SYMBOLIC TWO-LEVEL MINIMIZATION

by

Tiziano Villa, Alex Saldanha, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/109

19 December 1995

SYMBOLIC TWO-LEVEL MINIMIZATION

by

Tiziano Villa, Alex Saldanha, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/109

19 December 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Symbolic two-level minimization

Tiziano Villa Alex Saldanha Robert K. Brayton
Alberto L. Sangiovanni-Vincentelli
Department of EECS
University of California at Berkeley
Berkeley, CA 94720

December 19, 1995

Abstract

We present a symbolic minimization procedure to obtain optimal two-level implementations of finite-state machines. Encoding based on symbolic minimization consists of optimizing the symbolic representation, and then transforming the optimized symbolic description into a compatible two-valued representation, by satisfying encoding constraints (bit-wise logic relations) imposed on the binary codes that replace the symbols. Our symbolic minimization procedure captures completely the sharing of product-terms due to "OR-ing" effects in the output part of a two-level implementation of the symbolic cover. Encoding constraints are generated by the minimization procedure. Product-terms are accepted in a symbolic minimized cover only when they induce compatible encoding constraints. At the end a set of codes that satisfy all constraints is computed. The quality of this synthesis procedure is shown by the fact that the cardinality of the cover obtained by symbolic minimization and of the cover obtained by replacing the codes in the initial cover and then minimizing with a two-level minimizer such as ESPRESSO are very close. Experiments exhibit a set of hard examples where our procedure improves on the best results of state-of-art tools.

1 Introduction

The optimization of logic functions performed on the Boolean representation depends heavily on the encoding chosen to represent the symbolic variables.

The cost function that estimates the area optimality of an encoding depends on the target implementation: two-level or multi-level or field-programmable gate arrays (FPGA's). The cost of a two-level implementation is the number of product-terms or the area of a programmable logic array (PLA). A commonly used cost of a multi-level implementation is the number of literals of a technology-independent representation of the logic. FPGA's come in different architectures with associated costs. Other optimization objectives may be related to power consumption, speed and testability. It may even be the case that the objective is a correctness requirement, as is race-freeness in state assignment of asynchronous circuits.

The following optimal encoding problems may be defined:

(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 finite state machine (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 the encoding of the signals connecting two (or more) combinational circuits.

Here we concentrate on problems in class D for optimal two-level implementations. In particular we will refer mostly to the problem of encoding FSM's, since there is no loss of generality and they are of great practical interest.

We will build on the paradigm started by [7]. It involves optimizing the symbolic representation (symbolic minimization), and then transforming the optimized symbolic description into a compatible two-valued representation, by satisfying encoding constraints (bit-wise logic relations) imposed on the binary codes that replace the symbols. This approach guarantees an upper bound on the size of the encoded symbolic function provided all the encoding constraints are satisfied. Encoding via symbolic minimization may be considered a three step process. The first phase consists of multiple-valued optimization. The second step is to extract constraints on the codes of the symbolic variables, which, if satisfied, guarantee the existence of a compatible Boolean implementation. The third step is assigning to the symbols codes of minimum length that satisfy these constraints, if the latter imply a set of non-contradictory bit-wise logic relations.

When the target implementation is two-level logic, the first step may consist of one or more calls [7, 6] to a multiple-valued minimizer [9], after representing the symbolic variables with positional cube notation [13, 9]. Then constraints are extracted and a constraints satisfaction problem is set up.

Using the paradigm of symbolic minimization followed by constraints satisfaction, the most common types of constraints that may be generated [7, 6, 3, 11] are four. The first type, generated by the input variables, are *face-embedding* constraints. The three types generated by the output variables are *dominance*, *disjunctive* and *disjunctive-conjunctive* constraints. Each face-embedding constraint specifies that a set of symbols is to be assigned to one *face* of a binary n-dimensional cube and no other symbol should be in that same face. Dominance constraints require that the code of a symbol covers bit-wise the code of another symbol. Disjunctive constraints specify that the code of a symbol must be expressed as the bit-wise disjunction (*oring*) of the codes of two or more other symbols. Disjunctive-conjunctive constraints specify that the code of a symbol must be expressed as the bit-wise disjunction (*oring*) of the bit-wise conjunction (*anding*) of the codes of two or more other symbols.

Our approach wants to strike a balance between the exact, but computationally intractable exact formulation provided by generalized prime implicants [3], and solutions that make no attempt of using a complete set of operations in looking for a good code [15, 6]. One of the issues that we will clarify is the completness of sets of encoding constraints to find an optimal solution. Then we will propose a heuristic search strategy to trade-off quality of results vs. computing time.

The presentation is organized as follows. In Section 3 we present the encoding problem for optimal two-level implementations. In Section 4 the new symbolic minimization algorithm is described, while procedures for symbolic reduction and symbolic oring are explained, respectively, in Section 5 and in Section 6. Section 7 analyzes some ordering schemes. In Section 8 mention is made of the algorithms used for checking encodeability. An example is demonstrated in Section 9, and experiments are reported in Section 10, with final conclusions drawn in Section 11.

2 Definitions

2.1 Finite State Machines

For definitions of two-level logic minimization we refer to [1]. Here we describe the connection between a FSM tabular description and its interpretation as a multiple-valued logic function.

A logic function may have multiple-valued (called also symbolic) input variables and symbolic output variables. A symbolic input or output variable takes on symbolic values. 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 or symbolic 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 representation 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 and outputs, if they are symbolic).

3 Encoding for Two-level Implementations

3.1 Multi-valued Minimization

Advances in the state assignment problem, reported in [8, 1, 7], made a key connection to multiple-valued logic minimization, by representing the states of a FSM as the set of possible values of a single multiple-valued variable. A multiple-valued minimizer, such as [9], can be invoked on the symbolic representation of the FSM. This can be done by representing the symbolic variables using the positional cube notation [13, 9]. 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. To get a compatible boolean representation, one must assign each of the groups of states obtained by MV minimization, (called face or 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 face constraint. This problem is called face embedding problem.

It is worth mentioning that the face constraints obtained through straightforward symbolic minimization are sufficient, but not necessary to find a two-valued implementation matching the upper bound of the multi-valued minimized cover. As it was already pointed out in [6], for each implicant of a minimal (or minimum) multi-valued cover, one can compute an *expanded implicant*, whose literals have maximal (maximum) cardinality and a *reduced implicant* whose literals have minimal (minimum) cardinality. By bit-wise comparing the corresponding expanded and reduced implicant, one gets *don't cares* in the input constraint, namely, in the bit positions where the expanded implicant has a 1 and the reduced implicant has a 0. The face embedding problem with *don't cares* becomes one of finding a cube of minimum dimension k, where, for every face constraint, one can assign the states asserted to vertices of a subcube that does not include any state not asserted, whereas the *don't care* states can be put inside or outside of that subcube. One can build examples where the presence of *don't cares* allows to satisfy the input constraints in a cube of smaller dimension, than it would be possible otherwise.

3.2 Symbolic Minimization

Any encoding problem, where the symbolic variables only appear in the input part, can be solved by setting up a multiple-valued minimization problem followed by satisfaction of the induced face constraints.

However, the problem of state assignment of FMS's is only partially solved by this scheme, because the encoding of the symbolic output variables is not taken into account (e.g. the next state variable). Simple multiple-valued minimization disjointly minimizes each of the on-sets of the symbolic output functions, and therefore disregards the sharing among the different output functions taking often place when they are implemented by two-valued logic. We will now see more powerful schemes to deal with both input and output encoding.

In [6, 15] a new scheme was proposed, called *symbolic minimization.* Symbolic minimization was introduced to exploit bit-wise dominance relations between the binary codes assigned to different values of a symbolic output variable. The core of the approach is a procedure to find useful bit-wise dominance (called also covering) constraints between the codes of next states, based on the fact that the input cubes of the onset of a dominating code can be used as don't cares for minimizing the input cubes of the onset of a dominated code. For instance, consider a fragment of a symbolic cover:

10	st1	st2	11
00	st2	st2	11
01	st2	st2	00
00	st3	st2	00
10	st2	st1	11
00	st1	st1	–
01	st3	st0	00

If $enc(st1) > enc(st2)$, and $enc(st0) > enc(st2)$ (i.e. $st1$ asserted implies $st2$ asserted and $st0$ asserted implies $st2$ asserted), then the transitions with next states $st1$ and $st0$ can be used as don't cares when minimizing the transitions with next state $st2$:

10	st1	st2	11
00	st2	st2	11
01	st2	st2	00
00	st3	st2	00
10	st2	-	11
00	st1	-	–
01	st3	-	00

An equivalent minimized symbolic cover is:

-0	st1,st2	st2	11
0-	st2,st3	st2	00
10	st2	st1	11
00	st1	st1	–
01	st3	st0	00

Notice that also face constraints $(st1, st2)$ and $(st2, st3)$ must be satisfied. The translation of a cover obtained by symbolic minimization into a compatible boolean representation defines simultaneously a face embedding problem and a dominance constraints satisfaction problem. Notice that any output encoding problem can be solved by symbolic minimization. Symbolic minimization was applied also in [10], where a particular form of PLA partitioning is examined, by which the outputs are encoded to create a reduced PLA that is cascaded with a decoder.

(1)	10	st1	st2	11
(2)	00	st2	st2	11
(3)	01	st2	st2	00
(4)	00	st3	st2	00
(5)	10	st2	st1	11
(6)	10	st3	st1	11
(7)	00	st1	st1	- -
(8)	01	st3	st0	00
(9)	11	st1	st1	10
(10)	11	st3	st3	01
(11)	11	st0	st0	11

(1')	-0	st1,st2	st2	11
(2')	0-	st2,st3	st2	00
(3')	10	st2,st3	st1	11
(4')	00	st1	st1	- -
(5')	01	st3	st0	00
(6')	11	st1,st0	st1	10
(7')	11	st0,st3	st3	01

Figure 1: Covers of FSM before and after symbolic minimization

However, to mimic the full power of two-valued logic minimization, another fact must be taken into account. When the code of a symbolic output is the bit-wise disjunction of the codes of two or more other symbolic outputs, the on-set of the former can be minimized by using the on-sets of the latter outputs, by "redistributing" the task of implementing some cubes. An extended scheme of symbolic minimization can therefore be defined to find useful dominance and disjunctive relations between the codes of the symbolic outputs. The translation of a cover obtained by extended symbolic minimization into a compatible boolean representation induces a face embedding, output dominance and output disjunction satisfaction problem.

In Figure 1, we show the initial description of a FSM and an equivalent symbolic cover returned by an extended symbolic minimization procedure.

The reduced cover is equivalent to the original one if we impose the following constraints on the codes of the states.

Product terms (1'), (3') and (4') are consistent with the original product terms (5) and (7) if we impose $code(st1) > code(st2)$. In a similar way, product terms (2') and (5') are consistent with the original product term (8) if we impose $code(st0) > code(st2)$. The product terms (1') and (2') yield also the face constraints $face(st1, st2)$ and $face(st2, st3)$, meaning that the codes of $st1$ and $st2$ ($st2$ and $st3$) span a face of a cube, to which the code of no other state can be assigned. The previous face and dominance constraints together allow to represent the four original transitions (1), (2), (3), (4) by two product terms (1') and (2').

Product term (3') is equivalent to the original transitions (5) and (6) and yields the face constraint $face(st2, st3)$. This saving is due to a pure input encoding join effect.

Finally the product terms (6'), (7') represent the original transitions (9), (10) and (11). The next state of (11) is $st0$, that does not appear in (6') and (7'). But, if we impose the disjunctive constraint $code(st0) = code(st1) \vee code(st3)$, i.e., we force the code of $st0$ to be the bit-wise *or* of the codes of $st1$ and $st3$, we can redistribute the transition (11) between the product terms (6') and (7'). The product terms (6') and (7') yield also the face constraints $face(st1, st0)$ and $face(st0, st3)$; together with the previous disjunctive constraint they allow the redistribution of transition (11).

We point out that if we perform a simple MV minimization on the original description we save only one product term, by the join effect taking place in transition (3').

An encoding satisfying all constraints can be found and the minimum code length is two. A solution is given by $st_0 = 11, st_1 = 01, st_2 = 00, st_3 = 10$. If we replace the states by the codes in the minimized symbolic cover, we obtain an equivalent Boolean representation that can be implemented with a PLA, as shown in Figure 2. Note that we replace the groups of states in the present state field with the unique face

(1")	-0	0-	00	11
(2")	0-	-0	00	00
(3")	10	-0	01	11
(4")	00	01	01	- -
(5")	01	10	11	00
(6")	11	-1	01	10
(7")	11	1-	10	01

Figure 2: Encoded cover of FSM

assigned to them and that product term (2") is not needed, because it asserts only zero outputs. Therefore the final cover has only six product terms.

3.3 Completness of Encoding Constraints

An important question is whether the constraints described earlier are sufficient to explore the space of all encodings. More precisely, the question is: find the class of encoding constraints such that by exploring all of them one is guaranteed to produce a minimum encoded implementation. Of course exploring all the encoding constraints of a given class may be impractical, but if the answer to the previous question is affirmative, one has characterized a complete class that can lead in line-of-principle to an optimal solution. This would make more attractive an heuristic that explores the codes satisfying the constraints of such a class. We now draw a distinction between a *symbolic state* and an *hardware state*. The former is a state of the original FSM. The latter is a state of the encoded FSM. If the number of encoding bits is k and the number of symbolic states is n, there are $2^k - n$ hardware states that do not correspond to an original symbolic state. If $2^k = n$, there are as many hardware states as there are symbolic states.

Theorem 3.1 *Face and disjunctive constraints are sufficient to obtain a minimum two-level implementation of a state-minimized FSM if the minimum implementation has as many hardware states as there are symbolic states.*

Proof. Consider an FSM F. Let the codes that produce a minimum implementation of the FSM be given, together with the best implementation C (here minimum or best refers to the smallest cardinality of a two-level cover). Suppose that the product-terms of the minimum encoded implementation C are all prime implicants. Consider each cube of C. Its present state part will contain the codes of one or more states and it will translate into a face constraint. Its next state part will correspond to the code of a symbolic state (using the hypothesis that there are as many hardware states as symbolic states). Consider now each minterm of the original FSM F. It will be covered in the input part (proper input and present state) by one or more cubes of C; this will translate into a disjunctive constraint whose parent is the next state of the minterm and whose children are the next states of the covering cubes of C.

The face constraints and disjunctive constraints so obtained are necessary for a set of codes to produce such a minimum implementation, when they are replaced in the original cover and then the cover is minimized. But are they sufficient ? There may be many sets of codes that satisfy these constraints. Is any such set sufficient to obtain a minimum cover ? The answer if yes, if after that the set of codes is replaced in the original FSM, an exact logic minimizer is used. Indeed, if this set of codes satisfies the encoding constraints, by construction they make possible to represent the minterms of the original FSM cover by the

cubes of the minimum cover C. Therefore an exact logic minimizer will produce either C or a different cover of the same cardinality as C [1]. □

Theorem 3.2 *Face and disjunctive-conjunctive constraints are sufficient to obtain a minimum two-level implementation of a state-minimized FSM.*

Proof. If there are as many hardware states as there are symbolic states the previous result applies. If the best implementation has more hardware states than symbolic states, one must introduce disjunctive-conjunctive constraints. The reason is that it is not anymore always true that the next state of a cube $c \in C$ corresponds to the code of a symbolic state. Suppose that the next state of a cube c is not the code of a symbolic state. c cannot be a minterm in the input part, otherwise, since we suppose that C contains only prime implicants, the next state of c must be exactly the code of the state of the symbolic minterm in F to which c corresponds. So c must contain more than one minterm in the input part, say w.l.o.g. that c contains exactly two minterms m_1 and m_2, each corresponding to a symbolic minterm of the care set of F. If the symbolic minterms corresponding in F to c_1 and c_2 assert next states s_1 and s_2, the next state of c must be the intersection of the codes of s_1 and s_2 (for sure the next state of c must be dominated by the intersection of the codes of s_1 and s_2, but we suppose that c is a prime implicant and that it contains exactly minterms m_1 and m_2 of the care set, so we can say that the next state of c is exactly the intersection of the codes of s_1 and s_2).

Therefore for each symbolic minterm m_s in F one defines a disjunctive-conjunctive constraint enforcing that the code of the next state of m_s is a disjunction of conjunctions, where each disjunct is contributed by one of the cubes of C that contain the input part of the minterm corresponding to m_s, and for each such cube c_{m_s} the conjuncts are the codes of the next states asserted by all the care set minterms that c_{m_s} contains. The rest of the reasoning goes as in the previous theorem. □

Disjunctive-conjunctive constraints were introduced for the first time in [3], as the constraints induced by generalized prime implicants. Our derivation shows that they arise naturally when one wants to find a complete class of encoding constraints. In our symbolic minimization algorithm we used as the class of encoding constraints face constraints, dominance constraints and disjunctive constraints. Dominance constraints are not necessary, but they have been considered useful in developing an heuristic search strategy. We did not use disjunctive-conjunctive constraints in the heuristic procedure presented here.

4 A New Symbolic Minimization Algorithm

4.1 Structure of the Algorithm

In this section a new more powerful paradigm of symbolic minimization is presented. An intuitive explanation of symbolic minimization as proposed in [6] and enhanced in [15] has been given in Section 3. To help in highlighting the differences of the two schemes, the one in [15] is summarized in Figure 3.

The new scheme of symbolic minimization features the following novelties.

- Symbolic oring. Disjunctive constraints are generated corresponding to the case of transitions of the initial cover implicitly expressed by other transitions in the encoded two-level representation, because of the oring effects in the output part.

[1] The hypothesis that the FSM is state-minimized guarantees that the minimum implementation does not have fewer hardware states than there are symbolic states.

1. Input data cover C with q symbolic outputs,
   optional binary outputs,
   empty acyclic graph G,
   and empty cover FinalP
   Output is the graph G and the minimal cover FinalP
2. On_k = on-set implicants of k-th output symbol
   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 symbol i
9. $G = G \cup \{(j, i)$ such that M_i intersects $On_j\}$
   $P = P \cup MB_i$
10. FinalP = minimize(P)

Figure 3: Old Symbolic Minimization Scheme

- Implementability. Product-terms are accepted in the symbolic cover, only when they yield satisfiable encoding constraints.

- Symbolic reduction. Symbolic minimization is iterated until an implementable cover is produced. A symbolic reduction procedure guarantees that this always happens.

At last, codes satisfying the given encoding constraints are generated. The accuracy of the synthesis procedure can be measured by the fact that the cardinality of the symbolic minimized cover is very close to the cardinality of the original encoded FSM minimized by ESPRESSO [1]. This will be shown in the section of results.

We introduce the following abbreviations useful in the description of the algorithm:

- $IniCov = (Fc, Dc, Rc)$ is the initial cover of a 1-hot encoded FSM, where Fc, Dc and Rc are, respectively, the on-set, dc-set and off-set of the 1-hot encoded FSM.

- Ns is the set of next states of a FSM. Fc_{ns}, Dc_{ns} and Rc_{ns} are the set of product-terms asserting ns, respectively, in Fc, Dc and Rc, $\forall ns \in Ns$.

- On_{ns}, $Dcare_{ns}$ and Off_{ns} are, respectively, the on-set, dc-set and off-set of next state ns, $\forall ns \in Ns$, On_{ns}.

- On_{bo}, Dc_{bo} and Off_{bo} are, respectively, the on-set, dc-set and off-set of the binary output functions.

- $PartCov = (OnCov, DcCov, OffCov)$ is the cover of a fragment of a 1-hot encoded FSM, where $OnCov$, $DcCov$ and $OffCov$ are, respectively, the on-set, dc-set and off-set of the given fragment.

- $Cons_{ns}$ is the set of input and output constraints yielded by symbolic minimization of Fc_{ns}, $\forall ns \in Ns$. The sets $Cons_{ns}$ are cumulated in $Cons$.

- $ExpCov_{ns}$ and $RedCov_{ns}$ are, respectively, a maximally expanded and a maximally reduced minimized cover of Fc_{ns}, $\forall ns \in Ns$. The sets $ExpCov_{ns}$ and $RedCov_{ns}$ are cumulated, respectively, in $ExpCov$ and $RedCov$.

At the each step of the symbolic minimization loop a new next state ns is chosen by the procedure *SelectState*, described in Section 7. The goal is to determine a small set of multiple-valued product-terms that represent the transitions of Fc_{ns}. The procedure *SymbOring*, described in Section 6, determines Or_{ns}, the transitions of Fc_{ns} that can be realized by expanding some product-terms in the current $RedCov$ and choosing the expansions in the interval ($RedCov$, $ExpCov$). This expansion operation yields updated encoding constraints (here also disjunctive constraints are generated) that must be imposed to derive an equivalent two-level implementation. The rest of Fc_{ns} is minimized, putting in its off-set the on-sets of all states selected previously [2]. The minimization is done calling ESPRESSO, without the final *make_sparse* step. This produces $ExpCov_{ns}$, a maximally expanded minimized cover. Calling the ESPRESSO procedure *mv_reduce* on $ExpCov_{ns}$ produces $RedCov_{ns}$, a maximally reduced minimized cover. The reduced minimized cover $RedCov_{ns}$ yields new encoding constraints $Cons_{ns}$.

If it turns out that the constraints in $Cons_{ns}$ are not compatible with the constraints already in $Cons$, a *SymbReduce* procedure is invoked to redo the minimizations of Fc_{ns} and produce covers that yield encoding constraints compatible with those currently accepted in $Cons$. In Section 5, where *symb_reduce* is described, it is shown that this always happens, i.e. this symbolic reduction step always produces an implementable symbolic minimized cover of Fc_{ns}. The compatible constraints $Cons_{ns}$ are added to $Cons$ and the new

[2] This is not required: one should put only those states that ns covers.

accepted covers $ExpCov_{ns}$ and $RedCov_{ns}$ are added, respectively, to $ExpCov$ and $RedCov$. Finally, codes satisfying the encoding constraints in $Cons$ are found and replaced in the reduced symbolic minimized cover $RedCov$. The resulting encoded minimized cover $EncRedCov$ is usually of the same cardinality as the cover obtained by replacing the codes in the original symbolic cover and then minimizing it with ESPRESSO. $EncRedCov$ can be minimized again using ESPRESSO to produce a cover $MinEncRedCov$, that rarely has fewer product-terms than $EncRedCov$. These statements will be supported by results in the experimental section. To check the correctness of this complex procedure a verification is made of $MinEncRedCov$ against $EncIniCov$. A non-equivalence of them signals an error in the implementation.

The outlined procedure is shown in Figure 4. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactical usage), while the routines with initial letter in the upper case are new and will be described in the following sections.

Proposition 4.1 *The algorithm of Figure 4 generates an implementable symbolic cover.*

Proof. By construction a product term is added to the symbolic cover only if it carries constraints on the codes that are compatible with the constraints of all the symbolic cubes accumulated up to then. Therefore one guarantees that the symbolic cover is always implementable at any stage of its construction. □

4.2 Slice Minimization and Induced Face and Dominance Constraints

The procedure *Constraints* computes the face and dominance constraints induced by a pair of minimized covers $(RedCov_{ns}, ExpCov_{ns})$ with respect to the original cover Fc. For each product-term $pexp \in ExpCov_{ns}$ there is a companion product-term $pred \in RedCov_{ns}$ obtained from $pexp$ by applying to it the multiple-valued reduce routine of ESPRESSO. For each pair of product-terms $(pred, pexp) \in (RedCov_{ns}, ExpCov_{ns})$ one gets the implied face constraint by considering the 1-hot representation of the input part. For each position k in the input part of the 1-hot representation of $pred$ and $pexp$, opposite bits yield a don't care in the face constraint and equal bits yield the common care bit in the face constraint. Face constraints are generated for all symbolic input variables, including proper symbolic inputs, if any.

Dominance constraints are computed by determining, for each product-term $pred \in RedCov$, the transitions of the original cover Fc that $pred$ intersects in the input part. The next states that these transitions assert must cover the next state of $pred$, for the functionality of the FSM to be maintained. Notice that currently we compute only the dominance constraints implied by the product-terms in $RedCov$. Computing them both for $RedCov$ and $ExpCov$ (as we do in the case of input face constraints with the notion of don't care input constraints), would allow to explore a larger part of the solution space. This is not currently done, because it would make the constraint satisfaction problem more complex.

Oring constraints are generated only in the *SymbOring* procedure described in Section 6. In Figure 5 the pseudo-code of *Constraints* is shown.

5 Symbolic Reduction

The procedure *SymbReduce* is invoked to set up a series of new minimizations that produce an implementable minimized cover of $OnCov$. This is required when a set of constraints $Cons_{ns}$ incompatible with those in $Cons$ are obtained at a certain iteration in the loop of *symbolic*. When this happens, it means that we cannot minimize the current $OnCov$ (with the current $DcCov$), because the minimization process would merge multiple-valued product-terms in such a way that incompatible constraints are generated. Instead we can minimize $OnCov$ by blocks and control the allowed companion dc-sets so that only compatible

```
procedure symbolic(Fc, Dc, Rc) {
    do { /* repeat until all next states are selected */
        /* Sel is a set of currently selected states */
        ns = SelectState(Ns − Sel);   Sel = Sel ∪ ns
        /* Or_ns are the transitions of Fc_ns expressed by oring */
        (Or_ns, ExpCov, RedCov, Cons)
            = SymbOring(IniCov,ExpCov,RedCov,Cons)
        /* OnCov are the transitions to be covered */
        OnCov = Fc_ns − Or_ns
        /* add the on-sets of states previously selected to the off-set */
        OffCov = ⋃_{i∈Sel−ns} On_i
        /* add binary output off-set */
        OffCov = OffCov ∪ Off_bo
        /* everything else (including Or_ns) is in dc-set */
        DcCov = complement(OnCov,OffCov)
        /* invoke espresso with no makesparse */
        ExpCov_ns = espresso(OnCov,DcCov,OffCov)
        RedCov_ns = mv_reduce(ExpCov_ns,DcCov)
        Cons_ns = Constraints(IniCov,ExpCov_ns,RedCov_ns)
        if (ConstraintsCompatible(Cons,Cons_ns) fails)
            (ExpCov_ns,RedCov_ns,Cons_ns) =
                SymbReduce(IniCov,PartCov,ExpCov_ns,RedCov_ns,Cons,Cons_ns)
        ExpCov = ExpCov ∪ ExpCov_ns
        RedCov = RedCov ∪ RedCov_ns
        Cons = Cons ∪ Cons_ns
    } while (at least one state in Ns − Sel)
    Codes = EncodeConstraints(Cons)
    EncRedCov = Encode(RedCov, Codes) /* encode symbolic min. cover */
    MinEncRedCov = minimize(EncRedCov)
    EncIniCov = Encode(IniCov, Codes) /* encode initial FSM */
    MinEncIniCov = minimize(EncIniCov)
    if (verify(MinEncRedCov, EncIniCov) fails) ERROR
}
```

Figure 4: New Symbolic Minimization Scheme

```
/* face and dominance constraints induced by (RedCov_ns, ExpCov_ns) */
Constraints(IniCov,ExpCov_ns,RedCov_ns) {
    foreach (pair of product-terms (pred, pexp) ∈ (RedCov_ns, ExpCov_ns)) {
        foreach (position k in the 1-hot representation) {
            if (I(pred)[k] and I(pexp)[k] are opposite bits) face[k] = dc
                else face[k] = I(pred)[k]
        }
        foreach (transition t ∈ Fc) {
            /* don't intersect if t and pred assert same next state */
            if (t and pred assert different next states) {
                if (distance(I(pred),I(t)) = 0) {
                create covering constraint (nxst(t) > nxst(pred))
                }
            }
        }
    }
}
```

Figure 5: Derivation of face and dominance constraints

constraints are generated. It is evident that in the worst-case, if only one transition of $OnCov$ is minimized at a time, with an empty dc-set, we always obtain implementable product-terms. This is equivalent to perform no minimization at all. In *SymbReduce*, the transitions of $OnCov$ are partitioned into maximal sets of transitions that can be minimized together. Maximal companion dc-sets are found for each previous on-set of transitions.

The routine *SymbReduce* is divided in two steps. In the first step, a maximal subset of $Cons_{ns}$ is sought that is compatible with $Cons$. The rationale is that the companion product-terms of $ExpCov_{ns}$ and $RedCov_{ns}$ are an acceptable cover for a subset of $OnCov$. This is done in a greedy fashion. The constraints of $Cons_{ns}$ compatible with $Cons$ are saved into $AConsTmp$. A new constraint of $Cons_{ns}$ is checked for compatibility with $Cons \cup AConsTmp$. If it is compatible, it is added to $AConsTmp$, otherwise the product-term companion to the constraint is deleted from both $ExpCov_{ns}$ and $RedCov_{ns}$. The transitions of $OnCov$ not covered by the resulting $RedCov_{ns}$ are the new cover that must be minimized in such a way that only implementable multiple-valued product-terms are found. The transitions of $OnCov$ covered by the resulting $RedCov_{ns}$ are instead added to the dc-set.

In the second part, the current $OnCov$ (i.e. the part of the initial $OnCov$ left uncovered by the previous step) is minimized. The transitions of $OnCov$ that can be minimized together are saved into $OnCovTmp$. A new transition t of $OnCov$ is minimized together with $OnCovTmp$ to return both $ExpCovTMp$ and $RedCovTmp$. The implied constraints are computed in $AConsTmp$. If they are compatible with $Cons$, t is added to $OnCovTmp$. In this way one determines sets of transitions that can be minimized together. The dc-set of each such set of transitions is enlarged in a similar greedy fashion. The rationale is that one may obtain more expanded resulting product-terms useful in later stages of the algorithm. Then $ExpCov_{ns}$, $RedCov_{ns}$ and $Cons_{ns}$ are updated, respectively, with the saved accepted sets $ExpCovTmp$, $RedCovTmp$ and $AConsTmp$. This is iterated until all transitions of $OnCov$ are minimized.

The outlined procedure is shown in Figures 6 and 7. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactic usage), while the routines with initial letter in the upper case are new.

```
/* PartCov is the triple (OnCov,DcCov,OffCov) */
procedure SymbReducePart1(IniCov,PartCov,ExpCov_ns,RedCov_ns,Cons,Cons_ns) {
    /* choose greedily a maximal subset of compatible constraints */
    /* pt(c) is a product-term companion to constraint c */
    AConsTmp is empty
    foreach (constraint c ∈ Cons_ns) {
        if (ConstraintsCompatible(Cons,AConsTmp,c) succeeds) {
            AConsTmp = AconsTmp ∪ c
        } else {
            ExpCov_ns = ExpCov_ns − pt(c) /* pt(c) ∈ ExpCov_ns */
            RedCov_ns = RedCov_ns − pt(c) /* pt(c) ∈ RedCov_ns */
        }
    }
    Cons_ns = Cons_ns ∪ AconsTmp
    foreach (transition t in OnCov) {
        /* if the product-terms in RedCov_ns cover t */
        if (sharp(t, RedCov_ns) returns empty) {
            OnCov = OnCov − t
            DcCov = DcCov + t
        }
    }
}
```

Figure 6: Symbolic reduction - Part1

```
procedure SymbReducePart2(IniCov,PartCov,ExpCov_ns,RedCov_ns,Cons,Cons_ns) {
    do { /* piece-wise minimizations of what left in OnCov */
        OnCovTmp = ∅;  DcCovTmp = ∅
        /* choose greedily a maximal on-set */
        foreach (transition t in OnCov) {
            OffCovTmp = complement(OnCovTmp ∪ t, DcCovTmp)
            /* invoke espresso with no makesparse */
            ExpCovTmp = espresso(OnCovTmp ∪ t,DcCovTmp,OffCovTmp)
            RedCovTmp = mv_reduce(ExpCovTmp,DcCovTmp)
            AConsTmp = Constraints(IniCov, ExpCovTmp, RedCovTmp)
            if (ConstraintsCompatible(Cons,AConsTmp) succeeds) {
                OnCovTmp = OnCovTmp ∪ t
                OnCov = OnCov − t
                SaveExpCovTmp = ExpCovTmp;  SaveRedCovTmp = RedCovTmp
                SaveAConsTmp = AConsTmp
            }
        }
        /* choose greedily a maximal dc-set of previous on-set */
        foreach (transition t in DcCov) {
            OffCovTmp = complement(OnCovTmp, DcCovTmp ∪ t)
            /* invoke espresso with no makesparse */
            ExpCovTmp = espresso(OnCovTmp,DcCovTmp ∪ t,OffCovTmp)
            RedCovTmp = mv_reduce(ExpCovTmp,DcCovTmp)
            AConsTmp = Constraints(IniCov, ExpCovTmp, RedCovTmp)
            if (ConstraintsCompatible(Cons,AConsTmp) succeeds) {
                DcCovTmp = DcCovTmp ∪ t
                SaveExpCovTmp = ExpCovTmp;  SaveRedCovTmp = RedCovTmp
                SaveAConsTmp = AConsTmp
            }
        }
        Cons_ns = Cons_ns ∪ SaveAConsTmp
        ExpCov_ns = ExpCov_ns ∪ SaveExpCovTmp;
        RedCov_ns = RedCov_ns ∪ SaveRedCovTmp
    } while (at least one transition in OnCov)
}
```

Figure 7: Symbolic reduction - Part2

6 Symbolic Oring

In two-level logic minimization of multi-output functions the fact of sharing cubes among single outputs reduces the cardinality of the cover. When minimizing symbolic logic to obtain minimal encodable two-level implementations, one should detect the most profitable disjunctive constraints so that - after encoding - sharing of cubes is maximized. In Section 4 an example was given where *oring* in the output part accounts for most savings in the minimum cover. In the symbolic minimization loop presented in Section 4, $SymbOring$ is invoked to that purpose.

The goal of the procedure *SymbOring* is to determine a subset (if it exists) of the transitions of Fc_{ns} that can be realized using the product-terms of the partial minimized symbolic cover ($ExpCov$, $RedCov$). If so, that subset is moved from the on-set to the dc-set of the cover to minimize in the current step. The procedure is heuristic because it handles a transition of Fc_{ns} at a time and it introduces some approximations with respect to an exact computation. For each transition t of Fc_{ns} the following algorithm decides whether t can be realized using or modifying product-terms in $RedCov$. Here we present the main features, leaving out minor design choices.

At a certain step of the procedure *symbolic* a pair of partial covers ($ExpCov$,$RedCov$) is available. For each cube $pexp \in ExpCov$ there is a companion cube $pred \in RedCov$ (and viceversa) such that $pred$ is obtained by $pexp$ by applying to it the multiple-valued *reduce* routine of ESPRESSO. A cube $pred \in RedCov$ potentially useful to espress implicitly t must satisfy the conditions that its input part (denoted $I(pred)$) has non-empty intersection with $I(t)$ and the output part of t (denoted $O(t)$) covers $O(pred)$. All such cubes are collected in the cover $Inter(t)$. It may happen that $I(pred)$ does not intersect $I(t)$, but that $I(pexp)$ intersects $I(t)$, because in $pred$ the bit of the present state of t is lowered, while in $pexp$ it is raised. If so, one may raise temptatively also the bit in $pred$ to obtain another potentially useful cube that is added to $Inter(t)$. The product-term $pred$ raised in the present state of t is denoted by $raised(pred)_t$ [3].

The set $OrNstates(Inter(t))$ of next states of cubes in $Inter(t)$ is computed. Define $Inter(t)_S$ as the set of transitions of $Inter(t)$ with next state included in set S. In order that a disjunctive effect occurs it is necessary that, for at least two next states $s1$ and $s2$, $I(t)$ is covered both by the union of the input parts of all cubes in $Inter(t)_{s1}$ and by the union of the input parts of all cubes in $Inter(t)_{s2}$. Here covering is meant to be restricted to the next state function assumed as a single output. Suppose that $OrNstates$ has at least two elements. We determine the states s of $OrNstates$ such that the union of the input parts of the cubes in $Inter(t)_s$ covers $I(t)$, and discard the others. Moreover, in order that a disjunctive effect occurs it is necessary that, for all binary output functions, $I(t)$ is covered by the union of the input parts of all cubes in $Inter(t)$. If all previous tests are not satisfied, the attempt of expressing t by symbolic oring fails.

If the previous necessary conditions are satisfied, all subsets of elements in the set $OrNstates$ are computed in $Subset(OrNstates)$. Each such subset, denoted by or, is an oring pattern potentially useful to espress implicitly the transition t. For each oring pattern or, the procedure *OringCover* returns $OrCov(t)$, a subset of transitions of $Inter(t)_{or \cup \phi}$ (it means $Inter(t)$ restricted to next states in or or empty next state) that cover t, both in the next state output space and in the binary output spaces. Notice that *OringCover* may fail to find a cover even if it exists, because while the input space of the binary output functions can be covered by considering the whole $Inter(t)$, only a subset of it ($Inter(t)_{or \cup \phi}$) is considered by *OringCover*. Notice also that there may be many possible such covers, but only one is found. This may penalize the quality of the final results, because the computed cover may yield uncompatible constraints, while there is another cover that yields compatible constraints. We do not give the details of *OringCover*, that is based on

[3] In the current implementation p is not added to $Inter(t)$ if $I(p)$ is covered by the input part of another cube already in $Inter(t)$. The rationale is that product-terms whith a more expanded input part are preferred, because they are more likely to cover other transitions in the future. An exact algorithm should define the notion of don't-care intersecting product-terms, if one knows how to handle conditional dominance constraints.

a greedy strategy.

If a cover $OrCov(t)$ is found, one considers the modified partial minimized cover $RedCovTmp$, obtained from $RedCov$ by raising the present state bits according to what done in the generation of $Inter(t)$. Then the constraints implied by the modified cover are derived and checked for compatibility with the oring constraint *or* (since some product-terms of $RedCov$ have been raised in the present state, there are raised face constraints and by consequence dominance constraints must be recomputed). If the answer is positive, the transition t is implementable by oring and both $RedCov$ and $Cons$ are updated. Otherwise a new oring pattern from $Subset(OrNstates)$ is considered. When they have been all exhausted, a new transition of Fc_{ns} is taken into consideration. [4].

The outlined procedure is shown in Figures 8. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactic usage), while the routines with initial letter in the upper case are new.

7 Ordering of Symbolic Minimization

In the procedure *symbolic* described in Section 4, at each cycle of the symbolic minimization loop, states are partitioned in two sets: those selected in previous iterations (Sel) and those still unselected ($Ns - Sel$). At the start of a new cycle, a new state ns is selected by the procedure *SelectState* from $Ns - Sel$ and the state partition is updated.

The transitions of the FSM are partitioned, accordingly, in the transitions asserting the states in Sel and already minimized and the transitions asserting the states in $Ns - Sel$ and not yet minimized. We observe the following facts:

1. When a new state ns is selected, the transitions asserting it cannot be used later to minimize the transitions asserting states in $Ns - Sel - \{ns\}$. Therefore if one measures how much an unselected state can help in minimizing the other unselected states by dominance ($DomGain$), the state of minimum gain should be selected first.

2. When a new state ns is selected, the transitions asserting it cannot be espressed later using the transitions asserting states in $Ns - Sel - \{ns\}$. Therefore if one measures how much the minimization of an unselected state is helped by the other unselected states by oring ($OrGain$), the state of minimum gain should be selected first.

Summarizing, the problem of the best selection of a new state can be reduced to one of measuring the dominance and oring gains and then choosing the state that minimizes their sum ($TotGain = DomGain + OrGain$).

As an example, consider that $Ns = st0, st1, st2, st3, st4, st5, st6$. Suppose that currently $st0$, $st5$, $st6$ have been already selected and that a new state must be chosen among $st1$, $st2$, $st3$, $st4$, by computing their gain and choosing the minimum. We have devised two slightly different schemes for computing the gain of a state. In the first scheme, the gain of a state, for instance $st1$, can be computed by setting up a minimization as shown in Figure 9 (in the figure the covers are shown for the next state functions asserted by the unselected states). After the minimization, the difference in cardinality between the resulting and original covers gives one component of the gain, $DomGain$ (associated to the dominance constraints: $st1 > st2$, $st1 > st3$, $st1 > st4$). The second component of the gain, $OrGain$ (associated to the disjunctive constraints: $st1 = st2 \vee st3 \vee st4$, $st1 = st2 \vee st3$, $st1 = st2 \vee st4$, $st1 = st3 \vee st4$), is found by computing, for each transition asserting $st1$, whether its input part is covered by the input parts of the

[4] A better alternative would be to check for constraints compatibility while building $OrCov(t)$: do not add a new product-term to the subset of $OrCov(t)$ currently accepted, if together with it, it yields infeasible constraints .

```
procedure SymbOring(IniCov,ExpCov,RedCov,Cons) {
    foreach (transition t ∈ Fc_ns) {
        foreach (pair of product-terms (pred, pexp) ∈ (RedCov, ExpCov)) {
            if (I(pred) ∩ I(t) non-empty and O(t) ⊇ O(pred)) {
                Inter(t) = Inter(t) ∪ pred
            } else {
                if (I(pexp) ∩ I(t) non-empty and O(t) ⊇ O(pexp)) {
                    Inter(t) = Inter(t) ∪ raised(pred)_t
                }
            }
        }
        compute OrNstates(Inter(t))
        if (at least two states in OrNstates) {
            foreach (next state s ∈ OrNstates)
                if (⋃_{p∈Inter(t)_s} I(p) ⊉ I(t)) OrNstates = OrNstates − s
            foreach (binary output function)
                if (⋃_{p∈Inter(t)} I(p) ⊉ I(t)) OrNstates empty
        }
        if (at least two states in OrNstates) {
            generate Subset(OrNstates)
            foreach (element or of Subset) {
                OrCov(t) = OringCover(Inter(t)_{or∪φ},t,ExpCov,RedCov)
                if (OrCov(t) is not empty) {
                    RedCovTmp = Raise(RedCov,Inter(t),t)
                    ConsTmp = Constraints(IniCov,ExpCov,RedCovTmp)
                    if (ConstraintsCompatible(ConsTmp,or) succeeds) {
                        Or_ns = Or_ns ∪ t
                        RedCov = RedCovTmp
                        Cons = ConsTmp ∪ or
                        goto outer foreach loop
                    }
                }
            }
        }
    }
}
```

Figure 8: Symbolic oring

OnCov:
on-set of st2 0010000
on-set of st3 0001000
on-set of st4 0000100
OffCov:
on-set of st2 0001100
on-set of st3 0010100
on-set of st4 0011000
on-set of st0 0011100
on-set of st5 0011100
on-set of st6 0011100
DcCov:
on-set of st1 0011100

Figure 9: First scheme to compute the gain

transitions asserting at least two other unselected states, for the related next state functions and all binary output functions.

In the second scheme, the gain of a state can be computed by setting up a minimization as shown in Figure 10 (referring again to $st1$ in the previous example). After the minimization, the difference in cardinality between the resulting and original covers gives the overall gain $TotGain$, inclusive of both the dominance and disjunctive components.

The pseudo-code in Figure 11 shows the first scheme to compute the gain. The second one is simpler, since it does not include explicitly the covering check to measure the oring contribution (that is implicitly taken into account by the minimization process) and it is not shown here.

8 Satisfaction of Encoding Constraints

The described procedures require algorithms to check satisfiability of a set of face, dominance and disjunctive constraints, and to find minimum codes that satisfy them. We used the algorithms reported in [11], to which we refer for a complete description. They are based on the notion of encoding dichotomies that are candidate encoding columns. The notion of encoding dichotomy was pioneered in [14] and the connection with satisfaction of face constraints was established in [16]. Other contributions on the subject can be found in [12, 2] and more recently in [4, 5].

9 Symbolic Minimization by Example

In this section we clarify with an example the mechanics by which the oring effects plays an important role in the minimization of two-level logic. Then we demonstrate our algorithm for symbolic minimization on a simple example.

OnCov:
on-set of st2 0010000
on-set of st3 0001000
on-set of st4 0000100
on-set of st1 0011100
OffCov:
on-set of st2 0001100
on-set of st3 0010100
on-set of st4 0011000
on-set of st0 0011100
on-set of st5 0011100
on-set of st6 0011100

Figure 10: Second scheme to compute the gain

9.1 The Oring Effect in Two-level Logic

In two-level logic minimization of multi-output functions the fact of sharing cubes among single outputs reduces the cardinality of the cover. As an example, consider the following cover of a logic function of four input and four output variables:

```
1000 0100
0100 0001
1100 0101
0001 1000
1001 1100
0101 1001
1101 1101
0010 0010
1010 0110
0110 0011
1110 0111
0011 1010
1011 1110
0111 1011
1111 1111
```

and an equivalent minimum cover, as found by ESPRESSO:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001.
```

Consider the product term 1001 1100 that appears in the original cover. In the minimum cover, when the input cube 1001 is true, the first two product terms of the minimum cover are excited and the output

```
procedure SelectState(UnSel) {
    foreach (state st ∈ UnSel) {
        gain(st) = ComputeGain(st,UnSel)
    }
    sel = st ∈ UnSel with minimum gain(st)
}
procedure ComputeGain(IniCov,st,UnSel) {
    /* measure potential gains by dominance */
    OnCov = ⋃_{i∈(UnSel−st)} Fc_i
    OldCard = #(OnCov)
    foreach (state j ∈ UnSel − st)
        OffCov_j = ⋃_{i∈UnSel−j−st} On_i ∪ ⋃_{i∈Ns−UnSel} On_i
    OffCov = (⋃_{j∈UnSel−st} OffCov_j) ∪ Off_bo
    DcCov = complement(OnCov,OffCov)
    /* invoke espresso with no makesparse */
    OnCov = espresso(OnCov,,DcCov,OffCov)
    DomGain = OldCard − #(OnCov)
    /* measure potential gains by oring */
    foreach (transition t ∈ Fc_st) {
        foreach (state i ∈ UnSel − st) {
            OnCov_i = product-terms of OnCov asserting next state i
            if (I(t) ⊆ I(OnCov_i) for next state and binary output functions) {
                increment OrCount
                if (OrCount > 1) { /* t can be expressed by oring */
                    increment OrGain
                    goto outer foreach loop
                }
            }
        }
    }
    TotGain = DomGain + OrGain
}
```

Figure 11: Ordering of symbolic minimization

part 1100 is asserted. Therefore the product term 1001 1100 is implemented by means of the product terms $---1$ 1000 and $1---$ 0100. Notice that two product terms must be in any cover to realize the following product terms of the original cover 1000 0100 and 0001 1000. Therefore a net saving of one product term (the one needed to realize 1001 1100) has been achieved in the minimum cover. We say that the product term 1001 1100 has been realized by oring or disjunctive effect (due to the semantics of the output part of a two-level implementation) or that it has been redistributed through the two product terms $---1$ 1000 and $1---$ 0100. The oring effect accounts for most savings in the minimum cover of this example.

9.2 A Worked-out Example of Symbolic Minimization

This subsection contains an example of symbolic minimization. The example is *shiftreg* from the MCNC suite. The symbolic cover of *shiftreg*, using the syntax of ESPRESSO, is:

```
.mv 4 1 -8 -8 1
.type fr
.kiss
0 st0 st0 0
1 st0 st4 0
0 st1 st0 1
1 st1 st4 1
0 st2 st1 0
1 st2 st5 0
0 st3 st1 1
1 st3 st5 1
0 st4 st2 0
1 st4 st6 0
0 st5 st2 1
1 st5 st6 1
0 st6 st3 0
1 st6 st7 0
0 st7 st3 1
1 st7 st7 1
```

Suppose that the ordering routine returned $st0, st4, st1, st2, st5, st3, st6, st7$ as the order in which the slices of next states must be minimized. Let each position in the 1-hot encoded notation correspond respectively to the states $st0, st4, st1, st2, st5, st3, st6, st7$. For instance 10000000 represents $st0$, while 01000000 represents $st4$. Slices including all the transitions that have the same next state are minimized in the given order. The result of each minimization is a set of symbolic cubes which realize the slice. A dc-set as specified by the theory is provided in each minimization. If terms of the dc-set having a different next state are used in a minimization, then covering constraints are introduced, together with companion face constraints (face constraints not related to output constraints can be introduced also, when transitions having the same next state are merged). Before each minimization, the algorithm figures out whether some transitions of the given slice can be realized by symbolic cubes already in the partial minimized symbolic cover, when a satisfiable oring constraint is imposed. Only the remaining transitions are kept in the onset of the slice under minimization. Whenever symbolic cubes that impose constraints on the codes are added to the cover, their consistency with respect to the constraints cumulated up to then is verified. As long as the consistency verification fails, different symbolic cubes are tried; eventually an encodeable symbolic cover is constructed. At the end codes of minimum code-length that satisfy the constraints are found and the codes are replaced

in the symbolic cover and in the original FSM (it is not necessary, but convenient to do both, because don't cares can be used differently, producing covers not of the same cardinality). A final step of two-valued minimization produces a minimal encoded FSM.

- Minimization of the slice of next state *st*0.
  The onset is:

```
0 10000000 100000000
0 00100000 100000001
```

  The dcset is:

```
1 11000000 100000000
- 01010010 100000000
1 00100000 111111111
- 00001101 111111111
- 11111111 011111110
```

  The minimized expanded cover is:

```
- 11111111 111111110
- 00101101 111111111
```

  The minimized reduced cover is:

```
- 11111111 100000000
- 00100000 000000001
```

  The constraints $code(st4) > code(st0), code(st1) > code(st0), code(st2) > code(st0), code(st5) > code(st0)$, $code(st3) > code(st0)$, $code(st6) > code(st0)$ and $code(st7) > code(st0)$ are introduced. The companion face constraints are trivial.

- Minimization of the slice of next state *st*4.
  The onset is:

```
1 10000000 010000000
1 00100000 010000001
```

  The dcset is:

```
- 01010010 010000000
0 00100000 000000001
- 00001101 111111111
- 11111111 101111110
```

  The minimized expanded cover is:

```
- 00101101 101111111
1 11111111 111111110
```

The minimized reduced cover is:

```
- 00100000 000000001
1 11111111 010000000
```

The constraints $code(st5) > code(st4)$, $code(st6) > code(st4)$ and $code(st7) > code(st4)$ are introduced. The companion face constraints are trivial.

- Minimization of the slice of next state $st1$.
The onset is:

```
0 00010000 001000000
0 00000100 001000001
```

The dcset is:

```
- 01000010 001000000
- 00100000 000000001
1 00010110 001000000
1 00000100 111111111
- 00001001 111111111
- 11111111 110111110
```

The minimized expanded cover is:

```
- 00101101 110111111
- 01011111 111111110
```

The minimized reduced cover is:

```
- 00000100 000000001
- 00010100 001000000
```

The constraints $code(st5) > code(st1)$ and $face(st2, st3)$ are introduced.

- Minimization of the slice of next state $st2$.
The onset is:

```
onset
0 01000000 000100000
0 00001000 000100001
```

The dcset is:

```
1 01011110 000100000
- 00100100 000000001
1 00001100 111111111
- 00000010 000100000
- 00000001 111111111
- 11111111 111011110
```

The minimized expanded cover is:

```
- 00101101 111011111
- 01001011 111111110
```

The minimized reduced cover is:

```
- 00001000 000000001
- 01001000 000100000
```

The constraints $code(st6) > code(st2)$ and $face(st4, st5)$ are introduced.

- Minimization of the slice of next state $st5$.
  The transitions of this slice are realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st5) = code(st4) \vee code(st1)$.

- Minimization of the slice of next state $st3$.
  One of the two transitions of this slice is realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st3) = code(st1) \vee code(st2)$. Consider the remaining transition.
  The onset is:

```
0 00000001 000001001
```

The dcset is:

```
1 01000011 000001000
- 00101100 000000001
1 00001001 111111111
- 00000010 000001000
- 11111111 111110110
```

The minimized expanded cover is:

```
- 00000001 111111111
```

The minimized reduced cover is:

```
- 00000001 000001001
```

The constraint $code(st7) > code(st3)$ is introduced.

- Minimization of the slice of next state $st6$.
  The transitions of this slice are realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st6) = code(st4) \vee code(st2)$.

- Minimization of the slice of next state $st7$.
  One of the two transitions of this slice is realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st7) = code(st4) \vee code(st1) \vee code(st2)$. Consider the remaining transition.
  The onset is:

```
onset
1 00000010 000000010
```

The dcset is:

```
- 00101101 000000001
1 00000001 111111111
- 11111111 111111100
```

The minimized expanded cover is:

```
1 00000011 111111110
```

The minimized reduced cover is:

```
1 00000010 000000010
```

No other constraint is introduced.

- Minimization of the slice of the proper binary outputs.
  The onset is:

```
- 00101101 000000001
- 00100000 000000001
- 00000100 000000001
- 00001000 000000001
```

The dcset is:

```
- 11111111 111111110
```

The minimized expanded cover is:

```
- 00101101 111111111
```

The minimized reduced cover is:

```
- 00101101 000000001
```

The constraint $face(st1, st5, st3, st7)$ is introduced.

- The final symbolic cover is:

```
- 11111111 100000000
1 11111111 010000000
- 00010111 001000000
- 01001011 000100000
- 00000001 000001001
1 00000010 000000010
- 00101101 000000001
```

Codes of the states that satisfy the previous constraints are: $code(st0) = 000$, $code(st4) = 010$, $code(st1) = 100$, $code(st2) = 001$, $code(st5) = 110$, $code(st3) = 101$, $code(st6) = 011$, $code(st7) = 111$. The minimized encoded symbolic cover is:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001
```

The minimized encoded FSM is:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001
```

10 Experimental Results

The algorithms described have been implemented in a program, called ESP_SA, that is built on top of ESPRESSO. We report one set of experiments that compare the results of performing state assignments of FSM's with ESP_SA and NOVA, a state-of-art tool. The FSM's come from the MCNC suite and other benchmarks. The experiments were run on a DEC 3100 work-station. Our program ESP_SA uses a library of routines described in [11] to check encodeability of constraints and produce minimum-length codes that satisfy them. Table 1 shows the statistics of the FSM's used. The statistics include the number of states, proper inputs and proper outputs, together with the number of symbolic produc-terms ("#cubes") of the original FSM description, the cardinality of a minimized 1-hot encoded cover of the FSM ("#1-hot") and the number of bits for an encoding of minimum length ("#bits").

In Table 2, data are reported for runs of ESP_SA with three different ordering options ("ord1", "ord2", "ord2n"). For each run, "#scubes" indicates the number of cubes of the cover of symbolic cubes obtained by ESP_SA, after encoding with the codes found by ESP_SA and minimization with ESPRESSO; "#cubes" indicates the number of cubes after encoding the original cover with the codes found by ESP_SA and minimization with ESPRESSO; "#bits" indicates the length of the codes found by ESP_SA.

In Table 3, some data related to the best of the three previous runs are reported. Under "cover", "#incomp" gives the number of pairwise incompatibilities in the final step of computing codes the satisfy the encoding constraints, and "size" gives the number of prime dichotomies. Under "calls", "#esp" gives the number of calls to ESPRESSO and "#check" gives the number of encodeability checks. Under "CPU times(sec.)", "order" gives the time in seconds for the ordering routine, "symb" gives the time for symbolic minimization, not including the time spent by the encodeability routines that is reported under "check(codes)" ("codes" is the time spent for finding the codes satisfying the constraints at the end), while "total" sums up all the contributions.

Table 4 compares the results of ESP_SA with those of NOVA, providing the number of cubes of the minimized encoded FSM ("#cubes") and the code-length ("#bits"). Of the results by NOVA, it is reported the one the minimizes the final cover cardinality (under the heading "NOVA(min.#cubes)") and the one that minimizes the final cover cardinality, if the code-length is kept to the minimum one, i e. to the logarithm of the number of states (under the heading "NOVA(min.#bits)").

A conclusion from the experiments is that ESP_SA improves on average at least 10% the number of product-terms of the best result of NOVA. The gain is more noticeable on hard examples like *s1*, *s1a*, *sand*,

styr, *tbk_m*. Since ESP_SA is heuristic it does not beat NOVA on all benchmarks, a noticeable poor performance being *dk16*. The ordering scheme is a main factor influencing the quality of the final results. Experiments show that the program is very sensitive to it. Our ordering scheme is static (i.e., decided at the beginning of the run) and it uses a limited amount of information on the affinity between the onsets of the next states of the original FSM. *Ad hoc* orders for various examples may improve strongly the quality of final results. For instance in the case of *s1a* we found a solution with 9 bits and 52 cubes, vs. 9 bits and 60 cubes produced by the standard options of ESP_SA. It is not simple to design an ordering algorithm that is fast and produces good orders across all examples. The strategy of ESP_SA to explore the space of all possible encodings can be seen as a two-layered mechanism: an ordering scheme and, once an ordering is found, the detection of profitable encoding constraints that yield good codes. The latter part is handled robustly by the program, as witnessed by the fact that the cardinality of the minimized encoded cover obtained by symbolic minimization is very close to the one of the minimized encoded original FSM. The ordering part instead is not so robust.

Notice also that the best result of NOVA and similar well-tuned existing tools is usually obtained by exercising a large number of different options. For instance, for NOVA all rotations of a given computed set of codes are tried and the best one is kept. Another problem is the size of the final unate table to compute codes of minimum length that satisfy the encoding constraints. An example like *planet* could not be completed because the number of columns of the table exceeded 50,000 and so was beyond the practical capability of the table solver available in ESPRESSO. As a last observation the length of the final codes is usually larger than the one obtained by NOVA. This is due to the fact that the search algorithm targets as a cost function the number of cubes and does not control directly the code length and to the fact that we stop the final unate covering step to the first solution (to save computing time).

11 Conclusions

We have presented a symbolic minimization procedure that advances theory and practice with respect to the seminal contribution in [6]. The algorithm described here is capable of exploring minimal symbolic covers by using face, dominance and disjunctive constraints to guarantee that they can be mapped into encoded covers. The treatment of disjunctive constraints is a novelty of this work. Conditions on the completness of sets of encoding constraints and a bridge to disjunctive-conjunctive constraints (presented in [3]) are given.

A key feature of the algorithm is that it keeps as invariant the property that the minimal symbolic cover under construction is encodeable, by means of efficient procedures that check encodeability of the encoding constraints induced by a candidate cover. Therefore this synthesis procedure has predictive power that precedent tools lacked, i.e. the cardinality of the cover obtained by symbolic minimization and of the cover obtained by replacing the codes in the initial cover and then minimizing with ESPRESSO are very close. Experiments show a set of hard examples where this procedure improves on the best results of state-of-art tools.

A direction of future investigation is to explore more at large the solution space of symbolic covers by escaping from local minima using some iterated expansion and reduction scheme, as it is done in ESPRESSO. Currently the algorithm builds a minimal symbolic cover, exploring basically a neighborhood of the original FSM cover. Another issue requiring more investigation is how to predict somehow the final code-length while building a minimal symbolic cover, to trade-off product-terms vs. encoding length. Finally it would be of interest to add the capability to detect disjunctive-conjunctive constraints. This requires extending the mechanism for symbolic oring and updating the library of routines used to check encodeability of constraints.

example	#states	#inputs	#outputs	#cubes	#1-hot	#bits
bbara	10	4	2	60	34	4
bbsse	16	7	7	56	30	4
bbtas	6	2	2	24	16	3
beecount	7	3	4	28	12	3
cse	16	7	7	91	55	4
dk14	7	3	5	56	25	3
dk15	4	3	5	32	17	2
dk16	27	2	3	108	55	5
dk17	8	2	3	32	20	3
dk27	7	1	2	14	10	3
dk512	15	1	3	30	21	4
donfile	24	2	1	96	24	5
ex1	20	9	19	138	44	5
ex2	19	2	2	72	38	5
ex3	10	2	2	36	21	4
ex4	14	6	9	21	21	4
ex5	9	2	2	32	19	4
ex6	8	5	8	34	23	3
ex7	10	2	2	36	20	4
keyb	19	7	2	179	77	5
kirkman	16	12	6	370	61	4
lion9	9	2	1	25	10	4
maincont	16	11	4	40	27	4
mark1	15	5	16	22	19	4
master	15	23	31	86	79	4
opus	10	5	6	22	19	4
pma	24	8	8	73	43	5
ricks	13	10	23	51	33	4
s1	20	8	6	107	92	5
s1a	20	8	6	107	92	5
s8	5	4	1	20	14	3
sand	32	9	11	184	114	5
saucier	20	9	9	32	30	5
shiftreg	8	1	1	16	9	3
styr	30	9	10	166	114	5
tbk_m	16	6	3	1024	92	4
tma	20	7	6	44	32	5
train11	11	2	1	25	11	4

Table 1: Statistics of FSM's

example	ord1			ord2			ord2n		
	#scubes	#cubes	#bits	#scubes	#cubes	#bits	#scubes	#cubes	#bits
bbara	27	27	5	31	28	6	24	23	5
bbsse	31	31	6	26	26	7	24	24	8
bbtas	10	9	3	10	10	4	11	11	4
beecount	10	10	4	12	12	6	10	10	4
cse	58	55	7	42	42	5	42	42	5
dk14	26	27	4	27	27	4	26	26	4
dk15	17	17	4	17	17	4	17	17	4
dk16	64	61	12	59	59	13	60	57	12
dk17	19	17	5	19	17	5	19	19	6
dk27	7	7	5	9	8	5	7	7	5
dk512	19	18	7	18	16	9	15	15	8
donfile	26	25	12	25	25	13	26	25	12
ex1	37	36	9	42	40	9	42	40	9
ex2	34	35	10	36	32	12	30	31	9
ex3	20	18	6	21	18	7	17	17	6
ex4	14	14	5	15	15	5	14	14	5
ex5	17	16	9	18	18	6	14	13	4
ex6	25	25	4	26	25	4	26	25	4
ex7	20	20	8	20	18	4	15	15	5
keyb	75	65	9	45	46	6	47	47	5
kirkman	102	74	11	54	53	10	55	54	9
lion9	8	7	6	9	8	5	9	8	6
maincont	12	12	8	14	14	7	13	13	9
mark1	17	18	6	17	17	6	17	17	6
master	69	68	5	70	68	5	70	69	5
opus	15	15	4	15	15	4	15	15	4
pma	40	37	9	42	42	7	42	42	7
ricks	29	29	4	30	30	4	30	30	4
s1	62	59	6	49	44	7	49	44	7
s1a	62	61	11	61	61	13	60	60	9
s8	11	9	4	11	10	4	11	10	4
sand	96	95	9	86	86	12	91	93	9
saucier	24	23	6	25	24	8	22	22	6
shiftreg	4	4	3	4	4	3	4	4	3
styr	87	89	10	na	na	na	na	na	na
tbk_m	102	83	15	59	58	8	52	51	7
tma	32	31	6	29	29	6	29	29	6
train11	10	9	5	13	12	6	10	9	5

Table 2: Results of ESP_SA with different ordering heuristics

example	cover		calls		CPU times (sec.)			
	#incomp	size	#esp	#check	order	symb	check(codes)	total
bbara	38	8	96	173	7.4	12.9	0(0)	20
bbsse	458	168	155	46	41.6	10	4(0)	56
bbtas	9	4	30	80	1	0	0(0)	2
beecount	104	15	66	55	2	1	0(0)	4
cse	1170	629	155	80	99	45	19(7)	145
dk14	316	186	38	29	7	2	1(0)	11
dk15	256	238	17	19	1	0	1(0)	3
dk16	14578	4710	799	2841	284	5166	1574(203)	7026
dk17	30	14	47	24	5	1	0(0)	7
dk27	1	2	38	30	0	0	0(0)	1
dk512	1	2	138	140	11	32	2(0)	46
donfile	17929	2701	432	1254	98	2044	143(117)	2286
ex1	2282	815	410	542	794	759	39(10)	1592
ex2	3934	826	212	1161	37	1493	28(21)	1559
ex3	148	14	68	52	3	3	0(0)	7
ex4	1048	359	122	22	15	5	3(2)	24
ex5	285	27	57	46	3	2	0(0)	6
ex6	219	16	47	29	8	1	0(0)	11
ex7	352	34	68	43	6	3	0(0)	10
keyb	967	1094	212	71	129	76	32(27)	239
kirkman	716	84	155	1164	1385	1187	172(3)	2746
lion9	26	7	86	75	5	2	0(0)	8
maincont	363	55	194	196	34	48	2(0)	85
mark1	443	112	247	155	44	42	2(0)	89
master	281	300	327	315	271	240	18(3)	530
opus	312	151	68	18	6	1	0(0)	9
pma	11381	7455	683	958	411	1174	580(558)	2166
ricks	353	408	107	60	53	19	7(3)	80
s1	969	288	233	92	253	126	10(4)	390
s1a	225	67	317	639	151	661	13(2)	826
s8	6	4	46	103	1	1	0(0)	3
sand	1545	860	799	1219	412	3374	74(9)	3861
saucier	1401	3340	256	124	45	99	157(156)	301
shiftreg	3	3	47	54	1	1	0(0)	2
styr	6581	17890	1145	1416	2136	4599	1503(1468)	8420
tbk_m	95	20	155	588	294	102	4(0)	401
tma	2221	1221	233	103	68	19	14(13)	103
train11	156	23	105	86	9	5	0(0)	15

Table 3: Measured parameters of ESP_SA

example	ESP_SA		NOVA(min.#cubes)		NOVA(min.#bits)	
	#cubes	#bits	#cubes	#bits	#cubes	#bits
bbara	23	5	24	4	24	4
bbsse	24	8	27	5	29	4
bbtas	9	3	8	3	8	3
beecount	10	4	10	3	10	3
cse	42	5	44	5	45	4
dk14	26	4	22	4	25	3
dk15	17	4	16	3	17	2
dk16	57	12	50	8	52	5
dk17	17	5	17	4	17	3
dk27	7	5	7	3	7	3
dk512	15	8	17	4	17	4
donfile	25	12	24	14	28	5
ex1	36	9	37	6	44	5
ex2	31	9	26	6	27	5
ex3	17	6	17	4	17	4
ex4	14	5	14	4	14	4
ex5	13	4	14	4	14	4
ex6	25	4	23	4	25	3
ex7	15	5	15	4	15	4
keyb	46	6	47	6	48	5
kirkman	53	10	52	6	77	4
lion9	7	6	8	4	8	4
maincont	12	8	16	4	16	4
mark1	17	6	17	4	17	4
master	68	5	71	4	71	4
opus	15	4	15	4	15	4
pma	37	9	41	5	41	5
ricks	29	4	39	4	30	4
s1	44	7	63	5	63	5
s1a	60	9	65	5	65	5
s8	9	4	9	3	9	3
sand	86	12	96	5	96	5
saucier	22	6	25	5	25	5
shiftreg	4	3	4	3	4	3
styr	87	10	94	5	93	6
tbk_m	51	7	53	4	53	4
tma	29	6	33	5	33	5
train11	9	5	9	4	9	4

Table 4: Comparison of FSM's encodings for two-level implementation

References

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

[2] M. Ciesielski, J-J. Shen, and M. Davio. A unified approach to input-output encoding for FSM state assignment. *The Proceedings of the Design Automation Conference*, pages 176–181, June 1991.

[3] S. Devadas and R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, pages 13–27, January 1991.

[4] E.I.Goldberg. Matrix formulation of constrained encoding problems in optimal PLA synthesis. *Preprint No. 19, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1993.

[5] E.I.Goldberg. Face embedding by componentwise construction of intersecting cubes. *Preprint No. 1, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1995.

[6] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, October 1986.

[7] G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, July 1985.

[8] G. De Micheli, T. Villa, and A. Sangiovanni-Vincentelli. Computer-aided synthesis of PLA-based finite state machines. In *The Proceedings of the International Conference on Computer-Aided Design*, September 1983.

[9] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6:727–750, September 1987.

[10] A. Saldanha and R. Katz. PLA optimization using output encoding. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1988.

[11] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Satisfaction of input and output encoding constraints. *IEEE Transactions on Computer-Aided Design*, 13:589–602, May 1994.

[12] G. Saucier, C. Duff, and F. Poirot. State assignment using a new embedding method based on an intersecting cube theory. In *The Proceedings of the Design Automation Conference*, 1989.

[13] Y. Su and P. Cheung. Computer minimization of multi-valued switching functions. *IEEE Transactions on Computers*, September 1972.

[14] J. Tracey. Internal state assignment for asynchronous sequential machines. *IRE Transactions on Electronic Computers*, August 1966.

[15] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment for optimal two-level logic implementations. In *IEEE Transactions on Computer-Aided Design*, pages 905–924, September 1990.

[16] S. Yang and M. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, January 1991.