and g denote the same boolean functions if, and only if, the reduced OBDDs have identical structure.

Test for validity. We can test a function $f(x_1, x_2, \ldots, x_n)$ for validity (i.e. f always computes 1) in the following way. Compute a reduced OBDD for f. Then f is valid if, and only if, its reduced OBDD is B_1.

Test for implication. We can test whether $f(x_1, x_2, \ldots, x_n)$ implies $g(x_1, x_2, \ldots, x_n)$ (i.e. whenever f computes 1, then so does g) by computing the reduced OBDD for $f \cdot \overline{g}$. This is B_0 iff the implication holds.

Test for satisfiability. We can test a function $f(x_1, x_2, \ldots, x_n)$ for satisfiability (f computes 1 for at least one assignment of 0 and 1 values to its variables). The function f is satisfiable iff its reduced OBDD is not B_0.

6.2 Algorithms for reduced OBDDs

6.2.1 The algorithm `reduce`

The reductions C1–C3 are at the core of any serious use of OBDDs, for whenever we construct a BDD we will want to convert it to its reduced form. In this section, we describe an algorithm `reduce` which does this efficiently for ordered BDDs.

If the ordering of B is $[x_1, x_2, \ldots, x_l]$, then B has at most $l + 1$ layers. The algorithm `reduce` now traverses B layer by layer in a bottom-up fashion, beginning with the terminal nodes. In traversing B, it assigns an integer label $\mathrm{id}(n)$ to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function if, and only if, $\mathrm{id}(n)$ equals $\mathrm{id}(m)$.

Since `reduce` starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction C1). Similarly, the 1-nodes all get the next label, say #1.

Now let us inductively assume that `reduce` has already assigned integer labels to all nodes of a layer $> i$ (i.e. all terminal nodes and x_j-nodes with $j > i$). We describe how nodes of layer i (i.e. x_i-nodes) are being handled.

Definition 6.8 Given a non-terminal node n in a BDD, we define $\mathrm{lo}(n)$ to be the node pointed to via the dashed line from n. Dually, $\mathrm{hi}(n)$ is the node pointed to via the solid line from n.

Let us describe how the labelling is done. Given an x_i-node n, there are three ways in which it may get its label:

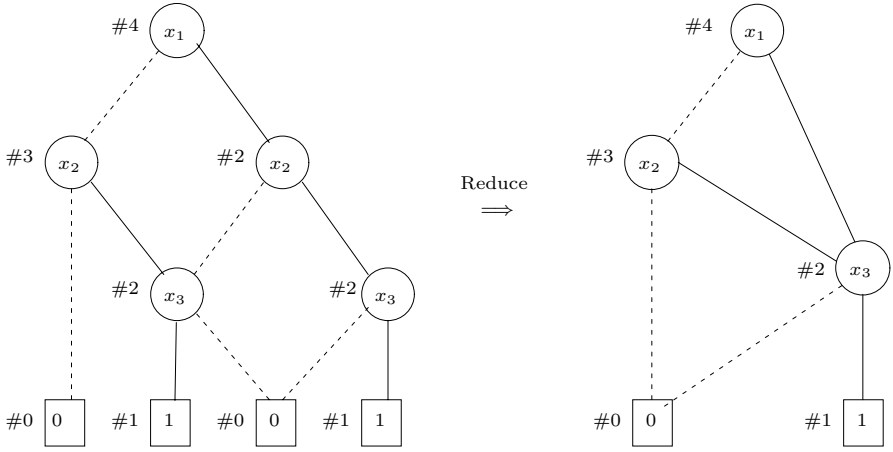

Figure 6.14. An example execution of the algorithm `reduce`.

- If the label $\mathrm{id}(\mathrm{lo}(n))$ is the same as $\mathrm{id}(\mathrm{hi}(n))$, then we set $\mathrm{id}(n)$ to be that label. That is because the boolean function represented at n is the same function as the one represented at $\mathrm{lo}(n)$ and $\mathrm{hi}(n)$. In other words, node n performs a redundant test and can be eliminated by reduction C2.
- If there is another node m such that n and m have the same variable x_i, and $\mathrm{id}(\mathrm{lo}(n)) = \mathrm{id}(\mathrm{lo}(m))$ and $\mathrm{id}(\mathrm{hi}(n)) = \mathrm{id}(\mathrm{hi}(m))$, then we set $\mathrm{id}(n)$ to be $\mathrm{id}(m)$. This is because the nodes n and m compute the same boolean function (compare with reduction C3).
- Otherwise, we set $\mathrm{id}(n)$ to the next unused integer label.

Note that only the last case creates a new label. Consider the OBDD in left side of Figure 6.14; each node has an integer label obtained in the manner just described. The algorithm `reduce` then finishes by redirecting edges bottom-up as outlined in C1–C3. The resulting reduced OBDD is in right of Figure 6.14. Since there are efficient bottom-up traversal algorithms for dags, `reduce` is an efficient operation in the number of nodes of an OBDD.

6.2.2 The algorithm `apply`

Another procedure at the heart of OBDDs is the algorithm `apply`. It is used to implement operations on boolean functions such as $+$, $\cdot$, $\oplus$ and complementation (via $f \oplus 1$). Given OBDDs B_f and B_g for boolean formulas f and g, the call `apply` (op, B_f, B_g) computes the reduced OBDD of the boolean formula $f \,\mathrm{op}\, g$, where op denotes any function from $\{0,1\} \times \{0,1\}$ to $\{0,1\}$.

The intuition behind the `apply` algorithm is fairly simple. The algorithm operates recursively on the structure of the two OBDDs:

1. let v be the variable highest in the ordering (=leftmost in the list) which occurs in B_f or B_g.
2. split the problem into two subproblems for v being 0 and v being 1 and solve recursively;
3. at the leaves, apply the boolean operation op directly.

The result will usually have to be reduced to make it into an OBDD. Some reduction can be done 'on the fly' in step 2, by avoiding the creation of a new node if both branches are equal (in which case return the common result), or if an equivalent node already exists (in which case, use it).

Let us make all this more precise and detailed.

Definition 6.9 Let f be a boolean formula and x a variable.

1. We denote by $f[0/x]$ the boolean formula obtained by replacing all occurrences of x in f by 0. The formula $f[1/x]$ is defined similarly. The expressions $f[0/x]$ and $f[1/x]$ are called restrictions of f.
2. We say that two boolean formulas f and g are semantically equivalent if they represent the same boolean function (with respect to the boolean variables that they depend upon). In that case, we write $f \equiv g$.

For example, if $f(x,y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x})$, then $f[0/x](x,y)$ equals $0 \cdot (y + \overline{0})$, which is semantically equivalent to 0. Similarly, $f[1/y](x,y)$ is $x \cdot (1 + \overline{x})$, which is semantically equivalent to x.

Restrictions allow us to perform recursion on boolean formulas, by decomposing boolean formulas into simpler ones. For example, if x is a variable in f, then f is equivalent to $\overline{x} \cdot f[0/x] + x \cdot f[1/x]$. To see this, consider the case $x = 0$; the expression computes to $f[0/x]$. When $x = 1$ it yields $f[1/x]$. This observation is known as the *Shannon expansion*, although it can already be found in G. Boole's book *'The Laws of Thought'* from 1854.

Lemma 6.10 (Shannon expansion) For all boolean formulas f and all boolean variables x (even those not occurring in f) we have

$$f \equiv \overline{x} \cdot f[0/x] + x \cdot f[1/x]. \tag{6.1}$$

The function `apply` is based on the Shannon expansion for $f \operatorname{op} g$:

$$f \operatorname{op} g = \overline{x_i} \cdot (f[0/x_i] \operatorname{op} g[0/x_i]) + x_i \cdot (f[1/x_i] \operatorname{op} g[1/x_i]). \tag{6.2}$$

This is used as a control structure of `apply` which proceeds from the roots

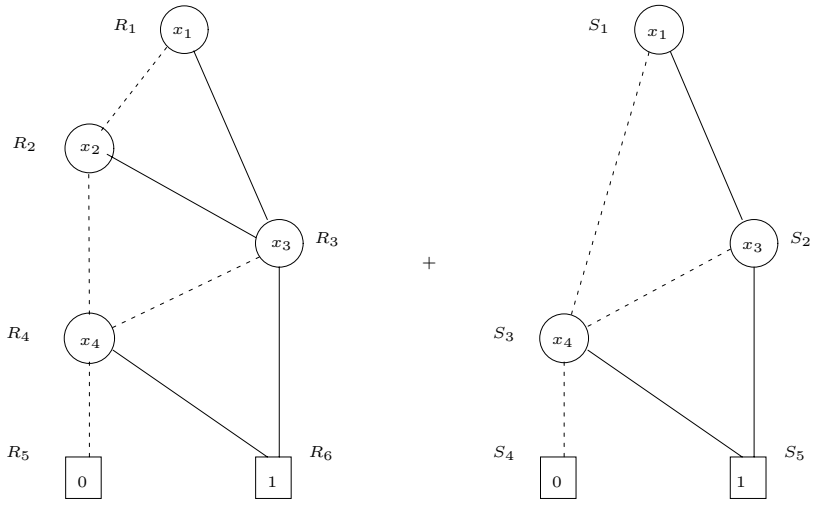

Figure 6.15. An example of two arguments for a call `apply` $(+, B_f, B_g)$.

of B_f and B_g downwards to construct nodes of the OBDD $B_{f \,\mathrm{op}\, g}$. Let r_f be the root node of B_f and r_g the root node of B_g.

1. If both r_f and r_g are terminal nodes with labels l_f and l_g, respectively (recall that terminal labels are either 0 or 1), then we compute the value $l_f \,\mathrm{op}\, l_g$ and let the resulting OBDD be B_0 if that value is 0 and B_1 otherwise.
2. In the remaining cases, at least one of the root nodes is a non-terminal. Suppose that both root nodes are x_i-nodes. Then we create an x_i-node n with a dashed line to `apply` $(\mathrm{op}, \mathrm{lo}(r_f), \mathrm{lo}(r_g))$ and a solid line to `apply` $(\mathrm{op}, \mathrm{hi}(r_f), \mathrm{hi}(r_g))$, i.e. we call `apply` recursively on the basis of (6.2).
3. If r_f is an x_i-node, but r_g is a terminal node or an x_j-node with $j > i$, then we know that there is no x_i-node in B_g because the two OBDDs have a compatible ordering of boolean variables. Thus, g is independent of x_i $(g \equiv g[0/x_i] \equiv g[1/x_i])$. Therefore, we create an x_i-node n with a dashed line to `apply` $(\mathrm{op}, \mathrm{lo}(r_f), r_g)$ and a solid line to `apply` $(\mathrm{op}, \mathrm{hi}(r_f), r_g)$.
4. The case in which r_g is a non-terminal, but r_f is a terminal or an x_j-node with $j > i$, is handled symmetrically to case 3.

The result of this procedure might not be reduced; therefore `apply` finishes by calling the function `reduce` on the OBDD it constructed. An example of `apply` (where op is +) can be seen in Figures 6.15–6.17. Figure 6.16 shows the recursive descent control structure of `apply` and Figure 6.17 shows the final result. In this example, the result of `apply` $(+, B_f, B_g)$ is B_f.

Figure 6.16 shows that numerous calls to `apply` occur several times with the same arguments. Efficiency could be gained if these were evaluated only

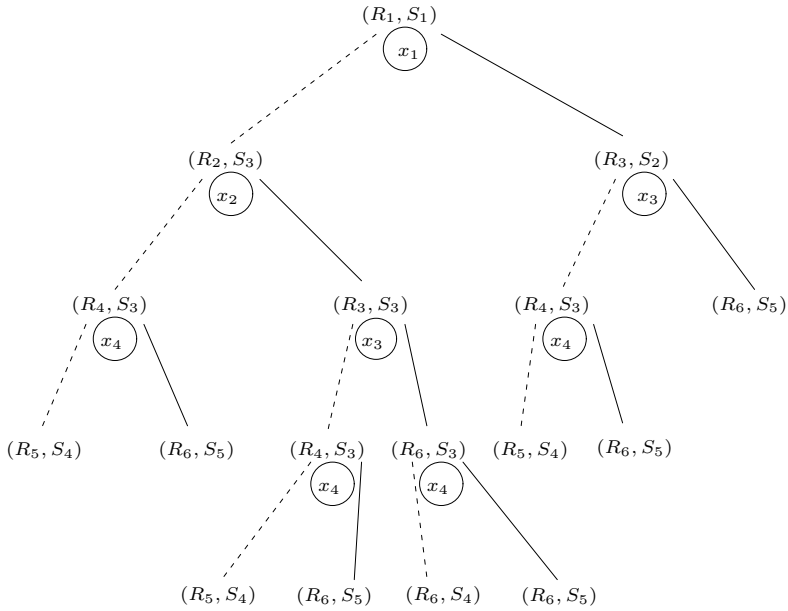

Figure 6.16. The recursive call structure of `apply` for the example in Figure 6.15 (without memoisation).

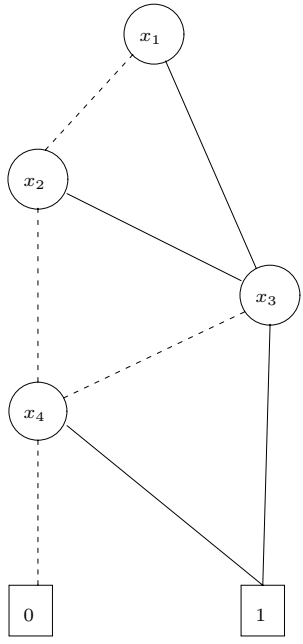

Figure 6.17. The result of `apply` $(+, B_f, B_g)$, where B_f and B_g are given in Figure 6.15.

the first time and the result remembered for future calls. This programming technique is known as memoisation. As well as being more efficient, it has the advantage that the resulting OBDD requires less reduction. (In this example, using memoisation eliminates the need for the final call to `reduce` altogether.) Without memoisation, `apply` is exponential in the size of its arguments, since each non-leaf call generates a further two calls. With memoisation, the number of calls to apply is bounded by $2 \cdot |B_f| \cdot |B_g|$, where $|B|$ is the size of the BDD. This is a worst-time complexity; the actual performance is often much better than this.

6.2.3 The algorithm `restrict`

Given an OBDD B_f representing a boolean formula f, we need an algorithm `restrict` such that the call $\texttt{restrict}(0, x, B_f)$ computes the reduced OBDD representing $f[0/x]$ using the same variable ordering as B_f. The algorithm for $\texttt{restrict}(0, x, B_f)$ works as follows. For each node n labelled with x, incoming edges are redirected to $\mathrm{lo}(n)$ and n is removed. Then we call `reduce` on the resulting OBDD. The call $\texttt{restrict}\ (1, x, B_f)$ proceeds similarly, only we now redirect incoming edges to $\mathrm{hi}(n)$.

6.2.4 The algorithm `exists`

A boolean function can be thought of as putting a constraint on the values of its argument variables. For example, the function $x + (\overline{y} \cdot z)$ evaluates to 1 only if x is 1; or y is 0 and z is 1 – this is a constraint on x, y, and z.

It is useful to be able to express the relaxation of the constraint on a subset of the variables concerned. To allow this, we write $\exists x.\, f$ for the boolean function f with the constraint on x relaxed. Formally, $\exists x.\, f$ is defined as $f[0/x] + f[1/x]$; that is, $\exists x.\, f$ is true if f could be made true by putting x to 0 or to 1. Given that $\exists x.\, f \stackrel{\text{def}}{=} f[0/x] + f[1/x]$ the `exists` algorithm can be implemented in terms of the algorithms `apply` and `restrict` as

$$\texttt{apply}\,(+, \texttt{restrict}\,(0, x, B_f), \texttt{restrict}\,(1, x, B_f))\ . \tag{6.3}$$

Consider, for example, the OBDD B_f for the function $f \stackrel{\text{def}}{=} x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$, shown in Figure 6.19. Figure 6.20 shows $\texttt{restrict}(0, x_3, B_f)$ and $\texttt{restrict}(1, x_3, B_f)$ and the result of applying + to them. (In this case the apply function happens to return its second argument.)

We can improve the efficiency of this algorithm. Consider what happens during the `apply` stage of (6.3). In that case, the `apply` algorithm works on two BDDs which are identical all the way down to the level of the x-nodes;

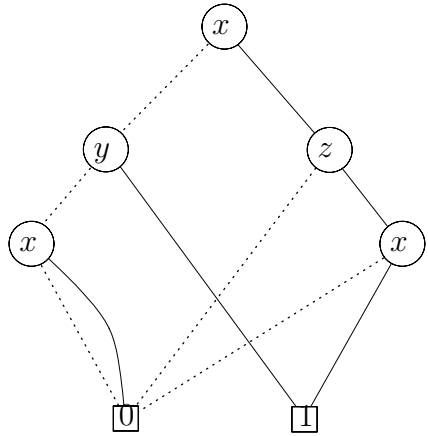

Figure 6.18. An example of a BDD which is not a read-1-BDD.

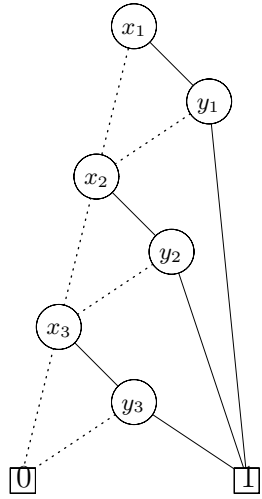

Figure 6.19. A BDD B_f to illustrate the `exists` algorithm.

therefore the returned BDD also has that structure down to the x-nodes. At the x-nodes, the two argument BDDs differ, so the `apply` algorithm will compute the apply of $+$ to these two subBDDs and return that as the subBDD of the result. This is illustrated in Figure 6.20. Therefore, we can compute the OBDD for $\exists x.\, f$ by taking the OBDD for f and replacing each node labelled with x by the result of calling `apply` on $+$ and its two branches.

This can easily be generalised to a sequence of `exists` operations. We write $\exists \hat{x}.\, f$ to mean $\exists x_1.\exists x_2. \ldots \exists x_n.\, f$, where $\hat{x}$ denotes $(x_1, x_2, \ldots, x_n)$.

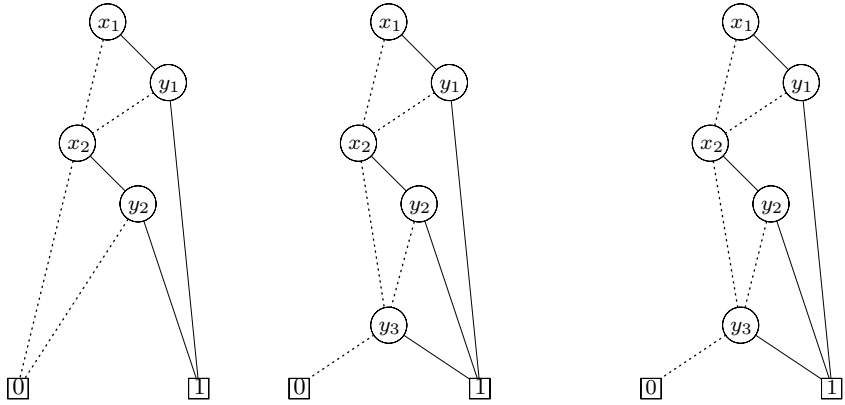

Figure 6.20. `restrict`$(0, x_3, B_f)$ and `restrict`$(1, x_3, B_f)$ and the result of applying $+$ to them.

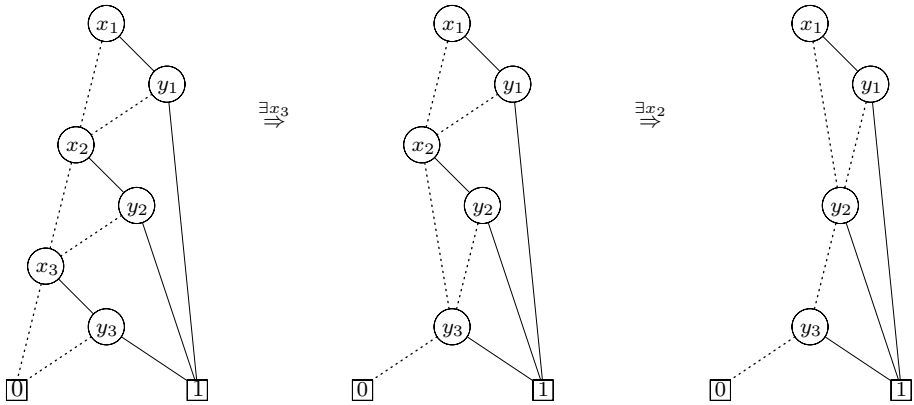

Figure 6.21. OBDDs for f, $\exists x_3.\, f$ and $\exists x_2.\exists x_3.\, f$.

The OBDD for this boolean function is obtained from the OBDD for f by replacing *every* node labelled with an x_i by the $+$ of its two branches.

Figure 6.21 shows the computation of $\exists x_3.\, f$ and $\exists x_2.\exists x_3.\, f$ (which is semantically equivalent to $x_1 \cdot y_1 + y_2 + y_3$) in this way.
The boolean quantifier $\forall$ is the dual of $\exists$:

$$\forall x.f \stackrel{\text{def}}{=} f[0/x] \cdot f[1/x]$$

asserting that f could be made false by putting x to 0 or to 1.

The translation of boolean formulas into OBDDs using the algorithms of this section is summarised in Figure 6.22.

Boolean formula f	Representing OBDD B_f
0	B_0 (Fig. 6.6)
1	B_1 (Fig. 6.6)
x	B_x (Fig. 6.6)
$\overline{f}$	swap the 0- and 1-nodes in B_f
$f + g$	`apply` $(+, B_f, B_g)$
$f \cdot g$	`apply` $(\cdot, B_f, B_g)$
$f \oplus g$	`apply` $(\oplus, B_f, B_g)$
$f[1/x]$	`restrict` $(1, x, B_f)$
$f[0/x]$	`restrict` $(0, x, B_f)$
$\exists x.f$	`apply` $(+, B_{f[0/x]}, B_{f[1/x]})$
$\forall x.f$	`apply` $(\cdot, B_{f[0/x]}, B_{f[1/x]})$

Figure 6.22. Translating boolean formulas f to OBDDs B_f, given a fixed, global ordering on boolean variables.

Algorithm	Input OBDD(s)	Output OBDD	Time-complexity
`reduce`	B	reduced B	$O(\|B\| \cdot \log \|B\|)$
`apply`	B_f, B_g (reduced)	$B_{f \text{ op } g}$ (reduced)	$O(\|B_f\| \cdot \|B_g\|)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (reduced)	$O(\|B_f\| \cdot \log \|B_f\|)$
$\exists$	B_f (reduced)	$B_{\exists x_1.\exists x_2.\ldots.\exists x_n.f}$ (reduced)	NP-complete

Figure 6.23. Upper bounds in terms of the input OBDD(s) for the worst-case running times of our algorithms needed in our implementation of boolean formulas.

6.2.5 Assessment of OBDDs

Time complexities for computing OBDDs We can measure the complexity of the algorithms of the preceding section by giving upper bounds for the running time in terms of the sizes of the input OBDDs. The table in Figure 6.23 summarises these upper bounds (some of those upper bounds may require more sophisticated versions of the algorithms than the versions presented in this chapter). All the operations except nested boolean quantification are practically efficient in the size of the participating OBDDs. Thus, modelling very large systems with this approach will work if the OBDDs

which represent the systems don't grow too large too fast. If we can somehow control the size of OBDDs, e.g. by using good heuristics for the choice of variable ordering, then these operations are computationally feasible. It has already been shown that OBDDs modelling certain classes of systems and networks don't grow excessively.

The expensive computational operations are the nested boolean quantifications $\exists z_1 \ldots \exists z_n . f$ and $\forall z_1 \ldots \forall z_n . f$. By exercise 1 on page 406, the computation of the OBDD for $\exists z_1 \ldots \exists z_n . f$, given the OBDD for f, is an NP-complete problem[2]; thus, it is unlikely that there exists an algorithm with a feasible worst-time complexity. This is not to say that boolean functions modelling practical systems may not have efficient nested boolean quantifications. The performance of our algorithms can be improved by using further optimisation techniques, such as parallelisation.

Note that the operations `apply`, `restrict`, etc. are only efficient in the size of the input OBDDs. So if a function f does not have a compact representation as an OBDD, then computing with its OBDD will not be efficient. There are such nasty functions; indeed, one of them is *integer multiplication*. Let $b_{n-1}b_{n-2}\ldots b_0$ and $a_{n-1}a_{n-2}\ldots a_0$ be two n-bit integers, where b_{n-1} and a_{n-1} are the most significant bits and b_0 and a_0 are the least significant bits. The multiplication of these two integers results in a $2n$-bit integer. Thus, we may think of multiplication as $2n$ many boolean functions f_i in $2n$ variables (n bits for input b and n bits for input a), where f_i denotes the ith output bit of the multiplication. The following negative result, due to R. E. Bryant, shows that OBDDs cannot be used for implementing integer multiplication.

Theorem 6.11 Any OBDD representation of f_{n-1} has at least a number of vertices proportional to 1.09^n, i.e. its size is exponential in n.

Extensions and variations of OBDDs There are many variations and extensions to the OBDD data structure. Many of them can implement certain operations more efficiently than their OBDD counterparts, but it seems that none of them perform as well as OBDDs overall. In particular, one feature which many of the variations lack is the canonical form; therefore they lack an efficient algorithm for deciding when two objects denote the same boolean function.

One kind of variation allows non-terminal nodes to be labelled with binary operators as well as boolean variables. *Parity OBDDs* are like OBDDs in that there is an ordering on variables and every variable may occur at

[2] Another NP-complete problem is to decide the satisfiability of formulas of propositional logic.

most once on a path; but some non-terminal nodes may be labelled with $\oplus$, the exclusive-or operation. The meaning is that the function represented by that node is the exclusive-or of the boolean functions determined by its children. Parity OBDDs have similar algorithms for `apply`, `restrict`, etc. with the same performance, but they do not have a canonical form. Checking for equivalence cannot be done in constant time. There is, however, a cubic algorithm for determining equivalence; and there are also efficient probabilistic tests. Another variation of OBDDs allows complementation nodes, with the obvious meaning. Again, the main disadvantage is the lack of canonical form.

One can also allow non-terminal nodes to be unlabelled and to branch to more than two children. This can then be understood either as non-deterministic branching, or as probabilistic branching: throw a pair of dice to determine where to continue the path. Such methods may compute wrong results; one then aims at repeating the test to keep the (probabilistic) error as small as desired. This method of repeating probabilistic tests is called *probabilistic amplification*. Unfortunately, the satisfiability problem for probabilistic branching OBDDs is NP-complete. On a good note, probabilistic branching OBDDs can *verify* integer multiplication.

The development of extensions or variations of OBDDS which are customised to certain classes of boolean functions is an important area of ongoing research.

6.3 Symbolic model checking

The use of BDDs in model checking resulted in a significant breakthrough in verification in the early 1990s, because they have allowed systems with much larger state spaces to be verified. In this section, we describe in detail how the model-checking algorithm presented in Chapter 3 can be implemented using OBDDs as the basic data structure.

The pseudo-code presented in Figure 3.28 on page 227 takes as input a CTL formula ϕ and returns the set of states of the given model which satisfy ϕ. Inspection of the code shows that the algorithm consists of manipulating intermediate sets of states. We show in this section how the model and the intermediate sets of states can be stored as OBDDs; and how the operations required in that pseudo-code can be implemented in terms of the operations on OBDDs which we have seen in this chapter.

We start by showing how sets of states are represented with OBDDs, together with some of the operations required. Then, we extend that to the representation of the transition system; and finally, we show how the remainder of the required operations is implemented.