Explicit and Implicit Algorithms for Binate Covering Problems

Tiziano Villa, Timothy Kam, *Member, IEEE*, Robert K. Brayton, *Fellow, IEEE*, and Alberto L. Sangiovanni-Vincentelli, *Fellow, IEEE*

***Abstract*— We survey techniques for solving binate covering problems, an optimization step often occurring in logic synthesis applications. Standard exact solutions are found with a branch-and-bound exhaustive search, made more efficient by bounding away regions of the search space. Standard approaches are said to be explicit because they work on a direct representation of the binate table, usually as a matrix. Recently, covering problems involving large tables have been attacked with implicit techniques. They are based on the representation by reduced-ordered binary decision diagrams of an encoding of the binate table. We show how table reductions, computation of a lower bound, and of a branching column can be performed on the table so represented. We report experiments for two different applications that demonstrate that implicit techniques handle instances beyond the reach of explicit techniques. Various aspects of our original research are presented for the first time, together with a selection of the most important old and new results scattered in many sources.**

I. Introduction

At the core of the exact solution of various logic synthesis problems often lies a so-called covering step that requires the choice of a set of elements of minimum cost that *cover* a set of ground items, under certain conditions. Prominent among these problems are the covering steps in the Quine–McCluskey procedure for minimizing logic functions, selection of a minimum number of encoding columns that satisfy a set of encoding constraints, selection of a set of encodable generalized prime implicants, state minimization of finite-state machines, technology mapping, and Boolean relations. Let us review first how covering problems are defined formally.

Suppose that a set $S = \{s_1, \cdots, s_n\}$ is given. The cost of selecting s_i is k_i where $k_i \geq 0$. In a general formulation also the cost of not selecting s_i may be nonnegative, but here it will be assumed that the cost of not selecting an item is strictly zero, unless otherwise stated. Most problems of practical interest in logic synthesis satisfy this assumption. The explicit algorithms that will be described can be extended easily to handle the general formulation. By associating a binary variable x_i to s_i, which is 1 if s_i is selected and 0 otherwise, the binate covering problem (BCP) can be defined as finding $S' \subseteq S$ that minimizes

$$\sum_{i=1}^{n} k_i x_i$$

subject to the constraint

$$A(x_1, x_2, \cdots, x_n) = 1$$

where A is a Boolean function, sometimes called the constraint function. The constraint function specifies a set of subsets of S that can be a solution. No structural hypothesis is made on A. Binate refers to the fact that A is in general a binate function.[1] BCP is the problem of finding an onset minterm of A that minimizes the cost function (i.e., a solution of minimum cost of the Boolean equation $A(x_1, x_2, \cdots, x_n) = 1$).

If A is given in product-of-sums form, it is possible to write A as an array of cubes (that form a matrix M with coefficients from the set $\{0, 1, 2\}$). Each variable of A is a column and each sum (or clause) is a row, and the problem can be interpreted as one of finding a subset C of columns of minimum cost, such that for every row r_i, either

1) $\exists j$ such that $a_{ij} = 1$ and $c_j \in C$, or
2) $\exists j$ such that $a_{ij} = 0$ and $c_j \notin C$.

In other words, each clause must be satisfied by setting to 1 a variable appearing in it in the positive phase or by setting to 0 a variable appearing in it in the negative phase. In a unate covering problem, the coefficients of A are restricted to the values 1 and 2, and only the first condition must hold. Here, we shall consider the minimum binate covering problem where A is given in product-of-sums form. In this case, the term *covering* is fully justified because one can say that the assignment of a variable to 0 or 1 covers some rows that are satisfied by that choice. The product-of-sums A is called covering matrix or covering table.

An example of binate covering formulation of a well-known problem is finding the minimum number of prime compatibles

Manuscript received October 18, 1995; revised July 23, 1997. This work was supported by DARPA, NSF, SRC, and industrial grants from Bell Northern, Cadence, Digital, Fujitsu, Intel, MICRO, and Motorola. This paper was recommended by Associate Editor M. Fujita.

T. Villa is with PARADES, 00186 Roma, Italy.

T. Kam is with the Strategic CAD Laboratories, Intel Corporation, Hillsboro, OR 97124-6497 USA.

R. K. Brayton and A. L. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720 USA.

Publisher Item Identifier S 0278-0070(97)07802-0.

[1] A function is binate if it is not unate. A function $f(x_1, x_2, \cdots, x_n)$ is unate if for every x_i, $i = 1, \cdots, n$, f is either positive or negative unate in the variable x_i. f is said to be positive unate in a variable x_i, if for all 2^{n-1} possible combinations of the remaining $n-1$ variables, $f(x_1, x_2, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n) \geq f(x_1, x_2, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n)$. In other words, changing variable x_i from 0 to 1, f does not decrease. Similarly, f is said to be negative unate in a variable x_i, if for all 2^{n-1} possible combinations of the remaining $n-1$ variables, $f(x_1, x_2, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n) \geq f(x_1, x_2, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n)$.

that are a minimum closed cover of a given FSM. A binate covering problem can be set up, where each column of the table is a prime compatible and each row is one of the covering or closure clauses of the problem [18]. There are as many covering clauses as states of the original machine, and each of them requires that a state is covered by selecting any of the prime compatibles in which it is contained. There are as many closure clauses as prime compatibles, and each of them states that if a given prime compatible is selected, then for each implied class in its corresponding class set, one of the prime compatibles containing it must be chosen too. In the matrix representation, table entry (i, j) is 1 or 0 according to the phase of the literal corresponding to prime compatible j in clause i; if such a literal is absent, the entry is 2.

A special case of the binate covering problem is the unate covering problem, where no literal in the negative phase is present. Exact two-level logic minimization [33], [39] can be cast as a unate covering problem. The columns are the prime implicants, the rows are the minterms, and there is a 1 entry in the matrix when a prime contains a minterm. Notice that the feasibility (i.e., finding if there is a choice of columns that cover all the rows) of unate covering is trivial to answer (always yes), while the feasibility of binate covering is NP-complete [16]. Moreover unate covering and *a fortiori* binate covering are NP-complete problems [16].

Various techniques have been proposed to solve binate covering problems. A class of them [3], [29] is branch-and-bound techniques that build explicitly the table of the constraints expressed as product-of-sum expressions and explore in the worst case all possible solutions, but avoid the generation of some of the suboptimal solutions by a clever use of reduction steps and bounding of search space for solutions. We will refer to these methods as explicit.

A second approach of binate covering [31] formulates the problem with binary decision diagrams (BDD's), and reduces to finding a minimum cost assignment to a shortest path computation. A BDD [4], [1] is a canonical directed acyclic graph that represents logic functions. The number of items that a BDD can represent corresponds to the number of paths of the BDD to the 1 terminal, while the size of the BDD is determined by the number of nodes of the DAG. There is no monotonic relation between the size of a BDD and the number of elements or paths that it represents. It is an experimental fact that often very large sets, that cannot be represented explicitly, have a compact BDD representation. In that case, the number of variables of the BDD is the number of columns of the binate table. A mixed technique has been proposed in [21]. It is a branch-and-bound algorithm, where the clauses are represented as a conjunction of BDD's. The usage of BDD's leads to an effective method to compute a lower bound on the cost of the solution.

Existing explicit methods do quite well in solving exactly small- and medium-sized examples, but fail to complete on larger ones. The reason is that either they cannot build the binate table because the number of rows and columns is too large, or that the branch-and-bound procedure would take too long to complete. The approach of building a BDD of the constraint function and computing the shortest path fails when the number of variables (i.e., columns) is too large because a BDD with many thousands of variables usually cannot be stored in available computer memory.

Explicit techniques fail when they are required to represent sets of very large cardinality. Fortunately, as an alternative, one may represent sets by encoding them over an appropriate Boolean space. In this way, one operates on the Boolean characteristic function of the encoded set, represented by reduced ordered binary decision diagrams (ROBDD) [4], [1]. Set operations are easily turned into Boolean operations on the corresponding BDD's. So we can manipulate sets by a series of BDD operations (Boolean connectives and quantifications) with a complexity depending on the sizes of the manipulated BDD's, but not depending linearly on the cardinality of the sets that are represented. One hopes that complex set manipulations of a given application have as counterparts Boolean propositions that can be represented with compact BDD's. Of course, this is not always the case, and it may happen that an intermediate BDD computation blows up. Sometimes, it helps to transform propositional sentences into logically equivalent ones, easier to compute with BDD manipulations.

The previous insight has already been tested in a series of applications from the implicit enumeration of subsets of states of a finite-state machine (FSM) in [9] and [43] to the implicit computation of implicants, primes, and essential primes of a two-valued or multivalued function in [10], [30], [42], and [13]. There are functions whose primes could be computed only implicitly.

The fixed-point dominance computation in the covering step of the Quine–McCluskey procedure has been implicitized in [12] and [42]. A key technology in both cases has been the use of quantifier-free recursive implementations of matrix reductions (as was the case for prime generation). Actually, Coudert went beyond the standard Quine–McCluskey formulation using the concept of transposing functions by which the problem is mapped into a lattice, and then row and column dominance is replaced by the computation of least upper bounds and greatest lower bounds of the lattice. Using these techniques, he computed the cyclic core of all logic functions of the ESPRESSO benchmark, for some of which ESPRESSO had failed the task. Another difference between the implementations in [12] and [42] is the usage of BDD's in the latter and of ZBDD's [34] in the former. It appears that ZBDD's are a better data structure for this application. We will not elaborate further on these techniques in this paper, and we refer the reader to the original exposition by Coudert in [7] and to [44] which also contains a worked-out example.

The implicit computation of prime compatibles of an FSM was described in [24], [25], and [20]. In some cases, their number is exponential in the number of states (the largest recorded number is 2^{1500}). Once prime compatibles have been obtained, one must solve a binate covering problem to choose a minimum closed cover. Of course, one cannot build and solve explicitly a table of such dimensions (this would defeat the purpose of computing implicitly prime compatibles in the first place). So it is necessary to extend implicit techniques to the solution of the binate covering problem. Another application of interest is the selection of a set of encodable generalized prime

```
sm_mincov(M, select, weight, lbound, ubound) {
    /* Apply row dominance, column dominance, and select essentials */                    (1)
    if (!sm_reduce(M, select, weight, ubound)) return empty_solution
    /* See if Gimpel's reduction technique applies */                                     (2)
    if (gimpel_reduce(M, select, weight, lbound, ubound, &best)) return best
    /* Find lower bound from here to final solution by independent set */                 (3)
    indep = sm_maximal_independent_set(M, weight)
    /* Make sure the lower bound is monotonically increasing */                           (4)
    lbound_new = max(cost(select) + cost(indep), lbound)
    /* Bounding based on no better solution possible */                                   (5)
    if (lbound_new ≥ ubound) best = empty_solution
    else if (M is empty) { /* New best solution at current level */                       (6)
        best = solution_dup(select)
    } else if (sm_block_partition(M, &M1, &M2) gives non-trivial bi-partitions) {         (7)
        select1 = empty_solution
        best1 = sm_mincov(M1, select1, weight, 0, ubound − cost(select))                  (8)
        /* Add best solution to the selected set */                                       (9)
        if (best1 = empty_solution) best = empty_solution
        else {                                                                           (10)
            select = select ∪ best1
            best = sm_mincov(M2, select, weight, lbound_new, ubound)
        }
    } else { /* Branch on cyclic core and recur */                                       (11)
        branch = select_column(M, weight, indep)
        select1 = solution_dup(select) ∪ branch
        let M_branch be the reduced table assuming branch column in solution             (12)
        best1 = sm_mincov(M_branch, select1, weight, lbound_new, ubound)
        /* Update the upper bound if we found a better solution */                       (13)
        if (best1 ≠ empty_solution) and (ubound > cost(best1)) ubound = cost(best1)
        /* Do not branch if lower bound matched */                                       (14)
        if (best1 ≠ empty_solution) and (cost(best1) = lbound_new) return best1
        let M_branch-bar be the reduced table assuming branch column not in solution     (15)
        best2 = sm_mincov(M_branch-bar, select, weight, lbound_new, ubound)
        best = best_solution(best1, best2)
    }
    return best
}
```

Fig. 1. Detailed branch-and-bound algorithm.

implicants (GPI's), as defined in [15] and [31]. It is not feasible to generate GPI's and to set up a related covering table by explicit techniques on nontrivial examples. Using techniques as in [30] and [42], GPI's can be generated implicitly. An implicit table solver is therefore needed there too. We will use mainly the two latter applications to illustrate our techniques, but one could list a host of other problems in logic synthesis where a binate table solver would play an important role. Another application reported in the literature is the implicit selection of the minimum number of encoding dichotomies that satisfy a set of encoding constraints [40], [14].

We describe an implicit formulation of the binate covering problem and present an implementation. The implicit binate solver has been tested for state minimization of ISFSM's and pseudo-NDFSM's [24], [25], and for the selection of an encodable set of GPI's [44]. The reported experiments show that implicit techniques have advanced the frontier of instances where binate covering problems can be solved exactly, resulting in better optimizations in key steps of sequential logic synthesis.

In the following sections, we will review the known algorithms to solve covering problems, and then we will describe a new branch-and-bound algorithm based on implicit computations. The remainder is organized as follows. We have defined the minimum cost binate covering problem in this section. The classical solution based on a branch-and-bound scheme is introduced in Section II. In Section III, we survey the classical reduction rules used in explicit algorithms. Methods to solve binate covering finding a shortest path in a graph-based representation of the clauses are found in Section IV. Our implicit binate covering algorithm is then outlined in Section V. Section VI illustrates how reduction techniques can be implicitized. Other kinds of implicit table manipulations are introduced in Section VII. Finally, we give experimental results in Section VIII, for two applications: state minimization of ISFSM's [18] and selection of generalized prime implicants [15].

II. A Branch-and-Bound Algorithm for Minimum Cost Binate Covering

We will survey in this section a branch-and-bound solution of minimum cost binate covering. This technique has been described in [19], [18], [2], and [3], and implemented in successful computer programs [38], [36], and [41]. The branch-and-bound solution of minimum binate covering is based on a recursive procedure. A run of the algorithm can be described by its computation tree. The root of the computation tree is the input of the problem, an edge represents a call to *sm_mincov*, and an internal node is a reduced input. A leaf is reached when a complete solution is found or the search is bounded away. From the root to any internal node, there is a unique path, that is the current path for that node. In the sequel, we will describe in detail the binary recursion procedure. The

presentation will refer to the pseudocode *sm_mincov*, shown in Fig. 1.

A. The Binary Recursion Procedure

We will see in the next section that BCP can be solved by the following recursive equation:

$$\mathrm{BCP}(M_f) = \textit{BestSolution}(\mathrm{BCP}(M_{f_{x_i}}) \cup \{x_i\}, \mathrm{BCP}(M_{f_{\overline{x_i}}}))$$

where M_f is the binate table that corresponds to a function in product-of-sum form f, and $\mathrm{BCP}(M_{f_{x_i}})$ [respectively, $\mathrm{BCP}(M_{f_{\overline{x_i}}})$] is the subproblem expressed by the function f_{x_i} (respectively, $f_{\overline{x_i}}$). $\mathrm{BCP}(M_f)$ returns an onset minterm of f that minimizes the cost function. The previous equation can potentially generate an exponential number of subproblems, but powerful dominance and bounding techniques as well as good branching heuristics help in keeping the combinatorial explosion under control.

The inputs to the algorithm are

- a covering matrix M;
- a current-path partial solution *select* (initially empty);
- a row of nonnegative integers *weight*, whose ith element is the cost or weight of the ith column of M;
- a lower bound *lbound* (initially set to 0), which is equal to the cost of the partial solution on the current path (a monotonic increasing quantity along each path of the computation tree);
- an upper bound *ubound* (initially set to the sum of weights of all columns in M), which is the cost of the best overall complete solution previously obtained (a globally monotonic decreasing quantity).

The output is the best column cover for input M extended from the partial solution *select* along the current path, called the best current solution, if this solution costs less than *ubound*. An empty solution is returned if a solution cannot be found which beats *ubound* or an infeasibility is detected. By infeasibility, we mean the case when no satisfying assignment of the product of clauses exists. Even though the initial problem in a typical logic synthesis application usually has at least a solution, some subproblems in the branch-and-bound tree may be infeasible. When *sm_mincov* is called with an empty partial solution *select* and initial *lbound* and *ubound*, it returns a best global solution.

As shown in Fig. 1. the algorithm *sm_mincov* first calls a procedure *sm_reduce* that applies to M essential column detection and dominance reductions. The type of domination operations and the way in which they are applied are the subject of Section III. Another more complex reduction criterion (Gimpel's rule) can also be applied (see Section III-L). These reduction operations delete from M some rows, columns, and entries. What is left after reduction is called a cyclic core. The final goal is to get an empty cyclic core. The value of the lower bound is updated using a maximal independent set computation (see Section II-C1). If no bounding is possible and the reductions do not suffice to completely solve the problem, a partition of the reduced problem into disjoint subproblems is attempted (see Section II-B), and each of them is solved recursively. When everything fails, binary recursion is performed by choosing a branch column (see Section II-D). Solutions to the subproblems obtained by including the chosen column in the covering set or by excluding it from the covering set are computed recursively, and the best solution is kept (the second recursion is skipped if the solution to the first one matches the updated lower bound).

B. N-Way Partitioning

If the covering matrix M can be partitioned into two disjoint blocks M_1 and M_2, the covering problem can be reduced to two independent covering subproblems, and the minimum covering for M is the union of the minimum coverings for M_1 and M_2. Such bipartition can be found by putting in M_1 a row and all columns that have an element in common with the row (i.e., the columns intersecting the row), and recursively all rows and columns intersecting any row or column in M_1. The remaining rows and columns (i.e., not intersecting any row or column in M_1) are put in M_2. This algorithm can be generalized to find partitions made by N blocks.

C. Lower Bounds

1) Maximal Independent Set: The cardinality of a maximum set of pairwise disjoint rows of M (i.e., no 1's in the same column) is a lower bound on the cardinality of the solution to the covering problem because a different element must be selected for each of the independent rows in order to cover them. If the size of current solution plus the size of the independent set is greater than or equal to the best solution seen so far, the search along this branch can be terminated because no solution better than the current one can possibly be found. It is also true that the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover, so that the search can be terminated if a solution is found of size equal to this lower bound. Since finding a maximum independent set is an NP-complete problem, in practice, a heuristic is used that provides a weaker lower bound. Notice that even the lower bound provided by solving exactly maximum independent set is not sharp. In [8], an example of size $O(n^2)$ is shown, whose minimal solution has a $O(n)$ cost, but whose lower bound by independent set is a constant 1. In practice, a lower bound by independent set is poor when the covering matrix is dense.

In [38], [36], and [41], the adjacency matrix B of a graph whose nodes correspond to rows in the cover matrix M is created. In the binate case, only rows are taken into consideration which do not contain any 0 element. An edge is placed between two nodes if the two rows have an element in common. While B is nonempty, a row R_i of B is found that is disjoint from a maximum number of rows (i.e., the row of minimum length in B). The column of minimum weight intersecting R_i is also found. The weight is cumulated in the independent set cost. All rows having elements in common with R_i are then deleted from B. At the end of the *while*-iteration, a set of pairwise disjoint rows (independent set) and their minimum covering cost are found. Notice that one could think of the problem in a dual way as finding a maximal clique in a graph with the same rows as before, and edges between two nodes representing two disjoint rows.

In [8], a detailed analysis of independent set computations is made. A quantitative ratio between a maximal cost independent set and the independent set computed by a greedy algorithm based on set packing is derived.

2) Logarithmic Ratio Lower Bound: There are problems for which approximation algorithms have been developed with a fixed ratio bound, i.e., independent of the size n of the input. For other problems, like the unate covering problem, the best that can be done is to let the ratio bound grow as a function of n. Therefore, as the size of the instance gets larger, the size of the approximate solution may grow with respect to the size of an optimal solution. In [8], a logarithmic ratio lower bound on unate problems has been presented. Since the logarithm function grows slowly, this bound is of practical value. The result is originally due to Chvatal; a textbook exposition and references are in Leiserson [6]. Consider an instance $(X, \mathcal{F})$ of a set-covering problem with a finite set X and a family $\mathcal{F}$ of subsets of X. A greedy algorithm[2] that selects at each step the set that covers the most remaining uncovered elements returns a solution whose ratio with respect to the optimal solution is bounded by

$$H(\max\{|S|\colon S \in \mathcal{F}\}) = \ln|X| + 1$$

where $H(d) = \Sigma_{i=1}^{d} 1/i$. The result holds also for positive weights on the sets and for satisfiable binate problems. Strong improvements are reported compared to the traditional lower bound computed by approximating a maximum independent set.

D. Selection of a Branching Column

The selection of a good branching column is essential for the efficiency of the branch-and-bound algorithm. Since the time taken by the selection is a significant part of the total, a tradeoff must be made between quality and efficiency.

In [38], [36], and [41], the selection of the branching variable is restricted to columns intersecting the rows of the independent set because a unique column must eventually be selected from each row of the maximal independent set. Among those rows, the selection strategy favors columns with large number of 1's and intersecting many short rows. Short rows are considered difficult rows, and choosing them first favors the creation of essential columns. More precisely, the column of highest merit is chosen. The merit of a given column is computed as the product of the inverse of the weight of the column multiplied by the sum of the contributions of all rows intersected in a 1 by the column. The inverse of the contribution of a row is equal to the number of all non-2 elements (each can contribute in covering the row) minus 1. The inverse is well defined because at this stage, each row has at least two elements (it is not essential).

E. New Bounding Criteria

In [11], two new rules to prune the search space have been introduced. We are going to survey them here. Given a covering problem C that corresponds to a node c of the computation tree, define the following notation.

- C_l is the subproblem of C generated assuming that a given branching column b is selected.
- C_r is the subproblem of C generated assuming that a given branching column b is not selected.
- $C.min$ is the cost of a minimum solution.
- $C.lower$ is the value of a lower bound on $C.min$.
- $C.path$ is the cost of the partial solution from the root to node c.
- $C.upper$ is the cost of the best solution found so far.

The algorithm described in Fig. 1 guarantees that the invariant $C.path + C.lower < C.upper$ is always true.

Theorem 2.1 (Left-Hand Side Lower Bound): Given a binate covering problem C, suppose we branch on a unate column b. If

$$C.path + C_l.lower \geq C.upper$$

then both C_l and C_r can be pruned and $C_l.lower$ is a strictly better lower bound for C.

For a proof, see [7] and [26].

The way in which the "old" lower bound and the "new" left-hand side lower bound work together is: if the current node is a left child and *lbound_new* $-$ *Cost*$(b) \geq$ *ubound*, then bound computation and return flag to skip also the right branch ("new" left-hand side lower bound); otherwise, if *lbound_new* $\geq$ *ubound*, then bound computation ("old" lower bound).

Theorem 2.2 (Limit Lower Bound): Given a binate covering problem C, let I be an independent set of the rows, i.e., a set of unate rows intersecting no common column. Let $C.lower$ be a lower bound from the independent set I, i.e., the sum of a minimum cost column for each row in I. Consider the set B of the columns b that do not intersect rows in I and such that $b \in B$ only if

$$C.path + C.lower + Cost(b) \geq C.upper.$$

Then the columns in B and the rows that intersect them in a 0 can be removed from the covering table, and a minimum solution can still be found. For a proof, see [7] and [26].

In practice, in the common case that all columns have cost 1 if included in a solution, one needs only to check whether

$$C.path + C.lower + 1 \geq C.upper$$

i.e.,

$$lbound\_new + 1 \geq C.upper$$

in which case all columns that do not intersect rows in the independent set I can be removed, together with the rows that they intersect in a 0. Experimental results in [11] on exact two-level minimization show strong gains by this new pruning technique, resulting in reductions of the search space up to three orders of magnitude.

III. Reduction Techniques

Three fundamental processes constitute the essence of the reduction rules.

[2] The proof in [8] holds for a class of greedy algorithms parametrized in a positive weighting function α that measures the difficulty of covering an element.

```
sm_reduce(A, solution, weight, ubound) {
    do {
        apply β-dominance or α-dominance
        find essential columns
        find unacceptable columns
        if (a column is both essential and unacceptable)
            return empty_solution
        for each essential column {
            delete each row intersecting the column in a 1
            if (a row of length 1 intersects the column in a 0)
                return empty_solution
            delete column
            add column to solution
            if (cost of solution ≥ ubound)
                return empty_solution
        }
        for each unacceptable column {
            delete each row intersecting the column in a 0
            if (a row of length 1 intersects the column in a 1)
                return empty_solution
            delete column
        }
        apply row_consensus
        apply row_dominance
    } while (reductions are applicable)
    return solution
}
```

Fig. 2. Flow of reduction rules.

1) Selection of a column: a column must be selected if it is the only column that satisfies a required constraint (Section III-G). A dual statement holds for unacceptable columns (Section III-H). Also related is the case of unnecessary columns (Section III-I).
2) Elimination of a column: a column C_i can be eliminated if its elimination does not preclude obtaining a minimal cover, i.e., if there exists in M another column C_j that satisfies at least all the constraints satisfied by C_i (Section III-E).
3) Elimination of a row: a row R_i can be eliminated if there exists in M another row R_j that expresses the same or a stronger constraint (Section III-A).

Even though more complex criteria of dominance have been investigated, the previous ones are basic in any table-covering solver. Reduction rules have previously been stated for the binate covering case [18], [19], [3], [2], and also for the unate covering case [33], [39], [2]. For each of them, we will first define the reduction rule, and then a theorem showing how that rule is applied. Proofs for the correctness of these reduction rules have been given in [18] [19], [3], and [2], and they will not be repeated here.

The effect of reductions depends on the order of their application. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core). Fig. 2 shows how reductions are applied in [38], [36], and [41].

A. Row Dominance

Definition 3.1: A row R_i dominates another row R_j if R_j has all the 1's and 0's of R_i.

Theorem 3.1: If a row R_j is dominated by another row R_i, R_j can be eliminated without affecting the solutions to the covering problem.

B. Row Consensus

Theorem 3.2: If R_i dominates R_j, except for a (unique) column C_k where R_i and R_j have different values, element $M_{j,k}$ can be eliminated from the matrix M (i.e., the entry in position $M_{j,k}$ becomes a 2) without affecting the solutions of the covering problem.

C. Column α-Dominance

Definition 3.2: A column C_i α-dominates another column C_j if

- $k_i \leq k_j$;
- C_i has all the 1's of C_j;
- C_j has all the 0's of C_i.

Theorem 3.3: Let M be satisfiable. If a column C_j is α-dominated by another column C_i, there is at least one minimum cost solution with column C_j eliminated $(x_j = 0)$, together with all the rows in which it has 0's.

In [8], column dominance is formulated in a more general way as follows.

Theorem 3.4: Suppose that v and v' are elements of $\{0, 1\}$. If the clauses satisfied by column C_y set to the value v are satisfied at a lower cost by setting column $C_{y'}$ to v', and the clauses satisfied by $C_{y'}$ set to $\overline{v'}$ are also satisfied at zero cost by C_y set to $\overline{v}$, one can set C_y to $\overline{v}$ and remove the rows that intersect C_y in $\overline{v}$, without missing any optimal solution.

By restriction to negative literals of zero cost and positive literals of positive cost, the criterion reduces to α-dominance.

D. Column β-Dominance

Definition 3.3: A column C_i β-dominates another column C_j if

- $k_i \leq k_j$;
- C_i has all the 1's of C_j;
- for every row R_p in which C_i has a 0, either C_j has a 0 or there exists a row R_q in which C_j has a 0 and C_i does not have a 0, such that disregarding entries in columns C_i and C_j, R_q dominates R_p.

Theorem 3.5: Let M be satisfiable. If C_i β-dominates C_j, there is at least one minimum cost solution with column C_j eliminated $(x_j = 0)$, together with all the rows in which it has 0's.

E. Column Dominance

Definition 3.4: A column C_i dominates another column C_j if either C_i α-dominates C_j or C_i β-dominates C_j.

Theorem 3.6: Let M be satisfiable. If C_i dominates C_j, there is at least one minimum cost solution with column C_j eliminated $(x_j = 0)$, together with all the rows in which it has 0's.

F. Column Mutual Dominance

Definition 3.5: Two columns C_i and C_j mutually dominate each other if

- C_i has a 0 in every row where C_j has a 1;
- C_j has a 0 in every row where C_i has a 1.

Theorem 3.7: Let M be satisfiable. If C_i and C_j mutually dominate each other, there is at least one minimum cost solution with columns C_i and C_j eliminated $(x_i = x_j = 0)$, together with all the rows in which they have 0's.

In [8], column mutual dominance is formulated in a more general way as follows.

Theorem 3.8: Suppose that v and v' are elements of $\{0, 1\}$. Suppose that column C_y has minimum cost when set to v and column $C_{y'}$ has minimum cost when set to v'. If the clauses satisfied by setting column $C_{y'}$ to $\overline{v'}$ are satisfied by setting column C_y to v, and the clauses satisfied by setting C_y to $\overline{v}$ are satisfied by setting $C_{y'}$ to v', then one can set C_y to v, $C_{y'}$ to v' and remove the rows that intersect C_y in v and $C_{y'}$ in v', without missing any optimal solution.

G. Essential Column

Definition 3.6: A column C_j is an essential column if there exists a row R_i having a 1 in column C_j and 2's everywhere else.

Theorem 3.9: If C_j is an essential column, it must be selected $(x_j = 1)$ in every solutions. Column C_j must then be deleted together with all the rows in which it has 1's.

H. Unacceptable Column

Definition 3.7: A column C_j is an unacceptable column if there exists a row R_i having a 0 in column C_j and 2's everywhere else.

This reduction rule is a dual of the essential column rule.

Theorem 3.10: If C_j is an unacceptable column, it must be eliminated $(x_j = 0)$ in every solution, together with all the rows in which it has 0's.

I. Unnecessary Column

Definition 3.8: A column of only 0's and 2's is an unnecessary column.

Notice that there is no symmetric rule for columns of 1's and 2's. The reason is that selecting a column to be in the solution has a cost, while eliminating it has no cost.

Theorem 3.11: If C_j is an unnecessary column, it may be eliminated $(x_j = 0)$, together with all the rows in which it has 0's.

J. Trial Rule

Theorem 3.12: If there exists in a covering table M a row R_i having a 0 in column C_j, a 1 in column C_k, and 2's in the rest, then apply the following test:

- eliminate C_k together with the rows in which it has 0's;
- eliminate C_j, which is now an unacceptable column, together with the rows in which it has 0's;
- continue as long as possible to eliminate the columns which become unacceptable columns.

If at least one row of M has only 2's at the end of this test, then column C_k must be selected[3] $(x_k = 1)$. Therefore, C_k can be deleted together with all of the columns in which it has 1's.

K. Infeasible Subproblem

Unlike the unate covering problem, the binate covering problem may be infeasible. In particular, an intermediate covering matrix M may found to be unsatisfiable by the following theorem. When an infeasible subproblem is found, that branch of the binary recursion is pruned.

Definition 3.9: A covering problem M is infeasible if there exists a column C_j which is both essential and unacceptable (implying $x_j = 1$ and $x_j = 0$).

L. Gimpel's Reduction Step

Another heuristic for solving the minimum cover problem has been suggested by Gimpel [17], [37]. Gimpel proposed a reduction step which simplifies the covering matrix when it has a special form. This simplification is possible without further branching, and hence is useful at each step of the branch-and-bound algorithm. In practice, Gimpel's reduction step is applied after reducing the covering matrix to the cyclic core. An extended presentation can be found in [39]. In [36] and [41], Gimpel's rule has been extended to handle the binate case.

IV. Semi-Implicit Solution of Binate Covering

A. Binary Decision Diagrams

Basic introductions to binary decision diagrams are found in [4] and [1].

Definition 4.1: A *binary decision diagram* (BDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a Boolean variable *var*(v). Vertex v has two outgoing arcs, *child*$_0(v)$ and *child*$_1(v)$. Each terminal vertex u is labeled 0 or 1.

Each vertex in a BDD represents a binary input, binary output function, and all accessible vertices are roots. The terminal vertices represent the constants (functions) 0 and 1. For each nonterminal vertex v representing a function F, its child vertex *child*$_0(v)$ represents the function $F_{\overline{v}}$, and its other child vertex *child*$_1(v)$ represents the function F_v, i.e., $F = \overline{v} \cdot F_{\overline{v}} + v \cdot F_v$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

Definition 4.2: A BDD is *ordered* if there is a total order $\prec$ over the set of variables such that for every nonterminal

[3] It is possible that a row is left with only 2's by a sequence of reduction steps.

vertex v, $var(v) \prec var(child_0(v))$ if $child_0(v)$ is nonterminal, and $var(v) \prec var(child_1(v))$ if $child_1(v)$ is nonterminal.

Definition 4.3: A BDD is *reduced* if

1) it contains no vertex v such that $child_0(v) = child_1(v)$;
2) it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.

Definition 4.4: A *reduced ordered binary decision diagram* (ROBDD) is a BDD which is both reduced and ordered.

Definition 4.5: The ITE operator returns function G_1 if function F evaluates true; else, it returns function G_0:

$$\mathrm{ITE}(F, G_1, G_0) = \begin{cases} G_1, & \text{if } F = 1 \\ G_0, & \text{otherwise} \end{cases}$$

where $\mathrm{range}(F) = \{0, 1\}$.

B. The Shortest Path Method

In [32], the solution of a binate covering problem was reduced to a shortest path computation on the BDD representing the clauses. The constraints are expressed as a product-of-sums (POS), and are represented by a matrix where each row is a clause (i.e., a minterm represented as a path from root to the 1 terminal) and each column is a variable. The attractive feature of a BDD-based algorithm is that finding the solution only requires computing the shortest path to the 1 terminal in the BDD. We will present the theorem supporting the reduction.

Suppose that the length (or cost) of a 0-edge of a BDD is 0 and the length of a 1-edge is a positive constant. A shortest path between two nodes is a path of total minimum length.

Theorem 4.1: A minimum cost assignment satisfying a Boolean formula $T(x_1, \cdots, x_n)$ is given by a shortest path from the root to the terminal 1 of an ROBDD representing T.

C. The Method Based on a Product of BDD's

In [21] and [22], a branch-and-bound algorithm for the binate covering problem expressed as a product of general Boolean formulas and represented by a conjunction of multiple BDD's is presented. Since in cases of practical interest, it happens often that a single BDD representing all clauses is too large to be built, it has been proposed to represent the constraints as a product of subconstraints, each of which can be represented by a BDD. The question is how to find a minimum solution, having a product of BDD's, instead of a single BDD. It is clear that if each subconstraint is a sum-of-products (SOP) clause, the BDD-based formulation is analogous to the one based on a matrix. This motivates the extension to a conjunction of BDD's of the reduction and bounding techniques devised to solve a table.

The algorithm assumes that the constraint function is in the form $f = \Pi_{i=1}^{n} f_i$ where each f_i is represented by a BDD F_i. Each f_i or F_i is called a subconstraint. The conjunction of the F_i is called F. Under this assumption, BCP amounts to finding an assignment for $x_1, x_2, \cdots, x_n$ that minimizes the cost function and that satisfies all f_i's simultaneously. If $n = 1$, we have a single BDD, and the minimum cost assignment that satisfies f can be found by computing the shortest path connecting the root of f to the "1" leaf. If $n > 1$, a branch-and-bound algorithm as in the matrix-based case can be devised. Reduction and bounding techniques are extended as shown next.

A variable x_j is essential for f if and only if $f_i \leq x_j$, for some i, $i = 1, 2, \cdots, n$. A variable x_j is unacceptable for f if and only if $f_i \leq x_j'$, for some i, $i = 1, 2, \cdots, n$.

Row dominance is extended to the more general definition of constraint dominance. Function f_i dominates function f_j if and only if $f_j \leq f_i$. Constraint dominance reduces to row dominance if subconstraints coincide with SOP clauses.

Column dominance is extended to the following definition of variable dominance. Variable x_i dominates variable x_j if and only if $k_i \leq k_j$ and $\exists x_i f_{x_j} \leq \exists x_i f'_{x_j}$. Since the constraint f is in the form of conjunction of subconstraints, the previous definition cannot be checked directly. However, the following sufficient conditions can be checked efficiently. If either of the following conditions is satisfied:

- $(f_k)_{x_j} \leq (f_k)_{x_i x'_j}$ for each f_k
- $(f_k)_{x_j} \leq (f_k)_{x'_i x'_j}$ for each f_k

where $k_i \leq k_j$, then x_i dominates x_j. As another special case, if $(f_k)_{x_j} \leq (f_k)_{x'_j}$ for each f_k, then any variable $x_i, (i \neq j)$ dominates variable x_j.

When x_j has cost 0, a more general definition of variable dominance is that variable x_i dominates variable x_j if and only if $\exists x_i f_{x_j} \leq \exists x_i f'_{x_j}$ or $\exists x_i f'_{x_j} \leq \exists x_i f_{x_j}$.

In [21], variable x_i is said to dominate variable x_j if and only if $k_i \leq k_j$ and one of the following conditions is satisfied $\forall k \in \{1, \cdots, n\}$:

1) $(f_k)_{x_j} \leq (f_k)_{x_i x'_j}$
2) $(f_k)_{x_j} = (f_k)_{x_j \prime} = (f_k)$, i.e., f_k does not depend on x_j, and there exists a p such that $(f_p)_{x'_i x_j} \leq (f_k)_{x_i}$.

If subconstraints coincide with SOP clauses, the first condition gives the definition of α-dominance. If subconstraints coincide with SOP clauses, the first and second conditions together give the definition of β-dominance.

A lower bound to the cost of satisfying F is given by the sum of the minimum costs of satisfying each BDD in a set of BDD's with disjoint supports (an independent set of BDD's). These minimum costs can be found by computing the shortest paths of those BDD's. If the shortest paths satisfy all of the other subconstraints, the solution determined by the independent set is optimal, and the current recursion node can be pruned.

A most common variable in the BDD's is chosen as a splitting variable (i.e., a variable whose corresponding column in the dependence matrix intersects most rows). This favors the simplification of as many BDD's as possible, the partitioning of the BDD's in sets with disjoint support, and the generation of larger independent sets. Experiments show that this splitting variable criterion is less effective that the one (in Section II-D) used for a matrix-based formulation, and as a consequence, the number of recursion nodes is greater.

We notice that in both approaches presented in this section, the usage of BDD's potentially allows us to handle problems with many clauses (if they have a compact BDD representa-

tion), but does not address the problem of covering matrices with many columns. In such problems, it is unlikely that the BDD can be built at all because each column is a variable in the support of the BDD.

It may be worthy of mention at this point that in [27] and [28], a more general algorithm to solve integer linear programming based on edge-valued binary decision diagrams has been presented.

V. Implicit Solution of Binate Covering

The classical branch-and-bound algorithm [18], [19] for minimum cost binate covering has been described in previous sections, and implemented by means of efficient computer programs (ESPRESSO and STAMINA). These state-of-the-art binate table solvers represent binate tables efficiently using sparse matrix packages. But the fact that each nonempty table entry still has to be explicitly represented puts a bound on the size of the tables that can be handled by these binate solvers. For example, one would not expect these binate solvers to handle examples requiring over 10^6 columns (up to 2^{1500} columns), reported in state minimization of FSM's [26]. To keep with our stated objective, the binate table has to be represented implicitly. We do not represent (even implicitly) the elements of the table, but we make use only of a set of row labels and a set of column labels, each represented implicitly as a BDD. They are chosen so that the existence and value of any table entry can be readily inferred by examining its corresponding row and column labels. In the sequel, we shall assume that every row has a unit cost.

A. Implicit Set Manipulation

In [26], it is shown how to represent and manipulate sets and sets of sets with BDD's.

Given a ground set G of cardinality less or equal to n, any *subset* S can be represented in a Boolean space B^n by a unique Boolean function $\chi_S: B^n \rightarrow B$, which is called its *characteristic function* [5], such that

$$\chi_S(x) = 1, \quad \text{if and only if } x \text{ is in } S.$$

Alternatively, a subset can also be represented in *positional-set* or *positional-cube* notation form,[4] using n Boolean variables, $x = x_1 x_2 \cdots x_n$. The presence of an element s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional set, whereas x_k takes the value 0 if element s_k is not a member of the set. One Boolean variable is needed for each element because the element can either be present or absent in the set. As an example, for $n = 6$, the set with a single element s_4 is represented by 000 100, and the set $s_2 s_3 s_5$ is represented by 011 010. The elements s_1, s_4, s_6 which are not present correspond to 0's in the positional set.

A set of subsets of G can be represented by a Boolean function, whose minterms correspond to the single subsets. In other words, a set of sets is represented as a set S of positional sets by a characteristic function $\chi_S: B^n \rightarrow B$ as

$$\chi_S(x) = 1, \quad \text{if and only if the set represented by the positional set } x \text{ is in the set } S \text{ of sets.}$$

Any *relation* $\mathcal{R}$ between a pair of Boolean variables can also be represented by a characteristic function $\mathcal{R}: B^2 \rightarrow B$ as

$$\mathcal{R}(x, y) = 1, \quad \text{if and only if } x \text{ is in relation } \mathcal{R} \text{ to } y.$$

$\mathcal{R}$ can be a one-to-many relation over the two sets in B. These definitions can be extended to any relation $\mathcal{R}$ between n Boolean variables, and can be represented by a characteristic function $\mathcal{R}: B^n \rightarrow B$ as

$$\mathcal{R}(x_1, x_2, \cdots, x_n) = 1, \quad \text{if and only if the } n\text{-tuple } (x_1, x_2, \ldots, x_n) \text{ is in relation } \mathcal{R}.$$

In this way, useful relational operators on sets can be derived. Operator Op acts on two sets of variables $x = x_1 x_2 \cdots x_n$ and $y = y_1 y_2 \cdots y_n$ and returns a relation $(x\ Op\ y)$ (as a characteristic function) of pairs of positional sets. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of objects, x and y. For example, given two sets of sets X and Y, the set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$. A detailed list of these operators is presented in [26].

B. Setting of Implicit Solution

A binate covering problem instance can be characterized by a 6-tuple $(r, c, R, C, 0, 1)$, defined as follows:

- the group of variables for labeling the rows: r;
- the group of variables for labeling the columns: c;
- the set of row labels: $R(r)$;
- the set of column labels: $C(c)$;
- the 0-entries relation at the intersection of row r and column c: $0(r, c)$;
- the 1-entries relation at the intersection of row r and column c: $1(r, c)$.

In other words, the user of our implicit binate solver would first choose an encoding for the rows and columns. Given a binate table, the user will then supply a set of row labels as a BDD $R(r)$ and a set of column labels as a BDD $C(c)$, and also the two inference rules in the form of BDD relations, $0(r, c)$ and $1(r, c)$, capturing the 0-entries and 1-entries.

The classical branch-and-bound solution of minimum cost binate covering is based on the recursive procedure as shown in Fig. 1. In our implicit formulation, we keep the standard branch-and-bound scheme, but we replace the traditional description of the table as a (sparse) matrix with an implicit representation, using BDD's for the characteristic functions of the rows and columns of the table. Moreover, we have implicit versions of the manipulations of the binate table required to implement the branch-and-bound scheme, namely, we perform implicitly table reduction, branching column selection, computation of the lower bound, and table partitioning.

At each call of the binate cover routine *mincov*, the binate table undergoes a reduction step *Reduce* and, if termination

[4] Called also *1-hot encoding*.

conditions are not met, a branching column is selected and *mincov* is called recursively twice, once assuming the selected column c_i in the solution set (on the table R_{c_i}, C_{c_i}), and once out of the solution set (on the table $R_{\overline{c_i}}, C_{\overline{c_i}}$). Some suboptimal solutions are bounded away by computing a lower bound L on the current partial solution and comparing it against an upper bound U (best solution obtained so far). A good lower bound is based on the computation of a maximal independent set.

C. Implicit Table Generation

Here, we define different ways of specifying the binate covering table in decreasing order of generality of the binate covering problem. A table is defined implicitly by generating BDD-based representations of the rows and columns and by giving relations specifying the 1 and 0 entries, given the rows and columns. By imposing restrictions on the way in which rows and columns are labeled and entries are defined, one gets representations with varying degrees of generality. We distinguish between 1) the case of a general binate covering table and 2) a specialized representation, sufficient to solve tables for exact state minimization of ISFSM's [23] (applicable to problems with similar covering tables, e.g., technology mapping for area minimization [39]). There is a tradeoff between generality of the representation and efficiency of the computations: "hard wiring" the rules that define a table may speed up table manipulations, at the price of more limited applicability.

1) General binate covering table
   - the group of variables for labeling the rows: r;
   - the group of variables for labeling the columns: c;
   - the set of row labels: $R(r)$;
   - the set of column labels: $C(c)$;
   - the 0-entries relation at the intersection of row r and column c: $\mathtt{0}(r, c)$;
   - the 1-entries relation at the intersection of row r and column c: $\mathtt{1}(r, c)$.
2) Specialized binate covering table for exact state minimization and similar problems:
   - the group of variables for labeling the rows (each label is a pair): (c, d);
   - the group of variables for labeling the columns: p;
   - the set of row labels: $R(c, d)$;
   - the set of column labels: $C(p)$;
   - the 0-entries relation at the intersection of row (c, d) and column p: $\mathtt{0}((c, d), p) = (p = c)$;
   - the 1-entries relation at the intersection of row (c, d) and column p: $\mathtt{1}((c, d), p) = (p \supseteq d)$.

As an example, for the problem of exact state minimization, $C(p)$ is the set of labels that denote the prime compatibles p of an FSM, i.e., p is in set C if it is the label of a prime compatible p. Prime compatibles are sets of states and they are represented using positional set notation. For instance, if an FSM has five states $s1, s2, s3, s4, s5$ and $p = \{s1, s4\}$ is a compatible, set C is represented with five Boolean variables, and p is labeled as 10 010. $R(c, d)$ is the relation expressing covering clauses and closure clauses. A covering clause for a state says that the state must be contained in at least one prime compatible. A binate clause for a compatible says that if the compatible is chosen in a solution, then at least another compatible from a related set must be in that solution, e.g., clause $(\overline{p} + p_1 + p_2 + \cdots + p_k)$, meaning that if p is in a solution, either one of $p_1, p_2, \ldots, p_k$ must be in that solution. A covering clause yields a unate row, labeled by a c part that denotes an empty set and by a d part that denotes a singleton set, requiring that a given state be covered. Whenever $p \supseteq d$, there is a 1 at the intersection of the row labeled by d and the column representing prime compatible p, meaning that the compatible p contains state d. A closure clause yields a binate row, labeled by a c part that is the label of the unique prime compatible whose corresponding column has a zero at the intersection with this row (condition $p = c$), and by a d part that is the label of a compatible such that there is a 1 at the intersection of this row and any column whose label p is a prime compatible that contains compatible d. We refer to [24] for a complete treatment of implicit state minimization of incompletely specified FSM's.

If the covering problem is unate, the $\mathtt{0}(r, c)$ relation is empty. A typical example is exact two-level minimization where $R(r) = R(m)$, for m labeling minterms, $C(c) = C(p)$, for p labeling prime implicants, and $\mathtt{1}(r, c) = (p \supseteq m)$. The label of an implicant can be constructed by representing each Boolean variable in multivalued notation, for instance, encoding 0 as 10, 1 as 01, and $-$ as 11. A complete treatment of this special case can be found in [42] and [7]. The more complex case of implicit exact minimization of generalized prime implicants is described in [44].

In the next section, we will describe how a binate covering table can be manipulated implicitly so as to solve the minimum cost binate covering problem.

VI. Implicit Table Reduction Techniques

Reduction rules aim at the following.

1) Selection of a column. A column must be selected if it is the only column that satisfies a given row. A dual statement holds for columns that must not be part of the solution in order to satisfy a given row.
2) Elimination of a column. A column c_i can be eliminated if its elimination does not preclude obtaining a minimum cover, i.e., if there is another column c_j that satisfies at least all the rows satisfied by c_i.
3) Elimination of a row. A row r_i can be eliminated if there exists another row r_j that expresses the same or a stronger constraint.

The order of the reductions affects the final result. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core).

In the reduction, there are two cases when no solution is generated.

1) The added cardinality of the set of essential columns, and of the partial solution computed so far, *Sol*, is larger than or equal to the upper bound U. In this case, a better solution is known than the one that can be found from

TABLE I
BINATE COVERING TABLE WITH CONSTRAINT $(x_{AB} + x_A)(x_{AB} + x_{BC} + x_B)(x_{BC} + x_{CD})(x_{CD})(\overline{x_{AB}} + x_{CD})(\overline{x_{BC}} + x_{CD})$ GENERATED FROM AN ISFSM STATE MINIMIZATION EXAMPLE

	(c, d)	AB 1100	BC 0110	CD 0011	A 1000	B 0100
A	0000 1000	1			1	
B	0000 0100	1	1			1
C	0000 0010		1	1		
D	0000 0001			1		.
$BC \Rightarrow CD$	0110 0011		0	1		
$AB \Rightarrow CD$	1100 0011	0		1		

TABLE II
TABLE AFTER DELETING THE ESSENTIAL COLUMN LABELED 0011 FROM TABLE I

		AB 1100	BC 0110	A 1000	B 0100
A	0000 1000	1		1	
B	0000 0100	1	1		1

TABLE III
TABLE AFTER DELETING DOMINATED COLUMNS FROM TABLE II

		AB 1100
A	0000 1000	1
B	0000 0100	1

now on, and so the current computation branch can be bounded away.

2) After having eliminated essential, unacceptable, and unnecessary columns and covered rows, it may happen that the rest of the rows cannot be covered by the remaining columns. In this case, the current partial solution cannot be extended to any full solution.

In the rest of the section, we will describe how reduction operations are performed implicitly on the general binate covering table. Proofs of each proposition can be found in [26], together with other equations for reducing the general binate covering table and for the specialized binate covering table. A simple example of implicit reduction is shown in Tables I–III.

A. Essential Columns

Definition 6.1: A column c is an *essential column* if there is a row having a 1 in column c and 2 everywhere else.

Proposition 6.1: The set of essential columns can be computed by

$$ess\_col(c) = C(c) \cdot \exists r\{R(r) \cdot 1(r,c) \cdot \nexists c'[C(c') \cdot (c' \neq c) \cdot (0(r,c') + 1(r,c'))]\}.$$

Essential columns must added to the solution. Each essential column must then be deleted from the table together with all rows where it has 1's.

$$\begin{aligned} solution(c) &= solution(c) + ess\_col(c) \\ C(c) &= C(c) \cdot \neg ess\_col(c) \\ R(r) &= R(r) \cdot \nexists c[ess\_col(c) \cdot 1(r,c)], \end{aligned}$$

B. Column Dominance

Some columns need not be considered in a binate table, if they are dominated by others. Classically, there are two notions of column dominance: α-dominance and β-dominance.

Definition 6.2: A column c' *α-dominates* another column c if c' has all the 1's of c, and c has all the 0's of c'.

Proposition 6.2: The α-dominance relation can be computed as

$$\alpha\_dom(c',c) = \nexists r\{R(r) \cdot [1(r,c) \cdot \neg 1(r,c') + 0(r,c') \cdot \neg 0(r,c)]\}.$$

Definition 6.3: A column c' *β-dominates* another column c if: 1) c' has all the 1's of c, and 2) for every row r' in which c' contains a 0, there exists another row r in which c has a 0 such that, disregarding entries in column c', r' has all the 1's of r.

Proposition 6.3: The β-dominance relation can be computed by

$$\beta\_dom(c',c) = \nexists r'\{R(r') \cdot [1(r',c) \cdot \neg 1(r',c') ; + 0(r',c') \cdot \nexists r[R(r) \cdot 0(r,c) \cdot \nexists c''[C(c'') \cdot (c'' \neq c') \cdot 1(r,c'') \cdot \neg 1(r',c'')]]]\}.$$

The conditions for α-dominance are a strict subset of those for β-dominance, but α-dominance is easier to compute implicitly. Either of them can be used as the column dominance relation *col_dom*.

Proposition 6.4: The set of dominated columns in a table (R, C) can be computed as

$$D(c) = C(c) \cdot \exists c'[C(c') \cdot (c' \neq c) \cdot col\_dom(c',c)].$$

Proposition 6.5: The following computations delete a set of columns $D(c)$ from a table (R, C) and all rows intersecting these columns in a 0

$$\begin{aligned} C(c) &= C(c) \cdot \neg D(c) \\ R(r) &= R(r) \cdot \nexists c[D(c) \cdot 0(r,c)]. \end{aligned}$$

C. Row Dominance

Definition 6.4: A row r' *dominates* another row r if r has all the 1's and 0's of r'.

Proposition 6.6: The row dominance relation can be computed by

$$row\_dom(r',r) = \nexists c\{C(c) \cdot [1(r',c) \cdot \neg 1(r,c) + 0(r',c) \cdot \neg 0(r,c)]\}.$$

Proposition 6.7: The set of rows not dominated by other rows can be computed as

$$R(r) = R(r) \cdot \nexists r'[R(r') \cdot (r' \neq r) \cdot row\_dom(r',r)].$$

VII. OTHER IMPLICIT TABLE MANIPULATIONS

To have a fully implicit binate covering algorithm as described in Section V, we must also compute implicitly a branching column and a lower bound. These computations as well as table partitioning involve solving a common subproblem of finding columns in a table which have the maximum number of 1's.

```
Lmax(F, r) {
    v = bdd_top_var(F)
    if (v ∈ r)
        return (1, bdd_count_onset(F))
    else { /* v is a c variable */
        (T, count_T) = Lmax(bdd_then(F), r)
        (E, count_E) = Lmax(bdd_else(F), r)
        count = max(count_T, count_E)
        if (count_T = count_E)
            G = ITE(v, T, E)
        else if (count = count_T)
            G = ITE(v, T, 0)
        else if (count = count_E)
            G = ITE(v, 0, E)
        return (G, count)
    }
}
```

Fig. 3. Pseudocode for the $Lmax$ operator.

A. Selection of Columns with Maximum Number of 1's

Given a binary relation $F(r, c)$ as a BDD, the abstracted problem is to find a subset of c's, each of which relates to the maximum number of r's in $F(r, c)$. An inefficient method is to cofactor F with respect to c taking each possible value c_i, count the number of onset minterms of each $F(r, c)|_{c=c_i}$, and pick the c_i's with the maximum count. Instead, our algorithm, $Lmax$, traverses each node of F exactly once as shown by the pseudocode in Fig. 3.

$Lmax$ takes a relation $F(r, c)$ and the variable set r as arguments, and returns the set G of c's which are related to the maximum number of r's in F, together with the maximum count. Variables in c are required to be ordered before variables in r. Starting from the root of BDD F, the algorithm traverses down the graph by recursively calling $Lmax$ on its *then* and *else* subgraphs. This recursion stops when the top variable v of F is within the variable set r. In this case, the BDD rooted at v corresponds to a cofactor $F(r, c)|_{c=c_i}$ for some c_i. The minterms in its onset are counted and returned as *count*, which is the number of r's that are related to c_i.

During the upward traversal of F, we construct a new BDD G in a bottom-up fashion, representing the set of c's with maximum count. The two recursive calls of $Lmax$ return the sets $T(c)$ and $E(c)$ with maximum counts *count_T* and *count_E* for the $then$ and the $else$ subgraphs. The larger of the two counts is returned. If the two counts are the same, the columns in T and E are merged by $ITE(v, T, E)$ and returned. If *count_T* is larger, only T is retained as the updated columns of maximum count, and symmetrically for the other case. To guarantee that each node of BDD $F(r, c)$ is traversed once, the results of $Lmax$ and *bdd_count_onset* are memorized in computed tables. Note that $Lmax$ returns a set of c's of maximum count. If we need only one c, some heuristic can be used to break the ties.

Example 7.1: To understand how $Lmax$ works, consider the explicit binate table

	00	01	10	11
00	1	2	1	1
01	2	1	1	2
10	2	1	2	1
11	2	1	2	1

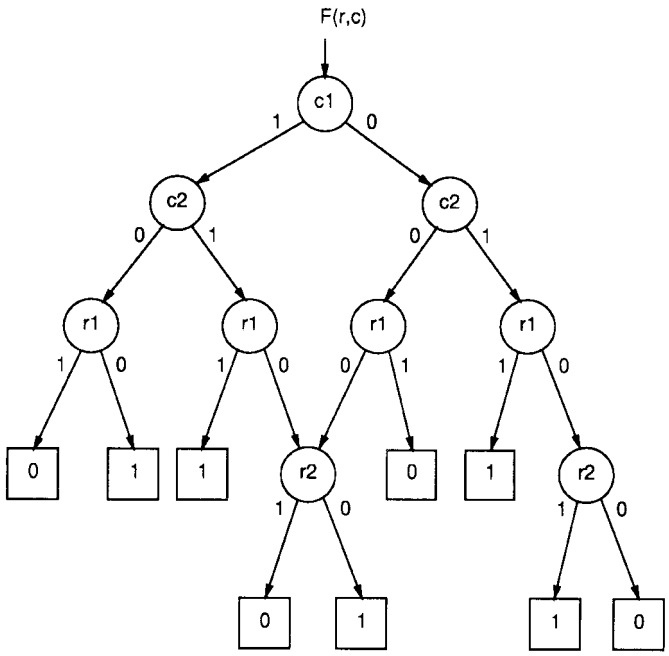

Fig. 4. BDD of $F(r, c)$ to illustrate the routine $Lmax$.

with four rows and four columns. The columns that maximize the number of 1's are the second and the fourth. If the rows and columns are encoded by two Boolean variables each, using the encodings given at the top of each column and to the left of each row, the 1 entries of the table are represented implicitly by the relation $F(c, r)$[5] whose minterms are

$$\{0000, 1000, 1100, 0101, 1001, 0110, 1110, 0111, 1111\}.$$

The BDD representing F is shown in Fig. 4. The result of invoking $Lmax$ on $F(r, c)$ is a BDD representing the relation $G(c)$ whose minterms are $\{01, 11\}$, corresponding to the encodings of the second and fourth columns.

B. Implicit Selection of a Branching Column

The selection of a branching column is a key ingredient of an efficient branch-and-bound covering algorithm. A good choice reduces the number of recursive calls, by helping to discover more quickly a good solution. We adopt a simplified selection criterion: select a column with a maximum number of 1's. By defining $F'(r, c) = R(r) \cdot C(c) \cdot 1(r, c)$ which evaluates true if and only if table entry (r, c) is a 1, our column selection problem reduces to one of finding the c related to the maximum number of r's in the relation $F'(r, c)$, and so it can be found implicitly by calling $Lmax(F', r)$. A more refined strategy is to restrict our selection of a branching column to columns intersecting rows of a maximal independent set because a unique column must eventually be selected from each independent row. A maximal independent set can be computed as follows.

[5] r and c are swapped in F so that minterms are listed in the order of the BDD variables.

TABLE IV
RANDOM FSM's

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
ex2.271	95323 x 96382	0 x 0	1	1	-	-	2	2	-	-	1	55	fails	fails
ex2.285	1 x 121500	0 x 0	1	1	-	-	2	2	-	-	0	0	fails	fails
ex2.304	1053189 x 264079	1052007 x 264079	2	-	-	-	2	-	-	-	463	fails	fails	fails
ex2.423	637916 x 160494	636777 x 160494	*2	-	-	-	*3	-	-	-	*341	fails	fails	fails
ex2.680	757755 x 192803	756940 x 192803	2	-	-	-	2	-	-	-	833	fails	fails	fails

C. Implicit Selection of a Maximal Independent Set of Rows

Usually, a lower bound is obtained by computing a maximum independent set of the unate rows. A maximum independent set of rows is a (maximum) set of rows, no two of which intersect the same column at a 1. Maximum independent set is an NP-hard problem, and an approximate one (only maximal) can be computed by a greedy algorithm. The strategy is to select *short unate rows* from the table, so we construct a relation $F''(c,r) = R(r) \cdot \mathit{unate\_row}(r) \cdot C(c) \cdot 1(r,c)$. Variables in r are ordered *before* those in c. The rows with the minimum number of 1's in F'' can be computed by $Lmin(F'', c)$, by replacing in $Lmax$ the expression $\max(\mathit{count\_T}, \mathit{count\_E})$ with $\min(\mathit{count\_T}, \mathit{count\_E})$. Once a shortest row, $\mathit{shortest}(r)$, is selected, all rows having 1-elements in common with $\mathit{shortest}(r)$ are discarded from $F''(c,r)$ by

$$F''(c,r) = F''(c,r) \cdot \not\exists c'\{\exists r'[\mathit{shortest}(r') \cdot F''(c',r')] \cdot F''(c',r)\}.$$

Another shortest row can then be extracted from the remaining table F'', and so on, until F'' becomes empty. The maximum independent set consists of all rows $\mathit{shortest}(r)$ so selected.

VIII. Experimental Results of Implicit Binate Covering

We implemented a specialized solver where the table is specified as in case 2) of Section V-C ("specialized binate covering table for exact state minimization and similar problems"), and we applied it to the problem of exact state minimization of incompletely specified FSM's (ISFSM's).

We also implemented a more general solver that is a variant of case 1) of Section V-C ("general binate covering table"). It works with relations $0(r,c)$ and $1(r,c)$ to determine 0 entries and 1 entries, but it makes the assumption that each row of the binate covering table has at most one 0 in order to simplify the table reduction operations.[6] We applied it to the problem of exact state minimization of pseudodeterministic FSM's [25]. The same binate solver was also applied to the problem of selection of generalized prime implicants [44].

In this section, we report results of two applications of the previous implicit binate covering algorithms. We will concentrate on the experimental performance of binate covering, referring to the original papers for a full-fledged description of the specific applications.

[6] See [26] for a more detailed description of variant formulations of the binate covering table in the implicit setting.

A. State Minimization of ISFSM's

Here, we provide data for a subset of them, sufficient to characterize the capabilities of our prototype program. Comparisons of our program ISM are made with STAMINA. The binate covering step of STAMINA was run with no row consensus, because row consensus has not been implemented in our implicit binate solver. Our implicit binate program does not feature Gimpel's reduction rule, that was instead invoked in the version of STAMINA used for comparison. This might sometimes favor STAMINA, but for simplicity, we will not elaborate further on this effect. Missing from our package is also table partitioning. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mbytes of memory.

The following explanations refer to the tables of results.

- Under "table size," we provide the dimensions of the original binate table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- "# mincov" is the number of recursive calls of the binate cover routine.
- "α" and "β" mean, respectively, α and β dominance.
- Data are reported with a "$*$" in front when only the first solution was computed.
- Data are reported with a "$\dagger$" in front when only the first table reduction was performed.
- "# cover" is the cardinality of a minimum cost solution (when only the first solution has been computed, it is the cardinality of the first solution).
- "CPU time" refers only to the binate covering algorithm. It does not include the time to find the prime compatibles.

Table IV presents a few randomly generated FSM's. They generate giant binate tables. The experiments show that ISM is capable of reducing those tables, and of producing a minimum solution or at least a solution. This is beyond the reach of an explicit technique, and substantiates the claim that implicit techniques advance decisively the size of instances that can be solved exactly.

Examples in Table IV demonstrate dramatically the capability of implicit techniques to build and solve huge binate covering problems on suites of contrived examples. Do similar cases arise in real synthesis applications? The examples reported in Table V answer the question in the affirmative. They are from the suite of FSM's described in [35]. It is not possible to build and solve these binate tables with explicit techniques. Instead, we can manipulate

TABLE V
LEARNING I/O SEQUENCES BENCHMARK

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
threer.20	6977 x 3936	6974 x 3936	*4	*6	*5	*3	*5	*5	*6	*6	*13	*26	*1996	*677
threer.25	35690 x 17372	34707 x 17016	*3	*6	-	-	*5	*6	-	-	*69	*192	fails	fails
threer.30	68007 x 33064	64311 x 32614	*4	*9	-	-	*8	*8	-	-	*526	*770	fails	fails
threer.35	177124 x 82776	165967 x 82038	*8	*9	-	-	*12	*10	-	-	*2296	*2908	fails	fails
threer.40	1209783 x 529420	1148715 x 526753	*8	-	-	-	*12	-	-	-	*6787	fails	fails	fails
fourr.16	6060 x 3266	5235 x 3162	*2	*3	*3	*3	*3	*3	*4	*4	*6	*23	*1641	*513
fourr.16	6060 x 3266	5235 x 3162	*2	623	*3	377	*3	3	*4	3	*6	9194	*1641	1459
fourr.20	26905 x 12762	26904 x 12762	*2	*4	-	-	*4	*4	-	-	*31	*68	fails	fails
fourr.30	1396435 x 542608	1385809 x 542132	*2	*5	-	-	*4	*5	-	-	*1230	*1279	fails	fails
fourr.40	6.783e9 x 2.388e9	6.783e9 x 2.388e9	†1	-	-	-	†-	-	-	-	†723	fails	fails	fails

TABLE VI
EXAMPLES FROM SYNTHESIS OF INTERACTING FSM'S

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
ifsm1	17663 x 8925	16764 x 8829	*4	2	*10	3	*14	14	*15	14	*388	864	*17582	805
ifsm1	17663 x 8925	16764 x 8829	*4	2	24	3	*14	14	14	14	*388	864	40817	805
ifsm2	1505 x 774	1368 x 672	4	3	41	44	9	9	9	9	136	230	49	3

them with our implicit binate solver and find a solution. In the example *fourr.40*, only the first table reduction was performed.

Table VI reports FSM's expressing the permissible behaviors in the synthesis of interacting FSM's. Prime compatibles are required only for the state minimization of $ifsm1$ and $ifsm2$. For $ifsm1$, ISM can find a first solution faster than STAMINA using α-dominance. But as the table sizes are not very big, the run times that ISM takes are usually longer than those for STAMINA.

B. Selection of Generalized Prime Implicants

Table VII reports the results of running our program ISA to select a minimal encodable cover of generalized prime implicants (GPI's). GPI's are an extension of the concept of prime implicants to the case of multivalued input and multivalued output Boolean functions. An encodable selection of GPI's translates into a two-valued implementation of the same size. Details can be found in [15] and [44]. For these experiments, ISA has been run with option $-m$, which computes a subset of the GPI's, to generate smaller tables. The tables provide the following information.

- Under the column "table size," we provide the dimensions of the original table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- The column "mincov calls" is the number of recursive calls of the implicit table solver.
- The column "table sol." is the cardinality of the cover of GPI's returned by the table solver.
- The column "CPU time table red." gives the time for the binate table solver. The time to compute the prime compatibles is not included.

TABLE VII
SELECTION OF A MINIMAL ENCODABLE GPI COVER

FSM	table size (row x col) before red.	table size (row x col) after red.	mincov calls	table sol.	CPU time (sec.) table red.
bbsse	3480 x 34727	-(a)	-	-	timeout
cf	30208 x 102781	-(b)	-	-	-
chanstb	169216 x 525	0 x 0	1	11	1218
cpab	208896 x 1892	683 x 73	4	8	7774
cse	2588 x 21798	0 x 0	1	23	6534
dk512	43 x 1777	0 x 0	1	6	4150
ex2	86 x 38410	0 x 0	1	3	830
ex4	1072 x 26759	0 x 0	1	10	803
fstate	5360 x 1605	11 x 11	2	8	12770
keyb	2666 x 361240	0 x 0	1	8	1706
kirkman	100252 x 1081088	-(a)	-	-	timeout
maincont	67586 x 245784	0 x 0	1	4	115
mark1	1936 x 50258	5 x 5	3	7	1313
modulo12	24 x 9039	24 x 36	17	2	50
pkheader	140288 x 29099	0 x 0	1	19	5850
ricks	31232 x 16561	14 x 14	18	27	3301
s1	15336 x 586240	-(b)	-	-	-
s1a	5120 x 586240	-(b)	-	-	-
saucier	18496 x 7106239	0 x 0	1	15	6802
scud	2966 x 2533	0 x 0	1	57	15633
slave	2207744 x 16845	-(a)	-	-	timeout
tma	2028 x 287558	-(b)	-	-	-
train11	43 x 583	0 x 0	1	2	177

(a) timeout 18000 in collapse columns
(b) out-of-memory in collapse columns

The part of ISA that computes an encodable cover of GPI's and gets the codes by a second call to an implicit table solver is not reported here.

ISA fails to complete some examples due to time-out or no more memory in the collapse column step of the first table reduction. For instance, ISA fails to complete the first table reduction of *slave* for time-out at 18 000 s, during the step of collapse columns.

FSM's *cse, dk512, keyb, ex2, maincont, pkheader, mark1* were run on a DEC 7000 Model 610 AXP with 1 Gbyte of memory. There is no program against which to compare.

We underline that the covering problems faced to select covers of GPI's, even though they are unate, are often harder than those encountered to select covers of prime implicants in the ESPRESSO benchmark [42], [7], a reason being the larger variable support of the BDD representations of columns and rows. To be able to solve the examples of the previous tables, the package described in [24] had to be further optimized, and inadequacies still remain to be addressed.

IX. Conclusion

Binate covering is a very useful paradigm for a host of optimization problems in automatic design. We have presented the main features of explicit and implicit algorithms to find an exact solution. The former solve routinely small and medium instances. The latter may solve large instances, but we cannot analyze *a priori* when they will succeed because it is difficult to determine tight bounds on the size of ROBDD representations.

Research is still in progress to improve the lower bound and column selection procedures to generate fewer subproblems. On the front of implicit techniques, work still must be done to make the implicit procedures more robust. When applicable, quantifier-free implicit table reductions are surely a useful tool as shown for unate covering of two-level exact minimization [7].

References

[1] K. Brace, R. Rudell, and R. Bryant, "Efficient implementation of a BDD package," in *Proc. Design Automation Conf.*, June 1990, pp. 40–45.

[2] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and R. Rudell, *Multi-Level Logic Synthesis*, unpublished book, 1992.

[3] R. Brayton and F. Somenzi, "An exact minimizer for Boolean relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1989, pp. 316–319.

[4] R. Bryant, "Graph based algorithm for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 667–691, Aug. 1986.

[5] E. Cerny, "Characteristic functions in multivalued logic systems," *Digital Processes*, vol. 6, pp. 167–174, June 1980.

[6] T. H. Cormen, C. E. Leiserson, and R. l. Rivest, *Introduction to Algorithms*. MIT Press, McGraw-Hill, 1990.

[7] O. Coudert, "Two-level logic minimization: An overview," *Integration*, vol. 17, pp. 97–140, Oct. 1994.

[8] ——, "On solving binate covering problems," in *Proc. Design Automation Conf.*, June 1996, pp. 197–202.

[9] O. Coudert, C. Berthet, and J. C. Madre, "Verification of sequential machines using functional Boolean vectors," in *Proc. IFIP Int. Workshop, Applied Formal Methods for Correct VLSI Design*, Nov. 1989.

[10] O. Coudert and J. C. Madre, "Implicit and incremental computation of prime and essential prime implicants of Boolean functions," in *Proc. Design Automation Conf.*, June 1992, pp. 36–39.

[11] ——, "New ideas for solving covering problems," in *Proc. Design Automation Conf.*, June 1995, pp. 641–646.

[12] O. Coudert, J. C. Madre, and H. Fraisse, "A new viewpoint on two-level logic minimization," in *Proc. Design Automation Conf.*, June 1993, pp. 625–630.

[13] O. Coudert, J. C. Madre, H. Fraisse, and H. Touati, "Implicit prime cover computation: An overview," in *Proc. SASIMI Conf.*, 1993, pp. 413–422.

[14] O. Coudert and C.-J. Shi, "Ze-Dicho: An exact solver for dichotomy-based constrained encoding" in *Proc. Int. Conf. Comput. Design*, 1996, pp. 426–431.

[15] S. Devadas and R. Newton, "Exact algorithms for output encoding, state assignment and four-level Boolean minimization," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 13–27, Jan. 1991.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: Freeman, 1979.

[17] J. Gimpel, "A reduction technique for prime implicant tables," *IRE Trans. Electron. Comput.*, vol. EC-14, pp. 535–541, Aug. 1965.

[18] A. Grasselli and F. Luccio, "A method for minimizing the number of internal states in incompletely specified sequential networks," *IRE Trans. Electron. Comput.*, vol. EC-14, pp. 350–359, June 1965.

[19] ——, "Some covering problems in switching theory," in *Networks and Switching Theory*. New York: Academic, 1968, pp. 536–557.

[20] H. Higuchi and Y. Matsunaga, "Implicit prime compatible generation for minimizing incompletely specified finite state machines," in *Proc. Asia and South Pacific Design Automation Conf.*, Sept. 1995, pp. 229–234.

[21] S.-W. Jeong and F. Somenzi, "A new algorithm for 0–1 programming based on binary decision diagrams," in *Proc. ISKIT-92, Int. Symp. Logic Synthesis and Microprocessor Architecture*, Iizuka, Japan, July 1992, pp. 177–184.

[22] ——, "A new algorithm for the binate covering problem and its application to the minimization of boolean relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 417–420.

[23] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A fully implicit algorithm for exact state minimization," Tech. Rep. UCB/ERL M93/79, Nov. 1993.

[24] ——, "A fully implicit algorithm for exact state minimization," in *Proc. Design Automation Conf.*, June 1994, pp. 684–690.

[25] ——, "Implicit state minimization of nondeterministic FSM's," in *Proc. Int. Conf. Computer Design*, Oct. 1995, pp. 250–257.

[26] ——, *Synthesis of FSMs: Functional Optimization*. Boston, MA: Kluwer Academic, 1996.

[27] Y.-T. Lai, M. Pedram, and S. B. K. Vrudhula, "FGILP: An integer linear program solver based on function graphs," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 685–689.

[28] ——, "EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition," *IEEE Trans. Computer-Aided Design*, vol. CAD–13, pp. 959–975, Aug. 1994.

[29] L. Lavagno, "Heuristic and exact methods for binate covering," EE290ls Rep., May 1989.

[30] B. Lin, O. Coudert, and J. C. Madre, "Symbolic prime generation for multiple-valued functions," in *Proc. Design Automation Conf.*, June 1992, pp. 40–44.

[31] B. Lin and F. Somenzi, "Minimization of symbolic relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1990, pp. 88–91.

[32] ——, "Minimization of symbolic relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1990.

[33] E. McCluskey, "Minimization of Boolean functions," *Bell Lab. Tech. J.*, vol. 35, pp. 1417–1444, Nov. 1956.

[34] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Boston, MA: Kluwer Academic, 1996.

[35] A. L. Oliveira and S. A. Edwards, "Inference of state machines from examples of behavior," UCB/ERL Tech. Rep. M95/12, Berkeley, CA, 1995.

[36] J.-K. Rho and F. Somenzi, "Stamina," computer program, 1991.

[37] S. Robinson, III and R. House, "Gimpel's reduction technique extended to the covering problem with costs," *IRE Trans. Electron. Comput.*, vol. EC-16, pp. 509–514, Aug. 1967.

[38] R. Rudell, "Espresso," computer program, 1987.

[39] ——, "Logic synthesis for VLSI design," Ph.D. dissertation, Univ. California, Berkeley, Apr. 1989; Tech. Rep. UCB/ERL M89/49.

[40] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "A uniform framework for satisfying input and output encoding constraints," in *Proc. Design Automation Conf.*, June 1991, pp. 170–175.

[41] F. Somenzi, "Cookie," computer program, 1989.

[42] G. Swamy, R. Brayton, and P. McGeer, "A fully implicit Quine-McCluskey procedure using BDD's," Tech. Rep. UCB/ERL M92/127, 1992.

[43] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit state enumeration of finite state machines using BDD's," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1990, pp. 130–133.

[44] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: Logic Optimization*. Boston, MA: Kluwer Academic, 1997.

Tiziano Villa, for a photograph and biography, see this issue, p. 675.

Timothy Kam (M'87), for a photograph and biography, see this issue, p. 675.

Robert K. Brayton (M'75–SM'78–F'81), for a photograph and biography, see this issue, p. 676.

Alberto L. Sangiovanni-Vincentelli (M'74–SM'81–F'83), for a biography, see this issue, p. 676.