A Fully Implicit Algorithm for Exact State Minimization

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

Department of EECS, University of California at Berkeley, Berkeley, CA 94720

Abstract

State minimization of incompletely specified machines is an important step of FSM synthesis. An exact algorithm consists of generation of prime compatibles and solution of a binate covering problem. This paper presents an implicit algorithm for exact state minimization of FSM's. We describe how to do implicit prime computation and implicit binate covering. We show that we can handle sets of compatibles and prime compatibles of cardinality up to 2^{1500}. We present the first published algorithm for fully implicit exact binate covering. We show that we can reduce and solve binate tables with up to 10^6 rows and columns. The entire branch-and-bound procedure is carried on implicitly. We indicate also where such examples arise in practice.

1 Introduction

Implicit techniques are based on the idea of operating on discrete sets 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. BDD's can be manipulated efficiently with the usual Boolean operators.

Previous work showed how to compute implicitly the primes of a Boolean function and how to reduce implicitly the unate table of the Quine-McCluskey procedure to its cyclic core ([4, 8]). Exact solutions to problems too hard for ESPRESSO were found. Implicit techniques increase the size of problems that can be solved exactly in logic synthesis and verification.

This paper presents an implicit algorithm for exact state minimization of incompletely specified FSM's (ISFSM's), an NP-hard problem. The classical algorithm for state minimization of ISFSM's [6] reduces the problem to the computation of prime compatibles and the selection of a minimum closed set of them by means of a binate covering step [14]. To compute prime compatibles one must examine compatible sets of states. If an FSM has too many compatibles, either the prime computation or the binate covering step will be intractable with explicit techniques. Interesting classes of FSM's yield such intractable problems.

In this paper we describe how to do implicit prime computation and implicit binate covering. Since generation of compatibles and solution of binate covering are common problems in logic synthesis, the techniques that we are going to describe have a large applicability. We show that we can handle sets of compatibles and prime compatibles of cardinality up to 2^{1500}, a size clearly unattainable by explicit enumeration. We present the first implicit exact algorithm for binate covering. We report results of an implementation capable of reducing and solving huge binate tables (up to 10^6 rows and columns). The entire branch-and-bound procedure is carried on implicitly. Previous implicit implementations of unate covering did not address the problem of finding implicitly a branching column and a lower bound.

The remainder of the paper is organized as follows. Section 2 introduces implicit representations and manipulations. Algorithms for implicit generation of compatibles are presented in Section 3. Section 4 gives 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 other implicit table manipulations are briefly given in Section 7. Results on a variety of benchmarks are reported and discussed in Section 8. Conclusions and future work are summarized in Section 9. These implicit algorithms are discussed in greater lengths in [9].

2 Implicit Representations and Manipulations

Algorithms for sequential synthesis have been developed primarily for State Transition Graphs (STG's). STG's have been usually represented in two-level form where state transitions are stored explicitly, one by one. Alternatively, STG's can be represented implicitly with Binary Decision Diagrams (BDD's) [2, 1]. BDD's represent Boolean functions (e.g. characteristic functions of sets and relations) and have been amply reported in the literature [2, 1], to which we refer.[1]

2.1 Implicit FSM Representation

A Finite State Machine (FSM) can be represented by a 5-tuple $(I, O, S, \mathcal{T}, \mathcal{O})$. I and O are the sets of input patterns and output patterns. S is the set of states. $\mathcal{T} \subseteq I \times S \times S$ is the transition relation that relates a next state to an input and a present state. $\mathcal{O} \subseteq I \times S \times O$ is the output relation that relates an output to an input and a present state. An FSM, where each (input, state) pair is related to exactly one next state and one output, is a **completely specified FSM**. An **incompletely specified FSM** is one where either the next state or the output is not specified for at least one (input, state) pair.

If a next state is unspecified, no transitions on the (input, state) pair need to be considered for the purpose of state minimization, so they are omitted from $\mathcal{T}$. On the other hand, we represent all unspecified output patterns in $\mathcal{O}$ corresponding to an (input, state) pair, to ensure correctness of the output compatibles computation described in Section 3.1. The **transition and output relations** are given by:

$\mathcal{T}(i,p,n) = 1$ iff n is the specified next state of state p on input i
$\mathcal{O}(i,p,o) = 1$ iff o is a (possibly unspecified) output of state p on i

where i and o are Boolean vectors of signals while p and n are represented by positional-sets defined below.

2.2 Positional-set Representation

To perform state minimization, one needs to represent and manipulate efficiently sets of states, or state sets, (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 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, each subset of states is represented in **positional-set** 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.[2] For example, if $n = 6$, the set

[*]Research supported by DARPA under contract J-FBI90-073 and California State MICRO Program

[†]Research supported by NSF under contract MIP-8719546 and California State MICRO Program

[1]$\exists x(\mathcal{F})$ ($\forall x(\mathcal{F})$) denotes the existential (universal) quantification of function $\mathcal{F}$ over variables x; $\Rightarrow$ denotes Boolean implication; $\Leftrightarrow$ denotes XNOR; $\neg$ denotes NOT.

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

with a single state s_4 is represented by 000100 while the set of states $s_2s_3s_5$ is represented by 011010.

A set of sets of states is represented as a set S of positional-sets by a characteristic function $\chi_S : B^n \to B$ 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.

2.3 Operations on Positional-sets

With our definitions of relations and positional-set notation for representing set of states, useful operators on sets and sets of sets can be derived. We have proposed in [9] a unified notational framework for set manipulation, extending the work by Lin *et al.* in [11]. Here we define some relationships between two sets, between two sets of sets, between a set and a set of sets, etc.

Theorem 2.1 *Set* **equality, containment** *and* **strict-containment** *between two positional-sets x and y can be captured by the following constraints: $(x = y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k$; $(x \supseteq y) = \prod_{k=1}^{n} y_k \Rightarrow x_k$; and $(x \supset y) = (x \supseteq y) \cdot (x \neq y)$.*

Theorem 2.2 *Given two sets of positional-sets,* **complementation**, **union**, **intersection**, *and* **sharp** *can be performed on them as logical operations $(\neg, +, \cdot, \cdot\neg)$ on their characteristic functions.*

Theorem 2.3 *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:*

$$Set\_Contain_x(\chi_A, \chi_B) = \forall x\, [\chi_B(x) \Rightarrow \chi_A(x)]$$

Theorem 2.4 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets,* **set union** *defines a positional-set y which represents the union of all state sets in A, and can be computed by:*

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

For each bit 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.

Theorem 2.5 *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 is given by:*

$$Maximal_x(\chi_F) = \chi_F(x) \cdot \nexists y\, [\chi_F(y) \cdot (y \supset x)]$$

The term $\exists y\, [\chi_F(y) \cdot (y \supset x)]$ is true iff there is a positional-set y in χ_F such that $y \supset x$. In such a case, x cannot be in the maximal set by definition, and are taken away from $\chi_F(x)$.

2.4 k-out-of-n Positional-sets

We define a family of sets of state sets, $Tuple_{n,k}(x)$, which contain all positional-sets $x \subseteq S$ with exactly k states in them. Their BDD's can constructed by the following algorithm, by calling $Tuple(n, k)$:

```
Tuple($i, j$) {
    if ($j < 0$) or ($i < j$)   return **0**
    if ($i = 0$) and ($i = j$)  return **1**
    return ITE($x_i$, Tuple($i-1, j-1$), Tuple($i-1, j$))
}
```

$Tuple(i, j)$ contains positional-sets of cardinality j with i variables, $x_1, x_2, \ldots, x_i$, which can be grouped into those that include state i and those that do not. The latter group simply corresponds to $Tuple(i-1, j)$, the set of positional-sets of cardinality j with only $x_1, x_2, \ldots, x_{i-1}$ (using one less variable). The former group can be obtained by adding state i to each positional-set in $Tuple(i-1, j-1)$, the set of positional-sets of cardinality $j-1$ with $i-1$ variables. Therefore $Tuple(i, j)$ can be computed recursively by $ITE(x_i, Tuple(i-1, j-1), Tuple(i-1, j))$. Recursion can stop when a termination condition as shown is met. The BDD size and time complexity of $Tuple(n, k)$ are both $O(nk)$, provided its intermediate results are memoized in a computed table ([1]).

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 ([13]), and the generation of the whole set of compatibles can be a challenging task.

The covering step (described in Sections 4 to 7) 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.

3.1 Output Incompatible Pairs

To generate compatibles, incompatibility relations between pairs of states are derived first from the given output and transition relations.

Definition 3.1 *Two states are an* **output incompatible pair** *if, for some input, they cannot generate the same output. The set of output incompatible pairs, $\mathcal{OICP}(y, z)$, can be computed as:*

$$\mathcal{OICP}(y, z) = Tuple_1(y) \cdot Tuple_1(z) \cdot \exists i\ \nexists o\, [\mathcal{O}(i, y, o) \cdot \mathcal{O}(i, z, o)]$$

Although y and z can represent any positional-sets, the conditions $Tuple_1(y) \cdot Tuple_1(z)$ restrict them to represent only pairs of 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 (i.e., output incompatible).

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 an incompatible pair. The set of incompatible pairs is the least fixed point of $\mathcal{ICP}$:*

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

and can be computed by the following iteration:

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

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

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 states (y, z) 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 cardinalities.

Definition 3.3 *A set of states is an* **incompatible** *if it contains at least one incompatible pair. The set of incompatibles can be computed as:*

$$\mathcal{IC}(c) = \exists y, z\, [\mathcal{ICP}(y, z) \cdot \prod_{k=1}^{n} y_k + z_k \Rightarrow c_k]$$

$\prod_{k=1}^{n} y_k + z_k \Rightarrow c_k$ performs bitwise OR on singletons y and z. If either of their k-th bits is 1, the corresponding c_k bit is constrained to 1. Otherwise, c_k can take any values. The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. Thus, it generates all positional-sets c which contain the union of the positional-sets y and z. The whole computation defines all state sets c each of which contains 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. The set of compatibles, $\mathcal{C}(c)$, can be computed as:*

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

$\mathcal{C}(c)$ 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 also need to compute the closure conditions for each compatible. This is done by finding the class set of a compatible, i.e., 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.*

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 state 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 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 $\mathcal{F}(c, i, n)$. All 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 3.1 *The implied sets d (in positional-set form) of a compatible c for all inputs are computed by the relation $\mathcal{CI}(c, d)$ 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 n 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 inputs i are existentially quantified from the implied sets of c of different inputs.

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 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 3.2 *The class set of a compatible c is defined 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 excluded according to condition 1 in Definition 3.6. By condition 2, we prune away implied sets d which are contained in their 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 $\mathcal{CI}(c, d)$.

3.7 Prime Compatibles

To solve exactly the covering problem, it is sufficient to consider a subset of compatibles called prime compatibles. As proved in [6], 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., c' dominates 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 3.3 *The prime dominance relation is given by:*

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

The two terms on the right express 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 2.1. 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 $\forall d \; \mathcal{CCS}(c', d) \Rightarrow \mathcal{CCS}(c, d)$, as described by Theorem 2.3.

Definition 3.8 *A* **prime compatible** *is a compatible not dominated by another compatible. The set of prime compatibles is given by:*

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

Compatibles c that are dominated by some compatible c' are computed by the expression $\exists c' \; [\mathcal{C}(c') \cdot Dominate(c', c)]$. By definition, the set of prime compatibles is simply given by the set of compatibles $\mathcal{C}(c)$ excluding those that are dominated.

4 Implicit Binate Covering

The classical branch-and-bound algorithm for minimum-cost binate covering has been described in [6, 7] and implemented by means of efficient computer programs (ESPRESSO and STAMINA). The branch-and-bound solution of minimum binate covering is based on the following recursive procedure. In our implicit formulation, we keep the 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 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, and implicit table partitioning.

```
mincov($R, C, U$) {
    ($R, C$) = Reduce($R, C, U$)
    if (Terminal_Case($R, C$))
        if (cost($R, C$) $\geq U$)  return no solution
        else  $U$ = cost($R, C$); return solution
    $L$ = Lower_Bound($R, C$)
    if ($L \geq U$)  return no solution
    $c_i$ = Choose_Column($R, C$)
    $S^1$ = mincov($R_{c_i}, C_{c_i}, U$)
    $S^0$ = mincov($R_{\overline{c_i}}, C_{\overline{c_i}}, U$)
    return Best_Solution($S^1 \cup \{c_i\}, S^0$)
}
```

At each call of the binate cover routine *mincov*, the binate table undergoes a reduction step $Reduce$ and, if termination 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 with an upper bound U (best solution obtained so far). A good lower bound is based on the computation of a maximal independent set.

5 Implicit Covering Table Generation

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 row and column labels and to exploit all the special features of the binate covering problem induced by state minimization (for instance, each row has at most one 0).

Definition 5.1 *A column is labelled by a positional-set p. The set of* **column labels** *C is obtained by 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 0 and 1's in the row. Each row label r consists of a pair of positional-sets (c, d). Since there is at most one 0 in the row, 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 the row, c is set to the empty set, $Tuple_0(c)$. Therefore a row label r corresponds to a unate clause iff relation $unate\_row(r) = Tuple_0(c)$ is true. Because of Definition 5.3 for row labels, the columns intersecting a row labelled $r = (c, d)$ in a 1 are labelled by the prime compatibles p that contain d. i.e.,

Definition 5.2 *The table entry at the intersection of a row labelled by $r = (c, d) \in R$ and a column labelled by $p \in C$ can be inferred by:*

the table entry is a 0 iff relation $\mathtt{0}(r, p) \overset{\text{def}}{=} (p = c)$ is true,

the table entry is a 1 iff relation $\mathtt{1}(r, p) \overset{\text{def}}{=} (p \supseteq d)$ is true.

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

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

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. 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 $p = c$, and the row has 1's in the columns labelled by p_i wherever $(p_i \supseteq d)$.

The covering conditions require that each state be contained by some prime compatible in the solution. For each state $d \in S$, 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 set to $Tuple_0(c)$. The 1 entries are correctly positioned at the intersection with all columns labelled by prime compatibles p_i which contain the singleton state d.

From now on, we will use c as column label and $C(c)$ will be the set of column labels, as we no longer manipulate compatibles.

6 Implicit Reduction Techniques

Reduction rules aim to 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 minimal 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). The reductions and order implemented in our reduction algorithm are summarized as follows:

```
Reduce($R, C, U$) {
  repeat {
    Collapse_Columns($C$); Column_Dominance($R, C$)
    $Sol = Sol \cup$ Essential_Columns($R, C$)
    if ($|Sol| \geq U$) return no solution
    Unacceptable_Columns($R, C$); Unnecessary_Columns($R, C$)
    if ($C$ does not cover $R$) return no solution
    Collapse_Rows($R$); Row_Dominance($R, C$)
  } until (both $R$ and $C$ unchanged)
  return ($R, C$)
}
```

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 or equal than the upper bound 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.
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 implicitly using BDD's on the special table representation described in the previous section.

6.1 Duplicated Rows and Columns

It is possible that 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). This avoids the problem of columns (rows) dominating each other when performing implicitly column (row) dominance. The following computations can be seen as finding the equivalence relation of duplicated columns (rows) and selecting one representative for each equivalence class.

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

$$dup\_col(c', c) = \forall r \, [R(r) \cdot \neg \mathtt{0}(r, c') \cdot \neg \mathtt{0}(r, c) \cdot (\mathtt{1}(r, c') \Leftrightarrow \mathtt{1}(r, c))]$$

$$C(c) = C(c) \cdot \not\exists c' \, [C(c') \cdot dup\_col(c', c) \cdot (c' \prec c)]$$

$$dup\_row(r', r) = \forall c \, [C(c) \cdot (\mathtt{0}(r', c) \Leftrightarrow \mathtt{0}(r, c)) \cdot (\mathtt{1}(r', c) \Leftrightarrow \mathtt{1}(r, c))]$$

$$R(r) = R(r) \cdot \not\exists r' \, [R(r') \cdot dup\_row(r', r) \cdot (r' \prec r)]$$

For the column labels c' and c to be in the relation dup_col, the first equation requires the following conditions to be met for every row label $r \in R$. Since each row has at most one 0, the row labelled r cannot intersect either column at a 0, (i.e., $\neg \mathtt{0}(r, c') \cdot \neg \mathtt{0}(r, c)$). In addition, the entry (r, c) is a 1 iff the entry (r, c') is a 1, (i.e., $\mathtt{1}(r, c') \Leftrightarrow \mathtt{1}(r, c)$).

The second computation picks a representative column label out of a set of columns labels corresponding to duplicated columns. A column label c is deleted from C iff there is a column label c' which has a smaller binary value than c and both label duplicated columns. Here we exploit the fact that any positional-set c 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.

Detection of duplicated rows, selection of a representative row, and table updating are performed by the third and last equations as in the case of duplicated columns.

From now on, sometimes we will blur the distinction between a column (row) label and the column (row) itself, but the context should say clearly which one it is meant.

6.2 Column Dominance

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

$$\alpha\_dom(c', c) = \not\exists r\ \{R(r) \cdot [\mathtt{1}(r, c) \cdot \neg \mathtt{1}(r, c') + \mathtt{0}(r, c')]\}$$

For column c' to α-dominate c, the right-hand expression ensures that there is not a row $r \in R$ such that either the table entry (r, c) is a 1 but the table entry (r, c') is not, *or* the table entry (r, c') is a 0.

Definition 6.2 *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.*

$$\beta\_dom(c', c) = \not\exists r'\ \{R(r') \cdot [\mathtt{1}(r', c) \cdot \neg \mathtt{1}(r', c') + \ \mathtt{0}(r', c') \cdot \not\exists r \\ [R(r) \cdot \mathtt{0}(r, c) \cdot \not\exists c''\ [C(c'') \cdot (c'' \neq c') \cdot \mathtt{1}(r, c'') \cdot \neg \mathtt{1}(r', c'')]\ ]]\}$$

According to the definition, the table should *not* contain a row $r' \in R$ if either of the following two cases is true at that row: (1) table entry at column c is a 1 while entry at column c' is not a 1 (i.e., $\mathtt{1}(r', c) \cdot \neg \mathtt{1}(r', c')$), *or* (2) c' has a 0 in row r' (i.e., $\mathtt{0}(r', c')$) but there does not exist a row $r \in R$ such that its column c is a 0 and disregarding entries in column c', row r' has all the 1's of row r. Rephrasing the last part of the condition 2, the expression $\not\exists c''\ [C(c'') \cdot (c'' \neq c') \cdot \mathtt{1}(r, c'') \cdot \neg \mathtt{1}(r', c'')]$ requires that there is no column $c'' \in C$ apart from column c' such that c'' has a 1 in row r, but not in row r'.

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 for col_dom. The set of dominated columns to be deleted from the table can be computed as:

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

A column $c \in C$ is dominated if there is another *different* $c' \in C$ which column dominates c.

Theorem 6.2 *The following computations delete a set of columns $D(c)$ from a table (R, C) and all rows intersecting these columns in a 0.*

$$\begin{aligned} C(c) &= C(c) \cdot \neg D(c) \\ R(r) &= R(r) \cdot \not\exists c\ [D(c) \cdot \mathtt{0}(r, c)] \end{aligned}$$

The first computation removes columns in $D(c)$ from the set of columns $C(c)$. The expression $\exists c\ [D(c) \cdot \mathtt{0}(r, c)]$ defines all rows r intersecting the columns in D in a 0. They are deleted from R.

6.3 Row Dominance

Definition 6.3 *A row r' **dominates** another row r if r has all the 1's and 0 of r'. Reduction by row dominance can be computed by:*

$$row\_dom(r', r) = \not\exists c\ \{C(c) \cdot [\mathtt{1}(r', c) \cdot \neg \mathtt{1}(r, c) + \mathtt{0}(r', c) \cdot \neg \mathtt{0}(r, c)]\}$$

$$R(r) = R(r) \cdot \not\exists r'\ [R(r') \cdot (r' \neq r) \cdot row\_dom(r', r)]$$

For r' to dominate r, the first equation requires that there is no column $c \in C$ such that either the table entry (r', c) is a 1 but the entry (r, c) is not, *or* the entry (r', c) is a 0 but the entry (r, c) is not. The second equation says that any row $r \in R$, dominated by another *different* row $r' \in R$, is deleted from the set of rows $R(r)$ in the table.

6.4 Essential and Unacceptable Columns

Definition 6.4 *A column c is an **essential column** if there is a row having a 1 in column c and 2 everywhere else.*

$$\begin{aligned} ess\_col(c) &= C(c) \cdot \exists r\ \{R(r) \cdot unate\_row(r) \cdot \mathtt{1}(r, c) \\ &\quad \cdot \not\exists c'\ [C(c') \cdot (c' \neq c) \cdot \mathtt{1}(r, c')]\ \} \end{aligned}$$

For a column $c \in C$ to be essential, there must exists a row $r \in R$ which (1) does not contain any 0 (i.e., $unate\_row(c)$), (2) contains a 1 in column c (i.e., $\mathtt{1}(r, c)$), and (3) there is not another *different* column intersecting the row in a 1 (i.e., $\not\exists c'\ [C(c') \cdot (c' \neq c) \cdot \mathtt{1}(r, c')]$).

Theorem 6.3 *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. The following computations add essential columns to the solution, delete them from the set of columns and delete all rows in which they have 1's:*

$$\begin{aligned} solution(c) &= solution(c) + ess\_col(c) \\ C(c) &= C(c) \cdot \neg ess\_col(c) \\ R(r) &= R(r) \cdot \not\exists c\ [ess\_col(c) \cdot \mathtt{1}(r, c)] \end{aligned}$$

The first two equations move the essential columns from the column set to the solution set. The last equation deletes from the set of rows R all rows intersecting an essential column c in a 1.

Definition 6.5 *A column c is an **unacceptable column** if there is a row having a 0 in column c and 2 everywhere else.*

$$unacceptable\_col(c) = C(c) \cdot \exists r\ \{R(r) \cdot \mathtt{0}(r, c) \cdot \not\exists c'\ [C(c') \cdot \mathtt{1}(r, c')]\}$$

For column $c \in C$ to be unacceptable, there must be a row $r \in R$ which intersects the column c at a 0 and no column c' intersects that row r in a 1 (i.e., $\not\exists c'\ [C(c') \cdot \mathtt{1}(r, c')]$).

Definition 6.6 *A column is an **unnecessary column** if it does not have any 1 in it.*

$$unnecessary\_col(c) = C(c) \cdot \not\exists r\ [R(r) \cdot \mathtt{1}(r, c)]$$

A column $c \in C$ is unnecessary if no row $r \in R$ intersects it in a 1.

Theorem 6.4 *Unacceptable and unnecessary columns should be eliminated from the table, together with all the rows in which such columns have 0's. The table (R, C) is updated according to Theorem 6.2 by setting $D(c) = unacceptable\_col(c) + unnecessary\_col(c)$.*

7 Other Implicit Covering Table Manipulations

To have a fully implicit binate covering algorithm as described in Section 4, 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.

7.1 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 values 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:

```
Lmax($F, r$) {
    $v$ = bdd_top_var($F$)
    if ($v \in r$) return ($\mathbf{1}$, 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($v, T, \mathbf{0}$)
        else if ($count = count\_E$)  $G$ = ITE($v, \mathbf{0}, E$)
        return ($G, count$)
    }
}
```

$Lmax$ takes a relation $F(r,c)$ and the variables 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 memoized 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.

7.2 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 $F'(r,c) = R(r)\cdot C(c)\cdot \mathtt{1}(r,c)$ which evaluates true iff 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.

7.3 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 $F''(c,r) = R(r)\cdot unate\_row(r)\cdot C(c)\cdot \mathtt{1}(r,c)$. 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 $max(count\_T, count\_E)$ with $min(count\_T, count\_E)$. Once a shortest row, $shortest(r)$, is selected, all rows having 1-elements in common with $shortest(r)$ are discarded from $F''(c,r)$ by:

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

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 $shortest(r)$ so selected.

7.4 Implicit Covering Table Partitioning

If a covering table can be partitioned into n disjoint blocks, the minimum covering for the original table is the union of the minimum coverings for the n sub-blocks. n-way partitioning can be accomplished by successive extraction of disjoint blocks from the table. When the following iteration reaches a fixed point, (R_k, C_k) corresponds to a disjoint sub-block in (R,C).

$$\begin{aligned} R_0(r) &= Lmax(R(r)\cdot C(c)\cdot[\mathtt{0}(r,c)+\mathtt{1}(r,c)]\ ,\ c) \\ C_k(c) &= C(c)\cdot \exists r\ \{R_{k-1}(r)\cdot[\mathtt{0}(r,c)+\mathtt{1}(r,c)]\} \\ R_k(r) &= R(r)\cdot \exists c\ \{C_k(c)\cdot[\mathtt{0}(r,c)+\mathtt{1}(r,c)]\} \end{aligned}$$

This sub-block should be extracted from the table (R,C) and the above iteration can be applied again to the remaining table, until the table becomes empty. [9] provides a more detailed explanation.

8 Experimental Results

We implemented the algorithms described in the previous sections in a program called ISM, an acronym for Implicit State Minimizer. We ran ISM on different suites of FSM's. They are: the MCNC benchmark and other examples, FSM's from asynchronous synthesis [10], FSM's from learning I/O sequences [5], FSM's from synthesis of interacting FSM's [15], constructed FSM's that exhibit a large number of maximal and prime compatibles, random FSM's. Each suite has different features with respect to state minimization. We present in two tables the most interesting experiments. Table 1 summarizes the results of computing prime compatibles. Table 2 summarizes the results of solving binate covering. Examples with a few compatibles were not included in Table 1. Examples where primes are not needed to find a minimum FSM were not included in Table 2. Comparisons are made with STAMINA ([12], shortened as STAM in the tables), a program that represents the state-of-art for state minimization based on explicit techniques. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory.

8.1 Computation of Compatibles

Table 1 reports the numbers of compatibles and prime compatibles of FSM's from various benchmarks. The CPU time refers to the computation of prime compatibles. For these experiments STAMINA was run with the option **-P** to compute all primes. Compatibles are an important measure of complexity because they are the candidates from which prime compatibles are selected. There are no interesting examples from the MCNC benchmark or similar hand-designed FSM's. In those cases an explicit algorithm is sufficient to get a quick answer and it may be faster than an implicit one. The reason is that ISM manipulates relations having a number of variables linearly proportional to the number of states. When there are many states and few compatibles, the purpose of ISM is defeated and its representation becomes inefficient.

			# prime	CPU time (sec)	
fsm	# states	# compat.	compat.	ISM	STAM
alex1	42	55928	787	24	16
intel_edge.dummy	28	9432	396	37	3
isend	40	22207	480	13	fails
pe-rcv-ifc.fc	46	1.528e11	148	114	fails
pe-rcv-ifc.fc.m	27	1.793e6	38	3	147
pe-send-ifc.fc	70	5.071e17	506	571	fails
pe-send-ifc.fc.m	26	8.978e6	23	3	312
vbe4a	58	1.756e12	2072	109	167
vmebus.master.m	32	5.049e7	28	26	fails
th.30	31	97849	33064	21	17256
th.40	41	1.456e6	529420	75	fails
th.55	55	3.622e7	1.555e7	1273	fails
fo.20	21	42193	12762	2	1369
fo.50	51	3.643e7	1.696e7	216	fails
fo.70	71	9.621e10	4.524e10	22940	fails
ifsm0	38	1064973	18686	43	4253
ifsm1	74	43006	8925	25	466
ifsm2	150	497399	774	267	356
rubin18	18	$2^{12}-1$	$2^{12}-1$	0	751
rubin600	600	$2^{400}-1$	$2^{400}-1$	1978	fails
rubin1200	1200	$2^{800}-1$	$2^{800}-1$	27105	fails
rubin2250	2250	$2^{1500}-1$	$2^{1500}-1$	271134	fails
e271	19	393215	96383	21	fails
e285	19	393215	121501	13	fails
e304	19	393215	264079	93	fails
e423	19	204799	160494	102	fails
e680	19	327679	192803	151	fails

Table 1: Computation of compatibles.

The examples from *alex1* to *vmebus.master.m* are FSM's generated as intermediate steps of an asynchronous synthesis procedure [10].

STAMINA failed on the examples *isend, pe-rcv-ifc.fc, pe-send-ifc.fc, vmebus.master.m,* while ISM was able to complete them. The running times of ISM track well with the size of the set of compatibles and when both programs complete they are usually well below those of STAMINA (*pe-rcv-ifc.fc.m, pe-send-ifc.fc.m, vbe4a*). 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 [10].

The examples from *th.30* to *fo.70* come from a set of FSM's constructed to be compatible with a given collection of examples of input/output behavior [5]. Here ISM shows all its power compared to STAMINA, both for the number of computed primes and running time. STAMINA fails on the examples from *th.25* and *fo.20* onwards and, when it completes, it takes almost two orders of magnitude more time than ISM. The examples *ifsm0, ifsm1, ifsm2* come from a set of FSM's produced by FSM optimization, using the input don't care sequences induced by a surrounding network of FSM's [15]. They exhibit often large number of prime compatibles.

The examples prefixed by *rubin* have been constructed to have a number of prime compatibles exponential in the number of states [13]. ISM is able to generate sets of prime compatibles of cardinality up to 2^{1500} with reasonable running times, unattainable for explicit enumeration. The examples from *e271* to *e680* have been randomly generated. Again only ISM could complete those exhibiting many primes.

8.2 Solution of Binate Covering

Table 2 reports results of the implicit binate covering algorithm implemented in ISM vs. the explicit one available in STAMINA. CPU time refers only to binate covering without the time to find prime compatibles. Data are given both for α and β dominance. Under table size we provide the dimensions of the initial binate table (rows times columns). # 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. Data are reported with a † 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). The examples are from the same benchmarks presented before.

fsm	table size	# mincov		# cover	CPU time (sec)	
		ISM	STAM	ISM	ISM	STAM
	r x c	α/β	α/β	α/β	α/β	α/β
ex2	4e3x1e3	*6/*14	*6/286	*10/*12	*58/*293	*116/2100
ex3	243x91	201/37	91/39	4/4	78/33	0/0
ex5	81x38	16/6	10/6	3/3	4/3	0/0
ex7	137x57	38/31	37/6	3/3	8/12	0/0
e271	9e4x9e4	1/1	-/-	2/2	1/55	fails/fails
e285	1x1e5	1/1	-/-	2/2	0/0	fails/fails
e304	1e6x2e5	2/-	-/-	2/-	463/fails	fails/fails
e423	6e5x1e5	*2/-	-/-	*3/-	*341/fails	fails/fails
e680	7e5x1e5	2/-	-/-	2/-	833/fails	fails/fails
th.20	6e3x3e3	*4/*6	*5/*3	*5/*5	*13/*26	*1996/*677
th.25	3e4x1e4	*3/*6	-/-	*5/*6	*69/*192	fails/fails
th.30	6e4x3e4	*4/*9	-/-	*8/*8	*526/*770	fails/fails
th.35	1e5x8e4	*8/*9	-/-	*12/*10	*2296/*2908	fails/fails
th.40	1e6x5e5	*8/-	-/-	*12/-	*6787/fails	fails/fails
fo.16	6e3x3e3	*2/*3	*3/*3	*3/*3	*6/*23	*1641/*513
fo.16	6e3x3e3	*2/623	*3/377	*3/3	*6/9194	*1641/1459
fo.20	2e4x1e4	*2/*4	-/-	*4/*4	*31/*68	fails/fails
fo.30	1e6x5e5	*2/*5	-/-	*4/*5	*1230/*1279	fails/fails
fo.40	6e9x2e9	†1/-	-/-	†-/-	†723/fails	fails/fails
ifsm1	1e4x8e3	*4/2	*10/3	*14/14	*388/864	*17582/805
ifsm2	1e3x774	4/3	41/44	9/9	136/230	49/3

Table 2: Solution of binate covering.

With the exception of *ex3, ex5, ex7* from the MCNC benchmark (where as expected ISM takes more time than STAMINA), the other examples generate large covering tables. Some of them are the largest binate tables ever mentioned in the literature (up to 10^9 rows and columns). The experiments show that ISM is capable of building and reducing those table and of producing a minimum solution or at least a solution. This achievement is beyond the reach of explicit techniques and substantiates the claim that implicit techniques advance decisively the size of instances that can be solved exactly.

When both programs complete, the number of recursive calls of the binate cover routine is often comparable for ISM and STAMINA. There are some exceptions and for those STAMINA is usually better. This indicates that our implicit branching selection is good, but still short of the target. We are aware of more optimizations that can improve the speed and increase the applicability of our implicit binate solver.

9 Conclusions and Future Work

We have presented an implicit algorithm for exact state minimization of ISFSM's. We have described how to do implicit prime computation and implicit binate covering. Sets of compatibles of size up to 2^{1500} have been generated. Tables with up to 10^6 rows and columns have been solved. We have also indicated where such examples arise in practice. The only explicit dependence is on the number of states of the FSM.

The implicit computations presented here to solve binate covering exploit some restrictions on the instances occurring in state minimization of ISFSM's, e.g., the fact that binate clauses have exactly one zero. This pays off in terms of computational efficiency. Moreover, typical occurrences of binate covering in logic synthesis share this feature. Our technique can be extended to general binate covering problems. How much generality one can afford and still expect efficiency is a matter of applications and object of current research.

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, June 1990.

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

[3] 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, June 1992.

[4] O. Coudert, J.C. Madre, and H. Fraisse. A new viewpoint on two-level logic minimization. In *The Proceedings of the Design Automation Conference*, pages 625–630, June 1993.

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

[6] 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.

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

[8] G.Swamy, R.Brayton, and P.McGeer. A fully implicit Quine-McCluskey procedure using BDD's. *Tech. Report No. UCB/ERL M92/127*, 1992.

[9] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. *Tech. Report No. UCB/ERL M93/79*, November 1993.

[10] 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*, pages 568–572, June 1992.

[11] 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, June 1992.

[12] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design*, 13(2):167–177, February 1994.

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

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

[15] H.-Y. Wang and R. K. Brayton. Input don't care sequences in FSM networks. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 321–328, November 1993.