Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

IMPLICIT COMPUTATION OF COMPATIBLE SETS FOR STATE MINIMIZATION OF ISFSM'S

by

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

Memorandum No. UCB/ERL M95/106

19 December 1995

IMPLICIT COMPUTATION OF COMPATIBLE SETS FOR STATE MINIMIZATION OF ISFSM'S

by

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

Memorandum No. UCB/ERL M95/106

19 December 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Implicit Computation of Compatible Sets for State Minimization of ISFSM's

Timothy Kam[1] Tiziano Villa[2] Robert K. Brayton[2]
Alberto L. Sangiovanni-Vincentelli[2]

[1]Intel Development Labs
Intel Corporation
Hillsboro, Oregon 97124-6497

[2]Department of EECS
University of California at Berkeley
Berkeley, CA 94720

December 19, 1995

Abstract

Computation of sets of compatibles of incompletely specified finite state machines (ISFSM's) is a key step in sequential synthesis. This paper presents implicit computations to obtain sets of maximal compatibles, compatibles, prime compatibles, implied sets and class sets. The computations are implemented by means of BDD's that realize the characteristic functions of these sets. We have demonstrated with experiments from a variety of benchmarks that implicit techniques allow to handle examples exhibiting a number of compatibles up to 2^{1500}, an achievement outside the scope of programs based on explicit enumeration [19]. We have shown in practice that ISFMS's with a very large number of compatibles may be produced as intermediate steps of logic synthesis algorithms, for instance in the case of asynchronous synthesis [10]. This shows that the proposed approach has not only a theoretical interest, but also practical relevance for current logic synthesis applications, as shown by its application to ISFSM state minimization [8].

1 Introduction

Finite state machines are a common formalism to describe sequential systems. Incompletely specified FSM's (ISFSM's) are one of the most useful subclasses of FSM's, because they capture naturally a family of input-output behaviors, any of which is a valid implementation of the original specification. The choice of which input-output behavior to implement may be dictated by different criteria. A common one is the minimization of the number of states of the deterministic automaton corresponding to the chosen behavior. Another criterion may be the implementability of the chosen deterministic FSM within a network of FSM's [27].

It has been shown [15, 7] that all contained behaviors can be explored by means of collections of compatibles, called closed sets. To explore closed sets, one must compute maximal compatibles, prime compatibles, class sets of compatibles and other related sets and subsets of sets of states. The number of compatibles can be exponential in the number of states of the original ISFSM. This may be a problem for computations based on the explicit enumeration of compatibles and their subsets. As an alternative we

propose an algorithmic frame based on the representation and computation of the characteristic functions of these sets by means of binary decision diagrams (BDDs). We show how compatibles, maximal compatibles, prime compatibles and class sets can be computed with BDD-based techniques and demonstrate that it is possible to handle examples exhibiting a number of compatibles up to 2^{1500}, an achievement outside the scope of algorithms based on explicit enumeration [19]. We indicate also where such examples arise in practice. We refer to [8] for an implicit solution to the binate covering problem arising from ISFSM state minimization.

The implicit techniques described here can be applied also to other problems of logic synthesis and combinatorial optimization. For instance it is straightforward to convert the implicit computation of maximal compatibles given here into an implicit computation of prime dichotomies [23].

The remainder is organized as follows. Section 2 gives an introduction to algorithms for exact state minimization of ISFSM's, while representations and computations based on binary decision diagrams are described in Section 3. An implicit version of the exact algorithm is presented in Section 4. Section 5 describes the method to generate the implicit binate table. Alternative implicit algorithms for prime compatible generation are explored in Section 6. Implementative issues are discussed in Section 7. A different approach from [6] is surveyed in Section 8. Results on a variety of benchmarks are reported and discussed in Section 9.

2 Background on ISFSM State Minimization

In this section, we revise the basic definitions and procedures for compatible sets of states of ISFSM's, with special focus on state minimization as a key application. The theory of ISFSM state minimization has been developed in classical papers and textbooks to which we refer [18, 4, 9].

Definition 2.1 *An* **incompletely specified FSM** *(ISFSM) can be defined as a 6-tuple $M = \langle S, I, O, \Delta, \Lambda, R\rangle$. S represents the finite state space, I represents the finite input space and O represents the finite output space. Δ is the next state relation defined by a characteristic function $\Delta : I \times S \times S \to B$ where each combination of input and present state is related to a single next state or to all states. Λ is the output relation defined by a characteristic function $\Lambda : I \times S \times O \to B$ where each combination of input and present state is related to a single output or to all outputs. $R \subseteq S$ represents the set of reset states.*

In the standard literature, no reset state is specified for an ISFSM, and it is assumed that all states can potentially be picked as a reset state for implementation. The same is assumed in this chapter, and this is reflected in covering conditions defined later. In addition, an unspecified next state is traditionally not represented in the next state relation Δ. i.e., if the next state is not specified for present state s and input i, there is no state s' such that $\Delta(i, s, s') = 1$. This assumption is made in subsequent definitions and computations in this chapter.

Definition 2.2 *A set of states is an* **output compatible** *if for every input, there is a corresponding output which can be produced by each state in the set.*

Two states are an **output incompatible pair** if they are not output compatible.

Lemma 2.1 *Two states are an output incompatible pair if and only if, on some input, they cannot produce the same output.*

Definition 2.3 *A set of states is a* **compatible** *if for each input sequence, there is a corresponding output sequence which can be produced by each state in the compatible.*

The set of compatibles can theoretically be computed as follows:

1. Assume that every output compatible is a candidate (to be a compatible).
2. A candidate is not a compatible if, on some input, its states cannot produce a common output and transit to a candidate compatible set.
3. Repeat 2, until no candidate can be deleted from the set of candidate compatibles.

Practically, the number of candidates is too large to be handled by such an explicit algorithm. Two states are a **incompatible pair** if they are not compatible.

Lemma 2.2 *Two states are an incompatible pair if and only if*

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

The set of incompatible state pairs can be computed as follows:

1. Compute the output incompatible pairs, which are incompatible pairs.
2. A pair of states is an incompatible pair if, on some input, its pair of next states is a previously determined incompatible pair.
3. Repeat 2, until no new pairs can be added to the set of incompatible pairs.

Theorem 2.1 *Given an ISFSM, a set of states is a compatible if and only if every pair of states in it are compatible.*

Proof: By Definition 2.3, if a set of states is a compatible, then each pair of states contained in it is a compatible.

Suppose that each pair of states in a set C of states is a compatible. We shall prove that C is a compatible by induction on the length k of an arbitrary input sequence. On an arbitrary input i (induction basis), each state in C can produce either one output or all outputs, because the machine is an ISFSM. No two states in C produce two different *specified* outputs because they are pairwise compatible. As a result, all states in C can produce a common output on input i. Assume that for an arbitrary input sequence σ_i of length k, every states in C can again produce a common output sequence and reach a set of states C'. As states in C are pairwise compatible, so must be C'. By applying one more arbitrary input i, each state in C' can produce either one output or all outputs. No two states in C' produce different specified outputs, so all states in C' can produce a common output on input i. This shows that the states in C, on any input sequence of length $k+1$, can also produce a common output sequence. Thus C is a compatible. ∎

Corollary 2.1 *Given an ISFSM, a set of states which does not contain an incompatible pair is a compatible.*

Proof: If a set of states does not contain an incompatible pair, each pair of states in it are compatible. By Theorem 2.1, the set of states is a compatible. ∎

Definition 2.4 *A set of states d_i (or $IS(c,i)$) is an* **implied set** *of a compatible c for input i if $d_i = \{s' | \Delta(i,s,s') = 1, \ \forall s \in c\}$, i.e., it is the set of next states from state set c on input i.*

Definition 2.5 *A set C of compatibles is a* **cover** *of an ISFSM if each state in the ISFSM is contained in a compatible in C.*

Definition 2.6 *A set C of compatibles is* **closed** *in an ISFSM if for each $c \in C$, all its implied sets c_i are contained in some element of C for each inputs i.*

Theorem 2.2 *[15] The state minimization problem of an ISFSM reduces to finding a closed set C of compatibles, of minimum cardinality, which covers every state of the original machine, i.e., a* **minimum closed cover**.

Definition 2.7 *A compatible set of states is a* **maximal compatible** *if it is not a subset of another compatible.*

A set of states is a **maximal incompatible** if it is not a maximal compatible.
We show by an example an elegant procedure to find all maximal compatibles found in [13].

1. Write down the pairs of incompatibles as a product of sums.
2. Multiply them out to obtain a sum of products, and minimize it with respect to single-cube containment.
3. For each resultant product, write down missing states to get maximal compatibles.

This is equivalent to compute all prime implicants of a unate function expressed as a product of sums (of pairs of states). For instance:

1. Product of pairs of incompatibles:
   $(s_4 + s_5)(s_4 + s_6)(s_4 + s_9)(s_5 + s_7)(s_6 + s_7)(s_6 + s_8)(s_8 + s_9)$
2. Unate function in sum of products: $s_4 s_5 s_6 s_8 + s_4 s_6 s_7 s_9 + s_4 s_7 s_8 + s_5 s_6 s_9$
3. Maximal compatibles: $s_1 s_2 s_3 s_7 s_9$, $s_1 s_2 s_3 s_5 s_8$, $s_1 s_2 s_3 s_5 s_6 s_9$, $s_1 s_2 s_3 s_4 s_7 s_8$

The set of all maximal compatibles of a completely specified FSM is the unique minimum closed cover. For an incompletely specified FSM, a closed cover consisting only of maximal compatibles may contain more sets than a minimum closed cover, in which some or all of the compatibles are proper subsets of maximal compatibles.

Definition 2.8 *An implied set d of a compatible c is in its* **class set** *$CS(c)$ if*

1. *d has more than one element, and*
2. *$d \not\subseteq c$, and*
3. *$d \not\subseteq d'$ if $d' \in CS(c)$.*

Definition 2.9 *A compatible c'* **prime dominates** *a compatible c if*

1. *$c' \supset c$, and*
2. *$C(c') \subseteq C(c)$.*

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.

Definition 2.10 *A compatible set of states is a* **prime compatible** *if it is not dominated by any other compatible.*

Definition 2.11 *A prime compatible is an* **essential prime compatible** *if it contains a state not contained in any other prime compatibles.*

The following procedure (which will be used in Section 6.3.2) generates all prime compatibles from the set of maximal compatibles [4].

1. Initially the set of prime compatibles is empty.
2. Order the maximal compatibles by decreasing size, say n is the size of the largest.
3. Add to the set of prime compatibles the maximal compatibles of size n.
4. For $k = n - 1$ down to 1 do:
   (a) Compatibles of size k (and their implied sets) are generated starting from the maximal compatibles of size n to $k + 1$ (only those having non-void class set).
   (b) Add to the set of prime compatibles the compatibles of size k not dominated by any prime compatible already in the set.
   (c) Add to the set of prime compatibles all maximal compatibles of size k.

The following facts are true about the above algorithm:

- A compatible already added to the set of primes cannot be excluded by a newly generated compatible.
- The same compatible can be generated more than once by different maximal compatibles. The question arises of finding the most efficient algorithm to generate the compatibles.
- Only the compatibles generated from maximal compatibles with non-void class set need be considered, because a maximal compatible with a void class set dominates any compatible that it generates.
- A single state s can be a prime compatible if every compatible set C with more than one state and containing s implies a set with more than one state.

In [12] it is shown that, after generation of prime compatibles, an iterative procedure can expose new non-prime compatibles by updating closure constraints where eliminated non-prime compatibles are replaced by prime compatibles. Other complex rules to eliminate more prime compatibles are also given. An observation is also made that no minimal cover S can contain a compatible contained in another compatible of S.
The following theorem is proved in [4], and its generalization to PNDFSM has been given in [7].

Theorem 2.3 *For any ISFSM M, there is a reduced ISFSM M_{red} whose states all correspond to prime compatibles of M.*

A minimum closed cover can then be found by setting up a table covering problem [4] whose columns are the prime compatibles and whose rows correspond to the covering and closure conditions.
The following facts are useful in the state minimization of ISFSM's:

- The cardinality of a maximal incompatible is a lower bound on the number of states of the minimized ISFSM.
- If there is a maximal compatible that contains all states of a given ISFSM, the ISFSM reduces to a single state.

- The cardinality of the set of maximal compatibles is an upper bound on the number of states of the minimized FSM.

- If a maximal compatible has a void class set, it must be a prime compatible. As a result, no compatible contained in it can be a prime compatible (result used in Section 6.3.1).

- The minimum number of maximal compatibles covering all states is a lower bound on the number of states of the minimized ISFSM.

- The minimum number of maximal compatibles covering all states and satisfying the closure conditions is an upper bound on the number of states of the minimized ISFSM.

3 Implicit Techniques

3.1 Binary Decision Diagrams

Basics on binary decision diagrams are found in [2, 1].

Definition 3.1 *A* **binary decision diagram** *(BDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a Boolean variable $var(v)$. Vertex v has two outgoing arcs, $child_0(v)$ and $child_1(v)$. Each terminal vertex u is labeled 0 or 1.*

Each vertex in a BDD represents a binary input binary output function and all accessible vertices are roots. The terminal vertices represent the constants (functions) 0 and 1. For each nonterminal vertex v representing a function F, its child vertex $child_0(v)$ represents the function $F_{\overline{v}}$ and its other child vertex $child_1(v)$ represents the function F_v. i.e., $F = \overline{v} \cdot F_{\overline{v}} + v \cdot F_v$.

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

Definition 3.2 *A BDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_0(v))$ if $child_0(v)$ is nonterminal, and $var(v) \prec var(child_1(v))$ if $child_1(v)$ is nonterminal.*

Definition 3.3 *A BDD is* **reduced** *if*

1. *it contains no vertex v such that $child_0(v) = child_1(v)$, and*

2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 3.4 *A* **reduced ordered binary decision diagram** *(ROBDD) is a BDD which is both reduced and ordered.*

Definition 3.5 *The ITE operator returns function G_1 if function F evaluates true, else it returns function G_2:*

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

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

3.2 Zero-suppressed BDD's

Definition 3.6 *A* **zero-suppressed BDD** *(ZBDD) is defined identically as a BDD.*

The functional interpretation of ZBDD's is the same as that for BDD's.

Definition 3.7 *A ZBDD is* **ordered** *if it is ordered when viewed as a BDD.*

Definition 3.8 *A ZBDD is* **reduced** *if*

1. *it contains no vertex v such that $child_1(v)$ is a terminal vertex labeled 0, and*
2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 3.9 *A* **reduced ordered zero-suppressed binary decision diagram** *(ROZBDD) is a ZBDD which is both reduced and ordered.*

The difference between ROBDD and ROZBDD is in one reduction rule. A ROBDD eliminates all vertices whose two outgoing arcs point to the same vertex. A ROZBDD eliminates all vertices whose 1-edge points to a terminal vertex 0. Once a redundant vertex is removed from a ZBDD, the incoming edges of the vertex is directly connected to the vertex to which the corresponding terminal vertex 0 points.

3.3 Implicit Set Manipulation

In [7] it is presented a full-fledged theory on how to represent and manipulate sets using a BDD-based representation. It extends the notation used in [11] An outline is available also in [8]. This theory is especially useful for applications where sets of sets need to be constructed and manipulated.

Given a ground set G of cardinality less or equal to 2^n, any **subset** S can be represented in a Boolean space B^n by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its **characteristic function** [3], such that:

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

In other words, a subset is represented in *positional-set* or *positional-cube* notation form [1], using n Boolean variables, $x = x_1 x_2 \ldots x_n$. The presence of an element s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if element s_k is not a member of the set. One Boolean variable is needed for each element because the element can either be present or absent in the set. As an example, for $n = 6$, the set with a single element s_4 is represented by 000100 and the set $s_2 s_3 s_5$ is represented by 011010. The elements s_1, s_4, s_6 which are not present correspond to 0's in the positional-set.

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

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

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

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

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

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

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

3.3.1 Operations on Positional-sets

We propose a unified notational framework for set manipulation which extends the notation used in [11]. In this section, each operators Op acts on two sets of variables $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_n$ and returns a relation $(x \; Op \; y)$ (as a characteristic function) of pairs of positional-sets. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of objects, x and y. For example, given two sets of sets X and Y, the set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$.

Lemma 3.1 *The* **equality relation** *evaluates to true if the two sets of objects represented by positional-sets x and y are identical, and can be computed as:*

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

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.*

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

Lemma 3.2 *The* **containment relation** *evaluates to true if the set of objects represented by x contains the set of objects represented by y, and can be computed as:*

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

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

Proof: $\prod_{k=1}^{n} y_k \Rightarrow x_k$ requires for all objects that, if an object 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 objects in y. ∎

Lemma 3.3 *The* **equal-union relation** *evaluates to true if the set of objects represented by x is the union of the two sets of objects represented by y and z, and can be computed as:*

$$(x = y \cup z) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k).$$

Proof: For each position k, x_k is set to the value of the OR between x_k and y_k. Effectively, $\prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k)$ performs a bitwise OR on y and z to form a single positional-set z, which represents the union of the two individual sets. ∎

Lemma 3.4 *The* **contain-union relation** *evaluates to true if the set of objects represented by x contains the union of the two sets of objects represented by y and z, and can be computed as:*

$$(x \supseteq y \cup z) = \prod_{k=1}^{n} (y_k + z_k) \Rightarrow x_k.$$

Proof: Note the similarity in the computations of $(x \supseteq y \cup z)$ and $(x = y \cup z)$. $(x \supseteq y \cup z)$ performs bitwise OR on singletons y and z. If either of their k-bits is 1, the corresponding x_k bit is constrained to 1. Otherwise, x_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. ∎

3.3.2 Operations on Sets of Positional-sets

The first three lemmas in this section introduces operators that return a set of positional-sets as the result of some implicit set operations on one or two sets of positional-sets.

Lemma 3.5 *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 on their characteristic functions, as follows:*

$$\begin{aligned} \chi_{A \cup B} &= \chi_A + \chi_B \\ \chi_{A \cap B} &= \chi_A \cdot \chi_B \\ \chi_{A-B} &= \chi_A \cdot \neg\chi_B \\ \chi_{\overline{A}} &= \neg\chi_A \end{aligned}$$

Lemma 3.6 *The* **maximal** *of a set χ of subsets is the set containing subsets in χ not strictly contained by any other subset in χ, and can be computed as:*

$$Maximal_x(\chi) = \chi(x) \cdot \nexists y\ [(y \supset x) \cdot \chi(y)].$$

Proof: The term $\exists y\ [(y \supset x) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \subset y$. In such a case, x cannot be in the maximal set by definition, and can be subtracted out. What remains is exactly the maximal set of subsets in $\chi(x)$. ■

Lemma 3.7 *Given a set of positional-sets $\chi(x)$ and an array of Boolean variables x, the maximal of positional-sets in χ with respect to x can be computed by the recursive BDD operator $Maximal(\chi, 0, x)$:*

```
Maximal(χ, k, x) {
    if (χ = 0) return 0
    if (χ = 1) return ∏_{i=k}^{n} x_i
    M_0 = Maximal(χ_{x̄_k}, k + 1)
    M_1 = Maximal(χ_{x_k}, k + 1)
    return ITE(x_k, M_1, M_0 · ¬M_1)
}
```

Proof: The operator starts at the top of the BDD and recurses down until a terminal node is reached. At each recursive call, the operator returns the maximal set of positional-sets within χ made up of elements from k to n. If terminal **0** is reached, there is no positional-set within χ so **0** (i.e., nothing) is returned. If terminal **1** is reached, χ contains all possible positional-sets with elements from k to n, and the maximum one is $\prod_{i=k}^{n} x_i$. At any intermediate BDD node, we find the maximal positional-sets M_0 on the *else* branch of χ, the maximal positional-sets M_1 on the *then* branch of χ. The resultant maximal set of sets contains (1) positional-sets in M_1 each with element x_k added to it as they cannot be contained by any set in M_1 which has $x_k = 0$, and (2) positional-sets that are in M_0 but not in M_1 because if a set is present in both, it is already accounted for in (1). Thus the ITE operation returns the required maximal set after each call. ■
To guarantee that each node of the BDD χ is processed exactly once, intermediate results should be cached by a computed-table.

The next operators check set equality and containment between two sets of sets, whereas Lemmas 3.1 and 3.2 check it on a pair of sets only. They return tautology if the test is passed.

Lemma 3.8 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set equality test** *is true if and only if sets A and B are identical, and can be computed by:*

$$Equal_x(\chi_A, \chi_B) = \forall x \, [\chi_A(x) \Leftrightarrow \chi_B(x)].$$

Alternatively, Equal can be found by checking if their corresponding ROBDD's are the same by $bdd_equal(\chi_A, \chi_B)$.

Proof: $\chi_A(x)$ and $\chi_B(x)$ represent the same set if and only if for every x, either $x \in A$ and $x \in B$, or $x \notin A$ and $x \notin B$. Since the characteristic function representing a set in positional-set notation is unique, two characteristic functions will represent the same set if and only if their ROBDD's are the same. ■

Lemma 3.9 *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 if and only if set A contains set B, and can be computed by:*

$$Contain_x(\chi_A, \chi_B) = \forall x \, [\chi_B(x) \Rightarrow \chi_A(x)].$$

Beside operating on sets of sets, the above operators can also be used on relations of sets. The effect is best illustrated by an example. Suppose A and B are binary relations on sets. $Contain_x(\chi_A(x, y), \chi_B(x, z))$ will return another relation on pairs (y, z) of sets. Positional-sets y and z are in the resultant relation if and only if the set of positional-sets x associated with y in relation A *contains* the set of positional-sets x associated with z in relation B.

The following operator takes a set of sets and a set of variables as parameters, and returns a singleton positional-set on those variables.

Lemma 3.10 *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 sets in A, and can be computed by:*

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

Proof: For each position k, the right hand expression sets y_k to 1 if and only if there exists an x in χ_A such that its k-th bit is a 1 ($\exists x \, [\chi_A(x) \cdot x_k]$). This implies that the positional-set y will contain the k-th element if and only if 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 on all positional-sets of χ_A to form a single positional-set y which represents the union of all such positional-sets. ■

3.3.3 k-out-of-n Positional-sets

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

An efficient way of constructing and storing such collections of k-tuple sets using BDD will be given next. Figure 1 represents a reduced ordered BDD of $Tuple_{5,2}(x)$.

The root of the BDD 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 element 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 $i - 1$ elements/variables

[2] $Tuple_{n,0}(x)$ will be denoted by $\emptyset(x)$.

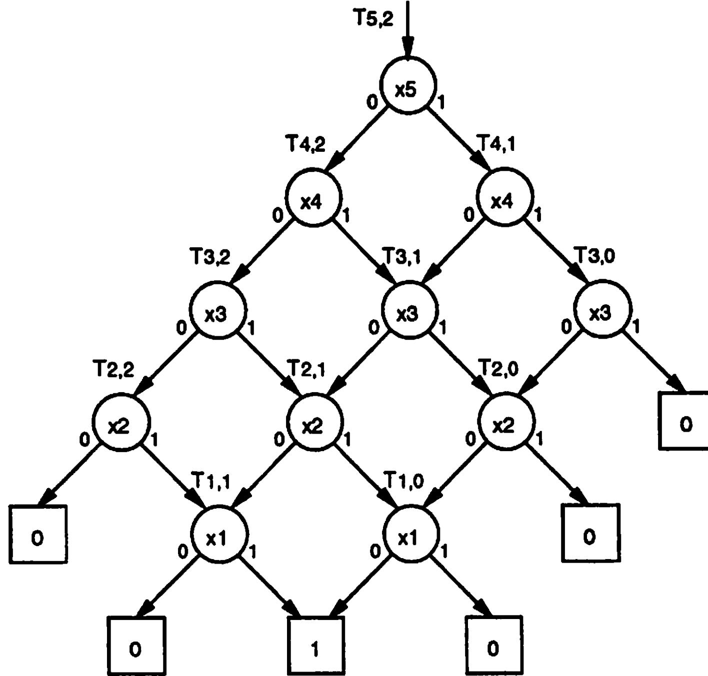

Figure 1: BDD representing $Tuple_{5,2}(x)$.

left to be processed. As we have put already element i in the positional-set, we still have to add exactly $j-1$ elements 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 element i has not been put in the positional-set and we have $j-1$ elements/variables left. Thus, the BDD for $Tuple_{i,j}$ can be constructed by the algorithm shown in Figure 2.

The total number of nonterminal vertices in the BDD of $Tuple_{n,k}$ is $(n-k+1)\cdot(k+1)-1 = nk-k^2+n = O(nk)$. 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 $k=n/2$.

3.4 FSM Implicit Representation

A state transition graph (STG) is commonly used as the internal representation of FSM's in sequential synthesis systems, such as SIS [24]. 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. This may degrade the performance of conventional optimization algorithms on large FSM's.

Assume that the given FSM has n states. To perform state minimization, one must represent and manipulate efficiently sets of states (such as compatibles) and sets of sets of states (such as sets of compatibles). Therefore *1-hot encoding* is used for the states of the FSM [3]. If inputs (outputs, respectively) of the FSM are specified symbolically, they can be represented as a multi-valued symbolic variable i (o, respectively) where each value of i (o, respectively) represents an input (output, respectively) combination. For compactness of representation, we used for these variables a *logarithmic encoding*, i.e. an m-valued variable is represented

[3] An alternative explained in [7] is to represent any set of sets of states (i.e., set of state sets) implicitly as a single 1-hot encoded MDD, and manipulate the state sets symbolically all at once. Different sets of sets of states can be stored as multiple roots with a single shared 1-hot encoded MDD.

```
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
}
```

Figure 2: Pseudo-code for the *Tuple* operator.

with $\log_2 m$ Boolean variables. The fact that different multi-valued variables use different encodings is not a problem as long as they are used consistently. However if inputs (outputs, respectively) of the FSM are already given in encoded form, each encoded bit of inputs (outputs, respectively) is represented by a single Boolean variable.

4 Implicit Generation of Compatibles

An exact algorithm for state minimization consists of two steps: the generation of various sets of compatibles, and the 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 ISFSM's may overlap. As a result, the number of compatibles can be exponential in the number of states [22], and the generation of the whole set of compatibles can be a challenging task.

The covering step 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 of ISFSM's. In each of the following subsections, we shall first restate the definition of some combinatorial object, give a logic formula to compute it, and then argue the correctness of the formula.

4.1 Output Incompatible Pairs

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

Definition 4.1 *Two states are an* **output incompatible pair** *if, for some input, they cannot generate the same output.*

Lemma 4.1 *The set of output incompatible pairs, $\mathcal{OICP}(y, z)$, can be computed as:*

$$\mathcal{OICP}(y, z) = Tuple_1(y) \cdot Tuple_1(z) \cdot \exists i \; \nexists o \, [\Lambda(i, y, o) \cdot \Lambda(i, z, o)] \quad (1)$$

Proof: 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 if and only if for some input i, there is no output pattern that both state y and z can produce (i.e., output incompatible). ■

In the above and subsequent formulas, we will mix notations between relations and their corresponding characteristic functions. Strictly speaking if we would have used the characteristic function notation, the above formula would have been more clumsy:

$$\mathcal{OICP}(y,z) = 1 \quad \text{if and only if} \quad (Tuple_1(y) = 1) \cdot (Tuple_1(z) = 1) \cdot \exists i \; \nexists o \, [(\Lambda(i,y,o) = 1) \cdot (\Lambda(i,z,o) = 1)]$$

4.2 Incompatible Pairs

Definition 4.2 *Two states are an* **incompatible pair** *if*

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

Lemma 4.2 *The set of incompatible pairs is the least fixed point of $\mathcal{ICP}$:*

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

and can be computed by the following iteration:

$$\begin{aligned} \mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) \\ \mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z) + \exists i,u,v \, [\Delta(i,y,u) \cdot \Delta(i,z,v) \cdot \mathcal{ICP}_k(u,v)] \end{aligned} \quad (2)$$

The iteration can terminate when $\mathcal{ICP}_{k+1} = \mathcal{ICP}_k$ and the set of incompatible pairs is $\mathcal{ICP}(y,z) = \mathcal{ICP}_k(y,z)$.

Proof: The fixed point computation starts with the set of output incompatible pairs. After the $(k+1)$-th

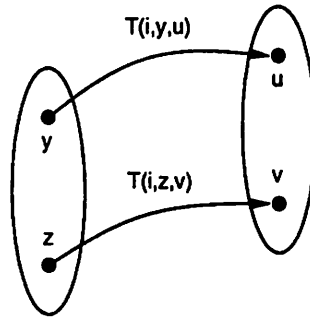

Figure 3: Finding incompatible pairs.

iteration of Equation 2, $\mathcal{ICP}_{k+1}(y,z)$ contains all the incompatible state pairs (y,z) that lead to an output incompatible pair in $k+1$ 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). ■

4.3 Incompatibles

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

Definition 4.3 *A set of states is an* **incompatible** *if it contains at least one incompatible pair.*

Lemma 4.3 *The set of incompatibles can be computed as:*

$$\mathcal{IC}(c) = \exists y, z\, [\mathcal{ICP}(y, z) \cdot (c \supseteq y \cup z)] \tag{3}$$

Proof: By Lemma 3.4, $(c \supseteq y \cup z) = \prod_{k=1}^{n} y_k + z_k \Rightarrow c_k$ 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 $(y, z) \in \mathcal{ICP}$. ∎

4.4 Compatibles

Definition 4.4 *A set of states is a* **compatible** *if it is not an incompatible.*

Lemma 4.4 *The set of compatibles, $\mathcal{C}(c)$, can be computed as:*

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

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

4.5 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 4.5 *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.*

Lemma 4.5 *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 if and only if on input i, n is a next state from state p in compatible c:*

$$\mathcal{F}(c, i, n) = \exists p\, [\mathcal{C}(c) \cdot (c \supseteq p) \cdot \Delta(i, p, n)] \tag{4}$$

Proof: $\mathcal{F}(c, i, n)$ associates a compatible $c \in \mathcal{C}$ and an input i with a 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) if and only 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 $\mathcal{F}(c, i, n)$. 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.

Lemma 4.6 *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:*

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

Proof: Considering the rightmost term, $\mathcal{F}(c,i,n)$ relates implied next states as singleton positional-sets n to compatible c and input i and $Union_{n \to d}(\mathcal{F}(c,i,n))$ 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 for different inputs. ■

4.6 Class Set of a Compatible

Definition 4.6 *An implied set d of a compatible c is in its* **class set** *if*

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.

Lemma 4.7 *The class set of a compatible c is defined by the relation $CCS(c,d)$ which evaluates to 1 if and only if the implied set d is in the class set of compatible c:*

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

Proof: The singleton implied sets $Tuple_1(d)$ are excluded according to condition 1 in Definition 4.6. 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 $\mathcal{CI}(c,d)$. ■

4.7 Prime Compatibles

To solve exactly the covering problem, it is sufficient to consider a subset of compatibles called prime compatibles. As proved in [4], at least one minimum closed cover consists entirely of prime compatibles.

Definition 4.7 . *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.

Lemma 4.8 *The prime dominance relation is given by:*

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

Proof: The two terms on the right express the two dominance conditions by which c' dominates c according to Definition 4.7. Since compatibles c and c' are represented as positional-sets, $(c' \supset c)$ is computed according to a variant of Lemma 3.2. 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 Lemma 3.9. ■

Definition 4.8 *A* **prime compatible** *is a compatible not dominated by another compatible.*

Lemma 4.9 *The set of prime compatibles is given by:*

$$\mathcal{PC}(c) = \mathcal{C}(c) \cdot \nexists c'\ [\mathcal{C}(c') \cdot Dominate(c', c)]$$

Proof: 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 4.8, the set of prime compatibles is simply given by the set of compatibles $\mathcal{C}(c)$ excluding those that are dominated. ■

4.8 Essential and Non-essential Prime Compatibles

Definition 4.9 *A prime compatible is an* **essential prime compatible** *if it contains a state not contained in any other prime compatibles.*

Because any solution must correspond to a closed *cover*, each state must be contained in a selected compatible, and thus every essential prime compatible must be selected.

Lemma 4.10 *The set of essential prime compatibles can be computed as:*

$$\mathcal{EPC}(c) = \mathcal{PC}(c) \cdot \sum_{k=1}^{n} \{c_k \cdot \nexists c'\ [c'_k \cdot \mathcal{PC}(c') \cdot (c \neq c')]\}$$

Proof: For a set c of states to be an essential prime compatible $\mathcal{EPC}(c)$, it must be a prime compatible $\mathcal{PC}(c)$. In addition, there must be a state s_k such that $s_k \in c$ and there is no $c' \in \mathcal{PC}$ different from c such that $s_k \in c'$. The positive literal c_k denotes the fact $s_k \in c$, and similarly for c'_k. ■

Definition 4.10 *A prime compatible is a* **non-essential prime compatible** *if it is not an essential prime compatible.*

Lemma 4.11 *The set of non-essential prime compatibles can be computed as:*

$$\mathcal{NEPC}(c) = \mathcal{PC}(c) \cdot \neg\mathcal{EPC}(c)$$

Proof: By Definition 4.10. ■

5 Implicit Generation of the Binate Covering Table

Once the set of (non-essential) prime compatibles is generated, the problem of exact state minimization can be solved as a binate table covering problem. In this section, we shall describe how such a binate table can be generated. To keep with our stated objective, the binate table is also represented implicitly. We describe an implicit representation of the covering table, that adroitly exploits how row and columns were implicitly computed. A description of how the binate table is then solved implicitly can be found in [8].

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, etc).

Definition 5.1 *A column is labeled by a positional-set p. The set of* **column labels** *C is obtained by prime generation as $C(p) = \mathcal{PC}(p)$.*

Besides distinguishing one 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, $Tuple_0(c)$. Because of Definition 5.3 for row labels, the columns intersecting a row labeled $r = (c, d)$ in a 1 are labeled by the prime compatibles p that contain d.

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

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

the table entry is a 1 iff relation $1(r, p) \stackrel{\text{def}}{=} (p \supseteq d)$ 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 labeled by the prime compatible $p = c$, and the row has 1's in the columns labeled 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 labeled by prime compatibles p_i which contain the singleton state d [4]. Since no minimal cover S can contain a

[4] Every closed cover of an ISFSM whose states are all reachable must cover all the states. We can say that the covering clauses express a property of the solution that speeds up the search. Alternatively one could impose that only the reset state is covered and let the search procedure find that a solution covers all states.

compatible contained in another compatible of S [12], one could introduce a new collection of clauses, one for each pair of compatibles p_1 and p_2 such that $p_1 \supset p_2$, each stating that at most one of the two can be chosen $(\overline{p_1} + \overline{p_2})$.

6 Improvements to the Implicit Algorithm

The experiments reported in Section 9 identify two bottlenecks in the implicit computations described in Section 4:

1. the fixed point computation of incompatible pairs;
2. the handling of closure information, i.e., implied sets and class sets.

Sections 6.1 and 6.2 describe alternative methods to perform those computations. Section 6.3 shows how maximal compatibles can be used with advantage in the computation of prime compatibles.

6.1 Incompatible Pairs Generation using Generalized Cofactor

This subsection describes a variation on the fixed point computation of incompatible pairs $\mathcal{ICP}(y,z)$, presented in Section 4.2. Each iteration of the computation of Equation 2 can be viewed as an inverse image projection from a set of state pairs in $\mathcal{ICP}_k(u,v)$ to a set of states pairs in $\mathcal{ICP}_{k+1}(y,z)$ via the product transition relation $\Delta(i,y,u)\cdot\Delta(i,z,v)$. In the original method, all state pairs in $\mathcal{ICP}_{k+1}(u,v)$ are projected during the $(k+2)$-th iteration. Some are not necessary because if the projected pair (y',z') of $\mathcal{ICP}_{k+1}$ is actually in $\mathcal{ICP}_k$ as shown in Figure 4 [5], its projection (y'',z'') must have already been calculated in a previous iteration. Thus at the $(k+2)$-th iteration, we need to project only the *new* incompatible state pairs discovered at the $(k+1)$-th iteration. This is done in the following modification of the fixed point computation of Section 4.2:

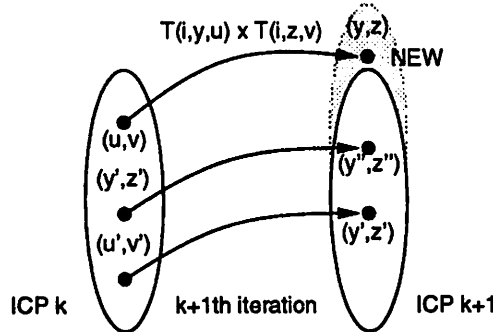

Figure 4: An alternative way of finding incompatible pairs.

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

[5] In Figure 4, the direction of the arrows shows the *inverse* projections.

Instead of finding a minimum cardinality set of state pairs for projection, a set of state pairs with a small BDD representation is more desirable for our implicit BDD formulation. A small BDD for $NEW(y,z)$ can be obtained by means of the generalized cofactor operator [25] using $\mathcal{ICP}_k(y,z)$ as the don't care set:

$$NEW(y,z) = NEW(y,z)|_{\neg \mathcal{ICP}_k(y,z)}$$

As a result of this modification, the geometric mean [6] of the ratio of CPU run time for computing $\mathcal{ICP}$ with this generalized cofactor method versus the original method is 0.68.

6.2 Handling of Closure Information

For FSM's with many compatibles, the most time consuming part of the implicit algorithm is the computation of implied sets and class sets of the compatibles. The reason is that these implicit computations deal with two sets of variables (c, d) in each relation, c representing a compatible and d representing its implied set or class set. Since each compatible may have a different class set, the size of the corresponding BDD's may blow up during the computation.

A way to cope with this problem is to represent the class sets by means of singletons n (by Lemma 4.5) instead of sets d of arbitrary cardinalities (by Lemma 4.6). This avoids the computation of the BDD's of $\mathcal{CI}(c,d)$ and $CCS(c,d)$ which are usually big. The computations in Sections 4.5, 4.6 and 4.7 can be replaced by the following ones:

Theorem 6.1 *The class sets of compatibles can be obtained by pruning the implied set relation $\mathcal{F}(c,i,n)$ in singleton form, by the following computations:*

$$\begin{aligned}
\mathcal{F}(c,i,n) &= \exists p \, [\Delta(i,p,n) \cdot C(c) \cdot (c \supseteq p)] \\
I(c,i) &= \exists n \, n' \, [\mathcal{F}(c,i,n) \cdot \mathcal{F}(c,i,n') \cdot (n \neq n')] && (5) \\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n) \cdot I(c,i) \\
J(c,i) &= \exists n \, [\mathcal{F}(c,i,n) \cdot (c \not\supseteq n)] \\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n) \cdot J(c,i) \\
K(c,i) &= \exists \, i' \, \{ \, [Contain_n(\mathcal{F}(c,i,n), \mathcal{F}(c,i',n)) + \neg J(c,i')] \\
&\quad \cdot \neg [Contain_n(\mathcal{F}(c,i',n), \mathcal{F}(c,i,n)) + \neg J(c,i')] \, \} && (6) \\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n) \cdot \neg K(c,i)
\end{aligned}$$

Proof: For each input i and each compatible c, we must compute a unique implied set denoted by c_i. Each implied set c_i is represented as a set of singleton-states n, i.e., $c_i = \{n | \mathcal{F}(c,i,n)\}$. The relations I, J and K guarantee the satisfaction of the first, second and third conditions, respectively, of Definition 4.6 of an implied set. The relation $I(c,i)$ finds all implied sets c_i which contain at least two distinct next states n and n', i.e., it computes all implied sets with more than one element. Equation 5 prunes the set $\mathcal{F}$ accordingly. Relation $J(c,i)$ contains all remaining implied sets not contained in c, thus satisfying the second condition.

To compute the third condition implicitly, its form needs to be transformed. From the set of implied sets, we delete an set class c_i if $c_i \subset c_{i'}$. We know that, $c_i \subset c_{i'}$ if and only if $c_i \subseteq c_{i'}$ and $c_{i'} \not\subseteq c_i$. The operation $Contain_n(\mathcal{F}(c,i,n), \mathcal{F}(c,i',n))$ is used to test if $c_i \subset c_{i'}$, but since its result may include invalid (c,i') pairs (i.e., implied sets) the term $\neg J(c,i')$ is needed in the equation. In the last equation, the implied sets in $K(c,i)$ are subtracted away from $\mathcal{F}$ because they violate the third condition. ■

[6] The geometric mean of k numbers is the the k-th root of the product of the k numbers.

Theorem 6.2 *The condition that compatible c' dominates compatible c can be computed as:*

$$Dominate(c', c) = (c' \supset c) \cdot \forall i' \, \exists i \, Contain_n(\mathcal{F}(c, i, n), \mathcal{F}(c', i', n))$$

Proof: By Definition 4.7, c' dominates c if c' covers all states covered by c and the conditions on the closure of c' are a subset of the conditions on the closure of c. ■

After computing the dominance relation $Dominate(c', c)$, the prime compatibles can be generated using Lemma 4.9.

6.3 Prime Compatible Generation using Maximals

The prime compatible generation algorithm given in Section 4 does not rely on the computation of maximal compatibles, whereas the method in [4] does. We are going to present an alternative implicit generation algorithm that does make use of maximal compatibles. The motivation is to expedite the prime generation step, that is usually the most time consuming step.

Theorem 6.3 *The set of maximal compatibles can be computed as:*

$$\mathcal{MC}(c) = Maximal_c(\mathcal{C}(c))$$

Proof: By Definition 2.7, the set of maximal compatibles $\mathcal{MC}(c)$ is simply the *maximal* set of positional-sets in $\mathcal{C}(c)$ with respect to c (Lemma 3.6). ■

Corollary 6.1 *The set of singleton next states implied by a maximal compatible c under input i, $\mathcal{F}(c, i, y)$, can be computed by:*

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

The class set information for the maximal compatibles can then be obtained using the class set computation procedure as described in Theorem 6.1.

6.3.1 Compatible Pruning by Maximal Compatibles with Void Class Set

When generating prime compatibles from compatibles, only the compatibles generated from maximal compatibles with a non-void class set need to be considered, because a maximal compatible with a void class set dominates any compatible that it generates.

Corollary 6.2 *The maximal compatibles with a void class set, $\mathcal{MCV}(c)$, can be obtained by:*

$$\mathcal{MCV}(c) = \mathcal{MC}(c) \cdot \nexists i \, K(c, i)$$

where $K(c, i)$ is given by Equation 6. The set of compatibles can then be pruned by $\mathcal{MCV}(c)$:

$$\mathcal{C}(c) = \mathcal{C}(c) \cdot \nexists c' \, [\mathcal{MCV}(c) \cdot (c' \supseteq c)]$$

6.3.2 Slicing Procedure for Prime Compatible Generation

The following slicing procedure is an implicit version of the procedure outlined in Section 2.

$$
\begin{array}{l}
\mathcal{PC}(c) = \emptyset \\
\text{for } k = n \text{ down to 1 do}\{ \\
\quad \mathcal{MC}_k(c) = \mathcal{MC}(c) \cdot Tuple_k(c) \\
\quad \mathcal{C}_k(c) = \mathcal{C}(c) \cdot Tuple_k(c) \cdot \neg\mathcal{MC}_k(c) \\
\quad \mathcal{F}_{\mathcal{C}_k}(c,i,n) = Prune(\mathcal{C}_k(c), \Delta(i,p,n)) \\
\quad \mathcal{F}_{\mathcal{PC}}(c,i,n) = Prune(\mathcal{PC}(c), \Delta(i,p,n)) \\
\quad Dominate(c',c) = (c' \supset c) \cdot \forall i' \, \exists i \, Contain_n(\mathcal{F}_{\mathcal{PC}}(c,i,n), \mathcal{F}_{\mathcal{C}_k}(c',i',n)) \\
\quad \mathcal{PC}_k(c) = \mathcal{C}(c) \cdot \not\exists c' \, [Dominate(c',c) \cdot \mathcal{C}(c')] \\
\quad \mathcal{PC}(c) = \mathcal{PC}(c) + \mathcal{PC}_k(c) + \mathcal{MC}_k(c) \\
\}
\end{array}
$$

$\mathcal{PC}(c)$ is a set of prime compatibles accumulated during each iteration, and is originally empty. $\mathcal{MC}_k(c)$ contains maximal compatibles with cardinality k. $\mathcal{C}_k(c)$ contains compatibles c of cardinality k, excluding those in $\mathcal{MC}_k(c)$. $Prune(\mathcal{C}_k(c), \Delta(i,p,n))$ is the procedure to prune class sets described in Theorem 6.1, by substituting $\mathcal{C}_k(c)$ for $\mathcal{C}(c)$, and $\mathcal{F}_{\mathcal{C}}(c,i,n)$ for $\mathcal{F}(c,i,n)$ in the equations. $Prune(\mathcal{PC}(c), \Delta(i,p,n))$ is similarly defined. So $\mathcal{F}_{\mathcal{C}_k}(c,i,n)$ and $\mathcal{F}_{\mathcal{PC}}(c,i,n)$ contain the class sets of $\mathcal{C}_k(c)$ and $\mathcal{PC}(c)$ respectively. To test for $Dominate(c',c)$, we only need to know if a compatible $c' \in \mathcal{C}_k$ is dominated by an already discovered prime compatible $c \in \mathcal{PC}$, because (1) for any other $c' \in \mathcal{C}_k$, $c \not\subset c'$, and (2) c can be dominated only by prime compatibles with cardinalities greater than k. $\mathcal{PC}_k(c)$ contains the newly discovered prime compatibles with cardinality k, and this set is added to $\mathcal{MC}_k$ and $\mathcal{PC}$ to update the set of prime compatibles found so far.

Experimentally during construction of the BDD's of prime compatibles, this slicing method uses on average half the memory as compared to the method in Section 6.2. This method is particularly efficient if the sizes of the compatibles are localized to a few cardinalities.

7 Implementation Details

7.1 BDD Variable Assignment

Simple accounting shows that:

1. 10 state variable vectors $(p, n, y, z, u, v, c, c', d, d')$ are used in all previous equations,
2. in positional-set notation, each state variable vector corresponds to n Boolean variables where n is the number of states.

Looking into each equation carefully reveals that we never operate on more than four sets of variables simultaneously in a single BDD operation. For example, 4 sets of variables y, z, u and v are used in Equation 2, and 3 sets p, n and c in Equation 4. The idea of *BDD variable assignment* is to use a set of BDD variables for more than one purpose, by binding at different times more than one set of variables from the equations to a single set of BDD variables. The assignments should be made in such a way that no two sets of variables appearing in an equation will be assigned to the same set of BDD variables. Such an assignment for our previous implicit algorithm is shown in Figure 5.

There is a conflict with the above BDD variable assignment in Equation 3, because variable c is assigned to the same BDD variables as variable y. To get around it, an extra variable e is used instead:

$$\mathcal{IC}(c) = [e \rightarrow c] \exists y, z \, [\mathcal{ICP}(y,z) \cdot (y \supseteq z \cup e)].$$

BDD variable sets			
0	1	2	3
		p	n
y	u	z	v
c	d	c'	d'
			e

Figure 5: Assignments of equation variables to BDD variables.

Note that two functions containing different variables being assigned to the same BDD variable, e.g., $\Delta(i,p,n)$ and $\mathcal{CCS}(c',d')$, can co-exist within a multi-rooted BDD *at the same time*, without interfering with one another. Conflict will occur only when they become operands to a BDD operation. Actually as a result of overlapping in variable supports, such unrelated functions can be constructed and manipulated more efficiently due to sharing of BDD subgraphs.

7.2 BDD Variable Ordering

Relations as equality, containment, maximal described in Section 3.3.1 and variants of them have BDD's of exponential size if the different sets of BDD variables are not interleaved with each other.

Also the ordering between individual state variables within a set of BDD variables (i.e., a positional-set) is important, especially when handling the closure information. An heuristics that we used is to put at the top of the BDD the Boolean variables corresponding to states that occur most frequently in the compatibles. This should leave the BDD sparse in the lower part where most state variables take a value of 0. As the set of compatibles is usually very large, we approximate the count by counting the occurrences of states in maximal compatibles.

7.3 Don't Cares in the Positional-set Space

The main advantage of a positional-set representation of FSM's is that sets of sets of states can be represented by a single multi-rooted BDD. As a result, sets of compatibles ($\mathcal{C}$), prime compatibles ($\mathcal{PC}$), and the likes can be represented and manipulated compactly. However during the computation of $\mathcal{OCP}$, $\mathcal{OICP}$ and $\mathcal{ICP}$, we are manipulating only sets of singleton states and so we only *care* about a small portion of the encoding space. Since no positional-set of cardinality greater than 1 will appear there, we can make use of these don't care code points in the positional-set space.

For example, the computations involved in Equations 1 to 2 manipulate a product of two singleton states (y,z). The don't care condition with respect to this pair of singletons is captured by:

$$DC(y,z) = \neg Tuple_0(y) \cdot \neg Tuple_1(y) + \neg Tuple_0(z) \cdot \neg Tuple_1(z)$$

and can be used to simplify the BDD computation of these sets using the generalized cofactor operator [25].

8 Approximations of Prime Compatibles

In this section, we review a related research reported in [6] where a subset (pw-primes) and a superset (ipw-primes) of prime compatibles are defined. The contribution rests on the key notion of signature set of a prime compatible, that plays a similar role to the notion of signature cubes in two-level minimization [14]. To decide if compatible c dominates compatible c' (for $c' \subset c$), it is sufficient to know that c' contains a compatible in the signature set of c. This allows to compute primes with a quantifier-free recursive BDD

operator. Of course the difficulty of computating primes has simply been shifted to the computation of signatures sets, but a superset of primes has been defined whose signature sets can be computed implicitly in an efficient manner. In the sequel we will introduce two new definitions of dominance: ipw-dominance and pw-dominance, while we will reserve the word "dominance" without qualifications for the classical definition 2.9 .

Definition 8.1 *A* **signature** *s of compatible c is a minimal subset of c such that $CS(c) \subseteq CS(s)$. A signature of a compatible is a minimal compatible dominated by c. The set of all signatures of c is $Sign(c)$.*

Theorem 8.1 *Compatible c dominates compatible c' if and only if*

1. *$c' \subset c$, and*
2. *$\exists d \in Sign(c)$, $c' \supseteq d$.*

Proof: **Only If Part.** If c dominates c' then $CS(c) \subseteq CS(c')$, so either $c' \in Sign(c)$ or $\exists d \in Sign(c)$ such that $d \subset c'$, by definition of signature, and this is the thesis.

If Part. Suppose that

1. $c' \subset c$, and
2. $\exists d \in Sign(c), c' \supseteq d$.

Then, from the definition of signature, $CS(c) \subseteq CS(d)$. We must show that $CS(c) \subseteq CS(c')$ to have that c dominates c'. Suppose by contradiction that $CS(c) \not\subseteq CS(c')$, then there exists an implied set s such that $s \in CS(c)$ and $s \notin CS(c')$. If $s \in CS(c)$ - and so $s \in CS(d)$ - and $s \notin CS(c')$ there must exist an input i such that $s = IS(d, i) \subset IS(c', i) \subseteq IS(c, i) = s' \in CS(c)$, but then $s \subset s'$ and this goes against part 3 of the definition of class set that requires that no implied set is contained in another implied set of a class set.
■

8.1 An Overapproximation of Prime Compatibles

Definition 8.2 *An* **input-labeled pairwise class set** *$IPWCS(c)$ of compatible c is the set of all the input-labeled implied pairs $(s_a, s_b)_i$ under input i of any state pairs in c such that:*

1. *$s_a \neq s_b$, and*
2. *$\{s_a, s_b\} \not\subseteq c$.*

Definition 8.3 *Compatible c* **ipw-dominates** *compatible c' if and only if*

1. *$c' \subset c$, and*
2. *$IPWCS(c') \supseteq IPWCS(c)$.*

Definition 8.4 *An* **ipw-signature** *s of compatible c is a minimal subset of c such that $IPWCS(c) \subseteq IPWCS(s)$. A signature of a compatible is a minimal compatible ipw-dominated by c. The set of all ipw-signatures of c is $Sign_{ipw}(c)$.*

Definition 8.5 *An* **ipw-prime compatible** *is one that is not ipw-dominated by any other compatible.*

Theorem 8.2 *If c is a prime compatible, then it is also an ipw-prime compatible.*

Proof: Suppose by contradiction that c is not an ipw-prime, i.e., that there is another compatible c' that ipw-dominates c. Then $c \subset c'$ and $IPWCS(c) \supseteq IPWCS(c')$. The second inclusion implies that for every input i $IS(c,i) \supseteq IS(c',i)$ and so it follows that $CS(c) \supseteq CS(c')$, therefore c is dominated by c' and it is not a prime. ■

The theorem shows that ipw-primes are a superset of primes. It turns out that the inclusion is strict, i.e., there are ipw-primes that are not primes [7].

Theorem 8.3 *Compatible c ipw-dominates compatible c' if and only if*

1. $c' \subset c$, *and*
2. $\exists d \in Sign_{ipw}(c), c' \supseteq d$.

8.2 An Underapproximation of Prime Compatibles

Definition 8.6 *A* **pairwise class set** $PWCS(c)$ *of compatible c is the set of all the implied pairs* (s_a, s_b) *of any state pairs in c such that:*

1. $s_a \neq s_b$, *and*
2. $\{s_a, s_b\} \not\subseteq c$.

Definition 8.7 *Compatible c* **pw-dominates** *compatible c' if and only of*

1. $c' \subset c$, *and*
2. $PWCS(c') \supseteq PWCS(c)$.

Definition 8.8 *A* **pw-signature** *s of compatible c is a minimal subset of c such that* $PWCS(c) \subseteq PWCS(s)$. *A signature of a compatible is a minimal compatible pw-dominated by c. The set of all pw-signatures of c is* $Sign_{pw}(c)$.

Definition 8.9 *A* **pw-prime compatible** *is one that is not pw-dominated by any other compatible.*

Theorem 8.4 *If c is a pw-prime compatible, then it is also a prime compatible.*

The theorem shows that pw-primes are a subset of primes. It turns out that the inclusion is strict, i.e., there are primes that are not pw-primes. Therefore using pw-primes does not guarantee an exact solution, but only an approximation [8].

Theorem 8.5 *Compatible c pw-dominates compatible c' if and only if*

1. $c' \subset c$, *and*
2. $\exists d \in Sign_{pw}(c), c' \supseteq d$.

In [6] it is shown how it is possible to generate the ipw-primes by an implicit computation. It is of particular interest that pw-primes are obtained by a recursive quantifier-free computation that operates on pairs (c, s) where c is a compatible and s is a subset in $Sign_{ipw}(c)$. Experimental results show faster run times, but a smaller set of completed benchmarks, than reported in Section 9.

[7] In [5] it is stated that if no implied set of compatible c contains any other implied set then the set of primes is equal to the set of ipw-primes.

[8] In [5] it is proved that if any two implied sets in the class set of compatible c are disjoint then the set of primes is equal to the set of pw-primes.

9 Experimental Results

We implemented the algorithms described in previous sections in a program called ISM, an acronym for Implicit State Minimizer. We ran ISM on different suites of FSM's. We report results on the following suites of FSM's. They are:

1. the MCNC benchmark and other examples,
2. FSM's generated by a synthesis procedure for asynchronous logic [10],
3. FSM's from learning I/O sequences [17],
4. FSM's from synthesis of interacting FSM's [26],
5. FSM's with exponentially many prime compatibles,
6. FSM's with many maximal compatibles, and
7. randomly generated FSM's.

Each suite has different features with respect to state minimization. We discuss features of the experiments and results in different subsections. Comparisons are made with STAMINA [20], 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 prime compatibles.

For each example, we report the number of states in the original ISFSM, the number of maximal compatibles if applicable, the number compatibles, the number of prime compatibles, the number of non-essential prime compatibles if applicable, and the run time for our implicit algorithm ISM and that for the explicit algorithm STAMINA. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory. The CPU run time refers to the computation of the prime compatibles only.

9.1 FSM's from MCNC Benchmark and Others

Table 1 reports the results 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). 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. But when the number of primes is not negligible as in *ex2* and *green*, ISM ran as fast or faster than STAMINA.

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.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
arbseq	94	2	96	9	3	12	0
bbsse	16	11	97	13	0	0	0
beecount	7	4	11	7	5	0	0
ex1	20	2	22	19	1	1	0
ex2	19	36	2925	1366	1366	7	13
ex3	10	10	195	91	91	0	0
ex5	9	6	81	38	38	0	0
ex7	10	6	135	57	57	0	0
fsm1	256	47	302	208	0	83	0.6
green	54	524	1234	524	524	90	125
lion9	9	5	20	5	2	0	0
mark1	15	12	41	18	11	0	0
scf	121	12	1201	175	87	22	0
squares	371	45	473	307	0	731	1
tbk	32	16	48	48	48	3	1
tma	20	15	35	20	4	1	0
train11	11	5	85	17	15	0	0
viterbi	68	5	329	57	3	6	0

Table 1: The MCNC benchmark and others.

9.2 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 [10]. 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 a thousand. To explain this data reported in Table 2, we notice that in order to compute the prime compatibles, every compatible has to be generated by STAMINA 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 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 formulation of the covering problem [10].

9.3 FSM's from Learning I/O Sequences

Table 3 and Figure 6 show the results of running a parametrized set of FSM's constructed to be compatible with a given collection of examples of input/output traces [17]. These machines exhibit very large number of compatibles.

Here ISM shows all its power compared to STAMINA, both in terms of number of computed prime compatibles and running time. STAMINA runs out of memory on the examples from *threer.35* and *fourr.30* 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	24	16
future	36	49	7.929e8	49	49	3	0
future.m	28	16	2.621e7	16	16	2	0
intel_edge.dummy	28	120	9432	396	396	37	3
isend	40	128	22207	480	480	13	spaceout
isend.m	20	15	22207	19	19	1	0
mp-forward-pkt	20	1	1.048e6	1	0	0	0
nak-pa	56	8	4.741e15	8	8	9	0
nak-pa.m	18	8	44799	8	8	1	0
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
ram-read-sbuf	36	2	3.006e10	2	0	2	0
sbuf-ram-write	58	24	1.433e6	24	24	14	0
sbuf-ram-write.m	24	12	1.433e6	12	12	2	0
sbuf-send-ctl	20	10	81407	10	10	0	0
sbuf-send-pkt2	21	2	622591	2	0	0	0
vbe4a	58	2072	1.756e12	2072	2072	109	167
vbe4a.m	22	13	73471	13	13	2	0
vbe6a.m	16	8	527	8	4	1	0
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	21	1344
threer.40	41	1.456e6	529420	75	spaceout
threer.55	55	3.622e7	1.555e7	1273	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	*18181	spaceout
fourr.70	71	9.621e10	4.524e10	*22940	spaceout

Table 3: FSM's from learning I/O sequences.

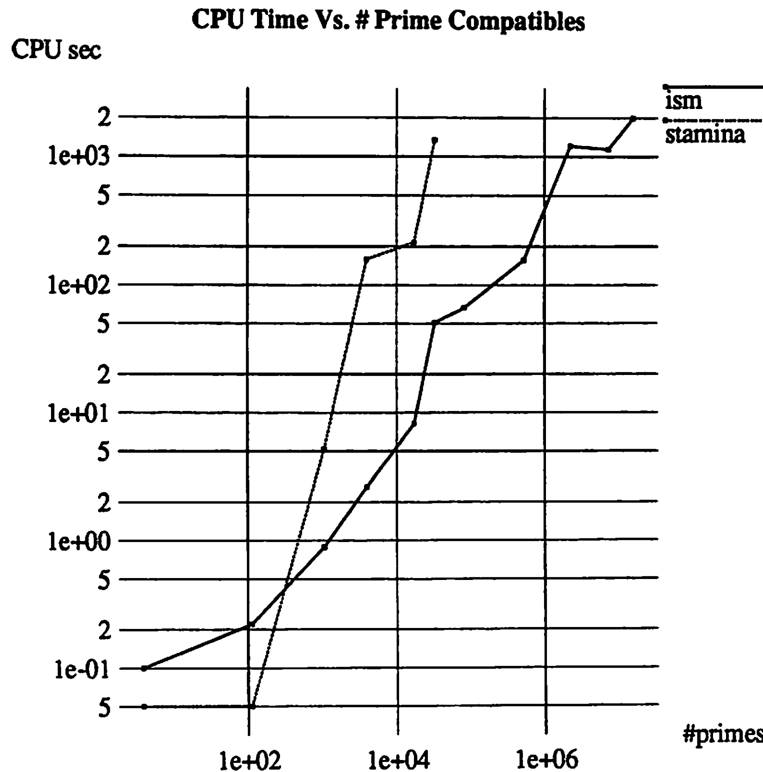

Figure 6: Comparison between ISM and STAMINA on learning I/O sequences benchmark.

9.4 FSM's from Synthesis of Interacting FSM's

It has been reported by Rho and Somenzi in [21] 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.

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 [26]. They exhibit often large number of compatibles and prime compatibles, as shown in Table 4. For such cases, the run times of the implicit algorithm ISM are shorter than those by STAMINA.

machine	# state	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
ifsm0	38	1064973	18686	43	4253
ifsm1	74	43006	8925	25	466
ifsm2	150	497399	774	267	356

Table 4: Examples from synthesis of interacting FSM's.

9.5 FSM's with Exponentially Many Prime Compatibles

In the previous examples, the number of prime compatibles is not large compared to the number of states. A natural question to ask is whether there are FSM's that generate a large number of prime compatibles with respect to the number of states. We were able to construct a suite of FSM's where the number of prime compatibles is exponential in the number of states.

Rubin gave in [22] 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 [9]. 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 $F(n)$ of ISFSM's [10] (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 component subgraphs F_i are defined on the same input and outputs. Each subgraph 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 5 and Figure 7 show 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^{1500} with reasonable running times, STAMINA, based on an explicit enumeration runs out of memory soon (and where it completes, it takes much longer).

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) 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
rubin2250	2250	3^{750}	$2^{1500}-1$	$2^{1500}-1$	$2^{1500}-1$	271134	spaceout

Table 5: Constructed FSM's.

[9] 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.

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

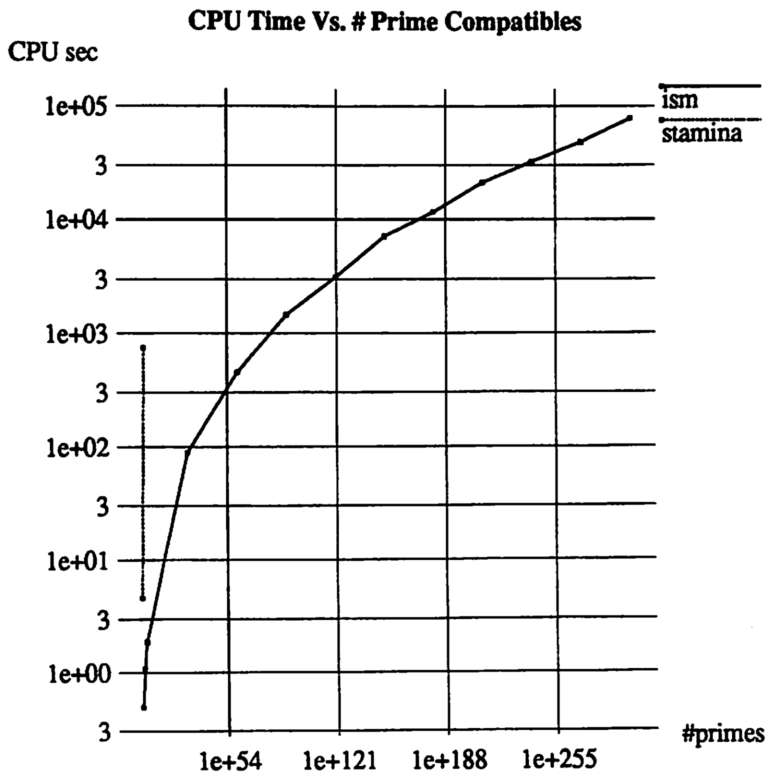

Figure 7: Comparison between ISM and STAMINA on constructed FSM's.

9.6 FSM's with Many Maximal Compatibles

Table 6 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 University of Colorado, Boulder. The example *lavagno* is from asynchronous synthesis as those reported in Section 9.2. For these examples the program STAMINA was run with the option -M to compute all maximal compatibles. 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. The prime compatibles for these examples could not be computed by either program.

9.7 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 [11]. A small percentage of the randomly generated FSM's were found to exhibit this behavior. Table 4 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.

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

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 6: FSM's with many maximals.

machine	# states	# max compat.	# compat.	# prime compat.	# $\mathcal{NEPC}$	CPU time (sec) ISM	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	21	spaceout
ex2.285	19	2	393215	121501	121500	13	spaceout
ex2.304	19	2	393215	264079	264079	93	spaceout
ex2.423	19	4	204799	160494	160494	102	spaceout
ex2.680	19	2	327679	192803	192803	151	spaceout

Table 7: Random FSM's.

9.8 Experiments Comparing BDD and ZBDD Sizes

As mentioned before, the objects we manipulate in our implicit algorithms are sets and relations of positional-sets. In our application positional-sets (e.g., compatibles) are usually sparse sets. Minato showed in [16] that Zero-suppressed BDD's are a good representation of sets of sparse sets. As a preliminary investigation of the effectiveness of a ZBDD-based algorithm for exact state minimization of ISFSM's, we converted some key BDD objects into ZBDD's and observed the change in the number of used nodes.

	Compatibles # nodes in		Prime Compat. # nodes in		table→R			
					# nodes in		CPU Time (sec)	
machine	BDD	ZBDD	BDD	ZBDD	BDD	ZBDD	bdd2zbdd	zbdd2bdd
alex1	2562	1203	1243	415	167	43	0.00	0.00
ex2.271	21	51	236	286	5646	4307	0.44	0.27
ex2.285	22	38	461	471	39	2	0.00	0.00
ex2.304	22	38	787	826	56928	34263	5.66	3.19
ex2.423	29	66	675	782	40857	27060	4.66	3.32
ex2.680	23	41	1819	1953	57446	40277	8.06	4.25
ex2	161	108	222	148	4421	1521	0.18	0.16
ex3	28	33	45	41	584	290	0.02	0.01
ex5	34	31	33	26	269	111	0.01	0.00
ex7	26	30	41	39	322	159	0.01	0.01
green	197	78	194	62	211	54	0.01	0.01
keyb_ex2	581	173	1260	324	14539	2053	0.77	0.58
s386_keyb	320	188	404	175	6386	1270	0.25	0.22
room4.16	75	79	201	159	1213	501	0.05	0.04
room4.20	122	134	458	407	2863	1431	0.16	0.11
room3.20	113	110	265	228	1717	790	0.07	0.05
room3.25	220	185	544	402	3922	1567	0.18	0.12
room3.30	637	473	1205	785	9391	3389	0.59	0.38

Table 8: Comparison between BDD and ZBDD sizes.

Experiments have been performed on FSM's which require non-trivial covering steps for state minimization. During state minimization of each machine, ZBDD's are generated from BDD's representing the set of compatibles C, the set of prime compatibles $\mathcal{PC}$ (which is usually also the set of column labels), and the set of row labels R. From the rightmost two columns of the above table, the conversion routines between ZBDD's and BDD's seem to execute fast enough to make feasible switching between BDD and ZBDD representations.

Out of the 18 examples, 9 of them have smaller ZBDD's than BDD's for representing C, 13 of them have smaller ZBDD's than BDD's for representing $\mathcal{PC}$ (the remaining 5 examples are all randomly generated machines), and 18 of them have smaller ZBDD representations of R. Disregarding the second set of examples, ex2.$* * *$, which are randomly generated machines, the comparison shows that ZBDD's are usually smaller than BDD's and are always comparably close for the exceptional cases.

When converting BDD's to ZBDD's, the most reduction in sizes occurs for the representation of R, and less for $\mathcal{PC}$, and the least for C. A possible explanation is that $\mathcal{PC}$ is a more sparse set than C, because all state sets that are in $\mathcal{PC}$ are in C but not vice versa. A similar explanation also applies between R and

$\mathcal{PC}$. Each row label in R consist of two parts: a positional-set representing a prime compatible, and another positional-set representing an implied set. As we don't expect much sharing between different implied sets, the set of row labels (whose support has twice as many variables as the the one of $\mathcal{PC}$ and $\mathcal{C}$) should be a more sparse set.

10 Results of State Minimization of ISFSM's

Once a binate table has been generated, ISM calls an implicit binate table solver to find a contained behavior with a minimum number of states. In this section we summarize the final results. We refer to [8] for a detailed exposition.
The following explanations refer to the tables of results:

- Under table size we provide the dimensions of the binate table before and after the table reduction step (that replaces the original table with an equivalent one).
- # mincov is the number of recursive calls of the binate cover routine.
- α and β mean, respectively, α and β column dominance, two ways of reducing the columns of a covering table.
- 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).
- CPU time refers only to the binate covering algorithm. It does not include the time to find the prime compatibles.

10.1 Minimizing Small and Medium Examples

With the exception of *ex2*, *ex3*, *ex5*, *ex7*, the examples from the MCNC and asynchronous benchmarks do not require prime compatibles for exact state minimization and yield simple covering problems. Table 9 reports those 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.

FSM	table size (rows x columns)		# mincov				# cover				CPU time (sec)			
	before	after first	ISM		STAMINA		ISM		STAMINA		ISM		STAMINA	
	reduction	α reduction	α	β	α	β	α	β	α	β	α	β	α	β
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	*4	*10	*12	*10	*9	*58	*293	*116	*91
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	286	*10	*12	*10	5	*58	*293	*116	2100
ex3	243 x 91	151 x 84	201	37	91	39	4	4	4	4	78	33	0	0
ex5	81 x 38	47 x 31	16	6	10	6	3	3	3	3	4	3	0	0
ex7	137 x 57	62 x 44	38	31	37	6	3	3	3	3	8	12	0	0

Table 9: Examples from the MCNC benchmark.

These experiments suggest that

- the number of recursive calls of the binate cover routine (*# mincov*) of ISM and STAMINA is roughly comparable,
- the running times are better for STAMINA except in the largest example, *ex2*, where ISM is slightly faster than STAMINA. 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, some implicit reduction computations have not yet been fully optimized.

10.2 Minimizing Constructed Examples

Table 10 presents a few randomly generated FSM's. They generate 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.

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

Table 10: Random FSM's.

10.3 Minimizing FSM's from Learning I/O Sequences

Examples in Table 10 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 11 answer in the affirmative the question. They are the from the suite of FSM's described in [17]. 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.

10.4 Minimizing FSM's from Synthesis of Interacting FSM's

Prime compatibles are required only for the state minimization of $ifsm1$ and $ifsm2$. For $ifsm1$, ISM can find a first solution faster than STAMINA using α-dominance. But as the table sizes are not very big, the run times ISM take are usually longer than those for STAMINA.

11 Conclusions

We have presented an algorithm to generate implicitly sets of compatibles of ISFSM's. An application is the exact solution of state minimization. Compatibles, maximal compatibles, prime compatibles and implied sets are all represented by the characteristic functions of relations implemented with BDD's. The only explicit dependence is on the number of states of the initial problem. We have demonstrated with

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

Table 11: Learning I/O sequences benchmark.

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

Table 12: Examples from synthesis of interactive FSM's.

experiments from a variety of benchmarks that implicit techniques allow to handle examples exhibiting a number of compatibles up to 2^{1500}, an achievement outside the scope of programs based on explicit enumeration [19]. We have shown, when discussing the experiments, that ISFMS's with a very large number of compatibles may be produced as intermediate steps of logic synthesis algorithms, for instance in the cases of asynchronous synthesis [10], and of learning I/O sequences [17]. A similar situation is expected to occur also in the synthesis of interacting FSM's [21]. This shows that the proposed approach has not only a theoretical interest, but also practical relevance for current logic synthesis applications. The final step of an implicit exact state minimization procedure, i.e., solving implicitly a binate covering problem [19], has been described in [8].

The techniques described here can be easily applied to similar problems in sequential synthesis. For instance the implicit computation of maximal compatibles given here can be converted in a straighforward manner into an implicit computation of prime dichotomies [23]. Therefore this algorithmic frame has wide applicability in logic synthesis and combinatorial optimzation.

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

[4] 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.

[5] H. Higuchi and Y. Matsunaga. Implicit generation of prime compatibles for incompletely specified finite state machines. *Manuscript*, May 1995.

[6] H. Higuchi and Y. Matsunaga. Implicit prime compatible generation for minimizing incompletely specified finite state machines. In *Proceedings of ASP-DAC*, 1995.

[7] T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, U.C. Berkeley, Electronics Research Laboratory, University of California at Berkeley, May 1995.

[8] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *The Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

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

[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] F. Luccio. Extending the definition of prime compatibility classes of states in incomplete sequential machine reduction. *IEEE Transactions on Computers*, C-18(6):537–540, June 1969.

[13] M. Marcus. Derivation of maximal compatibles using Boolean algebra. *IBM Journal of Research and Development*, November 1964.

[14] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincenetelli. Espresso-signature: a new exact minimizer for logic functions. *IEEE Transactions on VLSI Systems*, pages 432–440, December 1993.

[15] R. E. Miller. *Switching theory. Volume I: sequential circuits and machines*. J. Wiley and & Co., N.Y., 1965.

[16] S. Minato. Zero-suppressed BDD's for set manipulation in combinatorial problems. In *The Proceedings of the Design Automation Conference*, pages 272–277, June 1993.

[17] Arlindo L. Oliveira and Stephen A. Edwards. Inference of state machines from examples of behavior. Technical report, UCB/ERL Technical Report M95/12, Berkeley, CA, 1995.

[18] M. Paull and S. Unger. Minimizing the number of states in incompletely specified state machines. *IRE Transactions on Electronic Computers*, September 1959.

[19] 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.

[20] J.-K. Rho and F. Somenzi. Stamina. *Computer Program*, 1991.

[21] 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.

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

[23] 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.

[24] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *The Proceedings of the International Conference on Computer Design*, pages 328–333, October 1992.

[25] 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.

[26] 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.

[27] Y. Watanabe. Logic optimization of interacting components in synchronous digital systems. *Ph.D. Thesis, Tech. Report No. UCB/ERL M94/32*, April 1994.