Implicit State Enumeration of Finite State Machines using BDD's*

Hervé J. Touati Hamid Savoj Bill Lin
Robert K. Brayton Alberto Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences Department
University of California, Berkeley, CA 94720, USA

Abstract

Coudert *et al.* have proposed in [4] an efficient method to compute the set of reachable states of a sequential finite state machine using BDD's. This technique can handle larger finite state machines than previously possible and has a wide range of applications in sequential synthesis, testing and verification. At the heart of this method is an algorithm that computes the range of a set of Boolean functions under a restricted domain. Coudert *et al.* originally proposed a simpler and more general algorithm for range computation that was based on relations, but dismissed it as impractical for all but the simplest examples. We propose a new approach based on relations that outperforms Coudert's algorithm with the additional advantage of simplicity and wider applicability.

1 Introduction

Efficient algorithms to perform state enumeration for finite state machines can be applied to a wide range of problems: implementation verification (checking that an implementation conforms to its specification [4, 3]), design verification (e.g. checking that an implementation or a specification satisfies fairness and liveness properties [3]), sequential testing and sequential synthesis.

Traditional state enumeration methods cannot handle machines with more than a few million states. To break this limitation, Ghosh *et al.* [6] proposed an implicit enumeration technique based on cubes. Unfortunately, in their approach, sets of states can be manipulated as a unit only if they can be represented as a cube in some binary encoding. Coudert *et al.* [4] were first to recognize the advantage of representing set of states with reduced ordered binary decision diagrams (*BDD's*) [2]. Their technique was initially applied to checking finite state machine equivalence and was later extended by Burch *et al.* to the computation of temporal logic formulas [3]. Sequential testing of non-SCAN designs and extraction of sequential don't cares are other immediate applications of this technique.

In all these applications, the solution can be obtained by iterative use of one of two operations: the computation, for a given Boolean function, of the *image* of a subset of its domain, and the computation of the *inverse image* of a subset of its co-domain. A conceptually simple and elegant method to perform these operations, originally introduced in [4], consists of forming a BDD representation of the state transition relation of a finite state machine (i.e. the single output function $f(x, i, y)$ which is equal to 1 if y is the state reached by the machine in state x upon receiving input i). The image and reverse image of a set of states by the transition function can be obtained from the transition relation in one BDD operation.

Unfortunately the BDD for the transition relation can often grow too large to be practical. Coudert *et al.* in [4, 5] proposed a recursive method for image computation that only requires the ability to compute the BDD of the state-transition functions $f(x, i) = (f_1, \ldots, f_n)$. We propose a new method based on transition relations that only requires the ability to compute the BDD for f_i and outperforms Coudert's algorithm for most examples.

The main contributions of this paper are: in section 2 a simple notational framework to express the basic operations used in BDD-based state enumeration algorithms in a unified way; in section 3 a set of techniques that can speed up range computation dramatically, including a new variable ordering heuristic and a new method based on transition relations. We present and discuss our experimental results in section 4.

2 Terminology and Previous Work

2.1 Image and Inverse Image Computations

In what follows B designates the set $\{0, 1\}$.

Definition 1 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A a subset of B^n. The* image *of A by f is the set $f(A) = \{y \in B^m \mid y = f(x),\ x \in A\}$. If $A = B^n$, the image of A by f is also called the* range *of f.*

Definition 2 *Let $f : B^n \rightarrow B^m$ be a Boolean function and A a subset of B^m. The* inverse image *of A by f is the set $f^{-1}(A) = \{x \in B^n \mid f(x) = y,\ y \in A\}$.*

Example Let $f(x, i) : B^n \times B^k \rightarrow B^n$ be the next state function of a finite state machine, where n is the number of state variables and k the number of input variables. Let c_∞ be the set of states reachable from a set of initial states c_0. c_∞ can be obtained by repeated computation of an image as follows

$$\begin{aligned} c_{i+1} &= c_i \cup f(c_i \times B^k) \\ c_\infty &= c_i \quad \text{if} \quad c_{i+1} = c_i \end{aligned}$$

The sequence is guaranteed to converge after a finite number of iterations.

2.2 Sets, BDD's, and Characteristic Functions

Definition 3 *Let E be a set and $A \subseteq E$. The* characteristic function *of A is the function $\chi_A : E \rightarrow \{0, 1\}$ defined by $\chi_A(x) = 1$ if $x \in A$, $\chi_A(x) = 0$ otherwise.*

Characteristic functions are nothing but a functional representation of a subset of a set. In the rest of this paper we will not make any distinction between the BDD representing a set of states, the characteristic function of the set, and the set itself.

*This project is supported in part by Defense Advanced Research Projects Agency under contract number N00039-87-C-0182 and NSF/DARPA contract MIP-8719546.

2.3 The Smoothing Operator

Definition 4 *Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_{i_1}, \ldots, x_{i_k})$ a set of input variables of f. The smoothing of f by x is defined as in [8], with f_a denoting the cofactor of f by literal a:*

$$\begin{aligned} S_x f &= S_{x_{i_1}} \ldots S_{x_{i_k}} f \\ S_{x_{i_j}} f &= f_{x_{i_j}} + f_{\overline{x_{i_j}}} \end{aligned}$$

If f is interpreted as a logical predicate, the smoothing operator computes the existential quantification of f relative to the x variables. If f is interpreted as the characteristic function of set, the smoothing operating computes the projection of f to the subspace of B^n orthogonal to the domain of the x variables. We will make use of the following simple property of the smoothing operator (the Boolean *and* is denoted by a dot):

Lemma 2.1 *Let $f : B^n \times B^m \rightarrow B$ and $g : B^m \rightarrow B$ be two Boolean functions. Then:*

$$S_x(f(x,y) \cdot g(y)) = S_x(f(x,y)) \cdot g(y) \quad (1)$$

2.4 The Transition Relation Method

Definition 5 *Let $f : B^n \rightarrow B^m$ be a Boolean function. The transition relation associated with f, $F : B^n \times B^m \rightarrow B$, is defined as $F(x,y) = \{(x,y) \in B^n \times B^m \mid y = f(x)\}$. Equivalently, in terms of Boolean operations:*

$$F(x,y) = \prod_{1 \leq i \leq m} (y_i \equiv f_i(x)) \quad (2)$$

We can use F to obtain the image by f of a subset A of B^n, by computing the projection on B^m of the set $F \cap (A \times B^m)$. In terms of BDD operations, this is achieved by a Boolean *and* and a smooth. The smoothing and the Boolean *and* can be done in one pass on the BDD's to further reduce the need for intermediate storage [3]:

$$f(A)(y) = S_x(F(x,y) \cdot A(x)) \quad (3)$$

The inverse image by f of a subset A of B^m can be computed as easily:

$$f^{-1}(A)(x) = S_y(F(x,y) \cdot A(y)) \quad (4)$$

2.5 The Generalized Cofactor

The *generalized cofactor* is an important new operator that can be used to reduce an image computation to a range computation. This operator was initially proposed by Coudert *et al.* in [4] and called the *constraint* operator. Given a Boolean function: $f = (f_1, \ldots, f_m) : B^n \rightarrow B^m$ and a subset of B^n represented by its characteristic function c, the generalized cofactor $f_c = ((f_1)_c, \ldots, (f_m)_c)$ is a function from B^n to B^m whose range is equal to the image of c by f. In addition, in most cases, the BDD representation of f_c is smaller than the BDD representation of f. For a single output function $f : B^n \rightarrow B$, the pair (f,c) can be interpreted as an incompletely specified function whose onset is $f \cdot c$ and don't care set $\overline{c}$. Under this interpretation, the generalized cofactor f_c can be seen as a heuristic to select a representative of the incompletely specified function (f,c) that has a small BDD representation. The generalized cofactor f_c depends in general on the variable ordering used in the BDD representation. If c is a cube the generalized cofactor f_c is equal to the usual cofactor of a Boolean function, and is, in that case, independent of the variable ordering.

```
function cofactor(f,c) {
  assert (c ≠ 0);
  if (c = 1 or is_constant(f))      return f;
  else if (c_{x̄1} = 0)              return cofactor(f_{x1}, c_{x1});
  else if (c_{x1} = 0)              return cofactor(f_{x̄1}, c_{x̄1});
  else return x1 · cofactor(f_{x1}, c_{x1})
              + x̄1 · cofactor(f_{x̄1}, c_{x̄1});
}
```

Figure 1: Generalized Cofactor Algorithm

Definition 6 *Let $c : B^n \rightarrow B$ be a non-null Boolean function and $x_1 \prec x_2 \prec \ldots \prec x_n$ an ordering of its input variables. We define the mapping $\pi_c : B^n \rightarrow B^n$ as follows:*

$$\begin{aligned} &\text{if } c(x) = 1 \quad \pi_c(x) = x \\ &\text{if } c(x) = 0 \quad \pi_c(x) = \arg\min_{y, c(y)=1} d(x,y) \\ &\text{where } d(x,y) = \sum_{1 \leq i \leq n} |x_i - y_i| 2^{n-i} \end{aligned}$$

Lemma 2.2 *π_c is the projection that maps a minterm x to the minterm y in the onset of c which has the closest distance to x according to the distance d. The particular form of the distance guarantees the uniqueness of y in this definition, for any given variable ordering.*

Proof Let y and y' be two minterms in the onset of c such that $d(x,y) = d(x,y')$. Each of the expressions $d(x,y)$ and $d(x,y')$ can be interpreted as the binary representation of some integer. Since the binary representation is unique, $|x_i - y_i| = |x_i - y'_i|$ for $1 \leq i \leq n$ and thus $y = y'$. ∎

Definition 7 *Let $f : B^n \rightarrow B$ and $c : B^n \rightarrow B$, with $c \neq 0$. The generalized cofactor of f with respect to c, denoted by f_c, is the function $f_c = f \circ \pi_c$ (i.e. $f_c(x) = f(\pi_c(x))$). If $f : B^n \rightarrow B^m$, then $f_c : B^n \rightarrow B^m$ is the function whose components are the cofactors by c of the components of f.*

The generalized cofactor can be computed very efficiently in a single bottom-up traversal of the BDD representations of f and c by the algorithm given in Figure 1.

Lemma 2.3 *If c is a cube (i.e. $c = c_1 c_2 \ldots c_n$ where $c_i = \{0, 1, *\}$), π_c is independent of the variable ordering. More precisely, if y satisfies*

$$\begin{aligned} y_i = 0 \quad &\text{if} \quad c_i = 0 \\ y_i = 1 \quad &\text{if} \quad c_i = 1 \\ y_i = x_i \quad &\text{if} \quad c_i = * \end{aligned}$$

then $y = \pi_c(x)$ and $f_c = f \circ \pi_c$ is the usual cofactor of a Boolean function by a cube.

Proof Any minterm y' in B^n such that $c(y') = 1$ is orthogonal to x in at least the same variables as y. Thus y minimizes $d(x,y)$ over c. ∎
In addition, the generalized cofactor preserves the following important properties of cofactors:

Proposition 2.1 *Let $g : B^m \rightarrow B$ and $f : B^n \rightarrow B^m$. Then $(g \circ f)_c = g \circ f_c$. In particular the cofactor of a sum of functions is the sum of the cofactors, and the cofactor of an inverse is the inverse of the cofactor.*

Proof $(g \circ f)_c = (g \circ f) \circ \pi_c$ and $g \circ f_c = g \circ (f \circ \pi_c)$. ■

Proposition 2.2 *Let $f : B^n \times B^m \rightarrow B$ and $c : B^n \rightarrow B$ be two Boolean functions, with $c \neq 0$. Then:*

$$S_x(f(x,y) \cdot c(x)) \quad = \quad S_x(f_c(x,y)) \tag{5}$$

Proof If $c(x) = 1$, then $f_c(x,y) = f(x,y)$. Thus $f(x,y) \cdot c(x) \subseteq f_c(x,y)$ and $S_x(f(x,y) \cdot c(x)) \subseteq S_x(f_c(x,y))$. Conversely, if y is such that $S_x(f_c(x,y)) = 1$, there exists an x such that $f_c(x,y) = 1$. Thus $f(\pi_c(x),y) \cdot c(\pi_c(x)) = f_c(x,y) = 1$, which implies that $S_x(f(x,y) \cdot c(x)) = 1$. ■

Proposition 2.3 *Let f be a Boolean function, and c a non-null Boolean function. Then c is contained in f if and only if f_c is a tautology.*

Proof Suppose that c is contained in f. Let x be an arbitrary minterm. $y = \pi_c(x)$ is such that $c(y) = 1$. Thus $f_c(x) = f(y) = 1$ which proves that f_c is a tautology. Conversely, suppose that f_c is a tautology. Let x be such that $c(x) = 1$. Then $\pi_c(x) = x$ and $f(x) = f(\pi_c(x)) = f_c(x) = 1$, which proves that c is contained in f. ■

Corollary 2.4 *Let f be a Boolean function, and c a non-null Boolean function. Then $c(x) = 1$ implies that $f(x) = f_c(x)$.*

Lemma 2.5 *If c is the characteristic function of a set A then $f_c(B^n) = f(A)$; that is the image of A by f is equal to the range of the cofactor f_c.*

Proof $\pi_c(B^n)$ is equal to the onset of c, which is A. Thus $f_c(B^n) = f \circ \pi_c(B^n) = f(A)$. ■

2.6 The Recursive Image Computation Method

Coudert *et al.* [4, 5] introduced an alternate procedure to compute the image of a set by a Boolean function that does not require building the BDD for the transition relation. This procedure relies on lemma 2.5 to reduce the image computation to a range computation, and proceeds recursively by cofactoring by a variable of the input space or the output space. We use the abbreviation $rg(f)$ to denote the range of a multiple output function $f = [f_1, \ldots, f_m]$:

$$\begin{aligned} rg(f)(y) &= y_1 \cdot rg([f_2, \ldots, f_m]_{f_1}) + \overline{y_1} \cdot rg([f_2, \ldots, f_m]_{\overline{f_1}}) \\ rg(f)(y) &= rg([f_1, \ldots, f_m]_{x_1}) + rg([f_1, \ldots, f_m]_{\overline{x_1}}) \end{aligned}$$

The procedure can be sped up dramatically by caching intermediate range computations, and detecting the case where, at any step in the recursion, the functions $[f_1, \ldots, f_m]$ can be grouped into two or more sets with disjoint support. The range computation can proceed independently on each group with disjoint support. This reduces the worst case complexity from 2^m to $2^{s_1} + \ldots + 2^{s_k}$, where $(s_1, \ldots, s_k)$ are the sizes of the groups $(s_1 + \ldots + s_k = m)$.

3 Heuristics

3.1 Variable Ordering Heuristics

Variable ordering heuristics are known to have a dramatic effect on BDD sizes. Good variable ordering heuristics have been developed for BDD representations of combinational circuits [7]. For sequential circuits, the variable ordering influences not only the size of the BDD representation of the transition function but also the size of the BDD representation of the set of reachable states. In addition, both for the transition relation method and the recursive image computation method, we usually need to use an ordering that interleaves input and output variables.

Our variable ordering heuristics are extensions of the heuristics described by Malik *et al.* in [7]. We first determine a good ordering of the next state variables $(y_1, \ldots, y_n)$, or equivalently the corresponding next state functions $(f_1, \ldots, f_n)$. We then use Malik's heuristics to order the supports of the functions f_i, $supp(f_i)$, individually. Finally we interleave the input and output variables as follows: $supp(f_1), y_1, \ldots, supp(f_n) - \cup_{1 \leq i \leq n-1} supp(f_i), y_n$.

To order the output functions, we want to use some permutation σ of the output functions that minimizes the following cost function, where $|A|$ denotes the number of elements of set A:

$$cost(\sigma) \quad = \quad \sum_{1 \leq i \leq n} | \bigcup_{1 \leq j \leq i} supp(f_{\sigma_j})|$$

Unfortunately, finding the optimal permutation is difficult in general. The best algorithm we could find is based on dynamic programming and has complexity $O(2^n)$. To find an approximate solution to this problem, we use a simple greedy algorithm with bounded look-ahead k. This algorithm computes all possible choices for the first k functions, and for each choice, completes the ordering by selecting for f_{σ_i}, $i \geq k+1$, the function that minimizes $|\cup_{1 \leq j \leq i} supp(f_{\sigma_j})|$. Experimentally, a look-ahead of 2 may yield significantly better orderings than a look-ahead of 0, and look-aheads or 3 or more are not practical for large examples. We use this variable heuristics in all examples reported in this paper except the MINMAX examples, for which a manual ordering yields significantly better results.

3.2 Partial Product Heuristics

Computing the transition relation may require too much memory to be feasible in some examples. However, to perform an image computation as in equation 3, we do not need to compute the transition relation explicitly. Using propositions 2.1 and 2.2, we can rewrite equation 3 as follows:

$$S_x(F(x,y) \cdot A(x)) \quad = \quad S_x(\prod_{1 \leq i \leq m} (y_i \equiv (f_i)_A(x))$$

One efficient way to compute the product is to decompose the Boolean *and* of the m functions $(g_i(x,y) = y_i \equiv (f_i)_A(x))$ into a balanced binary tree of Boolean *and*. Morever, after computing every binary *and* p of two partial products p_1 and p_2, we can smooth the x variables that only appear in p. As for equation 3, the smoothing and the *and* computations can be done in one pass on the BDD's to reduce storage requirements. This algorithm strictly dominates all the other range computation algorithms presented in this paper, in the sense that it can handle all the examples the other techniques can handle, and a few more.

3.3 Heuristics for Iterative Image Computation

In most applications we need to iterate image computations until we reach a fixed point. We use two techniques to speed up this iterative computation. The first technique was suggested to us by R. Rudell [9], the second was originally introduced by Coudert *et al.* in [4].

Reordering the Variables of the Image In the transition relation method, the image is obtained in terms of the next state variables. To use the image as initial set for the next iteration, we need to express it in terms of the present state variables. An efficient way to perform this computation is to order the present state variables and the next state variables in pairs, and perform the substitution in one pass over the BDD representing the image.

Next State Heuristics Let C_i be the set of states reachable in i steps or less from the initial state. To compute C_{i+1}, we do not need to compute the image of C_i. We simply need to compute the image of any set c_i such that $C_i - C_{i-1} \subseteq c_i \subseteq C_i$. C_{i+1} is then obtained by computing the union of C_i and the image of c_i. We only care about the value of c_i outside C_{i-1}. Thus we can choose for c_i any representative of the incompletely specified function $(C_i, \overline{C_{i-1}})$. This is an ideal case of application of the generalized cofactor. By choosing $(c_i)_{\overline{C_{i-1}}}$ as the set of states to consider for the next image computation, we can reduce in practice often quite dramatically the time required to enumerate all reachable states.

4 Results and Discussion

We present in this section some results comparing the recursive method proposed by Coudert *et al.* [4, 5], the transition relation method proposed by Burch *et al.* [3] and the new method described in section 3.2. All methods use the variable ordering heuristics presented in section 3.1.

For each of these three methods, we measured the time required to perform the verification of two identical copies of a finite state machine using the breadth-first traversal technique described in [4]. The two copies keep their state variables independent, but share the external input wires. The verification starts from an initial state, and implicitly enumerates all reachable states by doing a breadth-first traversal of the state transition diagram of the product machine. At each step in the verification, the outputs of the two machines are checked for equality. We report the time to perform the entire computation, including parsing the input files and computing the product machine. Run times were measured on an DEC-5400. The program was implemented as an extension of misII [1] and the reported time was obtained using the misII *time* command.

We use several ISCAS sequential benchmarks (`sand`, `scf`, `s344`, `s444`, `s526`, `s713`, `s953`, `s1238`), an accumulator made out of a 32 bit carry-bypass adder (`cbp.32.4`), a circuit computing the minimum and the maximum of a sequence of 32-bit integers (`minmax32`), a circuit computing the encryption key used in a VLSI implemention of the data encryption standard `key`, and `sbc`, the snooping bus controller for the SPUR multiprocessor. Except for the `minmax32` example, the variable ordering was performed automatically. The partial product method was the only one able to complete the verification of `key` and `sbc`; it is a clear winner for the large examples, and performs adequately well for the smaller ones. As can be observed from these examples, the computation times are very weakly related to the number of states visited. We also compared our results with the method introduced by Ghosh et al. [6]. All three methods in the table outperform the one in [6] when applied to most large finite state machines.

circuit	# latches	# size	# states	depth	trans	recur	prod
sand	6	1310	32	4	60.8	16.5	16.9
scf	8	2208	115	16	64.2	18.8	18.7
s344	15	269	2625	7	38.4	41.3	34.6
s444	21	352	8865	151	126.7	120.5	186.7
s526	21	445	8868	151	58.5	133.8	126.1
s713	19	591	1544	7	279.0	90.3	63.7
s953	29	766	504	11	spaceout	43.1	70.2
s1238	18	1042	2616	3	98.7	108.3	43.6
cbp.32.4	32	480	4.29e+09	2	16.0	31.8	14.1
minmax32	96	1874	1.32e+28	4	901.8	timeout	444.4
sbc	28	1670	154593	10	spaceout	timeout	2903.7
key	228	3865	1.35e+68	17	1778.6	timeout	5706.2

Table 1: Performance of Sequential Verification

 # latches: number of state variables
 # size: number of literals in factored form
 # states: number of reachable states
depth: number of iterations in breadth-first traversal
trans: using the transition relation method (seconds)
recur: using the recursive range computation method (seconds)
prod: using the partial product method (seconds)
timeout set to 6000 seconds of CPU time

References

[1] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, pages 1062–1081, November 1987.

[2] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *27th ACM/IEEE Design Automation Conference*, Orlando, June 1990.

[4] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[5] O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines Without Building their State Diagrams. In *Worshop on Computer-Aided Verification*, Rutgers, June 1990.

[6] A. Ghosh, S. Devadas, and A. R. Newton. Test Generation for Highly Sequential Circuits. In *IEEE International Conference on Computer-Aided Design*, pages 362–365, November 1989.

[7] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environments. In *IEEE International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[8] P. McGeer. *On the Interaction of Functional and Timing Behavior of Combinational Logic Circuits*. PhD thesis, U.C. Berkeley, November 1989.

[9] R. Rudell. private communication, 1990.