On Computing the Transitive Closure of a State Transition Relation

Yusuke Matsunaga* Patrick C. McGeer† Robert K. Brayton†

Abstract

We describe a new, recursive-descent procedure for the computation of the transitive closure of a transition relation. This procedure is the classic binary matrix procedure of [1], adapted to a BDD data structure. We demonstrate its efficacy when compared to standard iterative methods.

1 Introduction

The verification of interacting finite state machines has become a principal focus of investigation in the CAD community in recent years. The resurgence of activity can be traced to the convergence of two principal ideas:

1. The realization that verification problems could be phrased as reachable states computations on Finite State Machines; and
2. The development of a relatively efficient implicit, breadth-first reachable states algorithm.

In this brief paper, we focus on the latter algorithm. The seminal idea occured in the late 1980's, and appeared in several places[4, 5, 6, 7] within the span of a year: given an automaton $A = (S, r, \Gamma : S \mapsto \mathcal{P}^S)$, where S is the set of states, r is the reset or start state, and Γ is the next-state function, compute a Boolean function:

$$R : S \mapsto \{0, 1\}$$

where $R(s) = 1$ if and only if there is a finite sequence $\{s_0 = r, s_1, s_2, ..., s_n = s\}$ where $s_j \in \Gamma(s_{j-1})$. R is called the *characteristic function* of the set of reachable states.

The key idea was that the characteristic function of a set of states often had a compact representation, even when the set of states represented was very large. The characteristic function of the reachable states is derived from the characteristic function of the transition relation Γ:

$$T : S \times S \mapsto \{0, 1\}$$

$T(s, s') = 1$ if and only if $s' \in \Gamma(s)$.

It is easy to show that:

$$R(s) = 1 \iff \exists t \ni R(t) = 1 \textit{ and } T(t, s) = 1$$

From this, the derivation of the basic identity is easy. R is the least function satisfying:

$$\begin{aligned} R(r) &= 1 \\ R(s) &= R(s) \vee \exists t[R(t)T(t, s)] \end{aligned}$$

The procedure was then a simple iteration of the equation:

$$R^n(s) = R^{n-1}(s) \vee \exists t[R^{n-1}(t)T(t, s))] \qquad (1)$$

The iteration concluded when $R^n = R^{n-1}$, and was begun with $R^0 = r$.

Existential quantification was implemented using the Boolean smoothing operator $\mathcal{S}$. $\mathcal{S}_x f = f_x \vee f_{\overline{x}}$. It is fairly easy to demonstrate that, for a function f defined over variables $x_1, ..., x_n, y_1, ..., y_m$, $\exists x f(x, y)$ is equal to the quantity $\mathcal{S}_{x_1}\mathcal{S}_{x_2}....\mathcal{S}_{x_n} f(x, y)$.

The reachable states of a finite state machine may also be found by computing the transitive closure $T^+(s, t)$ of the transition function $T(s, t)$. The reachable states can therefore be found as $R(t) = T^+(r, t)$ where r is the reset state of the finite state machine.

In the remainder of this short paper, we discuss new methods of finding $T^+(s, t)$. First, we demonstrate a method of finding $T^+(s, t)$ from the (related) function $T^*(s, t) = [T(s, t) \vee (s = t)]^+$. Second, we demonstrate an adaptation of the classic recursive-descent procedure[1] for finding $T^*(s, t)$. Finally, we give some experimental results.

2 The Reflexive-Transitive Closure

Closely related to the transitive closure of a transition function T is the *reflexive-transitive closure of* T, $T^*(s, t)$.

$$T^*(s, t) \equiv [T(s, t) \vee (s = t)]^+ \qquad (2)$$

$T^*(s, t)$ is the transitive closure of the reflexive closure of T; the reflexive closure of T is the smallest reflexive relation containing T.

The reflexive-transitive closure is of interest since it is often easier to compute the transitive closure of a reflexive

*Fujitsu Laboratories, 1015 Kamikodanaka, Nakahara-Ku, Kawasaki 211, Japan

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

30th ACM/IEEE Design Automation Conference®

relation than it is to compute a general transitive closure. The purpose of this section is to validate the calculation of the reflexive-transitive closure, by demonstrating those cases where the reflexive-transitive closure is desired and how the transitive closure T^+ may be recovered from T^* where it is not desired.

Immediately from (2), $T^*(s,t) \supseteq T^+(s,t)$, since $T^*(s,t) = [T(s,t) \vee (s = t)]^+$, and $A \supseteq B$ implies $A^+ \supseteq B^+$. Of course, what we want is $T^+(s,t)$. We have immediately:

Lemma 2.1 *Let* $T^*(s,t) = 1$, *for* $s \neq t$. *Then* $T^+(s,t) = 1$.

Note that this lemma indicates that T^* is sufficient for the reachable states computation, since $T^+(r,t) = T^*(r,t)$ for $r \neq t$ by this lemma. However, we may wish the exact computation of T^+. We can find T^+ by a post-processing step.

Lemma 2.2

$$\overline{T^+(s,s)}T^*(s,s) = \overline{T(s,s) \vee \exists u[T^*(s,u)T^*(u,s)(u \neq s)]} \tag{3}$$

Proof: $T^+(s,s)$ iff $T(s,s)$ or there is some $u \neq s$ such that there is a path from s to u and a path from u to s, i.e., iff $T(s,s) \vee \exists u[T^*(s,u)T^*(u,t)(u \neq s)(u \neq t)]$, and hence the result. ∎

It is a fairly easy matter to obtain $T^+(s,t)$ by the following derivation:

$$\begin{aligned} T^+(s,t) &= T^*(s,t) \quad \{(s \neq t) \vee \overline{(\overline{T^+(s,s)}T^*(s,s))}\} \\ &= T^*(s,t) \quad \{(s \neq t) \vee T(s,s) \vee \\ &\qquad \exists u[T^*(s,u)T^*(u,s)(u \neq s)]\} \end{aligned} \tag{4}$$

3 Recursive Descent Procedure

Boolean problems are often best attacked by a recursive descent procedure. Recursive descent procedures are generally based on Shannon's famous theorem. Given a function f, variable x, we can write:

$$f = x f_x \vee \bar{x} f_{\bar{x}}$$

Applied recursively, the Shannon decomposition yields a tree of subfunctions. Many computations over a Boolean function can be expressed as compositions of computations over these subfunctions; these computations are fairly naturally expressed as computations over the Shannon Tree.

Computations over the Shannon Tree are, *prima facie*, exponential in nature. Shannon Trees for functions which are used in logic circuits turn out to be highly isotropic; this property can be exploited in a variety of ways. The first is

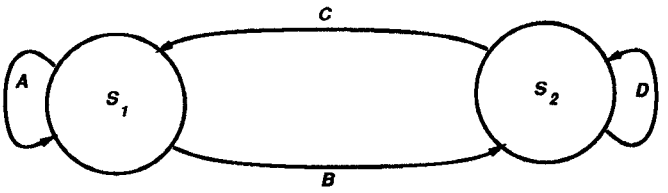

Figure 1: Bipartitioned State Space

to fold isomorphic subtrees together into a single structure; the resulting structure is called a Binary Decision Diagram, and, for a large class of interesting functions, the BDD is of size polynomial in the number of input variables. The second method, which exploits monotonicity, is not germane to this paper.

Our goal is to derive a recursive descent procedure for the computation of the transitive closure of a transition relation $T(x,y)$. We begin by noting the following. Any boolean function F can be regarded as a binary vector V of size 2^n, indexed by the vertices of the space of interest; $V_m = 1$ iff $m \in F$ for each vertex m. The rank and dimension of such a vector is, of course, arbitrary. However, it is convenient to regard a transition relation $T(x,y)$ as an $|S| \times |S|$ binary matrix, where the rows are indexed by the current state and the columns are indexed by the next state. The computation of the transitive closure of the transition relation is thus equivalent to the computation of the transitive closure of the associated binary matrix.

The classic procedure for computing the closure of an $n \times n$ binary matrix X is through a recursive-descent procedure. The matrix X is broken down into four quadrants:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

X is then interpreted as a representation of the edges of a directed graph, whose nodes are the set $\{1,...,n\}$ and whose edges are given by the matrix: (i,j) is an edge of the graph iff $X_{ij} = 1$. The division of the matrix X into quadrants corresponds to a bipartitioning of the nodes of the implied graph, into the sets $S_1 = \{1,...,m\}$, $S_2 = \{m+1,...,n\}$ (needless to say, any bipartitioning would do). A is then the set of edges from S_1 into S_1, B the set of edges from S_1 into S_2, C the set of edges from S_2 into S_1, and D the set of edges from S_2 into S_2. A graphic version of this decomposition is given in figure 1.

We wish to compute X^*, which we write as:

$$X^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

using the same implied bipartitioning. We begin by computing E, the set of *paths* from a state in S_1 into a state in

S_1. It is fairly easy to demonstrate that a path from S_1 into S_1 is either:

1. an element of A; or
2. a sequence consisting of an element of B followed by arbitrarily many elements of D, followed by an element of C; or
3. A sequence of paths of class (1) and class (2)

It is easy to express this in an equation[1]:

$$E = (A \vee B \circ D^* \circ C)^* \tag{5}$$

We turn to the computation of H. Of course, we could simply apply similar reasoning to that applied in computing E, and so arrive at:

$$H = (D \vee C \circ A^* \circ B)^*;$$

however, if it is assumed that closure operations are fairly expensive, and matrix multiplication fairly cheap, then we wish to minimize the number of closures taken. In fact, in computing E we've taken all the closures we need to take for this problem. We can do this by noting that a path from S_2 into S_2 is either:

1. A sequence of edges from D; or
2. A sequence of edges from D, followed by an element of C, followed by a path from S_1 into S_1, followed by an element of B, followed by a sequence of edges from D

This is easily translated into an equation:

$$H = D^* \vee D^* \circ C \circ E \circ B \circ D^* \tag{6}$$

Following similar reasoning, we may write:

$$F = E \circ B \circ D^* \tag{7}$$

$$G = D^* \circ C \circ E \tag{8}$$

the justification is a simplification of that used in the computation of H and is left as an exercise to the reader. Readers wishing a more detailed treatment are referred to Aho, Hopcroft, and Ullman's excellent text.

The recursive procedure suggested by (5)-(8)has been shown, by Aho, Hopcroft, and Ullman[1], to be of the same complexity as matrix multiplication.

Translation of this efficient procedure onto our problem requires only that we bipartition the states of our machine. There are many bipartitions, of course, but one that is particularly attractive exploits the binary encoding of the states. The states are encoded using the binary variables $\{x_1, ..., x_n\}$; S_1 is the set of states encoded with $x_1 = 0$, S_2 the set of states encoded with $x_1 = 1$. We can write the transition relation as the matrix:

$$T = \begin{pmatrix} T_{\overline{x_1}\overline{y_1}} & T_{\overline{x_1}y_1} \\ T_{x_1\overline{y_1}} & T_{x_1y_1} \end{pmatrix}^2$$

We can then easily adapt the classic procedure, using a little notation to simplify the computation:

Notation: In the sequel, the notation $T \circ R$ for Boolean functions T and R will denote the computation $\exists z[T(x,z)R(z,y)]$, or the computation $\exists z[T(x,z)R(z)]$, depending upon the dimensionality of R.

```
compute_closure(T) {
    if T ∈ {0,1} return T;
    T = ( T_{x̄1ȳ1}  T_{x̄1y1} )
        ( T_{x1ȳ1}  T_{x1y1} );
    U1 = compute_closure(T_{x1y1});
    U2 = U1 ∘ T_{x1ȳ1};
    E = compute_closure(T_{x̄1ȳ1} ∨ T_{x̄1y1} ∘ U2);
    U3 = T_{x̄1y1} ∘ U1;
    F = E ∘ U3;
    G = U2 ∘ E;
    H = U1 ∨ U2 ∘ F;
    return x̄1ȳ1E ∨ x̄1y1F ∨ x1ȳ1G ∨ x1y1H;
}
```

4 Memo Functions and Hashing

The classic procedure, given above, suffers from the fact that at each iteration, two recursive calls to `compute_closure` are required. This expense can be ameliorated by noting that one is in general not dealing with an arbitrary Boolean function represented as a binary matrix, but, rather, with a Boolean function with a compact representation.

We represent Boolean Functions with the Binary Decision Diagram[2, 3, 8] data structure. The fact that the function is represented as a BDD, and, further, has a compact representation as a BDD has some important consequences for us. First, we can ask what form of function has a small representation as a BDD; second, we can take advantage of that form when implementing the algorithm.

The key point about BDD's is that they are folded representations of the Shannon Cofactor Tree. Each node in a BDD represents a distinct subfunction. If there are relatively few nodes in a BDD, this indicates that the number of distinct subfunctions in the Cofactor Tree is fairly

[1] Everywhere in this paper, $\circ$ binds more tightly than $\vee$: $A \circ B \vee C = (A \circ B) \vee C$

[2] Here, subscript refers to the operation of cofactoring; e.g., $T_{x_1y_1} = T|_{x_1=y_1=1}$

small. Now, note that a cofactor of a function represented as a binary matrix is a rectangular submatrix. Identical subfunctions will map to identical submatrices; hence, the binary matrix representation of a function whose BDD representation is small will be a tiled matrix, one tile per edge of the BDD and one tile-type per node of the BDD.

The tiled nature of the binary matrix representation leads us to suspect that in the recursive descent algorithm we will meet the same subfunction in a number of different places. Plainly, we only need to compute the transitive closure of each subfunction once. As a result, we adopt a memo function approach: we keep a global hash table of arguments and results, and store each computation in the table; whenever a result is computed, we store the argument and result in the table. Before attempting any computation, we check to see if we've encountered this argument before, if we have, we simply return the precomputed result.

```
compute_closure(T) {
    if lookup(T, Result) return Result;
    T = ( T_{x̄1ȳ1}  T_{x̄1y1} )
        ( T_{x1ȳ1}  T_{x1y1} );
    U1 =compute_closure(T_{x1y1});
    U2 = U1 ∘ T_{x1ȳ1};
    E = compute_closure(T_{x̄1ȳ1} ∨ T_{x̄1y1} ∘ U2);
    U3 = T_{x̄1y1} ∘ U1;
    F = E ∘ U3;
    G = U2 ∘ E;
    H = U1 ∨ U2 ∘ F;
    Result = x̄1ȳ1 E ∨ x̄1 y1 F ∨ x1 ȳ1 G ∨ x1 y1 H;
    store(T, Result);
    return Result;
}
```

The previous special cases are subsumed by the table-lookup paradigm if we simply initialize the results table with the lines:

```
store(0, 0);
store(1, 1);
```

5 Transitive Closure of a Counter

Consider a simple n-bit counter. The next-state function $T(x, y)$ for this state machine may be written:

$$T(x, y) = [(y = x + 1) \; mod \; 2^n] \vee [y = x]$$

Where, in this case, $+$ refers to the operation of arithmetic addition. The matrix representation of T is captured in figure 2.

We now decompose the transition function, to obtain:

$$\begin{aligned} T_{x_1y_1} = T_{\overline{x_1}\overline{y_1}} &= (x \leq y \leq x+1) \\ T_{\overline{x_1}y_1} = T_{x_1\overline{y_1}} &= (x = 2^{n-1} - 1) \wedge (y = 0) \end{aligned}$$

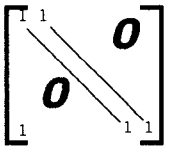

Figure 2: Transition Matrix for a Counter

Figure 3: Transition Matrices for the Cofactors of T

The transition matrix for $T_{x_1y_1}$ and $T_{\overline{x_1y_1}}$ is depicted in figure 3(a), and the transition matrix for $T_{\overline{x_1}y_1}$ and $T_{x_1\overline{y_1}}$ is depicted in figure 3(b).

The recursive descent procedure demands we compute the closure of $T_{x_1y_1}$, at the first step, and so we recursively decompose this matrix to obtain:

$$\begin{aligned} T_{x_1y_1x_2y_2} &= T_{x_1y_1\overline{x_2y_2}} = (x \leq y \leq x+1) \\ T_{x_1y_1\overline{x_2}y_2} &= [x = 2^{n-1} - 1] \wedge [y = 0] \\ T_{x_1y_1x_2\overline{y_2}} &= 0 \end{aligned}$$

The transition matrix for $T_{x_1y_1x_2y_2}$ and $T_{x_1y_1\overline{x_2y_2}}$ is depicted in figure 4(a), the transition matrix for $T_{x_1y_1\overline{x_2}y_2}$ is depicted in figure 4(b) and the transition matrix for $T_{x_1y_1x_2\overline{y_2}}$ is depicted in figure 4(c).

We solve for the transitive closure of $T_{x_1y_1x_2y_2}$ recursively, obtaining:

$$T^*_{x_1y_1x_2y_2} = y \geq x$$

The computation is given pictorially in figure 5. At this point, from the algorithm, we compute U_2 and $T_{x_1y_1\overline{x_2y_2}} \vee T_{x_1y_1\overline{x_2}y_2} \circ U_2$; it is easy to see that $U_2 = 0$, and hence $T_{x_1y_1\overline{x_2}y_2} \circ U_2 = 0$, and hence $T_{x_1y_1\overline{x_2y_2}} \vee T_{x_1y_1\overline{x_2}y_2} \circ U_2 = T_{x_1y_1\overline{x_2y_2}}$. The computation of E is thus exactly the computation of U_1, and the recursive computation is trapped out by the hash table lookup. It is easy to see that this was done at each level of the recursive computation of U_1, and hence there have been exactly $n - 2$ recursive calls to `compute_closure` to this point.

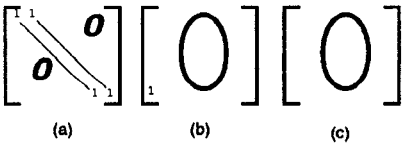

Figure 4: Transition Matrices for the Cofactors of $T_{x_1y_1}$

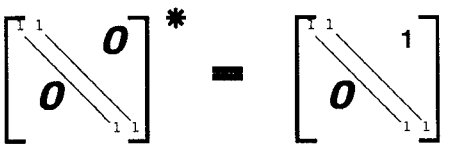

Figure 5: Transition Matrix for $T^*_{x_1y_1x_2y_2}$

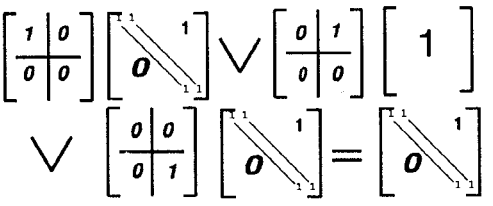

Figure 6: Computation of The Transition Matrix for $T^*_{x_1y_1}$

We have $E = U_1 = y \geq x$. Direct computation (most easily performed on the matrices) suffices to demonstrate that:

$$\begin{aligned} U_3 &= T_{x_1y_1\overline{x_2}y_2} \circ U_1 \\ &= \exists z\{[x = 2^{n-1} - 1] \wedge [z = 0] \wedge [y \geq z]\} \\ &= [x = 2^{n-1} - 1] \wedge [y \geq 0] \\ &= [x = 2^{n-1} - 1] \end{aligned}$$

Hence, $F = E \circ U_3 = \exists z\{[z \geq x] \wedge [z = 2^{n-1} - 1]\} = [x \leq 2^{n-1}] = 1$.

Since $U_2 = 0$ we have $G = 0, H = U_1$, completing the computation; the final matrix is pictured in figure 6.

Now, at the top level of the computation, we compute:

$$\begin{aligned} U_2 &= U_1 \circ T_{x_1\overline{y_1}} \\ &= \exists z\{[z \geq x] \wedge [z = 2^{n-1} - 1] \wedge [y = 0]\} \\ &= [2^{n-1} - 1 \geq x] \wedge [y = 0] \\ &= [y = 0] \end{aligned}$$

We then recursively compute the closure of $T_{\overline{x_1}\overline{y_1}} \vee (T_{\overline{x_1}y_1} \circ U_2)$, which we find is $[(y = x+1)\ mod\ 2^{n-1}] \vee (y = x)$ The transition matrix of this latter expression is shown in figure 7. The reader will note that this is simply a smaller version of the transition matrix of T. Immediately,

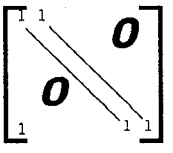

Figure 7: Transition Matrix Computation for $T_{\overline{x_1}\overline{y_1}} \vee T_{\overline{x_1}y_1} \circ U_2$

Circuit Name	Depth	Reachable States	Reachable Iteration	Linear Method	Square Method	Recursive Method
s208	18	17	0.02	0.02	0.02	0.02
s298	20	218	0.17	0.73	0.92	0.17
s344	8	2625	0.32	>40M	>40M	36.50
s349	8	2625	0.33	>40M	>40M	36.52
s382	152	8865	0.77	40.35	12.65	1.98
s386	9	13	0.00	0.03	0.03	0.00
s400	152	8865	0.72	40.35	12.65	1.97
s420	18	17	0.02	0.05	0.07	0.02
s444	152	8865	0.3	36.52	93.22	2.82
s510	50	47	0.02	0.72	0.20	0.02
s526	152	8868	0.37	>40M	>40M	4.20
s526n	152	8868	0.35	>40M	>40M	4.13
s641	8	1544	0.43	28.58	29.22	4.45
s713	8	1544	0.43	28.47	28.92	4.55
s820	12	25	0.02	0.07	0.03	0.02
s832	12	25	0.00	0.05	0.03	0.02
s838	18	17	0.03	2.95	3.02	1.50
s953	12	504	0.22	0.83	0.45	1.17
s1196	4	2616	0.87	2.57	2.37	4.77
s1238	4	2616	0.88	2.57	2.42	4.78
s1488	23	48	0.03	0.70	0.40	0.03
s1494	23	48	0.03	0.73	0.40	0.02

Table 1: CPU Times

therefore, the computation of the closure of the lower-right quadrant of this matrix is simply the computation of the closure of $y = x + 1$, and this was previously computed on the left branch of the top-level recursion; it is therefore not recomputed, but simply fetched from the hash table. By matrix similarity, the right-branch recursion is simply a recursion on a smaller version of this matrix again; the same observations concerning the lower-right quadrant of this matrix still apply, and as a result we conclude that each recursive call on a lower-right quadrant is caught by the table lookup check. Only the left-branch recursions require computation, and as a result there is precisely one invocation for each of the n variables. Hence there are $O(n)$ invocations of `compute_closure` in the computation of the original closure.

6 Experimental Results

We implemented the algorithms described above in a BDD package modeled after the efficient implementation of [2], and ran the algorithms over the sequential benchmark set on a Sun 4/2 with a memory limit of 40 MB. We obtained the results shown in tables 1 and 2. We implemented the reachable states iteration (1), the linear iteration for the computation of the transitive closure:

$$T_k = T_{k-1} \vee \exists z[T_{k-1}(x, z)T(z, y)]$$

a square iteration for the computation of the transitive closure:

$$T_k = T_{k-1} \vee \exists z[T_{k-1}(x, z)T_{k-1}(z, y)]$$

and the recursive computation given here. In table 1, the depth of the circuit (length of the longest simple path from

Circuit	BDD Sizes		Max BDD Size			
Name	$\|T\|$	$\|T^*\|$	Reach	Linear	Square	Recursive
s208	59	41	12	70	62	41
s298	453	688	60	1265	1265	688
s344	586	132639	629	>40M	>40M	132639
s349	586	132639	629	>40M	>40M	132639
s382	765	1143	172	4724	4412	1143
s386	79	11	13	94	94	21
s400	765	1143	172	4724	4412	1143
s420	171	223	20	289	275	223
s444	765	2055	203	11304	9841	2055
s510	148	56	16	285	273	76
s526	1334	5743	226	>40M	>40M	5743
s526n	1334	5743	226	>40M	>40M	5743
s641	4112	1212	164	7461	7461	2067
s713	4112	1212	164	7461	7461	2067
s820	94	9	12	107	101	32
s832	94	9	12	107	101	32
s838	2573	4054	33	5090	5033	4054
s953	886	548	548	1216	1216	753
s1196	4851	1176	985	4851	4851	1880
s1238	4851	1176	985	4851	4851	1880
s1488	177	10	21	406	386	76
s1494	177	10	21	406	386	76

Table 2: BDD Sizes for each method

the reset state) and the number of reachable states are also given.

Interestingly, in many of the examples, the linear method outperforms the square method. We attribute this to artifacts in the size of the transition relation at each step and to shallow state graphs. In almost every example the recursive method outperforms the iterative methods, suggesting that this method may have some practical promise.

Note that the reachable states method outperforms all of the transitive-closure methods on all the examples. This is unsurprising – when the reachable states operation terminates (i.e., when the state graph is shallow) it is a superior method for finding the reachable states.

What we find encouraging is that there is no circumstance yet found where the recursive method fails to find the closure – it seems to work equally well for both deep and shallow state graphs. Further, the maximum bdd size found during the computation, as shown in table 2, often approximates the size of the closure itself, indicating that this method is less vulnerable to intermediate expression swell than other methods.

To test our conclusion, derived in section 5, that the recursive method was effective for counters, we tested counters up to 64 bits. We found that the recursive method took less than 0.01 seconds for even the largest counter we tested.

7 Acknowledgements

The authors wish to thank M. Fujita, J. R. Burch, E. Sentovich and the DAC reviewers for comments on early drafts of this paper.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addision-Wesley, 1974.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference*, 1990.

[3] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference*, 1990.

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

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*. Springer-Verlag, June 1989.

[7] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *IEEE International Conference on Computer-Aided Design*, 1990.

[8] K. Karplus. Representing boolean functions with if-then-else DAGs. In *Decennial CalTech VLSI Conference*, 1989.