On the Construction of Multiple-Valued Decision Diagrams

D. Michael Miller
VLSI Design and Test Group
Department of Computer Science
University of Victoria
Victoria, BC
CANADA V8W 3P6
mmiller@csr.uvic.ca

Rolf Drechsler
Institute of Computer Science
University of Bremen
28334 Bremen
GERMANY
drechsle@informatik.uni-bremen,de

Abstract

Decision diagrams are the state-of-the-art representation for logic functions, both binary and multiple-valued. Here we consider ways to improve the construction of multiple-valued decisions diagrams (MDD). Efficiency is achieved through the use of a simple computed table. We compare the use of recursive MIN and MAX as primitive operations in multiple-valued decision diagram construction to the MV-CASE primitive which is a generalization of the if-then-else (ITE) commonly used in binary DD packages.

We also consider the use of cyclic negations and complements as MDD edge operations showing that for certain types of functions this approach can lead to significant reduction in MDD node count. They can also reduce the number of primitives that need to be explicitly implemented.

Experimental results showing the efficiency of the proposed approaches are given. The direct implementation of MDDs is briefly compared to representing MDDs using a BDD package.

1. Introduction

Reduced ordered binary decision diagrams (BDD) have been widely studied since their introduction by Bryant [2] in 1986. A good review can be found in [3] and the other articles included in that special issue. The extension to multiple-valued logic has been considered [7,8,9,15]. In the MVL case, a function is represented by a *directed acyclic graph* called a multiple-valued decision diagram (MDD). MDDs are ordered and reduced in a fashion analogous to the binary case and the resulting representation is termed a *reduced ordered MDD*. Since all diagrams considered in this work are reduced and ordered, we shall for brevity use MDD for multiple-valued decision diagrams and BDD for binary decision diagrams.

The efficient implementation of BDDs has been widely studied [1,11,13,14,15] and several highly efficient packages are available, *e.g*. CUDD [13]. Many binary techniques, or extensions thereof, are useful when implementing a package for the creation and manipulation of MDDs. But, there are issues new to the MDD case, particularly the choice of logic primitives to use in diagram construction and the appropriate use of edge operations.

Many available packages employ the *if-then-else* (ITE) primitive in BDD construction. This generalizes to a *CASE* operation [15] in the MVL situation. Drechsler and Thornton [4] have shown that BDDs can be efficiently constructed using *NAND* rather then ITE. Here we consider the use of *MIN* and *MAX* primitives and show this is more efficient than using CASE. We use recursive implementations of the MIN and MAX. Use of these dyadic recursive primitives allows a simpler computed table than is needed for CASE. An efficient computed table is critical to using the package for large problems.

It is quite common to use edge negations in BDDs [10,13,14]. In moving to MDDs, the concept of edge negation can be generalized. In [9] the present authors considered the use of cyclic negations on edges. Here we employ cyclic negations and complements together.

Adjacent level interchange and operator node techniques [5,6] based on adjacent level interchange for MDDs were considered in [9]. Here our concern, as in [4], is to construct the MDD as efficiently as possible and to minimize the implementation complexity of the MDD package. To that end, adjacent level interchange, dynamic variable reordering [12] and garbage collection are not implemented. The resulting package is applicable for quite large problems as shown by the experimental results.

The work presented here is an extension to our earlier work presented in [9]. A better approach to handling the unique table and the introduction of a computed table has greatly improved the efficiency of our MDD package, which for most problems is now at least an order of magnitude faster. As a result this paper shows it is now applicable to very large problems e.g. the 4-valued input, 2-valued output problem derived from apex5 has 59 inputs and 88 outputs and 1,227 cubes. The MDD is build in less than 3 CPU seconds on a SUN dual 166 MHz processor 690MP.

2. Multiple-Valued Decision Diagrams

We consider totally-specified functions $f(X)$, $X = \{x_0,x_1,\ldots,x_{n-1}\}$, where the x_i are p-valued. The function takes values from $(0,\ldots p-1)$ as well. Such a function can be represented by a *multiple-valued decision diagram* (MDD) which is a directed acyclic graph (DAG) with up to p terminal nodes each labelled by a distinct logic value $0,1,\ldots,p-1$. Every non-terminal node is labelled by an input variable and has p outgoing labelled edges; one corresponding to each logic value. The diagram is *ordered* if the variables adhere to a single ordering on every path in the graph, and no variable appears more than once on any path from the root to a terminal node.

A *reduced* MDD has no node where all p outgoing edges point to the same node and no isomorphic subgraphs. Clearly, no isomorphic subgraphs exist if, and only if, no two non-terminal nodes labelled by the same variable, have the same direct descendants. With proper management, reduction can be achieved as the decision diagrams are built. We assume all MDDs are reduced and ordered through the rest of this paper as that is the case of practical interest.

For multiple-output problems, we represent the functions by a single DAG with multiple top nodes, a structure called a *shared* MDD.

In the binary case, the number of edges from each non-terminal node is fixed at two. Here we must allow for a variable number of edges determined by the value of p for the function being represented. Our package uses the following node structure (expressed in C):

```
typedef struct node *DDedge;
typedef struct node *DDlink;

typedef struct node
{
   char value,flag;
   DDlink next;
   DDedge edge[0];
}node;
```

A node is a structure with several components:

value: This component is the index of the variable labelling a non-terminal node or the value associated with a terminal node.

flag: This field is used to track. visits to nodes in recursive descent algorithms.

next: This pointer is used to chain a node into linked lists for memory management and unique table management (see Section 4).

edge: This is an array of DDedge's which is declared empty but which is actually allocated as the number of edges needed for the node being created.

Note that DDedge and DDlink are similar pointer types but are used in different contexts – edges for the MDD structure and links for node management.

The node structure shown looks very much like the structure used in binary decision diagram packages. The critical difference is the specification of an array of edges rather than a fixed number (2 in binary). As noted, the dimension of the array is assigned dynamically when a node is created. For that reason, the array must be at the end of the structure.

Node space is allocated as required. Since garbage collection is not performed in the current package, 'freed' space is not recovered and we do not include a reference counter with a node. Discussion of appropriate garbage collection techniques can be found in [9,15].

Traversing an MDD from the top towards the terminal nodes can be accomplished with conventional recursive graph traversal techniques. In some instances, it is necessary to ensure a node is visited only once. Our package provides **flag** for that purpose.

3. Logic Primitives

Decision diagrams are normally constructed using one or a small set of primitive operations. For example, many binary packages construct diagrams by implementing the normal logic operations using the if-then-else (ITE) primitive defined as

$$\mathrm{ITE}(a,b,c) = \text{if } a \text{ then } b \text{ else } c \qquad (1)$$

For example, $\mathrm{AND}(a,b) = \mathrm{ITE}(a,b,0)$.

The MVL generalization of ITE is defined as

$$\mathrm{CASE}(a,b_0,b_1,b_2,\ldots) = b_a \qquad (2)$$

Common MVL logical operations can be expressed in terms of CASE. For example, for $p=4$,

$$\mathrm{MIN}(a,b) = \mathrm{CASE}(a,0,\mathrm{CASE}(b,0,1,1,1),\ \mathrm{CASE}(b,0,1,2,2),b) \qquad (3a)$$

$$\mathrm{MAX}(a,b) = \mathrm{CASE}(a,b,\mathrm{CASE}(b,1,1,2,3),\ \mathrm{CASE}(b,2,2,2,3),3) \qquad (3a)$$

The implementation of ITE on binary decision diagrams (see [15]) is readily extended to implementing CASE on MDDs as outlined in Figure 1. Given that, MIN and MAX can be implemented using (3a) and (3b). Other common logic operations such as truncated-SUM can be implemented in a similar fashion.

Drechsler and Thornton [4] have considered the direct use of NAND in place of ITE in constructing binary DDs. Here we consider MIN and MAX. Figure 2 outlines the recursive implementation of MIN (MAX is analogous). It is important to note that Figure 2 is the direct recursive implementation of MIN. This is an alternative to implementing MIN using CASE.

```
CASE(A,B0,B1,…,Bp-1)
  if(terminal(A)) return(BA)
  TOP=top variable of A,B0,B1,…,Bp-1
  for 0<=i<=p-1
    if(id(A)==TOP) EA=CHILD(A,i)
    else EA = A
    for 0<=j<p
      if(id(Bj)==TOP) EBj=CHILD(Bj,i)
      else EBj = Bj
    Ci=ITE(EA,EBj, …, EBj)
  R=create node(TOP,C0,...,Cp-1)
  return(R)
```

Figure 1: Implementation of CASE

The structure of the implementations of CASE and MIN are quite similar. The basic difference is that CASE has p+1 parameters whereas MIN (and likewise MAX) has only 2. This affects the efficiency of the computed table (see next Section). Experiments reported in Section 5 show the direct implementation of MAX and MIN is considerably more efficient than using CASE to implement them.

4. Unique and Computed Tables

It is common practice to use a unique table to avoid creating multiple instances of a node. We use a simple hashing approach. Our unique table has B buckets. When a node is created as in the implementations of CASE and MIN shown in Figures 1 and 2, respectively, it is 'hashed' to a particular bucket position k using

$$k = \Sigma\,(E_i + E_i >> b\,) \bmod B \qquad (4)$$

where E_i is the i^{th} edge from the node treated as an integer, B is the number of buckets (should be a power of two) and b is the base 2 log of B. Because B is a power of two, the *mod* operation can be implemented as a logical *and* with $B-1$ rather than the expensive *mod* computation. Be use B = 4,096 in our experiments described below.

Each bucket in the hash table heads a list (initially empty) of nodes. When a node is required, the chain for the bucket it is hashed to is linearly searched. If the node is found, the instance found is used, otherwise, the node is created and added to the beginning of the bucket chain, for possible further use. Experiment has shown that adding to the front of the chain is most effective due to the local computation nature of MDD construction.

Use of a unique table as described ensures that only one instance of a node is ever created which deals with the need to identify isomorphic subgraphs.

MDDs are constructed by performing logic operations on MDDs, e.g. CASE, MIN and MAX described above. As shown the procedures are recursive. It is common to have to perform the same operation on the same diagrams many times during the overall construction of an MDD for a large problem. This duplication of effort can be significantly reduced if not eliminated using a computed table.

```
MIN(A,B)
  if(A==0 or B==0) return(0)
  if(A==p-1) return(B)
  if(B==p-1) return(A)
  if(A and B are terminals)
     return(min of A and B)
  TOP=top variable of A and B
  for 0<=i<=p-1
    if(id(A)==TOP) EA=CHILD(A,i)
    else EA = A
    if(id(B)==TOP) EB=CHILD(B,i)
    else EB = B
    Ci=MIN(EA,EB)
  R=create node(TOP,C0,...,Cp-1)
  return(R)
```

Figure 2: Implementation of MIN

In our case, the computed table consists of a number of rows each of which stores the result of a previous computation by retaining the type of operation, edge pointers identifying the MDDs that were combined and a final edge pointer that identifies the MDD resulting from the computation. It is a global computed table [15] in that prior computations are shared from one MDD computation to the next. This is particularly useful in a shared MDD environment.

The computed table rows are initially empty. Each time a computation is to be performed, it is 'hashed' to a row of the computed table using a formula analogous to (4) which combines the edge pointers for the MDDs to be combined. The identified row of the computed table is checked to see if it contains the required computation. If it does, the result pointer is returned. If it does not, the required computation is performed, and the appropriate information is stored in that row of the computed table. Note that this information replaces any prior computation stored there. In our experiments below we use a table with 4,096 rows.

The computed table approach is readily inserted into the CASE and MIN procedures given in Figures 1 and 2. In each case, the check for the required computation in the computed table is placed just after the handling of terminal cases and before computation of TOP. In addition, the code to place the information about the computation in the computed table goes just before the `return(R)` that returns the result. A very significant difference is that the computed table for CASE has p+1 edge pointers whereas the computed table for MIN (and MAX) has only two.

5. MDD Construction with MIN and MAX

To evaluate the methods discussed thus far, we present results on converting cube list specifications to decision diagrams. We use mostly well-known binary benchmark problems converted to 4-valued input, 2-valued output problems. The conversion of the binary inputs to 4-valued inputs is done by taking the inputs in pairs from left to right in order of the normally available specification. If there is an odd number of inputs, the rightmost input remains a binary input. There is no attempt to optimise the pairing of inputs as our objective is simply a set of test benchmarks to validate MDD techniques. In particular, we emphasize that we are using these binary problems as a set of unbiased benchmarks – our interest here is not the optimal encoding of binary functions as MVL ones.

The outputs are left as binary to allow the constructed MDD to be mapped back to the binary case to verify the correctness of the MDD. In particular, a simple depth first traversal of the paths in the MDD is performed. Each path represents a disjoint 4-valued input cube which is easily mapped to the binary domain. The path leads to a 0 or 1. Those leading to 1 are output. This process is repeated for each function output resulting in a disjoint cube list specification of the binary problem from which the 4-valued input binary output test case was derived. The 'verify' option of the minimizer *espresso* is then used to compare the original problem specification to the cube list obtained from the MDD.

Table I shows the results of constructing MDDs for 4-valued input binary output problems derived from binary benchmark problems as described above. These results were obtained using the recursive implementations of MIN and MAX. The construction involves building MDD literals for each element of an input cube and then using MIN to combine those to form a single MDD representing the input side of the cube. The MIN of that cube and 1 is then found and the result is combined into the MDD for each appropriate output function using MAX.

Table I shows that the approach is quite efficient even for some quite large problems. The table is sorted in decreasing order of the number of MDD nodes. Comparing the two timings, (a) which is with the computed table approach outlined above and (b) which is without a computed table, clearly shows the advantage of using such a table. The speedup varies from virtually nothing for the small problems to a factor of about 30 to 50 for the large *apex* problems.

Table I shows the number of calls of MIN and MAX when the computed table is used. Note that only calls requiring computation are counted including those found in the computed table. Those resolved as terminal cases are not included in the counts. The recursive calls of MIN and MAX are counted. It is peculiar to the nature of our test cases that calls to MAX generally dominate. This is because the input side of the two level specification deals with combining literals which are relatively simple MDD structures whereas the output side must combine the more complex MDDs representing the cubes. That requires considerably more computation. Problem *e64* is a unique case where each output is a single cube so no calls to MAX are required. To further emphasize the importance of the computed table, we note that constructing the MDDs for *apex2* without a table requires 22.2 million MAX calls and 1.1 million MIN calls.

Table II compares MDD construction using the recursive MIN and MAX and the CASE-based MIN and MAX for the same problems as above. Again, no edge negation operations are used. The table is ordered by decreasing number of computed table references for the recursive case. For each approach, the table shows the number of computed table references, the percentage that were 'hits' where a hit means the required value was found in the computed table, and the CPU time required.

The right two columns compare the number of computed table references and the CPU time required for the two methods. The recursive MIN and MAX approach is clearly significantly faster and requires significantly fewer computed table references. It is interesting that the CASE-based approach consistently shows a much higher computed table hit rate but this does not translate into a faster method. This is because each non-terminal computation of the recursive MIN or MAX requires p recursions, whereas the CASE approach requires three CASE computations for a MIN or MAX and each non-terminal computation of a CASE requires p+1 recursions. The higher hit rate tends to be concentrated on the 'inner' CASEs with predominantly constant arguments.

6. Edge Negations

There have been several suggestions [1,10,14] for the use of edge negations as a means to further reducing the size of a decision diagram. Unlike the binary case where there is only one definition of negation, we must consider alternatives in the multiple-valued case.

Cyclic negation of x by k will be denoted x^k, and is defined as $x^k = (x+k) \bmod p$. A second form of negation in multiple-valued logic is the *complement* defined as $\overline{x} = p-1-x$. We also use x' to denote the complement. Note that for p=2, both are binary negation.

The following identities are of particular use here:

Identity 1: $(x^k)^j = x^{k+j}$

Identity 2: $(x^k)' = \overline{x}^{p-k}$

Identity 3: $\overline{x}' = x$

Identity 4: $((x^k)')^j = (x^{k+p-j})'$

Identity 5: $(x^k)^{p-k} = x$

Identity 6: $(((x^k)')^k)' = x$

A review of the above identities shows that any sequence of cycles and complements can be reduced to a single cycle possibly followed by a single complement. Hence we will label each edge in an MDD with a single cycle 0…*p-1* and an optional complement that when present follows the cycle. The interpretation is that the edge identifies the subfunction found by applying the indicated cyclic possibly followed by a complement to the function represented by the subgraph to which the edge points.

Use of cyclic negation as an edge operation in MDDs was discussed by the present authors in [9]. For cycles alone, two rules are required to ensure the representation is canonic:

1. there is a single terminal node whose value is 0 (other terminal values required can be generated using a cycle on the edge pointing to the terminal 0);
2. the 0-edge from a node never has a cycle (when one is called for it is promoted to the edge pointing to the node itself and the cycles on all other edges from the node are adjusted accordingly).

When using a cycle and complement together on a single edge, rule 1 remains the same. Rule 2 is extended so that if a cycle-complement pair is moved from the 0-edge to the edge leading to a node, every other edge from the node is adjusted by post-multiplying its cycle negation pair by the inverse of the cycle-negation pair from the 0-edge. The inverse is identified using identity 5 or identity 6 and the post-multiplication is carried out using identity 1 or 4.

Rather than add a separate negation component to our representation of an edge, we store the cycle and complement in the bottom three bits of the pointer: one bit for complement and two for the cycle allowing us to handle the cases *p=2*, *3* or *4*. This is possible because memory management systems typically align dynamically allocated memory blocks on fixed address boundaries (e.g. many systems align the block on addressed that are multiples of 8). Hence, there are unused bits (fixed to 0) at the low end of dynamic allocation addresses. There may also be 'available' bits at the high end if a sufficiently long address is used. In our implementation, the actual bits 'borrowed' from the pointer to store the cycle are controlled by symbolic constants and macros to facilitate porting the package to alternative systems. This approach also makes it straightforward to make the negations a separate field in the node structure if that is required as would be necessary in our case for *p>4*. Note that the hashing function used (4) would have to be altered to be effective if the position of the 'borrowed' bits is moved.

7. MDD Construction with Edge Operations

Table III shows results of MDD construction using edge operations. The problems are those from before but there is a very significant difference here. For the experiments described above the binary outputs were coded 0 and 1 as one would expect. But that coding does not allow for the effective use of cyclic negation. Hence for the experiments presented in Table III the binary output values 0 and 1 are coded as 0 and 2 in formulating the MVL problems.

The table is sorted by the MDD size without negations (a). For each problem, we present the size of the MDD when edge negations are used (b) and the relation between the two sizes. Many functions show little improvement but there is substantial improvement for arithmetic functions (e.g. alu4 and 74181) as well as others. Similarly good improvement is also found for the symmetric problems (rd53, rd73 etc. not shown due to space limitations) but these are small examples and larger symmetric problems need to be tried to confirm that this is a real effect.

MDDs were built for each problem using three methods:

1. recursive MIN with MAX implemented in terms of MIN;
2. recursive MAX with MIN implemented in terms of MAX;
3. recursive MIN and recursive MAX implemented separately.

The MDDs are the same size for all three methods but it is important to note that the resulting MDDs are not identical. The MDD are canonic for each separate method, i.e. for each method the MDD constructed for a particular function is unique, but canonicity does not carry across the three approaches. The MDD do have the same structure but the selection and placement of edge negations is not the same. A traversal of an MDD using the relations given above does allow one MDD to be transformed to another. The best way to deal with the lack of overall canonicity is under investigation. For many applications it is not an issue since only one of the three approaches offered would be used.

The results in Table III show the three approaches are very similar in terms of overall complexity and computational cost. The results show the computed table approach is quite effective

We observe that the total nodes allocated in constructing the MDD is the same regardless of the method used. The MIN only approach consistently makes fewer references to the unique and computed tables with generally lower hit rates but the differences are not appreciable.

8. Representing MDDs using BDDs

An MDD can be encoded as a BDD and implemented using a BDD package. In such an approach, an MV input is represented by a binary variable encoding. An MV output can be similarly encoded, but another alternative (the one used in VIS [16]) is to encode a p-valued output as a set of p binary functions, *i.e.* as decisive functions with one function per logic value.

Using a BDD package to represent an MDD does have advantages since BDD packages such as CUDD [13] are highly optimized and implement key features such as edge negation, variable reordering and garbage collection. CUDD is typically two to five times faster than our package for the examples described earlier. This is due to its highly optimized and clever implementation and not indicative that the BDD approach is in principle superior to the direct MDD approach.

However, there are advantages to representing an MDD directly as described in this paper. Edge negations in a BDD representation of an MDD can reduce the complexity but they do not correspond to MVL negation or complementation. Indeed implementation of MVL negation and complementation appears to be quite complex on the BDD representation of an MDD. Similarly, other common operation needed for example in logic synthesis applications are more complex on a BDD representation of an MDD then they are on a direct MDD implementation.

A proper comparison of the two approaches is a complex task which is ongoing. It certainly appears that the best approach depends on the intended application and that neither approach will be globally better. Indeed transformation between the two methods of representation is a possibility that should be pursued.

9. Concluding Remarks

This paper has examined the construction of MDDs. A simple computed table greatly improves the efficiency. Experimental results on a set of benchmark functions clearly show the advantage of using recursive MIN and MAX primitives rather than implementing them using a CASE primitive. Use of cyclic negation and complement as edge operations can reduce the size of the MDD with no noticeable increase in computation cost. The results show that there is little to choose between the use of the three methods applied in Table III. As noted in [15] for the binary case, the fact edge operations (cycle and complement) do not generally significantly reduce the size of the MDD does not render them unimportant, since using them reduces the number of primitive that must be explicitly implemented. For example, using complement requires only MIN or MAX but not both be explicitly implemented. We do note that use of MIN and MAX together with cycles does not require complements on the edges. There are clearly tradeoffs depending on the primitives required which must be carefully investigated for each application.

Future work will include implementation of other MVL logic operations such as truncated–sum and development of routines to construct MDDs from circuit level descriptions rather than cube list specifications. Optimisation of the package continues and we are continuing to consider the relation between the diagrams constructed using MIN and MAX as the single primitive.

As noted in Section 8, we are pursuing an in-depth comparison of the direct MDD approach with the BDD representation of MDDs, as well as the complexity and usefulness of transforming between the two representations.

The MDD package described in this paper is robust and provides sufficient efficiency to be of interest to researchers interested in applying MDD in various application areas.

The MDD package described is freely available at www.csr.uvic.ca/~mmiller/MDD.

Acknowledgements

The work reported in this paper was supported in part by a Research Grant from the Natural Sciences and Engineering Research Council of Canada. We thank the referees for their helpful suggestions regarding this paper.

References

[1] Brace, K. S., R L. Rudell and R. E. Bryant, "Efficient implementation of a BDD package", *Proc. Design Automation Conference*, pp. 40-45, 1990.

[2] Bryant, R.E., "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers*, V. C-35, no. 8, pp. 677-691, 1986.

[3] Drechsler, R, and D. Sieling, "Binary decision diagrams in theory and practice," *Int. Journal on Software Tools for Technology Transfer*, 3, pp. 112-136, 2001.

[4] Drechsler, R., and M.A. Thornton. "Fast and Efficient Equivalence Checking based on NAND-BDDs," *Proceedings of IFIP International Conference on Very Large Scale Integration* (VLSI'01), Montpellier, pp. 401-405, 2001

[5] Hett, A., R. Drechsler and B. Becker, "MORE: Alternative implementation of BDD packages by multi-operand synthesis," *Proc. European Design Automation Conference*, pp. 164-169, 1996.

[6] Hett, A., R. Drechsler and B. Becker, "Reordering based synthesis," *Proc. Reed-Muller Workshop 97*, pp. 13-22, 1997.

[7] Lau, H.T., and C.-S. Lim, "On the OBDD representation of general Boolean functions," *IEEE Trans. on Comp.*, C-41, No. 6, pp. 661-664, 1992.

[8] Miller, D.M., "Multiple-valued logic design tools," (Invited Address) *Proc. 23rd Int. Symp. on Multiple-Valued Logic*, pp. 2-11, May 1993.

[9] Miller, D. M., and R. Drechsler, "Implementing a multiple-valued decision diagram package," *Proc. 28th Int. Symp. on Multiple-Valued Logic*, pp. 52-57, May 1998.

[10] Minato, S., N. Ishiura and S. Yajima, "Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation," *Proc. ACM/IEEE Design Automation Conference*, pp. 52-57, 1990.

[11] Minato, S., "Graph-based representations of discrete functions," *Proc. IFIP WG 10.5 Workshop on the Application of Reed-Muller Expansion in Circuit Design*,, pp. 1-10, 1995.

[12] Rudell, R. "Dynamic variable ordering for ordered binary decision diagrams," *Proc. IEEE/ACM ICCAD*, pp. 43-47, 1993.

[13] Somenzi, F., "CUDD: CU Decision Diagram Package," http://bessie.colorado.edu/~fabio/ CUDD

[14] Somenzi, F., "Efficient manipulation of decision diagrams," *Int. Journal on Software Tools for Technology Transfer*, 3, pp. 171-181, 2001.

[15] Srinivasan, A., T. Kam, S. Malik, and R.E. Brayton, "Algorithms for discrete function manipulation," *Proc. ICCAD*, pp. 92-95, 1990.

[16] VIS Group, "VIS: A System for Verification and Synthesis," *Proc. Computer-Aided Verification*, LNCS 1102, pp. 428-432, 1996.

	Binary problem specification			**MDD using min-max and computed table**				**No computed table**	
	in	**out**	**Cubes**	**MDD nodes**	**calls to MAX**	**calls to MIN**	**CPU (a) msec.**	**CPU (b) msec.**	**(a) / (b)**
apex5	117	88	1,227	3,543	98560	13387	2831	803,530	0.4%
apex2	39	3	1,035	3,471	445,905	24,605	5,381	1,865,383	0.3%
apex1	45	45	206	3,082	87034	4047	930	556,421	0.2%
seq	41	35	1,459	1,301	20605	20857	1300	6,061	21.4%
e64	65	65	65	1,020	0	2969	90	12,602	0.7%
alu4	14	8	1,028	787	38914	11577	700	1,240	56.5%
vg2	25	8	110	733	7636	1630	90	1,830	4.9%
apex4	9	19	438	640	8184	4786	200	340	58.8%
apex3	54	50	280	598	4468	3022	300	360	83.3%
duke2	22	29	87	562	3475	1537	70	610	11.5%
misex3	14	14	1,848	434	18192	19099	720	1,110	64.9%
74,181	14	8	1,133	403	17246	8772	420	740	56.8%
clip	9	5	167	118	1483	1450	40	70	57.1%
misex2	25	18	29	115	136	340	20	40	50.0%
mdiv7	8	10	256	104	3276	2329	60	120	50.0%
alu2	10	8	70	91	793	417	10	40	25.0%
bw	5	28	25	89	250	409	20	20	100.0%
sao2	10	4	58	82	886	505	20	30	66.7%
misex1	8	7	32	48	92	114	10	10	100.0%
rd84	8	4	256	32	1164	1585	50	60	83.3%
rd73	7	3	141	27	647	946	40	40	100.0%
9sym	9	1	88	19	988	856	20	30	66.7%
rd53	5	3	32	17	77	166	10	10	100.0%
postal	8	1	256	12	199	125	10	10	100.0%

Table I: Benchmark results for MDD constructed using MIN-MAX and a computed table

	Recursive MIN-MAX			CASE-based MIN-MAX			Comparison	
	computed table references (a)	% hits	CPU msec. (b)	computed table references (c)	% hits	CPU msec. (d)	% (a) / (c)	% (b) / (d)
apex2	325,259	35.8%	5,381	1,041,387	64.90%	9,571	31.23%	56.22%
apex5	67,529	29.6%	2,671	180,483	71.60%	3,171	37.42%	84.23%
seq	57,357	35.2%	1,300	153,969	66.30%	1,650	37.25%	78.79%
misex3	50,443	34.8%	720	131,047	67.30%	1,060	38.49%	67.92%
alu4	49,486	34.0%	700	142,906	66.00%	990	34.63%	70.71%
apex1	46,414	15.3%	930	210,639	70.40%	1,570	22.03%	59.24%
74181	30,561	30.8%	420	94,272	68.30%	650	32.42%	64.62%
apex4	15,141	21.9%	200	60,349	68.20%	440	25.09%	45.45%
apex3	8,248	18.9%	300	35,264	62.80%	460	23.39%	65.22%
vg2	6,178	22.3%	90	25,120	66.70%	160	24.59%	56.25%
mdiv7	4,971	21.4%	60	22,752	77.50%	130	21.85%	46.15%
e64	4,935	28.3%	90	16,017	59.10%	170	30.81%	52.94%
duke2	4,204	20.9%	70	15,029	63.40%	140	27.97%	50.00%
rd84	3,936	31.9%	50	9,482	72.30%	80	41.51%	62.50%
clip	3,619	28.7%	40	10,379	63.30%	90	34.87%	44.44%
9sym	2,315	24.8%	20	7,513	62.20%	80	30.81%	25.00%
rd73	1,848	62.4%	40	6,241	62.60%	30	29.61%	133.33%
sao2	1,559	27.7%	20	3,907	54.80%	40	39.90%	50.00%
alu2	1,147	26.2%	10	3,474	62.40%	40	33.02%	25.00%
bw	979	30.1%	20	2,223	63.20%	30	44.04%	66.67%
misex2	577	23.4%	20	1,908	60.40%	30	30.24%	66.67%
rd53	385	22.6%	10	1,019	53.70%	10	37.78%	100.00%
postal	353	22.9%	10	1,066	59.30%	20	33.11%	50.00%
misex1	278	29.1%	10	694	54.60%	20	40.06%	50.00%

Table II: Comparison of MIN-MAX and CASE based MDD construction

(Note: All experiments reported in this paper were performed on a SUN 690MP with dual Ross Technology 166 MHz processors and 128MB)

	(a)	(b)	% (b) / (a)	Method	Total nodes	Unique table references	Unique table % hits	Computed table references	Computed table % hits	Computed table % replacements	CPU msec.
apex5	3,543	3,535	99.8%	MIN	27,584	45,158	38.9%	105,384	63.3%	32.9%	2,741
				MAX	27,584	47,878	42.4%	111,981	62.9%	35.2%	2,670
				Both	27,584	47,483	41.9%	112,163	63.4%	33.4%	2,680
apex2	3,471	3,403	98.0%	MIN	91,195	194,508	53.1%	422,193	53.7%	45.3%	5,311
				MAX	91,195	207,577	56.1%	458,516	53.9%	45.7%	5,381
				Both	91,195	206,333	55.8%	466,286	55.0%	44.2%	5,341
apex1	3,082	3,033	98.4%	MIN	32,177	39,049	17.6%	90,407	58.0%	37.4%	940
				MAX	32,177	39,349	18.2%	90,642	57.8%	39.9%	960
				Both	32,177	39,251	18.0%	90,975	58.1%	38.8%	830
seq	1,301	1,217	93.5%	MIN	16,868	36,766	54.1%	40,746	39.2%	50.9%	1,170
				MAX	16,868	37,727	55.3%	41,697	38.0%	57.1%	1,230
				Both	16,868	36,658	54.0%	41,314	40.2%	50.4%	1,150
e64	1,020	1,014	99.4%	MIN	2,141	3,502	38.9%	2,941	29.8%	33.6%	70
				MAX	2,141	3,507	39.0%	2,945	29.7%	35.0%	80
				Both	2,141	3,502	38.9%	2,941	29.8%	33.6%	80
alu4	787	648	82.3%	MIN	15,601	32,264	51.6%	48,210	41.1%	50.5%	580
				MAX	15,601	32,714	52.3%	48,639	40.5%	55.3%	600
				Both	15,601	32,564	52.1%	50,383	42.9%	49.8%	570
vg2	733	717	97.8%	MIN	3,409	4,693	27.4%	8,893	52.9%	19.3%	90
				MAX	3,409	4,725	27.9%	8,929	52.7%	28.5%	80
				Both	3,409	4,732	28.0%	9,106	53.5%	25.1%	100
apex4	640	632	98.8%	MIN	8,493	11,669	27.2%	12,892	26.3%	46.0%	220
				MAX	8,493	11,923	28.8%	12,994	24.9%	59.7%	230
				Both	8,493	11,733	27.6%	12,944	26.1%	53.3%	220
apex3	598	552	92.3%	MIN	5,091	6,672	23.7%	7,472	24.9%	36.8%	280
				MAX	5,091	6,737	24.4%	7,506	24.2%	50.9%	290
				Both	5,091	6,703	24.1%	7,507	24.7%	44.4%	310
duke2	562	544	96.8%	MIN	2,431	3,286	26.0%	4,919	44.7%	19.5%	80
				MAX	2,431	3,289	26.1%	4,921	44.7%	26.5%	70
				Both	2,431	3,296	26.3%	4,978	45.1%	23.8%	70
misex3	434	379	87.3%	MIN	15,254	32,555	53.1%	36,620	42.1%	47.0%	630
				MAX	15,254	33,830	54.9%	37,599	40.1%	54.5%	650
				Both	15,254	32,769	53.5%	37,241	42.4%	47.3%	680
74,181	403	350	86.8%	MIN	11,630	20,920	44.4%	24,898	38.4%	45.9%	370
				MAX	11,630	21,261	45.3%	25,175	37.5%	54.4%	420
				Both	11,630	21,043	44.7%	25,923	40.3%	46.6%	390
clip	118	106	89.8%	MIN	1,524	2,555	40.4%	2,849	33.7%	22.6%	40
				MAX	1,524	2,562	40.6%	2,852	33.5%	27.6%	50
				Both	1,524	2,575	40.9%	2,923	34.5%	25.0%	40
misex2	115	113	98.3%	MIN	307	434	29.5%	468	40.4%	1.5%	10
				MAX	307	435	29.7%	470	40.4%	5.5%	20
				Both	307	436	29.8%	472	40.5%	2.3%	10

Table III: Comparing use of cycles and complements with recursive MIN and MAX
NOTE: binary outputs are coded as 0 (logic 0) and 2 (logic 1) see Section 7