Representation of Boolean Functions with ROBDDs

Aleš Časar, *Student Member, IEEE*, and Robert Meolic, *Student Member, IEEE*
Faculty of Technical Sciences, University of Maribor, Slovenia

Abstract— **This paper describes data structures and algorithms for the representation of Boolean functions with reduced ordered binary decision diagrams (ROBDDs). A hash table is used for quick search. Additional information about variables and functions is stored in binary trees. Manipulations on functions are based on a recursive algorithm of ITE operation. The primary goal of this article is to describe programming technics needed to realize the idea. For the first time here recursive algorithms for composing functions and garbage collection with a formulae counter are presented. This is better than garbage collection in other known implementations. The results of the tests show that the described representation is very efficient in applications which operate with Boolean functions.**

1 Introduction

Boolean functions are a frequent form of data in computer science. Efficiency of algorithms for operations involving Boolean functions depends on data structure used for the representation. Smaller problems can be solved by hand using truth table, Karnaugh's diagram, Veitch's diagram or disjunctive or conjunctive normal form. But these representations are not suitable for problem solving by computer because space and time grow exponentially with the number of variables for the majority of common functions.

The most common problems are equivalence testing and tautology checking. It is easy to solve both problems if the representation of Boolean functions constitutes a canonical form. This means that every function has only one representation and two different functions have two different representations.

Binary decision diagram (BDD) is a very good data structure for the representation of Boolean functions in computer. Space and time complexity are exponential in the worst case, but in most cases they are very reduced. In a BDD, a function is represented recursively with triples $(x_i, f|_{x_i=1}, f|_{x_i=0})$, which are a consequence of Shannon's decomposition theorem :

$$f = x_i \cdot f|_{x_i=1} + \overline{x_i} \cdot f|_{x_i=0} \qquad (1)$$

Algorithms for creating BDDs and algorithms for manipulating functions represented by BDDs have advanced considerably from the beginning studies [1]. The shortcomming of BDDs not being a canonical form for the representation of Boolean functions was removed with an upgrade to reduced ordered binary decision diagrams (ROBDDs) [2]. Algorithms for operations with ROBDDs are recursive and they are derived from Shannon's decomposition theorem (1).

Time and space complexity of ROBDDs depend on variable ordering. A simple reordering of variables alone may result in the reduction of the size of the diagram or vice versa. Determining the optimal variable ordering is unfortunately a NP problem. Usually, simple reordering is good enough. This paper, however, is not concerned with this kind of problems.

2 Definitions

A number of terms whose meaning has already been defined in [3] are used in the paper.

A *hash table* is a data structure in which a lot of data are stored and can be found or checked very quickly [4, 5].

A *hash table with chaining* is an array of lists. Every element has an exactly determined list of which it makes a part. The list is computed from the element by a *hash function*. In searching for an element the list number should be computed and the corresponding list for the element checked. We term the hash table with chaining simply *hash table*.

A *hash-based cache* is an array. Every element has an exactly determined field in the array. The field is also computed from an element by a *hash function*. A hash function maps more elements into the same field. Only one element is stored, others are lost. In searching for an element, we should first check if an element exists in the field. Afterwards it should be checked if it is the rigth one.

A *binary decision diagram* (BDD) is a directed, acyclic graph. It includes sink nodes '0' and '1', which represent constant Boolean functions 0 and 1. These nodes have no descendants. All other nodes include a variable and two edges to the descendants—subgraphs. The edges are labeled by 'then' and 'else'. Every path through the graph finishes in one of the sink nodes. The result of the function, represented by BDD, is computed simply by travelling through the graph. In every node corresponding subgraph is chosen. It depends on the truth value of the variable. The value of the function is determined by the sink node that ends the path.

An *ordered binary decision diagram* (OBDD) is a BDD where all variables are ordered on previously known ordering and every path visits variables in an ascending order.

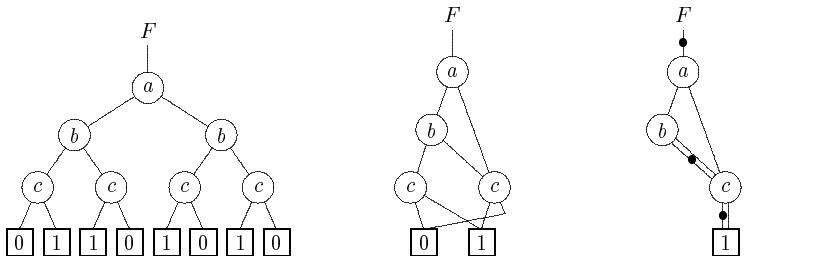
A *reduced ordered binary decision diagram* (ROBDD) is an OBDD where any two nodes differ from each other. Functions share ROBDD. So every node can belong to more functions. Therefore it can be also termed a *shared binary decision diagram* (SBDD) [6].

A possible extension of ROBDD is the introduction of *complement edges*. Every edge has an additional field (a single bit is sufficient) which tells us whether it is a regular or a complement edge. Changing the edge from regular to complement and vice versa is called *complementing the edge*. A complement edge complements the function that follows. There is no need to keep the sink node '0' in the graph. It is replaced by the complement edge to the node '1'. The value of the function is computed by travelling through the graph down to the sink node and counting complement edges. An odd number of complement edges means that the result is 0 and an even number means that it is 1. To maintain a canonical form, the 'then' edge must be regular in every node. We name this kind of node a *regular node*.

Three representations of the same function are shown in Figure 1.[1]

The *If-Then-Else* or ITE is a three-variable Boolean opera-

[1] In figures the 'then' edge leaves a node on the bottom-right and 'else' edge on the bottom-left side. A bullet denotes the complement edge.



a) OBDD b) ROBDD c) ROBDD with complement edges

Figure 1: Three representations of the function $a \cdot \overline{c} + \overline{a} \cdot \overline{b} \cdot c + b \cdot \overline{c}$

tor, defined as :

$$\mathsf{ITE}(f,g,h) = f \cdot g + \overline{f} \cdot h \quad . \tag{2}$$

With ITE, all two-variable Boolean operations can be implemented (see Table 1). Shannon's decomposition theorem (1) can be shortly written as $f = \mathsf{ITE}(x_i, f|_{x_i=1}, f|_{x_i=0})$, and therefore it forms a basic operation in BDD.

3 Implementation

3.1 Notation

Variables are denoted by lowercase letters and functions by uppercase ones.

Every edge in the ROBDD represents one Boolean function. The edge which represents function F is named an *input edge* of function F. When we operate with function F, we actually operate with its input edge. Capital letters, which represent functions in formulae and algorithms, in fact also denote input edges of corresponding functions. Every input edge leads to a *top node* of the function. A variable in top node is named a *top variable* of the function. The top variable of a set of functions is the smallest of the top variables of those functions.

Every node is a top node of functions F and of $\overline{F}$ with the regular and the complement input edge, respectively.

Each node can be denoted by the triple (v, G, H)—the node is fully decribed by variable v and functions G and H for its 'then' and 'else' descendants, respectively. Expression $F = (v, G, H)$ means that the node (v, G, H) is the top node of the function F.

A function known to the user is named a *formula*. Formulae are denoted by capital letters like all other functions. All nodes included in a formula are called *internal nodes*.

3.2 Basic Data Structures

3.2.1 Unique-Table

Before adding a new node (v, G, H) into the ROBDD, we have to check if such a node already exists. If it exists, it is used instead of creating a new one. In this way functions share the same ROBDD (different functions include the same node or subgraph). This characteristic reduces the memory space needed. For checking the existence of a node, we need a data structure in which nodes can be accessed quickly. We use a hash table with chaining and name it the *unique-table*. A sufficiently efficient hash function is simply the sum of the parameters (pointers) modulo table size.

3.2.2 Computed-Table

Efficiency of *ite* function can be increased by the introduction of the *computed-table*. Computed-table is a data structure which remembers parameters F, G, H and the result $ite(F, G, H)$ at every call of *ite*. At the next call of *ite* with the same parameters we simply read the result without any computing. Searching for results in the table must be efficient because they are searched at every call of *ite*. It is not necessary to keep all the results in the table. Therefore it is better to avoid a dynamic data structure, although in this case it may happen that we discard a result and have to compute it later on again. So we use hash-based cache. An efficient hash function is again simply the sum of the parameters modulo table size.

3.2.3 Symbol Tree

If we want to evaluate the function, we need a data structure which includes the variable name and its Boolean value. Because one variable appears several times in one graph, a separate data structure is the best solution. Instead of a variable in each node there is a pointer to the field in this data structure, where the name and the value of the variable are actually stored. The realization without pointers uses indices instead. This is a very good solution, because in the separate structure supplementary information can be stored. This data structure is named the *symbol tree* because a binary tree is used for the realization.

A node with variable v and descendants 0 and 1 is needed very often. This node is named *basic node of the variable* v. Edges to basic nodes are also stored in the symbol tree, so that it is not necessary to look for them again every time.

3.2.4 Formulae Tree

The manipulation of several formulae at the same time calls for a data structure where the formula name and its input edge will be stored. For this purpose we also use a binary tree and we name it a *formulae tree*.

3.3 Normalization

Each triple (F, G, H) belongs to one class of triples. For arbitrary triples (F_i, G_i, H_i) and (F_j, G_j, H_j) in a class either $ite(F_i, G_i, H_i) = ite(F_j, G_j, H_j)$ or $ite(F_i, G_i, H_i) = \overline{ite(F_j, G_j, H_j)}$ is valid. So all triples in a class are equivalent with regard to *ite*. It is enough if only one triple from each class is stored. We have named this triple a *standard triple*. Before searching in computed table, the corresponding standard triple must be formed. We have named this procedure the *normalization*. Note that normalization

effectively prevents unnecessary computing owing to commutativity and DeMorgan's laws.

It has been proved that in every class there exists one triple which has functions with regular input edges for arguments F and G. If this triple is used as a standard triple, then only one mark must be stored in the computed-table— whether input edge of H is regular or not (see data structure in appendix A). The algorithm for this kind of normalization is resumed from [3].

- First, if it is possible, input arguments are simplified :

$$\begin{aligned} ite(F,F,G) &\Longrightarrow ite(F,1,G) \\ ite(F,G,F) &\Longrightarrow ite(F,G,0) \\ ite(F,G,\overline{F}) &\Longrightarrow ite(F,G,1) \\ ite(F,\overline{F},G) &\Longrightarrow ite(F,0,G). \end{aligned}$$

- Between the triples which form one of the following patterns, one from the corresponding pair should be chosen :

$$\begin{aligned} ite(F,1,G) &= ite(G,1,F) \\ ite(F,G,0) &= ite(G,F,0) \\ ite(F,G,1) &= ite(\overline{G},\overline{F},1) \\ ite(F,0,G) &= ite(\overline{G},0,\overline{F}) \\ ite(F,G,\overline{G}) &= ite(G,F,\overline{F}). \end{aligned}$$

  The triple which has a function with the smallest top variable for the first argument is selected. Note that addresses of nodes must be compared as these are different for every single node. If edges are pointers then the address is simply the value of the input edge.

- Finally, one of the following four forms is chosen :

$$ite(F,G,H) = ite(\overline{F},H,G) = \overline{ite(F,\overline{G},\overline{H})} = \overline{ite(\overline{F},\overline{H},\overline{G})} \quad .$$

  A triple is chosen according to the rule that the first and the second argument should be functions with regular input edge. The algorithm for this part of normalization is presented as Algorithm 1. Sometimes we get complemented results from the computed-table. This happens if variable *result* gets value $FALSE$.

```
result := TRUE;
if F.complement then begin
   F.complement := FALSE;
   if H.complement then begin
      result := FALSE;
      swap and complement edges G and H
      end
   else swap edges G and H
   end
else if G.complement then begin
   G.complement := FALSE;
   result := FALSE;
   complement edge H
end;
```

Algorithm 1: Third step of normalization

3.4 Algorithm of *ite* Function

Building of a ROBDD is done with the *ite* function. It is presented as Algorithm 2. Let v be the top variable of functions F, G and H. Let F_v and $F_{\overline{v}}$ denote functions $F|_{v=1}$ and $F|_{v=0}$, respectively. Then the following recursive formula can be derived[2]:

$$ite(F,G,H) = ite(v, ite(F_v,G_v,H_v), ite(F_{\overline{v}},G_{\overline{v}},H_{\overline{v}})) \quad . \qquad (3)$$

Recursion proceeds until one of terminal calls occurs :

$$ite(F,1,0) = ite(1,F,G) = ite(0,G,F) = ite(G,F,F) = F; \quad ite(F,0,1) = \overline{F} \quad .$$

```
function ite(F,G,H: EdgeType): EdgeType;
  begin
    execute the first step of normalization;
    if terminal case then return result
    else begin
      execute the second and the third step of normalization;
      if result is in the computed-table then return result
      else begin
        let v be the top variable of functions F, G and H;
        T := ite(F_v, G_v, H_v);
        E := ite(F_v̄, G_v̄, H_v̄);
        if T = E then ite := T;
        R :=find_or_add_node(v,T,E);
        insert_in_computed-table(F,G,H,R);
        ite := R
      end
    end
  end;
```

Algorithm 2: Recursive algorithm of *ite* function

Function *find_or_add_node* creates new nodes. It takes a node (v,T,E) for an argument and returns the edge pointing to that node as a result. If and only if the needed node does not exist, the function creates a new one. In a ROBDD with complemented edges, only the nodes which have regular 'then' edge are valid (see Section 2). Function *find_or_add_node* takes care of this itself. If an invalid node is claimed, then the request is transformed into the equivalent, valid form as shown in Figure 2. During the transformation the edge leading to the node is complemented.

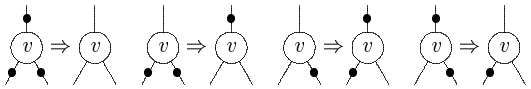


Figure 2: Transformation into valid form

4 Operations on ROBDD

4.1 Boolean Operations

All two-variable operations can be realized by the ITE operator, as shown in Table 1. A function can be complemented in a simplier way by complementing its input edge.

[2]See derivation in appendix B.1.

Table	Name	Expression	Equivalent form
0000	0	0	0
0001	f **and** g	$f \cdot g$	ITE$(f, g, 0)$
0010	$f > g$	$f \cdot \overline{g}$	ITE$(f, \overline{g}, 0)$
0011	f	f	f
0100	$f < g$	$\overline{f} \cdot g$	ITE$(f, 0, g)$
0101	g	g	g
0110	f **xor** g	$f \oplus g$	ITE$(f, \overline{g}, g)$
0111	f **or** g	$f + g$	ITE$(f, 1, g)$
1000	f **nor** g	$\overline{f + g}$	ITE$(f, 0, \overline{g})$
1001	f **xnor** g	$\overline{f \oplus g}$	ITE$(f, g, \overline{g})$
1010	**not** g	$\overline{g}$	ITE$(g, 0, 1)$
1011	$f \geq g$	$f + \overline{g}$	ITE$(f, 1, \overline{g})$
1100	**not** f	$\overline{f}$	ITE$(f, 0, 1)$
1101	$f \leq g$	$\overline{f} + g$	ITE$(f, g, 1)$
1110	f **nand** g	$\overline{f \cdot g}$	ITE$(f, \overline{g}, 1)$
1111	1	1	1

Table 1: All two-variable functions realized with ITE

4.2 Composition of Functions and Evaluation of Restricted Function

A composition of functions ($f|_{x_i=g}$) is an operation that assigns a multi-variable function to a variable.

Let v be a top variable of function f. Let f_v and $f_{\overline{v}}$ denote functions $f|_{v=1}$ and $f|_{v=0}$, respectively. The following recursive formula shows how to compute the composition[3] :

$$f|_{x_i=g} = \begin{cases} f & ; \ v > x_i \\ \text{ITE}(g, f_v, f_{\overline{v}}) & ; \ v = x_i \\ \text{ITE}(v, f_v|_{x_i=g}, f_{\overline{v}}|_{x_i=g}) & ; \ v < x_i \end{cases} \quad (4)$$

A special case of composition is restriction : $f|_{v=0}$ or $f|_{v=1}$. The algorithm for restriction is very similar to that for composition. In practice, one algorithm (for composition) is good enough for both operations.

```
function Compose(f: EdgeType; x: WordType; g: EdgeType):
    EdgeType;
  begin
    if f is a constant function then Compose := f
    else begin
      let v be the top variable of function f;
      if v > x then Compose := f
      else
        if v = x then
          Compose := ite(g, f_v, f_v̄)
        else
          Compose := ite(v, Compose(f_v, x, g), Compose(f_v̄, x, g))
      end
    end;
```

Algorithm 3: Algorithm for composition of functions

5 Garbage Collection

Computers have limited memory and they run out of it quickly. To avoid this problem, nodes that are unnecessary should be removed. We named the procedure which performs this function garbage collection (GC).

Even if memory is not completely full, GC is very useful. A smaller number of nodes means faster operations on BDDs. However, some extra time is spent on GC. For an effective GC additional information about each node is needed. Unfortunately, this increases the usage of memory. Therefore, a more frequent GC, which is faster, is preferable.

GC deletes all nodes which are not part of any formula. If the user wants, a complete formula can be removed, too. Internal nodes of other formulae must not be deleted by this operation.

Note that all records in the computed-table which contain bad node must be removed before deleting bad nodes from the unique-table.

5.1 Garbage Collection with a Reference Counter

This algorithm was presented in [3]. Each node has a count of the number of other nodes and the number of user formulae that reffer to it. A node with a reference count of 0 is called *dead node*.

When a formula is deleted, the reference count of the corresponding top node is decremented by 1. If the new reference count is 0, then the reference counts of successor nodes are recursively decremented. If any of them becomes a dead node, recursion continues there.

It can happen that an existing node which is already dead should be included into the formula. In this case the reference count of this node and of all dead descendants are incremented by 1. In recursion it should be considered that all successors of a non-dead node are non-dead nodes.

The procedure for GC removes all dead nodes.
An advantage :

- Less additional information is needed.

Drawbacks :

- GC can not be activated while the formula is being created.
- The reference counter can not distinguish between a reference from an internal node of a formula and from a dead predecessor node. Therefore, GC removes only a portion of nodes which are not internal nodes of any formula.

5.2 Garbage Collection with a Formulae Counter

GC is the most effective if all unnecessary nodes are removed. These are above all the nodes which are not internal nodes of any formula. As GC with a reference counter does not enable us to do this, we propose a different approach.

There are three kinds of nodes in the ROBDD when GC is started. Those that do not belong to any formula may be deleted. We named them *bad nodes*. The nodes that belong to one or more formulae must remain. We named them *fortified nodes*. The third kind of nodes are those that could be possibly included into the formula which is currently being built. These are *fresh nodes* and GC may not remove them.

Formulae are numbered by the *formulae counter* as 1, 2, 3, etc. Every node has a field named *mark*. It is an unsigned number and tells us the kind of a node. Fortified nodes have mark 0. Fresh nodes have mark that is equal to the current value of the formulae counter. Any other mark means bad node. It always holds that all descendants of fortified nodes are also fortified and all descendants of a fresh node can not be bad.

[3] See derivation in appendix B.2.

We named the recursive procedure which sets mark in each internal node of the formula to 0 *formula fortifying*. A similar procedure which changes a bad node and all its bad descendants to fresh ones is named *node refreshing*.

When a new node is created, it gets mark which is equal to the current value of the formula counter. Fortified and fresh nodes which are included into the formula remain unchanged. But bad nodes which are included must be refreshed. When the whole formula is built, it has to be fortified.

To avoid checking all nodes in every call for GC we create a *list of new nodes*. Whenever a new node is created, it is included into this list. Nodes must be added to the end. The list requires a supplement field in every node.

GC examines the list of new nodes. All bad nodes are deleted, all fortified nodes are removed only from the list, and all fresh nodes remain unchanged both in the unique-table and in the list. It is not even necessary to look over the whole list, as from an element forward to the end the list contains only fresh nodes. This element is marked by a pointer. Whenever the formulae count is incremented, this pointer is transfered to the end of list.

Note that all lists in the unique-table must be linked in both directions, because lists linked only in one direction do not enable deleting without knowing a predecessor.

There is an extra procedure for removing complete formulae. First, all nodes are changed to bad ones. All formulae which remain are then fortified. And at the end, all bad nodes are deleted. It is better to remove more formulae at once. As this operation would at the same time remove all fresh nodes, it is impossible to perform it during the creation of a formula. However, this is not a problem, because deleting during creating is never used.

A list of new nodes effectively detects unnecessary calls of GC (two calls successively, two calls during creation of the same formula).

A *list of free nodes* is used to improve GC. It reduces the number of memory reservation requirements that would appear by the creation of each new node. When a node is removed, it is put into this list and not deleted. When a new node is to be created, we do not allocate a new memory space for it, but simply use an existent spare place in the list of free nodes if it is not empty. Note that this improvement does not require any additional information in a node.

Advantages :

- GC is fast.
- GC does not disturb other operations. We can start it at any time and after its termination continue the interrupted operation.
- GC removes all unnecessary nodes.

Drawbacks :

- More supplementary information about nodes is required.
- Because of keeping the list of new nodes, creation of new nodes is more time consuming.

6 Experimental Results

6.1 Equivalence Testing

Boolean functions represented with ROBDDs are useful in many computer applications. Our software is applied for testing equivalence of digital circuits. This application fits in the previous research in the domain of formal verification [7, 8]. We have two circuits with the same inputs and equal number of outputs. Input variables are denoted by $I_1, I_2, \ldots, I_n$, outputs from the first circuit by $O_{11}, O_{12}, \ldots, O_{1m}$ and from the second one by $O_{21}, O_{22}, \ldots, O_{2m}$. The problem is stated as follows: given all output functions in the form

$$O_{ki} = F_{ki}(I_1, I_2, \ldots, I_n), \quad k = 1, 2;\ i = 1, 2, \ldots, m \ ,$$

determine if outputs from both circuits are equal or, more formally, if

$$O_{1i} = O_{2i}, \quad i = 1, 2, \ldots, m \ .$$

Equivalence testing can be limited to a set of input combinations. In that case there is also a function which denotes when the equivalence is not important (*don't care set*—*dcs*). If this function is denoted by $D(I_1, I_2, \ldots, I_n)$, then the equivalence is relevant only for those input combinations, where $D = 0$.

Digital circuits are described in .BE format [9] in files, which have been used on *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design* in Houthalen, Belgium, November 1989. Every file contains a pair of circuits which have to be compared. A reader will find more information about these files in [9].

6.2 Measuring

Equivalence testing has to be finished as soon as possible and with a sufficienly small usage of memory. Our program is written in Pascal on VAX 4000–600 (see technical data in 6.3). Measurements are carried out on benchmark tests described in [9]. Results are shown in Table 2.

The CPU time includes time spent for reading and checking input file, forming the ROBDD and equivalence testing for all outputs.

6.2.1 What Have We Measured ?

First we have performed measurements on original files without any previous changes. We have used alphabetical variable ordering. These results are in the third column in Table 2. Then we have renamed variables using the principle "the most often used variable gets the smallest name". New files were created for the new ordering, and this is not comprised in the presented times. For some circuits (typical is *add4*) reordering significantly decreases the CPU time. On the contrary, for some other circuits (e.g. *mul08*) testing takes a little bit more time. Of course our variable ordering is not optimal. It is only a heuristic approximation of the optimal one. We have not carried out a detailed research for optimal ordering. As finding an optimal order is also a NP problem, it is questionable if it is worth doing it at all.

6.2.2 Parameters by Realization of the ROBDD

We have tested many combinations of parameters. We have been interested in :

- the size of the unique-table (is it better to have a large or a small one);
- the size of the computed-table (is it better to have a large or a small one);
- the best time to start GC :
  - when the number of bad nodes exceeds a certain constant (if it is small, GC will be executed frequently, if it is big it will be executed rarely),

Results of Measuring					
circuit name	number of variables	alphabetical order time (s)	param.	reordered time (s)	param.
add1	9	0.06	uCg+2	0.05	uCG+3
add2	13	0.24	ucG+2	0.20	ucg+3
add3	21	2.44	ucg+3	0.32	ucg+3
add4	29	298.60	UCg*3	3.63	UCg+3
addsub	29	.....	.....	0.57	uCg+3
alu	11	0.22	uCg+2	0.15	Ucg+1
ex2	4	0.00	Ucg*1	0.00	ucg+2
mul03	6	0.03	Ucg+1	0.00	UCg*2
mul04	8	0.12	ucG+1	0.09	ucg*3
mul05	10	0.35	ucg+3	0.30	ucg+1
mul06	12	1.23	ucg+3	1.13	ucg+2
mul07	14	5.03	ucg*3	4.17	uCg+3
mul08	16	18.97	uCg+1	16.45	uCg+3
rip02	4	0.01	UCg+3	0.00	UCG+1
rip04	8	0.03	UcG+2	0.03	ucG+1
rip06	12	0.13	ucG+3	0.06	UCG+3
rip08	16	0.69	ucg+3	0.10	ucG+3
transp	4	0.00	uCg*1	0.00	Ucg*3
ztwaalf1	12	0.03	ucG+3	0.04	ucg+1
ztwaalf2	12	0.04	ucG+1	0.04	Ucg+1
alupla20	19	0.11	ucg+1	0.08	uCG+3
alupla21	22	0.19	ucG+3	0.15	ucg+2
alupla22	25	0.46	ucg+3	0.52	ucg+3
alupla23	25	0.76	ucg+3	0.72	ucg+3
alupla24	20	0.24	ucG+1	0.24	ucg+3
dc2	8	0.06	uCG+3	0.06	ucg+3
dk17	9	0.07	UCG+1	0.05	Ucg*3
dk27	10	0.03	uCg*1	0.02	UCg*2
f51m	8	0.06	uCG+2	0.09	ucg+1
misg	56	0.07	uCg+2	0.07	ucG+1
mlp4	8	0.25	ucg*3	0.23	ucg+3
rd73	7	0.09	UcG+3	0.09	ucG+1
risc	8	0.04	UCg+3	0.04	ucg+1
root	8	0.09	UCg+3	0.08	ucg+3
sqn	7	0.07	ucG+3	0.07	ucg*2
vg2	25	0.10	Ucg+3	0.08	ucg*3
x1dn	27	0.12	ucG+2	0.08	ucg+3
x6dn	39	0.58	ucg+3	0.23	ucg+1
z4	7	0.03	Ucg+3	0.03	uCg+1
z5xpl	7	0.07	uCg*3	0.08	Ucg+1
z9sym	9	0.18	ucG+3	0.16	ucg+3
counter	6	0.04	UCg+3	0.05	uCg+1
d3	7	0.88	uCG+1	1.12	ucg+1
hostint1	5	0.03	uCg+3	0.04	uCg+1
in1	15	10.39	ucg+3	8.22	Ucg+3
mp2d	14	0.29	Ucg+3	0.39	ucG+1
mul	7	0.06	uCg+2	0.06	ucg+1
pitch	16	0.97	ucg+1	0.92	ucG+1
rom2	13	1.00	ucg+3	0.40	ucg+1
table	17	0.71	ucg+1	0.56	ucg+2
werner	6	0.01	UCg+3	0.01	uCg+1

Table 2: Results of tautology checking of logic circuits

Legend :

U/u ... unique-table is large/small
C/c ... computed-table is large/small
G/g ... garbage collection is frequent/seldom
+ ... garbage collection is executed when *number* of bad nodes exceeds a given value
***** ... garbage collection is executed when *portion* of bad nodes exceeds a given value
1 ... no elements are overlaid in computed-table
2 ... only bad elements are overlaid in computed-table
3 ... all elements are overlaid in computed-table

  - when a portion of bad nodes exceeds a certain constant (if it is near to 1, GC will be executed very rarely, and very often if it is near 0);

- whether elements in the computed-table can overwrite older ones (never, only bad elements can be overwritten, always).

It is difficult to say which combination is the best, because one is good for some circuits and bad for others. This is evident from the Table 2. However, good combinations have some common characteristics. We propose the following strategy.

Unique-table should be large. Owing to the use of a list of new nodes it is hardly ever completely examined. Computed-table should be small; especially at frequent GC it is not good to have very large computed-tables. GC must start when the number of bad nodes exceeds a certain constant. In our tests, activating GC when a portion of bad nodes exceeded a certain constant did not turn out well. It remains an open question, when elements in computed-table overwrite other. The strategy that always overwrites the elements was a bit better then other two strategies, perhaps because no additional computing was needed.

6.3 Technical Data

Complete program package and complete measuring have been carried out with the following equipment :

- 32 bit computer VAX 4000–600, approximately 30 times faster than VAX 750 ;
- 4 GB address space ;
- physical memory : 128 MB ;
- virtual memory : 525 MB ;
- operating system : VAX/VMS V5.5–2 ;
- programming language : VAX PASCAL V4.4 .

It is necessary to say that all measurings have been done in "multi user" mode. Therefore computer has simultaneously executed other processes. At times there have been more than 100 of them. This important difference has to be considered when our results are compared with [6, 10]. Our process has been allowed to use at most 50 MB out of 128 MB of physical memory and the whole virtual memory has been shared with other active processes.

7 Conclusions

Best solutions on as to how to represent Boolean functions with ROBDD are shown. Our representation uses hash table, If-Then-Else operator and supplementary list for garbage collection. Data structures are described in detail and their declarations in Pascal are given in Appendix A. The algorithms used are recursive and based on Shannon's decomposition theorem (1). Because the same theorem is the basic idea of the ROBDD, these algorithms are the most natural and also the most efficient.

A practical result of our research is a ROBDD programming package written in Pascal. For each node in the unique-table we need 36 bytes and for each record in the computed-table 28 bytes. Due to a compiler, that we have, the type "boolean" spent the whole 32 bits (1 longword) instead of only 1, that would be otherwise needed. We have written some different parsers for various forms of files (prefix and infix form of functions). Our practical results confirm that ROBDD is the best data structure for such

problems. Results are similar to those by other authors [10, 6] and in most cases even better.

Finally we suggest possible improvements. The meaning of marks by complemented edges can be changed. In [6] the usage of input inverters is described. By this method a complement edge means complementing variable in the node (it is different from complementing the whole function that follows). Input inverters have similar characteristics as our marks. It depends on a function whether input inverters or complement edges are better.

Instead of one, there could be more sink nodes. Each of them would contain a small truth table. They would represent simple functions of few variables. Therefore it would be not necessary to create these functions and we would save time. In this case truth tables would also need less space than equivalent ROBDDs. Truth tables would be used like any other node.

The presented results show that time and space complexity depend heavily on variable ordering. Therefore greater improvements can be achieved with algorithms for optimal ordering of input variables.

A Data Structures in Pascal

```
type UniqueTablePtr   = ^UniqueTableType;
     SymbolTreePtr    = ^SymbolTreeType;
     FormulaeTreePtr  = ^FormulaeTreeType;

     EdgeType = record                              {complement edge}
       pointer    : UniqueTablePtr;
       complement : boolean
     end;

     UniqueTableArray = array [0..size] of UniqueTablePtr;

     UniqueTableType = record
       pred, succ : UniqueTablePtr;
                                  {linked list in both directions}
       variable   : SymbolTreePtr;
                    {information about variable in the node}
       e, t       : EdgeType;
                    {'else' and 'then' edge from the node}
       mark       : integer;
                    {mark is used by garbage collection}
       list       : UniqueTablePtr          {list of new nodes}
     end;


     ComputedTableType = record
       F, G, H : UniqueTablePtr;  {input parameters to ite}
       Hmark   : boolean;              {mark of parameter H}
       result  : EdgeType                    {result of ite}
     end;

     ComputedTableArray = array [0..size] of
       ComputedTableType;

     SymbolTreeType = record
       left, right : SymbolTreeType;
                                  {descendants in binary tree}
       name        : EdgeType;                {variable name}
       value       : boolean;          {logic value of variable}
       basic       : EdgeType          {basic edge of variable}
     end;

     FormulaeTreeType = record
       left, right : FormulaeTreePtr;
                                  {descendants in binary tree}
       name        : WordType;                {formula name}
       input       : EdgeType                   {input edge}
     end;
```

B Derivations of some Formulae

B.1 Recursive Formula of ITE Operation

There are functions F, G, H and we want to compute $T = \mathsf{ITE}(F, G, H)$. Let v be the top variable of F, G and H. We can write

$$T = v \cdot T_v + \overline{v} \cdot T_{\overline{v}} \ .$$

Considering formula (2), T can be replaced by

$$T = v \cdot (F \cdot G + \overline{F} \cdot H)_v + \overline{v} \cdot (F \cdot G + \overline{F} \cdot H)_{\overline{v}} \ .$$

Because distributivity law is valid for computing of restricted function, we can continue with

$$T = v \cdot (F_v \cdot G_v + \overline{F}_v \cdot H_v) + \overline{v} \cdot (F_{\overline{v}} \cdot G_{\overline{v}} + \overline{F_{\overline{v}}} \cdot H_{\overline{v}}) \ .$$

Considering formula (2) again, but in the opposite direction, we get

$$T = v \cdot \mathsf{ITE}(F_v, G_v, H_v) + \overline{v} \cdot \mathsf{ITE}(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}}) \ .$$

After applying the same formula to the entire expression we obtain the final result

$$T = \mathsf{ITE}(v, \mathsf{ITE}(F_v, G_v, H_v), \mathsf{ITE}(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \ .$$

We can summarize that

$$\mathsf{ITE}(F, G, H) = \mathsf{ITE}(v, \mathsf{ITE}(F_v, G_v, H_v), \mathsf{ITE}(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \ ,$$

where v is the top variable of formulae F, G and H. Therewith, formula (3) is derived.[4] ■

B.2 Recursive Formula of *Compose* Function

There are functions F and G and a variable x. We are interested in $F|_{x=G}$. Let v be the top variable of function F. We have to consider three cases :

$x < v$: Because each function is represented with the ROBDD, where all variables are ordered (see section 2), variable x does not exist in function F at all. Therefore, function F does not depend on x and the result is F, accordingly. We can write

$$F|_{x=G} = F \ .$$

$x = v$: The result is now either the right (F_v) or the left ($F_{\overline{v}}$) branch of node v, which depends on function G. So

$$F|_{x=G} = G \cdot F_v + \overline{G} \cdot F_{\overline{v}} = \mathsf{ITE}(G, F_v, F_{\overline{v}}) \ .$$

$x > v$: It is always true that

$$F|_{x=G} = (\mathsf{ITE}(v, F_v, F_{\overline{v}}))|_{x=G} \ .$$

After using distribution law, we get

$$F|_{x=G} = \mathsf{ITE}(v, F_v|_{x=G}, F_{\overline{v}}|_{x=G}) \ .$$

At the end we can rewrite all three formulae in the single one :

$$F|_{x=G} = \begin{cases} F & ; \ x < v \\ \mathsf{ITE}(G, F_v, F_{\overline{v}}) & ; \ x = v \\ \mathsf{ITE}(v, F_v|_{x=G}, F_{\overline{v}}|_{x=G}) & ; \ x > v \end{cases}$$

And that is exactly the formula (4) from section 4. ■

[4] This derivation is summarized after [3, page 41].

References

[1] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, pp. 509–516, June 1978.

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

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," 27^{th} *ACM/IEEE Design Automation Conference*, pp. 40–45, 1990.

[4] N. Wirth, *Računalniško programiranje, 2. del*, vol. 21b of *Izbrana poglavja iz matematike in računalništva*. Ljubljana: Društvo matematikov, fizikov in astronomov Slovenije, 3^{rd} ed., 1986. In Slovene.

[5] J. Kozak, *Podatkovne strukture in algoritmi*, vol. 27 of *Matematika-Fizika – Zbirka univerzitetnih učbenikov in monografij*. Ljubljana, Jadranska c. 19: Društvo matematikov, fizikov in astronomov Slovenije, September 1986. In Slovene.

[6] S. Minato, N. Ishiura, and S. Yajima, "Fast Tautology Checking Using Shared Binary Decision Diagram – Benchmark Results," in *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design* (L. Claesen, ed.), (Houthalen, Belgium), pp. 580–584, November 1989.

[7] Z. Brezočnik and B. Horvat, "Formal hardware specification and verification using Prolog," *Microprocessing and Microprogramming*, vol. 27, no. 1-5, pp. 163–170, 1989.

[8] Z. Brezočnik and B. Horvat, "A functional-based approach to formal VLSI design verification," in *COMPEURO 91, Advanced Computer Technology, Reliable Systems and Applications*, (Bologna, Italy), pp. 317–321, IEEE Computer Society Press, 1991.

[9] D. Verkest and L. Claesen, "Special benchmark session on tauthology checking," in *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design* (L. Claesen, ed.), (Houthalen, Belgium), pp. 568–569, November 1989.

[10] A. L. Fisher and R. E. Bryant, "Performance of COSMOS on the IFIP Workshop Benchmarks," in *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design* (L. Claesen, ed.), (Houthalen, Belgium), pp. 595–599, November 1989.

Aleš Časar (S'93) was born in Kranj, Slovenia, on June 23, 1971. He finished high school in Murska Sobota, in 1990. He placed several times first, second or third at Slovene contests in mathematics, physics and logic. He studies computer science in the Faculty of Technical Sciences at the University of Maribor since 1990. He is the University of Maribor IEEE Student Branch chairman.

Robert Meolic (S'93) was born in Murska Sobota, Slovenia, on April 23, 1972. He studies computer science in the Faculty of Technical Sciences at the University of Maribor since 1990. His research interest is in a domain of Boolean algebra.