McBOOLE: A New Procedure for Exact Logic Minimization

MICHEL R. DAGENAIS, STUDENT MEMBER, IEEE, VINOD K. AGARWAL, SENIOR MEMBER, IEEE, AND NICHOLAS C. RUMIN, SENIOR MEMBER, IEEE

Abstract—**A new logic minimization algorithm is presented. It finds a minimal cover for a multiple-output boolean function expressed as a list of cubes. A directed graph is used to speed up the selection of a minimal cover. Covering cycles are partitioned and branched independently to reduce greatly the branching depth. The resulting minimized list of cubes is guaranteed to be minimal in the sense that no cover with less cubes can exist. The *don't care* at output is handled properly. This algorithm was implemented in C language under UNIX BSD4.2. An extensive comparison with ESPRESSO IIC shows that the new algorithm is particularly attractive for functions with less than 20 input and 20 output variables.**

I. Introduction

THE design of VLSI circuits has been automated to the point where, in particular, there are programs which read logical relations and convert them to an equivalent list of cubes that can realize the desired function. Then, layout assemblers take this list of cubes and produce a programmable logic array (PLA) circuit. One example of such a system was presented by Meyer *et al.* [12]. Many programs have been developed to reduce the number of cubes needed to represent a function and, accordingly, to reduce the chip area consumed by the PLA circuit.

Many years ago, Quine [14] and McCluskey [11] presented a procedure that will always lead to a cover with the minimal number of cubes for a given function. Later, Tison [20], Roth [17], and Dietmeyer [8], among others, modified this procedure to improve its computational requirements. Some programs have been reported lately that provide a minimal cover, based on the procedures mentioned earlier. Ishikawa [10] presented a minimizer that produces a minimal cover based on the procedure of Tison [20]. However, he mentions that it cannot handle, in reasonable time on a mainframe computer, functions having more than 10 input variables. Brayton *et al.* [6] also state that, with existing minimizers, it is impractical to produce a minimal cover for even a medium-sized function with 10 to 15 input variables.

On the other hand, many heuristic procedures have been developed and give good results in minimizing a list of cubes. Even though these procedures often reach a solution which is a minimal cover, in general they give a near-minimal cover with no information on how far it is from a true minimal solution. These minimizers include PRESTO [7], PRESTOL-II [2], MINI II [17], and ESPRESSO II [5]. The latter and its more recent version, ESPRESSO-IIC, have been carefully evaluated [6]. They have been shown to give a very good minimal or near-minimal cover for fairly large functions, in a reasonable CPU time. Other programs such as CAMP [3] produce a minimal cover for medium sized functions when no covering cycles are present.

The new minimizer McBOOLE 21 is based on some very efficient graph and partitioning techniques, and it can find a minimal cover for larger functions than those that could be practically handled up to now. In this paper, the general theory to obtain a minimal cover is first briefly reviewed. The new procedure and structures implemented in McBOOLE are then detailed. Some results and comparisons of McBOOLE and ESPRESSO IIC are presented. Finally the performance and limitations of McBOOLE are discussed. It is assumed that the reader is familiar with the cubic notation for logic functions [8].

II. Notation

A multiple output boolean function of m inputs and n outputs is defined as follows:

$$\text{input space: } B = \{0, 1\}$$

$$\text{output space: } Y = \{0, 1, d\}$$

$$\text{function } f\text{: } B^m \rightarrow Y^n.$$

The value d (*don't care*) at output means that the value is unspecified, and a value of 0 or 1 will be accepted to realize this part of the function. Such a function can be represented by a list of cubes. Each cube contains an input part and an output part as follows:

$$\text{input part: } m \text{ literals that can be } \{0, 1, x\}$$

$$\text{output part: } n \text{ literals that can be } \{0, 1, d\}.$$

The input part identifies the portion of the input space to which a cube applies. The x in the input part matches all the points of the function that have either a 1 or a 0 for this variable. For example a cube with k x's in its input part applies to 2^k points of the function. In a typical truth

Manuscript received March 18, 1985; revised June 7, 1985. This work was supported by a Fellowship to the first author, and by a strategic grant, both from the Natural Science and Engineering Research Council of Canada.

The authors are with the Department of Electrical Engineering, McGill University, Montreal, P.Q., Canada H3A 2A7.

IEEE Log Number 8406324.

table, all the cubes are disjoint and there is no ambiguity. In some cases however, the cubes at input are not disjoint, and many cubes do apply to a point of the function. Such a case usually arises when the cubes were produced by some program; the value of the output vector for this point has then to be determined by priority rules, consistent with the other programs. A d in any of the cubes overrides the 0 or 1 present in any other cube for the same output variable. Similarly, the 1 overrides the 0. When no cubes apply to a point of the function, the output vector is composed of all 0's by default.

For each point in the input space, the output variables having the value 1 are called the *minterms;* therefore, there can be up to n minterms for each of the 2^m points of the input space. Some useful operators are described below. They can apply to a list of cubes as well as to single cubes. Also, any set of minterms of a multiple-output logic function can be expressed as a list of cubes.

The *intersection* of two tubes, $c_1 \sqcap c_2$, is the set of minterms that belong to both c_1 and c_2. Two cubes are said not to intersect when their intersection is empty.

A cube c_1 *covers* another cube c_2, $c_2 \sqsubseteq c_1$, when all the minterms contained in c_2 are also contained in c_1. One can also say that c_2 is contained in c_1.

The *sharp* product of two cubes, $c_1 \# c_2$, is the set of minterms that belong to c_1 but not to c_2. One can also say that c_2 is subtracted from c_1.

The *star* product of two nonintersecting cubes, $c_1 * c_2$, produces a new cube c_3 which is the largest cube that can be formed from both c_1 and c_2.

III. The Extraction Algorithm

The procedure for reaching a minimal cover presented by Quine [14] and McCluskey [11] forms the basis of McBOOLE. Following is a summary of this procedure.

Suppose a given function is represented by a list of cubes F. Let PI be the list of *prime* cubes. A prime cube is any cube that satisfies the condition:

$$c \sqsubseteq F \text{ and for all possible cubes } c_i \sqsubseteq F$$
$$c \not\sqsubseteq c_i \text{ unless } c = c_i. \tag{1}$$

The procedure uses three lists. One list of cubes contains the *don't care* cubes and is named DC. A second list contains the prime cubes for which no decision has been reached as to whether they will be part of the minimal cover; it is the list of *undecided* cubes U. Finally, the cubes that are part of the minimal cover are stored in the list of *retained* cubes R. The procedure to extract a minimal cover from F is as follows:

1) Compute all the prime cubes of F and put them in the undecided list U. Place all the *don't care* cubes in the list DC.
2) Extract all the *extremal* prime cubes, remove them from U, and place them in the *retained* list R. An extremal prime cube is defined as follows:

$$c_i \in U \quad \text{and} \quad c_i \# (U \cup R \cup DC - c_i) \neq \emptyset \tag{2}$$

3) Delete from the list U all the *inferior* prime cubes for which the part not intersecting with $R \cup DC$ is contained in a single other cube of U. More formally, an inferior cube is defined as follows:

$$c \in U, \quad \exists c_i \in U, \quad c_i \neq c, \quad c\#(R \cup DC) \sqsubseteq c_i. \tag{3}$$

Loop through 2 and 3 until no more inferior or extremal prime cube can be found.
4) If the list U is empty a minimal cover is found; the minimal cover is R. If the list is not empty, covering cycles are present. In such a case, branch to the different possible solutions and retain the one with the lowest cost.

IV. The Procedure in McBOOLE

The procedure implemented in McBOOLE contains basically the same steps as listed above. However, new graph and partitioning techniques are used. The highlights of the new procedure are presented for each of the steps listed.

Generation of Prime Cubes

Efficient ways to generate prime cubes were presented by Reush [15] Morreale [13] and Brayton *et al.* [4]. The recursive generation of prime cubes as defined by Brayton is used here, but is modified to avoid pairwise star-products between cubes in lists. This procedure can be summarized as follows.

The list of cubes representing the function is recursively partitioned along the input variables, up to subfunctions that can be expressed by a single cube. This cube is therefore prime in this sub-space. During partitioning, one of the input variables not already used for partitioning this sub-space is selected. Let $nb0$, $nb1$ and nbx be, respectively, the number of cubes with a 0, 1, and x for an unpartitioned variable. The heuristic used to select the next partitioning variable is

$$\min_j (nbx[j]).$$

When the number of cubes with x is the same for many input variables, the second condition is used:

$$\min_j (\max (nb0[j], nb1[j])).$$

A new node in the partitioning tree is then allocated:

```
structure binary node
  {subtree branch0;
   subtree branch1;
   integer partitioning_variable;
   cube_list listx;
  }
```

All the cubes having an x for the selected variable are split into two cubes having, respectively, a 1 and a 0 instead. All the cubes with a 0 for the partitioning variable are sent to the 0 branch and the cubes with a 1 to the 1

branch. The *listx* is empty for the moment. The recursive procedure to generate prime cubes is performed on the 0 and 1 sub-functions.

At this point, there are two subtrees for this node, each containing all the prime cubes for their respective partitioned sub-space. The next step is to produce the list of prime cubes for the space that includes both those partitioned sub-spaces. Brayton *et al.* [4] do this by pair-wise star-products between all the prime cubes of the two sub-spaces. The new cubes formed are compared together for containment and the covered cubes are deleted. In the present algorithm, the process is roughly the same but some unnecessary trials of star-products between distant cubes are avoided. All the cubes in the 0 subtree are scanned, ($\text{cube}_{\text{subtree0}}$). For each of them, the 1 subtree is scanned for adjacent cubes. Since cubes in the 1 and 0 subtree differ in the selected variable, they will be adjacent only if all the other variables intersect. At a node in the 1 subtree, only some sub-branches are explored, depending on the value of $\text{cube}_{\text{subtree0}}$ for the partitioning variable of the node.

If value of variable: 0, scan 0 subtree and *listx*
1, scan 1 subtree and *listx*
x, scan 0 and 1 subtree and *listx*.

This way, only the branches that can contain an adjacent cube are scanned. When an adjacent cube is reached, the newly formed cube is placed in the *listx* of the node at which the sub-spaces are being merged. The new cube is compared with all the cubes already in *listx*. If it is covered by a cube already in the list, the new cube is rejected. If a cube in the list is covered by the new cube, it is deleted from the list. The adjacent cubes, used to form the new cube, are deleted if they are covered by this new cube. When all the star-products are finished, only the prime cubes remain in this space.

Construction of the Covering Graph

A new directed graph is used in McBOOLE to solve the covering problem. When a new cube is formed, it is linked to the cubes that formed it. This way, information about the interaction between the cubes is kept in the graph and makes it possible to solve the covering problem locally. The formation of this new graph is explained below. It is built during the generation of prime cubes. When the termination condition is reached, a single cube remains in a sub-space of the function. The *don't care* part of the cube is extracted and placed in a special list. The *don't care* output bits of the cube are changed to 1 to help the generation of prime cubes. A link ties the *don't care* cube as the ancestor of the single cube in the sub-space. The cube in the sub-space is flagged *basic* and no other cube intersects with it except the *don't care* cube linked to it. All the new cubes are formed during star-product operations:

$$\text{cube}_{\text{new}} = \text{cube}_1 \bigstar \text{cube}_2.$$

When a new cube is added to the graph, the combination rules are:

Rule 1: If the new cube covers $cube_1$ and $cube_2$, the ancestors and descendants of $cube_1$ and $cube_2$ are transferred to the new cube. If either $cube_1$ or $cube_2$ was *basic*, the new cube is flagged *basic*.

Rule 2: If $cube_1$ or $cube_2$ is covered, the ancestors and descendants of the covered cube are transferred to the new cube. If the covered cube was *basic* the new cube inherits its status. A new link is created with the new cube, which becomes a descendant of the uncovered cube.

Rule 3: If neither $cube_1$ nor $cube_2$ are covered, the new cube is a descendant of both $cube_1$ and $cube_2$ and is not *basic*.

Some cubes are deleted when they are covered by another cube in a *listx* being formed. When a deletion occurs, a different rule applies.

Rule 4: If the covered cube was *basic*, the cube covering it will inherit its status. The descendants of the covered cube are passed to the cube covering it. The ancestors of the covered cube are simply ignored since they must already be ancestors of the cube that covers it.

The directed graph formed in this way has some interesting properties. Two of those properties will be defined and explained below.

Lemma 1: Only the *basic* cubes may contain a part not shared with any other prime cube and thus may have

$$\forall\, c \in PI, \quad \text{if } c \text{ not basic}, \quad c\#(PI - c) = \emptyset \qquad (4)$$

Proof: The star-product has the property

$$c_1 \bigstar c_2 = c_3 \text{ implies } c_3 \sqsubseteq \{c_1, c_2\}.$$

When c_1 and c_2 are kept according to Rule 3, c_3 is not *basic* but $c_3\#\{c_1, c_2\} = \emptyset$, and property (4) is verified. When c_1 is deleted by Rules 1 or 2, it passes its status to c_3. So c_3 is not *basic* only if the cube deleted was not *basic* and was covered by its ancestors. If c_1 is deleted and is not *basic* then

$$c_1\#\{\text{ancestors of } c_1\} = \emptyset$$

$$\text{so, since } c_3\#\{c_1, c_2\} = \emptyset$$

$$\text{then } c_3\#(\{c_2\} \cup \{\text{ancestors of } c_1\}) = \emptyset.$$

The same is true when c_2 is deleted and covered.

When a cube gets deleted because it was covered by another one (Rule 4) it passes its status to the cube covering it.

$$c_1 \sqsubseteq c_2 \quad \text{and} \quad c_1 \text{ is deleted}$$

$$\text{if } c_1 \text{ was not basic } c_1\#\{\text{ancestors of } c_1\} = \emptyset$$

$$\text{if } c_2 \text{ was not basic } c_2\#\{\text{ancestors of } c_2\} = \emptyset$$

When c_1 gets deleted, c_2 will not be *basic* only if neither c_1 nor c_2 were *basic*

$$c_2\#((\{\text{ancestors of } c_2\} - c_1) \cup \{\text{ancestors of } c_1\}) = \emptyset$$

$$\text{since } c_1\#\{\text{ancestors of } c_1\} = \emptyset.$$

(Q.E.D.)

The graph has a second and very important property.

Lemma 2: All the cubes intersecting with another one are either its descendants or ancestors, or the descendants of its ancestors. Moreover, all the cubes intersecting with a part L of a cube c_a can be found by recursively intersecting c_a only with those direct descendants, and ancestors with their descendants, which intersect with L. This is expressed more formally by the procedure:

$$\text{let } L = \{c_1, c_2, \cdots, c_i\} \text{ be a part of } c_a$$

$$\{c_1, c_2, \cdots, c_i\} \sqsubseteq c_a.$$

All cubes c such that $c \sqcap \{c_1, c_2, \cdots, c_i\}$ *are found as follows:*

```
{
   scan_intersecting_ancestors(c_a)
   scan_intersecting_descendants(c_a)
}

   scan_intersecting_ancestors(cube)
{
   for all direct ancestors c_di
      {if c_di not already scanned and c_di ⊓ {c_1, c_2, ··· , c_i}
         {put c_di in the list of cubes intersecting
            scan_intersecting_ancestors(c_di)
            scan_intersecting_descendants(c_di)
         }
      }
}

   scan_intersecting_descendants(cube)
{
   for all direct descendants c_di of cube
      {if c_di not already scanned and c_di ⊓ {c_1, c_2, ··· , c_i}
         {put c_di in the list of cubes intersecting
            scan_intersecting_descendants(c_di)
         }
      }
}
```

This property is used extensively in the program to reduce the search time to determine if cubes are extremal or inferior, as defined in (2) and (3).

Proof: When the function is totally partitioned, each partition contains one cube which may be linked to a corresponding *don't care* cube. Since all the partitions are different sub-spaces of the function, all the cubes are disjoint. At that point, *Lemma 2* trivially holds. It must be shown that the property is not altered by the operations that will create or delete cubes in the graph.

When a new cube is formed, if neither c_1 or c_2 are covered, Rule 3 applies:

$$c_1 \star c_2 = c_3, \; c_3 \text{ is a descendant of } c_1 \text{ and } c_2$$

if $c_3 \sqcap L$ a list, then c_1 or $c_2 \sqcap L$ since $c_3 \sqsubseteq \{c_1, c_2\}$.

When, say, c_1 is scanned for a list L, c_3 will also get scanned if it intersects L, since it is a direct descendant of c_1. When c_3 is scanned for L, c_1 will get scanned if $c_1 \sqcap L$. The same thing could, by symmetry, be said for c_2. Also, since it is assumed that the property was verified before c_3 was added to the graph, all the cubes intersecting with L will be scanned, including c_3, because c_1 and c_2 are direct ancestors of c_3.

When c_1 is covered and deleted (Rule 2), its ancestors and descendants are passed to c_3. This way, c_3 simply takes the place of c_1 in the graph, and nothing is changed regarding the cubes intersecting with c_1. The cube c_2 intersects with c_3 but c_2 is kept and linked as a direct ancestor of c_3, and this case is equivalent to Rule 3. The same applies, by symmetry, when c_2 is covered and c_1 is kept.

When both c_1 and c_2 are covered and deleted (Rule 1), c_3 takes the place of both in the graph, and nothing is affected.

When a cube is deleted because it is covered by another one (Rule 4), its descendants are passed to the cube covering it. Suppose c_1 gets covered by c_2 and is deleted. Before c_1 was deleted, all the cubes intersecting with L could be found when $c_1 \sqcap L$.

$$\text{since } c_1 \sqsubseteq c_2$$

$$\text{if } c_1 \sqcap L \quad \text{then} \quad c_2 \sqcap L.$$

When c_1 is deleted, and L is being scanned, the ancestors of c_1 will get scanned anyway since the property holds for c_2 and $c_1 \sqsubseteq c_2$. Also c_2 will get scanned since it intersects with L. When c_2 is scanned, the descendants of c_1 will also get scanned since they were passed to c_2 when c_1 was deleted. Q.E.D.

It has been shown that the formation of new cubes and

the deletion of cubes does not affect the properties stated in Lemma 1 and Lemma 2, provided the rules are used to modify the graph when a cube is added or removed. So at the end of the generation of the prime cubes, a graph that links together related cubes is obtained. Also some of the cubes have the special status *basic*. The reader should note that the generation of prime cubes as well as all the definitions apply to multiple output cubes are presented in [7].

Extraction of a Minimal Cover

All the prime cubes, as well as the list of *don't care* cubes, have now been obtained. A subset of the prime cubes will form the minimal cover. The status of the cubes can be one of three: *undecided*, *unretained*, or *retained*. The cubes that will be *retained* will compose the final solution. All the cubes start with the status *undecided*. Associated with each cube is an uncovered part defined as

$$\text{uncov}(\text{cube}) = \text{cube}\#(\{\text{retained cubes}\} \cup DC).$$

At the beginning, the uncovered part of a cube is the cube itself. The space of the function is divided in two parts: the part covered by the *retained* cubes and that covered only by the *undecided* cubes. The solution is reached when the whole function is covered by the *retained* cubes.

When the cover extraction procedure is started, no prime cube is inferior (property (3)) by definition (1) of a prime cube, since all the prime cubes are *undecided*. Also, since at the beginning all cubes are *undecided*, only the essential prime cubes will be extremal. An essential prime cube is defined by

$$c_i \in \{\text{prime cubes}\}, \quad c_i\#(\{\text{prime cubes}\} - c_i) \neq \emptyset \quad (5)$$

To begin with, the *don't care* cubes are sharped from the uncovered part of the intersecting cubes:

$$\text{uncov}(\text{cube}) = \text{uncov}(\text{cube})\ \#DC.$$

All the cubes that have their uncovered part updated get a special status *affected retained*.

Next all the essential prime cubes are identified and retained. Only the *basic* cubes can satisfy relation (4) from *Lemma 1*, and be essential prime. For each of them L is computed:

$$L = c_i\#\{\text{direct ancestors and descendants of } c_i\}$$

If $L = \emptyset$, c_i is not an essential prime cube. When $L \neq \emptyset$, since no direct ancestors or descendants intersect with the remaining part, no other cube can intersect with it from *Lemma 2*. The cube is then essential and its status is set to *retained*. All the cubes intersecting with c_i get c_i sharped from their uncovered part. The cubes that have their uncovered part updated are flagged *affected retained*.

The next step in the procedure is to find the inferior cubes (Definition (3)). A cube c is inferior or equal to a cube c_j if:

$$c, \exists c_j, \quad c \neq c_j$$
$$c_j \in \{\text{undecided cubes}\}, \quad \text{uncov}(c) \sqsubseteq c_j.$$

Lemma 3: A cube can become inferior only when its uncovered part is modified.

Proof: Initially, the uncovered part of a cube was the cube itself, and no prime cube was inferior by definition (1). So each cube satisfies the relations:

$$\text{uncov}(c) \neq \emptyset$$
$$\forall\, c_j \in (U - c), \quad \text{uncov}(c) \not\sqsubseteq c_j. \quad (6)$$

Since no new cube can be added to the list U, the only way a cube can become inferior is when its uncovered part is modified. Furthermore, since all the cubes having their uncovered part updated are flagged *affected retained*, only those need to be examined to find all the inferior cubes.

Q.E.D.

All the *affected retained* cubes are next examined to identify the inferior cubes (definition (3)). When a cube is inferior, it is removed from the list U. All the cubes intersecting with the uncovered part of the *unretained* inferior cube will get the special status *affected unretained*. If the cube is not inferior, its status is cleared and it cannot become inferior and will not be examined again until it gets *affected* again.

Some extremal cubes now have to be found. Initially, the essential prime cubes were identified and retained. None of the remaining cubes were extremals.

Lemma 4: A cube can become extremal only when an *unretained* cube intersects with its uncovered part.

Proof: Relation (2) which defines extremals, can be expressed in a different but strictly equivalent form:

$$\text{uncov}(c)\#(\{\text{prime cubes}\} - c - \{\text{unretained}\}) \neq \emptyset. \quad (7)$$

It was shown (*Lemma 1*) that only the basic cubes could be extremal at the beginning, since no cubes were *unretained*. Now all the basic cubes have been examined and the extremals have been identified and *retained*. None of the remaining cubes can satisfy (7) since no cubes were *unretained*. Also, this relation can be true for a cube only when another cube intersecting with its uncovered part is *unretained* and removed from U. All the cubes intersecting with the uncovered part of an *unretained* cube were set to *affected unretained* and, therefore, only those cubes can become extremal.

Q.E.D.

All the *affected unretained* cubes are scanned next to identify the extremals. The extremal cubes are removed from U and retained in R. The cube *retained* is sharped from the uncovered part of the intersecting cubes. When a cube gets its uncovered part modified, it gets the status *affected retained*. The cubes which are not extremal have their status cleared and cannot become extremal and will not be examined until they get *affected* again.

This notion of *affected* cubes appears to be new and has been introduced to avoid unnecessary processing of cubes that could not be extremal or inferior. In this way, only cubes which are very likely to be extremals or inferior are examined, and a lot of computation time is saved. As the minimization proceeds, decisions on cubes are made, new

ones get *affected* until, for all the remaining cubes, the following relations apply:

$$\text{uncov}(c) \# (\{\text{undecided cubes}\} - c) = \emptyset$$

$$\forall\, c_j \in \{\text{undecided cubes}\},\quad c \neq c_j,\quad \text{uncov}(c) \not\sqsubseteq c_j.$$

If U is empty at that point, a minimal solution has been reached. The minimal cover is in R. However, if U is not empty, conditions defining covering cycles have been encountered.

Partitioning and Branching Cycles

At this point, the only way to guarantee a minimal cover is to perform branching. An *undecided* cube is picked, retained, and the uncovered part of cubes that will get *affected retained* is updated. The solution that follows from this choice is computed. Next, the initial cycle condition is restored, the same *undecided* cube is *unretained* and the *affected unretained* cubes are appropriately flagged. This alternate solution is computed and, finally, both solutions are compared. The solution with the lowest cost is kept. Note that when these solutions are computed, cubes get *affected* and the procedure explained before applies recursively. In particular, nested cycles can be encountered and many branches might have to be computed and compared.

In the present algorithm, a new partitioning scheme is used that can greatly reduce the number of branches. Partitions are branched and solved independently. When nested cycles are encountered in a partition, they are also recursively partitioned and branched. When the undecided list with cycles is obtained, one cube is picked. All the cubes related with this cube and to each other are put in a partition, which is then solved. From the undecided cubes left, another partition is built and solved. This continues until the undecided list is empty. The procedure to build a partition is:

```
{
 select a cube c
 put it in the partition
 scan_partition(c)
}

 scan_partition(cube)
{
 for all c_d direct ancestors and descendants of cube
   {if c_d not already scanned and c_d ⊓ uncov(cube)
      {put c_d in the partition
       scan_partition(c_d)
      }
   }
}
```

Let c_{pi} be the cubes in the partition and c_{nj} the cubes that are not. The following relation then holds

$$\forall\, (c_{pi}, c_{nj}),\quad \text{uncov}(c_{pi}) \sqcap c_{nj} \quad \text{or} \quad \text{uncov}(c_{nj}) \sqcap c_{pi}. \tag{8}$$

Lemma 5: The cubes, satisfying relation (8) that are not in the solved partition are not affected by the decisions taken for the cubes in the partition. Therefore the covering problem in a partition can be solved independently from the remaining cubes.

Proof: The only cubes that are affected by a decision on a cube c are, from *Lemmas 3 and 4*, $c_i \sqcap \text{uncov}(c)$. In that case, when a decision is taken on any node in the partition, no cube outside the partition (8) is affected, and the two sets of cubes are, therefore, completely independent.

Q.E.D.

It is very important to differentiate independent cycles from nested cycles since, in the first case, the number of branches grows linearly and, in the second, exponentially with the number of cycles encountered. In a typical case examined, the branching depth reached 12 without partitioning and only 4 with this new partitioning scheme. Both ways obviously lead to the same solution, but the partitioning saves a lot of CPU time when independent cycles are encountered.

V. Results

The algorithm described above was programmed in C language under UNIX 4.2BSD, and the results presented were obtained on a VAX11/750. The program is highly portable and should run easily on most computers. Many different logic functions, used in industrial PLA's and provided with the program ESPRESSO IIC [6], were minimized. Table I summarizes the results and compares them to those obtained with ESPRESSO IIC on the same computer. For all these examples Table I shows the number of input and output variables, the number of prime cubes and the number of essential prime cubes. The table also shows the number of cubes for the unminimized function, minimized using McBOOLE and, finally, minimized using ESPRESSO IIC. The CPU time and the memory consumed by each program are also included. The examples in the table are sorted by increasing $time_{\text{McBOOLE}}/time_{\text{ESPRESSO}}$. Thus the examples at the top of the list were functions that could be minimized much faster by McBOOLE than by ESPRESSO IIC, whereas those at the end of the table were relatively difficult for McBOOLE.

In all cases, McBOOLE produced a solution with a lower or equal number of cubes, since it insures a minimal cover. ESPRESSO IIC, on the other hand, uses only heuristics but is quite efficient and, in almost all the cases examined, reached the minimal solution or very near to it.

A careful study of these results helps to identify the factors that affect most the computational requirements of the two algorithms. The major factors identified are listed below.

Storage

The storage requirements of ESPRESSO IIC seem to be most strongly correlated to:

The size of the function (# cubes × the number of variables);

TABLE I
COMPARISON OF MCBOOLE (mc.) AND EXPRESSO IIC (esp.)

name	#input variables	#output variables	#cubes in	#prime cubes	#essential cubes	#cubes out mc.	#cubes out esp.	memory in K mc.	memory in K esp.	CPU seconds mc.	CPU seconds esp.
xor12	12	1	2048	2048	2048	2048	2048	431	6112	10.6	1167.2
xor10	10	1	512	512	512	512	512	190	998	8.4	103.0
pla2x	50	69	183	953	145	178	178	571	2590	141.3	2431.3
dk14mv	15	12	107	56	56	56	56	111	436	1.8	29.3
dk15mv	12	9	68	32	32	32	32	100	320	1.0	11.8
dk17mv	12	11	68	32	32	32	32	100	318	1.0	11.2
simple	9	6	38	20	20	20	20	94	268	0.6	4.7
dk512mv	17	18	138	30	30	30	30	110	348	1.8	13.9
dk27mv	9	9	38	14	14	14	14	90	232	0.5	3.3
col4	14	1	47	14	14	14	14	92	284	0.6	3.6
pi2	4	3	14	19	7	12	12	104	246	0.6	3.0
dc1	4	7	15	22	3	9	9	104	262	0.7	3.4
mult2	4	4	12	12	2	7	7	104	242	0.5	2.2
pi	5	5	10	15	5	8	8	104	244	0.6	2.6
add2	4	3	17	17	7	11	11	104	256	0.6	2.5
example1	3	3	4	5	3	4	4	104	238	0.4	1.5
add4	8	5	397	397	35	75	75	220	570	27.0	83.0
dist	8	5	256	401	23	120	121	272	666	48.3	129.9
in0	15	11	128	706	60	107	107	304	581	49.3	124.6
dc2	8	7	58	173	18	39	40	132	324	6.5	15.9
adr4	8	5	256	397	35	75	75	206	618	27.5	59.9
bigpla	10	10	93	427	20	85	85	238	486	41.2	80.8
gary	15	11	214	706	60	107	107	310	592	61.1	106.6
f51m	8	8	256	561	13	76	76	278	820	63.1	102.3
in1	16	17	110	928	54	104	101	328	842	118.1	170.0
apla	10	12	134	201	0	25	26	238	424	31.3	27.1
in2	19	10	137	666	85	134	137	514	558	184.3	114.3
dk17	10	11	93	111	0	18	18	250	348	31.1	18.1
ryy6	16	1	112	112	112	112	112	288	582	150.6	79.3
dk27	9	9	52	82	0	10	10	246	326	33.6	15.6
alu2	10	8	91	134	36	68	68	416	434	189.1	73.7
mult4 *	8	8	606	606	12	124	133	516	670	859.1	305.7
alu3	10	8	72	540	27	64	66	536	418	296.8	61.3
clpl	11	5	20	143	20	20	20	224	256	28.0	5.3
pope*	6	48	64	593	12	62	63	642	806	5239.1	393.3
sqr6 **	6	12	280	205	3	19	50	290	121	972.7	51.0
alu1	12	8	19	780	19	19	19	656	232	531.3	4.1

*branching abandoned after 6 nested cycles
**branching abandoned after 8 nested cycles

The size of the complement of the function.

For McBOOLE the storage requirements are directly correlated to the number of prime cubes and the number of variables in the function.

CPU Time

The CPU time of ESPRESSO is mostly related to:

The number of cubes in the function which are not essential prime cubes, since the latter are identified early in the procedure and need far less computation;

The number of variables in the function;

The size of the complement of the function. This will affect the expansion of cubes, since these cubes are expanded until a cube in the complement intersects with them.

For McBOOLE the CPU time is related to:

The number of prime cubes;

The number of nested cycles in the function.

In some difficult cases, the number of nested cycles can exceed a limit fixed by the user, whereupon the branching in McBOOLE is abandoned. In such a case the user is warned that the minimal solution is not guaranteed. Three such cases are presented in the table and are identified. It is interesting to see that, at least in these three cases, even though a minimal cover could not be insured by McBOOLE, the solution obtained was superior to the solution generated by ESPRESSO; the difference between the solutions obtained by the two programs is particularly interesting in the case of *mult*4.

Because of their different requirements, the two minimizers take very different amounts of CPU time depending on the function. It is apparent that when the number of prime cubes is not much larger than the initial number of cubes, McBOOLE is faster. For functions where the number of cubes is large but they are not too closely related, the number of prime cubes is not too big. The graph and partitioning techniques are very efficient for these loosely coupled cubes. A very good example to illustrate this is the 12 input exclusive or (XOR12) function. For that case, McBOOLE performed the minimization more than 100 times faster than ESPRESSO IIC.

On the other hand, for functions having a high number of prime cubes, McBOOLE is severely penalized because it generates all the prime cubes. The example *alu*1 illustrates this very well. Since one cannot know in advance the number of prime cubes, and it is not always easy to see if cubes are loosely coupled, the following rules can be used to select algorithms. It has been observed that when the number of either input or output variables is large the problem gets more difficult; a problem with 5 input and 50 output variables will be, in general, much more difficult than a problem with 10 input and 10 output variables. Because of that, we will refer to functions with, for example, up to 10 input and 10 output variables as simply functions with up to 10 variables.

For up to 10 variables the number of prime cubes is usually reasonable and McBOOLE is generally faster than ESPRESSO IIC.

For up to 20 variables, the CPU time of McBOOLE is usually in the same range as ESPRESSO IIC for most cases where the number of prime cubes does not grow too much. McBOOLE is, therefore, still very attractive for these cases, since it always produces a minimal cover. There will always be, however, some difficult functions with less than 20 variables for which an exact solution is impractical.

Bigger functions must be examined carefully to determine if McBOOLE is suitable. Some functions like *pla*2*x* in the table were easily minimized although they contained more than 50 input and 50 output variables.

The results obtained in the comparisons of McBOOLE with ESPRESSO IIC and other minimizers, lead to the following conclusions.

The previously reported limit of 10 input variables [6], [10] for producing a minimal cover, has been improved to around 20 input variables for most typical industrial PLA's. The features of McBOOLE that lead to this improvement are:

a) A more efficient generation of prime cubes which avoids pair-wise star products.
b) The covering graph gives all the intersecting cubes more rapidly.
c) Because the *affected* cubes are flagged, many cubes are not processed needlessly.
d) Finally, the cycles are partitioned and branched independently.

The results obtained are also useful because they provide an evaluation of ESPRESSO IIC. It has been verified

that it gives generally very near to minimal solutions for the medium to large functions for which the minimal cover could be obtained with McBOOLE. Therefore, one can be confident that ESPRESSO IIC most probably also gives good results for very large functions that currently cannot be handled by McBOOLE.

VI. Further Improvements

The very high number of prime cubes was the limitation in most of the examples that McBOOLE could not handle. There were also some other cases where the number of nested cycles was unmanageable and the branching had to be abandoned. The heuristic minimizers ESPRESSO IIC and MINI II can handle much larger functions because they do not generate all the prime cubes and use some very efficient logic operators like Tautology [4]. Some tests exist in ESPRESSO IIC and MINI II to identify the essential prime cubes without generating all the prime cubes.

A procedure has been developed by the authors to form and identify the other extremal cubes, by only generating and examining a small subset of the prime cubes, locally around a cube being examined. This new procedure has very low memory requirements since only a small subset of the prime cubes is stored at any time. It has also the interesting property that the initial list of cubes is iteratively improved and the procedure can be stopped at any point yielding a good solution.

However, the branching cannot easily be implemented using this approach so the minimal solution is found only when no covering cycles are present. Also, even though the memory requirements are improved, it appears that many prime cubes have to be generated locally more than once, since they are not stored. The CPU time required by the new procedure is not affected much by the fact that not all the prime cubes are generated.

The goal of McBOOLE was to minimize the number of cubes. It is also desirable to reduce the number of literals, or the number of transistors, in a PLA. Reducing the number of transistors can improve the foldability of a PLA. Also it can improve the speed of a PLA if the removed transistors are in the critical path. Therefore, the minimization of the number of literals is a useful secondary goal; for a given number of cubes, try to minimize the number of literals. One way to improve McBOOLE would be to take its output and pass it to a very fast minimizer, like in [7], that reduces the number of literals. ESPRESSO includes some steps at the end to reduce the number of literals, and, for a given number of cubes, very often produces a solution which, compared with McBOOLE's, has fewer literals.

Appendix I

McBOOLE Procedure for Logic Minimization

```
main( )
  {
    read cubes
    generate_prime_cubes_by_recursive_partitioning( )
    Set the status of all cubes to undecided
    For all cubes set uncov(c) = c
    For all the don't care cubes retain(cube)
    For all basic cubes, if (c # {direct ancestors, descendants} ≠ ∅) retain(c)
    Solve (remaining undecided cubes)
  }
Solve (list of cubes)
  {
    Until the affected queue is empty
    {cube = get next cube from queue
      if the cube is affected retained
        {if (uncov(cube) = ∅) discard the cube
         if (uncov(cube) ⊑ c_i, c_i undecided) unretain_inferior(cube)
         else clear the cube status
        }
      if the cube is affected unretain
        {if (uncov(cube) # {undecided cubes} ≠ ∅) retain(cube)
         else clear the cube status
        }
    }
    Until undecided list is not empty
    {Take an undecided cube
      put in partition all related cubes: scan_partition(cube)
      branching_cube = select_branching_cube(partition)
```

```
            Store the partition
            retain(branching_cube)
            solve(partition)
            store the solution and restore the original partition
            unretain_inferior(branching_cube)
            solve(partiton)
            compare the two solutions and keep the best one
        }
    return( )
}

retain(cube)
    {For all c_i ∈ U, if c_i ⊓ uncov(cube)
        {If c_i not already in queue, put it on the queue
         uncov(c_i) = uncov(c_i) # cube its status is set to affected retained
        }
      If cube is a don't care cube, remove it from U
      else transfer cube from U to R
    }

unretain_inferior(cube)
  {For all c_i ∈ U, if c_i ⊓ uncov(cube)
      {If c_i not already in queue, put it in the queue.
       The status of c_i is set to affected unretain
      }
    remove cube from the undecided list
  }
```

ACKNOWLEDGMENT

The authors are grateful to Ajoy Bose and Michael Thong for inviting the first author to AT&T Bell Labs where he was able to compare McBOOLE to several logic minimizers including ESPRESSO IIC. While there he had stimulating discussions with Jean Dussault and Prathima Agrawal. In particular, Jean Dussault also helped in the evaluation of McBOOLE and ESPRESSO IIC.

REFERENCES

[1] Z. Arevalo and J. G. Bredeson, "A method to simplify a Boolean function in a near minimal sum-of-products for programmable logic arrays," *IEEE Trans. Computers*, vol. C-27, p. 1028, Nov. 1978.

[2] M. Bartholomeus and H. de Man, "PRESTOL-II: Yet another logic minimizer for programmed logic arrays," in *Proc. 1985 Int. Symp. Circuits and Systems*, Kyoto, Japan, June 1985.

[3] N. Biswas, "Computer aided minimization procedure for Boolean functions," in *Proc. 21st Design Automation Conf.*, p. 699, June 1984.

[4] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Trager, and D. Y. Y. Yun, "Fast recursive Boolean function manipulation," in *Proc. 1982 Int. Symp. on Circuit and Systems*, p. 58, May 1982.

[5] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, "ESPRESSO II a new logic minimizer for PLA," in *IEEE Proc. Custom Integrated Circuits Conf.*, p. 370, May 1984.

[6] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic, Boston, 1984.

[7] D. W. Brown, "A state machine synthesizer SMS," in *Proc. 18th Design Automation Conf.*, p. 301, June 1981.

[8] L. Dietmeyer, *Logic Design of Digital Systems*. Boston, MA: Allyn and Bacon, Mar. 1979.

[9] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. Develop.*, pp. 443–458, Sept. 1974.

[10] K. Ishikawa, T. Sasao, and H. Terada, "A minimization algorithm for logical expressions and its bounds of application," *Trans. IECE Japan*, vol. J-65D, p. 797, June 1982.

[11] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 35, p. 1417, Apr. 1956.

[12] M. J. Meyer, P. Agrawal, R. G. Pfister, "A VLSI FSM design system," in *Proc. 21st Design Automation Conf.*, p. 434, June 1984.

[13] E. Morreale, "Recursive operator for prime implicant and irredundant normal form determination," *IEEE Trans. Computers*, vol. C-19, p. 504, June 1970.

[14] W. V. Quine, "A way to simplify truth functions," *Amer. Math. Monthly*, vol. 62, p. 627, Nov. 1955.

[15] B. Reush, "Generation of prime implicants from subfunctions and a unifying approach to the covering problem," *IEEE Trans. Computers*, vol. C-24, p. 924, Sept. 1975.

[16] T. Rhyme, P. S. Noe, M. H. McKinney, and U. W. Pooch, "A new technique for the fast minimization of switching functions," *IEEE Trans. Computers*, vol. C-26, p. 757, Aug. 1977.

[17] J. P. Roth, R. M. Karp, "Minimization over Boolean graphs," *IBM J.*, vol. 6, no. 2, p. 227, Apr. 1962.

[18] T. Sasao, "Input variable assignment and output phase optimization of PLAs," *IEEE Trans. Computers*, vol. C-33, p. 879, Oct. 1984.

[19] A. Svoboda, "The concept of term exclusiveness and its effect on the theory of Boolean functions," *J. ACM*, vol. 22, no. 3, p. 425, July 1975.

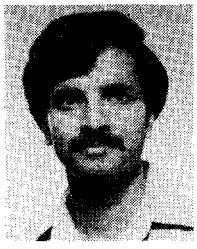
[20] P. Tison, "Generalization of consensus theory and application to the minimization of Boolean functions," *IEEE Trans. Electron. Computers*, vol. EC-16, p. 446, Aug. 1967.

[21] M. R. Dagenais, V. K. Agarwal, N. C. Rumin, "The McBOOLE logic minimizer," in *Proc. 22nd Design Automation Conf.*, p. 667, Las Vegas, NV, June 1985.

*

Michel Dagenais (S'85) was born in 1962. He received the B.Eng. degree from Ecole Polytechnique de Montreal, Canada, in 1983. He is currently a Ph.D. candidate at McGill University, Montreal.

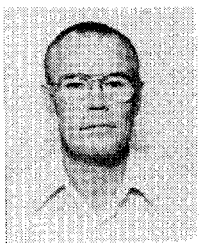
His research interests include several aspects of the computer-aided-design of integrated circuits with emphasis on logic synthesis, design verification and systolic architectures.



Vinod K. Agarwal (S'74-M'77-SM'84) received the B.E. (Hons.) degree in electronics engineering from Birla Institute of Technology and Science, Pilani, India, in 1973; the M.S. degree in electrical engineering from University of Pittsburgh, Pittsburgh, PA, in 1974; and the Ph.D. degree in electrical engineering from the Johns Hopkins University, Baltimore, MD, in 1977.

Since 1978 he is an Associate Professor in the Department of Electrical Engineering, McGill University, Montreal, Canada. His teaching and research interests are in switching theory, computer structure and organization, fault-tolerant computing, and VLSI design.

Dr. Agarwal is a member of ACM, Sigma Xi, and IFIP Working Group 10.4 on Reliability and Fault-Tolerant Computing.



Nicholas Rumin (S'60-M'65-SM'78) was born in Cairo, Egypt, in 1933. He received the B.Eng. degree in engineering physics, the M.Sc. degree in physics, and the Ph.D. degree in electrical engineering, from McGill University, Montreal, P.Q., Canada, in 1957, 1961 and 1966, respectively.

From 1957 to 1959 he worked for Canadian Marconi on the design of transistorized communications circuits. He was a research associate at the Montreal Neurological Institute from 1961 to 1963, involved in the application of radio-isotopes to the localization of intracranial lesions and to the study of blood circulation. From 1961 to 1964 he was also a Lecturer in the Department of Electrical Engineering at McGill University. In 1965 he joined the RCA Research Laboratories in Montreal where he was involved in the characterization of bipolar transistors and monolithic integrated circuits as well as in the development of photoconductive detectors. Since 1967 he has been with the Department of Electrical Engineering at McGill University where he currently holds the rank of Professor. In 1975 he spent nine months on sabbatical leave with the Bell Northern Research Laboratories in Ottawa, Ont., Canada, where he worked on the modeling of microwave transistors and the computer-aided design of high speed nonlinear circuits. His current research interests are in the general area of VLSI circuits, and include simulation, modeling and IC design.

Dr. Rumin is a member of the Order of Engineers of Quebec.