NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementation

TIZIANO VILLA AND ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

***Abstract*—The problem of encoding the states of a synchronous finite state machine (FSM), so that the area of a two-level implementation of the combinational logic is minimized, is addressed. As in previous approaches, the problem is reduced to the solution of the combinatorial optimization problems defined by the translation of the cover obtained by a multiple-valued logic minimization or by a symbolic minimization into a compatible Boolean representation. In this paper we present algorithms for their solution, based on a new theoretical framework that offers advantages over previous approaches to develop effective heuristics. The algorithms are part of NOVA, a program for optimal encoding of control logic. Final areas averaging 20% less than other state assignment programs and 30% less than the best random solutions have been obtained. Literal counts averaging 30% less than the best random solutions have been obtained.**

I. Introduction

The AUTOMATIC synthesis of a sequential circuit as a programmable logic array (PLA)-based finite state machine (FSM) involves functional design, logic design, topological design, and physical design. The step of logic design maps the functional description into a logic representation in terms of logic variables. A representation of the symbolic states (and also of the proper inputs and outputs, if they are symbolic) in terms of Boolean variables, called state assignment, is chosen. The complexity of the combinational component of the FSM depends heavily on the state assignment and selection of memory elements. PLA optimization aims at minimizing the area occupied by the PLA and the delay through it (proportional to the number of product-terms, to a first-order approximation). The PLA area is proportional to the product of the number of rows (product-terms) times the number of columns. The optimum state assignment (or encoding) problem looks for the assignment corresponding to a PLA implementation of minimum area. The (minimum) number of rows is the cardinality of the (minimum) cover of the FSM combinational component according to a given assignment. The number of bits used to represent the states (and the proper inputs and outputs, in case they are symbolic) is related to the number of PLA columns. Therefore, the PLA area depends in a complex way on the state assignment.

Manuscript received June 22, 1989; revised October 9, 1989. This work was supported in part by DARPA under Contract N00039-87-C-0182, by the National Science Foundation under Grant ECS 84-30435, and by grants from MICRO, AMD, AT&T Bell Laboratories, Bell Communications Research, GE, Harris, HP, Hughes, Intel, Microelectronics and Computer Technology, Olivetti, Philips/Signetics, Rockwell, Silicon Computers, and Xerox. This paper was recommended by Editor M. R. Lightner.

This is an expanded version of the work presented at DAC'89.

The authors are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

IEEE Log Number 9036179.

There is a rich early literature on the state assignment problem. Armstrong [3] was the first to formulate the encoding problem as a graph embedding problem, where a graph defines adjacency relations (in terms of Hamming distance) between the codes of the states to be preserved by a subgraph isomorphism on the encoding cube, with the objective to minimize the number of gates of the final implementation. Others [1], [2], [4], [5] proposed algebraic methods based on partition theory and on a reduced dependence criterion. In [6] conditions to find a critical race free encoding of asynchronous sequential machines were reduced to a graph embedding problem. Each input defined a partition of the states (or of a subset of the states) by the successor relation. The states were assigned to vertices in the cube so that for each partition the images of all states in the same block formed a path (using, if necessary, states not also included in the partition or unused vertices, each for at most one block) disjoint from the ones associated to the other blocks. In terms of implementations of minimum area, these approches suffered from a weak connection with the logic optimization steps after the encoding.

Advances in the state assignment problem [7]–[9] have made a key connection with multiple-valued logic minimization: the states of a FSM are represented as the set of possible values for a single multiple-valued variable. Logic minimization is applied on a symbolic representation of the combinational component of the FSM. The effect of multiple-valued logic minimization is to group together the states that are mapped by some input into the same next state and assert the same output. A new combinatorial optimization problem arises (called FACE HYPERCUBE EMBEDDING) of assigning each of these sets (called input constraints) to subcubes of a Boolean k-cube, for a minimum k, in a way that each subcube contains all and only all the codes of the states included in the corresponding constraint. More recently, symbolic minimization [10], [17] has been proposed to take into account the effect of the encoding on the next state part. Symbolic minimization is a technique that yields a minimal encoding-independent sum-of-products representation of a symbolic function. It builds up a directed acyclic graph, where the nodes are the next states and an edge (u, v) corresponds to the covering constraint (called output constraint) that the code of u covers bit-wise the code of v. The translation of the cover obtained by symbolic minimization into a compatible Boolean representation defines simultaneously a face hypercube embedding problem and an output covering problem (called ORDERED FACE HYPERCUBE EMBEDDING).

Ongoing work [16], [18] is focusing on the output encoding problem defined in the optimal state assignment. Output minimization techniques may be seen as setting disjunctive constraints on the codes of the symbolic states (the code of some states is the logical disjunction of the codes of two or more other states). Finding a compatible Boolean representation entails solving a difficult encoding problem based on input, output, and disjunctive constraints.

Other recent approaches [13] rely on local optimization rules defined on a control flowgraph. These rules are expressed as constraints on the codes of the internal variables and an encoding algorithm tries to satisfy most of these constraints.

In this paper we present algorithms for optimal state assignment of FSM's based on the solution of face hypercube embedding and ordered face hypercube embedding. We revisit symbolic minimization and describe an effective version of it. The proposed theoretical framework offers substantial advantages over previous approaches to develop effective algorithms. The algorithms are part of NOVA, a program for optimal encoding of control logic, available as a tool of the Berkeley logic synthesis system [11], [15]. The first three algorithms: *iexact_code, ihybrid_code*, and *igreedy_code*, solve face hypercube embedding. The last algorithm, *iohybrid_code*, solves ordered face hypercube embedding. *iexact_code* is an exact algorithm that finds an encoding satisfying all input constraints and minimizing the encoding length. *ihybrid_code* and *igreedy_code* are heuristic encoding algorithms that maximize input constraint satisfaction, for a (user or default)-given encoding length. *ihybrid_code*, based on a polynomial version of *iexact_code*, yields solutions of high quality and guarantees the satisfaction of all input constraints, for an encoding space large enough. *iohybrid_code* is a heuristic encoding algorithm that maximizes simultaneous input and output constraint satisfaction, according to an appropriately defined metric. It is based on an adaptation of *ihybrid_code* to deal with both input and output constraints.

We present results over a wide range of benchmarks that show that the final areas obtained by the best solution of NOVA average 20% less than those obtained by KISS [9], and 30% less than the best of a number of random state assignments. Final areas obtained by *iohybrid_code* average 30% less than the results reported for Cappuccino/Cream [10]. Although NOVA targets two-level logical implementations, running our examples also through MIS-II, a multilevel logic synthesis system developed at UCB, we found that the final literal counts in a factored form of the logic when encoded by NOVA average 30% less than the literal counts obtained by the best of a number of random state assignments. Comparisons with MUSTANG [12] in the two-level and multilevel case are also reported. Even though NOVA was not designed as a multilevel state assignment program, its performances compare successfully with MUSTANG.

The paper is organized as follows. Algorithms *iexact_code, ihybrid_code, igreedy_code* and *iohybrid_code* are described, respectively, in Sections III–VI. Results on the benchmark examples are presented in Section VII together with final remarks and future work.

II. Preliminaries

In this section, we introduce some background material that is used throughout the paper. The definitions are consistent with [9]–[10], to which we refer for details.

2.1. Encoding Problems in Logic Synthesis

Tabular descriptions of logic functions at the structural level are transformed into Boolean representations by replacing each symbolic entry by Boolean vectors. An assignment of Boolean vectors to symbolic entries is called an *encoding*. The optimization of logic functions performed on the Boolean representation is heavily dependent on the representation of the variables. For instance, the complexity of the combinational component of a FSM depends on the assignment of Boolean variables to the internal states. The following optimal encoding problems may be defined, with respect to a proper cost function:

a) optimal encoding of inputs of a logic function. A problem in class A is the optimal assignment of opcodes for a microprocessor,
b) optimal encoding of outputs of a logic function,
c) optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function.
d) optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function, where the encoding of the inputs (or some inputs) is the same as the encoding of the outputs (or some outputs). Encoding the states of a FSM is a problem in class D since the state variables appear both as input (present state) and output (next state) variables. Another problem in class D is encoding the signals connecting two (or more) combinational circuits.

In this paper we study the problem of the optimal state (and proper input) assignment of FSM's, using as a cost function the area of a two-level implementation. In Sections III–V we approximate the solution of the state assignment problem by modeling it as a problem in class A, i.e., driving the assignment only from information related to the optimal encoding of the present states (and the proper inputs). The algorithms proposed are applicable to any problem in class A. In Section V we model the state assignment problem as a problem in class D, using a scheme of symbolic minimization [10] that captures partially the effect of the next states in case of two-level implementations.

2.2. Multiple-Valued Minimization of FSM's

FSM's can be represented by state transition tables. State transitions tables have as many rows as transitions in the FSM. The rows of the table are divided into four fields corresponding to the primary inputs, present states, next states, and primary outputs of the FSM. Each field is a string of characters. The primary inputs may be in Boolean form or symbolic. We assume that the primary outputs of the FSM are always in Boolean form. Note that the input and output patterns may contain don't care entries. A state transition table defines a symbolic cover of the combinational component of a FSM. The rows of the state transition table are called symbolic implicants of the symbolic cover. The symbolic cover reresentation may be seen as a multiple-valued logic representation, where each present state mnemonic is one of the possible values of a present-state multiple-valued variable. A similar identification holds for the next states (and the proper inputs, if they are symbolic). A multiple-valued logic minimizer can be used to compute a minimal or minimum multiple-valued symbolic cover. The effect of multiple-valued logic minimization is to determine subsets of states that are mapped by some input combination into the same next state and assert the same output. These subsets of states are called input constraints, because they constrain the encoding of the present states (and of the proper inputs, if symbolic) when transforming the symbolic representation into a compatible Boolean representation. The goal of state assignment is to assign each of these subsets of states to subcubes of a Boolean k-cube, for a minimum k, in a way that each subcube contains all and only the codes of the states included in the corresponding constraint. This problem is called face hypercube embedding.

From now on, algorithms for the satisfaction of input constraints are algorithms that can solve any problem in class A, although here we are particularly concerned with FSM encoding approximated as a problem in class A, by means of multiple-valued minimization applied to a symbolic cover of the FSM. Later, in Section VI, we will define other kinds of constraints arising when dealing with FSM encoding solved more generally as a problem in class D. In the next section we describe a theoretical framework and an exact algorithm to solve face hypercube embedding.

III. An Exact Algorithm for Face Hypercube Embedding

In this section we present *iexact_code*, an exact algorithm that finds an encoding satisfying all input constraints and minimizing the encoding length.

3.1 Theoretical Background

Consider the problem FACE HYPERCUBE EMBEDDING: given a collection of subsets of states or symbols

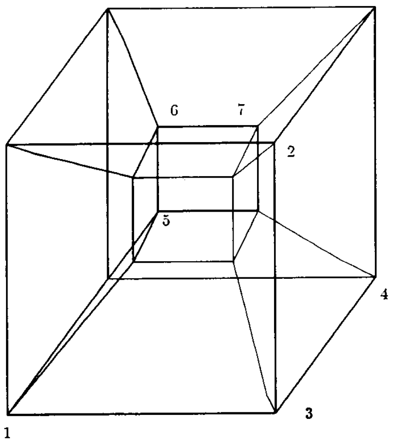


Fig. 1. Encoding of examples 2.1 and 3.1.

(called input constraints and represented as characteristic vectors of the subsets), assign each of these subsets to subcubes (called faces) of a Boolean hypercube of minimum dimension in such a way that each face does not intersect the Boolean vector (called encoding, code or assignment) assigned to any state not contained in the corresponding constraint. Formally it may be stated as the following.

INSTANCE: Set $S = \{1, \cdots, n\}$ and a collection IC of subsets $ic \subseteq S$.

QUESTION: Find the minimum k and an injective map f from the sets $\in IC \cup S$ to B^k, where $B = \{0, 1, x\}$, such that for all subsets $ic \in IC$ and singletons $s \in S$:

$$f(ic) \cap f(s) \neq \Phi \Leftrightarrow s \in ic.$$

Example 3.1.1: Consider $IC = \{1110000, 0111000, 0000111, 1000110, 0000011, 0011000\}$: A solution is $k = 4$ and $f(1110000) = x0x0$, $f(0111000) = 1xx0$, $f(0000111) = x1x1$, $f(1000110) = 0xxx$, $f(0000011) = x111$, $f(0011000) = 1x00$, $f(1000000) = 0000$, $f(0100000) = 1010$, $f(0010000) = 1000$, $f(0001000) = 1100$, $f(0000100) = 0101$, $f(0000010) = 0111$, $f(0000001) = 1111$. Fig. 1 shows the encoding on the 4-cube.

To verify the correctness for $ic = 1110000$, we compute:

$$f(1110000) = x0x0 \cap f(1000000) = 0000 \neq \Phi$$
$$f(1110000) = x0x0 \cap f(0100000) = 1010 \neq \Phi$$
$$f(1110000) = x0x0 \cap f(0010000) = 1000 \neq \Phi$$
$$f(1110000) = x0x0 \cap f(0001000) = 1100 = \Phi$$
$$f(1110000) = x0x0 \cap f(0000100) = 0101 = \Phi$$
$$f(1110000) = x0x0 \cap f(0000010) = 0111 = \Phi$$
$$f(1110000) = x0x0 \cap f(0000001) = 1111 = \Phi.$$

To capture the inclusion relations among the faces of the hypercube, we choose to represent it with its under-

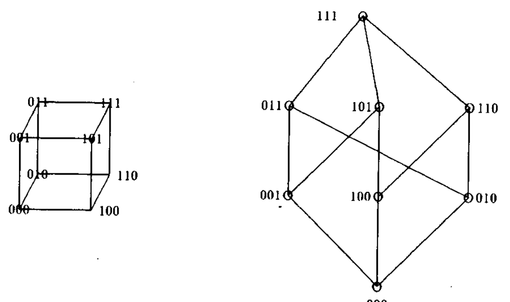


Fig. 2. The 3-cube poset.

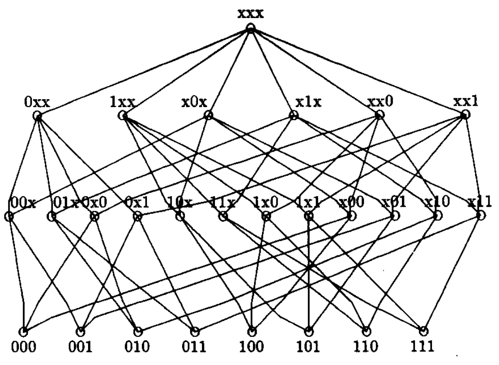


Fig. 3. The 3-cube face-poset.

lying face-poset (partially ordered set), obtained by ordering all faces of all available dimensions according to the Boolean inclusion relation. Formally the n-cube face-poset (or n-face-poset) is the set of all sequences of 0, 1, x (don't care) of length n (called faces). It is a poset with the natural partial ordering defined by $f \leq g$ iff $f(i) \leq g(i)$ for all $i \leq n$ ($0 \leq x$, $1 \leq x$). Notice that the n-cube face-poset is completely different from the n-cube poset, i.e., the poset structure induced on the n-cube by the natural partial ordering on the Hamming codes of the vertices and isomorphic to the poset consisting of all subsets of an n-element set ordered by inclusion. Fig. 2 shows the 3-cube poset and Fig. 3 shows the 3-cube face-poset, drawn as Hasse diagrams, i.e., directed acyclic graphs whose nodes are faces and an arc from node v_i to node v_j denotes inclusion between the corresponding faces. Level of a face, *level*, is the number of x's contained in the sequence. There are $n + 1$ levels in a n-face-poset. Cardinality or dimension of a face is 2^{level}. We define the intersection of faces, with the usual Boolean rules.

Also, the collection of constraint relations of the problem instance can be seen as a poset by ordering them according to the set inclusion relation and can be drawn as an Hasse diagram. We call singleton constraints (or simply singletons) the constraints including exactly one element of S, and universe constraint (or simply the universe) the constraint including all elements of S. We define Closure$_\cap$ $[IC]$:

$$\triangleq S \cup IC \cup \{ic_j: ic_j = ic_{j_1} \cap ic_{j_2}, ic_{j_1}, ic_{j_2} \in IC\}$$

where $\cap$, $\cup$ are assumed in a set theoretical sense.

Example 3.1.2: Consider $IC = \{1110000, 0111000, 0000111, 1000110, 0000011, 0011000\}$. Closure$_\cap$ $[IC]$ $= \{1110000, 0111000, 0000111, 1000110, 0000011, 0011000, 0110000, 0000110, 1000000, 0100000, 0010000, 0001000, 0000100, 0000010, 0000001\}$.

We say that f preserves set theoretic inclusion when

$$ic_1 \supset ic_2 \Leftrightarrow f(ic_1) \supset f(ic_2)$$

and that f preserves set theoretic intersection when

$$ic_1 \cap ic_2 = ic_3 \Leftrightarrow f(ic_1) \cap f(ic_2) = f(ic_3).$$

We denote the cardinality of set S by $\#(S)$, and the cardinality of face $f(S)$ by $\#(f(S))$. Within this setting, the optimization version of FACE HYPERCUBE EMBEDDING can be stated as SUBPOSET DIMENSION.

INSTANCE: Set $S = \{1, \cdots, n\}$ and a collection IC of subsets $ic \subseteq S$.

QUESTION: Find the minimum k and an injection map f from the sets $ic \in$ Closure$_\cap$ $[IC]$ to the faces of the k-cube, satisfying $\#(ic) \leq \#(f(ic))$ and such that the k-cube contains a poset equivalent to the given one, i.e., for all elements $\in$ Closure$_\cap$ $[IC]$, f preserves the set theoretic operations of inclusion and intersection.

The decision version can be stated as SUBPOSET EQUIVALENCE.

INSTANCE: Set $S = \{1, \cdots, n\}$ and a collection IC of subsets $ic \subseteq S$, and a positive integer k.

QUESTION: Does the k-cube contain a poset equivalent to the given one, i.e., is there an injective map f from the sets $ic \in$ Closure$_\cap$ $[IC]$ to the faces of the k-cube, satisfying $\#(ic) \leq \#(f(ic))$ and such that, for all elements $\in$ Closure$_\cap$ $[IC]$, f preserves the set theoretic operations of inclusion and intersection?

Figs. 4 and 5 show an example of this decision problem reduced to subgraph isomorphism. A restriction to subgraph homeomorphism is shown by Figs. 6 and 7. A general example is shown by Figs. 8 and 9. We notice that this decision problem can always be seen as a special instance of subgraph isomorphism into the transitive closure of the directed acyclic graph induced by the k-cube face-poset.

3.2. Processing a Problem Instance of SUBPOSET DIMENSION

The input exact encoding algorithm has a preprocessing stage that, given the set of input constraints IC, builds a representation of the closure of the poset generated by IC. We call this closure, augmented by the universe, the input poset of the problem. The input poset can be represented

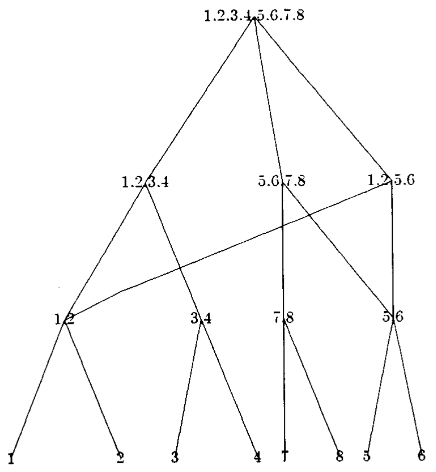


Fig. 4. Completely leveled input poset.

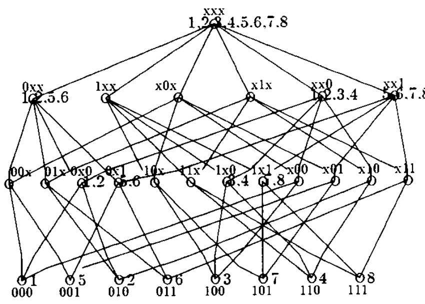


Fig. 7. 3-face poset with subgraph homeomorphic to poset of Fig. 6.

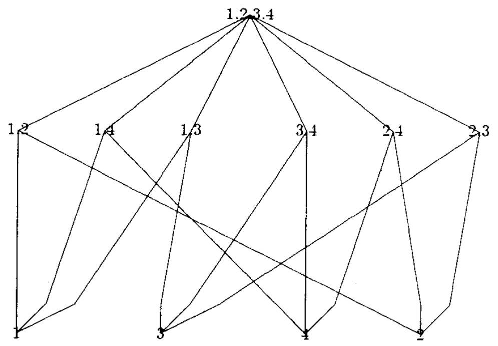


Fig. 8. Generic input poset.

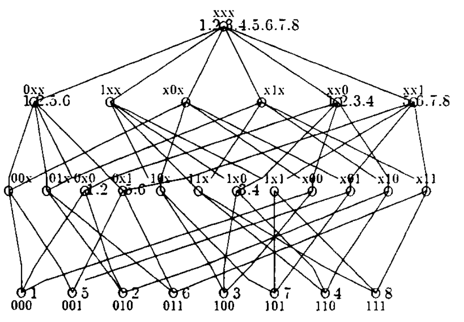


Fig. 5. 3-face poset with subgraph isomorphic to poset of Fig. 4.

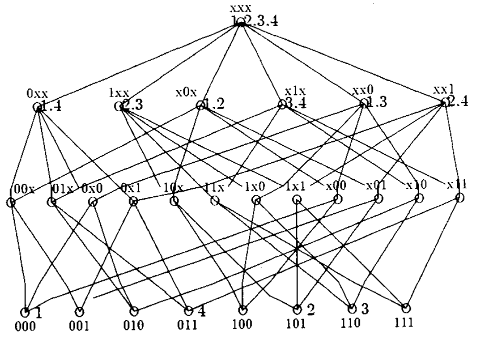


Fig. 9. 3-face poset with subposet equivalent to poset of Fig. 8.

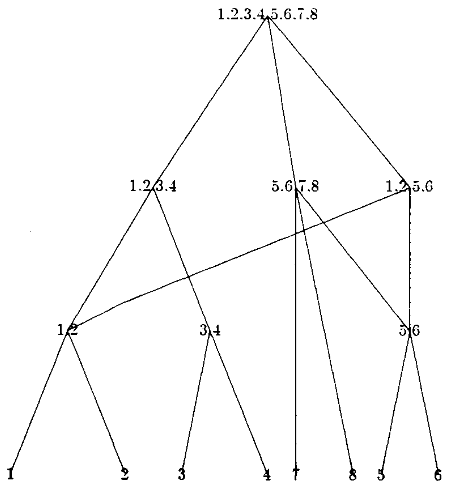


Fig. 6. Partially leveled input poset.

by an Hasse diagram, or by a more compact representation, that we call input graph, $IG(V, E)$. To every constraint ic in the input poset (nodes of IG), we associate the set of his fathers $F(ic)$ and the set of his children $C(ic)$, where $F(ic)$ is the set of minimal constraints that include ic, and $C(ic)$ is the set of maximal constraints included in ic (edges of IG). The relations of fathers and children, are a succinct representation of the Hasse diagram arcs, instead of the arcs between all possible comparable constraints. We walk through the input poset from a constraint to another, through the edges of IG.

Example 3.2.1: Consider IC = {1110000, 0111000, 0000111, 1000110, 0000011, 0011000}. V = {1111111, 1110000, 0111000, 0000111, 1000110, 0000011, 0011000, 0110000, 0000110, 1000000, 0100000, 0010000, 0001000, 0000100, 0000010, 0000001}.

To compute E, it is sufficient to determine $F(ic)$ for all $ic \in V$. The pairs (ic, ic_i), for all $ic_i \in F(ic)$, represent all the ingoing edges of *ic*:

$$\begin{aligned}F(1111111) &= \Phi,\ F(1110000) = F(0111000)\\ &= F(0000111) = F(1000110) = 1111111,\\ F(0011000) &= 0111000,\ F(0110000)\\ &= (0111000,\ 1110000),\ F(0000011)\\ &= 0000111,\\ F(0000110) &= (0000111,\ 1000110),\ F(0010000)\\ &= (0011000,\ 0110000),\\ F(0001000) &= 0011000,\ F(0100000)\\ &= 0110000,\ F(0000010)\\ &= (0000011,\ 0000110),\\ F(0000001) &= 0000011,\ F(0000100)\\ &= 0000110,\ F(1000000)\\ &= (1110000,\ 1000110).\end{aligned}$$

Fig. 10 shows the input graph, *IG*, associated to the input constraints *IC*.

3.3. General Scheme of iexact_code

The input exact encoding algorithm, *iexact_code*, finds an answer to SUBPOSET DIMENSION, by answering exactly SUBPOSET EQUIVALENCE for increasing dimensions of the hypercube. Solving exactly SUBPOSET EQUIVALENCE decides whether the input poset can be embedded in a given *k*-face-poset, and finds a satisfactory assignment, *ENC*, if any. If it exists for dimension *k*, and we already answered "no" for dimensions $< k$, we have an answer to SUBPOSET DIMENSION. The procedure *mincube_dim* computes a lower bound on the dimension of the *k*-face-poset. The procedure *subposet_equivalence* decides SUBPOSET EQUIVALENCE. The pseudocode below illustrates the general scheme of the algorithm.

```
iexact_code(IG)
{
    mincube = minimum feasible cube dimension
    ENC = $\Phi$
    mincube = mincube_dim(IG)
    for ($k$ = mincube; $k \le \#(S)$; $k++$) {
        ENC = subposet_equivalence(IG, k)
        if (ENC $\ne \Phi$) return(ENC)
        }
    }
}.
```

3.3.1 Refined Scheme of iexact_code: The general scheme of Section III-3.3 can be refined, replacing a single call to *subposet_equivalence* with a number of calls to a procedure, *pos_equiv*, that decides a restriction of SUBPOSET EQUIVALENCE with fixed $\#(f(ic))$, for all $ic \in IG$ such that $F(ic)$ is the universe (the constraint including all elements of S). We recall that an assignment of constraint *ic* of the input poset to face $f(ic)$ of the face-poset satisfies $\#(ic) \le \#(f(ic))$. For a given dimension of the hypercube on which we invoke *pos_equiv*, in many cases a solution exists only when the previous inequality is proper for one or more of the constraints. One reason may be that some constraints need to be adjacent to many others and so they require a large face, i.e., a large boundary, to satisfy their numerous intersections. So, for a cube of dimension *k*, we have for every constraint *ic* the choice of different *levels* of the corresponding face, and the choice of different faces of a given *level*.

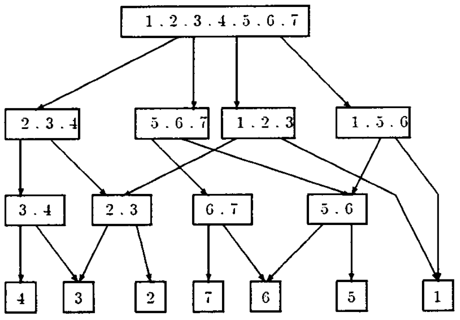


Fig. 10. Input Graph *IG* (example 2.2.1).

Constraint $ic \in IG$ is classified of category 1 and called a primary constraint if $\#(F(ic)) = 1$ and $F(ic))$ is the universe; of category 2 if $\#(F(ic)) > 1$; of category 3 if $\#(F(ic)) = 1$ and $F(ic)$ is not the universe. We use the notation $cat(ic) = i$ to denote that *ic* has category *i*. In general if $\#(ic) = c$ and $cat(ic) = 1$ then $\#(f(ic))$ is such that $\log c \le level \le k - 1$, where $2^{level} = \#(f(ic))$. Notice that if $cat(ic) = 2$, then $\#(f(ic)) = \#(\cap f(ic_j))$ for all $ic_j \in F(ic)$. If $cat(ic) = 3$, then $\#(f(ic)) < \#(f(F(ic)))$.

Example 3.3.1.1: Consider the input graph *IG* given in Example 3.2.1:
$cat(1110000) = cat(0111000) = cat(0000111) = cat(1000110) = 1$, $cat(0000110) = cat(0110000) = cat(0010000) = cat(0000010) = cat(1000000) = 2$, $cat(0011000) = cat(0000011) = cat(0001000) = cat(0100000) = cat(0000001) = cat(0000100) = 3$.

The current chosen *levels* of primary constraints are stored in a vector *dimvect*, called primary level vector. The routine *face_levels* returns at every call a new primary level vector. *face_levels* is invoked when *pos_equiv* answers no to the decision problem on (*IG*, *k*) restricted to *dimvect*. The primary level vectors are generated in increasing lexicographic order. If, for a given embedding dimension *k*, all possible primary level vectors have been unsuccessfully tried, the main routine updates the hypercube dimension to $k + 1$. Notice that $\#(S)$ is a trivial upper bound on the hypercube dimension.

Example 3.3.1.2.: Continues from Example 3.3.1.1. Given the set of ordered primary constraints (1110000, 0111000, 0000111, 1000110), and cube dimension $k =$

4, the successive values of the primary level vector *dimvect* are

(2, 2, 2, 2), (2, 2, 2, 3), (2, 2, 3, 2), (2, 2, 3, 3), (2, 3, 2, 2), (2, 3, 2, 3), (2, 3, 3, 2), (2, 3, 3, 3), (3, 2, 2, 2), (3, 2, 2, 3), (3, 2, 3, 2), (3, 2, 3, 3), (3, 3, 2, 2), (3, 3, 2, 3), (3, 3, 3, 2), (3, 3, 3, 3).

Since *pos_equiv*(*IG*, 4, (2, 2, 2, 2)) returns a valid encoding, in this example only the first vector (2, 2, 2, 2) is actually generated.

It is still an open problem how to decide *a priori* that some values of the primary level vector are useless to obtain a positive answer to *pos_equiv*. A solution to it would strongly speed up the running time of *iexact_code*. The following pseudocode illustrates the flow of the algorithm.

```
iexact_code(IG)
{
  mincube = minimum feasible cube dimension
  dimvect = vector of face levels
  ENC = $\Phi$
  mincube = mincube_dim(IG)
  for ($k$ = mincube; $k \leq \#(S)$; $k++$) {
    dimvect = face_levels($k$, dimvect)
    while (dimvect $\neq \Phi$) {
        ENC = pos_equiv(IG, $k$, dimvect)
        if (ENC $\neq \Phi$) return(ENC)
        else dimvect = face_levels($k$, dimvect)
    }
  }
}.
```

3.3.2. Lower Bounds on the Cube Dimension: To save some useless calls to *pos_equiv*, one needs to find good lower bounds on the hypercube dimension, as starting points of the *for* outer cycle of *iexact_code*. The routine *mincube_dim* implements some counting arguments and returns *mincube*, the initial cube dimension passed to *pos_equiv*. Some counting arguments are explained in the following.

```
mincub_dim(IG)
{
  min_cube = count_cond1(IG)
  min_cube = count_cond2(IG, min_cube)
  min_cube = count_cond3(IG, min_cube)
}.
```

3.3.2.1. Two straightforward counting arguments: A first counting argument, (*count_cond1*), enforces the obvious condition that the cube should have at least as many faces of a given cardinality as the input graph has constraints of a given cardinality.

A second counting argument, (*count_cond2*), enforces the condition that, in a feasible cube, the face assigned to any constraint must have as least as many minimal including faces (i.e., faces of the least larger *level*), as many fathers the constraint has. The condition is checked in the stronger case that the face is of the minimum feasible *level*, because it cannot hold for larger *levels*, if it does not hold for the minimum one. Analytically, we look for the smallest cube dimension such that the following inequalities are satisfied: $\#(F(ic)) \leq \#(S) - level(f(ic))$ and $\#(f(ic)) = \#(ic)$.

3.3.2.2. A counting argument based on uneven constraints: A third counting argument, (*count_cond3*), enforces the condition that, in a feasible cube of dimension *min_cube*, in case some constraints have a cardinality that is not a power of 2 (called uneven constraints), there will be enough faces of *level* 0 to accommodate them, in the hypothesis of the densest possible packing of the uneven constraints. We can think of the uneven constraints as adding new virtual states to S. The question is: what is the minimum number of virtual states introduced by a given set of uneven constraints? Two facts must be taken into account.

1) If $\#(ic) = c$, then ic introduces $minpow2(c) - c$ virtual states, where *minpow*2 is the minimum power of $2 \geq c$.

2) At most *min_cube* constraints may intersect in the same virtual state.

They are used in the algorithm illustrated by the pseudocode given in the following. We keep an array of the # of virtual states introduced by each uneven constraint (fact 1), and we compute the maximum number of identifications between the virtual states introduced by different uneven constraints (using fact 2). This is the densest possible packing of the uneven constraints. Notice that such a packing could be unfeasible, among other reasons, when the uneven constraints, completed by the virtual vertices, generate new uneven constraints, and all level 0 faces of the hypercube are already used.

```
count_cond3(IG, min_cube)
{
    entry $u$ of array VRT stores the # of virtual
    states introduced by uneven constraint $ic_u$
    for (each uneven $ic_u \in IG$) {
       VRT[$u$] = minpow2($ic_u$) $-$ #($ic_u$)
    }
    sort VRT in nondecreasing order
    while (a feasible dimension is found) {
     iter_count = 0
     while (VRT $\neq$ zero vector) {
        decrease by 1 the first min_cube nonzero en-
        tries of VRT in nondecreasing order
        iter_count is increased by 1
     }
     if ($2^{min\_cube} - \#(S) <$ iter_count) {
       min_cube is increased by 1
     }
     if ($2^{min\_cube} - \#(S) \geq$ iter_count) {
       return(min_cube)
       }
    }
}.
```

Example 3.3.2.2.1: Consider the graph $IG(V, E)$ given in Example 3.2.1, where $V = \{1111111, 1110000, 0111000, 0000111, 1000110, 0000011, 0011000, 0110000, 0000110, 1000000, 0100000, 0010000, 0001000, 0000100, 0000010, 0000001\}$.

The routines *count_cond1* and *count_cond2* return *min_cube* = 3; *count_cond3* returns *min_cube* = 4, because of two virtual states introduced by the uneven constraints 1110000, 0111000, 0000111, 1000110. Therefore, 4 is the initial cube dimension passed to *pos_equiv*.

3.4. Solving Restricted Subposet Equivalence

The solution to subposet equivalence, restricted to a given cube dimension and primary level vector, is based on a backtracking scheme. Basic operations of it are selecting the nodes (constraints) of the input graph (*next_to_code*), and assigning them to faces of the cube, so to preserve intersection and inclusion relations in the two posets (*assign_face*). The selected nodes are inserted in a list *Sic*, ordered by selection time. Notice that *next_to_code* selects only constraints *ic* such that $cat(ic) = 1$ or $cat(ic) = 3$, (i.e., $\#(F(ic)) = 1$). We recall that the codes of the constraints *ic* such that $cat(ic) = 2$ are determined by the codes assigned to theirs fathers, and therefore, they do not need to be selected at this level. A selected constraint is given a code consistent with *ENC*, if any. In this way, an assignment, *ENC*, is built incrementally and when it cannot be consistently extended to a new constraint, because of previous wrong choices, an old constraint-face map is undone (*next_to_recode*), and a new one, consistent with the reduced *ENC*, is attempted. While a backtracking phase starts or continues, *next_to_recode* chooses the last constraint of *Sic*, among those successfully encoded. While a backtracking phase ends, *next_to_recode* chooses the first constraint in *Sic*, among those whose assignment was undone. In case no feasible assignment exists, *pos_equiv* returns an empty encoding. The pseudocode that follows illustrates the general scheme of the procedure.

```
pos_equiv(IG, k, dim_vect)
{
   ENC = $\Phi$
   Sic = $\Phi$
   backtrack Boolean flag to signal backtracking
   next_to_code(IG, Sic) returns unencoded $ic \in V$
   first selected constraint ic is stored as fic
   Sic = Sic $\cup$ ic
   while(there are unencoded constraints) {
        backtrack set to FALSE
        assign_face(ic, ENC, k, dim_vect) returns
        $f(ic) \neq \Phi$ iff ic can be encoded consistently
          with ENC
        ENC = ENC $\cup$ f(ic)
        while ($f(ic) = \Phi$ or backtrack is TRUE) {
          if($f(ic) = \Phi$ & backtrack is FALSE) {
            a backtracking phase starts
            backtrack set to TRUE
            if (ic coincides with fic) {
                  no feasible assignment exists
                  return($\Phi$)
            } else {
                  next_to_recode(IG, Sic) returns $ic \in$ Sic
                  ENC = ENC $-$ f(ic)
                  assign_face(ic, ENC, k, dim_vect)
                  returns f(ic)
                  ENC = ENC $\cup$ f(ic)
            }
          }
          if ($f(ic) = \Phi$ & backtrack is TRUE) {
            the current backtracking phase continues
            if (ic coincides with fic) {
                  no feasible assignment exists
                  return($\Phi$)
            } else {
                  next_to_recode(IG, Sic) returns ic
                  $\in$ Sic
                  ENC = ENC $-$ f(ic)
                  assign_face(ic, ENC, k, dim-
                  _vect) returns f(ic)
                  ENC = ENC $\cup$ f(ic)
            }
          }
          if ($f(ic) \neq \Phi$ & backtrack is TRUE) {
                a backtracking phase ends
                next_to_recode(IG, Sic) returns $ic \in$
                Sic
                assign_face(ic, ENC, k, dim_vect)
                returns f(ic)
                ENC = ENC $\cup$ f(ic)
                if (ic coincides with lic) {
                   backtrack set to FALSE
                }
          }
        }
        next_to_code(IG, Sic) returns unencoded $ic \in$
        V
        last selected constraint ic is stored as lic
        Sic = Sic $\cup$ ic
   }
   return(ENC)
}.
```

3.4.1. Walking Through the Input Graph: The routine *next_to_code* selects an unencoded constraint according to the following priority branching scheme. Recall that *lic* is the last constraint inserted in *Sic*.

1) Choose, if any, a constraint of category 1 not already coded, mappable to a face of the same level as $f(lic)$, and sharing children with it.

2) Choose, if any, a constraint of category 1 not already coded, mappable to a face of the same *level* as $f(lic)$.

3) Choose, if any, a constraint not already coded, mappable to a face of the same *level* as $f(lic)$, and sharing children with it.

4) Choose, if any, a constraint not already coded, mappable to a face of the same *level* as $f(lic)$.

5) Choose, if any, a constraint of category 1 not already coded, mappable to a face of the maximum *level* less than $f(lic)$s.

6) Choose, if any, a constraint not already coded, mappable to a face of the maximum *level* less than $f(lic)$s.

The selection mechanism chooses the constraints in order of decreasing feasible face *level*, and within it gives higher priority to constraints of cardinality 1, and among them to those sharing children with constraints already coded. The rationale is that we want to first code the constraints needing larger faces, and among them those of category 1, and that we exploit a look-ahead of one level (sharing of children) to reject encodings at an upper level, if they are unable to satisfy face intersections at the next lower level. This allows us to discover at an early time when an assignment is unfeasible, i.e., cannot be extended downwards.

3.4.2. Walking Through the Face-Poset: The routine *assign_face* walks through the face-poset and assigns faces to constraints, if feasible. Faces are generated calling the routine *genface*, and verified for consistency with the incrementally built encoding *ENC* by the routine *verify*. A face is accepted if it also the case that, for all the children of category 2 of the constraint being encoded, the intersection of the faces assigned to their encoded (if any) fathers is a feasible code for them. The set of children of category 2 of *ic*, with some fathers already encoded, is denoted by $D(ic)$. When unable to map the constraint to a face, *assign_face* returns the empty face. The routine *genface* is invoked for constraints of category 1 and 3. The generation of the faces is based on the production of all combinations of patterns of x's, according to the *level* of the face, in lexicographic order.

```
assign_face(ic, ENC, k, dim_vect)
{
 genface(ic, k, dim_vect) returns $f(ic)$
 while ($f(ic) \neq \Phi$) {
   if (verify($f(ic)$, ENC) succeeds) {
      for (all constraints $ic_d \in D(ic)$) {
         if ($\cap f(ic_j)$, $ic_j \in F(ic_d)$, $f(ic_j) \neq \Phi$ w.d.
           & verify($\cap f(ic_j)$, ENC) succeeds) {
           $ic_d$ is mapped to $\cap f(ic_j)$
           } else break from the for cycle
           return($f(ic)$)
      }
   }
   genface(ic, k, dim_vect) returns $f(ic)$
 }
 return($\Phi$)
}.
```

Example 3.4.2.1: Consider the input graph *IG* given in Example 3.2.1. The procedure *pos_equiv(IG*, 4, (2, 2, 2, 2)) flows as follows.

Step 1:

next_to_code returns 0111000
after *assign_face:* $f(0111000) = x0x0$.

Step 2:

next_to_code returns 1110000 (branch n. 1)
after *assign_face:* $f(1110000) = x00x, f(0110000) = x000$.

Step 3:

next_to_code returns 1000110 (branch n. 1)
face $0xx0$ is generated and rejected
after assign_*face:* $f(1000110) = 0xx1, f(1000000) = 0001$.

Step 4:

next_to_code returns 0000111 (branch n. 1)
faces $0x1x$, $1x0x$, $1x1x$, $0x0x$, $00xx$ are generated and rejected
after *assign_face:* $f(0000111) = 01xx, f(0000110) = 01x1$.

Step 5:

next_to_code returns 0011000 (branch n. 6)
after *assign_face:* $f(0011000) = 00x0, f(0010000) = 0000$.

Step 6:

next_to_code returns 0000011 (branch n. 4)
after *assign_face:* $f(0000011) = 010x, f(0000010) = 0101$.

Step 7:

next_to_code returns 0000001 (branch n. 6)
after *assign_face:* $f(0000001) = 0100$.

Step 8:

next_to_code returns 0000100 (branch n. 4)
face 0101 is generated and rejected
after *assign_face:* $f(0000100) = 0111$.

Step 9:

next_to_code returns 00010000 (branch n. 4)
face 0000 is generated and rejected
after *assign_face:* $f(0001000) = 0010$.

Step 10:

next_to_code returns 0100000 (branch n. 4)
face 0000 is generated and rejected
after *assign_face:* $f(0100000) = 1000$.
ENC:

$$f(1111111) = xxxx$$

$$f(0111000) = x0x0$$

$$f(0000111) = 01xx$$
$$f(1110000) = x00x$$
$$f(1000110) = 0xx1$$
$$f(0000110) = 01x1$$
$$f(0110000) = x000$$
$$f(0011000) = 00x0$$
$$f(0000011) = 010x$$
$$f(0000001) = 0100$$
$$f(0000010) = 0101$$
$$f(0000100) = 0111$$
$$f(0001000) = 0010$$
$$f(0010000) = 0000$$
$$f(0100000) = 1000$$
$$f(1000000) = 0001.$$

3.4.3. Correctness: The correctness of the assignment is guaranteed incrementally. We suppose that up to the ith step we built a correct partial assignment, i.e., an assignment to a subset of constraints that verifies the subposet equivalence among the constraints already taken into consideration. Coding a new constraint, we want to make sure that we still get a correct assignment with respect to the enlarged set of encoded constraints and of inclusion/intersection relations holding among them. The verification on the input poset consists of the following checks: 1) if the new constraint has only one father, the latter's face must include the face proposed for the former (inclusion condition $ic_i \supset ic_j \rightarrow f(ic_i) \supset f(ic_j)$); 2) if the new constraint has more than one father, the faces assigned to the fathers must intersect in the face proposed for the child (intersection condition $ic_i \cap ic_j = ic_k \rightarrow f(ic_i) \cap f(ic_j) = f(ic_k)$). On the input poset we limit the checks to the fathers of the constraints being encoded, because we build the global assignment function incrementally fathers first, children after, and so we need to worry only about the local fathers/children relations.

The verification on the face-poset consists of the following checks: 1) the face proposed for the constraint being encoded must be different from the faces already assigned (the mapping has to be injective); 2a) if an assigned face includes properly the face proposed, the former's inverse must be a father of the constraint being encoded; 2b) if the face proposed includes properly a face already assigned, the latter's inverse has a to be a child of the constraint being encoded (both verify the inclusion condition $f(ic_i) \supset f(ic_j) \rightarrow ic_i \supset ic_j$); 3) if an assigned face has a nonempty intersection with the face proposed, their inverses must intersect in a nonempty constraint (intersection condition $f(ic_i) \cap f(ic_j) = f(ic_k) \rightarrow ic_i \cap ic_j = ic_k$). On the face-poset the checks are global, because a new proposed face may *a priori* lay anyway in it. Inductively, we can say that we always guarantee a correct partial assignment, so when we are able to extend it to the complete input poset, we have a correct solution of the problem.

3.5. Complexity

Two quantities measure the complexity of *iexact_code:* the number of upper level backtracking trials, *#Ulb*, and, for each of them, the number of lower level backtracking attempted assignments of faces to constraints, *#Llb*. We evaluate first *#Ulb*. Suppose that from the set of input constraints we have the following subset of constraints of category 1: $\{ic_i\}$, $i = 1, \cdots, n$. Call d_i the minimum dimension of a face that can be assigned to ic_i and d the current encoding length. In the worst case

$$\#Ulb = \prod_{i=1}^{i=n} (d - d_i).$$

If $d - d_i \approx (d/2)$, we have

$$\#Ulb \approx \left(\frac{d}{2}\right)^n.$$

Now we evaluate *#Llb*. We have for every constraint of category 1 the choice of many possible faces of minimum dimension. Constraints of category 2 and 3 are encoded, respectively, within the subspace assigned to their father and by the intersection of the codes assigned to their fathers and so their contribution can be neglected. Keeping the same notation as before, and noting that one can assign to ic_i at most $2^{d_i}\binom{d}{d_i}$ faces, in the worst case:

$$\#Llb = \prod_{i=1}^{i=n} 2^{d_i}\binom{d}{d_i}.$$

If $\Sigma\, d_i \approx (nd/2)$, we have

$$\#Llb \approx (2d)^{nd/2}.$$

The algorithm *iexact_code* can be computationally too expensive and it is not suggested to be the standard way of solving face hypercube embedding. Nevertheless, it allowed us to find solutions to the majority of the examples of our benchmark, producing a set of results against which to compare heuristic solutions. Moreover, as we will see in Section IV, a computationally bounded version of it, *semiexact_code* is the core of a very efficient approximate algorithm, *ihybrid_code*.

IV. A Hybrid Algorithm for Face Hypercube Embedding

In this section we describe an approximate algorithm, called *ihybrid_code*, that operates on the input constraints. The inputs to the algorithm are: *#bits*, a user-specified code-length and *IC*, the set of weighted input constraints (the weight of an input constraint is proportional to the number of repeated occurrences of the corresponding product term in the multiple-valued minimized cover). The algorithm outputs *ENC*, an encoding that maximizes heuristically the total sum of constraint weights satisfied in the given code-length. The rationale is that the weight of a constraint is proportional to the number of product terms saved in a final implementation by satisfying it. The strategy to choose the subset of satisfiable constraints is greedy, i.e., constraints are chosen one at a time in decreasing order of weight and they are accepted or rejected if they can be satisfied together with the subset of constraints already chosen. We do not try to find the set of constraints that give the minimum product-term cardinality for a given code-length, trading-off speed versus quality of solution. Since a new constraint is accepted or rejected by nonexhaustive simulation of a partial encoding, the greedy strategy favors the cluster of constraints that yield the largest saving of product terms in the final implementation. Satisfactory experimental results support this conclusion.

The algorithm is based on two encoding strategies, *semiexact_code* and *project_code*, the first one invoked on the minimum code-length and the second one on the successive code-lengths up to *#bits*. The routine *semiexact_code* is a modified version of the exact input encoding algorithm *iexact_code*, presented in Section III. The routine *project_code* is a quick encoding algorithm that eventually guarantees a complete satisfaction of all the input constraints.

The procedure *ihybrid_code* builds incrementally *SIC*, a set of satisfied input constraints and *RIC*, a set of unsatisfied input constraints. Both *SIC* and *RIC* are empty at the beginning. In the first part *ihybrid_code* tries to maximize the total sum of constraint weights satisfied in the minimum code-length. To do so it selects *ic*, the constraint of maximum weight from $IC - SIC - RIC$ and invokes *semiexact_code* on the set of constraints $SIC \cup ic$. If *semiexact_code* succeeds in satisfying all constraints of $SIC \cup ic$, *ENC* gets updated to the new found encoding and *ic* is added to the set of satisfied constraints, otherwise *ic* is added to the set of rejected constraints. This loop is repeated until the sets *SIC* and *RIC* become a partition of *IC*. If *semiexact_code* fails always, as it may happen in rare pathological situations, *ENC* gets a random encoding to guarantee that there is always a starting encoding for *project_code* to work properly. If *RIC* is not empty and *#bits* is larger than the minimum code-length, the embedding cube is increased by unitary steps and for each increase the encoding strategy *project_code* is invoked. For each dimension added to the cube, *project_code* is guaranteed to satisfy at least one more constraint from *RIC*, while still satisfying all constraints of *SIC*. So *project_code* adds at least (in general more than) one constraint to *SIC* and deletes it from *RIC*; *ENC* gets updated to the new found encoding. This is repeated until there are no more unsatisfied constraints and there is no more unused encoding space. At the end *SIC* and *RIC* are a new partition of *IC*, and they contain, respectively, the satisfied and unsatisfied constraints of *IC*. It is fairly obvious that *project_code*, if given an encoding space large enough, is guaranteed to satisfy completely any set of input constraints. The pseudocode that follows illustrates the steps of the algorithm.

```
ihybrid_code(IC, #bits)
{
    cube_dim = minimum encoding length
    $SIC = \Phi$
    $RIC = \Phi$
    while ( $(IC - SIC - RIC) \neq \Phi$ ) {
        selects ic, the constraint of maximum weight in
        $(IC - SIC - RIC)$
        if semiexact_code($SIC \cup ic$, cube_dim) succeeds {
            ENC gets a new encoding
            $SIC = SIC \cup ic$
            }
        else $RIC = RIC \cup ic$
    }
  if ($ENC = \Phi$) ENC gets random encoding
  while ( $RIC \neq \Phi$ and cube_dim < #bits ) {
        cube_dim is increased by 1
        project_code(ENC, SIC, RIC, cube_dim)
        i.e. ENC gets a new encoding
          NC = { new constraints satisfied by project_code }
          $SIC = SIC \cup NC$
          $RIC = RIC - NC$
    }
}.
```

Example 4.1: Consider $IC = \{1110000, 0111000, 0000111, 1000110, 0000011, 0011000\}$. The weights of the constraints are, respectively, 4, 2, 3, 5, 1, 1. A constraint has a 1 in the *i*th position if the *i*th state belongs to it. The bounded backtracking coding algorithm flows as follows.

Step 1:

$ic = 1000110$; *semiexact_code* satisfies the constraints in $SIC \cup ic$;
$SIC = \{ 1000110 \}$; $RIC = \Phi$.

Step 2:

$ic = 1110000$; *semiexact_code* satisfies the constraints in $SIC \cup ic$;
$SIC = \{ 1000110, 1110000 \}$; $RIC = \Phi$.

Step 3:

$ic = 0000111$; *semiexact_code* fails to satisfy the constraints in $SIC \cup ic$;
$SIC = \{ 1000110, 1110000 \}$; $RIC = \{ 0000111 \}$.

Step 4:

$ic = 0111000$; *semiexact_code* fails to satisfy the constraints in $SIC \cup ic$;
$SIC = \{ 1000110, 1110000 \}$; $RIC = \{ 0000111, 0111000 \}$.

Step 5:

ic = 0000011; *semiexact_code* satisfies the constraints in *SIC* ∪ *ic*;
SIC = { 1000110, 1110000, 0000011 }; *RIC* = { 0000111, 0111000 }.

Step 6:

ic = 0011000; *semiexact_code* satisfies the constraints in *SIC* ∪ *ic*;
SIC = { 1000110, 1110000, 0000011, 0011000 }; *RIC* = { 0000111, 0111000 };
ENC = { 000, 101, 100, 110, 010, 011, 111 }.

The projection coding algorithm raises the codes of states 5, 6, 7 into the added fourth dimension and so it is able to satisfy, in one last step, both constraints left in *RIC*.

Step 7:

NC = { 0000111, 0111000 };
SIC = { 1000110, 1110000, 0000111, 0111000, 0000011, 0011000 }; *RIC* = Φ;
ENC = { 0000, 1010, 1000, 1100, 0101, 0111, 1111 }.

Fig. 1 shows *ENC* computed after Step 7.

4.1. The Bounded Backtrack Coding Algorithm

The routine *semiexact_code* is a modified version of the exact input encoding routine *iexact_code*, presented in Section III. In *iexact_code* there are two sources of combinatorial explosion.

1) At the upper level backtracking, for a fixed embedding dimension, we have for every constraint of category 1 the choice of many possible cardinalities of the face to which it can be assigned.

2) At the lower level backtracking, we have for every constraint of category 1 the choice of many possible faces of minimum sufficient dimension. Constraints of category 2 and 3 are encoded, respectively, within the subspace assigned to their father and by the intersection of the codes assigned to their fathers and so their contribution to the cost of the lower level backtracking can be neglected.

The routine *semiexact_code* copes with the previous problems as follows.

1) A modified version of the routine *faces_dim_set* is used, which assigns to the constraints of category 1 only faces of the minimum possible dimension. It is well known that with such a limitation an existing solution is not guaranteed to be found, but we want to cut on the explosive number of possible face configurations assignable to constraints of category 1. The rationale of the choice is that in a cube whose dimension is the minimum encoding length, most constraints can only obtain the minimum face dimension, so it is not a dramatic loss of optimality to limit ourselves as was done.

2) At the lower level backtracking, we stop the search when the number of partial encoding assignments already tried surpasses a fixed number, *max_work*, set conventionally in the program. This magic number has been determined by an average analysis of the complexity of the lower level backtracking mechanism on the set of data of our benchmark. A useful improvement would be to have the program adapt this parameter to the current input instance.

4.2. The Projection Coding Algorithm

After *semiexact_code* has completely partitioned the constraints in the set of those satisfied and the set of those unsatisfied, the routine *project_code* is invoked until there are no more unsatisfied constraints and there is no more unused encoding space. At each call, *project_code* enlarges by one the dimension of the encoding space and produces *ENC*, an encoding that satisfies at least one more constraint from the set of constraints left unsatisfied by *semiexact_code*, while it still satisfies all constraints already satisfied. At the end of each call *SIC* and *RIC* are updated. The routine *project_code* is well defined because of the following proposition.

Proposition 4.2.1: Given an encoding of length l that satisfies a set of constraints C, there always exists an encoding of length $l + 1$ that satisfies all constraints of C, and moreover satisfies at least another arbitrary constraint not in C.

Proof: By construction. Suppose we have an encoding satisfying C and consider any constraint c not in C. Increase the codes of the states by 1 b, according to the following rule. If a state belongs to c, pad its code with a 1, otherwise pad it with a 0. In the first case we say that the state has been raised in the $(l + 1)$-st dimension. This moves the states belonging to c onto the added l-dimensional cube, where they occupy exactly the same positions that they had in the original l-dimensional cube. The constraint c becomes surely satisfied, because in the worst case it can span the whole added l-dimensional cube. Every constraint in C is still satisfied. Indeed constraints in C involving only states assigned all together either to the original l-dimensional cube or to the added one, remain the same as before (in the respective l-dimensional cubes). Constraints in C involving states distributed between the two l-dimensional cubes are still satisfied because no spurious constraint could be in the face they span, unless it was already intersecting it when restricted to the first l dimensions, against the hypothesis. Q.E.D.

Proposition 4.2.1 guarantees that one can always satisfy at least one more constraint. In the actual implementation of *project_code* we try to minimize the number of states that need to be raised in order to satisfy a given constraint. One implemented heuristic is to raise first the states that appear more often in the unsatisfied constraints. This makes it more likely that more than one constraint will be satisfied in the enlarged encoding space. The constraints are selected in *decreasing* order of weight, as it was the case with *semiexact_code*. It is fairly obvious that *project_code*, if given an encoding space large enough, is guaranteed to satisfy completely any set of input constraints, at least at the pace of one more constraint for each unitary increase of the dimension. We emphasize that *project_code* plays the role of a quick encoding routine that guarantees eventually a complete satisfaction of all the input constraints, while *semiexact_code* is a more expensive computational step, aimed to find an optimal encoding (heuristically speaking) on the minimum encoding length. We chose to concentrate the computing efforts on the minimum encoding length, because it is where we can get the maximum benefit in terms of final area. The combination of *semiexact_code* and *project_code* compares very favorably with the exact solution in terms of minimum encoding length needed to satisfy all constraints. This measure is off the optimum by 10% on the benchmark reported in Section VII.

4.3. Complexity

The computational complexity of *ihybrid_code* is linear in the number of constraints, but we point out that the linearity in *semiexact_code* depends on a hidden constant proportional to the magic number *max_work* which bounds the semi-exact search. The running times of *hybrid_code* on the benchmark reported in Section VII range from a few seconds on most examples to 1310 s for a very complex example on a VAX 11/8650.

V. A Greedy Algorithm for Face Hypercube Embedding

In NOVA, it is implemented also *igreedy_code*—an approximate algorithm for input constraint satisfaction (details are in [20]). It tries heuristically to satisfy as many constraints as it can, for a given code-length. The code-length may be specified by the user, otherwise the minimum one is assumed by default. Since we kept it simple and very fast, *igreedy_code* is especially tailored for short code-lengths (close to the minimum one). Indeed its heuristic encoding strategy does not undo previous suboptimal choices, and so it may leave unused some potentially useful encoding space, even when it is made available by the encoding length. The algorithm computes all the intersections of the input constraints and starts the encoding going upwards from the deepest of them. So doing, it gives priority to the satisfaction of common subconstraints, i.e., proper subsets of at least two input constraints. This is a suboptimal strategy, but an effective one. It is equivalent to replacing the original constrained embedding problem by an easier one, by simplifying the information produced by the multiple-valued logic minimization.

The running times of *igreedy_code* are not more than a few seconds of VAX 11/8650 even in the most complex examples where other encoding algorithms fail to complete.

VI. Encoding Based on Symbolic Minimization

In this section we present an encoding algorithm based on symbolic minimization [10], [17], a technique that yields a minimal encoding-independent sum-of-products representation of a symbolic function. The minimal symbolic representation has to then be encoded into a compatible Boolean representation. This is achieved by satisfying associated sets of input constraints and output covering relations. The symbolic minimization algorithm builds up a directed acyclic graph where a node is a next state and an edge (u, v) corresponds to the covering constraint that u bit-wise covers v, i.e., the Boolean code assigned to u must be 1 where the Boolean code assigned to v is 1, and in at least one position the codes of u and v are 1 and 0, respectively. These covering relations are called output constraints (OC). They are related to a companion set of input constraints (IC) that we get from the final cover obtained from the symbolic minimization procedure. In Section VI-6.1 we present a modified version of the symbolic minimization algorithm, and in Section VI-6.2 we present *iohybrid_code*, an algorithm to satisfy both input and output constraints.

6.1. Symbolic Minimization Revisited

We use the same definitions and notations as given in [10], to which we refer for a full-fledged description. The symbolic minimization algorithm builds up a directed acyclic graph where an edge (u, v) corresponds to the covering constraint that u bit-wise covers v. Our version of symbolic minimization differs in two respects from the symbolic minimization loop in presence of binary outputs described in [10]. We refer to the pseudocode that follows for the discussion. We assume that the input cover C is the result of a disjoint minimization step and that the symbolic cover does not have any unspecified next states. The cardinality of a set S is denoted by $\#(S)$. The first modification is that in the minimization of step 7 we carry a complete description of the binary outputs, by explicitly putting in the don't care-set of the i next state all the product-terms of C not already committed to its on-set or off-set. This ensures that both the on and off conditions for the binary outputs are taken into account in any stage of the minimization process, in the same way as they will be in the final compatible encoded Boolean representation. The second modification is that in step 9 we accept the covering relations of the ith minimization stage only when the minimization of step 7 decreases the cardinality of the on-set of next state i. The reason is that we want to exclude from G output covering relations that do not contribute to the decrease of the final cover cardinality. In this way the successive encoding problem is also eased.

```
1. Finite State Machine cover $C$ with $q$ next states,
   optional binary outputs,
   empty weighted acyclic graph $G$
     (edge $(i, j)$ with weight $w$ is denoted $(i, j, w)$ )
   and empty cover $FinalP$
   Output is the graph $G$ and the minimal cover $FinalP$
2. $On_k$ = on-set implicants of the $k$th next state
   with the corresponding binary outputs unchanged
3. Repeat Steps 4 through 9 $q$ times
```

```
4. $i$ = select a symbol
5. $Dc_i = \cup On_j$,
   for all $j$ for which there is no path from vertex $i$
   to vertex $j$ in $G$
6. $Off_i = \cup ON_j$,
   for all $j$ for which there is a path from vertex $i$
   to vertex $j$ in $G$
7. $MB_i$ = minimize($On_i$, $Dc_i$, $Off_i$)
8. $M_i$ = implicants of $MB_i$
   that are in the on-set of next state $i$
9. if ( $\#(M_i) < \#(On_i)$ ) {
      $w_i = \#(On_i) - \#(M_i)$
      $G = G \cup \{ (j, i, w_i)$ such that $M_i$ intersects $On_j$
      }
        $P = P \cup MB_i$
   }
   else $P = P \cup On_i$
10. $FinalP$ = minimize($P$).
```

6.2. Algorithms for Satisfaction of Input and Output Constraints

The symbolic minimization algorithm builds up a directed acyclic graph defining output constraints (OC) on the set of next states. They are associated to a companion set of input constraints (IC) that we get from the final cover *FinalP*. Indeed, *FinalP* is a multiple-valued logic cover and its translation to a compatible Boolean representation defines a face hypercube embedding problem. Therefore, to obtain a Boolean representation of *FinalP* of the same cardinality, we need to satisfy simultaneously the pair of input and output constraints sets (IC, OC), that we define as an ordered face hypercube embedding problem. Notice that, in general, the input constraints obtained by the symbolic minimization procedure are different from those obtained by a multiple-valued output-disjoint minimization. Any variation of the symbolic minimization procedure, e.g., in the selection of step 4, determines a different pair (IC, OC). Output-disjoint minimization is a special case where $OC = \Phi$. Ordered face hypercube embedding is a very hard combinatorial problem. One cannot guarantee the unconditional existence of a solution to it, and not much is known about the conditions of existence of an encoding that satisfies a pair (IC, OC). When a solution does not exist, the problem arises of which constraints to relax.

6.2.1. An Algorithm Biased Toward Satisfaction of Input Constraints: The encoding algorithm that we implemented in *iohybrid_code* is an adaptation of the encoding technique described in Section IV. There are three stages to it. The first two are invoked on the minimum code-length and the third one on the successive code-lengths up to *#bits*. In the first stage, we try to satisfy as many constraints from IC as possible, by a cycle of calls to *semiexact_code* as done in *ihybrid_code* and so we get the encoding *ENC* satisfying the input constraints of *SIC*. In the second stage, we look for an encoding that maximizes heuristically the total weight of the clusters of output constraints satisfied and still satisfies *SIC*. Let i vary in I, the set of indexes of the next states. A cluster, OC_i, is defined as the set of edges of G going into the next state i. The set of output constraints OC can be seen as partitioned in clusters: $OC = \cup OC_i$, where i varies on the number of next states. Each OC_i has associated a weight w_i. Since a gain of w_i product-terms is achieved only by satisfying all output constraints of OC_i, we try to satisfy an increasing collection, *SOC*, of sets of clusters selected greedily in decreasing order of weight. The encoding is attempted by the routine *io_semiexact_code*, which succeeds when it finds an encoding satisfying the constraints of *SIC* and $SOC \cup OC_i$, in which case OC_i is added to *SOC*. The routine *io_semiexact_code* is a modified version of *semiexact_code*, with an added mechanism of rejecting assignments of faces to states if some active output covering relation is violated. The third stage is a cycle of calls to *project_code* as described in Section IV, which guarantees the eventual satisfaction of all input constraints. Notice that in the unusual case that $IC = \Phi$, i.e., there are only output constraints, an algorithm specialized in output constraint satisfaction, *out_encoder*, is invoked. We refer for *out_encoder* to [14]. As a summary, our encoding strategy gives higher priority to input constraints over output constraints. The pseudocode that follows illustrates the steps of the algorithm.

```
iohybrid_code(IC, OC, #bits)
{
   cube_dim = minimum encoding length
   $SIC = \Phi$
   $RIC = \Phi$
   $SOC = \Phi$
   if ( $IC = \Phi$ ) {
      out_encoder(OC)
      return
   }
   while ( ($IC - SIC - RIC$) $\neq \Phi$ ) {
      selects ic, the constraint of maximum weight in
      ($IC - SIC - RIC$)
      if semiexact_code($SIC \cup ic$, cube_dim) succeeds {
            ENC gets a new encoding
            $SIC = SIC \cup ic$
      }
      else $RIC = RIC \cup ic$
   }
   all $OC_i$ are labeled unused
   while ( there are unused $OC_i$ ) {
      selects unused $OC_i$ of maximum weight
      if io_semiexact_code(SIC, $SOC \cup OC_i$,
      cube_dim) succeeds {
            ENC gets a new encoding
            $SOC = SOC \cup OC_i$
      }
      $OC_i$ is labeled used
   }
   if ($ENC = \Phi$) ENC gets a random encoding
```

```
    while ( $RIC \neq \Phi$ and $cube\_dim < \#bits$ ) {
        $cube\_dim$ is increased by 1
        $project\_code(ENC, SIC, RIC, cube\_dim)$
        i.e., $ENC$ gets a new encoding
            $NC$ = { new constraints satisfied by $project\_code$ }
            $SIC = SIC \cup NC$
            $RIC = RIC - NC$
    }
}.
```

6.2.2. An Algorithm Based on Clusters of Input and Output Constraints: We noticed already that the set of output constraints OC can be seen as partitioned in clusters: $OC = \cup OC_i$, where i varies on the number of next states and each OC_i has associated a weight w_i. It is true that also IC can be seen as clustered, although in this case the clusters are not a partition. Precisely, each OC_i has a companion set IC_i of input constraints associated to next state i in *FinalP* (some input constraints are not associated to any next state, but they are related to proper outputs and are denoted here by IC_o). To achieve a gain of w_i product-terms, it is necessary not only to satisfy the output constraints of OC_i, but also the associated input constraints of IC_i. The algorithm *iohybrid_code* disregards this fact, putting higher priority on the satisfaction of the input constraints, independently from the eventual satisfaction of the companion output constraints. We devised also an algorithmic variant, *iovariant_code*, where the ith call to *io_semiexact_code* succeeds only if both IC_i and OC_i happen to be satisfied. The constraints in IC_o are dealt with at the beginning by a cycle of calls to *semiexact_code*. It turns out that *iohybrid_code* has a better performance than *iovariant_code*. One can argue that it is in general more profitable to satisfy as many input constraints as possible, because the output covering relations are a weak way of modeling the effects of output encoding. Satisfying as many input constraints as possible may lead to convenient product-terms sharing between the different output functions. The following pseudocode illustrates the steps of the algorithm.

```
iovariant_code($IC$, $OC$, $\#bits$)
{
    $cube\_dim$ = minimum encoding length
    $SIC = \Phi$
    $RIC = \Phi$
    $SOC = \Phi$
    if ( $IC = \Phi$ ) {
        $out\_encoder(OC)$
        return
    }
    while ( $(IC_o - SIC - RIC) \neq \Phi$ ) {
        selects $ic$, the constraint of maximum weight in $(IC_o - SIC - RIC)$
        if $semiexact\_code(SIC \cup ic, cube\_dim)$ succeeds {
            $ENC$ gets a new encoding
            $SIC = SIC \cup ic$
        }
        else $RIC = RIC \cup ic$
    }
    all $OC_i$ are labeled unused
    while ( there are unused $OC_i$ ) {
        selects unused $OC_i$ of maximum weight
        $IC_i = IC_i - SIC$
        if $io\_semiexact\_code(SIC \cup IC_i, SOC \cup OC_i, cube\_dim)$ succeeds {
            $ENC$ gets a new encoding
            $SIC = SIC \cup IC_i$
            $SOC = SOC \cup OC_i$
            $RIC = RIC - IC_i$
            }
            else $RIC = RIC \cup IC_i$
            $OC_i$ is labeled used
    }
    if ($ENC = \Phi$) $ENC$ gets a random encoding
    while ( $RIC \neq \Phi$ and $cube\_dim < \#bits$ ) {
        $cube\_dim$ is increased by 1
        $project\_code(ENC, SIC, RIC, cube\_dim)$
        i.e. $ENC$ gets a new encoding
            $NC$ = { new constraints satisfied by $project\_code$ }
            $SIC = SIC \cup NC$
            $RIC = RIC - NC$
    }
}.
```

Example 6.2.2.1: Consider the following clustered sets of input and output constraints (IC_i; OC_i; w_i) for $I = 1, \cdots, 8$. Notice that $i > j$ means that state i must cover state j; input constraints and weights are interpreted as usual.

$$(IC_o;\ w_o) = (01010101;\ 1)$$

$$(IC_1;\ OC_1;\ w_1) = (\Phi;\ 2 > 1, 3 > 1, 4 > 1, 5 > 1, 6 > 1, 7 > 1, 8 > 1;\ 4)$$

$$(IC_2;\ OC_2;\ w_2) = (00110000;\ 6 > 2;\ 1)$$

$$(IC_3;\ OC_3;\ w_3) = (00001100;\ 7 > 3;\ 2)$$

$$(IC_4;\ OC_4;\ w_4) = (00000011;\ 8 > 4;\ 1)$$

$$(IC_5;\ OC_5;\ w_5) = (\Phi;\ 6 > 5, 7 > 5, 8 > 5;\ 1)$$

$$(IC_6;\ OC_6;\ w_6) = (00110000;\ \Phi;\ 3)$$

$$(IC_7;\ OC_7;\ w_7) = (00001100;\ \Phi;\ 1)$$

$$(IC_8;\ OC_8;\ w_8) = (00000011;\ \Phi;\ 1).$$

A solution ($\#bits = 3$) to such encoding problem, as found by *iohybrid_code* and *iovariant_code*, is ENC = (000, 010, 100, 110, 001, 011, 101, 111).

6.3. Complexity

The computational complexity of *ihybrid_code* is linear in the number of input constraints and next states. The warning of Section IV-4.3 also holds here.

TABLE I
STATISTICS OF BENCHMARK EXAMPLES

EXAMPLE	#inp	#out	#states
dk14	8*	5	7
dk15	8*	5	4
dk16	4*	3	27
dk17	4*	3	8
dk27	2*	2	7
dk512	2*	3	15
ex1	9	19	20
ex2	2	2	19
ex3	2	2	10
ex5	2	2	o
ex6	5	8	8
bbara	4	2	10
bbsse	7	7	16
bbtas	2	2	5
beecount	3	4	7
cse	7	7	16
donfile	2	1	24
iofsm	5	6	10
keyb	7	2	19
mark1	5	16	15
physrec	8	15	10
planet	7	19	48
s1	8	6	20
sand	11	9	32
scf	27	56	121
scud	7	6	8
shiftreg	1	1	8
styr	9	10	30
tbk	6	3	32
train11	2	1	11

* : # of symbolic inputs

TABLE II
COMPARISONS OF *iexact*, *ihybrid*, *igreedy*

EXAMPLE	*iexact*			*ihybrid*			*igreedy*			*1-hot+*
	#bits	#cubes	area	#bits	#cubes	area	#bits	#cubes	area	#cubes
dk14	8*	22	550	6*	26	520	6*	26	520	24
dk15	6*	16	320	5*	17	289	5*	20	340	17
dk16	9*	49	1372	7*	54	1188	7*	68	1496	55
dk17	6*	17	323	5*	17	272	5*	18	288	20
dk27	4*	8	104	4*	8	104	4*	7	91	10
dk512	6*	17	340	5*	18	306	5*	17	289	21
ex1	7	40	2320	6	40	2200	5	46	2392	44
ex2	6	28	372	5	27	567	5	31	651	38
ex3	5	17	357	4	18	324	4	17	306	21
ex5	5	15	315	4	14	252	4	17	306	19
ex6	4	23	690	3	25	675	3	25	675	23
bbara	5	24	600	4	24	528	4	25	550	34
bbsse	6	27	1053	5	27	972	4	29	957	30
bbtas	3	8	120	3	8	120	3	10	150	16
beecount	4	11	242	3	12	228	3	10	190	12
cse	5	44	1584	4	46	1518	4	45	1485	55
donfile	11**	23	874	5	28	560	5	41	820	24
iofsm	4	16	448	4	16	448	4	16	448	19
keyb	7	47	1739	5	48	1488	5	55	1705	77
mark1	5	18	738	4	18	684	4	17	646	19
physrec	4	33	1419	4	33	1419	4	34	1462	38
planet	6	87	4437	6	87	4437	6	86	4386	92
s1	5	80	2960	5	80	2960	5	81	2997	92
sand	6	89	4361	5	97	4462	5	99	4554	114
scf	8**	-	-	8	138	18492	7	143	18733	151
scud	6	71	2698	3	71	2059	4	62	1984	86
shiftreg	3	4	48	3	4	48	3	8	96	9
styr	6	89	4094	5	94	4042	5	97	4171	111
tbk	-***	-	-	5	147	4410	5	173	5190	173
train11	5	9	180	4	9	153	4	11	187	11

* : encoding of inputs and states
** : best known solution, not guaranteed the best, iexact could not complete the search
*** : iexact could not complete the search
#bits : code-length
#cubes : Number of product-terms after espresso logic minimization
area : (2*(#inputs + #bits) + #bits + #outputs) * #cubes
+ : upper bound of 1-hot encoding

VII. RESULTS AND FUTURE WORK

We have run more than 50 benchmark examples (which have been obtained from various university and industrial sources and include the MCNC benchmark set) representing a wide range of finite state automata on different state assignment programs as well on our two algorithms. The size statistics of 30 significant examples (including all the largest ones) are given in Table I. The product-term cardinality of the 1-hot encoding is given under the column 1-hot in Table II.

Tables II–IV summarize the results obtained running the algorithms of NOVA, the program KISS, and random state assignments. The results were obtained running ESPRESSO-MV in order to obtain the input constraints and our symbolic minimizer built on top of ESPRESSO-MV to obtain the mixed input/output constraints, by running NOVA to encode the states and the symbolic inputs (if any), and by running ESPRESSO again to obtain the final area of the encoded FSM. The areas under random assignments are the best and the average of a statistical average of a number of different (number of states of the FSM plus the number of symbolic inputs, if any) random state assignments on each example. The final areas obtained by the best solution of NOVA average 20% less than those obtained by KISS, and 30% less than the best of a number of random state assignments.

Tables VIII and IX show plots summarizing the most important data of Tables II–IV. On the x-axis the 30 examples of Table I are ordered by increasing number of states, on the y-axis ratios of the areas of different algorithms over the best results of NOVA are plotted. The examples, tabulated by increasing number of states, are *dk15, bbtas, beecount, dk14, dk27, dk17, ex6, scud, shiftreg, ex5, bbara, ex3, iofsm, physrec, train11, dk512, mark1, bbsse, cse, ex2, keyb, ex1, s1, donfile, dk16, styr, sand, tbk, planet,* and *scf.*

TABLE III
COMPARISONS OF *ihybrid/igreedy* WITH KISS AND RANDOM

EXAMPLE	*ihybrid/igreedy***			KISS			RANDOM	
	#bits	#cubes	area	#bits	#cubes	area	area+	area++
dk14	6*	26	520	9*	24	550	720	809
dk15	5*	17	289	6*	17	391	357	376
dk16	7*	54	1188	12*	55	2035	1826	1994
dk17	5*	17	272	6*	19	361	320	368
dk27	4*	7	91	4*	9	117	143	143
dk512	5*	17	289	7*	18	414	374	418
ex1	6	40	2200	7	42	2436	3120	3317
ex2	5	27	567	6	31	744	798	912
ex3	4	17	306	6	18	432	342	387
ex5	4	14	252	5	15	315	324	358
ex6	3	25	675	5	24	792	810	850
bbara	4	24	528	5	26	650	616	649
bbsse	4	29	957	6	27	1053	1089	1144
bbtas	3	8	120	3	13	195	165	215
beecount	3	10	190	4	11	242	285	293
cse	4	45	1485	6	45	1756	1947	2087
donfile	5	28	560	12	24	984	1200	1360
iofsm	4	16	448	4	16	448	560	579
keyb	5	48	1488	8	47	1880	3069	3416
mark1	4	17	646	5	19	779	760	782
physrec	4	33	1419	5	34	1564	1677	1741
planet	6	86	4386	6	89	4539	4896	5249
s1	5	80	2960	5	81	2997	3441	3733
sand	6^	89	4361	6	95	4655	4278	4933
scf	8	138	18492	8	140	18760	19650	21278
scud	4	62	1984	6	71	2698	2262	2533
shiftreg	3	4	48	3	6	72	132	132
styr	5	94	4042	6	91	4186	5031	5591
tbk	5	147	4410	-$	-	-	5040	6114
train11	4	9	153	6	10	230	221	241
TOTAL			55326			na	65453	72002
%			84			na	100	110

* : encoding of inputs and states
** : solution of minimum area between ihybrid and igreedy
^ : solution found by iexact
+ : best random solution; ++ : average of random solutions
$: not completed due to too many symbolic implicants
#bits : code-length; #cubes : Number of product-terms after espresso logic minimization
area : (2*(#inputs + #bits) + #bits + #outputs) * #cubes

TABLE IV
COMPARISONS OF *iohybrid*, *ihybrid/igreedy*, BEST OF NOVA WITH RANDOM

EXAMPLE	*iohybrid*			*ihybrid/igreedy***			NOVA***			RANDOM	
	#bits	#cubes	area	#bits	#cubes	area	#bits	#cubes	area	area+	area++
dk14	6*	25	500	6*	26	520	6*	25	500	720	809
dk15	5*	17	289	5*	17	289	5*	17	289	357	376
dk16	7*	57	1254	7*	54	1188	7*	54	1188	1826	1994
dk17	5*	19	304	5*	17	272	5*	17	272	320	368
dk27	4*	8	104	4*	7	91	4*	7	91	143	143
dk512	5*	20	340	5*	17	289	5*	17	289	374	418
ex1	6	37	2035	6	40	2200	6	37	2035	3120	3317
ex2	5	35	735	5	27	567	5	27	567	798	912
ex3	4	18	324	4	17	306	4	17	306	342	387
ex5	4	15	270	4	14	252	4	14	252	324	358
ex6	3	25	675	3	25	675	3	25	675	810	850
bbara	4	26	572	4	24	528	4	24	528	616	649
bbsse	5	28	1008	4	29	957	4	29	957	1089	1144
bbtas	3	10	150	3	8	120	3	8	120	165	215
beecount	3	11	209	3	10	190	3	10	190	285	293
cse	4	45	1485	4	45	1485	4	45	1485	1947	2087
donfile	5	42	840	5	28	560	5	28	560	1200	1360
iofsm	4	15	420	4	16	448	4	15	420	560	579
keyb	5	48	1488	5	48	1488	5	48	1488	3069	3416
mark1	4	19	722	4	17	646	4	17	646	760	782
physrec	4	34	1462	4	33	1419	4	33	1419	1677	1741
planet	6	94	4794	6	86	4386	6	86	4386	4896	5249
s1	5	63	2331	5	80	2960	5	63	2331	3441	3733
sand	5	96	4416	6^	89	4361	6	89	4361	4278	4933
scf	7	137	17947	8	138	18492	7	137	17947	19650	21278
scud	3	62	1798	4	62	1984	3	62	1798	2262	2533
shiftreg	3	4	48	3	4	48	3	4	48	132	132
styr	5	95	4058	5	94	4042	5	94	4042	5031	5591
tbk	5	57	1710	5	147	4410	5	57	1710	5040	6114
train11	4	10	170	4	9	153	4	9	153	221	241
TOTAL			**52458**			**55326**			**51053**	**65453**	**72002**
%			**80**			**84**			**77**	**100**	**110**

* : encoding of inputs and states
** : solution of minimum area between ihybrid and igreedy
*** : solution of minimum area between iohybrid and ihybrid/igreedy
^ : solution found by iexact
+ : best random solution; ++ : average of random solutions
#bits : code-length; #cubes : Number of product-terms after espresso logic minimization
area : (2*(#inputs + #bits) + #bits + #outputs) * #cubes

Table V shows that the final areas obtained running the algorithm *iohybrid_code* (symbolic minimization followed by ordered face hypercube embedding) average 30% less than the data reported for Cappuccino/Cream. Table VI reports statistics of the algorithm *ihybrid_code*.

Table VII reports the number of literals after running through the standard Boolean optimization script in the multilevel logic synthesis system MIS-II with encodings obtained by NOVA, MUSTANG, and random state assignments. In the case of NOVA only the best minimum code-length two-level result was given to MIS-II. MUSTANG was run with -p, -n, -pt, -nt options, and minimum code-length. The final literal counts in a factored form of the logic encoded by NOVA average 30% less than the literal counts of the best of a number of random state assignments. The best (minimum code-length) two-level results of MUSTANG with -p, -n, -pt, -nt options versus the best (minimum code-length) two-level results of NOVA are also reported. Notice that in the case of MUSTANG the run that achieved the minimum number of cubes is not necessarily the same that achieved the minimum number of literals. In the case of NOVA we fed into MIS-II only the best two-level result, so the data reported refer to the same minimized cover.

MUSTANG heuristically maximizes the number and size (fan-in and fan-out oriented algorithms, respectively) of common cubes in the encoded network to minimize the number of literals in the resulting combinational logic network after multilevel logic optimization. No tradeoff is made between the fan-in and fan-out oriented algorithms. Even though NOVA was not designed as a multilevel state-assignment program, its performances compare successfully with MUSTANG. Table X shows a plot summarizing the data of Table VII. On the *x*-axis the 30 examples of Table I are ordered by increasing number of states (as in Tables VIII and IX), on the *y*-axis the ratios for the cubes and the literals of MUSTANG over NOVA are plotted. The plot shows that a state assignment that

TABLE V
COMPARISONS OF *iohybrid* WITH CAPPUCCINO/CREAM

EXAMPLE	*iohybrid*			Cappuccino/Cream		
	#bits	#cubes	area	#bits	#cubes	area
bbtas	3	10	150	4	11	198
cse	4	45	1485	8	49	2205
lion	2	6	66	2	6	66
lion9	4	9	153	5	10	200
modulo12	4	11	165	7	17	408
planet	6	94	4794	10	89	5607
s1	5	63	2331	7	68	2924
sand	5	96	4416	9	107	6206
shiftreg	3	4	48	4	14	210
styr	5	95	4058	12	103	6592
tav	2	11	198	3	11	231
train11	4	10	270	6	10	230
dol	3	9	126	4	8	136
dk14	3	25	500	5	23	598
dk15	2	17	289	4	15	345
dk16	5	57	1254	11	49	1960
dk17	3	19	304	4	17	323
dk27	3	8	104	3	9	126
dk512	4	20	340	7	22	572
TOTAL			**20951**			**29137**
%			**71**			**100**

#bits : code-length
#cubes : Number of product-terms after espresso logic minimization
area : (2*(#inputs + #bits) + #bits + #outputs) * #cubes

TABLE VI
STATISTICS OF *ihybrid*

EXAMPLE	wsat	wunsat	clength	ex-clength	time
dk14	15	4	5	4	1.58
dk15	9	1	3	3	0.35
dk16	25	8	10	7	311
dk17	8	1	4	4	0.5
dk27	5	0	3	3	0.28
dk512	8	2	6	5	35
ex1	8	3	7	7	35
ex2	7	1	6	6	2.46
ex3	5	1	5	5	0.53
ex5	4	3	6	5	3.95
ex6	7	2	5	4	0.9
bbara	2	2	5	5	0.81
bbsse	3	2	6	6	0.91
bbtas	1	0	3	3	0.18
beecount	4	2	4	4	0.45
cse	8	4	5	5	16
donfile	8	16	15	<=11	555
iofsm	2	0	4	4	0.28
keyb	26	7	7	7	28
mark1	3	1	5	5	29
physrec	8	0	4	4	0.66
planet	12	0	6	6	37.6
s1	14	0	5	5	2.33
sand	6	1	6	6	39.71
scf	11	3	9	<=8	917.56
scud	15	14	8	6	5.2
shiftreg	9	0	3	3	0.35
styr	14	4	9	6	52
tbk	44	54	?	?	1310
train11	10	1	5	5	3.46

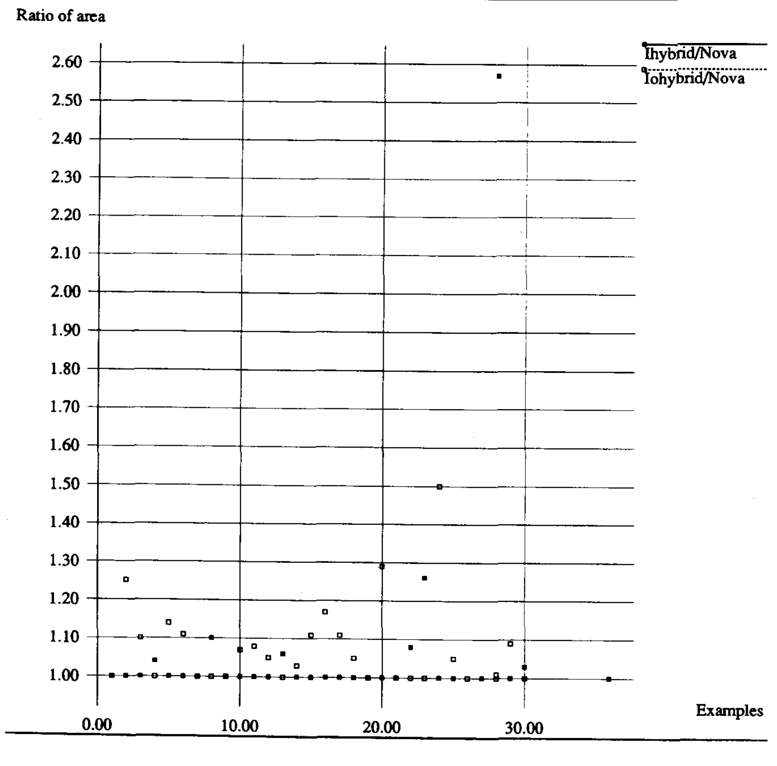
wsat : weighted sum of satisfied constraints in minimum code-length
wunsat : weighted sum of unsatisfied constraints in minimum code-length
clength : encoding length of ihybrid to satisfy all constraints
ex-clength : exact encoding length to satisfy all constraints
time : cpu-seconds on a VAX 11/8650

TABLE VII
COMPARISONS OF TWO-LEVEL AND MULTILEVEL LOGIC IMPLEMENTATIONS OBTAINED BY MUSTANG AND NOVA

EXAMPLE	MUSTANG+	NOVA++	MUSTANG+++	NOVA++++	RANDOM@
	#cubes	#cubes	#lit	#lit	#lit
dk14x	32	26	117	98	164
dk15x	19	17	69	65	73
dk16x	71	52	259	246	402
ex1	55	44	280	215	313
ex2	36	27	119	96	162
ex3	19	17	71	76	83
bbara	25	24	64	61	84
bbsse	31	29	106	132	149
bbtas	10	8	25	21	31
beecount	12	10	45	40	59
cse	48	45	206	190	274
donfile	49	28	160	88	193
keyb	58	48	167	200	256
mark1	19	17	76	86	116
physrec	37	33	159	150	178
planet	97	86	544	560	576
s1	69	63	183	265	444
sand	108	96	535	533	462
scf	148	137	791	839	890
scud	83	62	286	182	222
shiftreg	4	4	2	0	16
styr	112	94	546	511	591
tbk	136	57	547	289	625
train11	10	9	37	43	44
TOTAL	1288	1033	5394	4986	6407
%	124	100	108	100	130

#cubes : number of product-terms after espresso logic minimization, with minimum code-length
#lit : number of literals after multi-level logic optimization with MIS-II, standard script
+ : best (minimum code-length) two-level result of MUSTANG with -p, -n, -pt, -nt options
++ : best (minimum code-length) two-level result of NOVA
+++ : best multi-level result of MUSTANG with -p, -n, -pt, -nt options
++++ : multi-level result starting from best (minimum code-length) two-level result of NOVA
@ : best (in terms of area) of a number of different random state assignments

TABLE VIII
SUMMARY OF NOVA



gives a good two-level implementation also gives a good multilevel implementation. This is consistent with the experiments reported in [19]. We expect that real wins in state assignment for multilevel implementations will be achieved by programs detecting multicube common factors (kernels).

TABLE IX
KISS, BEST RANDOM, AND NOVA

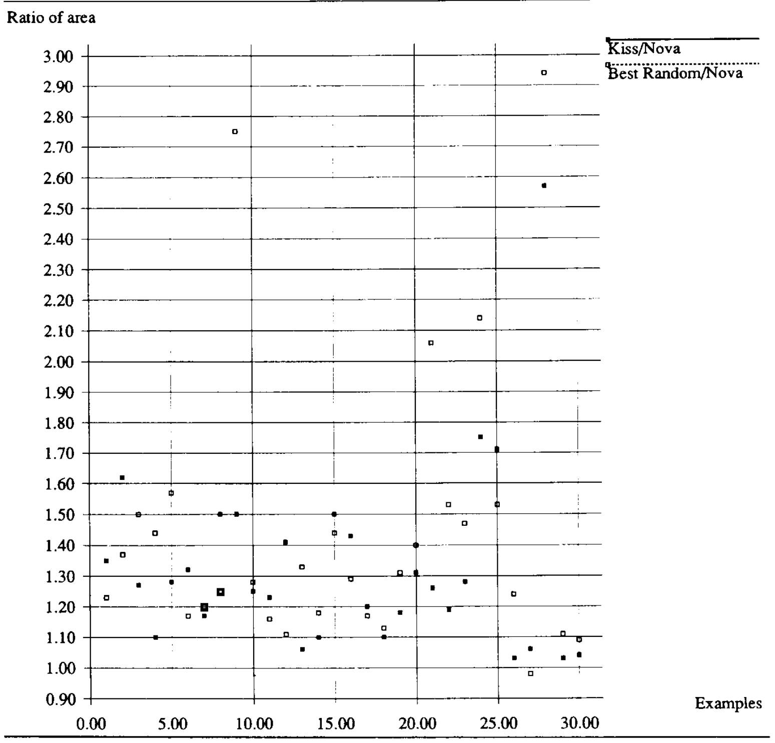
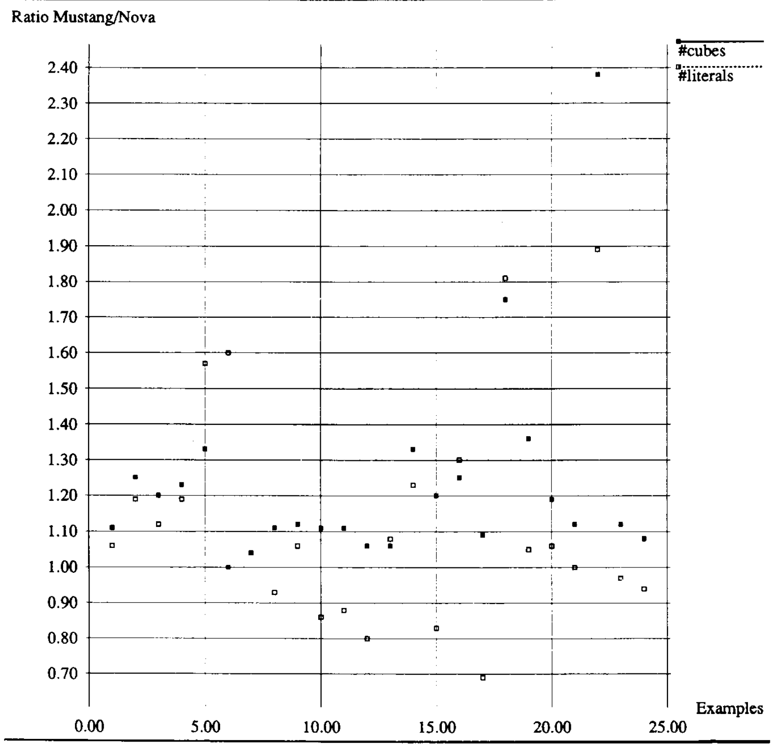


TABLE X
MUSTANG AND NOVA



NOVA can use any number of encoding bits greater than or equal to the minimum. The best results on the benchmark of Table I have been obtained with a minimum encoding length, but this is not always the case. Table II shows that although *iexact* achieves a number of product terms smaller than *ihybrid*, its final areas are always larger. This indicates that (at least in the case of input encoding only) increasing the code-length to satisfy all the constraints does not pay in terms of area. This explains why NOVA, even restricted to the algorithms that use only input constraints, achieves smaller areas than KISS does. KISS guarantees the satisfaction of all input constraints by an heuristic algorithm that does not always achieve the minimum necessary code-length. However, as noticed previously, even satisfying all input constraints with the exact code-length does not win in terms of area of a two-level logic implementation. Notice that in two cases (*ex2* and *ex5*) the number of cubes reported for *ihybrid* are fewer than those for *iexact*. The reason is that the codes found by *ihybrid* satisfy implicitly some conjunctive output relations [18] that help to achieve a better final cardinality of the product terms.

The issue of forecasting the effect on the encoding of conjunctive relations in the output part is being fully addressed in [18]. The code-length/product-terms tradeoff, when both input and output constraints are present, requires more powerful heuristics than currently implemented and we will experiment to find better ones. We plan also to analyze the variations of the basic scheme of symbolic minimization to characterize the pair (*IC*, *OC*) that translates into the best upper bound in the shorter encoding length. An extension of our algorithms to the case when the proper output part is given symbolically will be investigated.

Acknowledgment

Thanks are due to Alex Saldanha for lively discussions on symbolic minimization and providing the code of the *out_encoder* routine. The authors also acknowledge Sally Floyd for a careful reading of an earlier draft of the manuscript.

References

[1] J. Hartmanis, "On the state assignment problem for sequential machines—1," *IRE Trans. Elect. Comput.*, vol. EC-10, pp. 157-165, June 1961.

[2] R. E. Stearns and J. Hartmanis, "On the state assignment problem for sequential machines—2," *IRE Trans. Elect. Comput.*, vol. EC-10, pp. 593-603, Dec. 1961.

[3] D. B. Armstrong, "A programmed algorithm for assigning internal codes to sequential machines," *IRE Trans. Elect. Comput.*, vol. EC-11, pp. 466-472, Aug. 1962.

[4] R. Karp, "Some techniques for state assignment for synchronous sequential machines," *IEEE Trans. Elect. Comput.*, vol. EC-13, pp. 507-518, Oct. 1964.

[5] T. A. Dolotta and E. G. McCluskey, "The coding of internal states of sequential machines," *IEEE Trans. Elect. Comput.*, vol. EC-13, pp. 549-562, Oct. 1964.

[6] G. Saucier, "State assignment of asynchronous sequential machines using graph techniques," *IEEE Trans. Comput.*, vol. C-21, pp. 282-288, Mar. 1972.

[7] G. De Micheli, A. Sangiovanni-Vincentelli, and T. Villa, "Computer-aided synthesis of PLA-based finite state machines," presented at ICCAD, Sept. 1983.

[8] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* New York: Kluwer Academic, 1984.

[9] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 269-284, July 1985.

[10] G. De Micheli, "Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 597-616, Oct. 1986.

[11] T. Villa, "Constrained encoding in hypercubes: Algorithms and applications to logical synthesis," Memo UCB/ERL M87/37, Univ. California, Berkeley, May 1987.

[12] S. Devadas, H. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines for optimal multi-level logic implementations," presented at ICCAD, Nov. 1987.

[13] G. Saucier, M. Crastes de Paulet, and P. Sicard, "ASYL: A rule-based system for controller synthesis," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1088-1097, Nov. 1987.

[14] A. Saldanha, "Pla optimization using output encoding," U.C. Berkeley Master's Rep., Aug. 1988.

[15] T. Villa, "NOVA," User's manual, Univ. California, Berkeley, Oct. 1988.

[16] S. Devadas and R. Newton, "Exact algorithms for output encoding, state assignment and four-level Boolean minimization," Memo UCB/ERL M89/8, Univ. California, Berkeley, Feb. 1989.

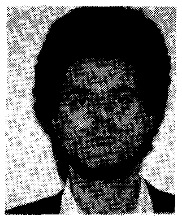
[17] A. Saldanha and T. Villa, "Symbolic minimization revisited," to be published.

[18] —, "Output encoding for optimal state assignment of finite state machines," to be published.

[19] W. Wolf, K. Keutzer, and J. Akella, "Addendum to 'A kernel-finding state assignment algorithm for multi-level logic'," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 925-927, Aug. 1989.

[20] T. Villa and A. Sangiovanni-Vincentelli, "Algorithms for state assignment of finite state machines for optimal two-level logic implementations," in *Proc. Int. Workshop on Logic Synthesis*, May 1989.

*



Tiziano Villa received the Laurea degree in mathematics from the University of Milano, Italy, in 1977, the Diploma of the Mathematical Tripos, Part III from the D.A.M.T.P., Cambridge, U.K. in 1982, and the M.S. degree in computer science from the University of California, Berkeley, in 1987.

From 1980 to 1985 he was a member of the technical staff of the C.S.E.L.T. Labs, Torino, Italy, in the division of computer-aided design of integrated circuits. Since 1987 he has been a researcher in the Electronics Research Laboratory, University of California, Berkeley. His research interests include switching and automata theory, logic synthesis and verification of VLSI circuits, combinatorial optimization, and artificial learning.

*

Alberto Sangiovanni-Vincentelli (M'74-SM'81-F'83), for a photograph and biography please see page 18 of the January 1990 issue of this TRANSACTIONS.