Binary Decision Diagrams: An Algorithmic Basis for Symbolic Model Checking

Randal E. Bryant[1]

Abstract Binary decision diagrams provide a data structure for representing and manipulating Boolean functions in symbolic form. They have been especially effective as the algorithmic basis for symbolic model checkers. A binary decision diagram represents a Boolean function as a directed acyclic graph, corresponding to a compressed form of decision tree. Most commonly, an ordering constraint is imposed among the occurrences of decision variables in the graph, yielding *ordered* binary decision diagrams (OBDD). Representing all functions as OBDDs with a common variable ordering has the advantages that (1) there is a unique, reduced representation of any function, (2) there is a simple algorithm to reduce any OBDD to the unique form for that function, and (3) there is an associated set of algorithms to implement a wide variety of operations on Boolean functions represented as OBDDs. Recent work in this area has focused on generalizations to represent larger classes of functions, as well on scaling implementations to handle larger and more complex problems.

1 Introduction

Ordered Binary Decision Diagrams (OBDDs) provide a symbolic representation of Boolean functions. They can serve as the underlying data structure to implement an abstract data type for creating, manipulating, and analyzing Boolean functions. OBDDs provide a uniform representation for operations to define simple functions and then construct representations of more complex functions via the operations of Boolean algebra, as well as function projection and composition. In the worst case, the OBDD representation of a function can be of size exponential in the number of

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
e-mail: Randy.Bryant@cs.cmu.edu

function variables, but, in practice, they remain of tractable size for many applications.

OBDDs have been especially effective as a data structure for supporting symbolic model checking, starting with the very first implementations of tools for symbolically checking the properties of finite-state systems [9, 19, 22, 45]. By encoding sets and relations as Boolean functions, the operations of model checking can be expressed as symbolic operations on Boolean functions, avoiding the need to explicitly enumerate any states or transitions.

In the spirit of viewing OBDDs as the basis for an abstract data type, we first define an Application Program Interface (API) for Boolean function manipulation, then the OBDD representation, and then how the API can be implemented with OBDDs. We describe some of the refinements commonly seen in OBDD implementations. We present some variants of OBDDs that have been devised to improve efficiency for some applications, as well as to extend the expressive power of OBDDs beyond Boolean functions. The many variants of OBDDs are sometimes referred to by the more general term *decision diagrams* (DDs). Many surveys of OBDDs and their generalizations have been published over the years [15, 16, 28]. Rather than providing a comprehensive survey, this chapter focuses on those aspects that are most relevant to model checking. We describe some efforts to increase the performance of OBDD programs, both to make them run faster and to enable them to handle larger and more complex applications. We conclude with a brief discussion on some relationships between OBDDs and Boolean satisfiability (SAT) solvers.

2 Terminology

Let $\mathbf{x}$ denote a vector of Boolean variables $x_1, x_2, \ldots, x_n$. We consider Boolean functions over these variables, which we write as $f(\mathbf{x})$ or simply f when the arguments are clear. Let $\mathbf{a}$ denote a vector of values $a_1, a_2, \ldots, a_n$, where each $a_i \in \{0,1\}$. Then we write the *valuation* of function f applied to $\mathbf{a}$ as $f(\mathbf{a})$. Note the distinction between a function and its valuation: $f(\mathbf{x})$ is a function, while $f(\mathbf{a})$ is either 0 or 1.

Let $\mathbf{1}$ denote the function that always yields 1, and $\mathbf{0}$ the function that always yields 0.

We can define Boolean operations $\wedge$, $\vee$, $\oplus$, and $\neg$ over functions as yielding functions according to the Boolean operations on the underlying elements. So, for example, $f \wedge g$ is a function h such that $h(\mathbf{a}) = f(\mathbf{a}) \wedge g(\mathbf{a})$ for all $\mathbf{a}$.

For function f, variable x_i and binary value $b \in \{0,1\}$, define a *restriction* of f as the function resulting when x_i is set to value b:

$$f|_{x_i \leftarrow b}(\mathbf{a}) = f(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n) \,.$$

The two restrictions of a function f with respect to a variable x_i are referred to as the *cofactors* of f with respect to x_i [11].

Given the two cofactors of f with respect to variable x_i, the function can be reconstructed as

$$f = \left(x_i \wedge f|_{x_i \leftarrow 1}\right) \vee \left(\neg x_i \wedge f|_{x_i \leftarrow 0}\right) . \tag{1}$$

This identity is commonly referred to as the *Shannon expansion* of f with respect to x_i, although it was originally recognized by Boole [12].

Other useful operations on functions can be defined in terms of the restriction operation and the algebraic operations $\wedge$, $\vee$, $\oplus$, and $\neg$. Let f and g be functions over variables $\mathbf{x}$. The *composition* of f and g with respect to variable x_i, denoted $f|_{x_i \leftarrow g}$, is defined as the result of evaluating f with variable x_i replaced by the evaluation of g:

$$f|_{x_i \leftarrow g}(\mathbf{a}) = f(a_1, \ldots, a_{i-1}, g(a_1, \ldots, a_n), a_{i+1}, \ldots, a_n) .$$

Composition can be expressed based on a variant of the Shannon expansion:

$$f|_{x_i \leftarrow g} = \left(g \wedge f|_{x_i \leftarrow 1}\right) \vee \left(\neg g \wedge f|_{x_i \leftarrow 0}\right) . \tag{2}$$

Another class of operations involves eliminating one or more variables from a function via quantification. That is, we can define the operations $\forall x_i.f$ and $\exists x_i.f$ as:

$$\forall x_i.f = f|_{x_i \leftarrow 1} \wedge f|_{x_i \leftarrow 0} \tag{3}$$

$$\exists x_i.f = f|_{x_i \leftarrow 1} \vee f|_{x_i \leftarrow 0} . \tag{4}$$

By way of reference, the resolution step of the original Davis–Putnam (DP) Boolean satisfiability algorithm [25] can be seen to implement existential quantification for a function represented in clausal form. Their method is based on the principle that function f is satisfiable (i.e., $f(\mathbf{a}) = 1$ for some $\mathbf{a}$) if and only if $\exists x_i.f$ is satisfiable, for any variable x_i.

Quantification can be generalized to quantify over a set of variables $X \subseteq \{x_1, \ldots, x_n\}$. Existential quantification over a set of variables can defined recursively as

$$\exists \emptyset.f = f$$
$$\exists (x_i \cup X).f = \exists x_i.(\exists X.f) ,$$

and the extension for universal quantification is defined similarly.

The ability of OBDDs to support variable quantification operations with reasonable efficiency is especially important for model checking, giving them an important advantage over Boolean satisfiability solvers. While deciding whether or not an ordinary Boolean formula is satisfiable is NP-hard, doing so for a quantified Boolean formula is PSPACE-complete [32].

Finally, we define the *relational product* operation, defined for functions $f(\mathbf{x})$, $g(\mathbf{x})$, and variables $X \subseteq \{x_1, \ldots, x_n\}$ as $\exists X.(f \wedge g)$. As is discussed in Chapter 9, this

operation is of core importance in symbolic model checking as the method to project a set of possible system states either forward (image) or backward (preimage) in time. Hence, it merits a specialized algorithm, as will be described in Section 5.

3 A Boolean Function API

Operation	Result
Base functions	
CONST(b)	**1** (b = 1) or **0** (b = 0)
VAR(i)	x_i
Algebraic operations	
NOT(f)	$\neg f$
AND(f,g)	$f \wedge g$
OR(f,g)	$f \vee g$
XOR(f,g)	$f \oplus g$
Nonalgebraic operations	
RESTRICT(f,i,b)	$f\|_{x_i \leftarrow b}$
COMPOSE(f,i,g)	$f\|_{x_i \leftarrow g}$
EXISTS(f,I)	$\exists X_I.f$
FORALL(f,I)	$\forall X_I.f$
RELPROD(f,g,I)	$\exists X_I.(f \wedge g)$
Examining functions	
EQUAL(f,g)	$f = g$
EVAL($f,\mathbf{a}$)	$f(\mathbf{a})$
SATISFY(f)	some $\mathbf{a}$ such that $f(\mathbf{a}) = 1$
SATISFY-ALL(f)	$\{\mathbf{a} \mid f(\mathbf{a}) = 1\}$

Fig. 1 Basic operations for a Boolean function abstract data type

As a way of defining an abstract interface for an OBDD-based Boolean function manipulation package, Figure 1 lists a set of operations for creating and manipulating Boolean functions and for examining their properties. In this figure f and g represent Boolean functions (represented by OBDDs), i is a variable index between 1 and n, b is either 0 or 1, and $\mathbf{a}$ is a vector of n 0s and 1s. For a set of indices $I \subseteq \{1,\ldots,n\}$, X_I denotes the corresponding set of variables $\{x_i \mid i \in I\}$. This figure is divided into several sections according to the different classes of operations.

The base operations generate the constant functions and functions corresponding to the individual variables. The algebraic operations have functions as arguments and generate new functions as results according to the operations $\wedge$, $\vee$, $\oplus$, and $\neg$. The nonalgebraic operations also have functions as arguments and as results, but

they extend the functionality beyond those of Boolean algebra, implementing the operations of restriction, composition, quantification, and relational product.

Operations in the final set provide mechanisms to examine and test the properties of the generated Boolean functions. The EQUAL operation tests whether two functions are equivalent, yielding either true or false. As special cases, this operation can be used to test for tautology (compare to **1**) and (un)satisfiability (compare to **0**). The EVAL operation computes the value of a function for a specific set of argument values. For a satisfiable function, we can ask the program to generate some arbitrary satisfying solution (SATISFY) or have it enumerate all satisfying solutions (SATISFY-ALL.) The latter operation must be done with care, of course, since there can be as many as 2^n solutions.

The set of operations listed in Fig. 1 makes it possible to implement a wide variety of tasks involving the creation and manipulation of Boolean functions, including symbolic model checking. The overall strategy when working with OBDDs is to break a task down into a number of steps, where each step involves creating a new OBDD from previously computed ones. For example, a program can construct the OBDD representation of the function denoted by a Boolean expression by starting with functions representing the expression variables. It then evaluates each operation in the expression using the corresponding algebraic operation on OBDDs until obtaining the representation of the overall expression.

As an illustration, suppose we are given the Boolean expression

$$(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3)\,. \tag{5}$$

We can create an OBDD for the function f denoted by this expression using a sequence of API operations:

$$\begin{aligned} f_1 &= \textsc{Var}(1) \\ f_2 &= \textsc{Var}(2) \\ f_3 &= \textsc{Var}(3) \\ f_4 &= \textsc{And}(f_1, f_2) \\ f_5 &= \textsc{Not}(f_3) \\ f_6 &= \textsc{And}(f_4, f_5) \\ f_7 &= \textsc{Not}(f_1) \\ f_8 &= \textsc{And}(f_7, f_3) \\ f &= \textsc{Or}(f_6, f_8) \end{aligned}$$

Similarly, given a combinational logic circuit, we can generate OBDD representations of the primary output functions by starting with OBDD representations of the primary input variables and then stepping through the network in topological order. Each step involves generating the OBDD for the function at the output of a gate according to the gate input functions and the gate operation.

4 OBDD Representation

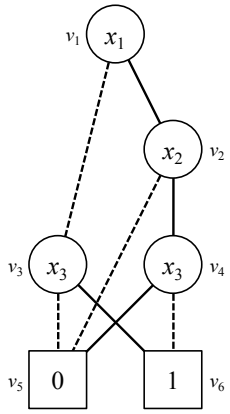

Fig. 2 OBDD representation of $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_3)$

A binary decision diagram (BDD) represents a Boolean function as an acyclic directed graph, with the nonterminal vertices labeled by Boolean variables and the leaf vertices labeled with the values 1 and 0 [1]. For nonterminal vertex v, its associated variable is denoted $var(v)$, while for leaf vertex v its associated value is denoted $val(v)$. Each nonterminal vertex v has two outgoing edges: $hi(v)$, corresponding to the case where its variable has value 1, and $lo(v)$, corresponding to the case where its variable has value 0. We refer to $hi(v)$ and $lo(v)$ as the *hi* and *lo children* of vertex v. The two leaves are referred to as the 1-*leaf* and the 0-*leaf*.

As an illustration, Fig. 2 shows a BDD representation of the function given by the expression in Eq. 5. In our figures, we show the arcs to the *lo* children as dashed lines and to the *hi* children as solid lines. To see the correspondence between the BDD and the Boolean expression, observe that there are only two paths from the root (vertex v_1) to the 1-leaf (vertex v_6): one through vertices v_2 and v_4, such that variables x_1, x_2, and x_3 have values 1, 1, and 0, and one through vertex v_3 such that variables x_1 and x_3 have values 0 and 1.

We can define the Boolean function represented by a BDD by associating a function f_v with each vertex v in the graph. For the two leaves, the associated values are **1** (1-leaf) and **0** (0-leaf). For nonterminal vertex v, the associated function is defined as

$$f_v = \Big(var(v) \wedge f_{hi(v)}\Big) \vee \Big(\neg var(v) \wedge f_{lo(v)}\Big) \; . \tag{6}$$

We see here the close relation between the BDD representation of a function and the Shannon expansion; the two children of a vertex correspond to its two cofactors with respect to its associated variable. Every vertex in a BDD represents a Boolean

function, but we designate one or more of these to be *root vertices*, representing Boolean functions that are referenced by the application program.

With *ordered* binary decision diagrams (OBDDs), we enforce an ordering rule on the variables associated with the graph vertices. For each vertex v having $var(v) = x_i$, and for vertex $u \in \{hi(v), lo(v)\}$ having $var(u) = x_j$, we must have $i < j$. For the rest of this chapter, we assume that all functions are represented as OBDDs with a common variable ordering. In the example BDD of Fig. 2, we see that the variable indices along all paths from the root to the leaves are in increasing order, and thus it is an OBDD.

We can define a *reduced* OBDD as one satisfying the rules:

1. There can be at most one leaf having a given value.
2. There can be no vertex v such that $hi(v) = lo(v)$.
3. There cannot be distinct nonterminal vertices u and v such that $var(u) = var(v)$, $hi(u) = hi(v)$, and $lo(u) = lo(v)$.

Given an arbitrary OBDD, we can convert it to reduced form by repeatedly applying transformations corresponding to these three rules:

1. If leaves u and v have $val(u) = val(v)$, then eliminate one of them and redirect all incoming edges to the other.
2. If vertex v has $lo(v) = hi(v)$, then eliminate vertex v and redirect all incoming edges to its child.
3. If vertices u and v have $var(u) = var(v)$, $hi(u) = hi(v)$, and $lo(u) = lo(v)$, then eliminate one of the vertices and redirect all incoming edges to the other one.

By working from the leaves upward, and by employing sparse labeling techniques, this reduction can be performed in time linear in the size of the original graph [58]. The example OBDD of Fig. 2 is, in fact, a reduced OBDD.

Bryant showed that reduced OBDDs serve as a *canonical form* for Boolean functions [13]. That is, for a given variable ordering, every Boolean function over these variables has a unique (up to isomorphism) representation as a reduced OBDD.

There are two different conventions for representing multiple functions as OBDDs. In a *split* representation, each function has a separate OBDD, and each graph has a single root. In a *shared* representation [48], the reduction rules are applied across the entire set of functions, and so the entire collection of functions is represented as a single OBDD having multiple roots. The shared representation not only reduces the space required to represent a set of functions, it has the property that two represented functions are equal if and only if they are represented by the same vertex in the OBDD. That is, there cannot be two distinct vertices u and v such that $f_u = f_v$. This is sometimes referred to as a *strong canonical form*.

5 Implementing OBDD Operations

As Fig. 1 indicates, an OBDD software package must implement a number of operations. For those having a Boolean function as an argument or result, we denote

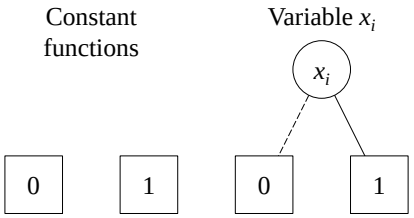

Fig. 3 OBDD representation for constant functions and variable x_i

this function by the root vertex in its OBDD representation. Thus, when describing OBDD algorithms, we define the operations in terms of vertex names, such as u and v, rather than abstract function names, such as f and g. As mentioned earlier, we can implement a set of Boolean functions as either a collection of separate OBDDs, each having a single root (split form), or as a single OBDD having multiple roots (shared form). In our presentation, we consider both approaches.

The base functions listed in Fig. 1 have simple representations as OBDDs, as shown in Fig. 3. Algorithms for the other operations follow a common framework based on depth-first traversals of the argument graphs. We present the *Apply* algorithm, a general method for implementing the binary Boolean algebraic operations, as an illustration.

The Apply algorithm has as arguments an operation *op* (equal to AND, OR, or XOR), as well as vertices u and v. The implementation, shown in Figure 4, performs a depth-first traversal of the two argument graphs and generates a reduced OBDD from the bottom up as it returns from the recursive calls. The algorithm makes use of two data structures storing keys and values. The *computed cache* stores results from previous invocations of the Apply operation, a process commonly referred to as *memoizing* [46]. Each invocation of Apply first checks this cache to determine whether a vertex corresponding to the given arguments has already been computed. As the name suggests, this data structure can be a cache where elements are evicted when space is needed, since the purpose of this data structure is purely to speed up the execution. The *unique table* contains an entry for every OBDD vertex, with a key encoding its variable and children. This table is used to ensure that it does not create any duplicate vertices. When using a split representation, this cache and table must be reinitialized for every invocation of Apply, while for a shared representation, the two data structures are maintained continuously.

Figure 5 shows cases where the recursive function implementing the Apply algorithm can terminate. As can be seen, the use of a shared representation enables additional terminal cases. In Section 6, we will discuss the use of *complement edges* to indicate the negation of a function. Their use enables even more terminal cases.

As the Apply algorithm illustrates, standard implementations of OBDD operations perform depth-first traversals of one or more argument graphs and generate reduced graphs as the recursions complete. The unique table is used to enforce reduction rules 1 and 3. The computed cache is used to stop the recursion when previously computed results are encountered. This cache can guarantee that the time

1. If the computed cache contains an entry with key $\langle op, u, v \rangle$, then return the associated value.
2. If one of the special cases shown in Fig. 5 applies, then return the specified value.
3. Recursively compute the two cofactors of the result as follows:
   a. Let $x_i = var(u)$, $x_j = var(v)$, and $k = \min(i, j)$.
   b. Compute u_1 and u_0 as $u_1 = hi(u)$ and $u_0 = lo(u)$ when $i = k$, and as $u_1 = u_0 = u$ when $i \neq k$.
   c. Compute v_1 and v_0 as $v_1 = hi(v)$ and $v_0 = lo(v)$ when $j = k$, and as $v_1 = v_0 = v$ when $j \neq k$.
   d. Compute $w_1 = \text{APPLY}(op, u_1, v_1)$ and $w_0 = \text{APPLY}(op, u_0, v_0)$.
4. Compute result vertex w:
   a. If $w_1 = w_0$, then $w = w_1$;
   b. else if the unique table contains an entry for key $\langle x_k, w_1, w_0 \rangle$, then let w be the associated value;
   c. else create a new vertex w with $var(w) = x_k$, $hi(w) = w_1$, and $lo(w) = w_0$. Add an entry to the unique table with key $\langle x_k, w_1, w_0 \rangle$ and value w.
5. Add an entry with key $\langle op, u, v \rangle$ and value w to the computed cache and return w.

Fig. 4 Recursive algorithm to compute $\text{APPLY}(op, u, v)$.

required by the algorithm is bounded by the number of unique argument combinations. If the top-level arguments to APPLY have N_u and N_v vertices, respectively, then the total number of calls to APPLY is at most $N_u \times N_v$. Typical implementations of the computed cache and the unique table use hashing techniques, which can yield constant average time for each access, and thus the overall time complexity of APPLY is $O(N_u \times N_v)$.

The other operations listed in Fig. 1 are implemented in a similar fashion. We give only brief descriptions here; more details can be found in [13]. The NOT operation proceeds by generating a copy of the argument OBDD, with the values of the leaf vertices inverted. To compute $\text{RESTRICT}(f, i, b)$, we want to eliminate every vertex v in the graph for f having $var(v) = x_i$ and redirect each incoming arc to either $hi(v)$ (when $b = 1$) or $lo(v)$ (when $b = 0$). Rather than modifying existing vertices, we create new ones as needed, applying the reduction rules in the process.

We have already seen that the composition, quantification, and relational product operations can be computed using combinations of restriction and Boolean algebraic operations (Eqs. 2–4.) However, these operations are of such critical importance in symbolic model checking and other applications that they are often implemented with more specialized routines. As with Apply, these algorithms use a combination of depth-first traversal, memoizing, and the unique table to generate a reduced graph as their result.

As an example, the algorithm to perform existential quantification can be expressed as a recursive function $\text{EXISTS}(u, I)$, where u is a BDD vertex and I is a set of variable indices. It maintains a *quantifier cache* using keys of the form $\langle u, I \rangle$. On a given invocation, if neither a previously computed result is found nor a terminal case applies, it retrieves u_1 and u_0, the two children of u, and recursively

Operation *op*	Condition	Restrictions	Result
AND	u, v are leaves		CONST($val(u) \wedge val(v)$)
AND	$u = \mathbf{0}$		CONST(0)
AND	$v = \mathbf{0}$		CONST(0)
AND	$u = \mathbf{1}$	S	v
AND	$v = \mathbf{1}$	S	u
AND	$u = v$	S	u
AND	$u =$ NOT(v)	S, C	CONST(0)
OR	u, v are leaves		CONST($val(u) \vee val(v)$)
OR	$u = \mathbf{1}$		CONST(1)
OR	$v = \mathbf{1}$		CONST(1)
OR	$u = \mathbf{0}$	S	v
OR	$v = \mathbf{0}$	S	u
OR	$u = v$	S	u
OR	$u =$ NOT(v)	S, C	CONST(1)
XOR	u, v are leaves		CONST($val(u) \oplus val(v)$)
XOR	$u = \mathbf{1}$		NOT(v)
XOR	$v = \mathbf{1}$		NOT(u)
XOR	$u = \mathbf{0}$	S	v
XOR	$v = \mathbf{0}$	S	u
XOR	$u = v$	S	CONST(0)
XOR	$u =$ NOT(v)	S, C	CONST(1)

S: Only with a shared representation
C: Only when complement edges are used

Fig. 5 Special cases for the Apply operation, with arguments *op*, u, and v.

computes $w_1 =$ EXISTS$(u_1, I - \{i\})$ and $w_0 =$ EXISTS$(u_0, I - \{i\})$. For $x_i = var(u)$, when $i \in I$, it computes the result as $w =$ APPLY(OR$, u_1, u_0)$. Otherwise, it either retrieves or creates a vertex w with $var(w) = x_i$, $hi(w) = w_1$, and $lo(w) = w_0$. Vertex w is then added to the quantifier cache with key $\langle u, I \rangle$. Universal quantification can be implemented similarly, or we can simply make use of DeMorgan's Laws to express universal quantification in terms of existential: $\forall X.f = \neg \exists X.(\neg f)$.

As mentioned earlier, the relational product operation implements $\exists X.(f \wedge g)$ for variables X, and functions f and g. In principle, this operation could proceed by first computing $f \wedge g$ and then existentially quantifying the variables in X. Experience has shown, however, that the graph representing $f \wedge g$ will often be of unmanageable size, even though the final result of the relational product is more tractable. By combining conjunction and quantification during a single traversal of the graphs for f and g, this problem of "intermediate explosion" can often be avoided. This algorithm is expressed by a recursive function RELPROD(f, g, I) that uses a combination of the rules we have seen in the implementations of APPLY and EXISTS [18, 63].

We are left with the operations that test or examine one or more functions. As already mentioned, when using a shared representation, testing for equality can be done by simply checking whether the argument vertices are the same. With a split representation, we can implement a simple traversal of the two graphs to test for isomorphism. To evaluate a function for a specified set of argument values, we follow a

path from the root to a leaf, at each step branching according to the value associated with the variable, with the leaf value serving as the result of the evaluation.

To find a single satisfying assignment for a function, we can search for a path from the root to the 1-leaf. This search does not require any backtracking, since, with the exception of arcs leading directly to the 0-leaf, each arc is part of a path leading to the 1-leaf. To find all satisfying solutions, we can perform a depth-first traversal of the graph to enumerate every path leading to the 1-leaf.

6 Implementation Techniques

Dozens of OBDD software packages have been created, displaying a variety of implementation strategies and features. Most implementations follow a set of principles described in a 1990 paper by Brace, Rudell, and Bryant (BRB) [10]. In an evaluation of many different packages for benchmarks arising from symbolic model checking problems [63], the best performance consistently came from packages very similar to the BRB package. Here we highlight some of its key features. In Sect. 10, we describe efforts to scale OBDD implementations to handle large graphs and to support parallel execution. An excellent discussion of implementation issues can be found in [59].

Most OBDD packages, including BRB, use a shared representation, with all functions represented by a single, multi-rooted graph [48]. Formally, we can define a shared OBDD as representing a set of functions $\mathscr{F}$, where each $f \in \mathscr{F}$ is designated by a root vertex in the graph. As we have seen, this approach has several advantages over a separate representation:

- it reduces the total number of vertices required to represent a set of functions,
- it simplifies the task of checking for equality, and
- it provides additional cases where the recursions for operations such as Apply and restriction can be terminated (see Fig. 5.)

On the other hand, using a shared representation introduces the need for some form of garbage collection to avoid having the available space exhausted by vertices that are no longer reachable from any of the active root vertices. Most shared OBDD implementations maintain a count of the total number of references to each vertex, including arcs from other vertices as well as external references to the root vertices. A vertex is a candidate for reclamation when its reference count drops to zero. Reclaiming a vertex also involves removing the corresponding entry from the unique table as well as every entry in the computed cache that references that vertex as part of its key or value.

The BRB package makes use of *complement edges*, where each edge has an additional attribute indicating whether or not the designated function is represented in true or complemented form. By adopting a set of conventions on the use of these attributes, it is possible to define a canonical form such that the NOT operation can be computed in constant time by simply inverting the attribute at the root [10, 42, 48].

By sharing the subgraphs for functions and their complements, such a representation can reduce the total number of vertices by as much as a factor of two. Perhaps more importantly, it makes it possible to perform the NOT operation in unit time. We can also see from Figure 5, that the combination of a shared representation and complement edges provides additional terminal cases for Apply and other operations.

The BRB package generalizes the two-operand Boolean operations to a single three-argument operation known as *ITE* (short for "If-Then-Else"), defined as:

$$ITE(f,g,h) = (f \wedge g) \vee (\neg f \wedge h) \ . \tag{7}$$

Using this single operation, we can implement other operations as:

$$\begin{aligned} \text{AND}(f,g) &= ITE(f,g,\mathbf{0}) \\ \text{OR}(f,g) &= ITE(f,\mathbf{1},g) \\ \text{XOR}(f,g) &= ITE(f,\text{NOT}(g),g) \\ \text{COMPOSE}(f,i,g) &= ITE(g,\text{RESTRICT}(f,i,1),\text{RESTRICT}(f,i,0)) \ . \end{aligned}$$

By rearranging and complementing the arguments according to a simple set of transformations, unifying the algebraic operations in this form can take advantage of DeMorgan's Laws to increase the hit rate of the computed cache [10]. This can dramatically improve overall performance, since each hit in the computed cache can potentially eliminate many recursive calls.

One feature of BRB and most other packages is that the individual node data structures are *immutable*. During program execution, new nodes are created, and ones that are no longer needed can be recycled via garbage collection, but the nodes themselves are not altered.[1] This functional programming model provides a useful abstraction for a Boolean function API, but it also implies that the package can expend much of its effort performing memory management tasks. New nodes must be created and old ones recycled, rather than simply letting the program modify existing nodes.

Several OBDD packages have been implemented that instead view the OBDD as a mutable data structure. For example, the SMART model checker [21] represents the set of states that have been encountered during a state-space exploration using a variant of OBDDs, called multiway decision diagrams (MDDs), that we describe in Sect. 9. As new states are encountered, the MDD is modified directly to include these states in its encoding. When performing model checking of asynchronous systems, as is the case with SMART, this approach seems appropriate, since each action of the system can be captured by a small change to the MDD.

[1] There is a nuance to this statement that we will discuss when we consider the implementation of dynamic variable reordering.

7 Variable Ordering and Reordering

The algorithms we have presented require that the variables along all paths for all represented functions follow a common ordering. Any variable ordering can be used, and so the question arises: "How should the variable ordering be chosen?" Some functions are very sensitive to variable ordering, ranging from linear to exponential in the number of variables. These include the functions for bit-level representations of integer addition and comparison. Others, including all symmetric functions, remain of polynomial size for all variable orderings [13]. Still others have exponential size for all possible variable orderings, including those for a bit-level representation of integer multiplication [14].

We can express the choice of variable ordering by considering the effect of permuting the variables in the OBDD representation of a function. That is, for Boolean function f and permutation π over $\{1,\ldots,n\}$, define $\pi(f)$ to be a function such that

$$\pi(f)(x_1,\ldots,x_n) = f(x_{\pi(1)},\ldots,x_{\pi(n)}) .$$

Different permutations π yield different OBDDs, but all of these can be viewed as just different representations of a single underlying function. The task of finding a good variable ordering for a function f can then be defined as one of finding a permutation π that minimizes the number of vertices in the OBDD representation of $\pi(f)$. For a shared OBDD representation, we wish to find a good variable ordering for the entire graph. That is, for permutation π and function set $\mathscr{F}$, define $\pi(\mathscr{F})$ to be $\{\pi(f) \mid f \in \mathscr{F}\}$. For a shared OBDD implementation, we seek a permutation π that minimizes the number of vertices in the OBDD representation of $\pi(\mathscr{F})$.

In general, the task of finding an optimal ordering π for a function f is NP-hard, even when f is given as an OBDD [8]. There is not even a polynomial-time algorithm that can guarantee finding a variable ordering within a constant factor of the optimum, unless $P = NP$ [57]. Similar results hold for a shared OBDD representation [61]. Published algorithms to find the exact optimal ordering have worst-case time complexity $O(n3^n)$ [29] for a function with n variables. Knuth has devised clever data structures that make the process practical for up to around $n = 25$ [38].

Instead of attempting to find the best possible ordering, a number of researchers have derived heuristic methods that have been found to generate reasonably good variable orders for specialized applications, such as when the Boolean function is derived from a combinational circuit [30, 43], a sequential circuit [36], a CNF representation [3], or a set of interacting state machines [6].

An alternate approach to finding a good variable ordering at the outset of the computation is to dynamically reorder the variables as the BDD operations proceed. This idea was introduced by Rudell [55], based on the observation that exchanging two adjacent variables in a shared OBDD representation can be implemented without making major changes to the Boolean function library API. Let π_i be the permutation that exchanges the values of i and $i+1$, while keeping all other elements the same. Exchanging variables i and $i+1$ in a shared OBDD representation involves converting the OBDD representation of function set $\mathscr{F}$ into one for function set

$\pi_i(\mathscr{F})$. This transformation can be implemented by introducing new vertices and relabeling and eliminating some of the existing vertices, but with the property that the identities of all root vertices are preserved. This is an important property, since external references to functions being manipulated by the application program consist of pointers to root vertices in the graph. Thus, the reordering can be performed without altering any of these external references. Even though the relabeling of vertices mutates the node data structures, these changes still preserve the "functional" property stated earlier—the underlying functions being represented do not change.

Using pairwise exchanges of adjacent variables as the basic operation, most OBDD libraries implement dynamic variable ordering by a process known as *sifting* [55]. A single variable, or a small set of variables [53], is moved up and down in the ordering via a sequence of adjacent-variable exchanges, until a location yielding an acceptable number of total vertices is identified. In the original formulation of sifting, a variable is moved across the entire range of possible positions and then back to the position that minimizes the overall OBDD size. More recently, lower bound techniques have been used to guide the range over which each variable is moved [27]. Sifting is a very time-consuming process, but it has been shown to greatly improve memory performance—often the limiting factor for OBDD-based applications.

8 Variant Representations

Researchers have examined many variants of OBDDs, both for representing Boolean functions and for extending to functions where the domain, the range, or both are non-Boolean. Here we survey some of the variants that have either proved effective for model checking or that seem especially promising. Some of these were also described in an earlier survey [16]. Other surveys provide even more comprehensive coverage of the many innovative variants of OBDDs that have been devised [28].

Zero-suppressed BDDs

Perhaps the most successful variant of OBDDs are *zero-suppressed* BDDs [47], sometimes referred to as ZDDs. This representation differs from traditional OBDDs only in the interpretation applied to the case where an arc skips one or more variables. That is, it concerns the case where there is an arc emanating from a vertex v with label $var(v) = x_i$ to a vertex u with label $var(u) = x_j$, such that $j > i+1$. In the example of Fig. 2 (reproduced on the left-hand side of Fig. 6), such an arc occurs from the root vertex v_1 to vertex v_3. With conventional OBDDs, such an arc indicates a case where the represented function is independent of any of the intervening variables. In the example, $f|_{x_1 \leftarrow 0}$ is independent of x_2. With a ZDD, such an arc indicates a case where the represented function is of the form $\neg x_{i+1} \wedge \cdots \wedge \neg x_{j-1} \wedge f_u$,

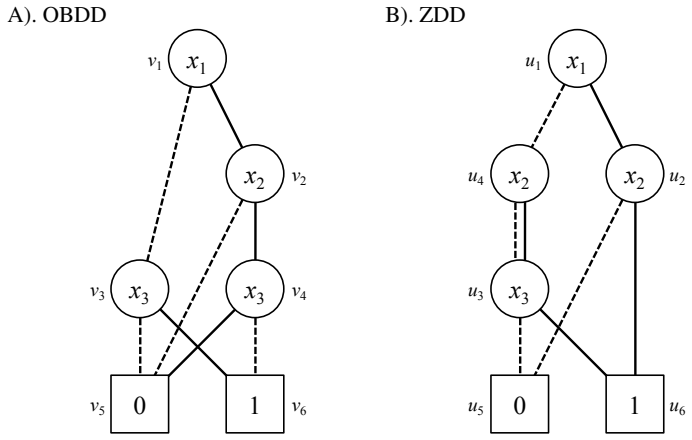

Fig. 6 BDD and ZDD representations of the set of sets $\{\{1,2\},\{3\},\{2,3\}\}$.

where f_u is the function associated with vertex u. For ZDDs, we replace the second reduction rule for OBDDs (that a vertex cannot have two identical children) with a rule that no vertex can have the 0-leaf as its *hi* child.

More formally, we can define the Boolean function denoted by a ZDD by defining a set of functions of the form f_v^j for each vertex v. For leaf vertex v, we define this set for all $j \leq n+1$ as follows:

$$f_v^j = \begin{cases} \mathbf{1}, & j = n+1 \text{ and } val(v) = 1 \\ \mathbf{0}, & j = n+1 \text{ and } val(v) = 0 \\ \neg x_j \wedge f_v^{j+1}, & j \leq n\,. \end{cases}$$

For nonterminal vertex v having $x_i = var(v)$ we define this set for all $j \leq i$ as:

$$f_v^j = \begin{cases} x_i \wedge f_{hi(v)}^{i+1} \vee \neg x_i \wedge f_{lo(v)}^{i+1}, & j = i \\ \neg x_j \wedge f_v^{j+1}, & j < i\,. \end{cases}$$

The function associated with root vertex v is then f_v^1.

Although ZDDs can be considered an alternate representation for Boolean functions, it is more useful to think of them as representing sets of sets. That is, let $M_n = \{1,\ldots,n\}$, and consider sets of sets of the form $S \subseteq \mathscr{P}(M_n)$. We can encode any set $A \subseteq M_n$, with a Boolean vector $\mathbf{a}$, where a_i equals 1 when $i \in A$, and equals 0 otherwise. The set of sets represented by Boolean function f consists of those sets A for which the corresponding Boolean vector yields $f(\mathbf{a}) = 1$. As examples, Fig. 6 shows both the OBDD (left) and the ZDD (right) representations of the set of sets $\{\{1,2\},\{3\},\{2,3\}\}$. The OBDD representation is identical to that of Fig. 2, because these are the only three satisfying assignments to Eq. 5. Comparing the ZDD, we see that, with two exceptions, each vertex v_i in the OBDD has a direct counterpart

u_i in the ZDD. The first exception is the introduction of new vertex u_4 having two identical children, since such vertices are no longer eliminated by our revised reduction rules. The second exception is that there is no counterpart to vertex v_4, since this vertex had the 0-leaf as its *hi* child.

ZDDs are especially well suited for representing sets of sparse sets, defined as having two general properties:

- The total number of sets is much smaller than 2^n.
- Most of the included sets have far fewer than n elements.

These conditions tend to give OBDD representations where many nonterminal vertices have the 0-leaf as their *hi* children, and these vertices are eliminated by using a ZDD representation.

The OBDD and ZDD representations of a function do not differ greatly in size. It can easily be shown that if these two representations have N_o and N_z vertices, respectively, then $N_z/n \leq N_o \leq n \times N_z$ [56]. Nonetheless, for complex functions and large values of n, the advantage of one representation over the other can be very significant.

ZDDs have proved especially effective for encoding combinatorial problems [38, 56]. They have been used in model checking for cases where the set of states have the sparseness properties we have listed, such as for Petri nets [65].

Partitioned OBDDs

The general principle of partitioned OBDDs is to divide the 2^n possible combinations of variable assignments into m different, nonoverlapping subsets, and then create a separate representation for a function over each subset.

More formally, define a set of *partitioning functions* as a set of functions $P = \langle p_1, \ldots, p_m \rangle$, such that $\bigvee_i p_i = \mathbf{1}$ and for each i and j such that $i \neq j$, we have $p_i \wedge p_j = \mathbf{0}$.

Each function f is then represented by a set of functions $\langle f_1, \ldots f_m \rangle$, where each f_i equals $f \wedge p_i$. It can readily be seen that the Boolean operations distribute over any partitioning. For example, for $h = f \vee g$, we have $h_i = f_i \vee g_i$ for each partition i. On the other hand, other operations, including restriction, quantification, and composition do not, in general, distribute over a partitioning.

Partitioning has been shown to be effective for applications where conventional, monolithic OBDDs would be too large to represent and manipulate. One approach is to allow different variable orderings for each partition [50]. This approach works well for applications where some small set of "control" variables determine important properties of how the remaining variables relate to one another. The different partitions then consist of all possible enumerations of these control variables.

As will be discussed later (Sect. 10), partitioning can also provide the basis for mapping an OBDD-based application onto multiple machines in a distributed computing environment.

9 Representing Non-Boolean Functions

Many systems for which we might wish to apply model checking involve state variables or parameters that are not Boolean. A number of schemes have been devised to represent such functions as decision diagrams, seeking to preserve the key properties of OBDDs: 1) they achieve compactness, mostly through the sharing of subgraphs, 2) key operations can be implemented via graph algorithms, and 3) properties of the represented functions can readily be tested. Here we describe some of the decision diagrams that have been used in model checking and related applications.

Functions over Discrete Domains

Consider the case where function variable x ranges over a finite set $D = \{d_0, \ldots, d_{K-1}\}$. There are several possible ways to represent a function over x as a decision diagram:

Binary encoding: Recode x in terms of Boolean variables $x_{k-1}, x_{k-2}, \ldots, x_0$, where $k = \lceil \log_2 K \rceil$. Each value d_i is encoded according to the binary representation of i. When K is not a power of 2, then we can either 1) add an additional constraint that any valid assignment to the Boolean variables must correspond to a binary value less than K, or 2) define multiple assignments to the Boolean variables to encode a single value from D. A binary encoding minimizes the number of Boolean variables required.

Unary encoding: Recode x in terms of Boolean variables $b_{K-1}, b_{K-2}, \ldots, b_0$, where value d_i is encoded by having $x_i = 1$ and all other values equal to zero. Except for very small values of K, this encoding would be impractical for OBDDs, but it works well for ZDDs.

Multiway branching: Generalize the OBDD data structure to *multivalued* decision diagrams [21, 37], where a vertex for a K-valued variable has an outgoing arc for each of its K children.

Indeed, all three of these approaches have been used successfully.

For representing functions over discrete domains having non-Boolean ranges, the most straightforward approach is to allow the leaves to have arbitrary values, leading to *multi-terminal* binary decision diagrams (MTBDDs) [31]. (These have also been called *algebraic* decision diagrams (ADDs) [7].) More precisely, for a function f mapping to some codomain R, define its *image* $Img(f)$ as those values $r \in R$ such that $r = f(\mathbf{a})$ for some argument value $\mathbf{a}$. Then the MTBDD representation of f has a leaf vertex for each value in $Img(f)$.

The set of operations on such functions depends on the types of functions being represented. Typically, they follow the same approach we saw with the algorithm for the Apply operation (Sect. 5)—they recursively traverse the argument graphs, stopping when either a terminal case is reached, or the arguments match those stored in a computed cache. For example, when R is either the set of reals or integers, such an approach can be used to perform algebraic operations such as addition or

multiplication over functions. It can also be used to generate a *predicate*, capturing some property of the function values. For example, for function f mapping to real values, let Z_f be the Boolean function that yields 1 for those arguments $\mathbf{a}$ for which $f(\mathbf{a}) = 0.0$, and 0 otherwise. We can generate an OBDD representation of Z_f by traversing the MTBDD representation of f, returning $\mathbf{1}$ when we encounter leaf value 0.0, $\mathbf{0}$ when we encounter a nonzero leaf value, and either generating a new vertex or retrieving one from the unique table for the nonterminal cases.

MTBDDs have been used for a variety of applications, encoding such values as data-dependent delays in transistor circuits [44], as well as transition probabilities in Markov chains [40]. Their biggest limitation is that the size of a function image can be quite large, possibly exponential in the number of function variables. Such a function will have many leaf vertices and therefore little sharing of subgraphs. This lack of sharing will reduce the advantage of decision diagrams over more direct encodings of the problem domain, both in the compactness of the representation and the speed of the operations on them. Successful applications of MTBDDs often avoid this "value explosion" by exploiting the modularity in the underlying system. For example, when performing model checking of stochastic systems, the transition probabilities for the different subsystems can be maintained as separate MTBDDs, rather than combined via a product construction [2].

Functions Over Unbounded Domains

When a function variable x ranges over an infinite domain D, we cannot simply encode its possible values with a set of binary values or add multiple branches to the vertices of a decision diagram. In some applications, however, we need only capture a bounded set of attributes of the state variables. In this section, we describe *Difference Decision Diagrams* (DDDs) [49] as an example of this approach. DDDs illustrate a general class of decision diagrams, where the decisions are based on predicates over some domain, rather than simple Boolean variables. We then discuss several variants and extensions of this representation.

The *Difference Decision Diagram* data structure was devised specifically for analyzing timed automata. As discussed in Chap. 27, a timed automaton operates over a discrete state space but also contains real-valued clocks that all proceed at the same rate, but they can be at different offsets with respect to one another [4]. Although the clock values can be unbounded, their behavior can be characterized during model checking in terms of a finite set of bounds on their differences. DDDs therefore express the values of the clocks in terms of a set of *difference constraints*, each of the form $x_i - x_j \leq c$ or $x_i - x_j < c$, where x_i and x_j are clock variables, and c is an integer or real value. In the spirit of OBDDs, DDDs also impose an ordering requirement over difference constraints, based on the indices i and j of the two variables, the comparison operator ($\leq$ vs. $<$), and the constant c.

Figure 7 show three examples of DDDs and serves to illustrate some subtle issues that arise when generalizing from a decision diagram where the decisions represent

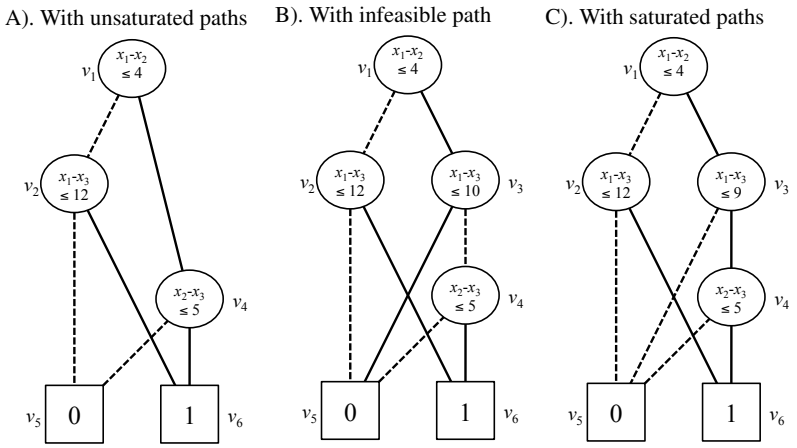

Fig. 7 Difference Decision Diagram (DDD) Examples. Vertices are labeled by difference constraints.

independent Boolean variables to one in which the decisions represent predicates over some other domain. The DDD on the left (A) represents a disjunction of two different constraints C_1 and C_2, defined as follows:

$$C_1 = (x_1 - x_2 > 4) \wedge (x_1 - x_3 \leq 12)$$
$$C_2 = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5)$$

The DDD in the center (B) also represents a disjunction of two constraints: C_1, as in (A), as well as a constraint C_2':

$$C_2' = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5) \wedge (x_1 - x_3 > 10).$$

On closer examination, however, we can see that constraint C_2' must be false for all values of x_1, x_2, and x_3. That is, if $x_1 - x_2 \leq 4$ and $x_2 - x_3 \leq 5$, then we must have $x_1 - x_3 \leq 9$, and this conflicts with the term $x_1 - x_3 > 10$. The possibility of infeasible paths implies that there is no simple way to determine whether a set of constraints represented as a DDD is satisfiable, whereas this is a trivial task with OBDDs. In particular, it is possible to determine whether any path in a DDD from the root to the 1-leaf is satisfiable in polynomial time, but there can be an exponential number of such paths.

The DDD on the right (C) represents a disjunction of constraint C_1, as before, and a constraint C_2'':

$$C_2'' = (x_1 - x_2 \leq 4) \wedge (x_2 - x_3 \leq 5) \wedge (x_1 - x_3 \leq 9).$$

We can see that constraint C_2'' is mathematically equivalent to C_2. As with C_2', the first and second terms of C_2'' already imply that the third term, $x_1 - x_3 \leq 9$, is redundant. In fact, constraint C_2'' is *saturated*, meaning that it contains a predicate for every pairwise constraint that can be inferred from it.

These examples show how the interdependencies between the predicates can lead to paths in a DDD that are infeasible, as well as ones where different combinations of terms can be mathematically equivalent. The developers of DDDs describe an algorithm that eliminates infeasible paths by testing each one individually and restructuring the DDD when an infeasible path is found [49]. In the worst case, this process can require time exponential in the size of the DDD, and it can also increase its size. Once infeasible paths have been eliminated, then satisfiability becomes easily testable. The developers also propose several rules for dealing with redundant tests, including ensuring that every path is saturated. This leads to a form that they conjecture is canonical, although this has apparently never been proven. Fortunately, most of the algorithms that use DDDs do not require having a canonical representation.

Several other decision diagrams have been devised specifically for model checking of timed automata. Clock difference diagrams [41] coalesce the predicates of difference decision diagrams, such that along any path there is a single node representing all constraints among a given pair of variables x_i and x_j. This node has multiple outgoing branches, corresponding to disjoint intervals representing possible values for $x_i - x_j$. Clock restriction diagrams [62] also have multiple branches emanating from a single node associated with variables x_i and x_j, but these represent possible upper bounds on the value of $x_i - x_j$. (Lower bounds on this value are represented as upper bounds on the value of $x_j - x_i$.)

Although the focus of much of the work in representing constraints among real-valued variables was motivated by the desire to perform symbolic model checking on timed automata, such constraints arise in other applications, as well. DDDs can represent difference constraints of the form $x_i - x_j \leq c$. Other constraints of interest include box constraints of the form $x_i \leq c$, or more generally, arbitrary linear constraints of the form $a_1 \cdot x_1 + a_2 \cdot x_2 + \cdots + a_n \cdot x_n \leq c$. Linear decision diagrams extend DDDs to include such constraints [20]. With both DDDs and LDDs, it is also possible to have nodes labeled by Boolean variables as well as ones labeled by constraints. Such decision diagrams can be used when verifying hybrid systems, containing both continuous and discrete state variables.

We can see a parallel between these different forms of decision diagrams and SMT solvers (Chap. 6.) Just as SMT extends Boolean satisfiability solvers to implement decision procedures for other mathematical theories, these generalizations of decision diagrams extend OBDDs to symbolically represent functions over other theories. Both must deal with cases where some combination of constraints is infeasible, leading to conflicts in SMT solvers and infeasible paths in decision diagrams.

10 Scaling OBDD Capacity

Although the introduction of OBDD-based symbolic model checking in the early 1990s provided a major breakthrough in the size and complexity of systems that could be verified, the nature of our field and our desire to apply our tools to real-world systems means that we will always seek to scale them to handle ever larger and more complex problems. Computer systems continue to scale—individual machines have more memory, more cores, and larger disk capacity. In addition, we routinely map problems onto larger clusters of machines that are programmed to work together on a single task. One would expect BDD libraries to have evolved to take advantage of these technological advantages, but unfortunately this is not the case. Most widely used BDD packages still execute on a single core of a single machine, and they are barely able to use the amount of physical memory available on high-end machines. In this section, we highlight some of the efforts to scale the capacity of OBDD implementations, and some of the challenges these efforts face.

In most applications of OBDDs, the ability to handle larger and more complex problems is limited more by the size of the OBDDs generated, rather than the CPU performance. In the extreme case, very large OBDDs can grow to exceed the memory capacity of a machine. On a 64-bit machine, storing the OBDD nodes and all of the associated tables requires, on average, around 40 bytes per node. Thus a machine with 16GB of RAM should, in principle, be able to support OBDD applications using up to around 400 million nodes. In practice, however, the performance of most OBDD implementations becomes unacceptably slow well before that point, due to poor memory-system performance. Traversing graphs in depth-first order (as occurs with the recursive implementation of the Apply algorithm described in Sect. 5) tends to yield poor virtual memory and cache performance, due a lack of locality in the memory access patterns. Standard implementations of the hashtables used for the unique table and the computed cache also exhibit poor memory locality.

Some efforts have been made to implement OBDDs with an eye toward memory performance [5, 51, 54, 64]. These typically employ breadth-first traversal techniques and try to pack the vertices for each level into a contiguous region of memory. A breadth-first approach also lends itself to an implementation where most of the data are stored on a large disk array [39]. Unfortunately, none of these ideas seem to have been incorporated into publicly available OBDD packages.

Early efforts to exploit parallelism in OBDD operations demonstrated the difficulty of this task. Most were implemented in a "shared nothing" environment, where each processor has its own independent memory and can only communicate with other processors via message passing. These implementations require some strategy for *partitioning* the OBDD, so that each node is assigned to some processor. In a message-passing environment, traversing a graph that is partitioned across machines requires message communications, versus the simply memory referencing that occurs on a single machine, and so the performance improvements due to greater parallelism must overcome the potentially high cost of node referencing. Implementations based on a random partitioning of the nodes [60] only showed performance superior to a sequential implementation when the size of the graph ex-

ceeded the capacity of a single processor's memory. In an attempt to minimize the need for message passing, other implementations used a layered partitioning, where the range of variable indices is divided into subranges, and all nodes within a given subrange are mapped onto a single machine. Implementations that were specialized to symbolic model checking could use a partitioning where different regions of the state space were mapped onto different machines [34], following the principles of partitioned OBDDs.

The recent availability of multicore processors supporting multiple threads executing within a single memory space has revived interest in exploiting parallelism in OBDD operations. There are two natural sources of parallelism: *internal*, in which individual operations such as Apply uses multiple threads [26], and *external*, in which a multi-threaded application can invoke multiple Apply operations concurrently [52]. An implementation that uses only internal parallelism requires no changes to the API, while those that support external parallelism can use some mechanism, such as futures, to allow one thread to invoke an operation on OBDDs that are still being generated by other threads.

With the entire OBDD and all of the tables held in a shared memory, any core can access any node or table entry via a memory reference. Obtaining good performance requires careful attention to memory locality and to the potential for thrashing, where multiple threads compete to read and write a small number of cache lines. Such thrashing can occur due to poor design of user data structures or due to excessive calls to synchronization primitives. Excessive synchronization can also lead to a loss of parallelism among the threads.

Perhaps the most ambitious effort on mapping an OBDD implementation onto multicore processors has been by researchers at the University of Twente [26]. Their system maintains a set of workers, each of which maintains a queue of tasks. The system implements the Apply operation with a task for each recursive step. To perform the recursion, each task then spawns two new tasks, with one performed by the current worker and the other added to the worker's queue. Workers are kept busy by having them execute the tasks in their own queues, and "stealing" tasks from other queues when needed. As the computation unfolds, this overall approach will have the effect of having many workers collaboratively executing the Apply operation over different parts of the argument graphs. The system maintains a single unique table and a single computed cache as a way of maintaining consistency and avoiding duplicate efforts by the workers. By carefully designing these tables to use lockless synchronization and cache-friendly data structures, they are able to achieve high performance.

Building a multi-processor system with coherent shared memory becomes prohibitively expensive as the system scales to thousands of processors. Thus, an important challenge remains to devise OBDD implementations that can operate effectively in a fully distributed, shared-nothing environment.

Comparison to SAT checking

We conclude with some observations about how OBDD-based reasoning systems and propositional satisfiability (SAT) checkers have important similarities and differences, both from conceptual and operational viewpoints. Clearly, both are related in the sense that they solve problems encoded in Boolean form. On the other hand, they differ greatly in their intended task—a SAT checker need only find a single satisfying assignment to a Boolean formula, while converting a Boolean formula to an OBDD creates an encoding that describes all of its satisfying solutions. Once we have generated the OBDD representation, it becomes straightforward to perform tasks that SAT solvers cannot readily do, such as counting the number of solutions, or finding an optimal solution for some cost function. Furthermore, OBDDs support operations, such as variable quantification, that have proved to be very challenging extensions for SAT checking.

For most applications of satisfiability testing, SAT checkers based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [25, 24] (Chap. 5) greatly outperform ones that construct an OBDD and then call the SATISFY operation to generate a solution. There are some notable exceptions, however. For example, Bryant conducted experiments on satisfiability problems to test the equivalence of *parity trees*—networks of exclusive-or logic gates computing the odd parity of a set of n Boolean values[17]. Each experiment tested whether a randomly generated tree was functionally equivalent to one consisting of a linear chain of logic gates. We performed tests using four state-of-the-art SAT solvers, but none could handle cases of $n = 48$ inputs within a 900 second time limit. These parity tree problems are known to be difficult cases for DPLL, or in fact any method based on the resolution principle. By contrast, an OBDD-based solver could readily handle such problems in well under 0.1 seconds. Indeed, the OBDD representation of the parity function grows only linearly in n.

This example illustrates the opportunity to devise SAT checkers that combine top-down, search-based strategies, such as DPLL, with ones based on bottom-up, constructive approaches, such as OBDDs. One approach is to replace the traditional clause representation of SAT solvers with BDDs, where the task becomes to find a single variable assignment that yields 1 for all of the OBDDs [23]. Beyond the usual steps of a SAT solver, the solver can also replace some subset of the OBDDs with their conjunction. This approach can deal with problems for which OBDDs outperform DPLL (e.g., the parity tree example), while also getting the performance advantages of DPLL-based SAT solvers.

Other connections between OBDDs and DPLL-based SAT solvers arise due to the observation that the search tree generated by DPLL bears much resemblance to a BDD: each selection of a decision variable in DPLL creates a vertex in the search tree, with outgoing branches based on the value assigned to the variable. Depending on the decision heuristic used, DPLL might follow a common variable ordering across the entire tree, yielding a tree that obeys the ordering constraint of OBDDs, or it may have different orderings along different paths. These are analogous to a

class of BDDs known as "free BDDs," in which variables can occur in any order from the root to a leaf in the graph, but no variable can occur more than once [33].

Huang and Darwiche exploit this relationship to modify an existing DPLL-based SAT solver to instead generate the OBDD representation of a formula given in CNF form [35]. They found this top-down approach to OBDD construction fared better for formulas expressed in CNF than did the usual bottom-up method based on the Apply algorithm. Along related lines, methods have been developed to analyze the clausal representation of a formula and generate a variable ordering that should work well for either SAT checking or for OBDD construction [3].

11 Concluding Remarks

Symbolic model checking arose by linking a model checking algorithm based on fixed-point computations with binary decision diagrams to represent the underlying sets and transition relations [9, 19, 22, 45]. This yielded a major breakthrough in the size and complexity of systems that could be verified. Since that time, OBDDs have been applied to many other tasks, but model checking remains one of their most successful applications. Even as model checkers have been extended to use other reasoning methods, especially Boolean satisfiability solvers, OBDDs have still proved valuable for supporting the range of operations required to implement full-featured model checkers.

Several major goals drive continued research on OBDDs and related representations. First, the desire to represent larger functions requires scaling OBDD implementations to exploit the memory sizes and multicore capabilities of modern processors, as well as large-scale, cluster-based systems. Second, possible variants on OBDDs may enable them to represent Boolean functions in more compact forms. Finally, the desire to verify systems having state variables that range over larger discrete domains, as well as infinite domains, provides a motivation to create types of decision diagrams that can represent other classes of functions.

The resulting research efforts continue to yield novel ideas and approaches, while taking advantage of the key property of OBDDs: that they can represent a variety of functions in a compact form, and that they can be constructed and analyzed using efficient graph algorithms. Future developments will certainly enhance the ability of OBDD-based methods to support model checking.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27**(6), 509–516 (1978)
2. de Alfaro, L., Kwiatkowska, M., Parker, G.N.D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: S. Graf,

M. Schwartzbach (eds.) Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 1785, pp. 395–410 (2000)
3. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Faster SAT and smaller BDDs via common function structure. In: Proceedings of the International Conference on Computer-Aided Design, pp. 443–448 (2001)
4. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994)
5. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 622–627 (1994)
6. Aziz, A., Taşiran, S., Brayton, R.K.: BDD variable ordering for interacting finite state machines. In: Proceedings of the 31st ACM/IEEE Design Automation Conference, pp. 283–288 (1994)
7. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of the International Conference on Computer-Aided Design, pp. 188–191 (1993)
8. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers **45**(9), 993–1002 (1996)
9. Bose, S., Fisher, A.L.: Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In: Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, pp. 759–764 (1989)
10. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proceedings of the 27th ACM/IEEE Design Automation Conference, pp. 40–45 (1990)
11. Brayton, R.K., Hachtel, G.D., McMullen, C.T., Sangiovanni-Vincentelli, A.L.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers (1984)
12. Brown, F.M.: Boolean Reasoning. Kluwer Academic Publishers (1990)
13. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986)
14. Bryant, R.E.: On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. IEEE Transactions on Computers **40**(2), 205–213 (1991)
15. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992)
16. Bryant, R.E.: Binary decision diagrams and beyond: Enabling technologies for formal verification. In: Proceedings of the International Conference on Computer-Aided Design, pp. 236–243 (1995)
17. Bryant, R.E.: A view from the engine room: Computational support for symbolic model checking. In: 25 Years of Model Checking, *Lecture Notes in Computer Science*, vol. 4925 (2007)
18. Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems pp. 401–424 (1994)
19. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992)
20. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear aritmetic. In: Formal Methods in Computer-Aided Design, pp. 53–60 (2009)
21. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. International Journal of Software Tools and Technology Transfer **8**, 4–25 (2006)
22. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Proceedings of the Workshop on Automatic Verification Methods for finite state systems, pp. 365–373 (1989)
23. Damiano, R., Kukula, J.: Checking satisfiability of a conjunction of BDDs. In: Proceedings of the 40th ACM/IEEE Design Automation Conference, pp. 818–923 (2003)
24. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5**(7), 394–397 (1962)

25. Davis, M., Putnam, H.: A computing procedure for quantification theory. J.ACM **3**, 201–215 (1960)
26. van Dijk, T., Laarman, A.W., van de Pol, J.C.: Multi-core BDD operations for symbolic reachability. In: 11th International Workshop on Parallel and Distributed Methods in Verification (2012)
27. Drechsler, R., Günther, W., Somenzi, F.: Using lower bounds during dynamic BDD minimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20**(1), 51–57 (2001)
28. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. International Journal on Software Tools for Technology Transfer **3**(2), 112–136 (2001)
29. Friedman, S.J., Supowit, K.J.: Finding the optimum variable ordering for binary decision diagrams. IEEE Transactions on Computers **39**(5), 710–713 (1990)
30. Fujita, M., Fujisawa, H., Kawato, N.: Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 2–5 (1988)
31. Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in Systems Design **10**, 149–169 (1997)
32. Garey, M.R., Johnson, D.S.: Computers and Intractability. W. H. Freeman and Company (1979)
33. Gunther, W., Drechsler, R.: Minimization of free BDDs. In: Proceedings of ASP-DAC '99, pp. 323–326 (1999)
34. Heyman, T., Geist, D., Grumberg, O., Shuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: Proceedings of the 12th International Conference of Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 1855, pp. 20–35 (2000)
35. Huang, J., Darwiche, A.: Using DPLL for efficient OBDD construction. In: Theory and Applications of Satisfiability Testing, *Lecture Notes in Computer Science*, vol. 3542, pp. 157–172 (2005)
36. Jeong, S.W., Plessier, B., Hachtel, G.D., Somenzi, F.: Variable ordering for FSM traversal. In: Proceedings of the International Conference on Computer-Aided Design (1991)
37. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: Theory and applications. Multiple-Valued Logic **4**(1–2), 9–62 (1998)
38. Knuth, D.S.: The Art of Computer Programming, Volume 4: Combinatorial Algorithms. Addison Wesley (2011)
39. Kunkle, D., Slavici, V., Cooperman, G.: Parallel disk-based computation for large, monolithic binary decision diagrams. In: International Workshop on Parallel and Symbolic Computation, pp. 63–72. ACM (2010)
40. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer Performance Evaluation: Modelling Techniques and Tools, *Lecture Notes in Computer Science*, vol. 2324, pp. 113–140 (2002)
41. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nordic Journal of Computing **6**(3), 271–298 (1999)
42. Madre, J.C., Billon, J.P.: Proving circuit correctness using formal comparison between expected and extracted behaviour. In: Proceedings of the 25th ACM/IEEE Design Automation Conference, pp. 205–210 (1988)
43. Malik, S., Wang, A., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Logic verification using binary decision diagrams in a logic synthesis environment. In: Proceedings of the International Conference on Computer-Aided Design, pp. 6–9 (1988)
44. McDonald, C.B., Bryant, R.E.: CMOS circuit verification with symbolic switch-level timing simulation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20**(3), 458–474 (2001)
45. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
46. Michie, D.: "memo" functions and machine learning. Nature **218**, 19–22 (1968)
47. Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th ACM/IEEE Design Automation Conference, pp. 272–277 (1993)

48. Minato, S.I., Ishiura, N., Yajima, S.: Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In: Proceedings of the 27th ACM/IEEE Design Automation Conference, pp. 52–57 (1990)
49. Møller, J., Lichtenberg, J., Andersen, H., Hulgaard, H.: Difference decision diagrams. In: J. Flum, M. Rodriguez-Artalejo (eds.) Computer Science Logic, *Lecture Notes in Computer Science*, vol. 1683, pp. 826–826 (1999)
50. Narayan, A., Jain, J., Fujita, M., Sangiovanni-Vincentelli, A.L.: Partititoned OBDDs—a compact, canonical, and efficiently manipulable representation for Boolean functions. In: Proceedings of the International Conference on Computer-Aided Design, pp. 547–554 (1996)
51. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 48–55 (1993)
52. Ossowski, J.: JINC—a multi-threaded library for higher-order weighted decision diagram manipulation. Ph.D. thesis, Rheinischen Friedrich-Wilhelms-Universität Bonn (2009)
53. Panda, S., Somenzi, F.: Who are the variables in your neighbourhood. In: Formal Methods in Computer-Aided Design, pp. 74–77 (1995)
54. Ranjan, R.K., Sanghavi, J.V., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: High performance BDD package based on exploiting memory hierarchy. In: Proceedings of the 33rd ACM/IEEE Design Automation Conference, pp. 635–640 (1996)
55. Rudell, R.L.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the International Conference on Computer-Aided Design, pp. 139–144 (1993)
56. Schröer, O., Wegener, I.: The theory of zero-suppressed BDDs and the number of knight's tours. Formal Methods in Systems Design **13**(3), 235–253 (1998)
57. Sieling, D.: On the existence of polynomial time approximation schemes for OBDD minimization. In: Symposium on Theoretical Aspects of Computer Science, *Lecture Notes in Computer Science*, vol. 1373, pp. 205–215 (1998)
58. Sieling, D., Wegener, I.: Reduction of OBDDs in linear time. Information Processing Letters **48**(3), 139–144 (1993)
59. Somenzi, F.: Efficient manipulation of decision diagrams. International Journal on Software Tools for Technology Transfer **3**(2), 171–181 (2001)
60. Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: Proceedings of the 33rd ACM/IEEE Design Automation Conference, pp. 641–644 (1996)
61. Tani, S., Hamaguchi, K., Yajima, S.: The complexity of the optimal variable ordering problems of shared binary decision diagrams. Algorithms and Computation **762**, 389–398 (1993)
62. Wang, F.: Efficient verification of timed automata with efficient BDD-like data structures. International Journal of Software Tools for Technology Transfer **6**(1), 77–97 (2004)
63. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: Formal Methods in Computer-Aided Design, *Lecture Notes in Computer Science*, vol. 1522 (1998)
64. Yang, B., Chen, Y.A., Bryant, R.E., O'Hallaron, D.R.: Space- and time-efficient BDD construction via working set control. In: Proceedings of ASP-DAC '98, pp. 423–432. Yokohoma,Japan (1998)
65. Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. zero-suppressed BDDs for CTL symbolic model checking of Petri nets. In: Formal Methods in Computer-Aided Design, *Lecture Notes in Computer Science*, vol. 1166, pp. 435–449 (1996)