IMPLICIT GENERATION OF COMPATIBLES FOR EXACT STATE MINIMIZATION

by

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

Memorandum No. UCB/ERL M93/60

3 August 1993

IMPLICIT GENERATION OF COMPATIBLES FOR EXACT STATE MINIMIZATION

by

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

Memorandum No. UCB/ERL M93/60

3 August 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

IMPLICIT GENERATION OF COMPATIBLES FOR EXACT STATE MINIMIZATION

by

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

Memorandum No. UCB/ERL M93/60

3 August 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Implicit Generation of Compatibles for Exact State Minimization

Timothy Kam Tiziano Villa Robert Brayton Alberto Sangiovanni-Vincentelli

Abstract

Implicit computations of the solution set of optimization problems arising in logic synthesis hold the promise of enlarging the size of input instances that can be solved exactly. The state minimization problem for incompletely specified machines is an important step for sequential circuit optimization. The problem is NP-hard and hence most techniques are heuristic. An exact algorithm consists of two steps: generation of sets of compatibles, and solving a binate covering problem. This paper presents implicit computations to obtain the sets of compatibles required for an exact state minimization of incompletely specified finite state machines (ISFSM's). Sets of maximal compatibles, compatibles, prime compatibles and implied class sets are all represented and manipulated implicitly by means of BDD's that realize the characteristic functions of the 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^{1200}, an achievement outside the scope of programs based on explicit enumeration [9]. 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 [13]. This shows that the proposed approach has not only a theoretical interest, but also practical relevance for current logic synthesis applications. A recasting of the final binate covering step as an implicit computation is under progress.

1 Introduction

Seminal work by researchers at Bull [5] and improvements at UC Berkeley [23] produced powerful techniques for implicit enumeration of subsets of states of a Finite State Machine (FSM). These techniques are based on the idea to operate on large sets of states by their characteristic functions represented by Binary Decision Diagrams (BDD's). In many cases of practical interest these sets have a regular structure that translates into small-sized BDD's. Once the related BDD's can be constructed, the most common Boolean operations on them (including satisfiability) have low complexity, and this makes feasible to carry on computations unaffordable in the traditional case where all states must be explicitly represented. Of course it may be the case that the BDD cannot be constructed, because of the intrinsic structure of the function to represent or because a good ordering of the variables is not found.

More recent work at Bull [6, 15] has shown how implicants, primes and essential primes of a two-valued or multi-valued function can also be computed implicitly. Reported experiments show a suite of examples where all primes could be computed, whereas explicit techniques implemented in ESPRESSO [2] failed to do so.

Therefore it is important to investigate how far these techniques based on implicit computations can be pushed to solve the core problems of logic synthesis and verification. When exact solutions are sought, explicit techniques run easily out of steam because too many elements of the solution space must be enumerated. It appears that implicit techniques offer the most realistic hope to increase the size of problems that can be solved exactly. This paper on exact state minimization of FSM's is a first step on the application of implicit techniques to solve optimization problems in the area of sequential synthesis.

State minimization of FSM's is a well-known problem [11]. State minimization of completely specified FSM's (CSFSM's) has a complexity subquadratic in the number of states [10]. This makes it an easy problem when the starting point is a two-level description of an FSM, because the number of states is usually less than a few hundred. The problem becomes difficult to manage when the starting point is an encoded sequential circuit with a large number of latches (in the hundreds). In that case the traditional method would be required to extract a state transition graph from the encoded network and then apply state minimization to it. But when latches are more than a dozen, the number of reachable states may be so huge to make state extraction and/or state minimization unfeasible. Recently it has been shown [16, 14] how to bypass the extraction step and compute equivalence classes of states implicitly. Equivalence classes are basically all that is needed to minimize a completely specified state machine. A compatible projection operator uniquely encodes each equivalence class by selecting a unique representative of the class to which a given state belongs. This implicit technique allows state minimization of sequential networks outside the domain of traditional techniques.

State minimization of incompletely specified FSM's (ISFSM's) instead has been shown to be an NP-hard problem [18]. Therefore even for problems represented with two-level descriptions involving a hundred states, an exact algorithm may consume too much memory and time. Moreover, it has been recently reported ([12]) that even examples with very few states generated during the synthesis of asynchronous circuits may fail to complete (or require days of CPU time) when run with a state-of-art exact state minimizer as STAMINA [9]. Therefore it is of practical importance to revisit exact state minimization of ISFSM's and address the issue of representing implicitly the solution space.

We underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this paper can be applied to other problems of logic synthesis and combinatorial optimization. For instance the implicit computation of maximal compatibles given here can be easily converted into an implicit computation of prime encoding-dichotomies (see [22]). Therefore the computational methods described here contribute to build a body of implicit techniques whose scope goes much beyond a specific application.

In this paper we address the problem of computing sets of compatibles for the exact state minimization of ISFSM's. We show how to compute sets of maximal compatibles, compatibles and prime compatibles with implicit techniques and demonstrate that in this way it is possible to handle examples exhibiting a number of compatibles up to 2^{1200}, an achievement outside the scope of programs based on explicit enumeration [9]. We indicate also where such examples arise in practice. The final step of an implicit exact state minimization procedure, i.e. solving a binate table covering problem [21], will be presented in a separate paper.

The remainder of the paper is organized as follows. In Section 2 an introduction to classical exact algorithms for state minimization of ISFSM's is given. Section 3 introduces representations of FSM's based on Binary Decision Diagrams (BDD's) [4, 1], that are the starting point of the implicit algorithms presented in Section 5. Section 4 presents a theory of representation and manipulation of sets and sets of sets. Alternative implicit algorithms are explored in Section 7. A discussion of more subtle aspects of the implementation of the presented algorithms is given in Section 8. Results on a variety of benchmarks are reported and discussed in Section 9. Conclusions and future work are summarized in Section 10.

2 Classical Algorithm

Most of the terminology used in this report is common parlance of the logic synthesis community [11, 2, 3].

2.1 Finite State Machines

A Finite-State Machine is represented by its **State Transition Graph** (STG) or equivalently, by its **State Transition Table** (STT). A STG is denoted by a sextuple $\{I, O, S, IS, \delta, \lambda\}$, where I and O are the sets of inputs and outputs, S is the set of states and IS is the set of initial states. δ (next state function) is a mapping from $I \times S$ to S that given an input and a present state defines a next state. λ (output function) is a mapping from $I \times S$ to O that given an input and a present state defines an output. An STG where the next-state and output for every possible transition from every state are defined corresponds to a **completely specified machine**. An **incompletely specified machine** is one where at least one of the functions δ and λ are partially defined, i.e. there is at least one pair (i, s) on which either the next state function or the output function (or both) are not defined.

An STT is a tabular representation of the FSM. Each row of the table corresponds to a single edge in the STG. Conventionally, the leftmost columns in the table correspond to the primary inputs and the rightmost columns to the primary outputs. The column following the primary inputs is the present-state column and the column following that is the next-state column.

2.2 Compatibles, Prime Compatibles and Minimum Closed Covers

In this subsection we will revise briefly the basic definitions and procedures for exact state minimization of ISFSM's, as presented in the original papers and standard textbooks [17, 8, 11].

Definition 2.1 *An input sequence is* **admissible** *for a starting state of a machine if no unspecified next state is encountered, except possibly at the final step.*

Definition 2.2 *States s_i and s_j are* **compatible** *iff they never generate different specified outputs for any admissible input sequence.*

Definition 2.3 *States s_i and s_j are* **output incompatible** *iff $\exists i_k$ such that $\lambda(i_k, s_i) \neq \lambda(i_k, s_j)$*

Definition 2.4 *States s_i and s_j are* **incompatible** *iff they are not compatible. States s_i and s_j are incompatible iff s_i and s_j are output incompatible, or $\exists i_k$ such that states $\delta(i_k, s_i)$ and $\delta(i_k, s_j)$ are incompatible.*

The set of all pairs of incompatible states can be computed as follows:

1. Compute output incompatible pairs.
2. Add any pair of states (s_i, s_j) if $\exists i_k$ such that $(\delta(i_k, s_i), \delta(i_k, s_j))$ is a previously determined incompatible pair of states.
3. Repeat 2. until no new pairs can be added to the incompatible state pairs set.

Definition 2.5 *A* set *of states is* **compatible** *(i.e. the set is a* **compatible***) iff every pair in it is compatible.*

Definition 2.6 *If C_i is a set of compatible states and $C_{ij} = \{s_k | s_k = \delta(I_j, s_i) \; \forall s_i \in C_i\}$, i.e. C_{ij} is the set of next states of the states in C_i for input I_j, then C_{ij} is said to be* **implied** *by the set C_i for input I_j.*

Definition 2.7 *Let C_i be a compatible set of states and C_{ij} be the set of next states implied by C_i for input I_j. The sets C_{ij} implied by C_i for all inputs I_j are the* **implied classes** *of C_i.*

Definition 2.8 *A set of compatible sets $C = \{C_1, C_2, ...\}$ is* **closed** *iff for every $C_i \in C$ all the implied sets C_{ij} are contained in some element of C for all inputs I_j.*

Definition 2.9 *The problem of minimizing the number of states reduces to finding a closed set C of compatible states, of minimum cardinality, which covers every state of the original machine, i.e. a* **minimum closed cover**.

Definition 2.10 *Sets of compatible states which are not subsets of any other compatible set of states are called* **maximal compatibles**.

Similarly one defines maximal incompatibles.

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 of maximal compatibles only may contain more sets than a closed cover in which some or all of the compatible sets are proper subsets of maximal compatibles.

Definition 2.11 *Let C_i be a compatible set of states and C_{ij} be the set of next states implied by C_i for input I_j. The* **class set** *P_i implied by C_i is the sets of all sets C_{ij} implied by C_i for all inputs I_j such that*

1. *C_{ij} has more than one element*
2. *$C_{ij} \not\subseteq C_i$*
3. *$C_{ij} \not\subseteq C_{ik}$ if $C_{ik} \in P_i$*

Definition 2.12 *A compatible C_i* **dominates** *a compatible C_j if*

1. *$C_i \supset C_j$*
2. *$P_i \subseteq P_j$*

i.e. C_i dominates C_j if C_i covers all states covered by C_j and the conditions on the closure of C_i are a subset of the conditions on the closure of C_j.

Definition 2.13 *A compatible set of states that is not dominated by any other compatible set is called a* **prime compatible** *set.*

The following procedure (used in section 7.3.2) computes all prime compatibles [8]. At the beginning the set of prime compatibles is empty.

1. Order the maximal compatibles by decreasing size, say n is the size of the largest.
2. Add to the set of prime compatibles the maximal compatibles of size n.
3. For $i = 1$ to $n - 1$:
   (a) Generate all compatibles of size $n - i$ and compute their implied classes. The compatibles of size $n - i$ are generated starting from the maximal compatibles of size n to $n - i + 1$ (only those that do not have a void class set).
   (b) Add to the set of primes the compatibles of size $n - i$ not dominated by any prime already in the set.
   (c) Add to the set of primes all maximal compatibles of size $n - i$.

The following facts are true:

- A compatible already added to the set of primes cannot be excluded by a newly generated compatible.
- In the previous algorithm, 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-empty class set need be considered, because a maximal compatible with an empty class set dominates any compatible that it generates.
- A single state s_i can be a prime compatible if every compatible set C_i with more than one state and containing s_i implies a set with more than one state.

Definition 2.14 *An* **essential prime compatible** *is a prime compatible which contains a state not contained in any other prime compatibles.*

The following theorem is proved in [8].

Theorem 2.1 *For any FSM M there is a minimum equivalent FSM M_{red} whose states all correspond to prime compatible sets of M.*

A minimum closed cover can be then found by setting up a table covering problem [8].
The following facts are useful in the minimization of FSM's:

- The cardinality of a maximal incompatible is a lower bound on the number of states of the minimized FSM.
- If there is a maximal compatible that contains all states of a given FSM, the FSM 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 7.3.1).
- The minimum number of maximal compatibles covering all states is a lower bound on the number of states of the minimized FSM.
- 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 FSM.

3 FSM Representation using BDD's

A good representation for a problem is key to the development of efficient algorithms, and this is true also for problems in sequential synthesis and verification. A state transition graph (STG) is commonly used as the internal representation of FSM's in sequential synthesis systems, such as SIS. Many algorithms for sequential synthesis have been developed to apply to STG's. However, large FSM's cannot be stored and manipulated without memory usage and CPU time becoming prohibitively large. 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 graph algorithms.

A binary decision diagram (BDD) [4, 1] provides an alternative way of representing FSM's. A BDD is a rooted, directed acyclic graph (DAG) where each node is associated with a Boolean variable. There are 2 outgoing arcs from each node. The left outgoing arc corresponds to the case when the variable takes the value 0 and the right arc corresponds to the case when the variable takes 1. The leaves of the graph are the terminal nodes 0 and 1. A path from the root to a terminal 1 represents a satisfying assignment of variables on which the BDD evaluates to 1. Thus a BDD can represent any Boolean function on any n Boolean variables $f : B^n \rightarrow B$ where $B = \{0, 1\}$. A ROBDD is a BDD that is both ordered and reduced. Ordered means that on each path from the root to a terminal the variables are encountered in the same order. Reduced means that in the DAG there are no two isomorphic subgraphs.

The literal x_i denotes that variable x_i has the value 1 and the literal $\overline{x_i}$ denotes that variable x_i has the value 0.

Any **subset** S in a Boolean space B^n can be represented by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its characteristic function, such that:

$$\chi_S(x) = 1 \text{ iff } x \text{ in } S \qquad (1)$$

In the sequel, we'll not distinguish the subset S from its characteristic function χ_S, and will use S to denote both.

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

$$\mathcal{R}(x, y) = 1 \text{ iff } x \text{ is in relation } \mathcal{R} \text{ to } y \tag{2}$$

$\mathcal{R}$ can be a one-to-many relation over the two sets in B. The **image** of x is the set $\{y \in B | (x, y) \in \mathcal{R}\}$, while the **inverse image** of y is the set $\{x \in B | (x, y) \in \mathcal{R}\}$. The image and inverse image of a set of states $\mathcal{S}(x)$ can be implicitly computed as:

$$\text{image of } \mathcal{S} \text{ under } \mathcal{R} = \exists x \; \mathcal{S}(x) \cdot \mathcal{R}(x, y) \tag{3}$$

$$\text{inverse image of } \mathcal{S} \text{ under } \mathcal{R} = \exists y \; \mathcal{S}(y) \cdot \mathcal{R}(x, y) \tag{4}$$

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:

$$\mathcal{R}(x_1, x_2, \ldots, x_n) = 1 \text{ iff the } n\text{-tuple } (x_1, x_2, \ldots, x_n) \text{ is in relation } \mathcal{R} \tag{5}$$

3.1 Positional-set Representation

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

Given that there are 2^n possible distinct sets of states, in order to represent collections of them it is not possible to encode the states using $log_2 n$ Boolean variables. Instead, each subset of states is represented in **positional-set** or positional-cube notation form, using a set of n Boolean variables, $x = x_1 x_2 \ldots x_n$. The presence of a state s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if state s_k is not a member of the set. One Boolean variable is needed for each state because the state can either be present or absent in the set[1].

In the above example, $n = 6$, and the set with a single state s_4 is represented by 000100 while the set of states $s_2 s_3 s_5$ is represented by 011010. The states s_1, s_4, s_6 which are not present correspond to 0's in the positional-set.

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

$$\chi_S(x) = 1 \text{ iff the set of states represented by the positional-set } x \text{ is in the set } S. \tag{6}$$

A BDD representing $\chi_S(x)$ will contain minterms, each corresponding to a state set in S. The operators for manipulating positional-sets and characteristic functions will be described in section 4.

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 resp.) represents an input (output resp.) combination. However if inputs (outputs resp.) of the FSM are given in encoded form, each encoded bit of inputs (outputs resp.) is represented as a single binary variable. For the latter case, BDD's will be sufficient for our purpose of implicit state minimization.

In the case of an ISFSM, some next states as well as the outputs may not be specified. So relations instead of functions must be used to represent the transition and output information. The transition relation $\mathcal{T}(i, p, n)$ and the output relation $\mathcal{O}(i, p, o)$ capture all the information contained in an STT.

Definition 3.1 *The* **transition relation** *is represented as:*

$$\mathcal{T}(i, p, n) = 1 \textit{ iff } n \textit{ is the specified next state of state } p \textit{ on input } i \;\; (i.e. \; n = \delta(p, i)) \tag{7}$$

[1]The representation of primes proposed by Coudert *et al.* [6] needs 3 values per variable to distinguish if the present literal is in positive or negative phase or in both phases.

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

Definition 3.2 *The* **output relation** *is represented as:*

$$\mathcal{O}(i, p, o) = 1 \text{ iff } o \text{ is a (possibly unspecified) output of state } p \text{ on input } i \;\; (i.e.\; o = \lambda(p, i)) \tag{8}$$

We represent all unspecified outputs in the relation $\mathcal{O}$, to ensure correctness of the output compatibles computation described in Section 5. An unspecified output in the STT corresponds to a set of minterms carrying all possible output combinations.

When states and transitions are represented implicitly, the BDD representation is often much smaller than STG. There is no direct correlation between the complexity of the STG and the size of the corresponding BDD. Using these BDD relations and the positional-set notation, we propose a new implicit algorithms for generating various sets of compatibles for solving the state minimization problem.

4 Implicit Manipulation of Sets and Sets of Sets

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

4.1 BDD Operators

A rich set of BDD operators has been developed and published in the literature [4, 1]. The following is the subset of operators useful in the present work.

Definition 4.1 *The* **substitution** *in the function $\mathcal{F}$ of variable x_i with variable y_i is denoted by:*

$$[x_i \rightarrow y_i]\mathcal{F} = \mathcal{F}(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_n) \tag{9}$$

and the substitution in the function $\mathcal{F}$ of a set of variables $x = x_1x_2 \ldots x_n$ qith another set of variables $y = y_1y_2 \cdots y_n$ is obtained simply by:

$$[x \rightarrow y]\mathcal{F} = [x_1 \rightarrow y_1][x_2 \rightarrow y_2] \ldots [x_n \rightarrow y_n]\mathcal{F} \tag{10}$$

In the description of subsequent computations, some obvious substitutions will be omitted for clarity in formulae.

Definition 4.2 *The* **cofactor** *of $\mathcal{F}$ with respect to the literal x_i ($\overline{x_i}$ resp.) is denoted by $\mathcal{F}_{x_i}$ ($\mathcal{F}_{\overline{x_i}}$ resp.) and is the function resulting when x_i is replaced by 1 (0 resp.):*

$$\mathcal{F}_{x_i}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \tag{11}$$
$$\mathcal{F}_{\overline{x_i}}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \tag{12}$$

The cofactor of $\mathcal{F}$ is a simpler function than $\mathcal{F}$ itself because the cofactor no longer depends on the variable x_i.

Definition 4.3 *The* **existential quantification** *(also called* **smoothing***) of a function $\mathcal{F}$ over a variable x_i is denoted by $\exists x_i(\mathcal{F})$ and is defined as:*

$$\exists x_i(\mathcal{F}) = \mathcal{F}_{\overline{x_i}} + \mathcal{F}_{x_i} \tag{13}$$

and the existential quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ is defined as:

$$\exists x(\mathcal{F}) = \exists x_1(\exists x_2(\ldots(\exists x_n(\mathcal{F})))) \tag{14}$$

Definition 4.4 *The* **universal quantification** *(also called* **consensus***) of a function $\mathcal{F}$ over a variable x_i is denoted by $\forall x_i(\mathcal{F})$ and is defined as:*

$$\forall x_i(\mathcal{F}) = \mathcal{F}_{\overline{x_i}} \cdot \mathcal{F}_{x_i} \tag{15}$$

and the universal quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ is defined as:

$$\forall x(\mathcal{F}) = \forall x_1(\forall x_2(\ldots(\forall x_n(\mathcal{F})))) \tag{16}$$

4.2 Operations on a Pair of Positional-sets

With our previous definitions of relations and positional-set notation for representing set of states, useful relational operators on sets can be derived. We propose a unified notational framework for set manipulation which extends the notation used in [15]. In this section, operators act on two sets of states represented as positional-sets $x = x_1x_2\ldots x_n$ and $y = y_1y_2\ldots y_n$, and return 1 iff (x, y) are in the particular relation. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of states, x and y. For example, given two sets of state sets X and Y, the state set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot Contain(x, y)$.

Theorem 4.1 *The* **equality relation** *tests if the two sets of states represented by positional-sets x and y are identical, and can be computed as:*

$$Equal(x, y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k \tag{17}$$

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 state k, either both positional-sets x and y contain it, or it is absent from both. Therefore, x and y contains exactly the same set of states and thus are equal. □

Theorem 4.2 *The* **containment relation** *tests if the set of states represented by x contains the set of states represented by y, and can be computed as:*

$$Contain(x, y) = \prod_{k=1}^{n} y_k \Rightarrow x_k \tag{18}$$

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 that for all states, if a state k is present in y (i.e. $y_k = 1$), it must also be present in x ($x_k = 1$). Therefore set x contains all the states in y. □

Theorem 4.3 *The* **strict containment relation** *tests if the set of states represented by x strictly contains the set of states represented by y, and can be computed as:*

$$Strict\_Contain(x, y) = Contain(x, y) \cdot \neg Equal(x, y) \tag{19}$$

Alternatively, $Strict\_Contain(x, y)$ can be computed by:

$$Strict\_Contain(x, y) = \prod_{k=1}^{n}[y_k \Rightarrow x_k] \cdot \sum_{k=1}^{n}[x_k \cdot \neg y_k] \tag{20}$$

Proof: Equation 19 follows directly from the two previous theorems. For equation 20, the first term is simply the containment constraint, while the second term $\sum_{k=1}^{n}[x_k \cdot \neg y_k]$ requires that for at least one state k, it is present in x ($x_k = 1$) but is absent from y ($y_k = 0$), i.e. x and y are not the same. So it is an alternative way of computing $Strict\_Contain(x, y)$. □

Theorem 4.4 *The* **union relation** *tests if the set of states represented by x is the union of the two sets of states represented by y and z, and can be computed as:*

$$Union(y, z, x) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k) \tag{21}$$

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. □

Theorem 4.5 *The* **intersection relation** *tests if the set of states represented by x is the intersection of the two sets of states represented by y and z, and can be computed as:*

$$Intersect(y, z, x) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k \cdot z_k) \tag{22}$$

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

4.3 Operations on Sets of Positional-sets

Theorem 4.6 *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:*

$$\chi_{A \cup B} = \chi_A + \chi_B \tag{23}$$
$$\chi_{A \cap B} = \chi_A \cdot \chi_B \tag{24}$$
$$\chi_{A - B} = \chi_A \cdot \neg\chi_B \tag{25}$$
$$\chi_{\overline{A}} = \neg\chi_A \tag{26}$$

Theorem 4.7 *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 iff sets A and B are identical, and can be computed by:*

$$Set\_Equal_x(\chi_A, \chi_B) = \forall x\ \chi_A(x) \Leftrightarrow \chi_B(x) \tag{27}$$

Alternatively, Set_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 $\chi_B(x)$ represents the same set iff for every x, either $x \in A$ and $x \in B$, or $x \notin A$ and $x \notin B$. As the characteristic function representing a set in positional-set notation is unique, two characteristic functions will represent the same set iff their ROBDD's are the same. □

Theorem 4.8 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set containment test** *is true iff set A contains set B, and can be computed by:*

$$Set\_Contain_x(\chi_A, \chi_B) = \forall x\ \chi_B(x) \Rightarrow \chi_A(x) \tag{28}$$

Theorem 4.9 *Given the characteristic functions χ_A and χ_B representing two sets A and B (of positional-sets), the* **set strict containment test** *is true iff set A strictly contains set B, and can be computed by:*

$$Set\_Strict\_Contain_x(\chi_A, \chi_B) = Set\_Contain_x(\chi_A, \chi_B) \cdot \neg Set\_Equal_x(\chi_A, \chi_B) \qquad (29)$$

Proof: The proof follows directly from previous two theorems. □

Theorem 4.10 *The* **maximal** *of a set F of sets is the set containing sets in F not strictly contained by any other set in F, and can be computed as:*

$$Maximal_x(F) = F(x) \cdot \neg\exists y\ [Strict\_Contain(y, x) \cdot F(y)] \qquad (30)$$

Proof: The term $\exists y\ [Strict\_Contain(y, x) \cdot F(y)]$ is true iff there is a positional-set y in F 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 states set in $F(x)$. □

Theorem 4.11 *The* **minimal** *of a set F of sets is the set containing sets in F not strictly containing any other set in F, and can be computed as:*

$$Minimal_x(F) = F(x) \cdot \neg\exists y\ [Strict\_Contain(x, y) \cdot F(y)] \qquad (31)$$

Proof: The term $\exists y\ [Strict\_Contain(x, y) \cdot F(y)]$ is true iff there is a positional-set y in F such that $x \supset y$. In such a case, x cannot be in the minimal set by definition, and can be subtracted out. What remains is exactly the minimal set of states set in $F(x)$. □

Theorem 4.12 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set union relation** *tests if positional-set y represents the union of all state sets in A, and can be computed by:*

$$Set\_Union_x(\chi_A, y) = \prod_{k=1}^{n} y_k \Leftrightarrow [\exists x\ \chi_A(x) \cdot x_k] \qquad (32)$$

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

Alternatively, we implemented the Set_Union operation as a recursive BDD operator. Bitwise OR is performed at the BDD DAG level, by traversing the BDD and performing OR on BDD vertices with the variables of interest.

Theorem 4.13 *Given a set of positional-sets $F(x)$ and an array of the Boolean variables x, the union of positional-sets in F with respect to x can be computed by the BDD operator $Bitwise\_Or(F, 0, x)$, assuming that the variables in x are ordered last:*

```
function $Bitwise\_Or(F, k, x)$ {
    if ($k \geq |x|$) return $F$
    $t = top\_var(F)$
    $v = x[k]$
    if ($t < v$) {
        $T = Bitwise\_Or(F_t, k, x)$
        $E = Bitwise\_Or(F_{\bar{t}}, k, x)$
        return $ITE(t, T, E)$
    } else {
        if ($F_v = \mathbf{0}$) return $\bar{v} \cdot Bitwise\_Or(F_{\bar{v}}, k+1, x)$
        else return $v \cdot Bitwise\_Or(F_v + F_{\bar{v}}, k+1, x)$
    }
}
```

Proof: v denotes the k-th variable in the array x. Assuming that the variables in x are ordered last, the above recursion terminates after all of them have been processed ($k \geq |x|$, and a $\mathbf{0}$ or a $\mathbf{1}$ is returned as F). At a BDD vertex where $t < v$, the recursion has not reached a variable of interest yet, and we simply recurse down its right and left children and merge the $Bitwise\_Or$ results by creating a new vertex $ITE(t, T, E)$. If $t \geq v$, we have to perform the bitwise OR operation on variable v. If $F_v = \mathbf{0}$, variable v never takes a value 1 in any satisfying assignments of F, so it is set to 0 by $\overline{v}$. The bitwise OR of the remaining variables is given by $Bitwise\_Or(F_{\overline{v}}, k+1, x)$. Otherwise if $F_v \neq \mathbf{0}$, there exists a satisfying assignment of F in which $v = 1$. So v is set to 1, while a bitwise OR is performed over all remaining satisfying assignments of F, i.e. $F_v + F_{\overline{v}}$. □

This recursive BDD operator is very fast, but unfortunately, its operation is valid only if the variables to be bitwise OR are at the bottom of the BDD DAG. So to execute this BDD operator, we need to perform variable substitutions before and after the operation. Experimentally, these substitution steps are too slow to be practical and sometimes cause exponential blowup in the BDD size.

Theorem 4.14 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set intersection relation** *tests if positional-set y represents the intersection of all state sets in A, and can be computed by:*

$$Set\_Intersect_x(\chi_A, y) = \prod_{k=1}^{n} y_k \Leftrightarrow [\forall x \; \chi_A(x) \cdot x_k] \qquad (33)$$

Proof: For each position k, the right hand expression sets y_k to 1 iff the kth bit of all x in χ_A is a 1. This implies that the positional-set y will contain the kth element iff all positional-sets x in χ_A have k has a member. Effectively, the right hand expression performs a multiple bitwise AND on all positional-sets of χ_A to form a single positional-set y which represents the intersection of all such positional-sets. □

4.4 k-out-of-n Positional-sets

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

An efficient way of constructing and storing such sets of k-tuple state 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 state i to be in the positional-set, x_i takes the value 1 and we follow the right outgoing arc. In doing so, we still have $i-1$ states/variables left to be processed. As we have put state i in the positional-set, we still have to add exactly $j-1$ states into the positional-set. That is why the right child of $Tuple_{i,j}(x)$ should be $Tuple_{i-1,j-1}(x)$. Similarly, the left child is $Tuple_{i-1,j}(x)$ because state i has not been put in the positional-set and we have $j-1$ states/variables left. Thus, the BDD for $Tuple_{i,j}$ can be constructed by the following algorithm:

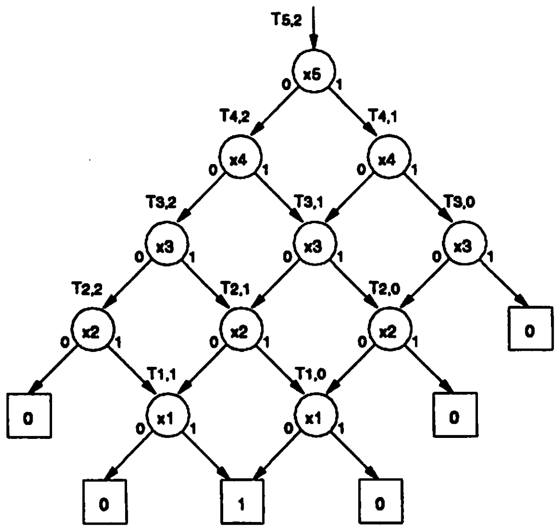

Figure 1: BDD representing $Tuple_{5,2}(x)$.

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

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

5 Implicit Computations for State Minimization

In this section, we will give a series of theorems stating how the sets defined in section 2.2 can be computed implicitly. An appendix is also provided where the main steps of the procedure are demonstrated on an example.

5.1 Output Incompatible Pairs

Theorem 5.1 *The set of output incompatible pairs, $\mathcal{OICP}(y,z)$, can be computed as:*

$$\mathcal{OICP}(y,z) = Tuple_{n,1}(y) \cdot Tuple_{n,1}(z) \cdot \neg\forall i\, \exists o\, \mathcal{O}(i,y,o) \cdot \mathcal{O}(i,z,o) \tag{34}$$

Proof: By definition 2.2, states y and z are output compatible iff their specified outputs match on all inputs, i.e. $\forall i\, \exists o\, \mathcal{O}(i,y,o) \cdot \mathcal{O}(i,z,o)$. $\mathcal{OICP}(y,z)$ simply contains all state pairs (y,z) which are *not* output compatible. □

5.2 Incompatible Pairs

Theorem 5.2 *The set of incompatible pairs, $\mathcal{ICP}(y,z)$, can be computed with the following fixed-point computation:*

$$\begin{aligned} \mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) & (35)\\ \mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z) + \exists i,u \, \{ \mathcal{T}(i,y,u) \cdot [\, \exists v \, \mathcal{T}(i,z,v) \cdot \mathcal{ICP}_k(u,v) \,] \} & (36) \end{aligned}$$

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

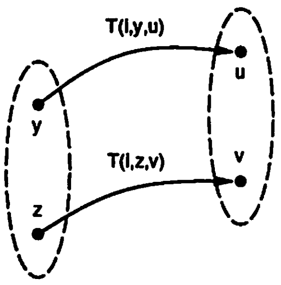

Figure 2: Finding incompatible pairs.

5.3 Compatibles

Theorem 5.3 *Given an incompatible pair of states (y,z), a position-set c satisfies $Contain\_Union(y,z,c)$ iff c contains both state y and state z. This constraint can be obtained by:*

$$Contain\_Union(y,z,c) = \prod_{k=1}^{n} y_k + z_k \Rightarrow c_k \qquad (37)$$

Proof: Note the similarity in the computations of $Contain\_Union(y,z,c)$ and $Union(y,z,c)$. $Contain\_Union(y,z,c)$ performs bitwise OR on singletons y and z. If either of their k-bit is 1, the corresponding c_k bit is constrained to 1. Otherwise, c_k can take any values (i.e. don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. One sees, from Figures 12 and 13, that $Contain\_Union(y,z,c)$ effectively performs bitwise OR and then changes the zero positions (0) to a don't-care ($-$). Thus, it generates all the positional-sets c which contain at least one incompatible state pair. □

Theorem 5.4 *The set of incompatibles, $\mathcal{IC}(c)$, can be computed as:*

$$\mathcal{IC}(c) = \exists y,z \, \mathcal{ICP}(y,z) \cdot Contain\_Union(y,z,c) \qquad (38)$$

Proof: The term $\mathcal{ICP}(y,z) \cdot Contain\_Union(y,z,c)$ generates all incompatible constraints. The right hand expression says that a positional-set c is an incompatible iff there exists an incompatible state pair $(y,z) \in \mathcal{ICP}(y,z)$ such that c contains both states y and z, because (y,z,c) satisfies the $Contain\_Union$. □

Theorem 5.5 *The set of compatibles, $C(c)$, can be computed as:*

$$C(c) = \neg\mathcal{IC}(c) \cdot \neg Tuple_{n,0}(c) \qquad (39)$$

Proof: The set of compatibles is simply the set of all subsets excluding the incompatible set of states, and is obtained by $1 \cdot \neg\mathcal{IC}(c) = \neg\mathcal{IC}(c)$. The empty positional-set is excluded from $C(c)$. □

5.4 Implied Classes of a Compatible

Lemma 5.1 *The set of singleton next states implied by a compatible c under input i, $\mathcal{F}(c,i,n)$, can be computed by:*

$$\mathcal{F}(c,i,n) = \exists p\,[\mathcal{T}(i,p,n)\cdot \mathcal{C}(c)\cdot Contain(c,p)] \tag{40}$$

Proof: Given a compatible $c \in \mathcal{C}(c)$ and an input i, a next state n is in relation $\mathcal{F}(c,i,n)$ with c and i (i.e. state n is implied by compatible c under input i) iff the right hand expression is true. i.e. if there exists a present state $p \in c$ and n is the next state of p on input i. □

Note that the implied next states are represented as singleton states in $\mathcal{F}(c,i,n)$. For each compatible c and input i, subsequent computations require that the corresponding singletons are combined into a single positional-set. Alternate computations that use $\mathcal{F}(c,i,n)$ directly will be given in section 7.2.

Theorem 5.6 *The implied classes of a compatible c, $\mathcal{CI}(c,d)$, can be computed by:*

$$\mathcal{CI}(c,d) = \exists i\,[\exists n\,\mathcal{F}(c,i,n)]\cdot Set\_Union_n(\mathcal{F}(c,i,n),d) \tag{41}$$

Proof: By definition, the implied class of c and i is just the set of next states implied by c and i. $\mathcal{F}(c,i,n)$ contains such next states in singleton positional-set form and $Set\_Union_n(\mathcal{F}(c,i,n),d)$ will perform the bitwise OR on all of them to produce a positional-set d which represents the union of the singleton sets. As F also depends on c and i, the Set_Union operation may produce triples (c,i,d) where c and i may not be a valid compatible and input respectively. So the term $[\exists n\,\mathcal{F}(c,i,n)]$ is needed to prune away invalid triples from the relation. Finally the class set of c defined as the set over different inputs of all implied next states of c is obtained simply by an existential quantification of the inputs i. □

5.5 Class Sets of Compatibles

Theorem 5.7 *The class set of the compatible c, $\mathcal{CCS}(c,d)$, can be computed as:*

$$\mathcal{CCS}(c,d) = Maximal_d(\mathcal{CI}(c,d))\cdot \neg Contain(c,d)\cdot \neg Tuple_{n,1}(d) \tag{42}$$

Proof: Given a compatible c, $Maximal_d(\mathcal{CI}(c,d))$ gives all its implied classes d which are not strictly contained by any other implied classes. This corresponds to condition 3 in definition 2.11 although a weaker condition, $C_{ij} \not\subset C_{ik}$, is used here because our implicit computation operates on all implied classes at once. By condition 2 in definition 2.11, we prune away implied classes d which are contained in their corresponding compatibles c. Then the singleton implied classes are thrown away according to condition 1. □

5.6 Prime Compatibles

Theorem 5.8 *A compatible c' dominates a compatible c iff the following $Dominate(c',c)$ relation is true:*

$$Dominate(c',c) = Strict\_Contain(c',c)\cdot\ Set\_Contain_d(\mathcal{CCS}(c,d),\mathcal{CCS}(c',d)) \tag{43}$$

Proof: The two terms on the right hand expression correspond to the two conditions for c' to dominate c according to definition 2.12. Since compatibles c and c' are represented as positional-sets, $c \supset c'$ is computed by $Strict\_Contain(c',c)$, as defined by theorem 4.3. On the other hand, class sets are sets of sets of states and are represented by their characteristic functions. Containment between such sets of sets of states is computed by $\forall d\,\mathcal{CCS}(c',d) \Rightarrow \mathcal{CCS}(c,d)$, as described by theorem 4.8. □

Theorem 5.9 *The set of prime compatibles, $\mathcal{PC}(c)$, can be computed as:*

$$\mathcal{PC}(c) = \mathcal{C}(c) \cdot \neg\exists c' \; Dominate(c', c) \cdot \mathcal{C}(c') \tag{44}$$

Proof: By definition 2.13, a compatible $c \in C(c)$ is not a prime compatible if it is dominated by another compatible $c' \in C(c)$. This condition is captured by the expression $\exists c' \, \{Dominate(c', c) \cdot \mathcal{C}(c')\}$. The set of prime compatibles is simply given by the set of compatibles excluding those that are dominated by other compatibles. □

Theorem 5.10 *The set of prime compatibles with class sets, $\mathcal{PCCS}(c, d)$, can be computed as:*

$$\mathcal{PCCS}(c, d) = \mathcal{PC}(c) \cdot \mathcal{CCS}(c, d) \tag{45}$$

Proof: Obvious. □

Theorem 5.11 *The set of essential prime compatibles, $\mathcal{EPC}(c)$, can be computed as:*

$$\mathcal{EPC}(c) = \mathcal{PC}(c) \cdot \sum_{k=1}^{n} \{c_k \cdot \neg[\exists c' \; c'_k \cdot \mathcal{PC}(c') \cdot \neg Equal(c, c')]\} \tag{46}$$

The set of non-essential prime compatibles, $\mathcal{NEPC}$, which constitutes the columns of the covering table, can be computed as:

$$\mathcal{NEPC}(c) = \mathcal{PC}(c) \cdot \neg\mathcal{EPC}(c) \tag{47}$$

6 Construction of the implicit covering table

A relation $T(c, z, \tilde{c}, e)$ representing the entries of the covering matrix can be defined as the disjunction of the relations $UT(c, z, \tilde{c}, e)$ and $BT(c, z, \tilde{c}, e)$ defined below. $UT(c, z, \tilde{c}, e)$ is the unate part and $BT(c, z, \tilde{c}, e)$ is the binate part. Both UT and BT are defined on the variables $c, z, \tilde{c}, e$, where c, z are indexes of rows, $\tilde{c}$ are indexes of columns and e is a Boolean variable that indicates the presence of an entry of value 0 or 1 at row c, z and column $\tilde{c}$. A unate part contains only entries assuming value 1, while a binate part contains some entries with value 0 and some entries with value 1. Notice that given a row and a column there is at most one entry in the matrix, either with value 0 or with value 1; it cannot happen that for the same row and column there are two entries, one carrying value 0 and one carrying value 1.

If the original FSM has n states, the relation representing the covering table has $3n + 1$ variables. This is the crucial point of being the representation implicit. In the explicit case, we would have as many columns as there are primes and as many rows as there are clauses. Both rows and columns could be exponential in the number of states of the FSM. In the case of our implicit representation instead the number of variables is linear in the number of states.

Theorem 6.1 *The unate part of the relation T is given by:*

$$UT(c, z, \tilde{c}, e) = Tuple_{n,1}(z) \cdot \mathcal{PC}(\tilde{c}) \cdot Contain(\tilde{c}, z) \cdot (e = 1) \tag{48}$$

Notice that c can be any vector of n variables, because rows in UT are uniquely distinguished by the fact that z's are singletons. UT represents the covering clauses of the exact formulation of state minimization ([8]).

Theorem 6.2 *The binate part of the relation T is given by:*

$$BT(c, z, \tilde{c}, e) = \mathcal{PCCS}(c, z) \cdot \mathcal{PC}(\tilde{c}) \cdot \{Contain(\tilde{c}, z) \cdot (e = 1) + Equal(c, \tilde{c}) \cdot (e = 0)\} \tag{49}$$

BT represents the closure clauses of the exact formulation of state minimization ([8]).[2]

Theorem 6.3 *The covering matrix is given by:*

$$T(c,z,\tilde{c},e) = UT(c,z,\tilde{c},e) + BT(c,z,\tilde{c},e) \tag{51}$$

Notice that

- $\{\tilde{c}$, s.t. $PC(\tilde{c})\}$ are column indexes, i.e. prime compatibles.
- $\{(c,z)$, s.t. $PCCS(c,z)\}$ are row indexes, i.e. clauses.
- Elements of T can be 0 or 1 or no entry and they are indicated respectively by $e = 0$, $e = 1$, no representation.

  One subtracts from $PCCS$ the cubes z of all zeroes or singletons, because they denote no closure condition.

7 Improvements on Implicit Algorithm

The experiments reported in section 9 identified two bottlenecks in the computations described in section 5:

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

Sections 7.1 and 7.2 describes alternative methods to perform those computations. Section 7.3 shows how maximal compatibles can be used with advantage in the computation of prime compatibles.

7.1 Computation of Incompatible Pairs using Generalized Cofactor

This subsection describes some variations of the fixed-point computation of incompatible pairs $\mathcal{ICP}(y,z)$, described in section 5.2. Each iteration of the computation of equation 36 can be viewed as an inverse image projection from a set of state pairs in $ICP_k(u,v)$ to a set of states pairs in $ICP_{k+1}(y,z)$ via the product transition relation $\mathcal{T}(i,y,u)\cdot\mathcal{T}(i,z,v)$. In the original method, all state pairs in $ICP_{k+1}(u,v)$ are projected during the $k+2$nd iteration. This is not necessary because if the projected pair (y',z') of ICP_{k+1} is actually in ICP_k as shown in figure 3[3], we can be sure that its projection (y'',z'') has already been calculated in a previous iteration. Thus at the $k+2$nd iteration, we need only to project the *new* incompatible state pairs discovered at the $k+1$st iteration, as it is done in the following modification of the fixed-point computation of section 5.2.

$$\begin{aligned}
\mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z)(=NEW(y,z)) && (52)\\
TMP(y,z) &= \exists i,u\,\{\,\mathcal{T}(i,y,u)\cdot[\,\exists v\,\mathcal{T}(i,z,v)\cdot NEW(u,v)\,]\,\} && (53)\\
NEW(y,z) &= TMP(y,z)\cdot\neg\mathcal{ICP}_k(y,z) && (54)\\
\mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z)+NEW(y,z) && (55)
\end{aligned}$$

[2]Notice that the following would be wrong

$$BT(c,z,\tilde{c},e) = \mathcal{PCCS}(c,z)\cdot\mathcal{PC}(\tilde{c})\cdot\{(Contain(\tilde{c},z)\Rightarrow(e=1))+(Equal(c,\tilde{c})\Rightarrow(e=0))\} \tag{50}$$

[3]Figure 3 represents the inverse image projections with direct arrows.

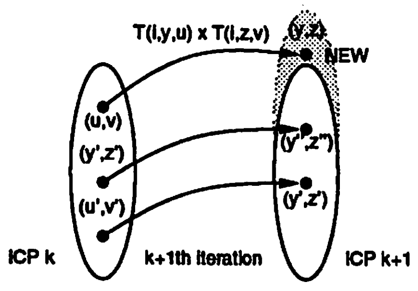

Figure 3: Finding incompatible pairs.

Instead of finding a minimum cardinality set of state pairs for projection, a minimal 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 using the generalized cofactor ([23]) using $\mathcal{ICP}_k(y,z)$ as the don't care set:

$$NEW(y,z) = NEW(y,z)|_{\neg\mathcal{ICP}_k(y,z)} \tag{56}$$

As a result, the geometric mean of the ratio of CPU time for computing $\mathcal{ICP}$ with this generalized cofactor method vs. the original method is 0.678.

7.2 Handling of Closure Information

For FSM's with many compatibles, the most time-consuming part of our implicit algorithm is the computation of implied classes and class sets corresponding to the compatibles. The complexity arises because these implicit computations deal with two sets of variables in each relation, c representing a compatible and d representing its implied class or class set. Since each compatible may have a different class set, the size of the corresponding BDD's may blowup during the computation.

A way to cope with this problem is to represent the class sets by means of singletons, as done in the following series of computations.

Theorem 7.1 *One can prune the relation $F(c,i,n)$ of compatibles with implied next states to obtain the class set by:*

$$\begin{aligned}
F(c,i,n) &= \exists p\,[\mathcal{T}(i,p,n)\cdot\mathcal{C}(c)\cdot Contain(c,p)] && (57)\\
I(c,i) &= \exists n\, n'\; F(c,i,n)\cdot F(c,i,n')\cdot\neg Equal(n,n') && (58)\\
F(c,i,n) &= F(c,i,n)\cdot I(c,i) && (59)\\
J(c,i) &= \exists n\, F(c,i,n)\cdot\neg Contain(c,n) && (60)\\
F(c,i,n) &= F(c,i,n)\cdot J(c,i) && (61)\\
K(c,i) &= \exists\, i'\,[\forall n\, F(c,i',n)\Rightarrow F(c,i,n)+\neg J(c,i')]\cdot\neg[\forall n F(c,i,n)\;\Rightarrow F(c,i',n)+\neg J(c,i')] &&\\
&= \exists\, i'\,[\neg\exists n\, F(c,i',n)\cdot\neg F(c,i,n)\cdot J(c,i')]\cdot[\exists n\, F(c,i,n)\cdot\neg F(c,i',n)\cdot J(c,i')] && (62)\\
F(c,i,n) &= F(c,i,n) - K(c,i) && (63)
\end{aligned}$$

Proof: By definition 2.11, an implied class C_i of compatible c can be in a class set only if

1. C_i has more than one element,
2. $C_i \not\subseteq c$,
3. $C_i \not\subseteq C_{i'}$ if $C_{i'} \in$ class set.

$I(c,i)$ computes all implied classes C_i which contains at least two distinct implied states n and n', i.e. all implied classes with more than one element. Equation 59 prunes the set F accordingly. $J(c,i)$ contains all remaining implied classes not contained in c.

We need to modify slightly the 3rd condition to be able to compute it implicitly using BDD's. From the set of implied classes, we want to take away an implied class C_i iff $C_i \subset C_{i'}$. It is $C_i \subset C_{i'}$ iff $C_i \subseteq C_{i'}$ and $C_{i'} \not\subseteq C_i$. The $Set\_Contain_n(F(c,i,n), F(c,i',n))$ operation can be used to test if $C_i \subset C_{i'}$, but since its result may include invalid (c,i') pairs (i.e. implied classes) the terms $J(c,i')$ are needed in the equation. In the last equation $K(c,i)$ is subtracted away, instead than AND-ed as $I(c,i)$ and $J(c,i)$, because it is the complement of the 3rd condition. □

Theorem 7.2 *The condition that compatible c' dominates compatible c is captured by:*

$$Dominate(c',c) = Strict\_Contain(c',c) \cdot \forall i' \; \exists i \; Set\_Contain_n(F(c,i,n), F(c',i',n)) \tag{64}$$

Proof: 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 found using theorem 5.9.

7.3 Methods using Maximal Compatibles

Theorem 7.3 *The set of all maximal compatibles $\mathcal{MC}(c)$ can be computed as:*

$$\mathcal{MC}(c) = Maximal_c(\mathcal{C}(c)) \tag{65}$$

Proof: By definition 2.10, the set of maximal compatibles is simply the maximal set of positional-sets in $\mathcal{C}(c)$ with respect to c. □

Note that the algorithm given in section 5 does not rely on the computation of the set of maximal compatibles, whereas the classical method in [8] does. We are going now to present alternative implicit algorithms that require their computation.

7.3.1 Compatible Pruning by Maximal Compatibles with Void Class Set

Theorem 7.4 *The set of singleton next states implied by a maximal compatible c under input i, $\mathcal{F}(c,i,y)$, can be computed by:*

$$\mathcal{F}(c,i,n) = \exists p \; [\mathcal{T}(i,p,n) \cdot \mathcal{MC}(c) \cdot Contain(c,p)] \tag{66}$$

The class set information for the maximal compatibles can then be obtained using the class set generation procedure described in theorem 7.1.

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

$$\mathcal{MCV}(c) = \mathcal{MC}(c) \cdot \neg \exists i \; K(c,i) \tag{67}$$

where $K(c,i)$ is given by equation 63.

The set of compatibles can then be pruned by $\mathcal{MCV}(c)$:

$$\mathcal{C}(c) = \mathcal{C}(c) \cdot \neg \exists c' \; \mathcal{MCV}(c) \cdot Contain(c',c) \tag{68}$$

7.3.2 Slicing Procedure for Prime Compatible Generation

The following slicing procedure is an implicit version of the procedure outlined at the end of section 2.2.

```
$\mathcal{PC}(c) = \emptyset$
for $k = n$ to 1 {
    $\mathcal{MC}_k(c) = \mathcal{MC}(c) \cdot Tuple_{n,k}(c)$
    $\mathcal{C}_k(c) = \mathcal{C}(c) \cdot Tuple_{n,k}(c) \cdot \neg\mathcal{MC}_k(c)$
    $F_{\mathcal{C}_k}(c, i, n) = Prune(\mathcal{C}_k(c), T(i, p, n))$
    $F_{\mathcal{PC}}(c, i, n) = Prune(\mathcal{PC}(c), T(i, p, n))$
    $Dominate(c', c) = Strict\_Contain(c', c) \cdot \forall i'\ \exists i\ Set\_Contain_n(F_{\mathcal{PC}}(c, i, n), F_{\mathcal{C}_k}(c', i', n))$
    $\mathcal{PC}_k(c) = \mathcal{C}(c) \cdot \neg\exists c'\ Dominate(c', c) \cdot \mathcal{C}(c')$
    $\mathcal{PC}(c) = \mathcal{PC}(c) + \mathcal{PC}_k(c) + \mathcal{MC}_k(c)$
}
```

$\mathcal{PC}(c)$ 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 with cardinality k excluding those in $\mathcal{MC}_k(c)$. $Prune(\mathcal{C}_k(c), T(i, p, n))$ is the class set pruning procedure described in theorem 7.1 by substituting $\mathcal{C}_k(c)$ for $\mathcal{C}(c)$, and $F_{\mathcal{C}}(c, i, n)$ for $F(c, i, n)$ in the equations. $Prune(\mathcal{PC}(c), T(i, p, n))$ is similarly defined. So $F_{\mathcal{C}_k}(c, i, n)$ and $F_{\mathcal{PC}}(c, i, n)$ contains the class sets of $\mathcal{C}_k(c)$ and $\mathcal{PC}(c)$ respectively. To test for $Dominance(c', c)$, we only need to know if a compatible $c \in \mathcal{C}_k$ is dominated by an already discovered prime compatible in $\mathcal{PC}(c)$, 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 cardinality 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, this slicing method, during BDD construction, uses on average half memory as compared to the method in section 7.2.

8 Implementation Details

8.1 BDD Variable Assignment

When dealing with BDD's, common wisdom is to keep the number of BDD variables used to a minimum. The rationale is that the smaller the number of BDD variables involved, the less probable is that a BDD operation will cause exponential blowup in the BDD size. In our case

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 the fact 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 36, and 3 sets p, n and c in equation 40. 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 onto 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 is shown in figure 4.

There is a conflict with the above BDD variable assignment in equation 38. Variable c is assigned the same BDD variables as variable y in these equations. To get around it, an extra variable e is used instead:

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

Figure 4: Assignments of equation variables to BDD variables

$$\mathcal{IC}(c) = [e \rightarrow c]\exists y, z\, \mathcal{ICP}(y, z) \cdot Contain\_Union(y, z, e) \tag{69}$$

. Note that two functions containing different variables being assigned to the same BDD variable, e.g. $\mathcal{T}(i, p, n)$ and $\mathcal{CCS}(c', d')$, can co-exist within a multi-rooted BDD at the same time, without any interference. Conflict will occur only when they become operands to a BDD operation. Actually, such overlapping functions can be constructed and manipulated more efficiently because of possible hits in the unique and computed hash tables in a BDD package [1].

8.2 BDD Variable Ordering

The equality, containment, strict containment, maximal and minimal relations described in section 4 have exponential BDD's size if the different sets of BDD variables are not interleaved with each other. Both for space and time efficiency, the four sets of BDD variables have to be interleaved.

It is found that the ordering between individual state variables within a set of BDD variables is also important, especially when handling the closure information. The heuristics we use is to put the states that occur most frequently in the compatibles at the top of the BDD. This should leave the BDD sparse in the lower part of the BDD 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 instead.

8.3 Using Don't Cares in the Positional-set Space

The main advantage of our positional-set representation of FSM's is that, with a single multi-rooted BDD, sets of sets of states can be represented. As a result, we can compactly represent and manipulate sets of compatibles ($\mathcal{C}$), prime compatibles ($\mathcal{PC}$), etc. 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 > 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 34 to 36 manipulate a product of two sets of singleton states (y, z). The don't care condition with respect to this pair of singletons is captured by:

$$DC(y, z) = \neg Tuple_{n,0}(y) \cdot \neg Tuple_{n,1}(y) + \neg Tuple_{n,0}(z) \cdot \neg Tuple_{n,1}(z) \tag{70}$$

and can be used to simplify the BDD computation of these sets.

9 Experimental Results

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

1. The MCNC benchmark and other examples.

2. FSM's generated by a synthesis procedure for asynchronous logic [13].
3. A constructed family of FSM's that exhibit a large number of prime compatibles.
4. Random FSM's.

We discuss features of the experiments and results in different subsections. Comparisons are made with STAMINA, a program that represents the state-of-art for state minimization based on explicit techniques. The program STAMINA was run with the option -P to compute all primes. All run times are reported in CPU seconds on a DEC DS5900/260 with 440 Mb of memory.

9.1 Examples from MCNC Benchmark and Others

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

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	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	11	13
ex3	10	10	195	91	91	1	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	125	125
lion9	9	5	20	5	2	0	0
mark1	15	12	41	18	11	0	0
scf	121	12	1201	175	87	26	0
squares	371	45	473	307	0	761	1
tbk	32	16	48	48	48	8	1
tma	20	15	35	20	4	1	0
train11	11	5	85	17	15	0	0
viterbi	68	5	329	57	3	8	0

Table 1: Examples from the MCNC Benchmark and others.

9.2 Examples of FSM's from Asynchronous Synthesis

Table 2 reports the results of a benchmark of FSM's generated as intermediate steps of an asynchronous synthesis procedure [13]. We notice that STAMINA ran out of memory on the examples *vmebus.master.m, isend, pe-rcv-ifc.fc, pe-send-ifc.fc*, while ISM was able to complete them. These examples (with the exception of *vbe4a*) have a number of primes below 1000. To explain the data reported in Table 2, we notice that in order to compute the prime compatibles, the set of compatibles needs to be generated too. The compatibles of the FSM's of this benchmark are usually of large cardinality and therefore their enumeration causes a combinatorial explosion. So the huge size of the set of compatibles accounts for the large running times and/or out-of-memory failures. About the behavior of ISM, we underline that the running times track well with the size of the set of compatibles and that in significant cases they are well below those of STAMINA (*pe-rcv-ifc.fc.m, pe-send-ifc.fc.m, vbe4a*).

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	STAMINA
alex1	42	787	55928	787	787	40	16
future	36	49	7.92986e8	49	49	8	0
future.m	28	16	2.62144e7	16	16	2	0
intel_edge.dummy	28	120	9432	396	396	40	3
isend	40	128	22207	480	480	19	spaceout
isend.m	20	15	22207	19	19	1	0
mp-forward-pkt	20	1	1.04858e6	1	0	0	0
nak-pa	56	8	4.74109e15	8	8	17	0
nak-pa.m	18	8	44799	8	8	1	0
pe-rcv-ifc.fc	46	28	1.52816e11	148	148	22	spaceout
pe-rcv-ifc.fc.m	27	18	1.79379e6	38	38	3	147
pe-send-ifc.fc	70	39	5.07174e17	506	506	701	spaceout
pe-send-ifc.fc.m	26	6	8.97843e6	23	22	3	312
ram-read-sbuf	36	2	3.00648e10	2	0	2	0
sbuf-ram-write	58	24	1.4336e6	24	24	15	0
sbuf-ram-write.m	24	12	1.4336e6	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	1.7562e12	2072	2072	141	167
vbe4a.m	22	13	73471	13	13	2	0
vbe6a.m	16	8	527	8	4	1	0
vmebus.master.m	32	10	5.04955e7	28	28	16	spaceout

Table 2: Asynchronous FSM benchmark.

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

Table 3 shows the results of running a parametrized set of FSM's constructed to be compatible with a given collection of examples of input/output behavior [7]. These machines exhibit very large number of compatibles.

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

machine	# state	# compat.	# prime compat.	CPU time (sec) ism	CPU time (sec) stamina
threer.6	7	55	4	0	0
threer.10	11	671	112	0	0
threer.14	15	6335	1052	0	5
threer.20	21	16829	3936	2	159
threer.25	26	60857	17372	8	215
threer.30	31	97849	33064	50	1344
threer.35	36	223705	82776	66	spaceout
threer.40	41	1456805	529420	156	spaceout
threer.45	46	5532323	2225468	1213	spaceout
threer.50	51	16809300	7246284	1142	spaceout
threer.55	55	36223548	15550092	1999	spaceout

Table 3: Learning I/O sequences benchmark.

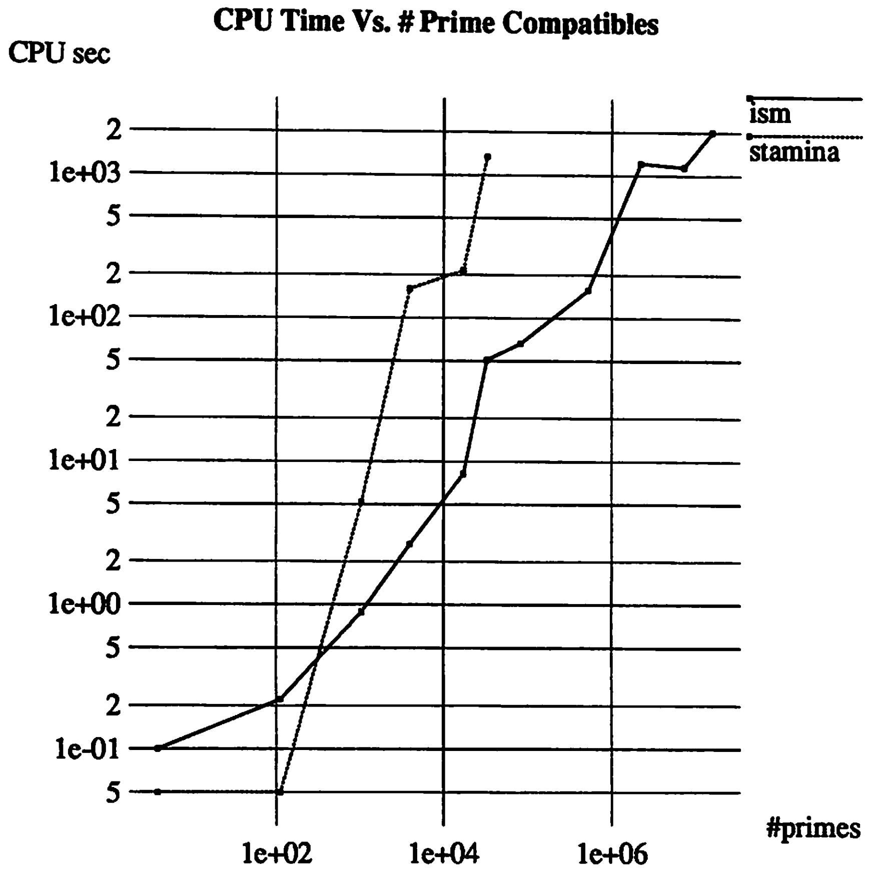

Figure 5: Comparison between ISM and STAMINA on learning I/O sequences benchmark.

9.4 A Family of FSM's with Exponentially Many Primes

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

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

Table 3 shows the results of running increasingly larger FSM's of the family. While ISM is able to generate sets of prime compatibles of cardinality up to 2^{1200} with reasonable running times, STAMINA, based on an explicit enumeration runs out of memory soon (and where it completes, it takes much longer).

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	STAMINA
rubin12	12	3^4	2^8-1	2^8-1	2^8-1	0	4
rubin18	18	3^6	$2^{12}-1$	$2^{12}-1$	$2^{12}-1$	1	751
rubin24	24	3^8	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$	1	spaceout
rubin150	150	3^{50}	$2^{100}-1$	$2^{100}-1$	$2^{100}-1$	88	spaceout
rubin300	300	3^{100}	$2^{200}-1$	$2^{200}-1$	$2^{200}-1$	452	spaceout
rubin450	450	3^{150}	$2^{300}-1$	$2^{300}-1$	$2^{300}-1$	1458	spaceout
rubin600	600	3^{200}	$2^{400}-1$	$2^{400}-1$	$2^{400}-1$	3106	spaceout
rubin750	750	3^{250}	$2^{500}-1$	$2^{500}-1$	$2^{500}-1$	7106	spaceout
rubin900	900	3^{300}	$2^{600}-1$	$2^{600}-1$	$2^{600}-1$	11588	spaceout
rubin1050	1050	3^{350}	$2^{700}-1$	$2^{700}-1$	$2^{700}-1$	21048	spaceout
rubin1200	1200	3^{400}	$2^{800}-1$	$2^{800}-1$	$2^{800}-1$	32202	spaceout
rubin1500	1500	3^{500}	$2^{1000}-1$	$2^{1000}-1$	$2^{1000}-1$	77590	spaceout
rubin1800	1800	3^{600}	$2^{1200}-1$	$2^{1200}-1$	$2^{1200}-1$	142824	spaceout

Table 4: Constructed FSM's.

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

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

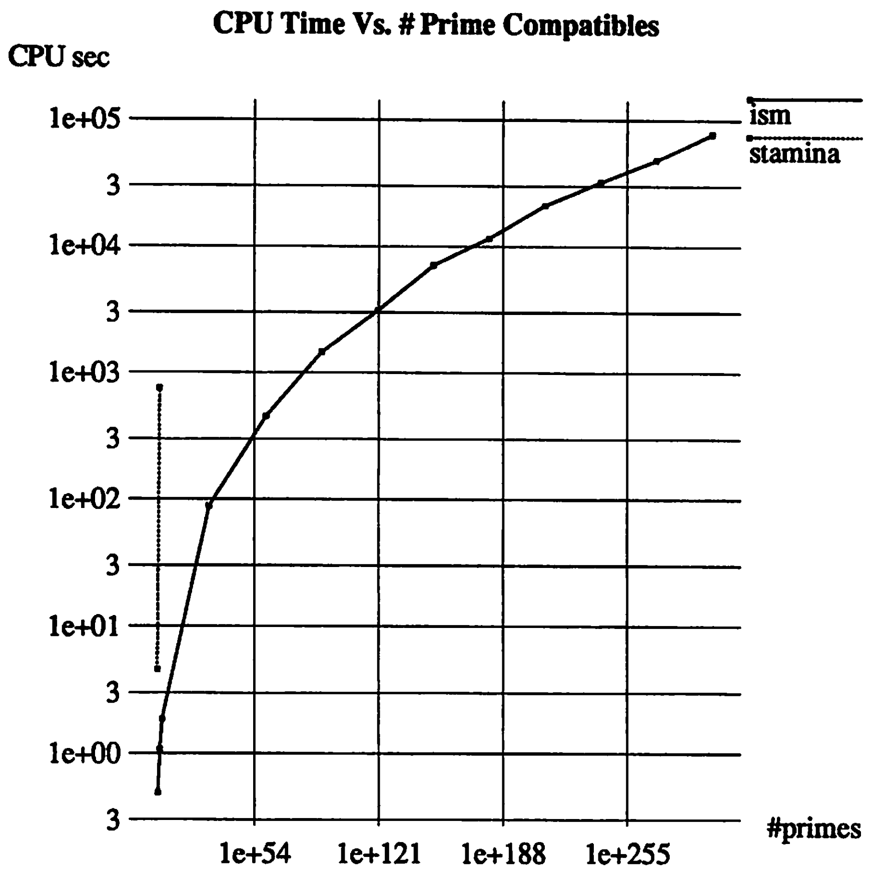

Figure 6: Comparison between ISM and STAMINA on constructed FSM's.

9.5 FSM's with Many Maximals

Table 4 shows the results of running some examples from a set of FSM's constructed to have a large number of maximal compatibles. The examples *jac4, jc43, jc44, jc45, jc46, jc47* are due to R. Jacoby and have been kindly provided by J.-K. Rho of UC Boulder. The example *lavagno* is from asynchronous synthesis as those reported in Section 9.2. For these examples the program STAMINA was run with the option -M to compute all maximals. While ISM could complete on them in reasonable running times, STAMINA could not complete on *jac4* and completed the other ones with running times exceeding those of ISM by one or two order of magnitudes. Notice that ISM could also compute the set of all compatibles even though the computation of prime compatibles cannot be carried to the end while STAMINA failed on both.

		# max		# prime	CPU time (sec)	
machine	# states	compat.	# compat.	compat.	ISM	STAMINA
jac4	65	3859641	41593120	?	34	spaceout
jc43	45	82431	1.55634e6	?	13	7739
jc44	55	4785	7.58463e9	?	20	662
jc45	40	17323	480028	?	10	1211
jc46	42	26086	1.1536e6	?	11	2076
jc47	51	397514	1.12096e7	?	19	41297
lavagno	65	47971	9.1631e6	?	163	40472

Table 5: FSM's with many maximals.

9.6 Randomly Generated FSM's

We investigated also whether randomly generated FSM's have a large number of prime compatibles. A program was written to generate random FSM's[6]. A small percentage of the randomly generated FSM's were found to exhibit this behavior. Table 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.

9.7 Summary of the Results

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

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

We expect that similar cases are going to arise, for instance, in the minimization of interacting FSM's. It has been reported by Rho and Somenzi [19] that the exact state minimization of the driven machine of a pair

[6]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.	#$\mathcal{NEPC}$	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	26	spaceout
ex2.285	19	2	393215	121501	121500	17	spaceout
ex2.304	19	2	393215	264079	264079	94	spaceout
ex2.423	19	4	204799	160494	160494	112	spaceout
ex2.680	19	2	327679	192803	192803	156	spaceout

Table 6: Random FSM's.

of cascaded FSM's is equivalent to the state minimization of an ISFSM that requires the computation of prime compatibles.

10 Conclusions

This paper has presented an algorithm that implicitly generates the various sets of compatibles needed to solve exactly state minimization. Compatibles, maximal compatibles, prime compatibles and implied classes are all represented implicitly by the characteristic functions of relations implemented with BDD's. If it is possible to build these BDD's, computations on these sets of compatibles are easy. The only explicit dependence is on the number of states of the initial problem. 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^{1200}, an achievement outside the scope of programs based on explicit enumeration [9]. 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 [13], and of learning I/O sequences [7]. A similar situation is expected to occur also in the synthesis of interacting FSM's [19]. 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 [21], is part of an ongoing research that will be presented in a separate paper. A complete formulation of an implicit binate covering algorithm has been already worked out and an implementation is in progress.

We underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this paper can be applied to other problems of logic synthesis and combinatorial optimization. For instance the implicit computation of maximal compatibles given here can be easily converted into an implicit computation of prime encoding-dichotomies (see [22]). Therefore the computational methods described here contribute to build a body of implicit techniques whose scope goes much beyond a specific application.

References

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

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

[3] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *The Proceedings of the IEEE*, february 1990.

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

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

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

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

[8] 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.

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

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

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

[12] L. Lavagno. Personal communication. *UC Berkeley*, December 1991.

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

A Appendix

The following example is used to illustrate the BDD's constructed in the implicit algorithm. FSM's are usually specified in STT form; a convenient way of writing STT's is given by a flow table. Each row in the flow table corresponds to a state and each column corresponds to an input combination (or vector). Each table entry gives the next state and output for the corresponding input and present state.

		encoded inputs		
		01	10	11
states	s1	s3/0	-/-	s2/-
	s2	-/-	s4/0	s6/-
	s3	s5/1	-/-	-/0
	s4	-/-	s1/1	s1/-
	s5	s1/-	-/-	s6/-

Figure 7: An FSM example.

i0	i1	p1	p2	p3	p4	p5	p6	n1	n2	n3	n4	n5	n6
0	1	0	0	0	0	0	1	0	0	0	1	0	0
0	1	0	0	0	0	1	0	1	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	0	1	0
1	0	0	0	0	1	0	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	1
1	1	0	0	0	0	1	0	0	0	0	0	0	1
1	1	0	0	0	1	0	0	1	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	0	1	0	0	0	0

Figure 8: Transition relation $\mathcal{T}(i,p,n)$ for the example.

i0	i1	p1	p2	p3	p4	p5	p6	o
0	0	0	0	0	0	0	1	-
0	0	0	0	0	0	1	0	-
0	0	0	0	0	1	0	0	-
0	0	0	0	1	0	0	0	-
0	0	0	1	0	0	0	0	-
0	0	1	0	0	0	0	0	-
0	1	0	0	0	0	0	1	-
0	1	0	0	0	0	1	0	-
0	1	0	0	0	1	0	0	-
0	1	0	0	1	0	0	0	1
0	1	0	1	0	0	0	0	-
0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	1	-
1	0	0	0	0	0	1	0	-
1	0	0	0	0	1	0	0	1
1	0	0	0	1	0	0	0	-
1	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	0	-
1	1	0	0	0	0	0	1	-
1	1	0	0	0	0	1	0	-
1	1	0	0	0	1	0	0	-
1	1	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	-
1	1	1	0	0	0	0	0	-

Figure 9: Output relation $\mathcal{O}(i,p,o)$ for the example.

y1	y2	y3	y4	y5	y6	z1	z2	z3	z4	z5	z6
0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0

Figure 10: Output compatible pairs $\mathcal{OCP}(y,z)$ for the example.

y1	y2	y3	y4	y5	y6	z1	z2	z3	z4	z5	z6
0	0	0	1	0	0	0	1	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	0	0	0

Figure 11: Output compatible Pairs $\mathcal{OCP}(y,z)$ for the example.

y1	y2	y3	y4	y5	y6	z1	z2	z3	z4	z5	z6
0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1	0	0	0

Figure 12: Incompatible pairs $\mathcal{ICP}(y,z)$ for the example.

c1	c2	c3	c4	c5	c6
-	-	-	1	-	1
-	-	1	-	1	-
1	-	-	-	1	-
-	1	-	1	-	-
1	-	1	-	-	-

Figure 13: Incompatibles $\mathcal{IC}(c)$ for the example.

c1	c2	c3	c4	c5	c6
0	0	0	0	0	1
0	0	0	0	1	-
0	0	0	1	-	0
0	0	1	0	0	-
0	0	1	1	0	0
0	1	0	0	-	-
0	1	1	0	0	-
1	0	0	0	0	-
1	0	0	1	0	0
1	1	0	0	0	-

Figure 14: Compatibles $\mathcal{C}(c)$ for the example.

c1	c2	c3	c4	c5	c6
0	0	0	1	1	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	0	0	1
1	0	0	1	0	0
1	1	0	0	0	1

Figure 15: Maximal compatibles $\mathcal{MC}(c)$ for the example.

```
i1 i2   c1 c2 c3 c4 c5 c6   n1 n2 n3 n4 n5 n6     i1 i2   c1 c2 c3 c4 c5 c6   n1 n2 n3 n4 n5 n6
-------------------------------------------------     -------------------------------------------------
0  1    0  0  0  0  1  1    0  0  0  1  0  0      1  0    0  1  0  0  1  0    0  0  0  1  0  0
0  1    0  0  0  0  1  1    1  0  0  0  0  0      1  0    0  1  0  0  1  1    0  0  0  0  1  0
0  1    0  0  0  1  1  0    1  0  0  0  0  0      1  0    0  1  0  0  1  1    0  0  0  1  0  0
0  1    0  0  1  0  0  1    0  0  0  0  1  0      1  0    0  1  1  0  0  0    0  0  0  1  0  0
0  1    0  0  1  0  0  1    0  0  0  1  0  0      1  0    0  1  1  0  0  1    0  0  0  0  1  0
0  1    0  0  1  1  0  0    0  0  0  0  1  0      1  0    0  1  1  0  0  1    0  0  0  1  0  0
0  1    0  1  0  0  0  1    0  0  0  1  0  0      1  0    1  0  0  0  0  1    0  0  0  0  1  0
0  1    0  1  0  0  1  0    1  0  0  0  0  0      1  0    1  0  0  1  0  0    1  0  0  0  0  0
0  1    0  1  0  0  1  1    0  0  0  1  0  0      1  0    1  1  0  0  0  0    0  0  0  1  0  0
0  1    0  1  0  0  1  1    1  0  0  0  0  0      1  0    1  1  0  0  0  1    0  0  0  0  1  0
0  1    0  1  1  0  0  0    0  0  0  0  1  0      1  0    1  1  0  0  0  1    0  0  0  1  0  0
0  1    0  1  1  0  0  1    0  0  0  0  1  0      1  1    0  0  0  0  1  1    0  0  0  0  0  1
0  1    0  1  1  0  0  1    0  0  0  1  0  0      1  1    0  0  0  1  1  0    0  0  0  0  0  1
0  1    1  0  0  0  0  1    0  0  0  1  0  0      1  1    0  0  0  1  1  0    1  0  0  0  0  0
0  1    1  0  0  0  0  1    0  0  1  0  0  0      1  1    0  0  1  0  0  1    0  0  0  0  0  1
0  1    1  0  0  1  0  0    0  0  1  0  0  0      1  1    0  0  1  1  0  0    1  0  0  0  0  0
0  1    1  1  0  0  0  0    0  0  1  0  0  0      1  1    0  1  0  0  0  1    0  0  0  0  0  1
0  1    1  1  0  0  0  1    0  0  0  1  0  0      1  1    0  1  0  0  1  -    0  0  0  0  0  1
0  1    1  1  0  0  0  1    0  0  1  0  0  0      1  1    0  1  1  0  0  -    0  0  0  0  0  1
1  0    0  0  0  0  1  1    0  0  0  0  1  0      1  1    1  0  0  0  0  1    0  0  0  0  0  1
1  0    0  0  0  1  1  0    1  0  0  0  0  0      1  1    1  0  0  0  0  1    0  1  0  0  0  0
1  0    0  0  1  0  0  1    0  0  0  0  1  0      1  1    1  0  0  1  0  0    0  1  0  0  0  0
1  0    0  0  1  1  0  0    1  0  0  0  0  0      1  1    1  0  0  1  0  0    1  0  0  0  0  0
1  0    0  1  0  0  0  1    0  0  0  0  1  0      1  1    1  1  0  0  0  -    0  0  0  0  0  1
1  0    0  1  0  0  0  1    0  0  0  1  0  0      1  1    1  1  0  0  0  -    0  1  0  0  0  0
```

Figure 16: $\mathcal{F}(c, i, n)$ for the example.

```
c1 c2 c3 c4 c5 c6   d1 d2 d3 d4 d5 d6     c1 c2 c3 c4 c5 c6   d1 d2 d3 d4 d5 d6
------------------------------------     ------------------------------------
0  0  0  0  1  1    0  0  0  0  0  1      0  1  1  0  0  0    0  0  0  0  0  1
0  0  0  0  1  1    0  0  0  0  1  0      0  1  1  0  0  0    0  0  0  0  1  0
0  0  0  0  1  1    1  0  0  1  0  0      0  1  1  0  0  0    0  0  0  1  0  0
0  0  0  1  1  0    1  0  0  0  0  -      0  1  1  0  0  1    0  0  0  0  0  1
0  0  1  0  0  1    0  0  0  0  0  1      0  1  1  0  0  1    0  0  0  1  1  0
0  0  1  0  0  1    0  0  0  0  1  0      1  0  0  0  0  1    0  0  0  0  1  0
0  0  1  0  0  1    0  0  0  1  1  0      1  0  0  0  0  1    0  0  1  1  0  0
0  0  1  1  0  0    0  0  0  0  1  0      1  0  0  0  0  1    0  1  0  0  0  1
0  0  1  1  0  0    1  0  0  0  0  0      1  0  0  1  0  0    0  0  1  0  0  0
0  1  0  0  0  1    0  0  0  0  0  1      1  0  0  1  0  0    1  -  0  0  0  0
0  1  0  0  0  1    0  0  0  1  -  0      1  1  0  0  0  0    0  0  0  1  0  0
0  1  0  0  1  0    0  0  0  0  0  1      1  1  0  0  0  0    0  0  1  0  0  0
0  1  0  0  1  0    0  0  0  1  0  0      1  1  0  0  0  0    0  1  0  0  0  1
0  1  0  0  1  0    1  0  0  0  0  0      1  1  0  0  0  1    0  0  0  1  1  0
0  1  0  0  1  1    0  0  0  0  0  1      1  1  0  0  0  1    0  0  1  1  0  0
0  1  0  0  1  1    0  0  0  1  1  0      1  1  0  0  0  1    0  1  0  0  0  1
0  1  0  0  1  1    1  0  0  1  0  0
```

Figure 17: Implied classes of compatibles $\mathcal{CI}(c, d)$ for the example.

c1	c2	c3	c4	c5	c6	d1	d2	d3	d4	d5	d6
0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	1	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0	1
0	0	1	0	0	1	0	0	0	0	0	0
0	0	1	0	0	1	0	0	0	1	1	0
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	1	1	0
0	1	0	0	1	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	1	1	0
0	1	0	0	1	1	1	0	0	1	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	1	1	0
1	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	1
1	1	0	0	0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	1	1	0
1	1	0	0	0	1	0	0	1	1	0	0

Figure 18: Class sets of compatibles $\mathcal{CCS}(c, d)$ for the example.

c1	c2	c3	c4	c5	c6
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	1	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	0	0	1	1
0	1	1	0	0	0
0	1	1	0	0	1
1	0	0	0	0	0
1	0	0	0	0	1
1	0	0	1	0	0
1	1	0	0	0	0
1	1	0	0	0	1

Figure 19: Prime compatibles $\mathcal{PC}(c)$ for the example.

c1	c2	c3	c4	c5	c6	d1	d2	d3	d4	d5	d6
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	1	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	0	0	0
0	1	0	0	1	1	0	0	0	1	1	0
0	1	0	0	1	1	1	0	0	1	0	0
0	1	1	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	0	0	0
0	1	1	0	0	1	0	0	0	1	1	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	0	1	0	0	0	1
1	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	1
1	1	0	0	0	1	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	1	1	0
1	1	0	0	0	1	0	0	1	1	0	0

Figure 20: Prime compatibles with class sets $\mathcal{PCCS}(c, d)$ for the example.

c1	c2	c3	c4	c5	c6	z1	z2	z3	z4	z5	z6	c̃1	c̃2	c̃3	c̃4	c̃5	c̃6	e
-	-	-	-	-	-	1	0	0	0	0	0	1	0	0	1	0	0	1
-	-	-	-	-	-	0	0	0	1	0	0	1	0	0	1	0	0	1
-	-	-	-	-	-	1	0	0	0	0	0	1	0	0	0	0	0	1
-	-	-	-	-	-	0	0	1	0	0	0	0	0	1	1	0	0	1
-	-	-	-	-	-	0	0	0	1	0	0	0	0	1	1	0	0	1
-	-	-	-	-	-	0	0	0	1	0	0	0	0	0	1	1	0	1
-	-	-	-	-	-	0	0	0	0	1	0	0	0	0	1	1	0	1
-	-	-	-	-	-	1	0	0	0	0	0	1	1	0	0	0	1	1
-	-	-	-	-	-	0	1	0	0	0	0	1	1	0	0	0	1	1
-	-	-	-	-	-	0	0	0	0	0	1	1	1	0	0	0	1	1
-	-	-	-	-	-	1	0	0	0	0	0	1	1	0	0	0	0	1
-	-	-	-	-	-	0	1	0	0	0	0	1	1	0	0	0	0	1
-	-	-	-	-	-	1	0	0	0	0	0	1	0	0	0	0	1	1
-	-	-	-	-	-	0	0	0	0	0	1	1	0	0	0	0	1	1
-	-	-	-	-	-	0	0	0	0	0	1	0	0	0	0	0	1	1
-	-	-	-	-	-	0	1	0	0	0	0	0	1	1	0	0	1	1
-	-	-	-	-	-	0	0	1	0	0	0	0	1	1	0	0	1	1
-	-	-	-	-	-	0	0	0	0	0	1	0	1	1	0	0	1	1
-	-	-	-	-	-	0	1	0	0	0	0	0	1	1	0	0	0	1
-	-	-	-	-	-	0	0	1	0	0	0	0	1	1	0	0	0	1
-	-	-	-	-	-	0	1	0	0	0	0	0	1	0	0	1	1	1
-	-	-	-	-	-	0	0	0	0	1	0	0	1	0	0	1	1	1
-	-	-	-	-	-	0	0	0	0	0	1	0	1	0	0	1	1	1
-	-	-	-	-	-	0	1	0	0	0	0	0	1	0	0	1	0	1
-	-	-	-	-	-	0	0	0	0	1	0	0	1	0	0	1	0	1
-	-	-	-	-	-	0	0	0	0	1	0	0	0	0	0	1	1	1
-	-	-	-	-	-	0	0	0	0	0	1	0	0	0	0	1	1	1

Figure 21: Unate clauses in covering table $UT(c, z, \tilde{c}, e)$ for the example.

c1	c2	c3	c4	c5	c6	z1	z2	z3	z4	z5	z6	c̃1	c̃2	c̃3	c̃4	c̃5	c̃6	e
1	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	0	0	1
0	0	0	1	1	0	1	0	0	0	0	1	1	1	0	0	0	1	1
0	0	0	1	1	0	1	0	0	0	0	1	1	1	0	0	0	0	1
0	0	0	1	1	0	1	0	0	0	0	1	1	0	0	0	0	1	1
1	1	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1
1	1	0	0	0	1	0	0	0	1	1	0	0	0	0	1	1	0	1
1	1	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1
1	1	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	1	1
1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	1	1	1
1	0	0	0	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1
1	0	0	0	0	1	0	1	0	0	0	1	1	1	0	0	0	1	1
1	0	0	0	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1
1	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1
0	1	1	0	0	1	0	0	0	1	1	0	0	0	0	1	1	0	1
0	1	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	0	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1
0	0	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0	0	1
1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	1	0	0	0
0	0	0	1	1	0	1	0	0	0	0	1	0	0	0	1	1	0	0
1	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0	0	1	0
1	1	0	0	0	1	0	0	0	1	1	0	1	1	0	0	0	1	0
1	1	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0
1	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1	0
0	1	1	0	0	1	0	0	0	1	1	0	0	1	1	0	0	1	0
0	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	1	1	0
0	1	0	0	1	1	0	0	0	1	1	0	0	1	0	0	1	1	0
0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	1	0

Figure 22: Binate clauses in covering table $BT(c, z, \tilde{c}, e)$ for the example.