A FULLY IMPLICIT ALGORITHM FOR EXACT STATE MINIMIZATION

by

Timothy Kam, Tiziano Villa, Robert Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M93/79

19 November 1993

A FULLY IMPLICIT ALGORITHM FOR EXACT STATE MINIMIZATION

by

Timothy Kam, Tiziano Villa, Robert Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M93/79

19 November 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A FULLY IMPLICIT ALGORITHM FOR EXACT STATE MINIMIZATION

by

Timothy Kam, Tiziano Villa, Robert Brayton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M93/79

19 November 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Fully Implicit Algorithm for Exact State Minimization

Timothy Kam* Tiziano Villa† Robert Brayton Alberto Sangiovanni-Vincentelli

Abstract

Implicit computations of the solution set of optimization problems arising in logic synthesis hold the promise of enlarging the size of input instances that can be solved exactly. The state minimization problem for incompletely specified machines is an important step for sequential circuit optimization. The problem is NP-hard. An exact algorithm consists of two steps: generation of sets of compatibles, and solution of a binate covering problem. This paper presents an implicit algorithm for exact state minimization of FSM's. There are various applications of logic synthesis that generate FSM's beyond the reach of state-of-art state minimization tools. Therefore it is of practical importance to revisit exact state minimization of ISFSM's and address the issue of representing implicitly the solution space. In this paper we show how to compute sets of maximal compatibles, compatibles and prime compatibles with implicit techniques and demonstrate that in this way it is possible to handle examples exhibiting a number of compatibles up to 2^{1200}, a number outside the scope of programs based on explicit enumeration [13]. We indicate also where such examples arise in practice. Then we address the final step of an implicit exact state minimization procedure, i.e. solving a binate table covering problem [24]. We present the first published algorithm for fully implicit exact binate covering. We report preliminary results of a prototype implementation capable of reducing huge binate tables (up to 10^6 rows and columns) and of carrying on the branch-and-bound procedure on an implicit representation of the table. Exact solutions to problems beyond the reach of traditional tools are so found.

1 Introduction

Seminal work by researchers at Bull [6] and improvements at UC Berkeley [26] produced powerful techniques for implicit enumeration of subsets of states of a Finite State Machine (FSM). These techniques are based on the idea to operate on large sets of states by their characteristic functions represented by Binary Decision Diagrams (BDD's). In many cases of practical interest these sets have a regular structure that translates into small-sized BDD's. Once the related BDD's can be constructed, the most common Boolean operations on them (including satisfiability) have low complexity, and this makes feasible to carry on computations otherwise impractical. Of course it may be the case that some BDD's cannot be constructed, because of the intrinsic structure of the function to represent or because a good ordering of the variables is not found.

Work at Bull [8, 19] showed how implicants, primes and essential primes of a two-valued or multi-valued function can also be computed implicitly. Reported experiments show a suite of examples where all primes could be computed, whereas explicit techniques implemented in ESPRESSO [2] failed to do so. More recently, implicit algorithms were presented to reduce the unate table of the Quine-McCluskey procedure to its cyclic core ([7] and [12]). In this way an exact solution to some hard problems beyond the reach of ESPRESSO could be found.

*Research support by DARPA under contract J-FBI90-073

†Research support by NSF under contract MIP-8719546 and California State MICRO Program

It is important to investigate how far these techniques based on implicit computations can be pushed to solve the core problems of logic synthesis and verification. When exact solutions are sought, explicit techniques run easily out of steam because too many elements of the solution space must be enumerated. It appears that implicit techniques offer the most realistic hope to increase the size of problems that can be solved exactly.

This paper presents an implicit algorithm for exact state minimization of FSM's. State minimization of FSM's is a well-known problem [16]. State minimization of completely specified FSM's (CSFSM's) has a complexity subquadratic in the number of states [14]. This makes it an easy problem when the starting point is a two-level description of an FSM, because the number of states is usually less than a few hundred. The problem becomes difficult to manage when the starting point is an encoded sequential circuit with a large number of latches (in the hundreds). In that case a traditional method would require the extracted state transition graph, with a number of states exponential in the number of latches, and so it would be impractical. Recently it has been shown [20, 18] how to bypass the extraction step and compute equivalence classes of states implicitly. Equivalence classes are basically all that is needed to minimize a completely specified state machine. A compatible projection operator uniquely selects a representative for each equivalence class.

State minimization of incompletely specified FSM's (ISFSM's) has been shown to be an NP-hard problem [21]. Therefore even for problems represented with two-level descriptions involving a hundred states, an exact algorithm may consume too much memory and time. As it will be shown in the experimental sections of this paper, there are various applications of logic synthesis that generate FSM's beyond the reach of state-of-art state minimization tools. Therefore it is of practical importance to revisit exact state minimization of ISFSM's and address the issue of representing implicitly the solution space.

We underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this paper can be applied to other problems of logic synthesis and combinatorial optimization. For instance the implicit computation of maximal compatibles described in this paper can be easily converted into an implicit computation of prime encoding-dichotomies (see [25]).

In this paper we show how to compute sets of maximal compatibles, compatibles and prime compatibles with implicit techniques and demonstrate that in this way it is possible to handle examples exhibiting a number of compatibles up to 2^{1200}, a number outside the scope of programs based on explicit enumeration [13]. We indicate also where such examples arise in practice. Then we address the final step of an implicit exact state minimization procedure, i.e. solving a binate table covering problem [24]. We present the first published algorithm for fully implicit exact binate covering. We report preliminary results of a prototype implementation capable of reducing huge binate tables (up to 10^6 rows and columns) and of carrying on the branch-and-bound procedure on an implicit representation of the table. Exact solutions to problems beyond the reach of traditional tools are so found.

The remainder of the paper is organized as follows. Section 2 introduces representations of FSM's based on Binary Decision Diagrams (BDD's) [5, 1]. Algorithms for implicit prime compatible computation are presented in Section 3. Section 4 presents some generalities on binate covering. Generation of the implicit binate table is described in Section 5, implicit table reduction is described in Section 6, while implicit column selection and other implicit computations are described in 7. Results on a variety of benchmarks are reported and discussed in Sections 8 and 9. Conclusions and future work are summarized in Section 10.

2 Implicit Representations

A Finite-State Machine is represented by its **State Transition Graph** (STG). A STG is denoted by a sextuple $\{I, O, S, IS, \delta, \lambda\}$, where I and O are the sets of inputs and outputs, S is the set of states and IS is the set of initial states. δ (next state function) is a mapping from $I \times S$ to S that given an input and a present state defines a next state. λ (output function) is a mapping from $I \times S$ to O that given an input and a present state

defines an output. An STG where the next-state and output for every possible transition from every state are defined corresponds to a **completely specified machine**. An **incompletely specified machine** is one where at least one of the functions δ and λ are partially defined, i.e. there is at least one pair (i, s) on which either the next state function or the output function (or both) are not defined.

Many algorithms for sequential synthesis have been developed for STG's. However, large FSM's cannot be stored and manipulated without prohibitively large memory usage and CPU time. A limitation of STG's is the fact that they are a two-level form of representation where state transitions are stored explicitly, one by one.

A binary decision diagram (BDD) [5, 1] provides an alternative way of representing FSM's. BDD is usually a more compact FSM representation than STG. A BDD is a rooted, directed acyclic graph (DAG) where each node is associated with a Boolean variable. There are 2 outgoing arcs from each node. The *then* arc corresponds to the case when the variable takes the value 1 and the *else* arc corresponds to the case when the variable takes 1. The leaves of the graph are the terminal nodes 0 and 1. A path from the root to a terminal 1 represents a satisfying assignment of variables on which the BDD evaluates to 1. Thus a BDD can represent any Boolean function on any n Boolean variables $f : B^n \rightarrow B$ where $B = \{0, 1\}$.

A rich set of BDD operators has been developed and published in the literature [5, 1], and their definitions will not be repeated here. We will use the notation $\exists x\ (\mathcal{F})$ to denote the existential quantification of a function $\mathcal{F}$ over a set of variables x, and $\exists x\ (\mathcal{F})$ to denote universal quantification. For simplicity in formulae, variable substitution will not be explicitly stated.

Any **subset** S in a Boolean space B^n can be represented by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its characteristic function, such that: $\chi_S(x) = 1$ iff x in S. In the sequel, we'll not distinguish the subset S from its characteristic function χ_S, and will use S to denote both.

Any **relation** $\mathcal{R}$ between n of Boolean variables can also be represented by a characteristic function $\mathcal{R} : B^n \rightarrow B$ as: $\mathcal{R}(x_1, x_2, \ldots, x_n) = 1$ iff the n-tuple $(x_1, x_2, \ldots, x_n)$ is in relation $\mathcal{R}$.

2.1 Positional-set Representation

To perform state minimization, one needs to represent and manipulate efficiently sets of states (such as compatibles) and sets of sets of states (such as sets of compatibles). Our goal is to represent any set of sets of states (i.e. set of state sets) implicitly as a single BDD, and manipulate such state sets symbolically all at once. Different sets of sets of states can be stored as multiple roots with a single shared BDD.

Suppose a FSM has n states, there are 2^n possible distinct subsets of states. In order to represent collections of them it is not possible to encode the states using $log_2 n$ Boolean variables. Instead, each subset of states is represented in **positional-set** or positional-cube notation form, using a set of n Boolean variables, $x = x_1 x_2 \ldots x_n$. The presence of a state 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 state s_k is not a member of the set. One Boolean variable is needed for each state because the state can either be present or absent in the set[1]. For example $n = 6$, the set with a single state s_4 is represented by 000100 while the set of states $s_2 s_3 s_5$ is represented by 011010. The states s_1, s_4, s_6 which are not present correspond to 0's in the positional-set.

A set of sets of states is represented as a set S of positional-sets by a characteristic function $\chi_S : B^n \rightarrow B$ as: $\chi_S(x) = 1$ iff the set of states represented by the positional-set x is in the set S. A BDD representing $\chi_S(x)$ will contain minterms, each corresponding to a state set in S. The operators for manipulating positional-sets and characteristic functions will be described in Section 2.2.

In the case of an ISFSM, some next states as well as the outputs may not be specified. So relations instead of functions must be used to represent the transition and output information.

[1]The representation of primes proposed by Coudert *et al.* [8] needs 3 values per variable to distinguish if the present literal is in positive or negative phase or in both phases.

Definition 2.1 *The* **transition relation** *is represented as:*

$$\mathcal{T}(i,p,n) = 1 \text{ iff } n \text{ is the specified next state of state } p \text{ on input } i \ (i.e.\ n = \delta(p,i)) \quad (1)$$

An unspecified next state from a state p under input i can be represented either by an entry (i,p,n) where the positional-set n is a vector of all 0's, or by not representing any entry with i and p in the relation at all. The latter is chosen for our implicit algorithm.

Definition 2.2 *The* **output relation** *is represented as:*

$$\mathcal{O}(i,p,o) = 1 \text{ iff } o \text{ is a (possibly unspecified) output of state } p \text{ on input } i \ (i.e.\ o = \lambda(p,i)) \quad (2)$$

We represent all unspecified outputs in the relation $\mathcal{O}$, to ensure correctness of the output compatibles computation described in Section 3. An unspecified output in the STG corresponds to a set of minterms carrying all possible output combinations.

2.2 Implicit Manipulation of Sets and Sets of Sets

We describe here how to represent and manipulate implicitly sets of objects. This theory is especially useful for applications where sets of sets of objects need to be constructed and manipulated, as it is often the case in logic synthesis and combinatorial optimization.

2.2.1 Operations on a Pair of Positional-sets

With our definitions of relations and positional-set notation for representing set of states, useful relational operators on sets can be derived. We propose a unified notational framework for set manipulation which extends the notation used in [19]. In this section, operators act on two sets of states represented as positional-sets $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_n$, and return 1 iff (x,y) are in the particular relation. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of states, x and y. For example, given two sets of state sets X and Y, the state set pairs (x,y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$.

Theorem 2.1 *The* **equality relation** *tests if the two sets of states represented by positional-sets x and y are identical, and can be computed as:*

$$(x = y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k \quad (3)$$

where $x_k \Leftrightarrow y_k = x_k \cdot y_k + \neg x_k \cdot \neg y_k$ designates the Boolean XNOR *operation and $\neg$ designates the Boolean* NOT *operation.*

$\prod_{k=1}^{n} x_k \Leftrightarrow y_k$ requires that for every state k, either both positional-sets x and y contain it, or it is absent from both. Therefore, x and y contains exactly the same set of states and thus are equal.

Theorem 2.2 *The* **containment relation** *tests if the set of states represented by x contains the set of states represented by y, and can be computed as:*

$$(x \supseteq y) = \prod_{k=1}^{n} y_k \Rightarrow x_k \quad (4)$$

where $x_k \Rightarrow y_k = \neg x_k + y_k$ designates the Boolean implication operation.

$\prod_{k=1}^{n} y_k \Rightarrow x_k$ requires that for all states, if a state k is present in y (i.e. $y_k = 1$), it must also be present in x ($x_k = 1$). Therefore set x contains all the states in y.

Theorem 2.3 *The* **strict containment relation** *tests if the set of states represented by x strictly contains the set of states represented by y, and can be computed as:*

$$(x \supset y) = (x \supseteq y) \cdot (x \neq y) \tag{5}$$

Equation 5 follows directly from the two previous theorems.

Theorem 2.4 *Given an incompatible pair of states (y, z), a position-set c satisfies $Contain\_Union(y, z, c)$ iff c contains both state y and state z. This constraint can be obtained by:*

$$Contain\_Union(c, y, z) = \prod_{k=1}^{n} y_k + z_k \Rightarrow c_k \tag{6}$$

$Contain\_Union(y, z, c)$ performs bitwise OR on singletons y and z. If either of their k-bit is 1, the corresponding c_k bit is constrained to 1. Otherwise, c_k can take any values (i.e. don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. Thus, it generates all the positional-sets c which contain the union of the positional-sets y and z.

2.2.2 Operations on Sets of Positional-sets

Theorem 2.5 *Given the characteristic functions χ_A and χ_B representing the sets A and B, set operations on them such as the* **union, intersection, sharp,** *and* **complementation** *can be performed as logical operations $(+, \cdot, \cdot\neg, \neg)$ on their characteristic functions.*

Theorem 2.6 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set containment test** *is true iff set A contains set B, and can be computed by:*

$$Set\_Contain_x(\chi_A, \chi_B) = \forall x\ \chi_B(x) \Rightarrow \chi_A(x) \tag{7}$$

Theorem 2.7 *The* **maximal** *of a set F of sets is the set containing sets in F not strictly contained by any other set in F, and can be computed as:*

$$Maximal_x(F) = F(x) \cdot \not\exists y\ [(y \supset x) \cdot F(y)] \tag{8}$$

The term $\exists y\ [(y \supset x) \cdot F(y)]$ is true iff there is a positional-set y in F such that $y \supset x$. In such a case, x cannot be in the maximal set by definition, and can be taken away. What remains is exactly the maximal set of states set in $F(x)$.

Theorem 2.8 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set union relation** *tests if positional-set y represents the union of all state sets in A, and can be computed by:*

$$Set\_Union_x(\chi_A, y) = \prod_{k=1}^{n} y_k \Leftrightarrow [\exists x\ \chi_A(x) \cdot x_k] \tag{9}$$

For each position k, the right hand expression sets y_k to 1 iff there exists an $x \in \chi_A$ such that its kth bit is a 1. This implies that the positional-set y will contain the kth element iff there exists a positional-set x in A such that k is a member of x. Effectively, the right hand expression performs a multiple bitwise OR over all positional-sets of χ_A to form a single positional-set y which represents the union of all such positional-sets.

2.2.3 k-out-of-n Positional-sets

Let n be the number of states. In subsequent computations, we will use extensively a suite of sets of state sets, $Tuple_{n,k}(x)$, which contains all positional-sets x with exactly k states in them (i.e. $|x| = k$). In particular, the set of singleton states $Tuple_{n,1}(x)$, the set of state pairs $Tuple_{n,2}(x)$, the set of all states $Tuple_{n,n}(x)$, and the set of empty state set $Tuple_{n,0}(x)$ are common ones.

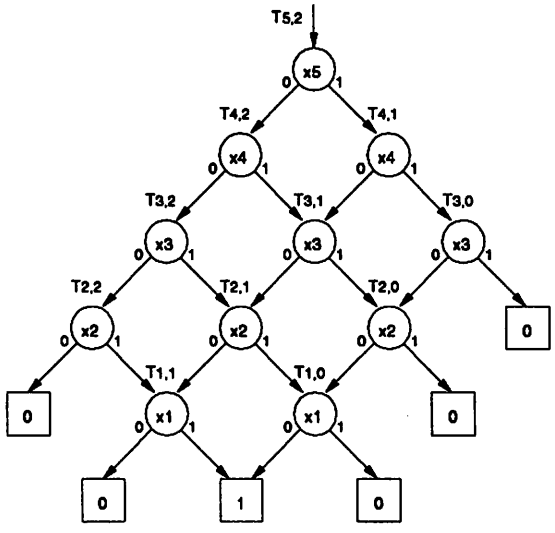

Figure 1: BDD representing $Tuple_{5,2}(x)$.

Such sets of k-tuple state sets have nice BDD structures. Figure 1 represents a BDD of $Tuple_{5,2}(x)$. Its root represents the set $Tuple_{5,2}(x)$, while the internal nodes represent the sets $Tuple_{i,j}(x)$ $(i < 5, j < 2)$. For ease of illustration, the variable ordering is chosen such that the top variable corresponding to $Tuple_{i,j}(x)$ is x_i. At that node, if we choose state i to be in the positional-set, x_i takes the value 1 and we follow the right outgoing arc. In doing so, we still have $i-1$ states/variables left to be processed. As we have put state i in the positional-set, we still have to add exactly $j-1$ states into the positional-set. That is why the right child of $Tuple_{i,j}(x)$ should be $Tuple_{i-1,j-1}(x)$. Similarly, the left child is $Tuple_{i-1,j}(x)$ because state i has not been put in the positional-set and we have $j-1$ states/variables left. Thus, the BDD for $Tuple_{i,j}$ can be constructed by the following algorithm:

```
Tuple(i, j) {
    if (j < 0) or (i < j) return 0
    if (i = j) and (i = 0) return 1
    if Tuple(i, j) in computed-table return result
    T = Tuple(i − 1, j − 1)
    E = Tuple(i − 1, j)
    F = ITE(x_i, T, E)
    insert F in computed-table for Tuple(i, j)
    return F
}
```

The total number of nonterminal vertices in the BDD of $Tuple_{n,k}$ is $nk - k^2 + n = O(nk)$. With the use of the computed table ([1]), the time complexity of the above algorithm is also $O(nk)$ as the BDD is built from bottom up and each vertex is built once and then re-used. Given any n, the BDD for $Tuple_{n,k}$ is largest when $k = n/2$.

3 Implicit Generation of Compatibles

An exact algorithm for state minimization consists of two steps: generation of various sets of compatibles, and solution of a binate covering problem. The generation step involves identification of sets of states called compatibles which can potentially be merged into a single state in the minimized machine. Unlike the case of CSFSM's, where state equivalence partitions the states, compatibles for incompletely specified FSM may overlap. As a result, the number of compatibles can be exponential in the number of states, and the generation of the whole set of compatibles can be a challenging task.

The covering step (described in Section 4) is to choose a minimum subset of compatibles satisfying covering and closure conditions, i.e. to find a minimum closed cover. The covering conditions require that every state is contained in at least one chosen compatible. The closure conditions guarantee that the states in a chosen compatible are mapped by any input sequence to states contained in a chosen compatible.

In this section, we describe implicit computations to find sets of compatibles required for exact state minimization. More details can be found in [15].

3.1 Output Incompatible Pairs

To generate compatibles, we must start with the given output and transition relations. Incompatibility relations between pairs of states are derived first.

Definition 3.1 *Two states are an* **output incompatible pair** *if, for some input, they cannot generate the same output.*

Theorem 3.1 *The set of output incompatible pairs, $\mathcal{OICP}(y,z)$, can be computed as:*

$$\mathcal{OICP}(y,z) = Tuple_1(y) \cdot Tuple_1(z) \cdot \exists i \; \not\exists o \, [\mathcal{O}(i,y,o) \cdot \mathcal{O}(i,z,o)]$$

Although y and z are positional-sets, they are used to capture a pair of states in this computation. The conditions $Tuple_1(y) \cdot Tuple_1(z)$ restrict them to represent only singleton states. The last term is true iff for some input i, there is no output pattern that both state y and z can produce.

3.2 Incompatible Pairs

Definition 3.2 *Two states are an* **incompatible pair** *if*

1. *they are output incompatible, or*
2. *on some input, their next states are incompatible.*

This is a recursive definition and we can unroll it into a fixed point iteration that can be implemented using BDD operations.

Theorem 3.2 *The set of incompatible pairs is the least fixed point of $\mathcal{ICP}(y,z)$ in the expression:*

$$\mathcal{ICP}(y,z) = \mathcal{OICP}(y,z) + \exists i,u,v \, [\mathcal{T}(i,y,u) \cdot \mathcal{T}(i,z,v) \cdot \mathcal{ICP}(u,v)]$$

and can be computed by the following iteration:

$$\begin{aligned} \mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) \\ \mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z) + \exists i,u,v \, [\mathcal{T}(i,y,u) \cdot \mathcal{T}(i,z,v) \cdot \mathcal{ICP}_k(u,v)] \end{aligned}$$

The iteration can terminate when $\mathcal{ICP}_{k+1} = \mathcal{ICP}_k$ and $\mathcal{ICP} = \mathcal{ICP}_k$.

The fixed point computation starts with the set of output incompatible pairs. After the kth iteration, $\mathcal{ICP}_{k+1}(y,z)$ contains all the incompatible state pairs (y,z) that lead to an output incompatible pair in k or less transitions. This set is obtained by adding state pairs (y,z) to the set $\mathcal{ICP}_k(y,z)$, if an input takes them into an already known incompatible pair (u,v).

3.3 Incompatibles

So far we established relationships between pairs of states. The following definition introduces sets of states of arbitrary cardinality.

Definition 3.3 *A set of states is an* **incompatible** *if it contains at least one incompatible pair.*

Theorem 3.3 *The set of incompatibles is computed as:*

$$\mathcal{IC}(c) = \exists y,z\,[\mathcal{ICP}(y,z) \cdot Contain\_Union(c,y,z)]$$

$Contain\_Union(c,y,z)$ captures all state sets c each containing at least an incompatible pair of singleton states $(y,z) \in \mathcal{ICP}$.

3.4 Compatibles

Definition 3.4 *A set of states is a* **compatible** *if it is not an incompatible.*

Theorem 3.4 *The set of compatibles, $\mathcal{C}(c)$, can be computed as:*

$$\mathcal{C}(c) = \neg Tuple_0(c) \cdot \neg\mathcal{IC}(c)$$

The set of compatibles simply contains all non-empty subsets of states which are not incompatibles. The empty set in positional-set notation is $Tuple_0(c)$ and all subsets which are not incompatible are given by $\neg\mathcal{IC}(c)$.

3.5 Implied Classes of a Compatible

To set up the covering problem we need also to compute the closure conditions for each compatible. They are captured by the notion of class set of a compatible, related to the set of next states implied by a compatible.

Definition 3.5 *A set of states d_i is an* **implied set** *of a compatible c for input i if d_i is the set of next states from the states in c on input i.*

Theorem 3.5 *The implied set (in singleton form) of a compatible c for input i can be defined by the relation $\mathcal{F}(c,i,n)$ which evaluates to 1 iff on input i, n is a next state from state p in compatible c.*

$$\mathcal{F}(c,i,n) = \exists p\,[\mathcal{C}(c) \cdot (c \supseteq p) \cdot \mathcal{T}(i,p,n)]$$

In $\mathcal{F}(c,i,n)$, a compatible $c \in \mathcal{C}(c)$ and an input i are associated with singleton next states n. Given c and i, n is in relation $\mathcal{F}(c,i,n)$ (i.e. state n is in the implied set of compatible c under input i) iff the right hand expression is true. i.e. if there is a present state $p \in c$ such that n is the next state of p on input i.

Note that the implied next states are represented here as singleton states in $\mathcal{F}(c,i,n)$. The singletons n in relation with a compatible c and an input i can be combined into a single positional-set, for later convenience. This positional-set representation of implied sets associates each compatible c with a set of implied sets d.

Theorem 3.6 *The implied sets d (in positional-set form) of a compatible c for all inputs, $\mathcal{CI}(c,d)$, can be computed as:*

$$\mathcal{CI}(c,d) = \exists i\,[\exists n(\mathcal{F}(c,i,n)) \cdot Set\_Union_n(\mathcal{F}(c,i,n),d)]$$

$\mathcal{F}(c,i,n)$ relates implied next states as singleton positional-sets to compatible c and input i and $Set\_Union_n(\mathcal{F}(c,i,n),d)$ forms the union of these singleton sets by bitwise OR and produces a positional-set d. The term $\exists n(\mathcal{F}(c,i,n))$ is needed, to exclude invalid compatible input combinations. Finally the implied sets of c over different inputs are obtained by an existential quantification of the inputs i.

3.6 Class Set of a Compatible

Definition 3.6 *An implied set d of a compatible c is in its* **class set** *iff*

1. *d has more than one element, and*
2. $d \not\subseteq c$, *and*
3. $d \not\subseteq d'$ *if* $d' \in$ *class set of c.*

We can ignore any implied set which contains only a single state, because its closure condition is automatically satisfied if the state is covered by some chosen compatible. Also if $d \subseteq c$, the closure condition is satisfied by the choice of c. Finally, if the closure condition corresponding to d' is stronger than that of d, the implied set d is not necessary.

Theorem 3.7 *The class set of a compatible c is captured by the relation $\mathcal{CCS}(c,d)$ which evaluates to 1 iff the implied set d is in the class set of compatible c.*

$$\mathcal{CCS}(c,d) = \neg Tuple_1(d) \cdot (c \not\supseteq d) \cdot Maximal_d(\mathcal{CI}(c,d))$$

The singleton implied sets $Tuple_1(d)$ are first excluded according to condition 1 in definition 3.6. By condition 2, we prune away implied sets d which are contained in their corresponding compatibles c. Finally given a compatible c, $Maximal_d(\mathcal{CI}(c,d))$ gives all its implied sets d which are not strictly contained by any other implied sets in $\mathcal{CCS}(c,d)$.

3.7 Prime Compatibles

To solve exactly the covering problem, it is sufficient to consider a subset of compatibles called prime compatibles. It has been proved in [10] that at least one minimum closed cover consists entirely of prime compatibles.

Definition 3.7 . *A compatible c'* **dominates** *a compatible c if*

1. $c' \supset c$, *and*
2. *class set of* $c' \subseteq$ *class set of c.*

i.e. a compatible c' dominates a compatible c if c' covers all states covered by c, and the closure conditions of c' are a subset of the closure conditions of c.

As a result, compatible c' expresses strictly less stringent conditions than compatible c. Therefore c' is always a better choice for a closed cover than c, thus c can be excluded from further consideration.

Theorem 3.8 *The prime dominance relation, Dominate(c',c), is defined by:*

$$Dominate(c', c) = (c' \supset c) \cdot Set\_Contain_d(CCS(c, d), CCS(c', d))$$

The two terms on the right-hand expression state the two dominance conditions by which c' dominates c according to definition 3.7. Since compatibles c and c' are represented as positional-sets, $(c' \supset c)$ is computed according to theorem 2.3. On the other hand, class sets are sets of sets of states and are represented by their characteristic functions. Containment between such sets of sets of states is computed by $\forall d\, CCS(c', d) \Rightarrow CCS(c, d)$, as described by theorem 2.6.

Definition 3.8 *A* **prime compatible** *is a compatible not dominated by another compatible.*

Theorem 3.9 *The set of prime compatibles can be computed as:*

$$\mathcal{PC}(c) = \mathcal{C}(c) \cdot \not\exists c' \, [\mathcal{C}(c') \cdot Dominate(c', c)]$$

By definition 3.8, a compatible c is not a prime compatible if it is dominated by another compatible c'. This condition is captured by the expression $\exists c'\, \mathcal{C}(c') \cdot Dominate(c', c)$. The set of prime compatibles is simply given by the set of compatibles $\mathcal{C}(c)$ excluding those that are dominated by other compatibles.

4 Implicit Binate Covering

In this section we introduce the classical branch-and-bound algorithm for minimum-cost binate covering. This technique has been described in [11, 10, 3, 4] and implemented by means of efficient computer programs (ESPRESSO and STAMINA). The referred papers offer a thorough description of the algorithm. The branch-and-bound solution of minimum binate covering is based on a recursive procedure shown in Fig. 2. In our implicit formulation we keep the branch-and-bound scheme, but we replace the traditional description of the table as a matrix (usually 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 computations on the binate table required to implement the branch-and-bound scheme. In the following sections we are going to describe the following:

- Implicit representation of the covering table
- Implicit reduction
- Implicit branching column selection
- Implicit computation of the lower bound.

At each call of the binate cover routine *mincov*, the binate table undergoes a reduction step and, if termination conditions are not met, a branching column is selected and *mincov* is called recursively twice, once assuming the selected column 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 on the current partial solution and comparing it with an upper bound U (best solution obtained so far). A good lower bound is based on the computation of a maximal independent set.

```
mincov(R, C, U) {
    (R, C) = Reduce(R, C, U)
    if (terminalCase(R, C)) {
        if (cost(R, C) < U) {
            U = cost(R, C)
            return solution
        }
        else return no solution
    }
    L = LowerBound(R, C)
    if (L ≥ U) return no solution
    c_i = ChooseColumn(R, C)
    S^1 = mincov(R_{c_i}, C_{c_i}, U)
    S^0 = mincov(R_{c̄_i}, C_{c̄_i}, U)
    return BestSolution(S^1 ∪ {c_i}, S^0)
}
```

Figure 2: Branch-and-bound algorithm for binate covering

5 Implicit Covering Table Generation

To keep with our stated objective, also the binate table is represented implicitly. We describe an implicit representation of the covering table, that adroitly exploits how row and columns were implicitly computed. 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.

This choice allows us to define all table manipulations needed by the reduction algorithms in terms of operations on rows and columns and to exploit all the special features of the binate covering problem set up in the case of state minimization (for instance, each row has at most one 0). Even though it seems that we are sacrifying generality, we point out that a similar technique could be applied to various binate covering problems that arise in logic synthesis, with a suitable encoding of the rows and columns.

Definition 5.1 *The set of columns* C [2] *is obtained from prime generation as:*

$$C(p) = \mathcal{PC}(p)$$

Beside distinguishing a row from another, each row label must also contain information regarding the positions of 1's and 0's in the row. Each row is labelled by a pair (c, d) which represents two positional sets. Positional set d relates to the 1 entries in the row, while c is related to the 0 entry. These definitions are motivated by the meaning of rows and columns in our application.

Definition 5.2 *The table entry at the intersection of the column labelled by* $p \in C$ *and of the row labelled by* $(c, d) \in R$ *is 1 iff* $(p \supseteq d)$.

The table entry at the intersection of the column labelled by $p \in C$ *and of the row labelled by* $(c, d) \in R$ *is 0 iff* $(p = c)$.

[2] $C(p)$ is used to represent the set of columns, and is different from $\mathcal{C}(c)$ which denotes the set of compatibles.

The columns of a table are labelled by prime compatibles p. Given a particular row labelled by (c, d), the columns intersecting it in a 1 are labelled by the prime compatibles p that contain d. Since there is at most one 0 in the row labelled by (c, d), the label of the column p intersecting it in a 0 is recorded in the row label by setting its c part to p. If there is no 0 in a row, c is set to the empty set, $Tuple_0(c)$. In this way, the table can be manipulated without representing its entries. E.g., a column labelled p contains a 0 iff $\exists c, d\,[R(c, d) \cdot (p = c)]$, i.e. iff $\exists d\, R(p, d)$.

Definition 5.3 *The set of row labels R is given by:*

$$R(c, d) = R_b(c, d) + R_u(c, d) = \mathcal{PC}(c) \cdot \mathcal{CCS}(c, d) + Tuple_0(c) \cdot Tuple_1(d)$$

The closure conditions associated with a prime compatible p are that if p is included in a solution, each implied set d in its class set must be contained in at least one chosen prime compatible. A binate clause of the form $(\overline{p} + p_1 + p_2 + \cdots + p_k)$ has to be satisfied for each implied set of p, where p_i is a prime compatible containing the implied set d for $1 \leq i \leq k$. With our interpretation of the row and column labels, the labels for binate rows are given succinctly by $\mathcal{PC}(c) \cdot \mathcal{CCS}(c, d)$. There is a row label for each (c, d) pair such that $c \in \mathcal{PC}$ is a prime compatible and d is one of its implied sets in $\mathcal{CCS}(c, d)$. This row label consistently represents the binate clause because the 0 entry in the row is given by the column labelled by the prime compatible $c = p$, and the row has 1's in the columns labelled by p_i whenever $(p_i \supseteq d)$.

The covering conditions require that each state be contained by some prime compatible in the solution. For each state d, a unate clause has to be satisfied which is of the form $(p_1 + p_2 + \cdots + p_j)$ where the p_i's are the prime compatibles that contain the state d. By specifying the unate row labels to be $Tuple_0(c) \cdot Tuple_1(d)$, we define a row label for each state in $Tuple_1(d)$. Since the row has no 0, its c part must be the empty set $Tuple_0(c)$. The 1 entries are correctly positioned at the intersection with all columns labelled by prime compatibles p_i which contain the singleton state d.

6 Implicit Reduction Techniques

Three fundamental operations constitute the essence of the reduction rules:

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 minimal cover, i.e. if there exists 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 results. Reductions are usually attemped in a given order, until nothing changes any more (i.e. the covering matrix has been reduced to a cyclic core). Fig. 3 shows the reductions and order implemented in our reduction algorithm. In the reduction there are two cases when no solution is generated:

1. The added cardinality of the set of essential columns, *ess_col*, and of the partial solution computed so far, Sol, is larger or equal than the upperbound U. In this case, a better solution is known than the one that can be found from now on and so the current computation branch can be bounded away.

```
Reduce(R, C, Sol, U) {
    do {
        Duplicated_Columns(C)
        Column_Dominance(R, C)
        Essential_Column(R, C)
        if (card(Sol) + card(ess_col) >= U) return(no solution)
        Sol = Sol ∪ ess_col
        Unacceptable_Column(R, C)
        Unnecessary_Column(R, C)
        if (C does not cover R) return(no solution)
        Duplicated_Rows(R)
        Row_Dominance(R, C)
    } while (R or C changed)
    return (R, C)
}
```

Figure 3: Fixed-point reduction computation

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.

We are going to describe how the reduction operations are performed using the special table representation described in the previous section.

6.1 Duplicated Rows and Columns

It is possible that, after some reductions, more than one column (row) label is associated with columns (rows) that coincide element by element. We need to identify such duplicated columns (rows) and collapse them into a single column (row)[3]. This can be seen as computing the equivalence relation of duplicated columns (rows) and selecting one representative for each equivalence class.

Theorem 6.1 Duplicated columns *can be detected and collapsed into one by:*

$$\begin{aligned} dup\_col(p', p) &= \nexists d\ R(p', d) \cdot \nexists d\ R(p, d) \cdot \nexists c, d\ [R(c, d) \cdot ((p' \supseteq d) \not\Leftrightarrow (p \supseteq d))] \\ C(p) &= C(p) \cdot \nexists p'\ [C(p') \cdot dup\_col(p', p) \cdot (p' < p)] \end{aligned}$$

The first equation says that column labels p' and p are in relation *dup_col* iff they identify two columns that agree element by element, i.e. they have the same 0's and 1's. Since each row has at most one 0, two duplicated columns cannot have 0 entries. This condition is expressed by $\nexists d\ R(p', d) \cdot \nexists d\ R(p, d)$, meaning that there is no row with a 0 in the column labelled by p' and there is no row with a 0 in the column labelled by p. In order that two columns agree on all entries to 1, there must not be a row (c, d) with a 1 in the column labelled by p and not in the column labelled by p', or vice versa. This condition is expressed by $\nexists c, d\ [R(c, d) \cdot ((p' \supseteq d) \not\Leftrightarrow (p \supseteq d))]$.

[3]This avoids the problem of columns (rows) dominating each other when performing implicitly column (row) dominance.

The second computation picks a representative column label out of a set of columns labels corresponding to duplicated columns. A column label p is deleted from C iff there is a column label p' which has a smaller binary value than p and both label duplicated columns. Here we exploit the fact that any positional-set p can be interpreted as a binary number. Therefore, a unique representative from a set can be selected by picking the one with the smallest binary value. In alternative, one could have used the *cprojection* BDD operator introduced in [20], but the latter was not available in our BDD package.

So far we have tried hard to distinguish in the exposition a column (row) label from the column (row) itself. From now on, sometimes we will blur the distinction to ease the write-up, but the context should say clearly which one it is meant.

Theorem 6.2 Duplicated rows *can be detected and collapsed into one by:*

$$\begin{aligned} dup\_row(c', d', c, d) &= (c' = c) \cdot \nexists p\,[C(p) \cdot ((p \supseteq d') \not\Leftrightarrow (p \supseteq d))] \\ R(c, d) &= R(c, d) \cdot \nexists c', d'\,[R(c', d') \cdot dup\_row(c', d', c, d) \cdot (d' < d)] \end{aligned}$$

Two row labels denote duplicated rows iff they have the same entry to 0 (if any), $(c' = c)$, and no column $p \in C$ intersects one row in a 1 iff the other doesn't. Selection of a representative from duplicated rows and table updating are performed as in the case of duplicated columns.

6.2 Column Dominance

Definition 6.1 *A column p'* **α-dominates** *another column p iff p' has all the 1's of p, and p' contains no 0.*

Theorem 6.3 *The set of α-dominated columns can be computed by:*

$$\alpha\_dominated(p) = \exists p'\,\{C(p') \cdot (p' \neq p) \cdot \nexists c, d\,[R(c, d) \cdot (p \supseteq d) \cdot (p' \not\supseteq d)] \cdot \nexists d\,R(p', d)\}$$

The expression $\exists c, d\,[R(c, d) \cdot (p \supseteq d) \cdot (p' \not\supseteq d)]$ defines pairs of columns p and p' such that in some row $(c, d) \in R$, p has a 1 while p' does not, i.e. p' doesn't dominates p. Suppose column p' has a 0, $R(p', d)$ would be true. Examining the whole equation, p is a α-dominated column iff there is another column p' *different* from p such that none of the above two possibilities is true.

Theorem 6.4 *These computations delete a set of columns $D(p)$ from a table and all rows intersecting them in a 0.*

$$C(p) = C(p) \cdot \neg D(p)$$

$$R(c, d) = R(c, d) \cdot \neg D(c)$$

The first computation updates the set of columns $C(p)$. The expression $R(c, d) \cdot \neg D(c)$ defines all rows not intersecting in a 0 the columns in D. In fact the rows (c, d) intersecting the deleted columns in a 0 have their c parts equal to some column label in D.

Suppose that $D(p) = \alpha\_dominated(p)$. Then α-dominated columns are deleted from the set of columns, and rows intersecting them in a 0 are deleted from the set of rows. The same computations can be used to delete unacceptable and unnecessary columns.

6.3 Row Dominance

Definition 6.2 *A row $r' = (c', d')$* **dominates** *another row $r = (c, d)$ iff r has all the 1's and 0 of r'.*

Theorem 6.5 *Reduction by row dominance can computed by:*

$$R(c,d) = R(c,d) \cdot \not\exists c', d' \{R(c',d') \cdot \not\exists p [C(p) \cdot (p \supseteq d') \cdot (p \not\supseteq d)] \cdot [Tuple_0(c') + (c' = c)] \cdot ((c',d') \neq (c,d))\}$$

Since each row contains at most one 0, if r' dominates r, either r' has no 0 (i.e. the c' part is $Tuple_0(c')$) or r' and r have a 0 entry in the same column, $(c' = c)$. Thus for the 0 entries, we require $[Tuple_0(c') + (c' = c)]$ to be true. For the 1 entries, the expression $\not\exists p\,[C(p) \cdot (p \supseteq d') \cdot (p \not\supseteq d)]$ requires that no column intersects in a 1 row (c', d') but not row (c, d). These two conditions together correspond to the definition of (c', d') as a dominating row. The equation says that rows (c, d) dominated by another *different* row (c', d') are deleted from the table.

6.4 Essential and Unacceptable Columns

Definition 6.3 *A column p is* **essential** *iff there is a row having a 1 in column p and 2 everywhere else.*

Theorem 6.6 *The set of essential columns can be computed by:*

$$ess\_col(p) = C(p) \cdot \exists c, d\,[R(c,d) \cdot Tuple_0(c) \cdot (p \supseteq d) \cdot \not\exists p'\,(C(p') \cdot (p' \supseteq d) \cdot (p' \neq p))]$$

The right part of the equation expresses the condition that a column p is essential. For such a p, there must be a row $(c, d) \in R$ which (1) doesn't contain any 0 (i.e. $Tuple_0(c)$), (2) contains a 1 in column p (i.e. $(p \supseteq d)$), and (3) there is not *another* column in the row with a 1 (i.e. $\not\exists p'\,(C(p') \cdot (p' \supseteq d) \cdot (p' \neq p))$).

Theorem 6.7 *Essential columns must be in the solution. Each essential column must then be deleted from the table together with all rows where it has 1's.*

Theorem 6.8 *These computations add essential columns to the solution, delete them from the set of columns and delete all rows in which they have 1's:*

$$solution(p) = solution(p) + ess\_col(p)$$

$$C(p) = C(p) \cdot \neg ess\_col(p)$$

$$R(c,d) = [R(c,d) \cdot \not\exists p\,(ess\_col(p) \cdot (p \supseteq d)) \cdot \neg ess\_col(c)] + [\exists c\,(R(c,d) \cdot ess\_col(c)) \cdot Tuple_0(c)]$$

The first two equations move the essential columns from the column set to the solution set.

When essential columns are deleted, rows intersecting them in a 1 must be deleted, but rows intersecting them in a 0 have only the 0 entry removed. This means that binate rows intersecting an essential column in a 0 become unate and the row labels must be updated accordingly. In the last equation, the term $R(c,d) \cdot \not\exists p\,(ess\_col(p) \cdot (p \supseteq d))$ defines rows (c, d) that do not intersect any essential column in a 1. Rows intersecting an essential column in a 0 are removed (by the term $\neg ess\_col(c)$) and then are added back (by the term $\exists c\,(R(c,d) \cdot ess\_col(c)) \cdot Tuple_0(c)$, where $Tuple_0(c)$ sets the c part to the empty set).

Definition 6.4 *A column p is an* **unacceptable column** *iff there is a row having a 0 in column p and 2 everywhere else.*

Theorem 6.9 *The set of unacceptable columns can be computed by:*

$$unacceptable\_col(p) = C(p) \cdot \exists d\,[R(p,d) \cdot \nexists p'\,(C(p') \cdot (p' \supseteq d))]$$

The term $R(p,d)$ asserts that there is a row (c,d) whith a 0 in column p, while $\nexists p'\,(C(p') \cdot (p' \supseteq d))$ requires that no other column intersects that row in 1.

Definition 6.5 *A column p is an* **unnecessary column** *iff it doesn't have any 1 in it.*

Theorem 6.10 *The set of unnecessary columns can be computed as:*

$$unnecessary\_col(p) = C(p) \cdot \nexists c,d\,[R(c,d) \cdot (p \supseteq d)]$$

A column p is unnecessary iff no row $(c,d) \in R$ intersects it in a 1, i.e. $\nexists c,d\,[R(c,d) \cdot (p \supseteq d)]$.

Theorem 6.11 *Unacceptable and unnecessary columns should be eliminated from the table, together with all the rows in which such columns have 0's.*

The table is updated according to Theorem 6.4 by setting $D(p) = unacceptable\_col(p) + unnecessary\_col(p)$.

7 Other Implicit Computations

To have a fully implicit binate covering algorithm, according to the scheme shown in Figure 2, we must also compute implicitly a branching column and a lower bound. These computations usually require some form of counting and comparison of integers. Such operations are not available as BDD primitive operations, as they are not well-suited for two-terminal BDD's. In the sequel we will describe how we designed a new family of BDD operations that select columns (or rows) that have a maximum (minimum) number of 1's (0's).

7.1 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. Usually the selection of a branching column 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.

Here we adopt a simplified criterion: select a column with a maximum number of ones. We show now how a set of columns with a maximum number of 1's can be found implicitly.

7.2 Selection of a Column with Maximum Number of 1's

Since we are interested only in the 1 entries of the table, we define the following relation:

$$F(p,c,d) = C(p) \cdot R(c,d) \cdot (p \supseteq d)$$

A column p and a row (c,d) intersect in a 1 iff $F(p,c,d) = 1$. Define the pair of variables (c,d) as r, so that $F(p,c,d) = F(p,r)$. Note that $F(p,r)$ can be decomposed as: $F(p,r) = \sum_{p_i \in C(p)} (p = p_i) \cdot F(p_i, r)$.

```
LineMaxElem(F, top_r_index) {
    v = bdd_top_var(F)
    if (index(v) >= top_r_index) {
        return (1, bdd_count_onset(F))
    } else { /* v is a p variable */
        (T, count_T) = LineMaxElem(bdd_then(F), top_r_index)
        (E, count_E) = LineMaxElem(bdd_else(F), top_r_index)
        count = max(count_T, count_E)
        if (count = count_T = count_E)
            C = ITE(v, T, E)
        else if (count = T)
            C = ITE(v, T, 0)
        else if (count = count_E)
            C = ITE(v, 0, E)
        return (C, count)
    }
}
```

Figure 4: Implicit computation of lines with maximum number of entries

Our original problem of finding the column with the maximum number of 1's in the table reduces to finding the p related to the maximum number of r's in the relation $F(p, r)$. A brute force method is to cofactor the BDD $F(p, r)$ with respect to each $p_i \in C(p)$, count the number of minterms in the onset of each $F(p_i, r)$, and pick the column with the maximum count. The obvious disadvantage of this method is that we are counting one BDD for each column in the table. We describe an algorithm, $LineMaxElem$, for implicit counting which traverses each node of the BDD $F(p, r)$ exactly once. Figure 4 shows an outline of $LineMaxElem$.

Variables in p are required to be ordered before variables in r. Define top_r_index as the smallest index of the variables in r (corresponding to the highest variable in BDD). $LineMaxElem$ takes the relation $F(p, r)$ and top_r_index as arguments and returns the BDD of the set of columns in p which are related to the maximum number of r's in F, together with the count. Starting from the root of BDD F, the algorithm traverses down the graph by recursively calling $LineMaxElem$ on the *then* and *else* subgraphs. This recursion stops when the top variable v of F is in the variable set r. In this case, the BDD rooted at v corresponds to a cofactor $F(p_i, r)$ for some p_i. The minterms in its onset are counted and returned as *count*, which is the number of r's that are related to p_i (i.e. the number of 1's in column p_i).

Next we construct a new BDD in a bottom up fashion, representing the set of columns with maximum count. The two recursive calls of $LineMaxElem$ return the sets T and E of columns with maximum count $count\_T$ and $count\_E$ for the *then* and the *else* subgraphs. The larger of the two counts, $count = max(count\_T, count\_E)$, is returned. If the two counts are the same, the columns in T and E are merged by $ITE(v, T, E)$. 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 the BDD $F(p, r)$ is traversed once, two computed tables ([1]) must be used, though they are not shown in the algorithm. Firstly, the *bdd_count_onset*() results must be saved across different calls of the routine. Secondly, the results of $LineMaxElem$ must also be stored in another computed table. Note that $LineMaxElement$ returns a set of columns of maximum count. Since we need

only one column, some heuristic is used to break the ties and to keep only one column.

One can compute the columns with minimum number of 1's, by replacing in $LineMaxElem$ the expression $max(count\_T, count\_E)$ with $min(count\_T, count\_E)$. Moreover, if variables in r are required to be ordered before variables in p one can compute the rows that maximize or minimize the number of 1's. Summing up, we have a family of procedures that can either maximize or minimize the number of 1's of either the columns or of the rows (similarly, for 0's in place of 1's). An application where rows with minimum number of 1's are needed is the implicit computation of the maximal independent set (Section 7.3).

7.3 Implicit computation of a maximal independent set

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. Maximum independent set is an NP-hard problem and an approximate one (only maximal) can be computed by a greedy algorithm. Let R be the set of rows out of which a maximal independent set must be found. While R is non-empty, a row r of R is found that is disjoint from a maximum number of rows (i.e. the row of minimum length in R). All rows having elements in common with r_i are then discarded from R. At the end of the iteration, a set of pairwise disjoint rows (independent set) and their minimum covering cost is found.

An analogue of the previous greedy procedure can be reproduced in our implicit frame, noticing that we showed already how to compute implicitly the rows with minimum number of 1's (Section 7.2). Since we are interested only in the unate rows of the table, we define the following relation:

$$\tilde{F}(p,d) = \exists c R(c,d).Tuple_0(c).C(p).Contain(p,d)$$

A column p and a unate row (c,d) intersect in a 1 iff $\tilde{F}(p,d) = 1$. Rows here are labelled only by the second half d of their complete label, because it is sufficient to distinguish them. Once a shortest row $r(d)$ is found, one can discard from $\tilde{F}(p,d)$ the row $r(d)$ together with all the other rows intersecting the same columns by:

$$\tilde{F}(p,d) = \tilde{F}(p,d).\ \not\exists p\{\exists d[r(d).Contain(p,d)].C(p).Contain(p,d)\}.$$

8 Experimental Results for Computation of Compatibles

We report results on different suites of FSM's. They are:

1. The MCNC benchmark and other examples.
2. FSM's from asynchronous synthesis [17].
3. FSM's from learning I/O sequences [9].
4. FSM's from synthesis of interacting FSM's [27].
5. Constructed FSM's that exhibit a large number of maximal and prime compatibles.
6. Random FSM's.

We discuss features of the experiments and results in different subsections. Our program is called ISM, an acronym for implicit state minizer. Comparisons are made with STAMINA, a program that represents the state-of-art for state minimization based on explicit techniques. The program STAMINA was run with the option -P to compute all primes. All run times are reported in CPU seconds on a DEC DS5900/260 with 440 Mb of memory. $\#\mathcal{NEPC}$ denotes the number of nonessential prime compatibles.

8.1 Examples from MCNC Benchmark and Others

Table 1 reports the results of the most interesting examples (as far as state minimization is concerned) from the MCNC benchmark and from other academic and industrial benchmarks available to us. Most examples have a small number of prime compatibles, with the exception of *ex2* and *green*. The running times of ISM are worse than those of STAMINA, especially in those cases where there are very few compatibles in the number of states (*squares* is the most striking example). But when the number of primes is not negligible as in *ex2* and *green*, ISM ran as fast or faster than STAMINA. This is consistent with our expectations, since ISM manipulates relations having a number of variables linearly proportional to the number of states. When very few compatibles need to be represented, the purpose of ISM is defeated and its representation becomes very inefficient.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
ex2	19	36	2925	1366	1366	8	13
green	54	524	1234	524	524	90	125
squares	371	45	473	307	0	731	1
tbk	32	16	48	48	48	3	1

Table 1: Examples from the MCNC Benchmark and others.

8.2 Examples of FSM's from Asynchronous Synthesis

Table 2 reports the results of a benchmark of FSM's generated as intermediate steps of an asynchronous synthesis procedure [17]. We notice that STAMINA ran out of memory on the examples *vmebus.master.m, isend, pe-rcv-ifc.fc, pe-send-ifc.fc*, while ISM was able to complete them. These examples (with the exception of *vbe4a*) have a number of primes below 1000. To explain the data reported in Table 2, we notice that in order to compute the prime compatibles, the set of compatibles needs to be generated too. The compatibles of the FSM's of this benchmark are usually of large cardinality and therefore their enumeration causes a combinatorial explosion. So the huge size of the set of compatibles accounts for the large running times and/or out-of-memory failures. About the behavior of ISM, we underline that the running times track well with the size of the set of compatibles and that in significant cases they are well below those of STAMINA (*pe-rcv-ifc.fc.m, pe-send-ifc.fc.m, vbe4a*). Notice that for asynchronous synthesis a more appropriate formulation of exact state minimization requires the computation of all compatibles or at least of prime compatibles and a different set-up of the covering problem [17].

8.3 Examples of FSM's from Learning I/O Sequences

Table 3 shows the results of running a parametrized set of FSM's constructed to be compatible with a given collection of examples of input/output behavior [9]. From a sequence of n input/output pairs, a machine is generated with $n + 1$ states and n transitions, one for each input/output pair. These machines exhibit very large number of compatibles.

Here ISM shows all its power compared to STAMINA, both in terms of number of computed primes and running time. STAMINA runs out of memory on the examples from *threer.35* onwards and, when it completes, it takes close to two order of magnitude more time than ISM.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	STAMINA
alex1	42	787	55928	787	787	26	16
intel_edge.dummy	28	120	9432	396	396	38	3
isend	40	128	22207	480	480	16	spaceout
pe-rcv-ifc.fc	46	28	1.528e11	148	148	18	spaceout
pe-rcv-ifc.fc.m	27	18	1.793e6	38	38	3	147
pe-send-ifc.fc	70	39	5.071e17	506	506	571	spaceout
pe-send-ifc.fc.m	26	6	8.978e6	23	22	3	312
vbe4a	58	2072	1.756e12	2072	2072	112	167
vmebus.master.m	32	10	5.049e7	28	28	16	spaceout

Table 2: Asynchronous FSM benchmark.

machine	# state	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
threer.10	11	671	112	0	0
threer.20	21	16829	3936	1	159
threer.30	31	97849	33064	50	1344
threer.40	41	1.456e6	529420	156	spaceout
threer.50	51	1.680e7	7.246e6	1142	spaceout
fourr.10	11	2047	1	0	0
fourr.20	21	42193	12762	2	217
fourr.30	31	1.346e6	542608	20	spaceout
fourr.40	41	5.266e9	2.388e9	105	spaceout
fourr.50	51	3.643e7	1.696e7	198	spaceout
fourr.60	61	1.052e10	5.021e9	6101	spaceout
fourr.70	71	9.621e10	4.524e10	22940	spaceout

Table 3: Learning I/O sequences benchmark.

8.4 Examples of FSM's from Synthesis of Interacting FSM's

It has been reported by Rho and Somenzi [22] that the exact state minimization of the driven machine of a pair of cascaded FSM's is equivalent to the state minimization of an ISFSM that requires the computation of prime compatibles.

Recently Wang and Brayton [27] have implemented a program to optimize a FSM, exploiting the input don't care sequences induced by a surrounding network of FSM's. Their procedure produces FSM's that requires a final step of state minimization and exhibit often large number of prime compatibles. We will report on these experiments in a final version of the paper, when these examples will be available to us.

8.5 A Family of FSM's with Exponentially Many Primes

We describe here a suite of FSM's whose number of prime compatibles is exponential in the number of states.

Rubin gave in [23] a sharp upper bound for the number of maximal compatibles of an ISFSM. He showed that $M(n)$, the maximum number of maximal compatibles over all ISFSM's with $n > 1$ states, is given by $M(n) = i.3^m$, if $n = 3.m + i$. The proof of this counting statement is based on the construction of a family of incompatibility graphs $I(n)$ parametrized in the number of states[4]. Each $I(n)$ is composed canonically of a number of connected components. Each maximal compatible contains exactly one state from each connected component of the graph. The number of such choices is shown to be $M(n)$.

The proof of the theorem does not exhibit an FSM that has a canonical incompatibility graph. Based on the construction of the incompatibility graphs given in the paper, we have built a family $F(n)$[5] of ISFSM's (parametrized in the number of states n) that have a number of maximal compatibles in the order of $3^{(n/3)}$ and a number of prime compatibles in the order of $2^{(2n/3)}$. $F(n)$ has 1 input and $n/3$ outputs. Each machine F is derived from a non-connected state transition graph whose components F_i are defined on the same input and outputs. Each FSM F_i has 3 states $\{s_{i0}, s_{i1}, s_{i2}\}$ and 3 specified transitions $\{e_{i0} = (s_{i0}, s_{i1}), e_{i1} = (s_{i1}, s_{i2}), e_{i2} = (s_{i2}, s_{i0})\}$. Each transition under the input set to 1 asserts all outputs to $-$, with the exception that e_{i0} and e_{i1} assert the i-th output to 0 and e_{i2} asserts the i-th output to 1. Under the input set to 0 the transitions are left unspecified.

Table 4 shows the results of running increasingly larger FSM's of the family. While ISM is able to generate sets of prime compatibles of cardinality up to 2^{1200} with reasonable running times, STAMINA, based on an explicit enumeration runs out of memory soon (and where it completes, it takes much longer).

8.6 FSM's with Many Maximals

Table 5 shows the results of running some examples from a set of FSM's constructed to have a large number of maximal compatibles. The examples *jac4, jc43, jc44, jc45, jc46, jc47* are due to R. Jacoby and have been kindly provided by J.-K. Rho of UC Boulder. The example *lavagno* is from asynchronous synthesis as those reported in Section 8.2. For these examples the program STAMINA was run with the option -M to compute all maximals. While ISM could complete on them in reasonable running times, STAMINA could not complete on *jac4* and completed the other ones with running times exceeding those of ISM by one or two order of magnitudes. Notice that ISM could also compute the set of all compatibles even though the computation of prime compatibles cannot be carried to the end while STAMINA failed on both.

[4]The incompatibility graph of an ISFSM F is a graph whose nodes are the states of F, with an undirected arc between two nodes s and t iff s and t are incompatible.

[5]Called *rubin* followed by n in the table of results.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	STAMINA
rubin12	12	3^4	2^8-1	2^8-1	2^8-1	0	4
rubin18	18	3^6	$2^{12}-1$	$2^{12}-1$	$2^{12}-1$	1	751
rubin24	24	3^8	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$	1	spaceout
rubin300	300	3^{100}	$2^{200}-1$	$2^{200}-1$	$2^{200}-1$	256	spaceout
rubin600	600	3^{200}	$2^{400}-1$	$2^{400}-1$	$2^{400}-1$	1995	spaceout
rubin900	900	3^{300}	$2^{600}-1$	$2^{600}-1$	$2^{600}-1$	6373	spaceout
rubin1200	1200	3^{400}	$2^{800}-1$	$2^{800}-1$	$2^{800}-1$	17711	spaceout
rubin1500	1500	3^{500}	$2^{1000}-1$	$2^{1000}-1$	$2^{1000}-1$	42674	spaceout
rubin1800	1800	3^{600}	$2^{1200}-1$	$2^{1200}-1$	$2^{1200}-1$	78553	spaceout

Table 4: Constructed FSM's.

machine	# states	# max compat.	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
jac4	65	3.859e6	4.159e7	?	34	spaceout
jc43	45	82431	1.556e6	?	13	7739
jc44	55	4785	7.584e9	?	20	662
jc45	40	17323	480028	?	10	1211
jc46	42	26086	1.153e6	?	11	2076
jc47	51	397514	1.120e7	?	19	41297
lavagno	65	47971	9.163e6	?	163	40472

Table 5: FSM's with many maximals.

8.7 Randomly Generated FSM's

We investigated also whether randomly generated FSM's have a large number of prime compatibles. A program was written to generate random FSM's[6]. A small percentage of the randomly generated FSM's were found to exhibit this behavior. Table 6 shows the results of running ISM and STAMINA on some interesting examples with a large number of primes. Again only ISM could complete the examples exhibiting a large number of primes.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
fsm15.232	14	4	7679	360	360	2	23
fsm15.304	14	2	12287	954	954	1	85
fsm15.468	13	2	4607	772	772	1	16
fsm15.897	15	2	20479	617	616	0	50
ex2.271	19	2	393215	96383	96382	26	spaceout
ex2.285	19	2	393215	121501	121500	17	spaceout
ex2.304	19	2	393215	264079	264079	94	spaceout
ex2.423	19	4	204799	160494	160494	112	spaceout
ex2.680	19	2	327679	192803	192803	156	spaceout

Table 6: Random FSM's.

8.8 Summary of the Results

The results of Tables 2, 3, 4, 5 and 6 show that when the sets of compatibles needed for exact state minimization are huge, an algorithm based on an explicit enumeration of those sets will be unable to complete due to an out-of-memory condition.

The question now arises of how it is realistic to expect such examples in logic design applications. One could object that the examples of Table 1 show that hand-designed FSM's can be handled very well by an existing state-of-art program like STAMINA. If this can be true for usual hand-designed FSM's, we argue that there are FSM's produced in the process of logic synthesis of real design applications that generate large sets of compatibles exceeding the capabilities of programs based on an explicit enumeration. The examples of Table 2 are such a case. They are FSM's produced as intermediate stages of an asynchronous logic design procedure and their minimization requires computing very large sets of compatibles. Another case is the one reported in Table 3, referring to the synthesis of finite state machines consistent with a collection of I/O learning examples.

9 Experimental Results of Binate Covering

In this section we report preliminary results of an implementation of the implicit binate covering algorithm, described in the previous sections. We use the benchmarks already introduced in Section 8 and concentrate on the examples where prime compatibles are needed to find a minimum solution of the state minimization problem [7]. Here we provide data for a subset of them, sufficient to characterize the capabilities of our

[6]Parameters: number of states, number of inputs, number of outputs, don't care output percentage, don't care target state percentage.

[7]Otherwise, a minimum solution of maximal compatibles is closed and therefore is a minimum solution.

prototype program. Since the following results are still preliminary, the analysis of the experiments is not final yet.

Comparisons are made with STAMINA. The binate covering step of STAMINA was run with alpha dominance and no row consensus, because beta dominance and row consensus have not yet been implemented in our implicit binate solver. Our implicit binate program currently lacks also routines for table partitioning and Gimpel's reduction rule, that were instead invoked in the version of STAMINA used for comparison. This might sometimes favour STAMINA, but for simplicity we will not elaborate further on this effect. In the near future we will implement beta dominance, row consensus and table partitioning in our package. All run times are reported in CPU seconds on a DEC DS5900/260 with 440 Mb of memory.

The following explanations refer to the tables of results:

- $\#\mathcal{NEPC}$ denotes the number of nonessential prime compatibles. Essential prime compatibles and rows covered by them are not needed in the binate table.
- 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.
- Data are reported with a $*$ in front, when only the first solution was computed.

9.1 Minimizing Small and Medium Examples

With the exception of *ex2*, *ex3*, *ex5*, *ex7*, the examples from the MCNC and asynchronous benchmarks do not require primes for exact state minimization and yield simple covering problems[8]. Table 7 reports the few non-trivial examples. They were all run to full completion, with the exception of *ex2*. In the case of *ex2*, we stopped both programs at the first solution. These experiments suggest that

- the number of recursive calls of the binate cover routine (*mincov*) of ISM and STAMINA is roughly comparable (ISM is surprisingly better in the examples *green* and *alex1*), showing that our implicit branching selection routine is satisfactory. This is an important indication, because selecting a good branching column is a more difficult task in the implicit frame.
- The running times are better for STAMINA in the small examples, but in the medium examples ISM recovers ground and it is sometimes much faster. This is to be expected because when the size of the table is small the implicit approach has no special advantage, but it starts to pay off scaling up the instances. Moreover, our implicit reduction computations have not yet been fully optimized.

9.2 Minimizing Constructed Examples

Table 8 presents a few examples from the constructed benchmarks. They yield giant binate tables. The experiments show that ISM is capable of reducing those table and of producing a minimum solution or at least a solution. This is beyond reach of an explicit technique and substantiates the claim that implicit techniques advance decisively the size of instances that can be solved exactly.

[8]Moreover, in the case of the asynchronous benchmark a more appropriate formulation of state minimization requires all compatibles and a different set-up of the covering problem.

machine	# states	$\#\mathcal{NEPC}$	table size (r x c) before red.	table size (r x c) after 1st red.	# mincov ISM	# mincov STAMINA	CPU time (sec) ISM	CPU time (sec) STAMINA
ex2	19	1366	4418 x 1366	3425 x 1352	*6	*6	*62	*116
ex3	10	91	243 x 91	151 x 84	217	160	85	0
ex5	9	38	81 x 38	47 x 31	11	23	4	0
ex7	10	57	137 x 57	62 x 44	34	31	9	0
green	54	524	53 x 524	51 x 524	5	2975	38	148
alex1	42	787	42 x 787	28 x 110	3	137	303	7

Table 7: Examples from the MCNC and asynchronous benchmarks

machine	# states	$\#\mathcal{NEPC}$	table size (r x c) before red.	table size (r x c) after 1st red.	# mincov ISM	# mincov STAMINA	CPU time (sec) ISM	CPU time (sec) STAMINA
ex2.271	19	96382	95323 x 96382	0 x 0	1	-	3	fails
ex2.285	19	121500	1 x 121500	0 x 0	1	-	0	fails
ex2.304	19	264079	1053189 x 264079	1052007 x 264079	2	-	554	fails
ex2.423	19	160494	637916 x 160494	636777 x 160494	*2	-	*373	fails
ex2.680	19	192803	757755 x 192803	756940 x 192803	2	-	868	fails

Table 8: Examples from the constructed benchmarks

9.3 Minimizing FSM's from Learning I/O Sequences

Examples in Table 8 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 9 answer in the affirmative the question. They are the simplest examples from the suite of FSM's described in Section 8.3. It is not possible to build and solve these binate tables with explicit techniques. Instead we can manipulate them with our implicit binate solver and find a solution. In the example *fourr.40*, only the first table reduction was performed. Notice that these are preliminary results. Experiments to find a minimum cost solution and to complete the benchmark are under progress.

10 Conclusions

This paper has presented an implicit algorithm for exact state minimization of incompletely specified FSM's (ISFSM's), an NP-hard problem [21]. It has been shown in the experimental sections that various applications of logic synthesis generate FSM's beyond the reach of state-of-art state minimizers. We have shown how to compute sets of maximal compatibles, compatibles and prime compatibles with implicit techniques and demonstrated that in this way it is possible to handle examples exhibiting a number of compatibles up to 2^{1200}, a number outside the scope of programs based on explicit enumeration [13]. The only explicit dependence is on the number of states of the initial problem. We have also indicated where such examples arise in practice. Then we have addressed the final step of an implicit exact state minimization procedure, i.e. solving a binate table covering problem [24]. We presented the first published algorithm for fully implicit exact binate covering. We report preliminary results of a prototype implementation capable of reducing

machine	# states	# $\mathcal{NEPC}$	table size (r x c) before red.	table size (r x c) after 1st red.	# mincov ISM	# mincov STAMINA	CPU time (sec) ISM	CPU time (sec) STAMINA
threer.20	21	3936	6977 x 3936	6974 x 3936	*4	*3	*14	*1788
threer.25	26	17372	35690 x 17372	34707 x 17016	*3	-	*74	fails
threer.30	31	33064	68007 x 33064	64311 x 32614	*4	-	*554	fails
threer.35	36	82776	177124 x 82776	165967 x 82038	*8	-	2390?	fails
threer.40	41	529420	1209783 x 529420	1148715 x 526753	*10	-	*5054	fails
fourr.16	21	3266	6060 x 3266	5235 x 3162	*2	*2	*7	*1266
fourr.20	21	12762	26905 x 12762	26904 x 12762	*2	-	*35	fails
fourr.30	31	542608	1396435 x 542608	1385809 x 542132	*2	-	*1317	fails
fourr.40	41	2.388e9	6.783e9 x 2.388e9	6.783e9 x 2.388e9	†1	-	†1651	fails

Table 9: Learning I/O sequences benchmark.

huge binate tables (up to 10^6 rows and columns) and of carrying on the branch-and-bound procedure on an implicit representation of the table. Exact solutions to problems beyond the reach of traditional tools are so found.

We underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this paper can be applied to other problems of logic synthesis and combinatorial optimization. For instance the implicit computation of maximal compatibles given here can be easily converted into an implicit computation of prime encoding-dichotomies (see [25]).

References

[1] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, 1990.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and R. Rudell. *Multi-level logic synthesis*. unpublished book, 1992.

[4] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[5] R. Bryant. Graph based algorithm for Boolean function manipulation. In *IEEE Transactions on Computers*, pages C–35(8):667–691, 1986.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using functional Boolean vectors. *IFIP Conference*, November 1989.

[7] O. Coudert, H.Fraisse, and J.C. Madre. A breakthrough in two-level logic minimization. In *The Proceedings of the Design Automation Conference*, June 1993.

[8] O. Coudert and J.C. Madre. Implicit and incremental computation of prime and essential prime implicants of boolean functions. In *The Proceedings of the Design Automation Conference*, pages 36–39, 1992.

[9] S. Edwards and A. Oliveira. Synthesis of minimal state machines from examples of behavior. *EE290LS Class Project Report, U.C. Berkeley*, May 1993.

[10] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[11] A. Grasselli and F. Luccio. Some covering problems in switching theory. In *Networks and Switching Theory*, pages 536–557. Academic Press, New York, 1968.

[12] G.Swamy, R.Brayton, and P.McGeer. A fully implicit Quine-McCluskey procedure using BDD's. In *Submitted for Publication*, 1993.

[13] G. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *The Proceedings of the European Design Automation Conference*, 1991.

[14] J.E. Hopcroft. n log n algorithm for minimizing states in finite automata. *Tech. Report Stanford Univ. CS 71/190*, 1971.

[15] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit generation of compatibles for exact state minimization. *Tech. Report No. UCB/ERL M93/60*, August 1993.

[16] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Book Company, New York, New York, second edition, 1978.

[17] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. *The Proceedings of the Design Automation Conference*, June 1992.

[18] B. Lin. Synthesis of VLSI designs with symbolic techniques. *Tech. Report No. UCB/ERL M91/105*, November 1991.

[19] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for multiple-valued functions. In *The Proceedings of the Design Automation Conference*, pages 40–44, 1992.

[20] B. Lin and A.R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *Proceedings of the International Conference on Computer Design*, pages 81–85, September 1991.

[21] C.P. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Transactions on Computers*, pages 1099–1102, October 1973.

[22] J.-K. Rho and F. Somenzi. The role of prime compatibles in the minimization of finite state machines. In *The Proceedings of the European Design Automation Conference*, 1992.

[23] Frank Rubin. Worst case bounds for maximal compatible subsets. *IEEE Transactions on Computers*, pages 830–831, August 1975.

[24] R. Rudell. Logic synthesis for VLSI design. *Tech. Report No. UCB/ERL M89/49*, April 1989.

[25] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A uniform framework for satisfying input and output encoding constraints. *The Proceedings of the Design Automation Conference*, June 1991.

[26] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *The Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[27] Huey-Yih Wang and R. K. Brayton. Input don't care sequences in fsm networks. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1993.