implementing further reduction possibilities, ensuring that canonicty is still preserved (this was already a major challenge to overcome for CESRBDDs). Furthermore, we are also exploring theoretical comparisons between BDD variants, such as what is the worst case or average case size for each BDD variant when encoding a function of L variables. This question can be tackled by a brute approach only up to $L = 5$, since already for $L = 6$, the number of possible functions to consider, $2^{2^6} = 2^{64} \approx 1.8 \times 10^{19}$, is too large to even just enumerate them; thus, we are investigating analytical approaches from first principles to answer this question for large values of L.

IV. Encoding Non-Boolean Functions

BDDs can be easily extended to encode non-boolean valued functions of the form $\mathbb{B}^L \to \mathcal{S}$ by allowing an arbitrary set $\mathcal{S}$ of terminal nodes instead of just $\{\mathbf{0}, \mathbf{1}\}$. The *multi-terminal BDDs* [22] are then a natural extension of BDDs and can be defined analogously to the QBDDs, resulting in MTQBDDs, where edges never skip levels, or in MTFBDDs, where redundant nodes are eliminated; in either case, the terminal nodes are the elements of $\mathcal{S}$, at level 0. Indeed, there is no restriction on $\mathcal{S}$, thus it can even be infinite (e.g., $\mathbb{N}$ or $\mathbb{R}$), since only a finite set of at most 2^L elements from $\mathcal{S}$ can be reached from any given root (i.e., any function of L boolean variables can at most have 2^L different valuations). We observe that an analogous extension of ZBDDs to encode multivalued functions is also in principle possible, but, to the best of our knowledge, it has not been proposed, probably because the type of applications where this extension would be needed do not lend themselves to an efficient "zero-suppressed" encoding.

However, a multi-terminal approach is not necessarily the most efficient in practice; in fact, it rarely is. An alternative approach consists then in attaching values to the edges of the BDD, so that the encoded function is evaluated on input $(i_1, ..., i_L)$ by *summing* the values encountered along the edges, including the value σ on the *root edge*, until reaching the only terminal node, Ω, which carries no value. These are the EVBDDs of [27], which can encode any function $\mathbb{B}^L \to \mathbb{Z}$, and achieve canonicity by requiring that the value associated to the 0-edge of each nonterminal node be 0; this in turn implies $f(0, ..., 0) = \sigma$, and requires storing only the value associated to the 1-edge of each terminal node, which can be any value in $\mathbb{Z}$. As originally defined, EVBDDs forbid redundant nodes

(which have both outgoing edges pointing to the same node with value 0), so they should be called EVFBDDs, while the term EVQBDDs should be used to indicate the EVBDD version that retains all redundant nodes.

An important application that requires storing and computing over integer functions is the building the *distance function* (of each reachable state from an initial state $\mathbf{i}_{init}$) in system analysis or model checking: $\delta(i_1,...,i_L)$ = "the minimum number of steps required to go from $\mathbf{i}_{init}$ to $\mathbf{i} = (i_1,...,i_L)$". However, δ is normally a *partial* function, since the distance of any unreachable state is undefined, which is algorithmically equivalent to being (positive) ∞, and this reveals an inherent limitation of the original EVBDD definition: canonicity requires that the path corresponding to $i_k = 0,...,i_1 = 0$ from a nonterminal node at level k to Ω be labeled with 0 values, so no state with $i_k = 0,...,i_1 = 0$ can be unreachable. In other words, EVBDDs can encode all functions of the form $\mathbb{B}^L \to \mathbb{Z}$, but only some functions of the form $\mathbb{B}^L \to \mathbb{Z} \cup \{\infty\}$, or even just $\mathbb{B}^L \to \mathbb{N} \cup \{\infty\}$, which would suffice because distances are non-negative.

We then defined an alternative form of EVBDDs, called EV$^+$BDDs (actually, EV$^+$MDDs [19], as we assume non-boolean domains, but that is a separate extension), which can encode any partial function of the form $\mathbb{B}^L \to \mathbb{Z} \cup \{\infty\}$ by imposing different canonicity rules: the values associated to the edges leaving a node are all non-negative (thus only the root edge value σ can be negative), at least one of them must be 0 (thus σ equals the minimum value of the function), and edges with value ∞ must point to Ω (since, once we add ∞ to the sum of the values seen so far, the value of any further edge becomes irrelevant). Again, both quasi-reduced, EV$^+$QBDDs, or fully-reduced, EV$^+$FBDDs, versions can be defined.

It can be shown that the number of nodes in an EV$^+$BDD can never be larger than that of the equivalent MTBDD, and it is often smaller (up to exponentially so), even when encoding partial functions. However, storing edge values requires slightly more memory: since one of the two edge values must be zero, it is enough to store one bit plus one integer per node, thus just one more bit than EVBDDs (in practical implementations, the value ∞ is best encoded by pointing to a special "∞-node" with an edge value of 0, rather than by pointing to Ω with an "∞-edge-value").

A third approach to encode non-boolean functions has not been explored much in the literature, but should be mentioned: given $f : \mathbb{B}^L \to \mathcal{S}$, we can store the collection of disjoint sets

$$\{f^{[a]} : a \in \mathcal{S}, f^{[a]} = \{(i_1,...,i_L) : f(i_1,...,i_L) = a\} \neq \emptyset\}$$

using a "forest" of BDDs, with as many roots as the number of different values the function actually takes for at least one tuple in the domain. Intuitively, this looks like "flipping the MTBDD on its head", in the sense that now there are as many BDD roots as there are reachable MTBDD terminals. Surprisingly, the overall size of the BDD forest is not comparable to that of the MTBDD encoding the same information, meaning that we have been able to show that either approach may be best (i.e., require fewer nodes), depending on the function being encoded. We suspect that the same holds even considering EV$^+$BDDs instead of MTBDDs. One disadvantage of a BDD forest is that evaluating $f(i_1,...,i_L)$ requires evaluating each $f^{[a]}(i_1,...,i_L)$ sequentially in some order, until we find the one that evaluates to 1 (of course, a parallel evaluation is also possible, if the hardware is available and the goal is to minimize evaluation time, not total CPU time).

A fundamental question is then how to characterize a priori what will be the best encoding among MTBDDs, EVBDDs, EV$^+$BDDs, and forests of BDDs, for certain classes of functions, either in terms of nodes or, better, in terms of memory (so that MTBDDs, which employ smaller nodes, have a fighting change against EVBDDs and EV$^+$BDDs, which can never have more nodes).

V. Encoding Relations and Matrices

Model checking is an important application where the use of BDDs has been quite successful, allowing verification of properties on discrete-state, discrete-event models with huge numbers of states [13]. A BDD can be used to encode a set of states $\mathcal{X} \subseteq \{0,1\}^L$, by encoding the boolean characteristic function of the set $f_{\mathcal{X}}$, where $f_{\mathcal{X}}(i_1,...,i_L) = 1$ iff $(i_1,...,i_L) \in \mathcal{X}$. The behavior of the model can be captured by a transition relation, as a subset of all pairs of states. Again, this can be encoded as a characteristic function, this time over $2L$ variables, where $f_{\mathcal{R}}(i_1,...,i_L,i'_1,...,i'_L)$ evaluates to one iff the model can change from state $(i_1,...,i_L)$ to state $(i'_1,...,i'_L)$ due to some event. For most practical models, an interleaved variable order, e.g., $(i_1,i'_1,...,i_L,i'_L)$, leads to a much smaller BDD than the state pair order $(i_1,...,i_L,i'_1,...,i'_L)$. Intuitively, this is because the boolean variables correspond to model state variables, and the new value of a state variable after an event occurs tends to depend strongly on its previous value.

Even using the interleaving heuristic, building a single, monolithic transition relation for a model can be time consuming and produce a large BDD. Some research efforts over the years have addressed this problem. The transition relation is typically *partitioned* into several relations, where the overall relation is the disjunction or conjunction of the smaller relations [11]. For asynchronous systems, a natural partitioning is to use a relation for each model event; if the choice of which event should occur among the set of enabled model events is nondeterministic (or probabilistic), the overall relation is the disjunction of the relations for each event.

In many types of models, an event typically affects, or depends on, only a few of the state variables. If a state variable x_k is independent of an event, then the next state will never be affected by x_k, the enabling of the event will not depend on x_k, and x_k will remain unchanged when the event occurs. Formally we have $f_e(i_1,i'_1,...,i_k,i_k,...,i_L,i'_L) = f_e(i_1,i'_1,...,\overline{i_k},\overline{i_k},...,i_L,i'_L)$ and $f_e(i_1,i'_1,...,i_k,\overline{i_k},...,i_L,i'_L) = 0$ where f_e is the characteristic function for the transition relation of event e. This leads to a distinct *identity pattern* in the BDD, illustrated in Fig. 2(a) for an event that is independent of variable x_3. If unprimed and primed levels are merged together into a *matrix*

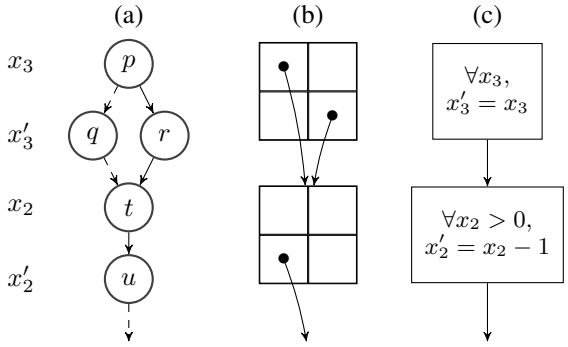

Fig. 2. Relations encoded using (a) BDDs, (b) MxDs, and (c) implicit nodes (edges to **0** are omitted).

diagram (MxD) node [31] (2×2 for BDDs, $n \times n$ for MDDs), then the identity pattern corresponds to an identity matrix: all edges along the diagonal are to the same node, and all non-diagonal edges are to terminal node **0**, see Fig. 2(b). Matrix diagrams thus use an *identity reduction*, where identity pattern nodes are eliminated instead of redundant nodes, which would correspond to a full matrix (state variable i_k can change to any value) and are unlikely to occur in most models. In practice, though, an L-level matrix diagram can be implemented as a $2L$-level BDD (or MDD) using a special *fully-identity* reduction rule [20], where unprimed levels eliminate redundant nodes, and primed levels eliminate "singleton nodes" whose children are all to **0** except for a single v-child, but only when the node itself is the v-child of a node at the the unprimed level immediately above. For example, in Fig. 2(a), nodes q and r are singleton nodes, where q's 0-child, and r's 1-child, are nonzero. Since q is the 0-child of p, and r is the 1-child of p, nodes q and r would be eliminated, causing updated the p to become redundant (since both of its edges now point to t); node p, belonging to an unprimed level, would then be in turn eliminated. The use of identity reductions lead to more compact transition relation encodings, and can be exploited by algorithms for state space generation and model checking, in particular *saturation* [15], [14], [21], which has been shown to be often many orders of magnitude better in terms of both memory and runtime than simpler breadth-first symbolic iterations when analyzing discrete-state asynchronous systems.

The idea of identity patterns can be extended to other common patterns, for example $i'_k = i_k + c$ for a constant c. This is more relevant for the case of non-binary state variables and for generating reachable states encoded as MDDs. The use of *implicit nodes* [7] allows for small, fixed-size relations, compatible with BDD and MDD, even when the model state variables are unbounded (or, more likely, are bounded but the bounds are not known). Each implicit node is associated with a single state variable, and contains enabling and update expressions in terms of that state variable (see Fig. 2(c)).

For *stochastic* model checking, instead of a transition relation we need to encode a stochastic process, typically a continuous-time Markov chain (CTMC). This can be viewed as changing the encoded function f from a characteristic function of the form $\mathbb{B}^{2L} \to \{0,1\}$ to a real-valued function of the form $\mathbb{B}^{2L} \to \mathbb{R}$, where f returns the non-negative transition *rate* from one state to another in the CTMC. Similar to the previous discussion in Section IV, this can be done using several real-valued terminal nodes (i.e., using MTBDDs or MTMDDs), or by attaching real values to the edges. However, unlike EVBDDs and EV$^+$BDDs, which sum integer edge values along a path, for encoding the rate matrix of a CTMC it is better to *multiply* real edge values along a path, in part because the identity reduction can still be utilized. This gives rise to matrix diagrams with edge values that are multiplied along a path (e.g., [17]), which again can be implemented as (EV*BDDs or) EV*MDDs [18].

Of course, storing large CTMCs compactly using EV*BDD or EV*MDDs is only the beginning, as one normally wants to compute the transient or steady-state probabilities for the CTMC; this is conceptually a vector π encoding a function of the form $\mathbb{B}^L \to [0,1]$. In general, π does not have a particularly compact encoding as an EV*MDD, but we have been able to show that it does, under certain *product-form* conditions, leading to exact fully-EV*MDD-based extremely efficient solution of enormous CTMCs when these conditions hold [32], [38]. More interestingly, the same fully-EV*MDD-based approach can be used when the product-form conditions "don't quite hold", resulting in a fixpoint style approximation, but more research is needed toward quantifying (bounding) the approximation error in this case.

VI. Variable Order Heuristics

It is well known that the variable order, i.e., the mapping of the L domain variables to the L levels of a BDD, can have a dramatic impact on the size of the encoding for given function: A few "good" functions may have a linear (or low-degree polynomial) size encoding for any variable order; most functions are "bad" and have an exponential size encoding for any variable order; however, many functions of interest may have an encoding size that ranges from linear to exponential depending on the variable order, or at least polynomial with a degree that depends on the variable order.

It is also well-known that finding an optimal variable order is NP-hard [8], thus much research effort has been spent on fast variable-order heuristics that aim at "good" orders rather than optimal orders that minimize the BDD size. These heuristics can be classified as *static* or *dynamic*, depending on whether they propose a variable order prior to building any decision diagram for the problem at hand (thus they attempt to exploit structure in the high-level description of the problem), or whether they periodically attempt to improve the current variable order based on the current structure of the BDD. Of course, it is possible to employ both a static heuristic that attempts to start with a good order, and then a dynamic heuristic to further improve the order as the BDDs are being manipulated. In practical applications, several BDDs are be stored in a single BDD forest, all with the same variable order; this makes the concept of optimal variable order even more

elusive, as different orders could be optimal at different stages of the computation.

The most commonly used dynamic variable-order heuristic is *sifting* [34], where each of the L variables is sequentially moved to each of the other $L-1$ positions, keeping track of the resulting BDD size. This is usually achieved by swapping a pair of adjacent variables at a time, an operation requiring only time proportional to the number of nodes at the two levels being swapped, even for FBDDs, which may have edges skipping over either or both levels. Of course, this approach is a greedy heuristic that overall explores only $\mathcal{O}(L^2)$ of the possible $L!$ orders, so it is not guaranteed to find a global optimum. Furthermore, the cost of each sifting attempt is still quite high, so that it often happens that, while peak memory consumption may be lowered, the overall runtime may substantially worsen compared to not using sifting altogether.

One of the advantages of sifting (or any other dynamic variable-order heuristic) is that it is application-agnostic: it only operates on the BDD without needing to know the meaning of its variables, but this is also its greatest weakness. Static variable-order heuristics offer instead the possibility of finding a good order in a more informed way, by considering how the variables describing the problem at hand are interrelated. For example, static heuristics for VLSI combinatorial circuits attempt to keep variables (corresponding to inputs and gate outputs) feeding to the same gate close to each other [3]; static heuristics for discrete-event models such as Petri nets [33] attempt to keep variables (Petri net places) that are input or output for the same event (Petri net transition) close to each other [16], [36], and so on. Of course, not all variables that should be close to each other can be, because each input may be connected to many gates of the circuit, or each place may be connected to several transition in the Petri net. This, too, becomes an NP-hard problem (related to finding a minimum sum-of-column-spans in a boolean matrix through row reordering), but it applies to a much more compact input (a circuit or a Petri net, instead of the large BDD encoding their possible states): relocating variables in a tentative order for a circuit or a Petri net is a much faster operation than swapping variables in a BDD.

Our recently proposed $\mathrm{i}_{\mathrm{Rank}}$ metric [4] performs very well to guide a simulated annealing search for a good order of the places of a Petri net subject to a set of linear invariants, resulting in generally excellent static variable orders computed at relatively low cost. Nevertheless, this is of course a heuristic approach, so improving it by taking into account, for example, the *domains* of the (non-boolean) variables, i.e., the possible number of tokens in each place, instead of just considering their position in the order, is a challenge worth exploring.

Incidentally, this work on encoding sets of reachable states for Petri nets has led us to the discovery that, for this type of problem, the QBDD and the FBDD encoding coincide perfectly if all places are part of at least one linear invariant: the QBDD cannot contain redundant nodes, thus there is no reason to use FBDDs which require slightly larger nodes and slower algorithms, but offer no advantages for this application.

VII. Conclusions and Future Directions

It is indisputable that BDDs have enabled great progress in the practice of digital design, hardware and software verification, and combinatorial search and optimization in discrete systems. This paper surveyed the key ideas in BDDs and some successful research efforts that have extended their efficiency and applicability.

We can summarize the common features of *all* the BDD variants we considered as follows: they are directed, acyclic graphs arranged by levels; they are a canonical representation; duplicate nodes at a particular level are merged together into a single node; operations are typically recursive, and require computation time that depends (usually linearly) on the number of nodes in the graph. The various forms of BDDs attempt to reduce storage and computation costs for particular applications, by lowering the number of nodes required to encode functions seen in practice. So far, this has been achieved by two main approaches. The first is to assign different meanings to the long edges (e.g., QBDDs vs. FBDDs vs. ZBDDs), so that node patterns that hopefully appear often can be eliminated entirely. The second is to assign information to the graph edges to modify the function encoded by the target node in some way (e.g., a complement bit complements the function, an integer edge value in EVBDDs adds an integer constant to the function); this essentially allows a single node to encode a *set* of functions, by encoding one representative from the set. Thus, these variants attempt to make the graph smaller either by removing the need to store nodes entirely, or by increasing the amount of node sharing in the graph.

Potential for further BDD improvements can continue these efforts in at least three directions. The first direction is to explore other node patterns that can be eliminated entirely (e.g., FBDDs vs. ZBDDs vs. identity reduced BDDs). The second direction is to explore different edge attributes for increased node sharing. Two ideas that have already been proposed along these lines, but have not yet gained much traction, are *input complement* bits and *variable shifters*, both proposed in [35]. Each of these edge attributes modifies the input variable associated with the non-terminal node pointed by the edge; the first one complements the variable (i.e., exchanges the node's children), the second determines a node's *variable* based on summing edge attributes along the path in the BDD (e.g., allowing functions x_5x_3 and x_8x_6 to share the same node). The third direction is to combine two or more approaches (from the first or the second directions, or both) into a single forest simultaneously. The challenge in all of these, in particular the third direction, is to develop rules that maintain canonicity. Different BDD features, even those that admit canonicity by themselves, can interact in interesting and subtle ways that pose challenges to canonicity. As an example, combining FBDDs and ZBDDs together allows two different ways to represent the constant function $f = 0$: a long "FBDD" edge to terminal $\mathbf{0}$, or a long "ZBDD" edge to terminal $\mathbf{0}$. Thus, in ESRBDDs one of these edges must be disallowed, and it suffices to choose one arbitrarily (but consistently) [6]. As

another example, ZBDDs (which eliminate high-zero nodes) and complement flags cannot be combined efficiently except in BDD variants that are able to eliminate *both* high-zero and high-one nodes, because the complement of a high-zero node is a high-one node and vice versa [5].

Finally, we note that developing new variants of BDDs usually also requires developing new BDD algorithms to exploit the new BDD features. There are a number of large technological gaps here, the most notable one being static variable ordering heuristics: all current such heuristics are designed for FBDDs (we do not know of any heuristic that specifically targets characteristics of other BDD variants). Another important gap is the development of BDD libraries: while several FBDD libraries, and some ZBDD libraries, are widely available in the public domain, the more advanced BDD variants require complex algorithms, thus their implementations tend to lag behind the most recent research advancements; yet, industrial acceptance of advanced BDD technologies requires extensive support of high-quality well-documented libraries.

ACKNOWLEDGMENT

The work of the two authors was supported in part by the National Science Foundation under grant CCF-2212142.

REFERENCES

[1] MCC: the Model Checking Competition. https://mcc.lip6.fr.

[2] S. B. Akers. Functional testing using binary decision diagrams. In *Proc. 8th Int. Symp. on Fault-Tolerant Computing*, pages 75–82, 1978.

[3] F. Aloul, I. Markov, and K. Sakallah. Improving the efficiency of circuit-to-BDD conversion by gate and input ordering. In *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 64–69, 2002.

[4] E. Amparore, S. Donatelli, G. Ciardo, and A. Miner. i_{Rank}: a variable order metric for DEDS subject to linear invariants. In T. Vojnar and L. Zhang, editors, *Proc. TACAS*, LNCS 11428, pages 285–302, Prague, Czech Republic, Apr. 2019. Springer.

[5] J. Babar, G. Ciardo, and A. Miner. CESRBDDs: binary decision diagrams with complemented edges and edge-specified reductions. *Software Tools for Technology Transfer*, 24:89–109, Feb. 2022.

[6] J. Babar, C. Jiang, G. Ciardo, and A. Miner. Binary decision diagrams with edge-specified reductions. In *Proc. TACAS*, pages 303–318. Springer, 2019.

[7] S. Biswal and A. S. Miner. Improving saturation efficiency with implicit relations. In *Application and Theory of Petri Nets and Concurrency*, LNCS 11522, pages 301–320. Springer, 2019.

[8] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.

[10] R. E. Bryant. Chain reduction for binary and zero-suppressed decision diagrams. In *Proc. TACAS*, pages 81–98. Springer, 2018.

[11] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Int. Conference on Very Large Scale Integration*, pages 49–58. IFIP Transactions, North-Holland, Aug. 1991.

[12] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. CAD of Integr. Circ. and Syst.*, 13(4):401–424, Apr. 1994.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[14] G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.

[15] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, 2001.

[16] G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In *Proc. ATPN*, LNCS 4546, pages 83–103. Springer, 2007.

[17] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. PNPM*, pages 22–31. IEEE Comp. Soc. Press, 1999.

[18] G. Ciardo, A. S. Miner, M. Wan, and A. J. Yu. Approximating stationary measures of structured continuous-time Markov models using matrix diagrams. *ACM SIGMETRICS Perf. Eval. Rev.*, 35(3):16–18, 2007.

[19] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD*, LNCS 2517, pages 256–273. Springer, 2002.

[20] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, LNCS 3725, pages 146–161. Springer, 2005.

[21] G. Ciardo, Y. Zhao, and X. Jin. Ten years of saturation: a Petri net perspective. *Trans. Petri Nets and Other Models of Concurrency*, V:51–95, 2012.

[22] E. Clarke, M. Fujita, P. C. McGeer, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *Proc. Intl. Workshop on Logic Synthesis*, May 1993.

[23] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Proc. CAV*, LNCS 1102, pages 419–422, 1996.

[24] 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.

[25] K. Karplus. Representing boolean functions with if-then-else DAGs. Technical report, University of California at Santa Cruz, 1988.

[26] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. ICCD*, pages 220–223. IEEE CS Press, 1990.

[27] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. DAC*, pages 608–613. IEEE Comp. Soc. Press, June 1992.

[28] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Techn. J.*, 38(4):985–999, July 1959.

[29] J.-C. Madre and J.-P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proc. ACM/IEEE DAC*, pages 205–210. IEEE CS Press, 1988.

[30] S. Minato. Zero-suppressed BDDs and their applications. *Software Tools for Technology Transfer*, 3:156–170, 2001.

[31] A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Performance Evaluation*, 56(1):145 – 165, 2004. Dependable Systems and Networks - Performance and Dependability Symposium (DSN-PDS) 2002: Selected Papers.

[32] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. ACM SIGMETRICS*, pages 207–216. ACM Press, 2000.

[33] T. Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.

[34] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int. Conference on CAD*, pages 139–144, Nov. 1993.

[35] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proc. ACM/IEEE DAC*, pages 52–57. IEEE CS Press, 1990.

[36] B. Smith and G. Ciardo. SOUPS: a variable ordering metric for the saturation algorithm. In *Proc. 18th International Conference on Application of Concurrency to System Design (ACSD)*, pages 1–10. IEEE Comp. Soc. Press, 2018.

[37] T. van Dijk, R. Wille, and R. Meolic. Tagged BDDs: combining reduction rules from different decision diagram types. In *Proc. FMCAD*, pages 108–115, 2017.

[38] M. Wan, G. Ciardo, and A. S. Miner. Approximate steady-state analysis of large Markov models based on the structure of their decision diagram encoding. *Perf. Eval.*, 68:463–486, 2011.

[39] I. Wegener. BDDs—design, analysis, complexity, and applications. *Discrete Applied Mathematics*, 138(1):229–251, 2004.

[40] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In *Proc. FMCAD*, LNCS 1166, pages 435–449, 1996.