Implementing a Multiple-Valued Decision Diagram Package

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
Albert-Ludwigs-University
79110 Freiburg im Briesgau
GERMANY
drechsle@informatik.uni-freibug.de

Abstract

Decision diagrams are the state-of-the-art representation for logic functions, both binary and multiple-valued. Here we consider issues regarding the efficient implementation of a package for the creation and manipulation of multiple-valued decision diagrams (MDDs). In particular, we identify issues that differ from binary decision diagram packages.

We describe a matrix method for level interchange in MDDs that is essential for implementing variable reordering strategies. In addition, it is the basis for a novel approach to performing logic operations on MDDs, which we also present. Experimental results demonstrate the efficiency of this approach.

1. Introduction

Since Bryant [2] introduced reduced ordered binary decision diagrams (ROBDD) in 1986, they have been much studied, including their extension to multiple-valued logic [6,8] where a function can be 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* (ROMDD).

The efficient implementation of ROBDDs has been widely studied [1] and several highly efficient packages are available [10]. Many of the binary techniques, or extensions thereof, are useful when implementing a package for the creation and manipulation of ROMDDs. But, there are new problems as well.

First, the number of edges emanating from a node becomes variable. Since the representations of practical functions can require quite a large number of nodes (into the thousands), it is not practical to assume the highest number of edges and thereby waste space for nodes not requiring that many. We consider how to make the number of edges flexible.

It is quite common to use edge negations in ROBDDs [7,8]. In moving to ROMDDs, the concept of edge negation can be generalised. We consider how to incorporate cyclic negation as an edge operation and how to integrate it with various operations on ROMDDs. A similar approach can be used for the MV complement.

A major contribution of this paper is that we generalise the adjacent level interchange approach introduced in [4] to the performing of logical operations on ROMDDs.

2. MDD Representation

We consider totally-specified p-valued functions $f(X)$, $X = \{x_0, x_1, \ldots, x_{n-1}\}$, where the x_i are also p-valued. 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 edges; one corresponding to each logic value. They are so labelled. 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, although as we will see, complications arise in the level exchange approach to performing logic operations. 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* ROMDD.

Two important issues in representing decision diagrams are the data structures for each node and the techniques used to provide quick navigation through the diagram.

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 *link;

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

DDedge and link are both pointer types directed to a node. Two types are used to distinguish the context of the pointers. DDedges are used for the edges in the ROMDD whereas links are used for node management. For clarity in the explanation below, we shall distinguish the two situations as edges and links.

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.

ref: This is a reference count which is the number of edges pointing to the node. It is used to implement conventional reference count based garbage collection techniques.

next, previous: These pointers are used to chain a node into an appropriate linked list for node management (see below).

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..

flag: Flag is used to ensure each node is visited once when traversing the decision diagram in a depth-first manner.

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, **edge** must be at the end of the structure.

The space allocated for a new node depends on *p*, the number of outgoing edges. For the structure above, we have

```
newp = malloc(sizeof(node) +
         p * sizeof(DDedge));
```

Nodes often cease to be required in the course of manipulating ROMDDs. Rather than deallocating the space (free in C), we chain free nodes into available space chains (this is an example of the use of the component next in structure node). We use a separate chain for each number of edges. When space for a new node is required we check the available space chain for the required number of edges and recover a node from there if possible. Only if the available space chain is empty do we allocate new space from the heap using the call to malloc indicated above.

Nodes in an ROMDD are traversed in various ways (i) beginning at the top node the ROMDD is traversed by following edges from nodes to the descendants, (ii) the ROMDD is traversed 'horizontally' *i.e.* the nodes labelled by the same variable are traversed for each variable in turn, (iii) a combination of the two approaches. In addition, an efficient method is required to identify equal nodes so that each time creation of a node is considered, it is quick to determine if that node already exists. This is the key to reducing a ROMDD as it is created.

Traversing an ROMDD from the top towards the terminal node can be accomplished with conventional recursive graph traversal techniques. In some instances, it is necessary to ensure a node is visited only once. This can be accommodated by associating a flag with each node. Our package use **flag** for that purpose.

Traversal across a variable level, and equal node identification, are facilitated by our package's use of hashing and node chains (using the components **next** and **previous**). We keep a separate hash table for each variable. Each table is an array of pointers where each pointer leads to a bi-directional linked list of nodes labelled by the relevant variable. To determine to which table entry a node belongs, we treat the outgoing edges as integers and compute their sum modulo the number of entries in the table. Note that by choosing the number of table entries to be a power of two, the *mod* operation reduces to a logical *and* operation which is typically faster.

Since addresses of dynamically allocated blocks are usually aligned on 8 byte (or higher) boundaries, this simple hashing scheme would seem to lose 3 (or more) bits of significance. However, as we show below, the bottom bits of the address are 'borrowed' for other purposes and do prove to be significant in the hashing operation.

Since there is a separate table for each variable, traversing all nodes associated with a variable simply requires we traverse all lists emanating from the table. Determining that a node to be created already exists requires we search only one list of nodes in the appropriate table, the one identified by hashing the outgoing edges. There are numerous alternatives to this approach to hashing, but this technique has been found to be effective, particularly given that the number of outgoing edges from a node is variable.

3. Edge Negations

There have been several suggestions [7,8] 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 $x^{\uparrow k}$, and is defined as $x^{\uparrow k} = (x+k) \bmod p$. We attach an output cycle value 0...*p-1* to every edge in the graph. The interpretation is that the edge identifies the subfunction found by applying the indicated cyclic negation to the function represented by the subgraph to which the edge points. For *p=2*, this is binary negation. An ingoing edge is added to the top node as the entire function may have to be cycled. To preserve the uniqueness of the representation, the single terminal node is always labelled 0, and all 0-edges have cycle value 0.

Since our primary interest is for functions with *p*=2, 3 or 4, rather than add a separate cycle component to our representation of an edge, we store the cycle value in the bottom two bits of the pointer. This is possible since, as noted above, there are usually unused bits (fixed to 0) at the low end of dynamic allocation addresses and 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. The hashing function used may have to be adapted to maintain full significance if the position of the 'borrowed' bits is moved

A second form of negation in multiple-valued logic is the complement defined as $\bar{x} = p-1-x$. Regardless of *p*, this requires a single bit which can again be 'borrowed' from the pointer address. In both cases, the negation bits must of course be masked out when the pointer is being used as an address. Once again the convention is that a complement never appears on a 0-edge.

4. Adjacent Level Interchange

Recall that an ROMDD has a certain underlying variable ordering. It is often necessary to exchange two adjacent variables, for example in searching for a variable ordering for which the size of the ROMDD is smaller, or when performing logical operations on ROMDDs (see section 6). We refer to the operation as *adjacent level interchange* since in the latter case nodes represent operators as well as variables. The notion of level comes from the fact that it is always possible, and in fact typical, that a ROMDD be drawn, or thought of, with all nodes labelled the same in a horizontal row across the ROMDD.

We consider the case of interchanging the levels associated with α and β where the former immediately precedes the latter. We assume that all nodes labelled α have *p* outgoing edges and all nodes labelled β have *q* outgoing edges where *p* and *q* may or may not be equal. This is enforcing the reasonable constraint that while nodes with different labels may have differing numbers of outgoing edges, all nodes with the same label have the same number of outgoing edges. The key is to perform the required interchanges as local transformations.

Consider a node γ labelled α. We construct a matrix **T** with *p* rows and *q* columns. For *i*=0,1,…, *p*-1,

(i) If the *i*-edge from γ leads to a node δ labelled β, then for *j*=0,1,…,*q*-1, **T**ij is set to point to the node pointed to by the *j*-edge of δ with the edge negations being the composition of the edge negations on the *i*-th edge from γ and the *j*-th edge from δ.

(ii) If the *i*-edge from γ leads to a node δ not labelled β, then **T**ij is set to the *i*-edge from γ for *j*=0,1,…,*q*-1.

Once **T** is constructed as above, the level interchange is affected by relabelling γ with β, and setting each *j*-edge from γ, *j*=0,1,…,*q*-1 to point to a node labelled α whose *i*-th edge, *i*=0,1,…,*p*-1, points to the node pointed to by **T**ij. During this construction if β denotes a variable, the edge negation operations are normalised as described above to ensure there is no negation operator on any 0-edge. It is easily confirmed that following this construction, node γ, now labelled β, is the top of a decision diagram representing the same function it did when originally labelled α.

The complete level interchange is accomplished by performing the above for all nodes originally labelled α. The idea of relabelling these nodes is critical as it means that edges leading to them, and the nodes from whence those edges originate, are unaffected by the level interchange. The node must of course be removed from its hashing chain, relabelled, and then rehashed.

The nodes originally labelled β are affected as edges to them are removed. Reference count based garbage collection is used as such a node can not be discarded unless no other nodes higher in the diagram point to it. Finally, no node below the two levels being interchanged is affected except for changing the reference count. The result is that the level interchange is a local operation affecting only the two levels being interchanged and reference counts for some nodes below those levels.

The above technique is a generalisation of the method introduced by Rudell [9]. The use of a matrix makes it convenient to deal with a variable number of edges and particularly for the case where $p \neq q$. Handling the latter case is essential for the approach described in section 6.

5. Variable Reordering

The size of a decision diagram is dependent on the variable ordering. Since finding an optimal variable ordering is in general an intractable problem, heuristic techniques for finding a good variable ordering have been extensively studied. Sifting, introduced by Rudell [9], is a search technique based on the systematic interchange of adjacent variables in the ordering. Given the discussion above on adjacent level interchange, sifting and similar techniques can be directly applied to ROMDDs. Our package implements sifting. Space does not permit a detailed discussion here.

6. Operator Nodes and Logical Operations

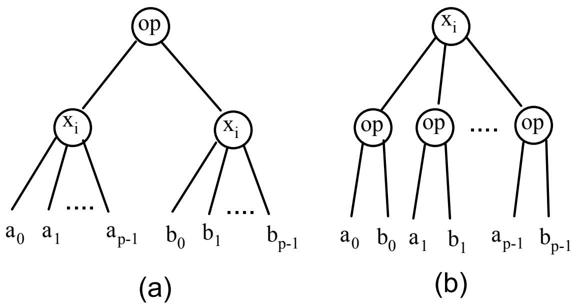
It is important to be able to efficiently apply logical operations to decision diagrams. One approach involves the recursive descent of the diagrams to be combined, but this can lead to a significant amount of recomputation because a pair of subdiagrams can be reached through multiple paths in the diagrams being combined. This is usually solved by using a table to record some number of recent computations. Complex hashing and table lookup techniques are required.

A novel approach recently proposed for ROBDDs in [4] uses operator nodes. It is this method that we here adapt to ROMDDs. The idea is to create an ROMDD whose top node is labelled by the operation to be performed and whose direct descendants are the ROMDDs of the functions to be combined. Level interchanges, as described in section 4 above, are used to move the operator node(s) to the bottom of the diagram. Once an operator node reaches the bottom of the diagram and points only to the terminal node, it can be directly evaluated and the node substituted by the appropriate constant representation.

Consider the case shown in *Fig. 1a*. Interchanging the node labelled *op* with the nodes labelled x_i yields the diagram shown in *Fig. 1b*. The matrix-based level interchange method described above is well suited to this situation because it deals efficiently with the fact that the nodes at the two levels have different numbers of edges.

In the binary case, operator nodes can be normalised so that a negation does not appear on the 0-edge by application of De Morgan's laws. In the MVL case, this can be done for the complement since
$\overline{MIN}(x,y) = MAX(\bar{x},\bar{y})$ and $\overline{MAX}(x,y) = MIN(\bar{x},\bar{y})$.

However, no transformation exists to remove a cycle from the 0-edge of an operator node. The solution is to go ahead with cycles on 0-edges from operator nodes as necessary, but to continue to normalise variable labelled nodes. In this way, the representation will again become unique once all operator nodes are removed by level interchange.


op
x_i
x_i
....
a_0 a_1 a_{p-1} b_0 b_1 b_{p-1}
(a)
x_i
op op .... op
a_0 b_0 a_1 b_1 a_{p-1} b_{p-1}
(b)

Level Exchange Operation for Operator Nodes
Figure 1

A clear optimisation is based on the observation that operators have particular conditions that can be used to shortcut an evaluation. For example, the MIN of 0 and anything is 0 so the level interchange could be aborted and the node labelled MIN replaced by the constant 0. But, this seemingly simple idea presents a major problem. We would in general be replacing a node in the middle of a decision diagram and this would require we identify all edges leading to the node being substituted and update them all. In fact, this problem arises frequently when dealing with operator nodes.

In [4] the solution suggested for this edge update problem is to perform a reduction pass over the decision diagram once the operator nodes have been removed by successive level interchange. Here we present a different approach.

The problem to be solved is to replace a node by an alternate one that in fact already exists and to update all edges leading to the node being replaced. The fact the substitute node already exists means the node to be replaced can not simply be updated in place. Our solution is that when a node must be replaced, we set a *replacement* bit in the node (we borrow a high order bit from the reference count) and set the 0-edge to point to the substitute node. In our package, each time we access an edge, the replacement bit in the target node is checked and if appropriate the edge is updated to point to the replacement node. The reference count is adjusted, and once all edges to the node being substituted are updated, it will pass out of existence through the normal garbage collection process. Note that once a node's replacement bit is set, it is ignored in the node equality check when a new node is being created. This ensures that multiple instances of the same node all map to a single hash chain entry.

Clearly, a node can be marked to be replaced because processing an operator creates a node that has already been created and hence already exists in the hash tables. Not so obvious is that a replacement can be created simply to resolve the normalisation of a node *i.e.* the need to add a cycle value to all edges leading to a node. This latter case is resolved by the same replacement process used to deal with duplicate nodes.

The above approach carries a minor execution cost as the replacement bit must be checked every time an edge is followed. That can be done very quickly and is seen as preferable to a complete reduction pass each time a logical operation is performed on ROMDDs.

The elimination of operator nodes can be made somewhat faster by the observation that our goal is to perform interchanges so that all operator nodes move to the bottom of the graph and then disappear. Hence rather than interchanging an operator node with the level below it, we treat each operator node separately. We determine the 'highest' variable labelling a direct descendant of the operator node and perform the interchange between the operator node and that variable. An interchange creates new operator nodes so we must repeat this process until all operator nodes are eliminated.

The order in which operator nodes are processed is critical. Our procedure for processing an operator node is as follows:

Algorithm 1: *Operator Node Processing*

Let *n* be the operator node to be processed:

1. If *n* is a terminal node return.
2. Remove *n* from the appropriate hash chain.
3. Apply the transformation illustrated in *Fig. 1*. Note that a check is made for simplifications as the new operator nodes are created *e.g.* the MIN of x and 0 is x, whereas the MAX of x and p-1 is p-1, *etc*.
4. Recursively apply this algorithm to the operator nodes created in step (3).
5. Now check each direct descendant of *n* and do the appropriate edge replacement in *n* if the replacement bit is set in the descendant.
6. Rehash *n*, and insert it into the appropriate hash chain. Note that the process of reinserting *n* may set its replacement bit.

For efficiency, the recursion can be removed by maintaining a last-in, first-out queue (stack) of nodes. This is the approach used in our package.

7. Experimental Results

As an example of the use of the techniques described above, we present results on converting cube list specifications to decision diagrams. We use binary benchmark problems in two ways, as given for the BDD case, and converted to 4-valued problems for the MDD case. The conversion of a binary problem to a 4-valued problem is done by taking the inputs in pairs from left to right. If there is an odd number of inputs, the rightmost input remains a binary input. The binary outputs are converted to 4-valued outputs in the same way.

	binary	specification		MDD	BDD	
	in	out	Cubes	size	size	%
9sym	9	1	88	18	25	72.00%
alu2	10	8	70	48	134	35.82%
alu4	14	8	1028	510	1197	42.61%
bw	5	28	25	66	108	61.11%
duke2	22	29	87	711	973	73.07%
mdiv7	8	10	256	78	183	42.62%
misex1	8	7	32	39	41	95.12%
misex2	25	18	29	128	136	94.12%
misex3	14	14	1848	351	1301	26.98%
postal	8	1	256	11	25	44.00%
rd53	5	3	32	14	17	82.35%
rd73	7	3	141	20	31	64.52%
rd84	8	4	256	22	42	52.38%
sao2	10	4	58	60	155	38.71%
74181	14	8	1133	431	858	50.23%
vg2	25	8	110	685	1044	65.61%
Total				**3192**	**6270**	**50.91%**

Table I: *Comparison of ROMDDs and ROBDDs*

In both the binary and the multiple-valued case, the cubes are converted to decision diagrams that are in turn combined to form the decision diagrams for the outputs. The decision diagram for the input side of a cube has a well-defined structure that can be determined directly. To combine a cube with an output, we use a level exchange based logic operation, OR in the binary case and MAX in the multiple-valued case. For the multiple-valued case, a MIN must first be applied to the cube to account for the value the cube is to take for the particular output.

Experimental results for a number of binary benchmarks are shown in Table 1. These were run on a Sun-690 dual 166 MHz processor system with 128MB of memory. The gnu C-compiler was used with level 4 optimisation.

BDD size is the number of nodes in the shared reduced ordered binary decision diagram build from the binary cube list. *MDD size* is the number of nodes in the shared ROMDD built from the 4-valued problem derived from the binary problem as described above. Variable reordering is not used, *i.e.* the variables are taken in the order given with the leftmost variable at the bottom of the decision diagram and the rightmost variable at the top.

As one would expect the total number of nodes in the 4-valued case is about 50% of the binary total. It is somewhat surprising however how much variation there is between examples, from 27% to 94%.

Table II shows the effect of using cycles for the 4-valued versions of the benchmarks. The overall saving is 4.25%. Once again, the reduction varies across the examples, from 0% to 12%.

	size cycles	size	%	max nodes	time (a)	time (b)
9sym	18	18	0.00%	81	110	140
alu2	48	53	9.43%	78	50	80
alu4	510	537	5.03%	639	7191	9671
bw	66	73	9.59%	98	70	70
duke2	711	726	2.07%	832	310	4940
mdiv7	78	79	1.27%	120	420	680
misex1	39	39	0.00%	53	10	20
misex2	128	128	0.00%	148	20	100
misex3	351	366	4.10%	435	3620	12652
postal	11	11	0.00%	40	10	10
rd53	14	15	6.67%	27	10	20
rd73	20	21	4.76%	39	70	110
rd84	22	25	12.00%	58	180	250
sao2	42	65	6.67%	98	50	50
74181	431	469	8.10%	732	4340	7541
vg2	685	690	0.72%	1023	600	23494
Total	**3174**	**3315**	**4.25%**		**17061**	**59828**

Table II: Effect of cycles and evaluation strategy.
(a) Time for level exchange (msec.)
(b) Time for recursive descent (msec.)

Table II also shows the execution time (a) for the level exchange method described in section 6, and (b) for the same program with the traversal described in algorithm 1 replaced with a depth-first traversal of the operator nodes. The latter is equivalent to a recursive descent implementation of logic operations without the use of a recent computation table.

The advantage of the level interchange method is clear. In total, over all the examples, it takes 28.5% of the time. Simple stack management of the operator nodes thus avoids the need for a more complex approach such as a recent computation table.

For the larger problems, the MDD package is typically two or three times slower than an optimised BDD package, and can be as much as eight times slower *e.g.* for example alu4. The BDD package we used is highly optimised and in particular makes heavy use of the fact there are only two edges from each node. The edge count flexibility in the MDD package implemented via the array **edge** in each node comes at an execution penalty. An alternative would be to use conditional compilation to customise the package to different numbers of logic values. This requires further study. There is also a higher cost associated for the handling of cycles than there is for the handling of edge negation in the binary case.

8. Concluding Remarks

This paper has considered issues that arise in implementing an MDD package. A matrix approach for adjacent level interchange has been presented and has been shown to be effective in implementing logical operations on ROMDDs. It is also the basis for variable reordering using a technique such as sifting.

We are continuing to optimise the implementation. Profiling has shown that looking-up nodes in the hash chains is the most time-consuming operation. We are considering how to improve the existing approach and we are also considering alternative hashing techniques. Experiments are required on larger functions to assess the true efficiency of the proposed methods.

One area for further development is to extend the package to other types of multiple-valued decision diagrams.

The techniques described in this paper have been incorporated in a ROMDD package written in C. This package is available at www.csr.uvic.ca/~mmiller/MDD.

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] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

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

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