Binary Decision Diagrams

Fabio SOMENZI
Department of Electrical and Computer Engineering
University of Colorado at Boulder

Abstract

We review Binary Decision Diagrams presenting the properties and algorithms that are most relevant to their application to the verification of sequential systems.

1 Introduction

Many tasks in the design and verification of digital systems manipulate large propositional formulae. It is therefore important to have efficient ways of representing and manipulating such formulae. In recent times Binary Decision Diagrams (BDDs) have emerged as the representation of choice for many applications. Though BDDs are relatively old [39, 1], it was the work of Bryant [8] that attracted the attention and renewed the interest of many researchers. Bryant observed that reduced, ordered, binary decision diagrams are a canonical representation of boolean functions. The import of this observation can be seen in the fact that nowadays, reduction and ordering are usually implied when referring to BDDs. Following the established use, we refer to reduced ordered binary decision diagrams simply as to BDDs.

Canonicity reduces the semantic notion of equivalence to the syntactic notion of isomorphism. It is the source of both efficiency and ease of use for BDDs. On the one hand, canonicity enhances the effectiveness of memoization techniques. On the other hand, it makes the test for equivalence inexpensive. Canonicity has also one important drawback: It is the prime reason why BDDs are less concise than circuits in general. There are families of functions for which conjunctive normal forms are exponentially more concise than BDDs and vice versa. The same happens when comparing some diagrammatic representations. It would therefore be a mistake to use BDDs indiscriminately whenever a representation for boolean functions is required. It would be equally mistaken to dismiss BDDs on the grounds that multipliers have provably exponential BDDs [9]. The best results in many applications come from a flexible approach that combines the strengths of several representations (e.g., [14]).

The application that led to the initial development of BDDs is switch-level simulation of MOS circuits [7]. Since then, their use has proliferated to the point that an exhaustive list of all the problems to which decision diagrams have been applied is not feasible here. We limit ourselves to mentioning some of the more popular and significant applications of BDDs and their derivatives. Verification of the correctness of hardware relies on BDDs for both the representation of the circuits and the manipulation of sets of states. Model checking algorithms based on this implicit representation of sets have successfully verified properties of systems with very large numbers of states (10^{100} or more) [12, 44, 6]. In the optimization of logic circuits, BDDs are used, among other things, to represent "don't care" conditions [56], and to translate boolean functions into circuits based on a specific implementation technology [42, 13, 25]. Testing and optimization of sequential circuits also benefit from the use of BDDs [18]. Various forms of decision diagrams can be

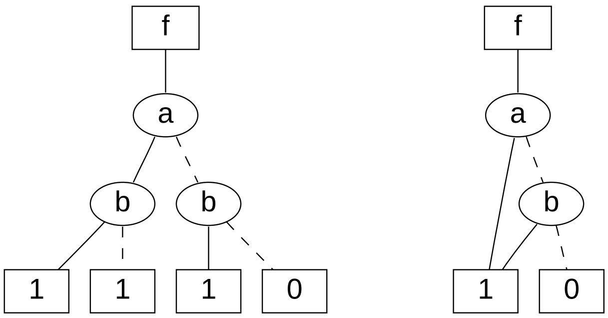


Figure 1: Decision tree and decision diagram for the disjunction of a and b.

applied to the solution of large systems of linear equations—in particular those found in the analysis of large Markov chains [30]—and to graph algorithms like maximum flow in a network [31].

The rest of this chapter is organized as follows. In Section 2 we present informally BDDs. After introducing notation and background in Section 3, we give a definition of BDDs in Section 4. In Section 5 we discuss the main algorithms to manipulate BDDs. Section 6 is devoted to the problem of variable ordering. Section 7 discusses the efficient implementation of BDD packages. Section 8 surveys the application of BDDs to the verification of sequential hardware. Section 9 summarizes and concludes.

2 From Decision Trees to Decision Diagrams

2.1 BDDs Are Reduced Decision Trees

The left part of Figure 1 shows an ordered decision tree for the disjunction of two variables a and b. The node at the top—labeled f—is the *function* node. The elliptical nodes labeled with variable names are the *internal* nodes, and the rectangular nodes at the bottom are the *terminal* nodes. The terminal nodes are labeled by either 1 (signifying *true*) or 0 (signifying *false*). Evaluation of f for a given valuation of a and b consists of following a path from the function node to a terminal node. The label of the terminal node is the value sought. At each internal node the path follows the solid arc if the variable labeling the node evaluates to 1, and the dashed arc otherwise. The solid arc is called the *then* arc, while the dashed arc is called the *else* arc. The order in which the variables appear is the same along all paths. In this case, it is $a < b$. (The smallest element of the order is the variable closest to the function node.)

The right part of Figure 1 shows the Binary Decision Diagram for the order $a < b$. It is obtained from the corresponding decision tree by a process called *reduction*.

Definition 1 *Reduction consists of the application of the following two rules starting from the decision tree and continuing until neither rule can be applied.*

1. *If two nodes are terminal and have the same label, or are internal and have the same children, they are merged.*

2. *If an internal node has identical children it is removed from the graph and its incoming nodes are redirected to the child.*

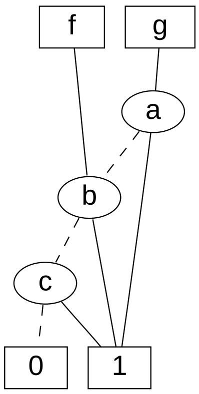


Figure 2: Shared BDDs.

In the case of Figure 1, reduction proceeds by merging the three terminal nodes labeled 1. As a consequence, the children of one of the internal nodes labeled b become the same node. This causes the application of the second rule which produces the graph on the right of Figure 1. No further application of either rule is possible and reduction terminates.

The result of reduction depends only on the function to which it is applied and on the order of the variables. It is independent of the order of application of the rules.

2.2 Shared BDDs and Strong Canonical Form

In practice one does not build the decision tree and then reduces it. Rather, BDDs are created starting from the BDDs for the constants and the variables by application of the usual boolean connectives and are kept reduced at all times. At the same time several functions are represented by one multi-rooted diagram. Indeed, each node of a BDD has a function associated with it. If we have several functions, they will have subfunctions in common. For instance, if we have $f = b + c$ and $g = a + b + c$, we represent them like in Figure 2. As a special case, two equivalent functions are represented by the same BDD (not just two identical BDDs). This approach, therefore makes equivalence check a constant-time operation. Its implementation is based on a dictionary of all BDDs nodes in existence in an application. This dictionary is called the *unique table*. Operations that build BDDs start from the bottom (the constant functions) and proceed up to the function nodes. Whenever an operation needs to add a node to a BDD that it is building, it knows already the two nodes (say, f_1 and f_0) that are going to be the new node's children, and the variable (say, x_i) that is going to label it. Therefore the operation first checks if $f_1 = f_0$. If indeed the two children are the same, no new node needs to be created. (This is in accordance to the second reduction rule.) If $f_1 \neq f_0$, it looks up the unique table for the existence of the triple (x_i, f_1, f_0), that is, for a node with the desired characteristics. Only if such node is not found, it is created (in accordance to the first reduction rule). The representation we have just outlined is called *strong canonical form* and is quite common in software packages that manipulate BDDs, in which equivalence is tested by a simple pointer comparison.

2.3 Attributed Arcs

The BDDs for f and f' are very similar. The only difference being the values of the leaves, which are interchanged. This suggests the possibility of actually using the same subgraph to represent both f and f'.

Suppose a BDD represents f. If we are interested in $g = f'$, it is then sufficient to remember that the function we have in the multi-rooted graph is the complement of g. This can be accomplished by attaching an attribute to the arc pointing to the top node of f. An arc with the complement attribute is called a *complement arc*. The arcs without the attributes are called *regular arcs*. The use of complement arcs slightly complicates the manipulation of BDDs, but has three advantages. Obviously, it decreases the memory requirements.[1] The second, more important consequence of using complement arcs is the fact that complementation can be done in constant time—the BDD is already in place—and checking two functions for one being the complement of the other also takes constant time. Finally, inexpensive complementation allows an implementor to use De Morgan's laws freely. For instance, once an algorithm for conjunctive decomposition of a function has been coded, disjunctive decomposition can be obtained with negligible overhead by complementing the conjunctive decomposition of the complement. Note that with complement arcs we need only one constant function (we choose **1**) and hence only one leaf in the multi-rooted DAG. In Section 5 we shall see that in order to preserve the canonicity of the representation, only the arcs out of function nodes and the *else* arcs of internal nodes can be complement arcs. In the figures, therefore, a dotted line unambiguously designates a complement arc. Also, in drawings we shall normally align all the nodes labeled by the same variable, and indicate the variable only once at the left. This leaves the interior of the node free for a unique identifier, to which the examples will often refer. The attribute mechanism is quite general: Other attributes have been used for other purposes [47, 36].

3 Preliminaries

Binary Decision Diagrams represent boolean functions. In this section we introduce the notation we use for boolean formulae and recall the basic result that we need for BDD manipulation. Given a boolean algebra B, the boolean formulae on the variables $x_1, \ldots, x_n$ are obtained by recursively applying negation, conjunction, and disjunction to the elements of B and to the variables. We denote conjunction by '$\cdot$', disjunction by '$+$', and negation by '$'$'. Unless otherwise noted, we assume $B = \{0, 1\}$. Thus, $x_1 \cdot (x_2 + x_3)'$ is a formula. When no confusion arises we write $x_i x_j$ instead of $x_i \cdot x_j$ and we drop parentheses assuming that negation has the highest precedence, and conjunction binds more strongly than disjunction. Formulae designate functions in the usual way.

Definition 2 *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then*

$$\begin{aligned} f_{x_i} &= f(x_1, \ldots, x_{i-i}, 1, x_{i+1}, \ldots, x_n) \\ f_{x_i'} &= f(x_1, \ldots, x_{i-i}, 0, x_{i+1}, \ldots, x_n) \end{aligned}$$

are the positive *and* negative cofactors *of f with respect to x_i.*

The cofactors are given several different names in the literature, e.g., *restrictions*. If $f_{x_i} \neq f_{x_i'}$ then f *depends* on x_i and x_i is said to belong to the *support* of f. The cofactors of f with respect to x_i do not depend on x_i. Cofactors commute, that is:

$$(f_{x_i})_{x_j} = (f_{x_j})_{x_i} = f_{x_i x_j},$$

[1] We are assuming that the overhead for storing the attributes is negligible. This indeed the case because it is customary to store the complementation flag in the otherwise unused least significant bit of a pointer. The maximum decrease in memory requirements is by a factor of 2, but in practice this limit is seldom approached.

and they distribute over negation, conjunction, and disjunction:

$$\begin{aligned}(f')_{x_i} &= (f_{x_i})' \\ (f \cdot g)_{x_i} &= f_{x_i} \cdot g_{x_i} \\ (f + g)_{x_i} &= f_{x_i} + g_{x_i}.\end{aligned}$$

The following result, known as *expansion theorem*, is due to Boole and forms the basis for most algorithms that manipulate BDDs.

Theorem 1 *Let $f(x_1, \ldots, x_n)$ be a boolean function. Then*

$$f(x_1, \ldots, x_n) = x_i' \cdot f_{x_i'} + x_1 \cdot f_{x_i}. \tag{1}$$

Example 1 Suppose $f(a, b, c, d) = ab + b'c + cd$. Then

$$\begin{aligned}f_b &= a + cd \\ f_{b'} &= c \\ f &= b \cdot (a + cd) + b' \cdot c \\ f_{ab'} &= f_{b'}.\end{aligned}$$

4 BDDs and Their Canonicity

Definition 3 *A BDD is a directed acyclic graph $(V \cup \Phi \cup \{\mathbf{1}\}, E)$ representing a vector of boolean functions F. The nodes are partitioned into three subsets. V is the set of the internal nodes. The outdegree of $v \in V$ is 2. Every node v has a label $l(v)$ in the support of F. $\mathbf{1}$ is the terminal node: Its outdegree is 0. Φ is the set of function nodes: The outdegree of $\phi \in \Phi$ is 1 and its indegree is 0. The function nodes are in one-to-one correspondence with the components of F. The outgoing arcs of function nodes may have the* complement *attribute. The two outgoing arcs for a node $v \in V$ are labeled T and E, respectively. The E arc may have the* complement *attribute. We use $(l(v), T(v), E(v))$ to indicate an internal node and its two outgoing arcs. The variables in the support of F are ordered: If v_j is a descendant of v_i ($v_i, v_j \in V$), then $l(v_i) < l(v_j)$. The function vector F represented by a BDD is defined as follows:*

1. *The function of the terminal node is the constant function 1.*

2. *The function of a regular arc is the function of the head node; the function of a complement arc is the complement of the function of the head node.*

3. *The function of a node $v \in V$ is given by $l(v)' f_E + l(v) f_T$, where f_T (f_E) is the function of $T(v)$ ($E(v)$).*

4. *The function of $\phi \in \Phi$ is the function of its outgoing arc.*

The distinction between the function of a node and the function of an arc allows us to deal with attributed arcs in a natural way.

Theorem 2 *BDDs are canonical (the representation of a vector of boolean functions F is unique for a given variable ordering) if:*

1. *There are no distinct internal nodes v_1 and v_2 such that $l(v_1) = l(v_2)$, $T(v_1) = T(v_2)$, and $E(v_1) = E(v_2)$.*

2. *For every node, $f_T \neq f_E$.*

3. *All internal nodes are descendants of some node in Φ.*

Proof. For this proof it is convenient to reverse the established use of having the smallest variable of the order at the top of the BDD. We suppose therefore that the variables in F are $x_1 < x_2 < \cdots < x_n$ with x_n closest to the function node. The proof is by induction on n. If $n = 0$, F is a vector of constant functions whose representation is clearly unique because the BDD contains only one terminal node. The constant 1 is represented as a regular arc pointing to the terminal node, and the constant 0 is represented as a complement arc pointing to the terminal node. Suppose now that $n > 0$ and all functions of $x_1, \ldots, x_{n-1}$ have unique representations as BDDs. Let f be a component of F. We can write $f = x'_n \cdot f_{x'_n} + x_n \cdot f_{x_n}$. Both $f_{x'_n}$ and f_{x_n} are uniquely determined by f and x_n and have unique representations. Suppose f does not depend on x_n. Then the representation of f is the representation of $f_{x'_n}$, which is unique (and identical to the representation of f_{x_n}). Indeed the second condition of the theorem prevents the representation of f from containing a node labeled x_n. Suppose now that f does depend on x_n. We need to distinguish the case in which the outgoing arc of the function node for f_{x_n} is regular from the case in which it is complemented. Suppose the arc is regular. Then the representation of f must consist of a regular arc pointing to a node labeled x_n with children representing $f_{x'_n}$ and f_{x_n}. Suppose the arc is complemented. Then the representation of f must consist of a complement arc pointing to a node labeled x_n with children representing $f'_{x'_n}$ and f'_{x_n}. Hence, in both cases it is unique by the induction hypothesis and the first condition of the theorem. The last condition of the theorem simply guarantees that the representation of F consists solely of the representations of its components, which are unique. $\square$

In the following, we only consider BDDs that conform to the requirements of Theorem 2. Note that the restriction that the T arc must be regular is imposed to guarantee canonicity. Suppose we did not impose the restriction. Then from

$$f' = x' \cdot f'_{x'} + x \cdot f'_x$$

we would have two distinct ways to represent f in terms of its cofactors:

$$f = x' \cdot f_{x'} + x \cdot f_x = (x' \cdot f'_{x'} + x \cdot f'_x)'. \tag{2}$$

Instead, we use this equivalence to choose the one form in which the *then* arc is regular.

Evaluation of a function represented according to Definition 3 amounts to counting the number of complement arcs on the path from the function node to the terminal node. An even number of complement arcs signals that the function evaluates to 1.

A consequence of the requirement that the *then* arcs be regular is that the value of a function when all variables evaluate to 1 can be determined by inspection of the arc out of the function node, since no other complement arc can be encountered along the path. This simple observation leads to the following interpretation of the restriction of the *then* arcs. We represent directly only half of the 2^{2^n} functions of n variables: Those that evaluate to 1 when all variables evaluate to 1.

5 Basic Manipulation of BDDs

5.1 Conjunction of BDDs and Related Operations

The usual way of generating new BDDs is to combine existing BDDs with connectives like conjunction (AND), disjunction (OR), and symmetric difference (EX-OR). As a starting point one takes the simple BDDs for the functions $f_i = x_i$, for all the variables in the functions of interest.[2] We are therefore interested in an algorithm that, given BDDs for f and g, will build the BDD for $f \langle op \rangle g$, where $\langle op \rangle$ is a binary connective (a boolean function of two arguments). The basic idea comes—not surprisingly—from the Theorem 1, since:

$$f \langle op \rangle g = x(f_x \langle op \rangle g_x) + x'(f_{x'} \langle op \rangle g_{x'}).$$

So, if x is the top variable of f and g, we can first cofactor the two functions with respect to x and solve two simpler problems recursively, and then create a node labeled x that points to the results of the two subproblems (if such a node does not exist yet; otherwise we just return the existing node).[3]

Finding the cofactors of f and g with respect to x is easy: If f does not depend on x, $f_x = f_{x'} = f$, that is, the cofactors are the function itself. If, on the other hand, x is the top variable of f, the two cofactors are the two children of the top node of f. Similarly for g.

5.1.1 Conjunction

The algorithm that takes f, g, and $\langle op \rangle$ as arguments and returns $f \langle op \rangle g$ is called *Apply* in the literature. We now examine in detail the special case that computes the conjunction of two BDDs. The pseudo-code is shown in Figure 3. As mentioned in Section 2.3, the application of De Morgan's laws incurs negligible overhead thanks to the complement arcs. Therefore all boolean functions of two operands can be computed efficiently given procedures that compute the conjunction and the symmetric difference of two operands. This is indeed the approach followed in many packages because the specialized algorithms are more efficient than the general one.

The algorithm of Figure 3 employs two major data structures. One is the *unique table* discussed in Section 2.2. Its purpose is to guarantee the canonicity of the BDDs. The second data structure is the *computed table*, which stores results of previous computations so that repeated computations may be avoided. The computed table is often referred to as the *cache*. We will not use this name to avoid confusion with the hardware cache of the computed on which the BDD application runs.

The terminal cases for the recursion are:

case	result
$f = 0$	0
$g = 0$	0
$f = g$	f
$f = 1$	g
$g = 1$	f
$f = g'$	0

[2] The function $f_i = x_i$ is called a *projection* function.

[3] Here for simplicity we ignore the treatment of complement arcs. It is almost always the case that once an algorithm has been devised to operate on BDDs without complement arcs, it can be easily extended to operate on BDDs with complement arcs by applying Equation 2.

```
AND(f, g) {
        if (terminal case) {
                return result;
        } else if (computed table has entry {f, g}) {
                return result;
        } else {
                let x be the top variable of {f, g};
                t = AND(f_x, g_x);
                e = AND(f_x', g_x');
                if (t equals e) r = t;
                else r = findOrAddUniqueTable(x, t, e);
                insertComputedTable({f, g}, r);
                return r;
        }
}
```

Figure 3: Pseudo-code for the conjunction algorithm.

All these conditions can be tested in constant time. If none of them prevails the computed table is consulted. Two observations are in order here. First, the results of trivial problems—the terminal cases—are not stored in the computed table, because memory is a precious resource which should not wasted to record results that can be recomputed in about the same time that it would take to look them up. (Or less, in the case of a hardware cache miss caused by the memory access necessary to read the computed table.) Second, the lookup is for the *set* $\{f, g\}$. In other words, the order of the operands is immaterial because conjunction is commutative.

If the computed table lookup fails to return a result, the procedure proceeds to apply the expansion theorem (Equation 1) and solve the problem recursively. It then applies the reduction rules to the partial result by making sure that no nodes with two identical children and no duplicates of existing nodes are ever created. Finally, the result is stored in the computed table and is returned to the caller. The details of handling complement arcs are not shown in Figure 3, but are analogous to those discussed in the proof of Theorem 2.

Example 2 Consider f and g of Figure 4. The computation of $h = f \cdot g$ proceeds as follows.

$$\begin{aligned}
\text{AND}(f, g) &= (a, \text{AND}(p, t), \text{AND}(q', 0)) \\
&= (a, (b, \text{AND}(1, 1), \text{AND}(0, u')), \text{AND}(q', 0)) \\
&= (a, (b, 1, 0), 0) \\
&= (a, p, 0) \quad \text{unique table lookup.}
\end{aligned}$$

Notice that before creating a node $(b, 1, 0)$ the procedure looks up the unique table. In this case the node is already part of f. Therefore, of the three nodes of h (two internal plus the terminal node—the function node is not usually included in the node count) only one needs to be created.

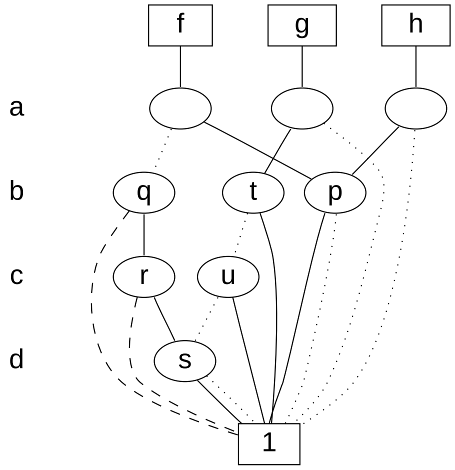


Figure 4: BDDs for Example 2.

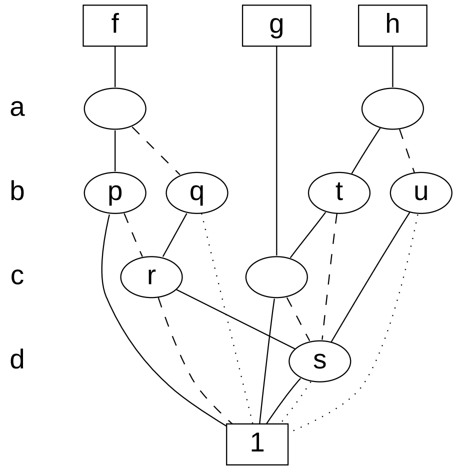


Figure 5: BDDs for Example 3.

Example 3 Consider f and g of Figure 5. These functions are intended to demonstrate the use of the computed table. There are two paths in f from the root to node r. As a result there are two recursive calls with operands r and g. The second benefits from the cached result of the first. The computation of $h = f \cdot g$ proceeds as follows.

$$
\begin{aligned}
\text{AND}(f, g) &= (a, \text{AND}(p, g), \text{AND}(q, g)) \\
&= (a, (b, \text{AND}(1, g), \text{AND}(r, g)), \text{AND}(q, g)) \\
&= (a, (b, g, \text{AND}(r, g)), \text{AND}(q, g)) \\
&= (a, (b, g, (c, \text{AND}(s, 1), \text{AND}(1, s))), \text{AND}(q, g)) \\
&= (a, (b, g, (c, s, s)), \text{AND}(q, g)) \\
&= (a, (b, g, s), \text{AND}(q, g)) \qquad s = \text{AND}(r, g) \text{ is cached here} \\
&= (a, (b, g, s), (b, \text{AND}(r, g), \text{AND}(0, g))) \\
&= (a, (b, g, s), (b, s, 0)) \qquad \text{cache lookup.}
\end{aligned}
$$

In this case three nodes are created: $t = (b, g, s)$, $u = (b, s, 0)$, and $h = (a, t, u)$.

The time complexity of procedure AND is readily established by observing that each pair of nodes (u, v) with u in f and v in g is examined at most four times thanks to the computed table. (The factor of four is due to possible paths with different complementation parity reaching u and v.) Hence, the number of recursive calls is less than or equal to $4 \cdot |f| \cdot |g|$, where $|f|$ is the number of nodes in the BDD for f. Since all steps of the procedure, except the recursive calls, take time bounded by a constant, we have established the following result.

Theorem 3 *Procedure AND runs in time $O(|f| \cdot |g|)$.*

Though the bound is tight, it is seldom achieved in practice if the order of the variables is reasonable. It is important to observe that without the computed table, the runtime of procedure AND would grow exponentially with the number of variables. On the other hand, one may question the practicality of storing all intermediate results of very large computations. Indeed, in most implementations the computed table has fixed maximum capacity, which depends on the available memory and the computation at hand, and is managed as a *hash-based cache*. This implies that each new result is stored in an entry of the table determined by computing a hash function. Various replacement policies can be used to determine what results, if any, are evicted from the table to make room for the new one. In the simplest scheme, each entry holds exactly one result, and every conflict result in the eviction of the older result. This simple scheme works well provided enough memory is available, and provided hashing works well. (Something not to be taken for granted when the computed table has several million entries.)

In summary, the worst case performance of practical implementations of AND is exponential in the number of variables, but the exponential behavior is seldom observed. Similar remarks apply to most algorithms that operate on BDDs. Almost invariably they use a computed table, and have runtimes that are linear in the size of each operand if a lossless computed table is used. It should be noted also that in many implementations the computed table is not flushed once a top-level call to AND terminates. This proves especially advantageous in some model checking experiments.

5.1.2 The If-Then-Else Operator

The approach based on the expansion theorem can be applied also to the *if-then-else* operator (*ite*)—a ternary operator defined as follows:

$$ite(f, g, h) = f \cdot g + f' \cdot h,$$

where f, g, h are three boolean functions. One interesting property of the ITE operator is that all two-argument operators can be expressed in terms of it. For instance, $f' + g' = ite(f, g', 1)$. A detailed analysis of *ite* can be found in [5].

5.2 Satisfiability Problems with BDDs

The satisfiability problem is the problem of deciding whether there is an assignment to the variables of a function that makes the function 1. We have seen that this problem is trivial when the function is given as a BDD, because it is sufficient to check whether the BDD is the constant 0 function and that takes constant time. A related, equally simple problem, is tautology. Checking whether a function is a tautology also takes constant time[4]. In this section we consider related problems, namely:

- Finding one satisfying assignment;
- Counting the number of satisfying assignments;
- Finding one minimum-cost satisfying assignment (this is also known as binate covering problem).

All these problems have in common the correspondence between satisfying assignments and paths in the BDD that go from a function node to the constant node. With complement arcs, all paths lead to the unique constant node and what determines the value associated with a path is its *complementation parity*. If a path contains an even number of complement arcs then its associated value is 1, otherwise it is 0.

Multiple assignments may correspond to the same path. This occurs whenever one or more variables do not appear along the path. The path actually identifies a *cube* of the function, that is the conjunction of some variables or their complements. The variables that appear in the cube are those encountered along the path. A variable appears complemented in the cube if and only if the path goes through an *else* arc coming out of a node labeled with that variable.

5.2.1 Finding One Satisfying Assignment

This problem corresponds to finding a path with even complementation parity. We present in Figure 6 a recursive algorithm, though a non recursive algorithm is obviously possible. The algorithm actually computes a cube in the ON-set of the function. Procedure *OneSat* is called with three arguments. The first (v) is the top node of the BDD for which a satisfying assignment is sought. The second (p) represents the complementation parity of the path. It is initially 1 if the outgoing arc of the function node is regular and 0 if it is complemented. Finally, the third argument (sat) is an array of n cells (n being the number of variables), initialized to all *don't cares*.

The analysis of the performance of this algorithm is based on the observation that every internal node of the BDD has paths to the terminal node with both parities: Otherwise, the node would correspond to a constant function, which contradicts canonicity. Therefore, the procedure may only backtrack at the nodes for which the first child examined is the terminal node. Backtrack occurs in such a case if the parity is odd

[4]Clearly, we assume that the BDD has been built already—a non negligible assumption.

```
OneSat(v, p, sat) {
        if (v is terminal node) return p;
        sat[v → index] = 1;
        if (OneSat(v → T, p, sat)) return 1;
        sat[v → index] = 0;
        if (v → E is complemented) complement p;
        return OneSat(v → E, p, sat);
}
```

Figure 6: Algorithm to find one satisfying assignment.

```
SatHowMany(v, n) {
        if (v is terminal node) return 2^n;
        if (v is in table) return result from table;
        countT = SatHowMany(v → T, n);
        countE = SatHowMany(v → E, n);
        if (v → E is complemented) countE = 2^n − countE;
        count = (countT + countE)/2;
        insert (v, count) in table;
        return count;
}
```

Figure 7: Algorithm to compute the number of satisfying assignments.

($p = 0$). However, in that case the procedure is guaranteed to succeed for the other child. Hence the total number of nodes visited is at most $2n + 1$ and the complexity is $O(n)$.

5.2.2 Counting the Number of Satisfying Assignments

This problems amounts to computing the number of minterms in the ON-set of a function. The result changes with the number of variables on which the function depends. For instance, $f(x_1, x_2) = x_1 x_2$ has one satisfying assignment, whereas $g(x_1, x_2, x_3) = x_1 x_2$ has two. Therefore, the computation of the number of satisfying assignments takes the number of variables as one of its arguments.

The algorithm we outline here is based on the post-order traversal of the BDD. For every node (arc) we compute the number α_v (α_e) of satisfying assignments for the function represented by that node (arc). We assume that the number of variables is n, the index of a function node is 0 and the index of the terminal node is $n + 1$. If v is the terminal node $\alpha_v = 2^n$. For a regular arc connecting node v to node w, $\alpha_e = \alpha_w$, whereas, for a complement arc, $\alpha_e = 2^n - \alpha_w$. Letting T and E be the two arcs emanating from an internal node v, we have:

$$\alpha_v = \frac{\alpha_T + \alpha_E}{2}.$$

This can be seen by noting that the two terms of the expansion with respect to x_i are disjoint and the cofactors do not depend on x_i. Figure 7 gives the pseudo-code of a procedure based on these remarks. The two parameters are the top node of the BDD and the number of variables. From the efficiency point of view, notice that without a table the procedure would take $O(2^n)$ time, whereas the table makes the complexity

$O(n)$, assuming additions and multiplications can be done in constant time.[5] The table is not normally saved across top level calls to this procedure. We need a top level procedure—not shown here—to perform the initialization of the table and also to take care of the possible complement attribute of the function pointer.

5.2.3 Finding One Minimum-Cost Satisfying Assignment

In this problem variable x_i has a non-negative cost c_i associated with it. The cost of an assignment is the sum of the costs of all the variables set to 1. For instance, $x_1 x_2' x_3$ is a satisfying assignment for $f(x_1, x_2, x_3) = x_1 x_2' + x_2 x_3$. Its cost is $c_1 + c_3$. This assignment is not of minimum cost unless $c_3 = 0$, because $x_1 x_2' x_3'$ is also a satisfying assignment of cost $c_1 \leq c_1 + c_3$.

When f is represented by a BDD F, the following result [40] allows the solution of the problem in time linear in the size of F.

Definition 4 *The length of an E arc is 0 and the length of a T arc out of a node labeled x_i is c_i.*

Theorem 4 *Let F be a BDD for $f(x_1, \ldots, x_m)$. Then the minimum cost assignment to $x_1, \ldots, x_m$ that is a solution to*

$$f(x_1, \ldots, x_m) = 1$$

is given by the shortest path of even parity connecting the root of F to the terminal node.

Proof. Every path of even parity from the root to the terminal node represents a set of satisfying assignments. By taking all the variables that do not appear along the path as 0, one obtains a minimal assignment corresponding to the path. For this assignment, the cost is equal to the length of the path. (If some costs are 0, there may be several minimal assignments.) Let A be a minimal assignment corresponding to a shortest path P of D. We prove that A is minimum. Suppose there is another assignment A' such that cost(A')<cost(A). A' corresponds to a path P' from the root of F to the terminal node. Then cost(A')≥length(P')≥length(P) = cost(A), a contradiction. □

Note that the length of the shortest path does not depend on the variable ordering chosen.

Example 4 Consider the function

$$f = a(b + c)(b + d + e)(b' + d')(d' + e').$$

The corresponding BDD for the ordering $a \leq \cdots \leq e$ is shown in Figure 8. One can see that the path $a = 1$, $b = 1$, and $d = 0$ (the path goes through the filled nodes) is the shortest path with even parity and that it identifies an assignment of cost 2 (assuming unit costs).

Finding the shortest path in a DAG with bounded outdegree (2 in our case) can be done in $O(n)$ time. (See, for instance, [20, p. 536].) The additional complication arising from the complement arcs does not increase the asymptotic complexity.

[5] This assumption is not superfluous, if we consider the large numbers that may be involved. In practice, one would use floating point numbers or arbitrary precision integer arithmetic to prevent integer overflow. 64-bit floating-point arithmetic may give accuracy problems even for relatively few variables, and does not work for, say, 2000 variables. Arbitrary precision arithmetic violates the assumption that additions and multiplications can be done in constant time. Furthermore, storing an integer with hundreds of digits for each node may cause the computed table to grow too large.

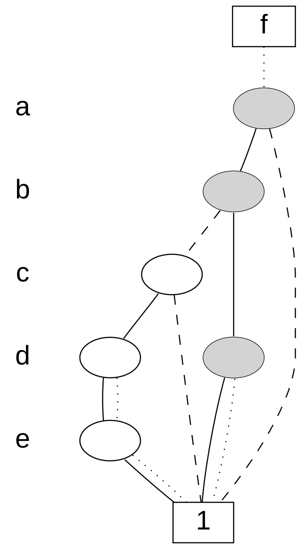


Figure 8: Example BDD for the minimum-cost assignment problem.

5.3 Cofactors, Function Composition, and Quantification

The computation of cofactors is a key ingredient in many BDD-based applications. Other important operations on BDDs, like function composition, and universal and existential quantification, are defined in terms of cofactors. We shall first analyze the algorithms for cofactors with respect to single variables and with respect to cubes in Section 5.3.1.

5.3.1 Cofactors with Respect to Single Variables and Cubes

We know how to efficiently compute the cofactors of a BDD with respect to its top variable—they are just the two children of the top node—and with respect to a variable that does not appear in the BDD. In this last case the cofactors coincide with the function. We use this knowledge to solve the general case. To this end, suppose we want to compute f_c—the cofactor of f with respect to c—where f is an arbitrary function and c is a cube. Let t be the index of the top variable of f and u be the index of the top variable of c. We shall also write $c = \hat{x}_u \tilde{c}$, where $\hat{x}_u$ is either x_u or x'_u and $\tilde{c}$ is a cube, possibly the function 1. We first notice that if $c = 1$ or f is constant, then $f_c = f$. This provides the termination condition. Assuming $c \neq 1$ and f not constant, we consider the three following cases.

$t > u$. In this case f, and f_c as a consequence, do not depend on x_u. (We assume that the indices grow in the BDD from the root to the leaf.) Hence,

$$f_c = f_{\hat{x}_u \tilde{c}} = f_{\tilde{c}}.$$

In practice, we just have to move down along the BDD for c and recur.

$t = u$. In this case we write:

$$f_c = (f_{\hat{x}_t})_{\tilde{c}}.$$

Hence, we cofactor f with respect to $\hat{x}_t$, we move down along c, and recur.

$t < u$. In this case we use the commutativity of cofactors and write:

$$f_c = x_t \cdot (f_{x_t})_c + x'_t \cdot (f_{x'_t})_c.$$

Hence, we need to recursively compute the cofactors of f_{x_t} and $f_{x'_t}$ with respect to c and connect them as *then* and *else* children to a node labeled x_t.

5.3.2 Function Composition

In function composition, we want to compute:

$$f|_{x_i=g} = f(x_1, \ldots, x_{i-1}, g, x_{i+1}, \ldots, x_n),$$

where g is also a function of $x_1, \ldots, x_n$. From the expansion theorem,

$$f = x_i f_{x_i} + x'_i f_{x'_i},$$

we derive by substitution:

$$f|_{x_i=g} = g \cdot f_{x_i} + g' \cdot f_{x'_i}. \tag{3}$$

This implies that we can compose f and g by finding the cofactors of f with respect to x_i and x'_i and then computing

$$\textit{ite}(g, f_{x_i}, f_{x'_i}).$$

We have seen the general method to compute the cofactors in Section 5.3.1. Here we present a more efficient procedure for function composition that only requires the computation of the cofactors with respect to the top variable of a BDD and its complement. As we saw, these cofactors are simply given by the two children of the top node. The algorithm for this case is indeed reminiscent of that for cofactors with respect to cubes, which we saw in Section 5.3.1.

Let t be the index of the top variable of f. We have three mutually exclusive cases.

$t > i$. In this case, f does not depend on x_i and the result of the composition is simply f.

$t = i$. In this case, x_i is the top variable of f and we can apply Equation 3 directly.

$t < i$. In this case, we find the variable of least index between the top variable of f and the top variable of g. Let x_u be that variable: We expand with respect to x_u and recur:

$$\begin{aligned} f|_{x_i=g} &= x_u \cdot (g \cdot f_{x_i} + g' \cdot f_{x'_i})_{x_u} + x'_u \cdot (g \cdot f_{x_i} + g' \cdot f_{x'_i})_{x'_u} \\ &= x_u \cdot (g_{x_u} \cdot (f_{x_u})_{x_i} + g'_{x_u} \cdot (f_{x_u})_{x'_i}) + x'_u \cdot (g_{x'_u} \cdot (f_{x'_u})_{x_i} + g'_{x'_u} \cdot (f_{x'_u})_{x'_i}) \end{aligned}$$

(Note that we used the commutativity of cofactors here.)

5.3.3 Quantification

Given a boolean function $f(x_1, \ldots, x_n)$ of n variables, we define the *existential quantification* of f with respect to x_i as:

$$\exists_{x_i} f = f_{x_i} + f_{x'_i}$$

and the *universal quantification* of f with respect to x_i as:

$$\forall_{x_i} f = f_{x_i} \cdot f_{x'_i}.$$

Example 5 As an example, let us consider the following function:

$$f = x'y'z + xz' + xy.$$

Suppose we want to quantify z in f. The two cofactors are:

$$f_z = x'y' + xy \quad \text{and} \quad f_{z'} = x.$$

Hence,

$$\exists_z f = x + y' \quad \text{and} \quad \forall_z f = xy.$$

If $\exists_{x_i} f$ and $\forall_{x_i} f$ are considered as defined over the same n-dimensional boolean space as f, then it is easy to verify that

$$\forall_{x_i} f \leq f \leq \exists_{x_i} f.$$

More specifically, $\exists_{x_i} f$ is the smallest function independent of x_i that contains f and $\forall_{x_i} f$ is the largest function independent of x_i that is contained in f.

Quantification is monotonic, i.e.,

$$\begin{aligned} f \leq g \quad &\Rightarrow \quad \exists_x f \leq \exists_x g \\ &\Rightarrow \quad \forall_x f \leq \forall_x g \end{aligned}$$

This can be easily seen by writing:

$$f = x f_x + x' f_{x'} \quad \text{and} \quad g = x g_x + x' g_{x'},$$

whence:

$$f \leq g \Rightarrow f_x \leq g_x \wedge f_{x'} \leq g_{x'},$$

from which the stated inequalities can be derived. Quantifications of the same type commute, i.e.,

$$\exists_{x_1 x_2} f = \exists_{x_2}(\exists_{x_1} f) = \exists_{x_1}(\exists_{x_2} f)$$

and

$$\forall_{x_1 x_2} f = \forall_{x_2}(\forall_{x_1} f) = \forall_{x_1}(\forall_{x_2} f).$$

The existential and universal quantifications with respect to a cube are therefore well-defined. However, quantifications of different types do not commute. Existential quantification distributes over disjunction and universal quantification distributes over conjunction, but not vice versa, unless f and g have disjoint support.

$$\exists_x(f + g) = \exists_x f + \exists_x g \qquad \forall_x(f \cdot g) = \forall_x f \cdot \forall_x g$$

$$\forall_x(f + g) \geq \forall_x f + \forall_x g \qquad \exists_x(f \cdot g) \leq \exists_x f \cdot \exists_x g.$$

These properties can be easily verified by substituting the definitions. One can also easily verify the following properties, that are useful when quantifications are performed on BDDs with complement arcs.

$$\exists_x(f') = (\forall_x f)' \qquad \forall_x(f') = (\exists_x f)'.$$

A recursive algorithm for the quantification of the variables of a cube c in a function f is derived as the one for the computation of cofactors.

Example 6 We compute $\exists_z f$, for $f = x'y'z + xz' + xy$ using the variable order $x < y < z$.

$$
\begin{aligned}
\exists_z f &= x \cdot \exists_z f_x + x' \exists_z f_{x'} \\
&= x(y \cdot \exists_z f_{xy} + y' \cdot \exists_z f_{xy'}) + x'(y \cdot \exists_z f_{x'y} + y' \cdot \exists_z f_{x'y'}) \\
&= x(y \cdot \exists_z 1 + y' \cdot \exists_z z') + x'(y \cdot \exists_z 0 + y' \cdot \exists_z z) \\
&= x \cdot 1 + x' \cdot y' = x + y'.
\end{aligned}
$$

Existential quantification is very easy when the function is given in sum of product form. It is indeed sufficient to suppress the quantified variables from each product term. If all variables are suppressed from a term, the term becomes the constant 1. In the previous example, $\exists_z f = x'y' + x + xy = x + y'$. Likewise, universal quantification is easy when f is given in product of sum form. By contrast, quantification of either type may increase the size of a BDD.

5.4 Cofactors with Respect to Functions

A set $\{b_1, \ldots, b_p\}$ of n-variable boolean functions is an orthonormal set if it satisfies:

$$\sum_{i=1}^{p} b_i = 1$$

and

$$b_i \cdot b_j = 0, \quad i \neq j.$$

A common example of orthonormal set is $\{x_i, x_i'\}$: It is used in Boole's expansion theorem. Also, for every function $g(x_1, \ldots, x_n)$, the set $\{g, g'\}$ is orthonormal.

Given an orthonormal set $\{b_1, \ldots, b_p\}$, we can expand a function f with respect to it in the form

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{p} f_i(x_1, \ldots, x_n) \cdot b_i(x_1, \ldots, x_n).$$

The coefficients of the expansion are related to f and $\{b_1, \ldots, b_p\}$ by the following result.

Theorem 5 *Let $\{b_1, \ldots, b_p\}$ an orthonormal set of n-variable functions and f an n-variable function. Then*

$$f = \sum_{i=1}^{p} f_i \cdot b_i$$

if and only if

$$f \cdot b_i = f_i \cdot b_i, \quad i = 1, \ldots, p.$$

Proof. Suppose $f \cdot b_i = f_i \cdot b_i$. Then

$$\sum_{i=1}^{p} f_i \cdot b_i = \sum_{i=1}^{p} f \cdot b_i = f \cdot \sum_{i=1}^{p} b_i = f.$$

Conversely, if $f = \sum_{i=1}^{p} f_i \cdot b_i$, then

$$f \cdot b_i = (\sum_{j=1}^{p} f_j \cdot b_j) b_i = f_i \cdot b_i.$$

This proves the theorem. □

It should be noted that this theorem does not define uniquely the coefficients of the expansion, but imposes restrictions on what they can be. In other words, the coefficients are incompletely specified. It is possible to prove that every function in the interval

$$[f \cdot b_i, f + b_i']$$

satisfies the condition of the theorem for f_i. This can also be interpreted by saying that f_i has to agree with f wherever b_i is 1 and is *don't care* otherwise. We shall see in the rest of this section how to exploit this freedom to obtain compact representations of the coefficients.

5.4.1 The Constrain Operator

We can now define the cofactors of a function with respect to another function, in terms of orthonormal expansions.

Definition 5 *Let f and g be two boolean functions and let*

$$f = g \cdot f_g + g' \cdot f_{g'}$$

be an expansion of f with respect to the orthonormal set $\{g, g'\}$. Then f_g ($f_{g'}$) is a positive (negative) generalized *cofactor of f with respect to g.*

The generalized cofactors are incompletely specified by their defining equality. Hence, the problem arises of finding the best cofactors according to some criterion. A common criterion is the size of the representations, that is, when BDDs are used, the number of nodes.

As we saw, the values of f_g are fixed wherever $g = 1$. The problem is therefore to choose the values of f_g where $g = 0$. Coudert *et al.* [21] introduced an operator on BDDs, called *constrain* that normally returns compact cofactors. The strategy of *constrain* is to map each minterm in the offset of g into a minterm of the onset of g and use this map to define the values of f_g wherever $g = 0$. The map depends on g and on the following definition of distance.

Definition 6 *Let $x_1, \ldots, x_n$ be variables with ordering*

$$x_1 \leq \ldots \leq x_n.$$

Let $r = (r_1, \ldots, r_n)$ and $s = (s_1, \ldots, s_n)$ be two minterms. The distance between r and s, written $\| r - s \|$, is given by:

$$\| r - s \| = \sum_{i=1}^{n} |r_i - s_i| 2^{n-i}.$$

This definition of distance reflects the dissimilarity of the paths associated to the two minterms in the BDD with ordering $x_1 \leq \cdots \leq x_n$. We are now ready to define the *constrain* operator.

Definition 7 *For functions f and g, the function f constrained by g, written $f \downarrow g$ is defined by*

$$(f \downarrow g)(r) = \begin{cases} f(r) & \text{if } g(r) = 1 \\ f(s) & \text{if } g(r) = 0 \end{cases}$$

where s is the minterm such that $g(s) = 1$ and $\| r - s \|$ is minimum.

Notice that $s \neq t$ implies $\| r - s \| \neq \| r - t \|$. Therefore, s is uniquely identified in Definition 6. The algorithm for *constrain* visits recursively the BDDs for f and g. The recursion is based on the following theorem:

Theorem 6 *Let f, g, and h be boolean functions. Let*

$$\hat{f}_g = h(f_h)_{g_h} + h'(f_{h'})_{g_{h'}} \quad \textit{and} \quad \hat{f}_{g'} = h(f_h)_{g'_h} + h'(f_{h'})_{g'_{h'}}.$$

Then:

$$f = g \cdot \hat{f}_g + g' \cdot \hat{f}_{g'}.$$

Proof. We need to prove that $\hat{f}_g$ and $\hat{f}_{g'}$ satisfy Theorem 5. We consider $\hat{f}_g$, since the case for $\hat{f}_{g'}$ is similar.

$$\begin{aligned}
\hat{f}_g \cdot g &= g \cdot h \cdot (f_h)_{g_h} + g \cdot h' \cdot (f_{h'})_{g_{h'}} \\
&= h \cdot g_h \cdot (f_h)_{g_h} + h' \cdot g_{h'} \cdot (f_{h'})_{g_{h'}} \\
&= h \cdot g_h \cdot f_h + h' \cdot g_{h'} \cdot f_{h'} \\
&= h \cdot g_h \cdot f + h' \cdot g_{h'} \cdot f \\
&= f \cdot (h \cdot g + h' \cdot g) = f \cdot g.
\end{aligned}$$

□

We choose h as the variable with the lowest index among those appearing in f and g, so that we can apply cofactoring easily. Note that the straightforward expansion yields an expression that is unsuitable for the recursive formulation:

$$f_g = h(f_g)_h + h'(f_g)_{h'}.$$

We cannot simply say that $(f_g)_h = (f_h)_{g_h}$, because we are dealing with incompletely specified functions.

The recursion has several termination cases. If f is constant, $f \downarrow g = f$. If $f = g$ we return 1 and if $f = g'$ we return 0. These are correct solutions in general and will be seen to be the only ones satisfying Definition 7. If $g = 1$, then the only solution is $f \downarrow g = f$. If $g = 0$ and we are at the top-level of the recursion, any solution is acceptable. We choose to return 0, for reasons that will become apparent later. We take care of avoiding recursive calls with $g = 0$ as follows.

Whenever $g_h = 0$ or $g_{h'} = 0$, we have identified a group of minterms of f that need to be mapped. We consider the case for $g_{h'} = 0$. We compute $(f_h)_{g_h}$ first and then perform the mapping by taking $(f_{h'})_{g_{h'}} = (f_h)_{g_h}$. Since the two cofactors are the same, we don't create a new node and just return $(f_h)_{g_h}$. Similarly, if $g_h = 0$ we return $(f_{h'})_{g_{h'}}$.

Example 7 An example of application of *constrain* is illustrated in Figure 9.

$$\begin{aligned}
f \downarrow g &= (x, z \downarrow q, p' \downarrow r') \\
&= (x, (y, z \downarrow 1, z \downarrow z'), z \downarrow z') \\
&= (x, (y, z, 0), 0) \\
&= xyz
\end{aligned}$$

This example demonstrates some of the special cases. In particular, $g_{x'y} = 0$ and we return the result from the other branch.

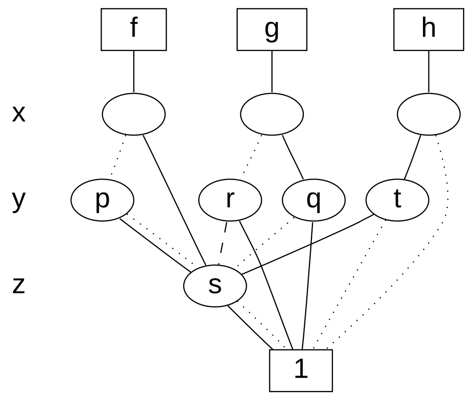


Figure 9: BDDs for Example 7.

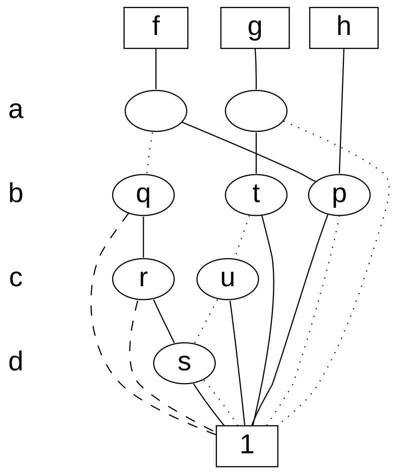


Figure 10: BDDs for Example 8.

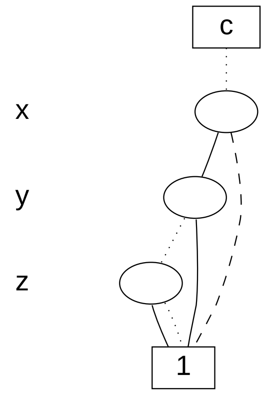


Figure 11: BDD for $c = xy'z$.

Example 8 For another example, let us consider f and g of Figure 10. We have:

$$\begin{aligned} f \downarrow g &= (a, p \downarrow t, q' \downarrow 0) \\ &= p \downarrow t \\ &= (b, 1 \downarrow 1, 0 \downarrow u') \\ &= (b, 1, 0) \\ &= b. \end{aligned}$$

It should be clear that the choices we have made in our algorithm are aimed at producing a result as small as possible. It remains to be proved that the algorithm we have outlined actually conforms to Definition 7. We shall outline a possible proof. We consider first the case when $f = g$. All minterms for which $g = 1$ cause f to be 1 as well. Hence, minterm s of Definition 7 must cause f to be 1. This is why we return 1. The case for $f = g'$ is similar.

Suppose next that $g_{h'} = 0$ at some point of the recursion. Let c be the cube of the variables on which splitting has occurred up to that point. We have to find the minterms with $g = 1$ whose distance from the minterms in the cube $c \cdot h'$ is minimum. We first notice that these minterms must be in $c \cdot h$. This is because the weights in the definition of distance decrease exponentially from the root to the leaf. Hence, there is a minterm in $c \cdot h$ closer to each minterm in $c \cdot h'$ than every other minterm in the rest of the BDD. Take $r = c \cdot h' \cdot t$. If $\tilde{r} = c \cdot h \cdot t$ is such that $g = 1$, then $s = \tilde{r}$. Otherwise, s will be the minterm such that $g = 1$ closest to $\tilde{r}$. But this is exactly what our algorithm does.

Figure 11 shows a BDD for the cube $c = xy'z$. It illustrates the typical structure of the BDD of any cube: It is a single string of nodes. All internal nodes except the one at the bottom have one arc pointing to 0 (pointing to 1 with odd parity). Because of this, the *constrain* algorithm always returns the result of one branch and it ends up computing exactly the ordinary cofactor. The effects of the use of the computed table on the complexity of *constrain* are the same as for AND.

We complete our discussion of *constrain* by discussing a problem it has and a possible optimization. Though in general the BDD for $f \downarrow g$ has fewer nodes than the BDD for f, sometimes the reverse is true. This most often occurs when the BDD for g is large and depends on many variables f does not depend on. These variables may be introduced in $f \downarrow g$, causing an undesirable growth of the BDD. This inconvenience can sometimes be avoided in a rather inexpensive way by existentially quantifying the undesired variables from g (we shall see how in Section 5.4.2), but the resulting operator loses one property that is important

```
restrict(F, G) {
    if (G = 1 or F is constant) return F;
    if (F = G) return 1;
    if (F = G') return 0;
    if (G = 0) return 0;
    let v be the top variable of {F, G};
    if (G_v = 0) return restrict(F_v', G_v');
    if (G_v' = 0) return restrict(F_v, G_v);
    if (v is not the top variable of F)
        return restrict(F, ite(G_v, 1, G_v'));
    return ite(v, restrict(F_v, G_v), restrict(F_v', G_v'));
}
```

Figure 12: Procedure *restrict* (without computed table).

for FSM verification, namely, being an image restrictor. For this reason, the 'un-optimized version' is also of interest and we have presented it here.

5.4.2 The Restrict Operator

In Section 5.4.1 we noted that the *constrain* operator may introduce variables that do not appear in f into $f \downarrow g$. We now see how we can use existential quantification to solve the problem. We need the following lemma.

Lemma 1 *Let f, g, and h be n-variable boolean functions and suppose $h \geq g$. Then*

$$g \cdot f_g = g \cdot f_h.$$

In other words, a cofactor computed with respect to h is also a cofactor with respect to g.

Proof. It is sufficient to observe that a function must agree with f, whenever $g = 1$, in order to be a cofactor with respect to g. This is certainly the case of f_h, since $g = 1 \Rightarrow h = 1$. □

Lemma 1 applies in particular to the *constrain* operator and to $h = \exists_{x_i} g$. Hence, we can existentially quantify all the variables in g that do not appear in f before computing $f \downarrow g$. Let h be the result of these quantifications. Then $f \downarrow h$ is a valid cofactor of f with respect to g.

It is also possible to integrate the quantification into the computation of *constrain*. The resulting algorithm is called *restrict* in [22] and indicated by $f \Downarrow g$. Whenever the top variable of g has a lower index than the top variable of f, the procedure returns

$$f \Downarrow (\exists_{x_t} g).$$

The pseudo-code for *restrict* is given in Figure 12. Note that the call $ite(G_v, 1, G_{v'})$ quantifies v in G. Also, some implementations prefer to return a failure value if $G = 0$. This may help catch bugs in the program, but may sometimes be restrictive. Notice that *restrict* may actually produce smaller BDDs than the approach previously described (quantification followed by *constrain*). It is indeed possible to quantify variables, which only some cofactors of f are independent of, from the corresponding cofactors of g. On the other hand, quantifications are performed one at the time and performance may be substantially slower, if *constrain* uses a cache table and *restrict* does not.

Example 9 We return now to the example of Figure 9 and we compute $f \Downarrow g$.

$$\begin{aligned} f \Downarrow g &= (x, f_x \Downarrow g_x, f_{x'} \Downarrow g_{x'}) \\ &= (x, z \Downarrow q, p' \Downarrow r') \\ &= (x, z \Downarrow (\exists_y q)), z \Downarrow z') \\ &= (x, z \Downarrow 1, 0) \\ &= (x, z, 0) \\ &= xz. \end{aligned}$$

One can see that in this case $f \Downarrow g$ has fewer nodes than $f \downarrow g$, because we avoided introducing a node labeled y in the positive cofactor with respect to x. In general, however, it may still be the case that $f \Downarrow g$ has more nodes than f. In that case, one may decide to return f as the result of cofactoring.

5.4.3 Properties of Generalized Cofactors

In the following we discuss properties of generalized cofactors that are useful in sequential verification. We use ∇ to indicate a generic operator.

Lemma 2 *Let $f : B^{m+n} \rightarrow B$ and $g : B^n \rightarrow B$ be boolean functions, $x = (x_1, \ldots, x_n)$, and $y = (y_1, \ldots, y_m)$. Let ∇ be an operator satisfying:*

$$(f \nabla g) \cdot g = f \cdot g,$$

and

$$\exists_x (f(x, y) \nabla g(x)) \leq \exists_x (f(x, y) \cdot g(x)). \tag{4}$$

Then:

$$\exists_x (f(x, y) \cdot g(x)) = \exists_x (f(x, y) \nabla g(x)).$$

Proof. From $(f \nabla g) \cdot g = f \cdot g$ (i.e., ∇ is a generalized cofactor) we get $f \cdot g \leq f \nabla g$. Since the quantification operators are monotonic, we have:

$$\exists_x (f(x, y) \cdot g(x)) \leq \exists_x (f(x, y) \nabla g(x)).$$

The reverse inequality follows directly from (4). □

In this lemma, ∇ is a generalized cofactor operator that satisfies (4). We are obviously interested in seeing if the lemma applies to the cofactoring operators that we use. We recall that the *constrain* operator is defined in terms of a mapping:

$$(f \downarrow g)(r) = \begin{cases} f(r) & \text{if } g(r) = 1 \\ f(s) & \text{if } g(r) = 0 \end{cases}$$

where s is the minterm such that $g(s) = 1$ and $\| r - s \|$ is minimum. Let μ_g be the mapping defined by this rule. Then:

$$(f \downarrow g)(x) = f(\mu_g(x)).$$

Let $f : B^{m+n} \rightarrow B$ and $g : B^n \rightarrow B$. In words, g does not depend on some of the variables in the support of f. We denote by μ^x the restriction of the mapping to the variables in x. It is easily seen that:

$$\mu_g((x, y)) = (\mu_g^x(x), y),$$

because the distance between (x, y) and $\mu_g((x, y))$ must be minimal. With these preliminaries, we can now prove the following result.

Theorem 7 *Let $f : B^{m+n} \to B$ and $g : B^n \to B$ be boolean functions, $x = (x_1, \ldots, x_n)$, and $y = (y_1, \ldots, y_m)$. Then:*

$$\exists_x (f(x, y) \cdot g(x)) = \exists_x (f(x, y) \downarrow g(x)).$$

Proof. Suppose $f(\tilde{x}, \tilde{y}) \downarrow g(\tilde{x}) = 1$. Then:

$$f(\mu_g^x(\tilde{x}), \tilde{y}) = 1, \quad \text{and} \quad g(\mu_g^x(\tilde{x})) = 1,$$

by the definitions of $f \downarrow g$ and of μ_g^x. Hence $\mu_g^x(\tilde{x})$ is the x sought in (4) and Lemma 2 applies. □

We can use the fact that *restrict* is related to *constrain* to prove a result similar to Theorem 7 for the former. Specifically,

$$f \Downarrow g = f \downarrow \tilde{g},$$

Where $\tilde{g}$ is a function obtained by quantifying variables in some cofactors of g. From the definition of *restrict*, $\tilde{g} \geq g$, and the support of $\tilde{g}$ is a subset of that of g. The following property of $\tilde{g}$ is of particular interest.

Lemma 3 *Let $f : B^n \to B$ and $g : B^n \to B$ be boolean functions, $x = (x_1, \ldots, x_n)$. Let $a = (a_1, \ldots, a_n)$, $a_i \in \{0, 1\}$. Let $\tilde{g} : B^n \to B$ be such that:*

$$f \Downarrow g = f \downarrow \tilde{g}.$$

If $\tilde{g}(a) > g(a)$, then there exists $b \in B^n$ such that:

$$\tilde{g}(b) = g(b) = 1 \quad \textit{and} \quad f(a) = f(b).$$

Proof. Let

$$\hat{x}_i = \begin{cases} x_i & \text{if } a_i = 1 \\ x_i' & \text{if } a_i = 0 \end{cases}$$

Since $\tilde{g}$ is obtained by existential quantification of some variables in some cofactors of g, $\tilde{g}(a) > g(a)$ implies that there is a subset of variables that were quantified along the path that includes a. Let this subset be $J = \{x_{j_1}, \ldots, x_{j_p}\}$, let $K = \{x_1, \ldots, x_n\} - J$, and let $c = \prod_{i \in K} \hat{x}_i$. Notice that $a \leq c$. If we cofactor f and $\tilde{g}$ with respect to c, the resulting functions, $f_c, \tilde{g}_c$ are constants. For f, this derives from the requirement that the quantification is performed only if the current cofactor of f does not depend on the current variable. For $\tilde{g}$, it follows from the quantifications performed on it. Furthermore, g must be 1 for at least one minterm of c, or $\tilde{g}(a) = \tilde{g}_c$ would be 0. Hence, there is a minterm $b \leq c$ such that $f(b) = f(a)$ and $g(b) = 1$. □

Lemma 3 allows us to prove the analog of Theorem 7 for *restrict*.

Theorem 8 *Let $f : B^{m+n} \to B$ and $g : B^n \to B$ be boolean functions, $x = (x_1, \ldots, x_n)$, and $y = (y_1, \ldots, y_m)$. Then:*

$$\exists_x (f(x, y) \cdot g(x)) = \exists_x (f(x, y) \Downarrow g(x)).$$

Proof. Let $\tilde{g}(x)$ be the function such that $(f \Downarrow g)(x, y) = (f \downarrow \tilde{g})(x, y)$. Then,

$$(f \Downarrow g)(\tilde{x}, \tilde{y}) = 1 \Leftrightarrow (f \downarrow \tilde{g})(\tilde{x}, \tilde{y}) = 1.$$

Then, with argument similar to the proof of Theorem 7,

$$f(\mu_{\tilde{g}}^x(\tilde{x}), \tilde{y}) = 1, \quad \text{and} \quad \tilde{g}(\mu_{\tilde{g}}^x(\tilde{x})) = 1.$$

Let $a = (\mu_{\tilde{g}}^{x}(\tilde{x}), \tilde{y})$. If $g(a) = 1$, we are done. Suppose $g(a) = 0$. Then, by Lemma 3, there exists b such that $\tilde{g}(b) = g(b) = 1$ and $f(b) = f(a) = 1$. Hence b is the x sought in (4) and Lemma 2 applies. □

Notice that it is possible to build generalized cofactors that do not satisfy (4). Each operator has a distinct advantage over the other. On the one hand, *constrain* distributes over product, as shown by the following result [59], where $\circ$ denotes composition.

Lemma 4 *Let $f : B^m \to B$, $g : B^n \to B^m$, and $h : B^n \to B$ be boolean functions. Then:*

$$(f \circ g) \downarrow h = f \circ (g \downarrow h).$$

Proof.

$$\begin{aligned}
(f \circ g)(x) \downarrow h(x) &= (f \circ g)(\mu_h(x)) \\
&= f \circ g(\mu_h(x)) \\
&= f \circ (g \downarrow h).
\end{aligned}$$

□

In particular, by taking $f = g_1 \cdot g_2$ one obtains distributivity with respect to product and can write the following equation that we shall use in Section 8.3.

$$\begin{aligned}
&\exists_x((\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \cdot g(x_1, \ldots, x_n)) = \\
&\quad = \exists_x(\prod_{i=1}^{m}((y_i \equiv f_i(x_1, \ldots, x_n)) \downarrow g(x_1, \ldots, x_n))).
\end{aligned} \tag{5}$$

Unfortunately, *restrict* does not enjoy the same property, due to the different mappings applied to each g_i. For instance, for $g_1 = x_2$, $g_2 = x_1 x_2'$, and $h = x_1' + x_2'$, we have $(g_1 \cdot g_2) \Downarrow h = 0$, but $(g_1 \Downarrow h) \cdot (g_2 \Downarrow h) = x_1 x_2$. Therefore we can write:

$$\exists_x((\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \cdot g(x_1, \ldots, x_n)) =$$

$$= \exists_x((\prod_{i=1}^{m}(y_i \equiv f_i(x_1, \ldots, x_n))) \Downarrow g(x_1, \ldots, x_n)) \tag{6}$$

$$= \exists_x(\prod_{i=1}^{m}((y_i \equiv f_i(x_1, \ldots, x_n)) \Downarrow g(x_1, \ldots, x_n)) \cdot g(x_1, \ldots, x_n)). \tag{7}$$

However, we cannot simultaneously apply *restrict* to the terms of the product and eliminate it, as we do with *constrain*. On the other hand, *constrain* may introduce in $f \downarrow g$ variables that are in g, but not in f. This is likely to occur when g is a complicated expression. This is a problem, especially for the method of Section 8.5.1, that *restrict* does not have.

5.5 Minimization

The *constrain* and *restrict* operators described the previous sections are designed to produce small resulting BDDs. For this reason they have been extensively used for the purpose of BDD minimization. Suppose we are given an interval $[l, u]$ and that we are asked for a function f such that $l \leq f \leq u$ and the BDD of f is as small as possible. The problem can be solved by computing $C = l + u'$, the *care set* of f, and then computing f itself as $l \downarrow C$ or as $l \Downarrow C$. Any operator that computes f_g could be used instead of *constrain* or *restrict*, and any function from $[l, u]$ can be used instead of l.[6]

Normally *restrict* performs better than constrain in this capacity because it does not pollute the support of the result, but there are exceptions. Besides, neither *constrain* nor *restrict* guarantees that the BDD for f is smaller than the BDD for l. Indeed, when the BDD for C is considerably larger than the BDD for l, it is often the case that the "minimized" BDD is larger than the original one. In the terminology of [33], these operators are not *safe*. The lack of safeness is easily explained. Suppose that in computing $l \downarrow C$ a node l_1 of l is visited twice—first in conjunction with node c_1 of C, and then in conjunction with node c_2 of C. Since the constraints are different in the two recursive calls originating at l_1, different results may be returned. The result may therefore contain more than one node corresponding to l_1. The *compaction* algorithm of [33] avoids this problem by performing minimization in two steps. In the first step the BDDs are analyzed and the cases in which duplication may not occur are identified. In the second step, safe minimization is carried out according to the information gathered in the first step. The *compaction* algorithm does produce on average better results than either *constrain* or *restrict*, but it is considerably slower.

Another approach to improving BDD minimization relies on extending the remapping mechanism of *constrain*. Consider the computation of $f \downarrow g = (x, f_x, f_{x'}) \downarrow (x, g_x, g_{x'})$. Simplification of the result occurs when either $g_x = 0$ or $g_{x'} = 0$. In such cases the result is either $f_{x'}$ or f_x. One can do better by observing that if $f_x \oplus f_{x'} \leq g'_x + g'_{x'}$ then there are solutions independent of x for f_g. One such solution is given by minimizing $f_x g_x + f_{x'} g_{x'}$ with the care set $g_x + g_{x'}$. The resulting procedure is more powerful than *restrict*, but much more expensive. Approaches intermediate in power are often preferable. Details and assessment can be found in [58]. Our discussion has been in terms of a given function to be minimized, and a *care set*, so that we could easily relate minimization to *constrain* and *restrict*; however, minimization algorithms can be directly formulated in terms of the interval $[l, u]$.

5.6 Approximation

BDD approximation is the problem of deriving from a given BDD another BDD smaller in size, and whose function is at a low Hamming distance from the input BDD. (That is, differing from the input BDD in a small number of input assignments.) Let $\alpha(f)$ be the BDD produced by the application of approximation algorithm α to the BDD of f. Usually, the function of the approximating BDD is required to be either a subset or a superset of the input function, that is,

$$\alpha(f) \leq f \quad \text{or} \quad \alpha(f) \geq f.$$

For an *underapproximation* algorithm α (such that $\alpha(f) \leq f$), $\neg\alpha(\neg f) \geq f$. Hence, we only discuss underapproximation.

Underapproximation algorithms must trade off the size of the result for the distance from f. The two trivial solutions, 0 and f itself, are seldom useful. A natural way to rank different approximations is by their density, denoted by $\delta(\alpha(f))$ and defined in [53] by:

$$\delta(g) = \frac{\|g\|}{|g|}.$$

($\|g\|$ is the number of minterms of g.) High density corresponds to a concise representation, and is therefore desirable. (For overapproximation, we want to maximize the density of the complement of the result.)

[6] In the case of *constrain*, replacing l with any function in $[l, u]$ does not affect the result.

The decision version of the underapproximation problem can be stated as follow: Given the BDD of a boolean function f of n variables for variable order π, and integers m and k, find a function $g \leq f$ of the same variables that is true for at least m assignments to the n variables, and such that its BDD for variable order π has at most k nodes. This problem is conjectured to be NP-complete. Hence, heuristic techniques are used to solve it. The algorithms proposed so far work by redirecting some arcs of the BDD for f is such a way that the resulting BDD loses nodes and does not acquire any new satisfying assignment. The simplest approach redirects arcs from internal nodes to the constant 0 node. A more sophisticated approach allows redirection of an arc from one internal node to another, subject to a containment check. As a result of these transformations, some internal nodes of the BDD become unreachable from the root are are dropped.

The various algorithms also differ in the order in which they select the arcs to be redirected. All algorithms examine the BDD for f before they compute the approximation to gather information on the distribution of nodes and satisfying assignment in the graph. In this Linear time algorithms like *Heavy-Branch Subsetting* and *Short-Path Subsetting* [53], the approximation then proceeds on the basis of this information. Quadratic algorithms [57, 52] update the information during the approximation, and produce denser results at the cost of higher runtimes. Finally, given an approximation algorithm $\alpha(f)$ and a minimization algorithm $\mu(l, u)$, it is easy to see that $\mu(\alpha(f), f)$ is another approximation algorithm.

5.7 Decomposition

It is often convenient to express a function f as either the conjunction or the disjunction of other functions. If $f' = \sum_i f_i'$, then $f = \prod_i f_i$. Hence, we can concentrate on disjunctive decomposition without loss of generality. A simple approach to this problem is based on the expansion theorem, which can be applied to obtain $f = f_0 + f_1$, with $f_0 = x_i' f_{x_i}'$ and $f_1 = x_i f_{x_i}$. The choice of the splitting variable x_i influences the quality of the result. A simple yet effective procedure to estimate the size of the decomposition was proposed in [15].

Another approach to decomposition is based on the recursive application of the following formula:

$$f = (\exists_x f) \cdot (f \downarrow (\exists_x f)).$$

This decomposition is shown to be canonical for a given variable order in [45], which also discusses several interesting properties of *constrain* and of the decomposition itself.

6 Variable Ordering

The size of the BDD for a given function depends on the order chosen for the variables. There are functions—sums of products where each product has support disjoint from the others—for which some orders lead to BDDs that are linear in the number of variables, whereas other orders give numbers of nodes that are exponential in the number of variables. The same occurs for the BDDs of adders. At the other end of the spectrum there are functions for which all possible BDDs are provably exponential in size and functions whose BDDs are linear for all orderings.

Though clearly one would not want to choose a worst case ordering for an adder, the importance of the ordering problem cannot be argued from the existence of functions exhibiting such an extreme behavior; equally well, the ordering problem cannot be dismissed simply because there are functions that are insensitive to the ordering. Indeed, even simple heuristics easily avoid the worst case behavior for adders. The practical relevance of the variable ordering problem rests on the existence of functions lacking a well understood structure, for which different orders—all derived by plausible methods—give widely different results.

In Section 6.1 we consider the problem of finding an optimum order. The exact algorithms can be used for the design of various types of CMOS gates, where an order of the input variables that minimizes the transistor count is sought and the number of variables is small. The complexity of the problem is such that the exact algorithms cannot be applied to medium and large-size functions. However, their analysis will illustrate some features of BDDs with respect to variable ordering and will suggest techniques that allow one to iteratively improve an existing BDD by changing the variable order. These techniques will be the subject of Section 6.2. Additional insight is provided by some results—some of general validity and some applicable to special classes of functions—that characterize optimal orders and that will be presented in Section 6.5. Finally, Section 6.6 will cover heuristic algorithms that are used to build a BDD starting from a circuit description.

6.1 Exact Ordering Algorithms

The simplest exact method to find an optimum variable order is to try all possible orders. For n variables there are $n!$ different orders and the cost of building a BDD is exponential in n in the worst case. Hence, a brute-force approach requires $O(n!2^n)$ time. This time can be improved by observing that two permutations of the variables that have a suffix in common will generate two BDDs with identical lower parts.

Suppose f is a boolean function of n variables $x_1, \ldots, x_n$. Let $N = \{1, \ldots, n\}$ be the set of the variable indices. An order π of $x_1, \ldots, x_n$ is a permutation of N. The BDD for f under order π is denoted by BDD(f, π). Let B be a subset of N and $\Pi(B)$ the set of orders whose last $|B|$ members belong to B, that is,

$$\Pi(B) = \{\pi | \pi[n - j + 1] \in B, j = 1, \ldots, |B|\}.$$

We shall refer to the variables whose indices are in B as to the *bottom* variables of $\Pi(B)$. The variables whose indices are in $N - B$ will be the *top* variables. Also, for an order π and a variable x_i, we define $N_i(f, \pi)$ as the number of nodes labeled x_i in BDD(f, π). The following lemma [26] expresses the fact that given a horizontal cut in a BDD for f that separates top variables from bottom variables, the order of the top variables does not influence the part of the BDD below the cut. (Likewise, the order of the bottom variables does not affect the part of the BDD above the cut.)

Lemma 5 *Let $B \subseteq N$, $k = n - |B| + 1$ and x_i a variable such that $i \in B$. Then there is a constant c such that for each $\pi \in \Pi(B)$ satisfying $\pi[k] = i$ we have $N_i(f, \pi) = c$.*

Proof. Note that k is the position in the order corresponding to the first bottom variable. We can cofactor f with respect to any combination of values (0 and 1) assigned to the top variables. Since cofactors with respect to variables commute, the resulting function is independent of the order of the top variables and is therefore the same for all $\pi \in \Pi(B)$. Repeating this argument for all possible assignments to the top variables, we see that for each unique cofactor of f with respect to an assignment to the top variables there must be a node (possibly the constant node) in the lower part of the BDD that is the root for that cofactor or its complement. This node is unique, because the BDD is canonical. The number of unique cofactors does not depend on the order of the top and the bottom variables, but only on f and B. Of the unique cofactors, some will depend on x_i. Again, the number of those that depend on x_i only depends on f and B, but not on the relative order of the bottom variables. For every cofactor that depends on x_i there will be exactly one node labeled x_i, because x_i is the first of the bottom variables, so that it must label the root node if it appears at all. Hence, the number of unique cofactors with respect to all different assignments to the top variables that depend on x_i gives the constant c and the theorem is proved. $\square$

Lemma 5 has many applications in the study of the size of BDDs. We now see how it is applied to the optimum ordering problem. If the optimal sizes of the lower parts of the BDD for all possible choices of B of size $s-1$ are known, we can derive the optimal size of the lower part of the BDD for a given B of size s as follows [32]. We consider each variable whose index is in B as first bottom variable. For x_i as first bottom variable, we consider $B' = B - \{i\}$. The minimum size for the lower part of the BDD with x_i first bottom variable is the minimum size previously computed for B' plus the number of nodes labeled x_i. Note that Lemma 5 guarantees that this number does not depend on the specific order of the variables in B' and hence it is sufficient to save the size of the best order of B'. (And the order itself, of course.)

The algorithm proceeds then by progressively increasing s. The size of the lower part of the BDD for $s = 0$ is known to be 1 (the constant node). The algorithm then works its way up, until the best overall order is found. It remains to discuss how the numbers of nodes labeled with the first bottom variable are computed. One method is to build a BDD for f for any order that is optimal for B' and that has x_i as first bottom variable. The number of nodes labeled by x_i can then be counted by inspecting the BDD. Computing the BDDs for all $B \subseteq N$ from scratch and storing them would be wasteful; hence we keep just one BDD and obtain all the desired orders by permuting the variables in that BDD.

In general, any permutation can be decomposed in a sequence of pairwise interchanges of variables. However, for our algorithm, all we need is to swap two variables at the time. We start from an arbitrary order π_0. We build BDD(f, π_0) and by pairwise interchange we obtain the remaining $n-1$ BDDs each having a different variable at the bottom. In this way, we have taken care of all the $B \subseteq N$ such that $|B| \leq 1$. We then consider all the $\binom{n}{2}$ subsets of size 2. Let $B = \{x_i, x_j\}$. We locate the two BDDs with x_i and x_j as bottom variables (they correspond to $B' = \{x_i\}$ and $B' = \{x_j\}$, respectively). If x_i is the bottom variable and x_j is already next to last, it is sufficient to count the nodes labeled x_j. However, if x_j is higher in the BDD and the next to last variable is x_k, we swap x_j and x_k before counting. The process is repeated with x_j as bottom variable and the best result is kept as representative of B. The other BDD is freed.

For $|B| = s$ we need to consider $\binom{n}{s}$ subsets. For each of them, we have to perform up to s swaps.

Example 10 As an example we consider finding an optimal order for $f = x_1x_2 + x_3$. The purpose of this example is rather to show how the algorithm works than to illustrate its efficiency. For the latter, larger functions are required.[7] Suppose the initial order is $\pi_0 = x_1 \leq x_2 \leq x_3$. The corresponding BDD is f_1 of Figure 13 and it represents both $B = \emptyset$ and $B = \{3\}$. We represent this by writing $f_\emptyset = f_{\{3\}} = f_1$. The algorithm first creates the BDDs with $B = \{2\}$ ($f_{\{2\}} = f_2$) and $B = \{1\}$ ($f_{\{1\}} = f_3$) by swapping x_2 with x_3 and x_1 with x_3 in $f_\emptyset$. This completes all the subsets of size 1. The costs are all the same, namely 2. (The constant node plus the one node labeled with the variable whose index is in B.) Suppose the first subset of size 2 that is considered is $\{1, 2\}$. Also, suppose that x_2 is the first variable considered as first bottom variable. The BDD $f_{\{1\}} = f_3$ already has x_2 right on top of x_1. Hence, it is sufficient to count the number of nodes labeled x_2. Since there is only one, the minimum cost for $B = \{1, 2\}$ is $1 + 2 = 3$ if x_2 is on top. Next the algorithm tries x_1 on top. To this end, it takes $f_{\{2\}} = f_2$ and swaps x_1 and x_3. The resulting BDD is f_4. The cost associated with this choice is again 3. Since this does not improve on the previous result and there are no other variables to try, the algorithm selects $f_{\{1,2\}} = f_3$ and releases f_4.

[7]Note that $3! < 2^3$, but $n! > 2^n$ for $n > 3$.

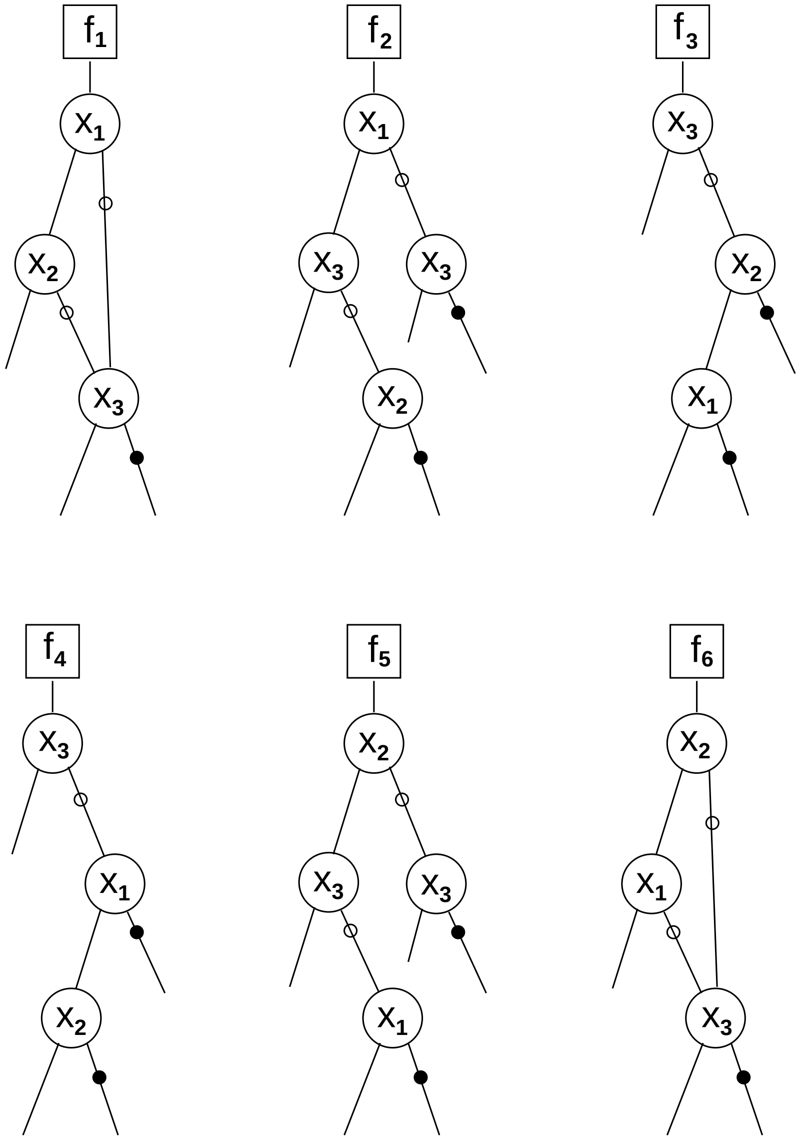


Figure 13: BDDs created by the exact ordering algorithm. The arcs with an open circle are regular else arcs. The arcs with a black dot are complement arcs. Dangling arcs point to the constant 1 node.

With similar procedure, the BDD for $B = \{1, 3\}$ is found to be the best between f_5 and f_6. The latter is chosen, so that $f_{\{1,3\}} = f_6$ and f_5 is released. Finally, $f_{\{2,3\}}$ is found to be f_1. Having completed the generation of the BDDs for subsets of size 2, the BDDs for the subsets of size 1 that are not also used by a larger subset are released. This is the case of f_2 in our example. All the costs are again the same, namely 3. The last step involves no swaps, since the only variable not in B' must be at the top. In this example there is a three-way tie that is broken in favor of the first solution found.

There are several observations that may speed up the algorithm, sometimes dramatically. First of all, if we ever get a BDD where there is exactly one node for each variable, we can immediately stop, because the order is optimal. In our example, the initial order was optimal; hence, we would have avoided the entire computation. As a second remark, some functions are symmetric in some of their variables. In our example, f was symmetric in x_1 and x_2. The presence of symmetric variables results in ties between BDDs where two or more symmetric variables are permuted. If some variables are known to be symmetric, some swaps may be avoided [35].

Finally, since the algorithm implicitly enumerates the variable orders, it is natural to think of a branch and bound approach [34]. The upper bound to the optimal size is given at any point in time by the size of the smallest BDD (not just the size of its lower part) representing a subset B. The lower bound for $B \subseteq N$ can be computed as the cost associated to B (the size of the lower part of the associated BDD) plus an estimate of the minimum number of nodes required to complete the BDD. If the number of nodes labeled with the first bottom variable is t, then at least $t - 1$ nodes are required to produce a connected DAG with a single root. Indeed with $t - 1$ nodes it is possible to build a binary tree with t leaves. Therefore, a lower bound to the cost of an order obtained from B is the cost associated to B plus $t - 1$. If this exceeds the current upper bound, B and its associated BDD are dropped. It is also possible to process the BDD top-down instead of bottom up. In many cases this allows a more effective bounding [24]. The effectiveness of bounding also depends on the initial order. The better the order, the tighter the upper bound. How to determine a good initial order will be covered in Section 6.6.

6.2 Iterative Improvement Techniques

The exact algorithm of Section 6.1 implicitly explores all variable orders by swapping variables in the BDDs. This idea can be applied as a heuristic by limiting the number of permutations that are explored. This results in a *local search* scheme, whereby an initial order is repeatedly perturbed by rearranging groups of variables until some termination condition is met. Three aspects characterize the possible variants of this method.

- The rule for generating a new order from the given one;
- the criterion for accepting a new order;
- the termination condition.

Among the rules proposed for generating a new order (these rules define the neighborhood of an order) we find:

- Pairwise interchange of adjacent variables [28];
- All permutations of groups of p adjacent variables, usually for a small value of p (≤ 4) [34].
- Pairwise interchange of arbitrary variables.

- Sifting, that is, the movement of each variable up and down in the order, until it reaches a local minimum [54].

- Group sifting, that is, the movement of a group of variables up and down in the order, until it reaches a local minimum [50, 49].

All strategies can achieve any permutation of an initial order given enough time. The advantage of considering only adjacent variables lies in the lower cost of performing the swap, which boils down to local manipulation of the BDDs. The interchange of non-adjacent variables can be effected by a series of swaps of adjacent variables. Sifting normally obtains better results than methods based on pairwise interchange or on the permutations of fixed size groups, because it performs a more systematic exploration of the search space. Even better results are obtained by group sifting.

As for the acceptance and termination criteria, all known combinatorial search strategies can be applied, going from greedy to simulated annealing [4] and genetic algorithms [23]. In selecting a time consuming strategy, one has to consider carefully whether the time spent in optimizing the size of the BDDs will be recovered in the successive manipulations.

6.3 Swapping Variables in Place

The iterative improvement methods of Section 6.2 are all based on successive swaps of adjacent variables. Each swap can be performed in time that is proportional to the number of nodes labeled by the two variables being swapped, if the data structures are properly chosen. Specifically, if the unique table is organized as a set of subtables—one for each variable—it is possible to access all the nodes labeled by a variable in time that does not depend on the total size of the BDDs being reordered, but only on the size of the subtable, and the number of nodes stored in it.

To make the swap time independent of the total size of the BDDs it is also necessary to perform the swapping "in place," that is, by modifying the given BDDs, instead of creating a modified copy of them. This is possible thanks to Lemma 5, which guarantees that the parts of the BDDs above and below the variables being swapped are unaffected by the swap.

Suppose variables x and y must be swapped, and suppose x precedes y in the order. The subtable corresponding to x is scanned and all nodes are considered in turn. Let f be the function associated with one such node; it can be written as:

$$f = xf_1 + x'f_0 = x(yf_{11} + y'f_{10}) + x'(yf_{01} + y'f_{00}).$$

Then swapping x and y corresponds to rearranging the terms of f:

$$f = yg_1 + y'g_0 = y(xf_{11} + x'f_{01}) + y'(xf_{10} + x'f_{00}).$$

Suppose, for simplicity, that f_{11}, f_{10}, f_{01}, and f_{00} correspond to four distinct nodes. Then the node labeled x can be relabeled y, f_{11}, f_{10}, f_{01}, and f_{00} can be reutilized without changes, while new nodes need to be created for g_1 and g_0.

Reutilization of the node labeled x is very important, because this node will be pointed in general by other nodes, or by the application. If a new node were used, the pointers to the old node would have to be searched for and changed. Thus, the operation would not be local to the subtables for x and y.

The nodes for f_1 and f_0 are no longer pointed by the node for f after the swap. Once the table for x has been scanned entirely, all nodes in the table for y that are no longer needed are freed. Such nodes are identified by associating a reference count to them. It is possible for some nodes in the y table to retain a non-zero reference count at the end of the process. These nodes have pointers from above x: They are moved to the table that now contains the nodes labeled y.

We now remove the assumption that f_{11}, f_{10}, f_{01}, and f_{00} correspond to distinct nodes. If neither child of the node for f is labeled y, then the node is moved to the other subtable; otherwise swapping proceeds as described above. However, if, for instance, $f_{11} = f_{01}$, no node for g_1 will be created. If in addition, $f_{11} \neq f_{10}$, and $f_{01} \neq f_{00}$, then the swap will cause the node count to decrease by one.

Each elementary swap operation may increase, decrease, or leave unchanged the node count. In the example just discussed, the decrease in node count is due to the elimination of a node with identical children. It is equally possible for a node to be saved as a result of increased sharing. Consider the two functions $f = xy$ and $g = x$. With the order $x < y$ they do no share any node, whereas with the order $y < x$, they share the one node labeled x. Hence, the total size decreases, even though the size of each function does not change.

6.4 Dynamic Reordering

Techniques that iteratively improve variable orders are frequently employed "dynamically:" if the size of the BDDs grows past a given threshold during the execution of an operation, the operation itself is suspended and reordering is performed, usually by some variant of sifting. After reordering the operation that was interrupted is usually restarted. This dynamic approach has proved very effective in many applications, because it allows reordering to intervene when it is actually needed. On the other hand, in sequential verification it is not uncommon that most of the time is taken by reordering, and it appears that a certain amount of control on the process by the application is beneficial [38]. Writing the manipulation functions in such a way that they can be safely interrupted by reordering requires some care. The details depend on the implementation of the BDD package.

6.5 Some Results on Optimal Ordering

When generating BDDs for the outputs of a circuit, we may use information on the structure of the circuit to find a good, sometimes optimal order for the variables. Consider the case of sums of products where each product term has support disjoint from the other terms. For instance:

$$x_1 x_2 x_3 + x_4 x_5 x_6 x_7 + x_8 x_9 .$$

For these functions every order that does not interleave the variables of two different product terms is optimal because there is exactly one node per variable. We shall refer to any such order as to a *non-interleaved* order. The reason why non-interleaved orders give linear-size BDDs for these sums of products is that the amount of information that needs to be remembered by a sequential processor that processes the inputs in the non-interleaved order is bound by a constant.

This argument can be generalized by introducing the notion of *width* of a circuit. We regard a combinational circuit as a DAG $G = (V, E)$. The nodes of the graph correspond to the gates of the circuit and the edges to the interconnections. A *topological order* T of V is a partial order such that:

$$(v_1, v_2) \in V \Rightarrow T(v_1) < T(v_2).$$

There are many such orders in general. The width of G at level r of order T is defined as:

$$W_T^r(G) = |\{(v_i, v_j) \in E : T(v_1) < r \leq T(v_2)\}|,$$

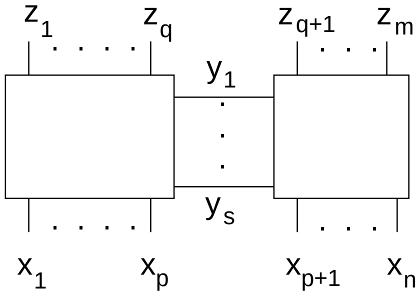


Figure 14: Illustration of the concept of width of a circuit.

that is, the number of edges connecting nodes at level less than r to nodes at levels r and above. The width of G with respect to T is:

$$W_T(G) = \max_r \{W_T^r(G)\},$$

and finally the width of G is given by:

$$W(G) = \min_T \{W_T(G)\}.$$

Through this definition, the width of a circuit is related to the amount of information that crosses a section of the circuit. The amount of information is related, in turn, to the number of wires that carry it. This is illustrated in Figure 14. Every level partitions the circuit in two. On the left in Figure 14 are all the gates and primary inputs with levels less than r. On the right are the gates and primary inputs with levels greater than or equal to r. Shown between the two sub-circuits are the connections whose number (s) gives the width of the circuit at level r, for the chosen topological order.

Example 11 A simple circuit is shown in Figure 15. One can see that $W_T(G) = W_T^2(G) = W_T^3(G) = 5$. However, it is possible to change the topological order so that the width is 4 and by decomposing the three input OR gate into two two-input OR gates, the width can be further reduced to 3. This example illustrates that the width depends on the circuit representation as well as on the topological order. A variable order consistent with this topological order is

$$f \leq g \leq e \leq a \leq b \leq c \leq d.$$

It is intuitive that circuits with low width have small BDDs. With reference to Figure 14, suppose the variables are ordered from x_n to x_1. Taking all 2^{n-p} cofactors of the function of the circuit with respect to all input variables at levels r or higher ($x_{p+1}, \ldots, x_n$) must result in 2^s unique cofactors at most. Hence the number of nodes labeled by x_p is less than or equal to 2^s. Let ω be a (total) order of the variables compatible with a topological order T[8]. It is possible to prove a result [3] linking the size of $BDD(G, \omega)$—the BDD for G built with order ω—to $W_T(G)$.

Theorem 9 *Let $G = (V, E)$ be the DAG of a circuit with n input variables, $x_1, \ldots, x_n$, and m outputs. If T is a topological order of V, and ω is a variable order compatible with T, then:*

$$|BDD(G, \omega)| < n \cdot m \cdot 2^{W_T(G)} + m.$$

[8]Note that a topological order specifies a level for all input variables.

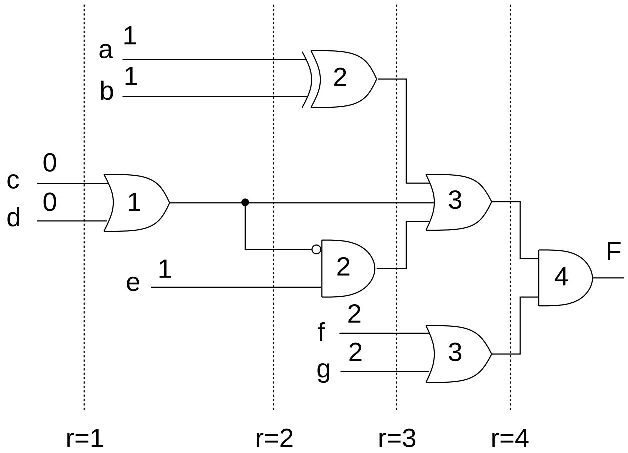


Figure 15: An example of a circuit with a topological order and the widths at the different levels.

Notice that even with the best topological order the bound may be loose, especially for multiple-output functions, where no sharing among the different outputs is assumed. The interest of Theorem 9 is not in the ability to establish a tight bound for the size of a specific BDD. It can be used to prove that a family of circuits has polynomial-sized BDDs, if it is possible to prove an upper bound on the width of the circuits in the set[9].

Non-interleaved orders are optimal for fanout-free circuits. The graph of a fanout-free circuit is a rooted tree. A non-interleaved order is obtained, for instance, by drawing the tree without intersecting arcs and then numbering the leaves—each corresponding to a variable—from left to right.

6.6 Heuristic Algorithms

In this section we consider the problem of finding a good order of the primary inputs, to build a BDD for a combinational circuit. The order may be used as starting point for the exact algorithm of Section 6.1 or the iterative improvement algorithms of Section 6.2. It may also be used directly, when building any BDD for the given function is the objective and the resulting BDD is not too large.

In all cases, the quality of the orders found affects the time required or even the ability to achieve the desired result. However, since the methods we shall consider here are intended to work on fairly large circuits, they are rather simple procedures, inspired by the results of Section 6.5 and by other heuristics that we shall introduce. We shall consider single output functions first, and then we shall deal with the case of multiple outputs.

6.6.1 Single-Output Functions

We begin by considering a circuit in sum-of-product form. A good choice for the variable to appear at the top of the BDD is the most binate variable, i.e., the variable that explicitly appears in most product terms. This choice minimizes the total number of product terms in the two cofactors. If we assume the number of product terms as a measure of the complexity of a function, then simplifying the cofactors, i.e., the two children of the top node, is likely to yield a smaller BDD. One can also see the most binate variable as the one having the largest effect on the value of the function. A simple, yet effective, method consists therefore of

[9]Specifically, the width should grow at most logarithmically.

ordering the variables from the most binate to the least binate. An alternative to this method is to transform the two-level circuit into multiple-level and then apply the methods discussed next.

For a fanout-free circuit composed of AND, OR, and NOT gates, a BDD with exactly one node per variable can be built by ordering the variables with a simple depth-first search of the circuit graph (a rooted tree in this case)[10]. The order of the variables is given by the order in which the corresponding leaves of the tree are visited. Assume for simplicity that the tree is binary. It is easily seen that the depth-first search will reach all the leaves in one sub-tree rooted at the top node of the tree, before reaching any leaf in the other sub-tree. The recursive application of this argument shows that the order produced by depth-first search is non-interleaved.

For circuits that contain fanout, it is still possible to use depth-first search to order the variables. This will have the effect of approximating a non-interleaved order, though no guarantee of optimality will be made. Furthermore, the order in which the children of a node are visited is no longer immaterial. We present two criteria to decide the order in which the children of a node are visited. The first criterion tends to place the input variables that fan out closer to the top of the BDD than those variables that do not fan out. More precisely, the children of each node are divided in three sets. The first set contains the input variables that fan out and that are not in the order yet.[11] These variables are immediately inserted in the order. The second set contains the internal lines and the third set contains the input variables that do not fan out. The variables in the third set are collected in a list and inserted in the order only after all the recursive calls for the internal lines have been completed. The variables that do not fan out are inserted after the last variables that fans out in that part of the tree.

Placing the variables that fan out closer to the top of the BDD can be seen as the analogous in the multiple-level context of the most binate variable heuristic.

The second criterion to rank the children of a node is given by the depth of the subgraph rooted at that node. In the case of the carry out of an adder, visiting the deepest subgraph first will order the inputs from the least significant to the most significant bit. Visiting the most shallow subgraph first will result in ordering the inputs in the reverse order. This suggests that the method may work well also for less structured circuits. Indeed one may observe that visiting the subgraphs in depth order is related to finding a topological order of the circuit graph with a low width.

An alternative approach to ordering the inputs of a circuit is also based on the idea of putting at the top of the BDD the variables that influence the values of the output most. To this purpose, a weight is assigned to each net in the circuit. The output is assigned a weight of 1, while the other nets are assigned weights according to the following rules.

- Each input of a gate with n inputs receives $1/n$ of the weight of the gate output;
- a fanout stem receives the sum of the weights of its branches.

These rules are applied until all inputs have weights assigned to them. The heaviest input is then chosen as the first variable of the order. The chosen input is then removed from the circuit. (I.e., it is assigned a non-controlling value; for an exclusive OR, the assigned value is immaterial.) The weights are then recomputed for the simplified circuits and the second variable of the order is thus identified. The process continues until all inputs are ordered.

Example 12 An example of weight assignment is shown in Figure 16. Assuming ties are broken in favor

[10] NAND and NOR gates pose no problem. However, if the circuit contains also exclusive OR and exclusive NOR gates, there may be more than one node per variable and the exact number of nodes may depend on the particular non-interleaved order chosen. This is because exclusive OR and NOR gates have no controlling values.

[11] These variables may have been visited during the search of another node. Hence, they may already be in the order.

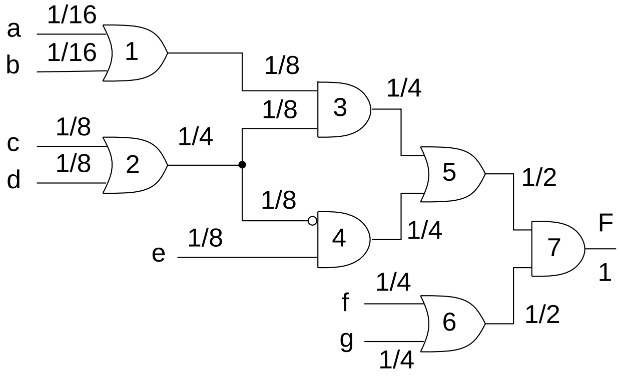


Figure 16: Example of weight assignment.

of the input that comes first in alphabetic order, the first input selected is f. The complete order is:

$$f \le g \le c \le d \le e \le a \le b.$$

This method does not directly address the main concern of the ones based on depth-first search, namely, keeping together the inputs that affect the same part of the circuit. However, when an input is removed from the circuit, the weights of its neighbors increase, thus increasing the likelihood that they will be selected next. One may verify that on a fanout-free circuit this method produces one of the depth-first orders with the depths of the subtrees used as tie breakers.

6.6.2 Multiple-Output Functions

The usual method to order the inputs of multiple output functions is to order the support of each output in turn. The problem is thus split into finding a good order for the outputs and finding a good order for the inputs. We have seen methods for the latter and hence we briefly discuss here the problem of ordering the outputs. Once again, we take the adder as a paradigm. If we want the most significant bits of the inputs at the top of the BDD, we need to consider the most significant output first. This corresponds to considering first the outputs with the deepest subtree. This heuristic based on depth is related, as when applied to input ordering, to the width of the circuit graph.

An alternative approach orders the outputs so as to minimize the number of variables that are added to the support when adding a new output to those already ordered. Let $f_1, \ldots, f_m$ be the outputs of the circuit for which a variable order is sought. Let S_i be the support of f_i, i.e., the set of variables on which f_i depends. Then one may try to minimize

$$\sum_{i=1}^{m} \left| \bigcup_{j=1}^{i} S_j \right|.$$

An exact solution to this problem is computationally expensive, and hence a greedy strategy is normally applied. One such strategy tries all the k-tuples of outputs as first k outputs of the order and then completes the rest of the order by choosing each time the locally best output, i.e., the one whose addition causes the least increase in the size of the support.

Once the orders for two or more outputs are given, they must be merged to yield an single order for all the outputs. The simplest approach is initialize the total order to the order of the first output and then append the variables of the successive outputs that do not appear yet in the order. This approach is often ineffective, as argued in [27], because it unnecessarily separates variables that the orders of the individual outputs put together. Consider the case of two outputs for which the orders $a < b < d < e$ and $b < c < d$ have been determined. The combined order based on appending yields $a < b < d < e < c$. However, the order $a < b < c < d < e$ is equally good for the first output (which does not depend on c) and is presumably better for the second output. This observation is at the basis of the *interleaving* algorithm of [27].

7 Implementation Issues

7.1 Memory Management

Consider building the BDD for the function of a single output circuit. The typical approach is to start by building BDDs for all the input variables of the circuit. One then computes the BDDs for the outputs of the gates fed by primary inputs only, and so on, until the BDD for the primary output is built. In the process, one computes the BDDs for many intermediate functions that are no longer of interest, once the result is obtained. One would like to release the memory used by those BDDs, but there are two problems. First, some subgraphs may be shared by more than one function and we must be sure that none of those functions is of interest any longer, before releasing the associated memory. Second, BDD nodes are pointed from the unique table and the computed table, as well as from other BDD nodes. There are therefore multiple threads and one cannot arbitrarily free a node without taking care of all the threads going through it[12].

A solution to these two problems is *garbage collection*. Garbage collection consists of deferring the release of memory until there is enough memory to be released to make the required effort worthwhile. A conceptually simple scheme for garbage collection is based on keeping a *reference count* for each internal and terminal node. This is a count of how many BDD nodes (internal and function nodes) point to an internal node. This count is incremented any time a new arc points to the node (for instance, during the execution of AND) and is decremented when nodes are freed. When a node is freed, its reference count is decremented. If the decrement results in a count of 0, then the reference counts of the children are decremented. The process is recursive. A node with a reference count of 0 is called *dead*.

When there are enough dead nodes, garbage collection is started. During garbage collection, the unique and computed tables are scanned and all entries pointing to dead nodes are cleared. The dead nodes are disposed of at this point. The cost of scanning the tables is amortized over a relatively high number of dead nodes. When garbage collection is invoked, if the collision lists are too long, one may increase the size of the tables and rehash all the entries that do not point to dead nodes.

One advantage of garbage collection is that a dead node may be resuscitated if a look-up in the computed or unique tables returns a dead node. Had the dead node been freed immediately, the re-computation would have been necessary. In this way, however, a better utilization of the allocated memory is possible. A *reclaim* operation is performed when a look-up returns a dead node. This operation undoes what the freeing of the node had done. The reference count of the node and of its children is incremented and if the children were also dead, the process is repeated recursively. Both freeing and reclaiming update the count of dead nodes.

The memory layout for a 32-bit machine is described in Figure 17. The computed table—not shown—is just a table pointing to some of the nodes. One should note that the collision list is implemented by an extra pointer in the node data structure. This integration of the unique-table with the DAG minimizes memory occupation and access time simultaneously. The reference count and the variable index are shown sharing a

[12] Locating the entry of the unique table pointing to a node just requires a hash-table look-up; on the other hand, deciding which entries, if any, of the computed table point to a node require keeping an extra pointer or scanning the table. Both solutions are less than optimal.

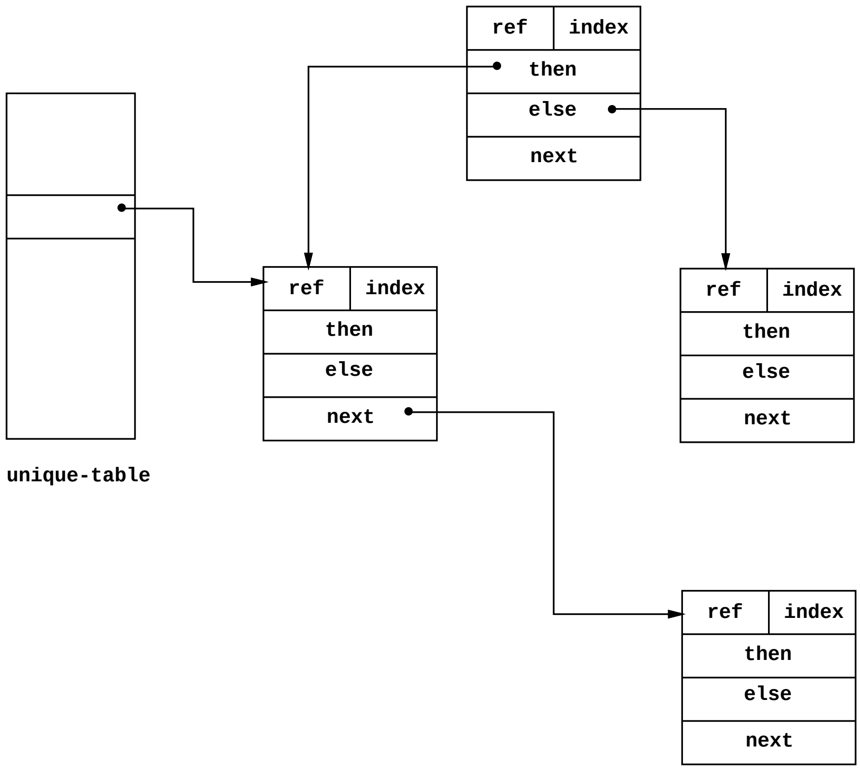


Figure 17: The memory picture.

single computer word. Assuming each field takes two bytes, we have limitations on the number of variables and the maximum reference count. The limit of 65535 variables is normally not serious, since in general one gets into trouble with much fewer variables for other reasons[13]. On the other hand, it is quite possible for the constant node to exceed a reference count of 65535. The solution to this problem—besides the obvious solution of allocating one four-byte word for each field—is based on *saturating increments*. If a reference count reaches 65535, then it is no longer changed. This may cause a dead node to be considered non-dead, but in practice the nodes that reach saturation are very few and hence the consequences are negligible.[14] All things considered, the memory occupation is about 18 bytes/node on a 32-bit machine. This number is based on a load factor of 2 for the unique table (the average length of the collision list) and is so divided:

- 4 words for the node itself (see Figure 17);
- one half of the hash table bucket (2 bytes).

To this one must add the memory requirements for the computed table, which vary considerably with the application. It is thus conceivable to store several million nodes on a machine with a large physical memory[15].

7.2 Efficient Utilization of the Memory Hierarchy

Modern computers rely on hierarchical organization to provide low average access time to large memories. The success of this approach is postulated on the locality of reference of programs. Unfortunately, the BDD algorithms that we have presented so far have poor locality, and therefore tend to incur large penalties in their accesses to memory. In practice, if a process that manipulates BDDs grows in size significantly beyond the available physical memory, its page fault rate normally becomes so severe as to prevent termination.

One approach to increase locality of reference in BDD manipulation consists of using contiguous memory locations for all the nodes labeled by the same variables and processing all the nodes for one variable before moving to another variable [48, 2, 55, 61]. This approach is commonly referred to as *breadth-first* processing of BDDs, though it is properly *levelized* processing. The algorithm for levelized calculation of a generic boolean connective is shown in Figure 18. It consists of two passes: During the fist, top-down pass requests are generated for the pairs of nodes. These requests correspond to nodes in a non-reduced version of the result. The second, bottom-up pass performs reduction by *forwarding* some requests to others.

The requests generated in the first pass double as entries of the computed table. In other words, the levelized approach features a built-in computed table that stores all intermediate results of a given call to *bfOp*. (Inter-operation caching of results is not covered by this mechanism.) The lossless computed table may be very memory-consuming for operations in which the non-reduced result is much larger than the reduced one. The extra memory required for the request queues may thus offset the advantages of the increased locality of access. Combining the levelized approach with the conventional recursive one can substantially alleviate the problem [61]. Additional speed-up techniques including *superscalarity* and *pipelining* are described in [55]. Reordering tends to destroy the segregation of nodes according to the variables. The typical approach is to restore such segregation after every reordering.

The levelized approach to BDD manipulation is quite effective in reducing the number of page faults. When the processes fit in main memory the situation is more complex. On the one hand, improved locality

[13] However, some schemes that assign variable indexes sparsely may conflict with such a limit.

[14] Sometimes, only 8 bits are reserved for the reference count, so as to free 8 bits for flags without increasing the size of the node. If this is done, care must be exercised to periodically adjust reference counts. Alternatively, an overflow table must be kept for those reference counts that become too large. The details of how this is done depend on the implementation.

[15] Virtual memory is of limited help in these cases, as explained in Section 7.2.

```
bfOp(op,F,G) {
    if terminal case (op,F,G) return result;
    minIndex = minimum variable index of (F,G);
    create a REQUEST (F,G) and insert in QUEUE[minIndex];
    /* Top-down APPLY phase. */
    for (i = minIndex; i ≤ numVars; i++) bfApply(op,i);
    /* Bottom-up reduce phase */
    for (i = numVars; i ≥ minIndex; i−−) bfReduce(i);
    return REQUEST or the node to which it is forwarded;
}

bfApply(op,i) {
    x is the variable with index i;
    while (REQUESTQUEUE[i] not empty) {
        REQUEST(F,G) = unprocessed request from REQUESTQUEUE[i];
        if (not terminal case ((op,F_x,G_x),result)) {
            nextI = minimum index of (F_x,G_x);
            result = find or add (F_x,G_x) in REQUESTQUEUE[nextI]; }
        REQUEST→THEN = result;
        if (not terminal case ((op,F_x',G_x'),result)) {
            nextI = minimum index of (F_x',G_x');
            result = find or add (F_x',G_x') in REQUESTQUEUE[nextI]; }
        REQUEST→ELSE = result;
    }
}

bfReduce(i) {
    x is the variable with index i;
    while (REQUESTQUEUE[i] not empty) {
        REQUEST(F,G) = unprocessed request from REQUESTQUEUE[i];
        if (REQUEST→THEN is forwarded to T) REQUEST→THEN = T;
        if (REQUEST→ELSE is forwarded to E) REQUEST→THEN = E;
        if (REQUEST→THEN == REQUEST→ELSE)
            forward REQUEST to REQUEST→THEN;
        else if ((REQUEST→THEN,REQUEST→ELSE) found in UNIQUETABLE[i])
            forward REQUEST to that node;
        else
            insert REQUEST in UNIQUETABLE[i];
    }
}
```

Figure 18: Levelized BDD manipulation algorithm.

of reference should translate into increased cache hit rate. On the other hand, the overhead inherent to the two-pass approach is fairly sizable [43]. Experimental evidence suggests that the levelized approach is better than the recursive one when building BDDs for combinational circuits, but the opposite is true for sequential verification [60]. Improving memory locality for the conventional recursive approach is also possible. In [41] it is shown that the number of cache misses incurred while accessing the unique table may be substantially reduced by sorting the collision lists and making sure that older nodes occupy lower addresses in memory. An additional benefit of this approach is an efficient *generational* garbage collection scheme.

8 Sequential Verification

We suppose we are given a sequential system modeled as a finite state machine, and a property that the system is supposed to verify. Since the system is finite-state, several logics commonly used to express properties admit decision procedures based on state enumeration. In this section we focus on the most elementary of these procedures: Reachability Analysis. Our choice is motivated by the following observations. First, reachability analysis is sufficient to decide an important class of properties: invariants, that is, propositional formulae that should hold in all possible states of the system. Second, reachability analysis exemplifies the fixed point computations that are the major constituents of more general decision procedures. Therefore reachability analysis gives us the opportunity to examine most of the issues connected with the efficient deployment of BDDs in model checking.

We define a deterministic finite state machine (FSM) as a 6-tuple $\langle I, S, O, \delta, \lambda, S^0 \rangle$, where I is the input alphabet, S is the set of states, O is the output alphabet, δ and λ are the next-state (or transition) and output functions, respectively, and S^0 is the set of initial (reset) states. In the sequel we assume that $S = B^m$ and $I = B^p$ for some m and p. Therefore $\delta : B^{m+p} \mapsto B^m$ is a multiple-output boolean function. We call such a FSM *encoded.*

A state σ is *reachable* from state ρ, if there exist a sequence of states $s_1, \ldots, s_k$ and a sequence of inputs $i_0, \ldots, i_k$ (possibly repeated), such that $\delta(\rho, i_0) = s_1$, $\delta(s_j, i_j) = s_{j+1}, j = 1, \ldots, k-1$, and $\delta(s_k, i_k) = \sigma$. A state is simply reachable if it is reachable from any state in S^0. We will define reachability analysis as the problem of enumerating all reachable states of a finite state machine. Note that in this problem we are only concerned with the next-state function δ. Hence, in the following, we ignore the output function λ.

Reachability analysis can be performed in time linear in the number of edges of the state transition graph by depth-first search. However, the number of states of a FSM grows exponentially with the number of memory elements. Hence, even a linear time algorithm may be inadequate in practice. One approach to achieve sub-linear runtimes for some problem instances is based on the use of characteristic functions.

8.1 Characteristic Functions and Relations

Let $B = \{0, 1\}$ and let $S \subseteq B^n$ be a set of points of the boolean space B^n. A function $\chi_S : B^n \to B$ is called the *characteristic function* of S if it is one exactly for the points of B^n that are in S. That is:

$$\forall x \in B^n, x \in S \Leftrightarrow \chi_S(x) = 1.$$

We can extend the definition of characteristic function to subsets of an arbitrary finite set Q, by providing an injective mapping from Q to B^n. Such a mapping is called a *binary encoding* of Q. Let $\mathcal{E}$ be a binary encoding of Q, that is,

$$\mathcal{E} : Q \to B^n,$$

for $n \geq \log_2 |Q|$. Then the *image* of $P \subseteq Q$ according to $\mathcal{E}$ is $A \subseteq B^n$ defined by:

$$A = \{a \in B^n : \exists p \in P, a = \mathcal{E}(p)\}.$$

We define the characteristic function of P according to $\mathcal{E}$ as the characteristic function of A:

$$\chi_P^{\mathcal{E}} = \chi_A.$$

Whenever the encoding is understood, we shall simply write χ_P. With these definitions, we can associate a characteristic function to every finite set. In the following we shall mainly deal with characteristic functions of subsets of B^n.

All set theoretic manipulations can be performed directly on the characteristic functions. In particular one can easily see that, for $S_1, S_2 \subseteq B^n$:

$$\begin{aligned} \chi_{S_1 \cup S_2} &= \chi_{S_1} + \chi_{S_2} \\ \chi_{S_1 \cap S_2} &= \chi_{S_1} \cdot \chi_{S_2} \\ \chi_{\overline{S_1}} &= \chi'_{S_1}, \end{aligned}$$

and for arbitrary finite sets $A_1, A_2 \subseteq Q$ and encoding $\mathcal{E} : Q \to B^n$, we have:

$$\begin{aligned} \chi_{A_1 \cup A_2} &= \chi_{A_1} + \chi_{A_2} \\ \chi_{A_1 \cap A_2} &= \chi_{A_1} \cdot \chi_{A_2} \\ \chi_{\overline{A_1}} &= \chi'_{A_1} \cdot \chi_Q. \end{aligned}$$

Characteristic functions are interesting because they often provide a compact representation and an efficient manipulation of sets. Consider the characteristic function

$$\chi_S(x_1, \ldots, x_{100}) = x_1 + x_{100}.$$

One can verify that:

$$|S| = 3 \cdot 2^{98}.$$

However, the representation of χ_S in the form of a BDD only takes two internal nodes. Though almost all sets require exponentially large BDDs to represent their characteristic functions (see [46] for a proof of this classical result), it is true in general that we can manipulate much larger sets by dealing with their characteristic functions than we would by explicitly representing the elements of the sets.

Among all sets that can be represented by characteristic functions, we now consider relations, i.e., subsets of a suitable cartesian product. Let $R \subseteq Q_1 \times Q_2$ be a binary relation[16]. We shall use different sets of variables to identify the elements of Q_1 and Q_2 and we shall therefore write

$$\chi_R(x_1, \ldots, x_n, y_1, \ldots, y_m) \leq \chi_{Q_1}(x_1, \ldots, x_n) \cdot \chi_{Q_2}(y_1, \ldots, y_m),$$

noting that the cartesian product of two sets is obtained by taking the product of the respective characteristic functions, when they have disjoint support. This easily generalizes to n-ary relations. Note that we have to use different sets of variables also in the case of $R \subseteq Q^2$, because the elements of the relation are ordered pairs and we must be able to tell the first element from the second. The most important relation that we shall deal with in the sequel is the *transition relation* of a FSM:[17]

[16] Binary here means that the elements of the relation are pairs of elements from Q_1 and Q_2.

[17] In the following, we implicitly assume that we are dealing with characteristic functions, so that, with abuse of notation, we write R instead of χ_R.

Definition 8 *Let* $\delta = (\delta_1, \ldots, \delta_m)$ *be the transition function of an encoded FSM* M*. Then the transition relation of* M *is given by:*

$$T_M(y_1, \ldots, y_m, x_1, \ldots, x_{m+p}) = \prod_{i=1}^{m} (y_i \equiv \delta_i(x_1, \ldots, x_{m+p})).$$

The transition relation describes all the pairs of states that are connected by an arc in the state transition graph of M and the inputs labeling that arc.

The transition relation is not restricted to the representation of deterministic machines. We could have defined FSMs directly in terms of their transition relations. We did not follow this approach for two reasons. First, there is no loss of generality in restricting ourselves to deterministic transition structures. We can apply Choueka's determinization procedure [19] to remove nondeterminism. Second, we need to define δ because we are going to discuss an algorithm based on it, and it is both easier and more commonly done to derive T_M from δ than vice versa.

8.2 Symbolic Breadth-First Search

Characteristic functions allow us to represent and manipulate large sets of states and relations among them. Depth-first reachability analysis, however, deals with states individually. Breadth-first search, on the other hand, allows one to deal naturally with multiple states simultaneously and has thus become the method of choice for the traversal of large machines.

When multiple states are processed simultaneously in breadth-first manner, the algorithm is called *symbolic breadth-first* search or traversal. The sets of states that are processed are represented by their characteristic functions and the problem is then to evaluate δ for a given characteristic function as input. The result of the evaluation is the characteristic function of the set of states that are reachable in one step from any state in the set whose characteristic function is the input to δ. This process is sometimes called *symbolic simulation* of δ. We shall view this evaluation as the computation of the image of an appropriate function. From this point on, we assume that all functions are represented by BDDs. In particular, δ can be represented in two ways:

1. By a vector of BDDs, one for each next state variable;
2. by the transition relation associated with δ.

These two methods result in two algorithms, called the *transition function* algorithm and the *transition relation* algorithm, that have a common structure. In order to present this structure, we now consider two definitions concerning the image of a function with domain B^n and range B^m.

Definition 9 *Let* $f : B^n \to B^m$ *be a multiple-output boolean function. The* image *of* f*, denoted by Img*(f)*, is defined by:*

$$Img(f) = \{v \in B^m : \exists x \in B^n, f(x) = v\}.$$

Informally, the image of f is the set of output values that f can produce. If v belongs to the image of f, then there is an input x such that $v = f(x)$. We want to impose a constraint on the inputs that can be applied to f and thus restrict the values that f may produce. To this end, we introduce the following definition.

Definition 10 *Let* $f : B^n \to B^m$ *be a multiple-output boolean function and let* C *be a subset of* B^n*. Then the image of* f constrained *by* C*, denoted by Img*(f, C)*, is defined by:*

$$Img(f, C) = \{v \in B^m : \exists x \in C, f(x) = v\}.$$

```
traverse(δ, S⁰) {
        Reached = From = S⁰;
        do {
                To = Img (δ,From);
                New = To − Reached;
                From = New;
                Reached = Reached ∪ New;
        } while (New ≠ ∅);
        return Reached;
}
```

Figure 19: Pseudo-code of the symbolic breadth-first traversal.

According to this definition, $\mathit{Img}(f) = \mathit{Img}(f, B^n)$. The set C is called the constraint and the problem of finding $\mathit{Img}(f, C)$ is called *constrained image computation*. In the traversal of a FSM, the constraint represents a group of current states and computing $\mathit{Img}(f, C)$ corresponds to finding all the states that are reachable in one step from the states represented by C. Also, in the case of FSM traversal, $n = m + p$, where m is the number of state variables (that are both inputs and outputs of the next state function) and p is the number of primary inputs. The constraint C is thus defined over B^m, but it can be regarded as defined over B^n as well, by just assuming that it is defined over $B^m \times B^p$, but does not depend on the variables ranging over the last p coordinates. We are now ready to present a first version of the symbolic breadth-first traversal procedure. The pseudo-code is given in Figure 19. The union and difference of sets are actually performed by manipulating the corresponding characteristic functions. The number of iterations performed gives the *sequential depth* of the FSM, i.e., the longest of the shortest paths connecting each state to an initial state. In other words, *traverse* reaches each state in the minimum number of steps. This is a general property of breadth-first methods. Before we consider in detail how to compute $\mathit{Img}(\delta, \mathit{From})$, we examine the assignment:

$$\mathit{From} = \mathit{New}. \tag{8}$$

This statement simply says that the newly reached states will be considered as current states at the next iteration. Notice, however, that from the point of view of correctness, it is possible to assign to *From* any set such that:

$$\mathit{New} \subseteq \mathit{From} \subseteq \mathit{New} \cup \mathit{Reached}.$$

(Note that *Reached* has not been updated yet when the assignment to *From* takes place.) In particular, *From* could be set to equal *To* or $\mathit{Reached} \cup \mathit{New}$. If $\mathit{From} \supset \mathit{New}$, then some states will be examined more than once. If states were individually processed that would be a poor choice. However, we are manipulating characteristic functions; the size of the BDDs and the complexity of the operations on them are only loosely related to the number of minterms of the corresponding functions. It is indeed possible that a function with more minterms than another actually has a smaller BDD. In our case, we could find a choice for *From* that includes some previously reached states, but results in a smaller BDD and/or a faster computation of $\mathit{Img}(\delta, \mathit{From})$. One may observe that:

$$\mathit{New} \subseteq (\mathit{New})_{\mathit{Reached}'} \subseteq \mathit{New} \cup \mathit{Reached}.$$

Therefore, a possible choice for *From* that is expected to result in a compact BDD is given by:

$$\mathit{New} \Downarrow \mathit{Reached}',$$

and this expression could replace *New* in the right-hand side of (8).

The image of δ can be derived by exhaustive simulation. If a constraint on the inputs is given, only those inputs that satisfy the constraint are applied. However, this method is impractical for all but the simplest cases. Therefore we seek methods of computing $Img(\delta, From)$, that only manipulate the characteristic functions of the sets involved in the computation.

8.3 Image Computation

In this section we concentrate on the problems of image computation for the two algorithms based on the transition function and the transition relation. Throughout this section, we shall use the following example to illustrate the various techniques.

Example 13 We want to compute the image of

$$f = (f_1, f_2, f_3),$$

where:

$$\begin{aligned} f_1 &= a(b+c) \\ f_2 &= b(a+c) \\ f_3 &= c(a+b) \end{aligned} \tag{9}$$

For this simple example, it is easy to compute the image by exhaustive simulation:

a	b	c	f_1	f_2	f_3
0	0	0	0	0	0
0	0	1	0	0	0
0	1	1	0	1	1
0	1	0	0	0	0
1	1	0	1	1	0
1	1	1	1	1	1
1	0	1	1	0	1
1	0	0	0	0	0

By inspection, we see that the image of f is $\{$000,011,101,110,111$\}$. The BDDs for the three functions are shown in Figure 20.

8.4 Image Computation for Transition Functions

We now examine how to compute the image of a function $f : B^n \to B^m$, when the transition function is given as a set of BDDs, one for each next state variable. It is possible to find $Img(f, C)$ by breaking the computation in two steps [21]:

1. Find a new function whose image over the entire domain B^n equals the image of f over the restricted domain C.

2. Find the (unconstrained) image of the new function.

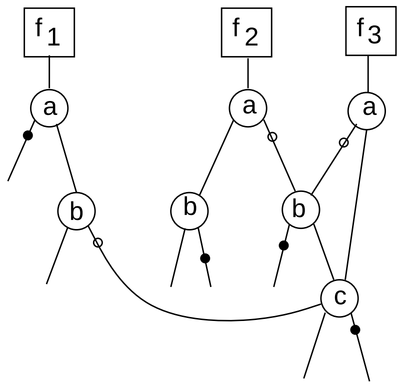


Figure 20: BDDs for the image computation examples.

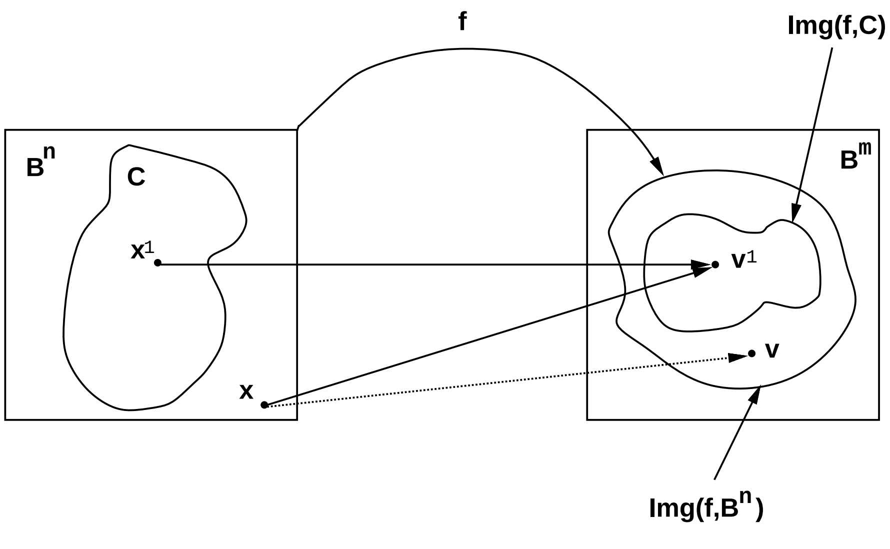


Figure 21: Constrained image computation.

The first step of the procedure is pictorially represented in Figure 21. It requires an operator that, when applied to a multiple-output function, maps a point $x \in B^n$ outside C onto the same point of B^m as another point $x^1 \in C$. Any such operator is called an *image restrictor*.

Noting the similarity between the re-mapping performed by an image restrictor and that of a generalized cofactor, we may ask whether any of the operators we already know is indeed an image restrictor. The answer is affirmative, in that the *constrain* operator (indicated by $\downarrow$) of Section 5.4.1 is indeed an image restricting generalized cofactor. This can be easily seen by observing that, when computing $g \downarrow h$, the mapping performed by *constrain* only depends on h. Therefore, when *constrain* is applied to all the components of $f = (f_1, \ldots, f_m)$, it performs the same mapping for all the components. This guarantees that the image of x will become the image of x^1, as shown in Figure 21. In summary, we have:

$$\mathit{Img}(f, C) = \mathit{Img}(f \downarrow C),$$

where it is understood that *constrain* is applied to each component of f.[18] Note that the *restrict* operator ($g \Downarrow h$) is not an image restrictor, because the mapping of the points outside h depends also on g, the function being cofactored; indeed, g determines what variables should be quantified in h. Though it is the only one that we shall consider in detail, *constrain* is not the only image restrictor we can define. For instance, one may choose an arbitrary element $\tilde{x} \in C$ and map all the points outside C onto $f(\tilde{x})$. In most cases this will be less efficient than applying *constrain*, though.

We now assume that the constraint C has already been taken care of, by applying the *constrain* operator and we concentrate on two techniques for the unconstrained image computation problem.

8.4.1 Image Computation by Input Splitting

The first technique is based on the following expansion:

$$\mathit{Img}(f) = \mathit{Img}(f_{x_i}) + \mathit{Img}(f_{x_i'}). \tag{10}$$

Intuitively this identity says that the image of the function f is the union of the set of output values that can be obtained by setting $x_i = 1$ and the set of output values that can be obtained by setting $x_i = 0$; hence the name of *image computation by input splitting*. Equation 10 is applied recursively until a terminal case is found. The simple terminal case is when f is constant. Suppose $f = \hat{y}_1 \cdots \hat{y}_m$, where $\hat{y}_i \in \{0, 1\}$. Then the characteristic function of the image of f is $\tilde{y}_1 \cdots \tilde{y}_m$, where $\tilde{y}_i = y_i$ if $\hat{y}_i = 1$ and $\tilde{y}_i = y_i'$ otherwise.

Though this terminal case is sufficient to build a correct algorithm, efficiency requires a more sophisticated analysis of terminal conditions. In particular, we consider three mechanisms.

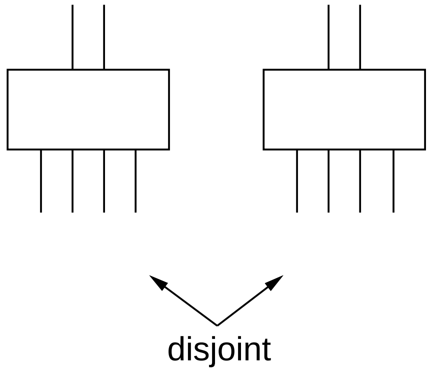
- Decomposition due to disjoint support;
- Identical and complementary components;
- Identical subproblems.

Decomposition due to Disjoint Support. Consider the case of $f = (f_1, f_2)$, where the supports of f_1 and f_2 are disjoint. Then,

$$\mathit{Img}(f) = \mathit{Img}(f_1) \times \mathit{Img}(f_2).$$

[18] It is important for the variable order not to change during image restriction with *constrain* because the mapping, although independent of the component of f, does depend on the variable order.

$$Img((f_1, f_2)) = \{00, 11\} \qquad Img((f_3, f_4)) = \{10, 01\}$$



$$Img(f) = \{0010, 0001, 1110, 1101\}$$

Figure 22: Example of image decomposition.

In terms of characteristic functions, it is sufficient in this case to take the product of the characteristic functions computed for the images of f_1 and f_2. This result can be easily generalized to the case of a multi-way partition of $(f_1, \ldots, f_m)$, such that the component functions in one block have support disjoint from the functions of the other blocks. An example is shown in Figure 22. The effect of partitioning, when it occurs, is dramatic, so that one should try to cause partitioning as early and as often as possible, by appropriately choosing the splitting variable.

Identical and Complementary Components. Suppose $f = (f_1, f_2)$ and $f_1 = f_2$. Assuming f_1 is not constant, then *Img*$(f) = y_1 \equiv y_2$. If $f_1 = f_2'$, then *Img*$(f) = y_1 \oplus y_2$. In general, it is possible to remove all but one identical or complementary components from a problem, solve the simplified problem, and finally reintroduce the missing variables, by adding clauses of the form $y_i \equiv y_j$ or $y_i \oplus y_j$. This technique does not directly affect the size of the search space. However, it reduces the amount of work for each recursive invocation of the algorithm and also increases the chances of early termination due to the technique to be discussed next.

Identical Subproblems. It is often the case that different cofactors of f are identical or strictly related. By maintaining a table of the problems solved thus far, an algorithm may avoid repeated computations of the same result. This is the familiar technique applied in BDD manipulation algorithms. As in that case, optimal exploitation of the table of computed results is an issue that may be addressed by trying to normalize the calls. The normalization is obtained by eliminating all repeated and complement components, complementing all the components pointed by complement arcs, and sorting the components according to the order of the pointers.

The recognition of identical subproblems is particularly important when the image being computed is represented by a small BDD.

Example 14 Consider the case when the image of a function f is defined by:

$$Img(f) = \{X, Y : X \geq Y\}.$$

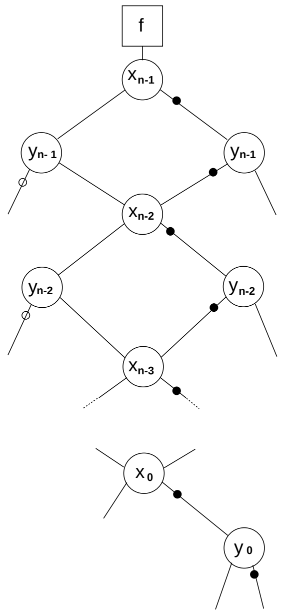


Figure 23: A BDD for $X \geq Y$.

(Here X and Y are both sets of output variables.) The corresponding BDD, for an appropriate variable order, is shown in Figure 23. This BDD grows linearly with n. This happens frequently with arithmetic functions that can be expressed iteratively. If identical subproblems are not detected, though, an exponential number of recursive calls will be performed, thus effectively expanding the BDD into a tree.

Example 15 For the function defined by (9), the computation based on input splitting proceeds as follows. We initially select an input variable for splitting. From symmetry considerations, we see that in this case all variables give similar results; hence we choose a. We then compute *Img*$(f)_a$. We have:

$$(f_1)_a = b + c \quad (f_2)_a = b \quad (f_3)_a = c.$$

This is not yet a terminal case, so we split again, this time with respect to b, and find:

$$(f_1)_{ab} = 1 \quad (f_2)_{ab} = 1 \quad (f_3)_{ab} = c.$$

This is a terminal case for which the solution is $y_1 y_2$. Considering then the negative cofactors, we get:

$$(f_1)_{ab'} = c \quad (f_2)_{ab'} = 0 \quad (f_3)_{ab'} = c.$$

This is again a terminal case, from which we get $(y_1 y_3 + y_1' y_3') y_2'$. Therefore:

$$\begin{aligned} \textit{Img}(f)_a &= y_1 y_2 + y_1 y_2' y_3 + y_1' y_2' y_3' \\ &= y_1 y_2 + y_1 y_3 + y_1' y_2' y_3'. \end{aligned}$$

We then consider *Img*$(f)_{a'}$. We have:

$$(f_1)_{a'} = 0 \quad (f_2)_{a'} = (f_3)_{a'} = bc.$$

This is a terminal case and we get:

$$Img(f)_{a'} = y_1'(y_2y_3 + y_2'y_3').$$

Finally, adding the two partial results, we have:

$$Img(f) = y_1y_2 + y_1y_3 + y_2y_3 + y_1'y_2'y_3',$$

in agreement with the results obtained by exhaustive enumeration.

8.4.2 Image Computation by Output Splitting

Rather than splitting the problem with respect to the input variables, one can partition with respect to the output functions, by using the following formula.

$$\begin{aligned} Img(f) &= y_i \cdot Img((f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_m), f_i) + \\ &\quad y_i' \cdot Img((f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_m), f_i'). \end{aligned} \tag{11}$$

Intuitively, this identity says that the image of f can be dichotomized into the points for which $f_i = 1$ and those for which $f_i = 0$. The resulting method is again recursive and at each step we have to compute two constrained images. This can be done by applying *constrain*, as we have seen at the beginning of Section 8.4. The termination conditions are similar to those used with input splitting.

The number of component functions decreases at each iteration. The number of inputs may or may not decrease, depending on the simplification that occurs in applying *constrain*. When the problem is reduced to a single component f_j, if f_j is not constant, the image is 1. If $f_j = 1$ the image is y_j, and if $f_j = 0$, the image is y_j'. The efficiency of this procedure can be increased by application of the same techniques we have seen for input splitting, namely decomposition, recognition of identical or complementary components, and the use of a computed table.

It is also possible to combine the input and output splitting techniques, by choosing at every step whether to decompose the domain or the range of the function. There are cases where input splitting is better than output splitting, and vice versa. At first glance, output splitting has the advantage of performing a partitioning of the problem (there are no common image points in the two sub-problems) and of visiting a smaller search space in the case of FSM traversal (because $m \leq n$). However, this is often offset by the increased complexity brought about by *constrain*, and by the decreased occurrence of decomposition. In most cases, it has been observed that input splitting is more efficient.

Example 16 We now apply the output splitting method to our example. We choose the splitting order as f_1, f_2, f_3. We initially compute $Img((f_2, f_3), f_1)$. Figure 24 shows the effect of applying *constrain* to f_2 and f_3. Let $f_2^{(1)} = f_2 \downarrow f_1$ and $f_3^{(1)} = f_3 \downarrow f_1$. We now compute

$$Img((f_3^{(1)}), f_2^{(1)}) = Img(c) = 1,$$

and

$$Img((f_3^{(1)}), f_2^{'(1)}) = Img(1) = y_3.$$

Hence,

$$Img((f_2, f_3), f_1) = y_2 + y_2'y_3 = y_2 + y_3.$$

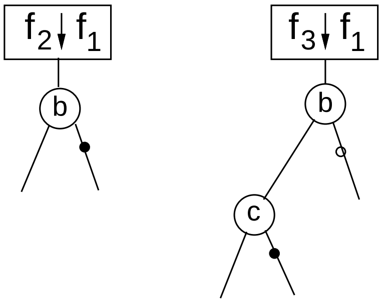


Figure 24: First step in the image computation by output splitting.

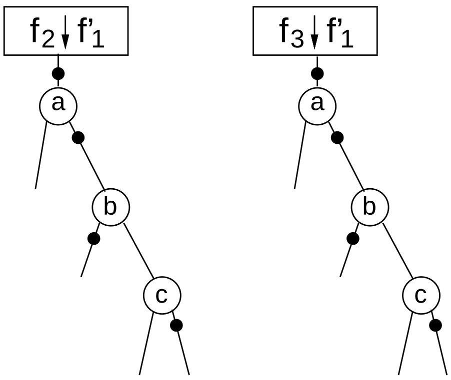


Figure 25: Second step in the image computation by output splitting.

We must now compute $Img((f_2, f_3), f_1')$. The result of applying *constrain* with respect to f_1' is shown in Figure 25. Since the two BDDs are identical and non-constant,

$$Img((f_2, f_3), f_1') = y_2 y_3 + y_2' y_3'.$$

Finally, combining the two sub-problems,

$$\begin{aligned} Img(f) &= y_1(y_2 + y_3) + y_1'(y_2 y_3 + y_2' y_3') \\ &= y_1 y_2 + y_1 y_3 + y_2 y_3 + y_1' y_2' y_3', \end{aligned}$$

in agreement with our previous results.

8.5 Image Computation for Transition Relations

The image of a next-state function described by the associated transition relation T can be computed as:

$$\exists_{x_1,\ldots,x_n}(T(y_1, \ldots, y_m, x_1, \ldots, x_n) \cdot C(x_1, \ldots, x_n)),$$

where C is a constraint on the inputs, describing a set of present states. If no such constraint is given—alternatively, if $C = 1$—we simply have to existentially quantify the input variables in T.

Example 17 For the function defined by (9), the transition relation is given by:

$$\begin{aligned} T = {} & (y_1 ab + y_1 ac + y_1' a' + y_1' b' c') \cdot (y_2 ab + y_2 bc + y_2' b' + y_2' a' c') \cdot \\ & (y_3 ac + y_3 bc + y_3' c' + y_3' a' b') \end{aligned}$$

and we want to compute:

$$Img((f_1, f_2, f_3)) = \exists_{a,b,c} T.$$

In this case the order of quantification is immaterial. We follow the alphabetic order.

$$\begin{aligned} \exists_a T &= T_a + T_{a'} \\ &= (y_1 b + y_1 c + y_1' b' c')(y_2 b + y_2 bc + y_2' b')(y_3 c + y_3 bc + y_3' c') + \\ &\quad (y_1')(y_2 bc + y_2' b' + y_2' c')(y_3 bc + y_3' c' + y_3' b') \end{aligned}$$

$$\begin{aligned} \exists_{ab} T &= (\exists_a T)_b + (\exists_a T)_{b'} \\ &= y_1 y_2 (y_3 c + y_3' c') + y_1'(y_2 c + y_2' c')(y_3 c + y_3' c') + \\ &\quad (y_1 c + y_1' c') y_2' (y_3 c + y_3' c') + y_1' y_2' y_3' \\ &= [y_1 y_2 + y_1'(y_2 c + y_2' c') + (y_1 c + y_1' c') y_2'](y_3 c + y_3' c') + y_1' y_2' y_3' \end{aligned}$$

$$\begin{aligned} \exists_{abc} T &= (\exists_{ab} T)_c + (\exists_{ab} T)_{c'} \\ &= (y_1 y_2 + y_1' y_2 + y_1 y_2') y_3 + y_1' y_2' y_3' + (y_1 y_2 + y_1' y_2' + y_1' y_2') y_3' \\ &= y_1 y_2 + y_1 y_3 + y_2 y_3 + y_1' y_2' y_3'. \end{aligned}$$

This simple example shows that the computation is more complex when using the transition relation than when using transition function. This is mostly due to the increased number of variables. In its original form, this method is indeed less efficient than the ones previously seen. However, there are ways to speed it up that make it competitive. We begin their study by proving some results on generalized cofactors, and *constrain* and *restrict* in particular, that are of interest in this context.

8.5.1 Partitioned Image Computation

The major computational problem in image computation with the transition relation derives from taking the conjunction of the components. Even if the individual BDDs for $y_i \equiv f_i$ may be small, their product may be too large. The final result—after quantification—may also be small, but we may exceed the allotted resources while building some intermediate result. A substantial increase in speed an memory efficiency comes from the following simple observation, whose proof is immediate.

Lemma 6 *Let $f : B^{m+n} \to B$ be a function of $y_1, \ldots, y_m, x_1, \ldots, x_n$ and $g : B^{m+n-i+1} \to B$ be a function of $y_1, \ldots, y_m, x_i, \ldots, x_n$, $1 < i \leq n$. Then:*

$$\exists_{x_1 \cdots x_n}(f \cdot g) = \exists_{x_i \cdots x_n}(\exists_{x_1 \cdots x_{i-1}}(f) \cdot g).$$

Lemma 6 says that we can partially distribute the quantifications over the product, if the terms of the product do not all depend on all the variables. Since the quantification of a variable normally simplifies the result, we can hope that by intertwining products and quantifications, the size of the intermediate BDDs will be better kept under control [11, 59]. The following example illustrates this point.

Example 18 We want to compute *Img*(f, g), with $f = (f_1, f_2, f_3)$,

$$\begin{aligned} f_1 &= x_1 + x_2 \\ f_2 &= x_2' + x_3 \\ f_3 &= x_2 x_4 + x_3', \end{aligned}$$

and $g(x_1, \ldots, x_4) = x_1 + x_2$. According to (7), we have:

$$\begin{aligned} \mathit{Img}(f,g) = \; & \exists_{x_1 x_2 x_3 x_4}[((y_1(x_1 + x_2) + y_1' x_1' x_2') \Downarrow (x_1 + x_2)) \cdot \\ & ((y_2(x_2' + x_3) + y_2' x_2 x_3') \Downarrow (x_1 + x_2)) \cdot \\ & ((y_3(x_2 x_4 + x_3') + y_3' x_3 (x_2' + x_4')) \Downarrow (x_1 + x_2)) \cdot (x_1 + x_2)]. \end{aligned}$$

Applying *restrict* is simple in these cases. In particular, the second and third term of the transition relation do not depend on x_1. This variable is therefore quantified in $x_1 + x_2$, resulting in the constant 1 function. Hence:

$$\begin{aligned} \mathit{Img}(f,g) &= \exists_{x_1 x_2 x_3 x_4}[y_1 \cdot (y_2(x_2' + x_3) + y_2' x_2 x_3') \cdot \\ & \quad (y_3(x_2 x_4 + x_3') + y_3' x_3 (x_2' + x_4')) \cdot (x_1 + x_2)] \\ \\ &= \exists_{x_2 x_3 x_4}[y_1 \cdot (y_2(x_2' + x_3) + y_2' x_2 x_3') \cdot \\ & \quad (y_3(x_2 x_4 + x_3') + y_3' x_3 (x_2' + x_4')) \cdot \exists_{x_1}(x_1 + x_2)] \\ \\ &= y_1 \cdot \exists_{x_2 x_3}[\exists_{x_4}(y_3(x_2 x_4 + x_3') + y_3' x_3 (x_2' + x_4')) \cdot \\ & \quad (y_2(x_2' + x_3) + y_2' x_2 x_3')] \\ \\ &= y_1 \cdot \exists_{x_2 x_3}[(y_3 x_3' + y_3' x_3 + y_3 x_2) \cdot (y_2(x_2' + x_3) + y_2' x_2 x_3')] \end{aligned}$$

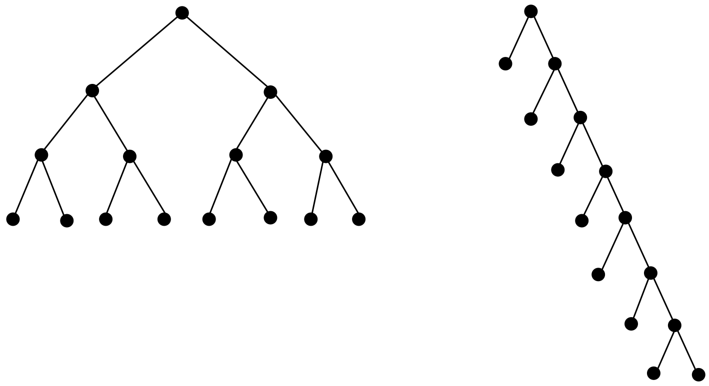


Figure 26: Different tree organizations for the partitioned computation of images.

$$= \quad y_1 \cdot \exists_{x_2 x_3} [y_2 y_3 x_2' x_3' + y_2' y_3 x_2 x_3' + y_2 y_3' x_3 + y_2 y_3 x_2 x_3]$$

$$= \quad y_1 \cdot (y_2 y_3 + y_2' y_3 + y_2 y_3') = y_1 \cdot (y_2 + y_3).$$

In this example, Equation 7 was particularly handy. However, it has been observed in practice that Equation 5 is often more efficient.

Several issues need to be considered in defining an algorithm based on Equations 5–7, besides the choice of what equation to adopt. The first is the general organization of the image computation, which consists of the product of several factors, and the quantification of part of the variables. If the product of two factors is taken at each step, a binary tree results. The leaves of the tree correspond to the factors of Equations 5–7. Each internal node corresponds to the product of its two children and the quantification of all the variables appearing in the result that do not appear in the rest of the tree. The tree is visited in post-order and at the end of the visit, one of Equations 5–7 is computed at the root of the tree. There are many different trees with the same number of leaves. Figure 26 shows the two extreme cases: A perfectly balanced tree and a totally unbalanced one. The number of nodes in the two trees is the same. For m leaves, both trees have $m - 1$ internal nodes.

A second, important issue is the order of the factors, that is, how the factors of Equations 5–7 are mapped onto the leaves of the binary tree. The order of the factors should try to minimize the size of the intermediate results. In one strategy, the factors f_i are ordered so as to minimize the increase in total support (see the output ordering for the BDDs of multiple-output functions). In another approach, the order is chosen so as to maximize the number of variables that will be quantified early. This is done by looking at the *private support* of each term, defined as the variables that only appear in that term. Practical solutions combine in various ways these and other heuristics [29, 51].

A final consideration concerns the clustering of the factors in Equations 5–7. If many image computations must be performed, it may be convenient to partition the factors into a few blocks, and then take the product of each block. This decreases the number of products that must be taken during image computation. There is a trade-off between the reduction in the number of products and the cost of each product. The usual approach to decide how many blocks should be created is to impose a limit on the size of the BDD for each block. One factor is chosen as the seed for a block, and other factors are conjoined to it, until the product grows larger than a given threshold. The procedure is then repeated on the remaining factors. Notice that clustering the blocks has the effect of changing the structure of the computation tree.

8.5.2 Combining Conjunction and Quantification

In the method described in the previous section, at each node of the tree two BDDs are conjoined and some variables are existentially quantified from the result. It is possible to gain some efficiency in terms of both CPU time and memory by combining the two steps into a single operation. The algorithm that computes $\exists_c(f \cdot g)$ is based on the usual recursion. As in the case of the quantification algorithm of Section 5.3.3, c is the cube of the variables to be quantified, t is the index of the lowest indexed variable of f and g and u is the index of the top variable of c. We shall write $c = x_u \tilde{c}$, where $\tilde{c}$ is a cube, possibly the function 1. If $c = 1$ or $f \cdot g$ is constant (either factor is 0 or both are 1) then $\exists_c(f \cdot g) = f \cdot g$. Otherwise, we consider the three following cases.

$t > u$. In this case f and g do not depend on x_u. Hence,

$$\exists_c(f \cdot g) = \exists_{\tilde{c}}(f \cdot g).$$

$t = u$. In this case we write:

$$\begin{aligned} \exists_c(f \cdot g) &= \exists_{\tilde{c}}(\exists_{x_t}(x_t, \text{AND}(f_{x_t}, g_{x_t}), \text{AND}(f_{x'_t}, g_{x'_t}))) \\ &= \exists_{\tilde{c}}(\text{AND}(f_{x_t}, g_{x_t})) + \exists_{\tilde{c}}(\text{AND}(f_{x'_t}, g_{x'_t})) \\ &= \text{OR}(\exists_{\tilde{c}}(f_{x_t} \cdot g_{x_t}), \exists_{\tilde{c}}(f_{x'_t} \cdot g_{x'_t})). \end{aligned}$$

If $\exists_{\tilde{c}}(f_{x_t} \cdot g_{x_t}) = 1$, then the result is 1 and it is not necessary to perform the second recursive call.

$t < u$. In this case we expand with respect to x_t and write:

$$\begin{aligned} \exists_c(f \cdot g) &= \exists_c(x_t, \text{AND}(f_{x_t}, g_{x_t}), \text{AND}(f_{x'_t}, g_{x'_t})) \\ &= (x_t, \exists_c(f_{x_t} \cdot g_{x_t}), \exists_c(f_{x'_t} \cdot g_{x'_t})). \end{aligned}$$

8.5.3 Partitioning, Identical Subproblems, and Splitting

Partitioning works in the case of the transition relation as in the case of the transition function, thanks to the following identity:

$$\exists_{xy}(f(x) \cdot g(y)) = \exists_x f(x) \cdot \exists_y g(y).$$

The recognition of identical sub-problems can also be made to work in the following way. Suppose that $T_i = y_i A + y'_i B$, the i-th term of Equations 5–7, has y_i as top variable. Suppose next that no y_k, $k \neq i$ appears in the the i-th term. This is certainly true if no terms are combined. Terms T_i and T_j represent identical sub-problems if $A \cdot B = 0$ and:

$$T_j = y_j A + y'_j B \quad \text{or} \quad T_j = y_j B' + y'_j A'.$$

They represent complementary subproblems if $A \cdot B = 0$ and:

$$T_j = y_j A' + y'_j B' \quad \text{or} \quad T_j = y_j B + y'_j A.$$

If terms T_i and T_i represent identical sub-problems, $T_i \cdot T_j$ is replaced by $(y_i y_j + y'_i y'_j) \cdot T_i$. If they represent complementary sub-problems, $T_i \cdot T_j$ is replaced by $(y_i y'_j + y'_i y_j) \cdot T_i$. The condition $A \cdot B = 0$ can be tested efficiently. It is also possible to test for a stronger condition, namely $A = B'$, in constant time. Notice that before *restrict* or *constrain* are applied, this stronger condition is always satisfied, since all terms are of the form $y_i \equiv f_i$.

Finally, it is possible to use a table to store the results of image computations, with an approach similar to the one taken for the transition function methods. Identical problems may occur at different iterations of the breadth-first traversal. It is also possible that identical sub-problems arise in the course of one computation, if the image computation is decomposed according to the following identity:

$$
\begin{aligned}
&\exists_{x_1 \cdots x_n}(\prod_{i=1}^{m} T_i(y_i, x_1, \ldots, x_n)) = \\
&= \quad y_j \cdot \exists_{x_1 \cdots x_n}(T_j(y_j, x_1, \ldots, x_n)_{y_j} \cdot \prod_{i=1}^{m, i \neq j} T_i(y_i, x_1, \ldots, x_n)) + \\
&\qquad y'_j \cdot \exists_{x_1 \cdots x_n}(T_j(y_j, x_1, \ldots, x_n)_{y'_j} \cdot \prod_{i=1}^{m, i \neq j} T_i(y_i, x_1, \ldots, x_n)).
\end{aligned}
$$

The similarity of this decomposition to the output splitting technique is apparent. The identity can be generalized by observing that the essential requirement for output splitting is that no term except T_j depend on y_j. Notice that Theorem 8 can be applied to the two sub-problems. There is also the equivalent of input splitting, thanks to the following identity:

$$
\begin{aligned}
&\exists_{x_1 \cdots x_n}(\prod_{i=1}^{m} T_i(y_i, x_1, \ldots, x_n)) = \\
&= \quad \exists_{x_1 \cdots x_{j-1} x_{j+1} \cdots x_n}(\prod_{i=1}^{m} T_i(y_i, x_1, \ldots, x_n)_{x_j}) + \\
&\qquad \exists_{x_1 \cdots x_{j-1} x_{j+1} \cdots x_n}(\prod_{i=1}^{m} T_i(y_i, x_1, \ldots, x_n)_{x'_j}).
\end{aligned}
$$

We finally notice that, although we have so far assumed that there is one term for each next state variable, it is possible to reduce the number of terms by selectively taking the product of groups of terms. This may be advantageous if the BDD obtained by taking the product is not larger than the sum of the operands. This technique can be seen as a way of altering the structure of the evaluation tree.

8.6 Renaming Variables in the Traversal Procedure

In a FSM we distinguish present state variables from next state variables. When the image of the next state function is computed, a set of next states is found that is reachable in one transition from a given set of present states. At each iteration of the traversal procedure, after the set of newly reached states is computed, we need to manipulate it as if it were a set of present states. This requires that the result of the image computation be expressed with the same variables that are used to represent the sets of present states (*Reached*, *From*). This problem is solved differently depending on the algorithm used for image computation.

In the case of the transition function, one can directly use the same variables for *From* and *To*. There is no problem there, because the BDDs for $\delta \downarrow$ *From* and *To* are separate entities at all times. Therefore, one can implicitly apply the renaming of the variables that corresponds to clocking the state register to make the next states present.

In the case of the transition relation, it is not possible to use the same variables for both present and next states, because these variables appear together in the characteristic function of the transition relation. We have to wait until a present state variables has been quantified, before renaming the corresponding next state variable. (The simplest approach is to rename the variables after the image is computed.)

The renaming of the variables is best seen as a case of function composition. Replacing y_i by x_i in the characteristic equation of the image is equivalent to computing:

$$To|_{y_i=x_i} = To(y_1, \ldots, y_{i-1}, x_i, y_{i+1}, \ldots, y_n),$$

that we know we can compute as:

$$To|_{y_i=x_i} = x_i \cdot To_{y_i} + x_i' \cdot To_{y_i'}.$$

When all next state variables must be replaced, it is advantageous to perform the substitution in a single pass, by applying the following identity:

$$To|_{y_1=x_1,\ldots,y_n=x_n} = x_1 \cdot (To_{y_1})|_{y_2=x_2,\ldots,y_n=x_n} + x_1' \cdot (To_{y_1'})|_{y_2=x_2,\ldots,y_n=x_n}.$$

This identity is the basis for a simple recursive algorithm.

8.7 Variable Selection in Image Computation

We have seen that for both the transition function and the transition relation methods image computation can be decomposed by input and output splitting. An obvious question concerns the criterion by which to choose the variable to be used for the expansion. We first consider input splitting for the transition function method. Remember that in this case, we have a vector of BDDs, one for each output. These BDDs only depend on the input variables. The simplest choice is therefore the variable with lowest index appearing in those BDDs. The advantages of this method are that the cofactors can be computed trivially and that no new nodes are created. The choice of the top variable is therefore attractive from the point of view of memory. However, we may be willing to trade some memory occupation for faster computation, by choosing a splitting variable that reduces the search space more than the top variable. To that effect, we notice that the single most dramatic factor in reducing the size of the search space is partitioning based on disjoint support. Hence we look for splitting variables whose choice leads to earlier decomposition of the image computation. According to this criterion, good candidates are those variables on which many outputs depend. This simple heuristic is fairly simple to implement and has excellent performance [37].

In the method based on the transition function, output splitting does not necessarily reduce the support of the residual functions. In the transition relation method, though, both kinds of splitting can be used to reduce the supports of the residual functions.

8.8 Backward Traversal

The form of reachability analysis we have examined so far can be considered *forward* traversal: We start from the initial states and we trace paths forward. Those states that are along the paths are clearly reachable from some initial state. If we want to know whether a given property holds at all reachable states of a finite state machine, we can find all the reachable states and then examine them for possible violations of the property. In finite state machine equivalence, the property is $\lambda_\times = 1$ for the product of the two machines being compared.

There is an alternative approach to the same problem: We identify all the states where the property is violated, and then find whether any of these states is reachable from any initial state. Finding whether a state is reachable from any initial state can be done by tracing a path *backward* in the state transition graph. Tracing paths backward requires the computation of the states from which a given set of states is reachable. This goes under the name of *pre-image* computation. Therefore, we now examine pre-image computation, and then we discuss the relative merits of forward and backward traversal. As in the case of image computation, there are two algorithms for pre-image computation: One is based on the transition function, and one is based on the transition relation. For the sake of conciseness, we only consider the latter.

8.8.1 Pre-Image Computation Based on the Transition Relation

The computation of the pre-image based on the transition relation can be performed with the image computation algorithm. It is sufficient to interchange the roles of input and output variables.

Example 19 Let us consider the following transition relation:

$$\begin{aligned} T(a,b,c,y_1,y_2,y_3) &= (y_1ab + y_1ac + y_1'a' + y_1'b'c')(y_2ab + y_2bc + y_2'b' + y_2'a'c') \\ &\quad (y_3ac + y_3bc + y_3'c' + y_3'a'b') \end{aligned}$$

where a, b, and c are the input variables and y_1, y_2, and y_3 are the output variables. Suppose we want to compute the pre-image of y_1y_3. With argument similar to that employed for image computation, we can write:

$$\begin{aligned} \mathit{Pre}(f,C) &= \exists_{y_1y_2y_3}(T(a,b,c,y_1,y_2,y_3) \cdot y_1y_3) \\ &= \exists_{y_1y_2y_3}(T(a,b,c,y_1,y_2,y_3) \downarrow y_1y_3) \\ &= \exists_{y_1y_2y_3}(T(a,b,c,y_1,y_2,y_3) \Downarrow y_1y_3) \end{aligned}$$

In this case the two generalized cofactors are identical, because the constraint is a cube. Performing the computation, we get:

$$\begin{aligned} \mathit{Pre}(f,C) &= \exists_{y_2}((ab+ac)(y_2ab + y_2bc + y_2'b' + y_2'a'c')(ac+bc)) \\ &= (ab+ac)(ac+bc) = ac \end{aligned}$$

In general, when the transition relation is given in the form:

$$T = \prod_{i=1}^{m}(y_i \equiv f_i),$$

and the constraint is a cube, considerable simplifications occur. Since the cofactor of a function with respect to a cube does not depend on the variables appearing in the cube and cofactoring with respect to a cube distributes, all terms of the product contain zero or one output variables after cofactoring. The existential quantification fully distributes, since each output variable appears in at most one term. Finally, if a term is still of the form $(y_i \equiv f_i)$ after the cofactoring (this is the case if y_i does not appear in the cube), then the existential quantification returns 1. Otherwise, it returns either f_i or f_i'. Therefore we see that the result of the pre-image computation is simply the product of those f_i such that y_i appears in the constraint cube, taken with the appropriate phase. In our example, we can write:

$$T = (y_1 \equiv (ab+ac))(y_2 \equiv (ab+bc))(y_3 \equiv (ac+bc))$$

and one can verify that the previous result is correct with minimum computation.

When the constraint is not a cube, the general algorithm based on interleaving quantification and product applies, as in the case of image computation.

8.8.2 Forward versus Backward Traversal

The relative effectiveness of the two methods depends primarily on the structure of the graph being traversed. Suppose, for instance, that we are comparing two identical counters for equivalence. We know that the forward traversal method has difficulty in dealing with this example, because of the large sequential depth of the produce machine. If we are comparing two modulo-m counters, each with one initial state, breadth-first forward traversal will require m iterations. At each iteration, one new state is added to the set of reachable states, and this accounts for the ineffectiveness of the procedure.

By contrast, in backward traversal, we initialize our search to all states of the product machine that give different counts in the two counters. This set of states has a compact characteristic function. A single pre-image computation leads to convergence, as a one minute's thought will confirm: The pre-image is precisely the same set of states that we started from, and it does not include the initial state. Hence, we arrive immediately at the conclusion that the two counters are equivalent.

Clearly, not all examples work like the counters. If the set of states where a violation occurs does not have a compact characteristic function, or if the machine to be traversed is not deep, forward traversal will typically perform better.

8.9 Transitive Closure and Iterative Squaring

Breadth-first symbolic traversal is remarkably efficient, in that it allows realistic machines with very many states (more than 10^{120}) to be traversed. The efficiency stems from the ability to process many states at one time. It is then easy to see that the advantage of breadth-first symbolic traversal is lost when the FSM to be traversed is, for instance, a counter. A counter is characterized by a very high sequential depth—it actually equals the number of states. In this section we consider a technique, called *iterative squaring*, that addresses this problem [12]. The basic idea is quite simple. Let $R(x, w, y)$ be the transition relation of FSM M, where x is the vector of present state variables, w is the vector of primary input variables, and y is the set of next state variables. R describes triples of the form $(\hat{x}, \hat{w}, \hat{y})$ such that from state $\hat{x}$ of M, state $\hat{y}$ can be reached by applying input $\hat{w}$. If we existentially quantify all primary input variables from R, we obtain another relation:

$$T(x, y) = \exists_w R(x, w, y),$$

that describes pairs of adjacent states, i.e., states that are connected by at least one edge in the transition graph of M. We can compute the *transitive closure* of T, that we call $C(x, y)$ in a way to be described shortly. The transitive closure C describes the pairs of states that are connected by at least one path in the state graph of M. Finding the set of states reachable from S^0 then amounts to computing:

$$\mathit{Reached}(y) = \exists_x (S^0(x) \cdot C(x, y)) + S^0(y),$$

since the product of S^0 and C yields the pairs of states connected by a path, such that the first state of the pair is an initial state; the quantification returns the second components of the pairs. The term $S^0(y)$ is necessary, because some initial states may not be reachable from any other initial state.

The transitive closure can be found by computing the least fixed point of the following recurrence equation:

$$C(x, y) = T(x, y) + \exists_z (C(x, z) \cdot C(z, y)).$$

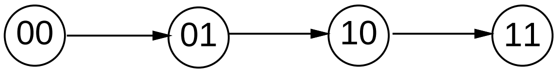


Figure 27: A state transition graph.

The term $C(x,z) \cdot C(z,y)$ gives the name of iterative squaring to this technique and is responsible for its efficiency. In order to find $C(x,y)$, $C_0(x,y)$ is set to $T(x,y)$ and then

$$C_i(x,y) = T(x,y) + \exists_z (C_{i-1}(x,z) \cdot C_{i-1}(z,y))$$

is computed until $C_n(x,y) = C_{n-1}(x,y)$ for some n. To see how the recurrence equation actually computes the transitive closure of T, suppose that $C_{i-1}(x,y)$ describes all the pairs of states (s_1, s_2) such that s_2 can be reached from s_1 in at least 1 and at most 2^{i-1} steps. This is certainly true for $i = 1$. We can then prove that C_i describes all the pairs of states (s_1, s_2) such that s_2 can be reached from s_1 in at least 1 and at most 2^i steps. This follows from observing that $C(x,z) \cdot C(z,y)$ describes triples of states (s_1, s_2, s_3), such that s_2 (s_3) is reachable from s_1 (s_2) in at least 1 and at most 2^{i-1} steps. Hence s_3 is reachable in at least 2 and at most 2^i steps from s_1. The pairs (s_1, s_3) obtained by quantification give all pairs of states that satisfy that condition. By adding $T(x,y)$, the pairs of states that are precisely one step apart are included, thus proving our statement.

Example 20 Consider the state transition graph of Figure 27, where the primary inputs do not appear for simplicity. We have:

$$T(x,y) = x_1'x_2'y_1'y_2 + x_1'x_2y_1y_2' + x_1x_2'y_1y_2.$$

The computation of the transitive closure proceeds as follows:

$$\begin{aligned}
C_0(x,y) &= T(x,y) \\
\\
C_1(x,y) &= T(x,y) + \exists_{z_1z_2}((x_1'x_2'z_1'z_2 + x_1'x_2z_1z_2' + x_1x_2'z_1z_2) \cdot \\
&\quad (z_1'z_2'y_1'y_2 + z_1'z_2y_1y_2' + z_1z_2'y_1y_2)) \\
&= T(x,y) + x_1'x_2'y_1y_2' + x_1'x_2y_1y_2 \\
&= x_1'x_2'(y_1'y_2 + y_1y_2') + x_1'x_2(y_1) + x_1x_2'y_1y_2 \\
\\
C_2(x,y) &= T(x,y) + \exists_{z_1z_2}((x_1'x_2'(z_1'z_2 + z_1z_2') + x_1'x_2(z_1) + x_1x_2'z_1z_2) \cdot \\
&\quad (z_1'z_2'(y_1'y_2 + y_1y_2') + z_1'z_2y_1 + z_1z_2'y_1y_2)) \\
&= T(x,y) + x_1'x_2'y_1 + x_1'x_2y_1y_2 \\
&= x_1'x_2'(y_1 + y_2) + x_1'x_2y_1 + x_1x_2'y_1y_2 \\
&= C(x,y)
\end{aligned}$$

One can verify that $C(x,y) \cdot C(y,x) = 0$, consistent with the fact that the graph of Figure 27 has no cycles.

It should be noted that the number of iterations required to compute the transitive closure is logarithmic in the maximum distance between two states[19]. The transitive closure also considers paths originating at

[19] This is also called the *diameter* of the state transition graph.

states that cannot be reached from the initial states. Hence it may be inefficient, if a large fraction of the state graph is unreachable or if the sequential depth of the machine is low. The major problem with the transitive closure is that the BDDs that are produced during the computation may grow too large.

8.10 Approximate Reachability Analysis

Even with the best algorithms and heuristics, there remain many circuits that are too difficult for the reachability techniques that we have discussed so far. Circuits with a few thousand latches are normally too difficult, and sometimes, even circuits with less than a hundred latches pose significant problems.

In such cases, however, it may still be possible to perform an approximate reachability analysis. Specifically, it may be possible to find a superset of all reachable states. Alternatively, it may be possible to compute a subset of all reachable states.

Both approximations may be useful. Suppose we want to prove an invariant. Let us examine the superset first. If the invariant hold for all the states of the superset, then it holds for all reachable states. On the other hand, if the invariant does not hold for some states of the superset, it may be that the states where violations occur are not really reachable. Therefore, verification based on supersets of the reachable states may produce *false negatives*, or, in other words, it is *conservative*. If we prove a circuit correct by applying a conservative technique, we know it is correct. On the other hand, for conservative verification to be useful, false negatives should be relatively rare, because proving a false negative false may be impractical, or at least very time consuming.

The closer the superset of reachable states to the actual set, the lower the chance of getting false negatives. On the other hand, better approximations require more time and memory to be computed. Notice that a trivial solution to the approximation of the reachable state set from above is given by the set of all 2^n states, where n is the number of latches.

Let us now turn our attention to the computation of subsets of the reachable states. A reachability analysis method based on forward traversal produces a sequence of sets of states that eventually converges to the set of all reachable states. The elements of the sequence provide increasingly accurate approximations from below to the exact solution. If we cannot complete reachability analysis, we can always stop when we run out of memory or time, and claim that we have obtained an approximation from below to the solution. The challenge, therefore, lies in devising algorithms that produce good approximations with limited resources.

When verification of an invariant is based on a subset of all reachable states, it is *partial*. It may be that the invariant does not hold, but the states where violations occur have not been visited. On the other hand, if a violation is reported for a state in the subset, then we know that the invariant does not hold. The usefulness of partial verification relies on the fact that a new design normally contain many mistakes, and partial verification may uncover most such mistakes quickly. In this respect, partial verification is not different from simulation: Indeed simulation is one form of partial verification. Methods based on the reachability analysis techniques we have examined thus far, however, can be much more efficient than simulation.

8.10.1 Computing Supersets of the Reachable States

In a sequential circuit, each state variable depends in general on some primary inputs and some state variables. If we replace some connections to state variables with connections to new primary inputs, the resulting circuit will exhibit a wider variety of behaviors: It will be able to do more things than the original circuit. This simple idea is at the heart of efficient methods for the computation of supersets of the reachable states. The circuit is partitioned in submachines, and the connections among the submachines are cut. (See Figure 28.) Once a circuit is partitioned, we can traverse each submachine in isolation. Then, we can take the

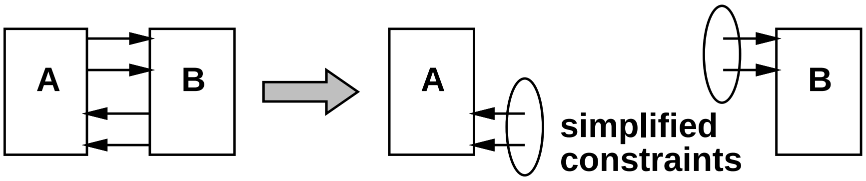


Figure 28: Partitioning for Approximate Reachability Analysis.

superset of the reachable states of the original machine to be the cartesian product of the sets of states of the submachines.

This procedure is simple, but a bit too drastic in simplifying the problem. The fact that the connections among the submachines are completely ignored causes a considerable loss of accuracy. Instead, we can devise several methods to retain in part the flow of information among the submachines. We outline one such approach. We process the submachines in some order. For the first machine, we perform reachability analysis, assuming that the inputs from the other machines are unrestricted. When we consider the second machine, though, we assume that the inputs coming from the first machine cannot take values corresponding to states that were found to be unreachable during the first traversal. We then consider the third machine, constraining the inputs from the first and second machine, and so on.

Once all machines have been considered, we can return to the first one. This time, instead of using unconstrained inputs from the other machines, we use the constraints determined by the first round of reachability analyses. This may produce a smaller reachable set of states then the initial traversal. We continue to refine the approximation, until we reach a point in which no reachable states set shrinks further. At that point we have converged to a superset of the reachable states for the original circuit.

Constraining the traversal of each submachine can be accomplished with minor extensions to the image computation techniques that we already know. Specifically, the constraint used in image computation comes from both the state variables of the submachine, and the inputs fed by the other submachines.

The scheme we have discussed can be improved and modified in several ways. Another important problem is how to partition the circuit into submachines. For these aspects, the reader is referred to [16, 17].

8.10.2 Computing Subsets of the Reachable States

The standard forward traversal algorithm provides a sequence of approximations from below for the reachable states. The challenge is how to provide better approximations within given time and memory limits. We outline one possible approach. We assume that we use BDDs to represent the sets of states visited during reachability analysis. We observe that BDDs are successful because they provide a *dense* representation for sets. Here dense means that the ratio of the cardinality of the set over the size of the representation is high.

Breadth-first traversal is more or less successful, depending on how dense the BDDs are. Therefore, if we want to improve the efficiency of traversal, we may want to increase the density of the BDDs. One way to improve the density is to improve the variable order. However, for some circuits, the set of reached states may still have a large BDD in spite of our best reordering efforts.

Our next observation is that breadth-first search goes through a fixed sequence of sets of states: All the initial states, followed by all the states at distance 0 or 1 from the initial states, and so on. If the states at distance k or less from the initial states are randomly scattered through the state space, their representation as a BDD is likely to be large. However, by visiting the states in some other order than the one imposed by breadth-first search, we may visit states that are tightly packed in some subspace. Such states will have a dense representation.

We are therefore going to mix breadth-first and depth-first search, leaving the density of the BDDs guide our meandering through the state graph: We still use as basic approach the breadth-first search algorithm. However, at each iteration we check the size of the *To* set. If it is too large, we extract a dense subset from *To* using the algorithms of Section 5.6 and we use it instead of *To* as constrain in the image computation.

The states in *To* that are not in the dense subset are discarded. Therefore, we do not have a breadth-first search any more. A few interesting properties of breadth-first search are lost. For instance, when the set of new states is empty, we are not guaranteed that all states have been reached. In spite of this and similar disadvantages, the traversal based on density is appealing, because it may visit many more states than pure breadth-first search in the same amount of time [53, 52].

9 Conclusions

Binary Decision Diagrams are a popular data structure for verification algorithms and for several tasks in the design of calculational systems. We have reviewed the properties of BDDs that make them amenable for such applications and we have presented the main algorithms for their manipulation. We have emphasized the application of BDDs to reachability analysis because reachability analysis is the computational task at the basis of model checking algorithms. Many variants of BDDs have been proposed and several are in use. We have not attempted to give even a cursory treatment of them, even though some of the variants have found applications in verification algorithms. The interested reader will find two good references on the subject in [10, 46].

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 622–627, San Jose, CA, November 1994.

[3] C. L. Berman. Circuit width, register allocation, and reduced function graphs. Technical Report RC 14129, IBM T. J. Watson Research Center, Yorktown Heights, NY, October 1988.

[4] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA, May 1995.

[5] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, Orlando, FL, June 1990.

[6] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[7] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd Design Automation Conference*, June 1985.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] R. E. Bryant. 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, February 1991.

[10] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the International Conference on Computer-Aided Design*, pages 236–243, San Jose, CA, November 1995.

[11] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 403–407, San Francisco, CA, June 1991.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.

[13] J. R. Burch and D. E. Long. Efficient boolean function matching. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–411, Santa Clara, CA, November 1992.

[14] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proceedings of the International Conference on Computer-Aided Design*, pages 570–576, San Jose, CA, November 1998.

[15] G. Cabodi, P. Camurati, and S. Quer. Improved reachability analysis of large finite state machines. In *Proceedings of the International Conference on Computer-Aided Design*, pages 354–360, Santa Clara, CA, November 1996.

[16] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design*, 15(12):1465–1478, December 1996.

[17] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Transactions on Computer-Aided Design*, 15(12):1451–1464, December 1996.

[18] H. Cho, G. D. Hachtel, and F. Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(7):935–945, July 1993.

[19] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[21] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[22] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, November 1990.

[23] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. Presented at the International Workshop on Logic Synthesis, Granlibakken, CA, May 1995.

[24] R. Drechsler, N. Drechsler, and W. Günther. Fast exact minimization of BDDs. In *Proceedings of the Design Automation Conference*, pages 200–205, San Francisco, CA, June 1998.

[25] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, and F. Somenzi. Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 235–241, San Jose, CA, November 1998.

[26] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[27] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 38–41, Santa Clara, CA, November 1993.

[28] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, Amsterdam, February 1991.

[29] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818.

[30] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer-Aided Design*, 15(12):1479–1493, December 1996.

[31] G. D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0-1 networks. *Journal of Formal Methods in Systems Design*, 10(2/3):207–219, 1997.

[32] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10(1):196–210, 1962.

[33] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD minimization using don't cares. In *Proceedings of the Design Automation Conference*, pages 208–213, Anaheim, CA, June 1997.

[34] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the International Conference on Computer-Aided Design*, pages 472–475, Santa Clara, CA, November 1991.

[35] S.-W. Jeong, T.-S. Kim, and F. Somenzi. An efficient method for optimal BDD ordering computation. In *International Conference on VLSI and CAD (ICVC'93)*, Taejon, Korea, November 1993.

[36] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Extended BDD's: Trading off canonicity for structure in verification algorithms. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 464–467, Santa Clara, CA, November 1991.

[37] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering and selection for FSM traversal. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 476–479, Santa Clara, CA, November 1991.

[38] G. Kahmi and L. Fix. Adaptive variable reordering for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 359–365, San Jose, CA, November 1998.

[39] C. Y. Lee. Binary decision programs. *Bell System Technical Journal*, 38(4):985–999, July 1959.

[40] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 88–91, Santa Clara, CA, November 1990.

[41] D. E. Long. The design of a cache-friendly BDD library. In *Proceedings of the International Conference on Computer-Aided Design*, pages 639–645, San Jose, CA, November 1998.

[42] F. Mailhot and G. De Micheli. Technology mapping using boolean matching. In *Proceedings of the European Conference on Design Automation*, pages 180–185, Glasgow, UK, March 1990.

[43] S. Manne, D. C. Grunwald, and F. Somenzi. Remembrance of things past: Locality and memory in BDDs. In *Proceedings of the Design Automation Conference*, pages 196–201, Anaheim, CA, June 1997.

[44] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[45] K. L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. In R. Alur and T. A. Henzinger, editors, *8th Conference on Computer Aided Verification (CAV'96)*, pages 13–25. Springer-Verlag, Berlin, August 1996. LNCS 1102.

[46] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer, Berlin, 1998.

[47] S.-I. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the Design Automation Conference*, pages 52–57, Orlando, FL, June 1990.

[48] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 48–55, Santa Clara, CA, November 1993.

[49] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, San Jose, CA, November 1995.

[50] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 628–631, San Jose, CA, November 1994.

[51] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA., May 1995.

[52] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of decision diagrams. In *Proceedings of the Design Automation Conference*, pages 445–450, San Francisco, CA, June 1998.

[53] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995.

[54] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993.

[55] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *Proceedings of the Design Automation Conference*, pages 635–640, Las Vegas, NV, June 1996.

[56] H. Savoj, R. K. Brayton, and H. J. Touati. Extracting local don't cares for network optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 514–517, Santa Clara, CA, November 1991.

[57] T. R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, October 1996. Memorandum No. UCB/ERL M96/76.

[58] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proceedings of the Design Automation Conference*, pages 225–231, San Diego, CA, June 1994.

[59] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990.

[60] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *Formal Methods in Computer Aided Design*, November 1998.

[61] B. Yang, Y.-A. Chen, R. Bryant, and D. O'Hallaron. Space- and time-efficient BDD construction via working set control. In *Proc. of Asia and South Pacific Design Automation Conference (ASPDAC'98)*, Yokohama, Japan, February 1998.