Chapter 8

Verification With BDDs

8.1 Introduction

This chapter discusses some practical aspects of formal verification. We have pointed out in the previous chapter how certain questions about the behaviour of a digital system may be formulated as CTL formulas and we have also presented some simple model checking algorithms. These algorithms predominantly use sets as a basic data type. In this chapter we will introduce a very versatile data structure to implement not only sets, but Boolean functions and relations as well. This data structure is known as a Binary Decision Diagram (BDD). We will use BDDs to represent the various types of mathematical objects in our programs and show how they can efficiently be manipulated.

8.2 Implementing Mathematical Objects

There are many well-known data structures to implement sets of objects. The one to choose depends on the objects and sizes of the sets to represent and of course also on the kind of operations to be performed on the sets and their elements. Data structures like bit-vectors, linked-lists, binary trees in all their variations (search trees, balanced trees, red-black trees), and hash-tables spring to mind. The latter in particular is a favourite choice for state-space enumeration based verification algorithms: the set of states already processed is kept in a hash-table and look-ups in that table can be performed in almost constant time on average. However, in this chapter we will concentrate on so-called symbolic enumeration techniques: the elements of a set are not represented explicitly, by in some way listing them one-by-one as individual objects, but we will use a description of the set instead and know of its elements only indirectly.

So far, we have encountered the following mathematical objects: sets, sequences, functions, relations, graphs, and logic and temporal formulas. Note that these objects are strongly related: a formula can typically be interpreted over a graph model; a graph may be regarded as a set of vertices with a relation; a sequence can be seen as an index function, assigning a symbol to each position; a function is a special case of a relation; a relation is a set of pairs. It is tempting to pronounce: *"Everything is a set"*. As hardware engineers we all know that the elements of a finite set can be encoded as Boolean numbers. For instance, if the universe is U, then any set over this universe can be recorded as a set of binary numbers of fixed length (= number of bits). If U has $|U|$ elements we could take the word-length of those numbers to be $n = \lceil {}^2\log(|U|) \rceil$. Once a suitable encoding is established, we are solely dealing with sets of binary numbers and functions and relations over such sets.

8.3 Binary Decision Diagrams

A binary decision diagram is a graph-based data structure that can be used to represent Boolean functions. The graph is binary, directed and acyclic. Binary here means that every vertex apart from the terminal vertices, which have no successors, has precisely two successors. This data structure can easily be made canonical: with each Boolean function over a fixed number of arguments we can uniquely associate a BDD. We will shortly see what restrictions need to be imposed on the BDD to achieve this.

Let it be clearly understood right up front: BDDs are no panacea. If every Boolean function could be efficiently represented by a BDD and it is the case that testing two BDDs for equivalence can be done in constant time, then we would have solved the $NP = P?$ problem of theoretical computer science. The second statement is true however, BDDs can indeed be tested for equivalence in constant time with a clever implementation. The first statement turns out to be false: there exist Boolean functions whose BDDs will grow exponential in size, taking the number of vertices in a BDD as a measure for its size. All we can hope for is that in practice many interesting functions have moderately sized BDDs.

8.3.1 Definition

A binary decision diagram is a (possibly multi-rooted) directed acyclic graph $G(V, E)$ with

V: a set of labelled vertices. Vertices without any successors in the graph are called terminal vertices; all other vertices are referred to as non-terminal or internal vertices. The non-terminals are labelled by a variable $x_i \in X$. There are precisely two terminal vertices: one labelled with 0 and the other one labelled with 1. For convenience we refer to these vertices as the 0-vertex and 1-vertex. Note that in general different internal vertices may have the same label.

$E \subseteq V \times V$: a set of directed edges, labelled by 0 (the else-edges) or 1 (the then-edges). We often drop the labels on the edges and use the convention that an edge drawn on the left is a then-edge, an edge drawn on the right is an else-edge. By $\text{then}(v)$ we will denote the successor vertex of v obtained via the then-edge. $\text{else}(v)$ is defined likewise.

X is a finite, ordered set of variables: $x_0 < x_1 < x_2 \cdots < x_{n-1}$. $\text{rank}(v)$ denotes the rank order number of the label of vertex v, i.e., the position of the variable labelling the vertex in the variable order. For convenience we define $\text{rank}(0) = \text{rank}(1) = \infty$.

In order to achieve canonicity (not proven here) we put the following restrictions on the occurrences of variable labels in the BDD and on its structure:

1. $\underset{v \in V \setminus \{0,1\}}{\forall} \text{rank}(v) < \min(\text{rank}(\text{then}(v)), \text{rank}(\text{else}(v)))$.
2. **no** internal vertex v has $\text{then}(v) = \text{else}(v)$, and
3. **no** isomorphic subgraphs are present, i.e., subgraphs with identical structure and identical labelling.

BDDs complying with the above 3 rules are called *Reduced Ordered BDDs*. In our discussion we will always assume the BDDs to be reduced and ordered, unless otherwise noted.

8.3.2 Interpretation

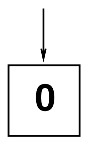
Consider all possible BDDs over a given set $X = \{x_0, x_1, \cdots, x_{n-1}\}$ of variables. With each BDD we can (uniquely because of canonicity) associate a Boolean function $f : B^n \to B$ by means of an interpretation function $\mathbf{I} : V \to (B^n \to B)$. The Boolean functions of interest all take n Boolean argument values and produce a single Boolean value as result. It's easy to calculate that there are 2^{2^n} different functions over n arguments and hence there are the same number of distinct BDDs over n variables. We will use a notation borrowed from λ-calculus to denote functions. The functions of interest will be denoted as $\lambda \underline{x}. E(\underline{x})$, where $\underline{x} = x_0, x_1, \cdots, x_{n-1}$ is the sequence (or vector) of the function place-holder variables that stand for arbitrary Boolean argument values and $E(\underline{x})$ is some Boolean expression (logical formula) in terms of the x_i variables (atomic propositions). The Boolean expressions are built from the usual logical operators $\neg$, $\wedge$, and $\vee$, and the logical constants false and true. The interpretation function is inductively defined as follows:

For the terminal vertices 0 and 1, $\mathbf{I}(0) = \lambda \underline{x}.\,\mathsf{false}$, $\mathbf{I}(1) = \lambda \underline{x}.\,\mathsf{true}$.

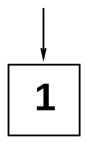
For a non-terminal vertex v with label x_i and successors $\mathsf{then}(v) = T$ and $\mathsf{else}(v) = E$, $\mathbf{I}(v) = \lambda \underline{x}.\, x_i \wedge \mathbf{I}(T) \vee \neg x_i \wedge \mathbf{I}(E)$.

With this definition of $\mathbf{I}$ we should be able to interpret any BDD vertex as a Boolean function, simply by recursing from that vertex 'downwards' in the graph. Of course we can use the identities of Boolean algebra to simplify the expression if necessary. The BDD graphs for 0, 1, and a single variable x are called the elementary BDDs. Figure 8.1 shows their graphs and gives their interpretation as a Boolean function.

The *false* or *zero* function: $f = \lambda \underline{x}.\,\mathsf{false}$:



The *true* or *one* function: $f = \lambda \underline{x}.\,\mathsf{true}$:



A single variable x as a projection function: $f = \lambda \underline{x}.\, x$

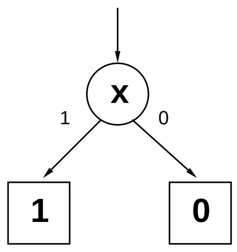


Figure 8.1. Elementary BDDs.

These elementary BDDs can be used as building blocks to construct BDDs for more complicated functions. Figure 8.2 depicts the BDD for the function $f = \lambda a, b, c.\, a \wedge b \vee c$.

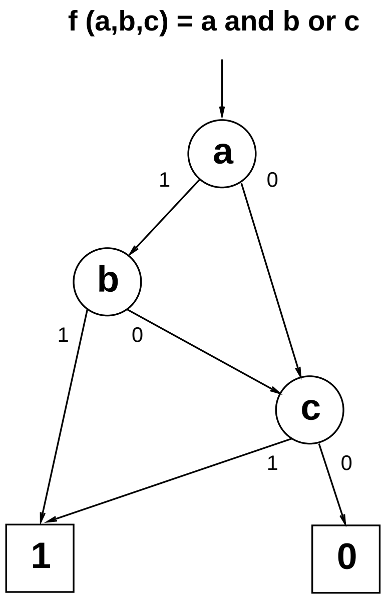


Figure 8.2. Example of a reduced ordered BDD.

8.3.3 BDD construction algorithm

We like to have algorithms for operations on BDDs. In particular we are interested in how to perform Boolean operations on BDDs, i.e., and-ing two BDDs, or-ing two BDDs, and complementing a BDD. Clearly, if we carefully select a small set of operators, it will be possible to express many others in terms of them. There is one ternary operator which is of special interest: the ITE operator. ITE stands for *If-Then-Else* and its meaning in terms of Boolean function operands F, G, and H, may be defined by

$$\mathrm{ITE}(F, G, H) = \text{if } F \text{ then } G \text{ else } H = F \cdot G \vee \overline{F} \cdot H.$$

Note that with abuse of notation we use the Boolean operators as higher-level operators, operating on Boolean functions instead of merely Boolean truth-values. Note also that we use a bar ($\bar{\ }$) to indicate complementation ($\neg$). Table 8.1 lists a number of familiar Boolean operators and indicates how they can be expressed by the ITE operator.

The ITE operator naturally coincides with the definition of the BDD vertex interpretation function **I**, in fact we have $\mathbf{I}(v) = \mathrm{ITE}(x, T, E)$ where v is the vertex $\langle x, T, E \rangle$, i.e., the vertex labelled with x and having successors T and E in the then and else direction respectively. The decomposition of a Boolean function f into its so-called cofactors around a certain variable x is known as the Shannon decomposition. We usually write Shannon decomposition as:

$$f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}}$$

Here + stands for logical 'or' and $\cdot$ is used to indicate logical 'and'. f_x is the so-called positive cofactor of f, i.e., the restriction of f with respect to x, or $f_x = f|_{x=1}$. Similarly, for the negative cofactor $f_{\bar{x}} = f|_{x=0}$.

Name:	Notation:	ITE form:
not	$\overline{F}$	ITE (F, 0, 1)
and	$F \cdot G$	ITE (F, G, 0)
xor	$F \oplus G$	ITE (F, $\overline{G}$, G)
or	$F \vee G$	ITE (F, 1, G)
nor	$\overline{F \vee G}$	ITE (F, 0, $\overline{G}$)
equiv	$F \leftrightarrow G$	ITE (F, G, $\overline{G}$)
implies	$F \rightarrow G$	ITE (F, G, 1)
nand	$\overline{F \cdot G}$	ITE (F, $\overline{G}$, 1)

Table 8.1. Some logical operators expressed by ITE.

For a BDD with top vertex labelled x we have $\mathbf{I}(T) = f_x$ and $\mathbf{I}(E) = f_{\bar{x}}$ (figure 8.3).

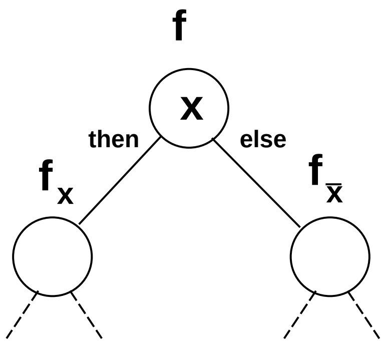


Figure 8.3. Shannon cofactors related to BDDs.

Using the fact that cofactoring distributes through all Boolean operations, we derive the following recurrent expression for ITE:

$$\text{ITE}(F, G, H)$$

$$= x \cdot \text{ITE}(F, G, H)_x + \bar{x} \cdot \text{ITE}(F, G, H)_{\bar{x}}$$

$$= x \cdot (F \cdot G + \overline{F} \cdot H)_x + \bar{x} \cdot (F \cdot G + \overline{F} \cdot H)_{\bar{x}}$$

$$= x \cdot (F_x \cdot G_x + \overline{F}_x \cdot H_x) + \bar{x} \cdot (F_{\bar{x}} \cdot G_{\bar{x}} + \overline{F}_{\bar{x}} \cdot H_{\bar{x}})$$

$$= x \cdot (\mathrm{ITE}(F_x, G_x, H_x)) + \bar{x} \cdot (\mathrm{ITE}(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$$

$$= \mathrm{ITE}(x, \mathrm{ITE}(F_x, G_x, H_x), \mathrm{ITE}(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}}))$$

and the latter is represented by the BDD:

$$\langle x, \mathrm{ITE}(F_x, G_x, H_x), \mathrm{ITE}(F_{\bar{x}}, G_{\bar{x}}, H_{\bar{x}})\rangle$$

We use this Shannon-expansion based recurrent expression for ITE as the basis for an implementation. Observe that the necessary cofactors can easily be calculated as follows. Choose the 'splitting' variable x such that it is the smallest in rank among the top variables of the argument BDDs F, G, and H. Then for any of these arguments, either x equals its top variable and the cofactors are the successors vertices, or x is smaller than the top variable which means the function does not depend on x and hence the cofactors are equal to the function itself. Clearly, our algorithm will be recursive. As bottom cases we choose $F = 0$ and $F = 1$. The above deliberations are summarized in algorithm 8.1.

```
BDD ite(BDD F,G,H)
{
  /* Bottom cases: */
  if (F = 0) return H;
  if (F = 1) return G;

  /* Recursive case: */
  x:=min(var(F),var(G),var(H));
  T:=ite($F_x$,$G_x$,$H_x$);
  E:=ite($F_{\bar{x}}$,$G_{\bar{x}}$,$H_{\bar{x}}$);
  if (T = E) return T;
  return the vertex < x,T,E >;
}
```

Algorithm 8.1. Implementation of the ITE operator for BDDs.

In practice, this algorithm is refined to ensure that when the arguments are reduced ordered BDDs the result is reduced and ordered as well. The proper ordering is ensured by the selection of the variable x. Reduction is achieved by keeping track of all vertices created so far, and checking whether a vertex already exists before creating a new one. We can speed up the algorithm by introducing memoization, which is a technique that avoids recomputation of the result for identical arguments, by simply recording all prior calls and their results in a (hash) table.

8.3.4 Constructing BDDs for a circuit

As a first example of the use of BDDs we consider combinational circuits. A combinational circuits can be represented by logic functions, i.e., each primary output bit can be associated with a Boolean function over the primary input bits. To check whether two circuits implement the same functionality, we can construct BDDs for all outputs of both circuits and compare them pairwise. The construction of a BDD for a primary output is illustrated in figure 8.4.

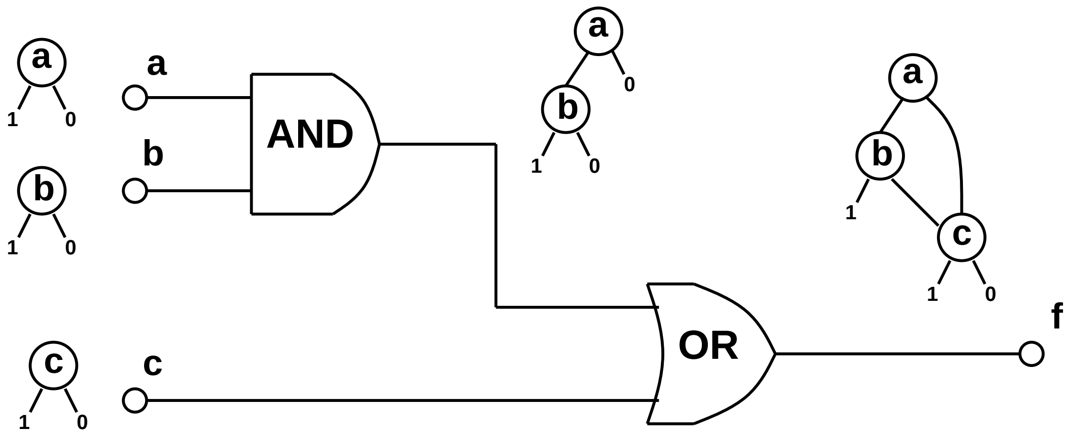


Figure 8.4. Deriving the BDD for a logic circuit.

We see that the process proceeds from primary inputs to primary outputs. First elementary projection BDDs are created, one for each primary input. At each gate we construct the BDD for its output by applying the appropriate ITE form to the BDDs associated with the gate's inputs. Eventually we obtain the BDDs for the primary outputs.

8.3.5 Disadvantages of BDDs and how to overcome them

As mentioned earlier, BDDs may 'blow-up'. With this we mean that for some Boolean functions the BDDs will have a number of vertices that exponentially depends on the number of variables. We might suggest that this can be remedied by choosing a different variable ordering. Unfortunately, there are Boolean functions for which the BDD size will be exponential no matter what variable ordering we use. On the other hand, changing the variable order might help for some functions, i.e., there are functions that have an exponentially sized BDD for one ordering, but are of polynomial size for a different ordering. So to be on the safe side, changing the order of the variables when the BDDs tend to grow large is generally a good idea. Although we will not go into the details of how this is accomplished, it is interesting to analyse the effect *dynamic* variable ordering can have on the BDD size. Dynamic variable ordering is an optimisation technique that rearranges the variables during BDD operations. Table 8.2 gives an overview of the results obtained for several benchmark circuits.

Circuit	Good		Bad		Bad+Dynamic	
	#nodes	secs	#nodes	secs	#nodes	secs
16 ROT	81	<1	1081328	56	81	1
8 ADD	36	<1	751	<1	36	<1
16 ADD	76	<1	196575	16	123	1
32 ADD	156	<1	>1000000	80	452	4
32 ALU	8869	<1	>1000000	83.4	4341	8.2
64 ALU	17829	<1	>1000000	81.4	9487	47.2
128 ALU	35749	1.9	>1000000	79.1	18086	149.6
256 ALU	71598	4.0	>1000000	82.2	44870	697.9
8 MM	890	<1	79007	6	883	3
16 MM	3310	<1	>1000000	50	3295	16
32 MM	12566	2	>1000000	39	39265	86
12 MUL	605883	255	1324674	340	1494828	2500

Table 8.2. Effects of dynamic variable ordering.

8.4 Finite State Machines

As a second example of the use of BDDs we will now show how finite state machines can be represented and analysed with the help of BDDs. Consider the Mealy-type sequential machine of figure 8.5. Clearly it consists of a combinational part that constitutes the next-state and output functions and a memory part that holds the current state inbetween clock-ticks. The clock signal itself is not drawn, but assumed to control the memory elements. This particular example implements a 3-bit incrementer circuit, i.e., a modulo-8 counter. The combinational part is readily converted into BDDs by the method described above for combinational circuits, but how about the memory elements? BDDs are by definition acyclic (functional), but now it seems that we have to express the output of a memory element in terms of its input which itself is a function of the output. There are two tricks involved: 1) cut the circuit at the memory elements

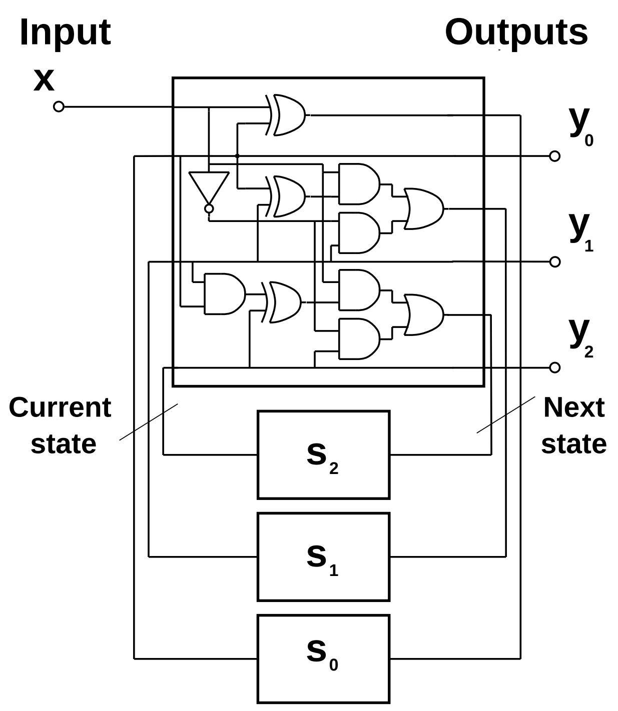


Figure 8.5. Mealy machine model (modulo-8 counter).

and consider the memory elements' outputs as new primary inputs (with current-state BDD variables assigned to them) and their inputs as new primary outputs (with next-state BDD variables assigned to them); and 2) represent sets of states as BDDs. For the latter we use the current-state variables (s_0, s_1, and s_2 in the example), and express a set of states by the BDD of its characteristic function. The characteristic function of a set A is the Boolean function $\chi_A : A \rightarrow B$ defined by $\chi_A = \lambda a. a \in A$.

Formally, a Mealy machine $M = (S, \mathbf{I}, O, N, S_0, Y)$ consists of the following components:

$S \subseteq B^n$ the set of states of interest (contents of the register);

$\mathbf{I} \subseteq B^m$ the set of m-bit input words;

$O \subseteq B^p$ the set of p-bit output words;

$N : B^n \times B^m \rightarrow B^n$ the (partial) next-state function;

$S_0 \subseteq S$ the non-empty set of initial states, and

$Y : B^n \times B^m \rightarrow B^p$ the (partial) output function.

In most cases, the sets of states, inputs, and outputs are not subsets but equal their universe. Also the next-state and output functions are usually total functions. For simplicity we will assume this indeed to be the case in the sequel.

Consider the task of calculating the set of states reachable from the initial states. In terms of sets we easily obtain the breadth-first search algorithm 8.2.

```
$2^S$ explore ($S_0$,H)
{
  Reach:=$\emptyset$;  New:=$S_0$;
  do {
    Reach:=Reach $\dot{\cup}$ New;
    Next:= H(New);
    New:=Next \ Reach;
  } while (New $\neq \emptyset$);
  return Reach;
}
```

Algorithm 8.2. Reachable states calculation.

In the above algorithm, H (Greek capital eta) is the so-called immediate neighbourhood function:

$$H : 2^S \to 2^S, H(A) = \{t \in S \mid \exists_{s \in A}\ \eta(s,t)\}$$

$$\text{with } \eta : S \times S \to B, \eta(s,t) = \exists_{x \in I}\ N(s,x) = t$$

Note that algorithm 8.2 is rather strict in the sense that it does not explicitly indicate the possible freedom of choice for the argument to H: there is no harm to include some states that have already been reached at that point. In other words, for the argument to H we may choose any superset of **New** as long as it is also a subset of **Reach**.

```
$2^S$ explore ($S_0$,H)
{
  k:=0;  Reach:=$\emptyset$;  New:=$S_0$;
  do {
    k++;
    Reach:=Reach $\dot{\cup}$ New;
    Choose Front|New $\subseteq$ Front $\subseteq$ Reach;
    Next:= H(Front);
    New:=Next \ Reach;
  } while (New $\neq \emptyset$);
  /* k = number of iterations. */
  return Reach;
}
```

Algorithm 8.3. More general state-space exploration.

This may seem not very useful in the current context, but depending on the representation of the sets in an implementation some computational advantage may be gained. This is particularly true when we use BDDs. Algorithm 8.3 shows the necessary modifications. A snapshot taken during its execution is depicted in figure 8.6.

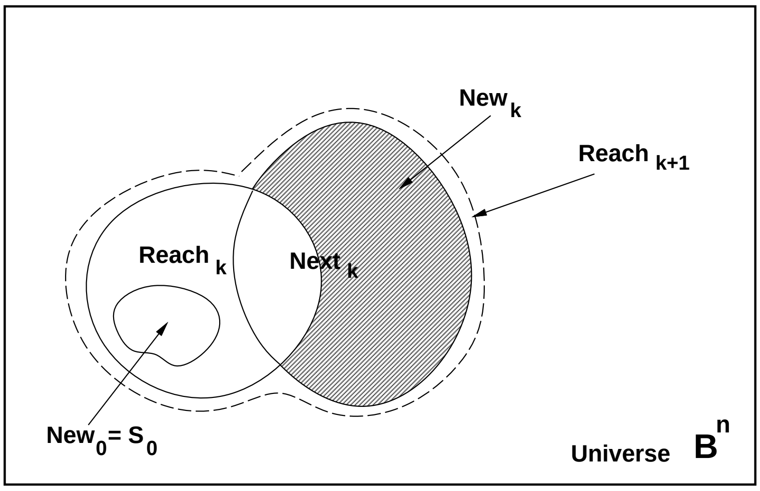


Figure 8.6. Various state-sets during execution of alg. 8.3.

A simple and elegant algorithm with less set-operations can be derived from algorithm 8.3 by choosing **`Front=Reach`** and performing a number of program transformations, namely changing the termination condition, eliminating **`New`** and **`Next`**, and modifying the loop to terminate in the middle of its body. This results in algorithm 8.4 (Mark the strong resemblance with algorithm 7.2; indeed finding the reachable states involves a least-fixpoint calculation).

```
2^S explore (S_0,H)
{
  k:=1;  Reach:=∅;
  do {
    New_Reach:=S_0 ∪ H(Reach);
    if (New_Reach = Reach) return Reach;
    k++;
    Reach:=New_Reach;
  } forever;
}
```

Algorithm 8.4. Alternate state-space exploration.

We take algorithm 8.4 as our basis for a BDD implementation. Table 8.3 lists the basic substitutions necessary to move from the set domain to the BDD domain.

Set	BDD
$\varnothing$	**BDD_0**
$\overline{S}$	**bdd_not(S)**
$S \cup T$	**bdd_or(S,T)**
$S \cap T$	**bdd_and(S,T)**
$S = T$	**S=T**
Universe	**BDD_1**

Table 8.3. Set notation versus BDD notation.

Applying this conversion to algorithm 8.4 we get algorithm 8.5 in pseudo-C code.

```
BDD explore (BDD $S_0$, BDD (*H)(BDD))
{
  Reach:=BDD_0;
  do {
    New_Reach:=bdd_or($S_0$,H(Reach));
    if (New_Reach=Reach) return Reach;
    Reach:=New_Reach;
  } forever;
}
```

Algorithm 8.5. BDD-based state-space exploration.

Our only concern now is to determine how to implement the function H in terms of BDDs. Its purpose is clear: H takes a set of states expressed as a BDD over the current-state variables and calculates all their immediate neighbouring states. The result, of course, again has to be a set of states expressed as a BDD over the current-state variables. Assume that we have derived the so-called next-state relation R for the Mealy machine:

$$R = \exists_{\underline{x}} \bigwedge_i (t_i \leftrightarrow N_i(\underline{s}, \underline{x})).$$

R can be represented as a BDD over the current-state variables s_i and next-state variables t_i. It is calculated by 'and'-ing together all results of taking the logical equivalence of next-state variable t_i with the ith component of the next-state function N (represented as a BDD in terms of the current-state variables s_i and the primary input variables x_i), and then existentially quantifying over all primary input variables. Even the last step is easily implemented as a BDD operation: $\exists_x f = f_x \vee f_{\bar{x}}$. Algorithm 8.6 summarizes the above process assuming that BDDs for the state and input variables and the components of the next-state function N are available.

```
BDD R;

void calc_R(void)
{
  conj:=BDD_1;
  for (i:=0; i<n; i++)
    conj:=bdd_and(conj, bdd_equiv(t_i,N_i));
  R:=bdd_exist(x̲,conj);
}
```

Algorithm 8.6. Calculation of R.

Using R, we can calculate $H(A)$ for any $A \subseteq S$ as stipulated in algorithm 8.7.

```
BDD H(BDD A)
{
  return bdd_exist(s̲, bdd_and(A,R))[t̲:= s̲];
}
```

Algorithm 8.7. Calculation of $H(A)$.

The notation `[t̲:= s̲]` is used to express the parallel substitution of all t_i variables with their corresponding s_i variables, in order to obtain a BDD result $H(A)$ that again is expressed over the current-state variables. With this provision, algorithm 8.5 will correctly calculate and return the set of states reachable from the initial states S_0 expressed as a BDD over the current-state variables.

8.4.1 Equivalence of Mealy machines

Consider two Mealy machines which we claim to be functionally equivalent, i.e., when both are started in an initial state and fed the same sequence of input words, their outputs will be identical. Of course, we presuppose that the machines have an identical number of input and output pins, and that their correspondence is known. How can this be verified apart from exhaustively simulating both machines for all possible input sequences? One solution is to construct a so-called product machine and calculate its reachable states. The product machine is sketched in figure 8.7. This is itself a Mealy machine with a single bit output function $Y : S \times \mathbf{I} \rightarrow B$ obtained by pairwise 'xnor'-ing the original outputs and 'and'-ing all result bits together. Obviously, the original machines are equivalent iff the product machine has output value true for all its reachable states.

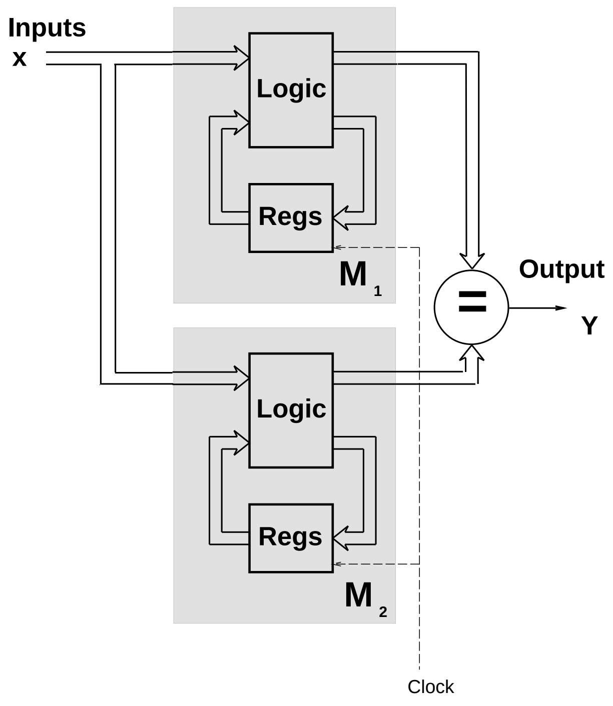


Figure 8.7. Product of two Mealy machines.

We simply modify the reachable state algorithm by including a test for Y = true, and thus get algorithm 8.8.

```
bool equivalent(BDD $S_0$, BDD (*H)(BDD))
{
  Reach:=$\emptyset$;
  do {
    New_Reach:=$S_0 \cup H$(Reach);
    if (!$\forall_{\underline{S} \in \text{New\_Reach}} \forall_{\underline{X}}$ Y($\underline{S}$,$\underline{X}$)) return 0;
    if (New_Reach = Reach) return 1;
    Reach:=New_Reach;
  } forever;
}
```

Algorithm 8.8. Correctness of product machine.

8.5 BDD-based CTL Model Checking

We will now briefly return to the problem of CTL model checking. As we have seen in chapter 7, all we need are a least fixpoint and a greatest fixpoint algorithm, but this time we like to state them in terms of BDDs. The algorithms we seek are very similar to the reachable states algorithms developed in the previous section. In fact, the reachable states are the least fixpoint of the function $F(Z) = S_0 \vee H(Z)$. The basis for the CTL algorithms, however, is not the neighbourhood function H but its converse, which we will denote by H^{-1}. When H is available as a BDD, we obtain H^{-1} by merely swapping the corresponding pairs of current-state s_i and next-state t_i variables. This can be efficiently implemented by a single traversal over the BDD as long as we take care to order the variables pairwise. The modified routines for the basic CTL operators are shown in algorithm 8.9.

```
BDD EX(BDD p)
{
  return H^{-1}(p);
}

BDD EG(BDD p)
{
  k:=1;  Z_k:=p;
  do {
    Z_{k+1}:=bdd_and(p,EX(Z_k));
    if (Z_{k+1}=Z_k) return Z_k;
    k++;
  } forever;
}

BDD EU(BDD p,BDD q)
{
  k:=1;  Z_k:=q;
  do {
    Z_{k+1}:=bdd_or(q,bdd_and(p,EX(Z_k)));
    if (Z_{k+1}=Z_k) return Z_k;
    k++;
  } forever;
}
```

Algorithm 8.9. Calculation of BDD state-sets for the basic CTL operators.

8.6 References

[**Brac90**] Brace, Karl S., Rudell, Richard L., and Bryant, Randal E., "Efficient Implementation of a BDD Package," *Proc. 27-th ACM/IEEE Design Automation Conference*, pp. 40-45, Orlando, Florida, June 24-28, 1990.

[**Brya86**] Bryant, Randal E., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677-691, August 1986.

[Rude93] Rudell, Richard, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *Proc. IEEE Int. Conf. on Computer-Aided Design*, 1993.

[Toua90] Touati, Herve J., Savoj, Hamid, Lin, Bill, Brayton, Robert K., and Sangiovanni-Vincentelli, Alberto, "Implicit State Enumeration of Finite State Machines using BDD's," *Proc. IEEE Int. Conf. on Computer-Aided Design*, pp. 130-133, Santa Clara, CA, 1990.