Int. J. Appl. Math. Comput. Sci., 2007, Vol. 17, No. 1, 113–128
DOI: 10.2478/v10006-007-0011-6

amcs

FSM ENCODING FOR BDD REPRESENTATIONS[‡]

WILSIN GOSTI [*], TIZIANO VILLA [**,***], ALEX SALDANHA [****],
ALBERTO L. SANGIOVANNI-VINCENTELLI [***,†]

[*] Cadence Design Systems, 2655 Seely Avenue, San Jose, CA 95134

[**] DI, Universita' di Verona, Ca' Vignal 2, Strada Le Grazie 15, 37134 Verona, Italy
e-mail: tiziano.villa@univr.it

[***] PARADES, Via S.Pantaleo,66 , 00186 Roma, Italy

[****] Exponential Interactive, Inc., Emeryville, CA 94608, USA

[†] Department of Electrical Engineering and Computer Sience
University of California, Berkeley, CA 94720, USA

We address the problem of encoding the state variables of a finite state machine such that the BDD representing the next state function and the output function has the minimum number of nodes. We present an exact algorithm to solve this problem when only the present state variables are encoded. We provide results on MCNC benchmark circuits.

Keywords: binary decision diagram, encoding, finite state machine, logic synthesis, formal verification, logic representation

1. Introduction

Reduced Ordered Binary Decision Diagrams (ROBDDs or simply BDDs) are a data structure used to efficiently represent and manipulate logic functions. They were introduced by Bryant (1986). Since then, they have played a major role in many areas of computer aided design, including logic synthesis, simulation, and formal verification.

The size of a BDD representing a logic function depends on the ordering of its variables. For some functions, BDD sizes are linear in the number of variables for one ordering while being exponential for another (Bryant, 1992). Many heuristics have been proposed to find good orderings, e.g., the sifting dynamic reordering algorithm (Rudell, 1993). An exact reordering based on lower bounds was proposed in (Drechsler *et al.*, 2000). An output-efficient algorithm was proposed to realize a reordering transformation (Bern *et al.*, 1996). Other techniques to reduce the BDD size include linear transformations for the variables of the represented function (Gunther and Drechsler, 1998) and a combination of linear transformations with sifting (Meinel *et al.*, 2000).

BDDs can also be used to represent characteristic functions of transition relations of finite state machines (FSMs). In this case, the size of BDDs depends not only on variable ordering, but also on state encoding. Meinel and Theobald (1999; 1996b) studied the effect of state encoding on autonomous counters. They analyzed three different encodings: the standard minimum-length encoding, which gives a lower bound of $5n - 3$ internal nodes for an n-bit autonomous counter, the Gray encoding, which gives a lower bound of $10n - 11$ internal nodes, and a worst-case encoding, which gives an exponential number of nodes in n.

The problem of reducing the BDD size of an FSM representation by state encoding is motivated by applications in logic synthesis and verification. As regards the synthesis, BDDs can be used as a starting point for logic optimization. An example is their use in timed Shannon circuits (Lavagno *et al.*, 1995), where the circuits derived are reported to be competitive in the area and often significantly better in power. BDDs permit to combine logic

[‡] This research was supported in part by UC Micro 532419.

synthesis and technology mapping for multiplexor-based circuits (Gunther and Drechsler, 2000), which can be realized efficiently with Pass Transistor Logic (PTL) (Buch *et al.*, 1997), realizing also a tighter integration between the logical and physical representations, and so a better layout (Macchiarulo *et al.*, 2001). Moreover, it is easy to generate fully testable circuits starting from functions described by BDDs (Drechsler *et al.*, 2004). Therefore, one would like to derive the smallest BDD with the hope that it leads to a smaller circuit derived from it. Regarding verification, re-encoding has been applied successfully to ease the comparison of "similar" sequential circuits (Cabodi, 1995).

In this paper, we look into the problem of finding an optimum state encoding which minimizes the BDD that represents a finite state machine. We call this problem the *BDD encoding problem*. To the best of our knowledge, this problem has never been addressed before. The work that is related to this paper is from Meinel and Theobald. In the effort to find a good re-encoding of the state variables to reduce the number of BDD nodes, Meinel and Theobald proposed in (Meinel and Theobald, 1996a) a dynamic re-encoding algorithm based on XOR-transformations. Although a little slower than the sifting algorithm, their technique was able to reduce the number of nodes in BDDs even in cases when the sifting algorithm could not.

1.1. An Example of Good Encoding. To motivate the effectiveness of encoding, we consider the functions (nodes) f and g shown in Fig. 1. They map $\{0, 1, \ldots, 7\}$ to $\{0, 1, \ldots, 8\}$, and can be regarded as next state functions where the present state variable is v and the next state values are the range. If we encode v as $e_1(0) =$

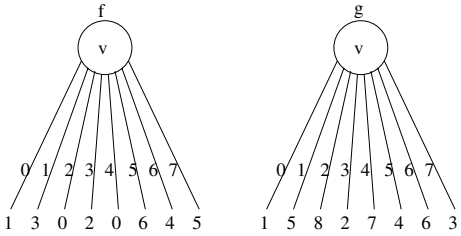


Fig. 1. Multi-valued functions f and g.

$010, e_1(1) = 100\ e_1(2) = 011, e_1(3) = 001$, $e_1(4) = 000, e_1(5) = 110$, $e_1(6) = 111, e_1(7) = 101$, with the ordering b_2, b_1, b_0 we get the BDD (i.e., part of a BDD) shown in Figure 2 with 14 nodes. *No* reordering will reduce the number of BDD nodes for this encoding.

But if we encode v as $e_2(0) = 010, e_2(1) = 100$, $e_2(2) = 001, e_2(3) = 011$, $e_2(4) = 000, e_2(5) = 111$, $e_2(6) = 101, e_2(7) = 110$, the BDD that we get has 10 nodes. Figure 3 shows the binary decision trees representing f and g using this encoding. The BDD is shown in Fig. 4. From this example, we see that state encodings affect the (size of the) BDD size representing the transition relation of an FSM.

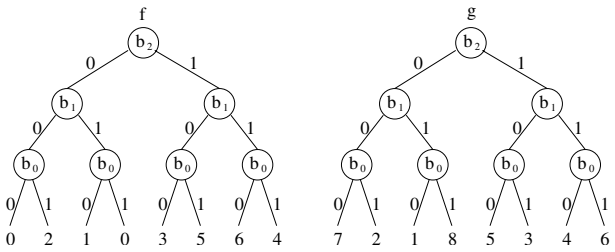


Fig. 2. BDD for f and g using the encoding e_1.

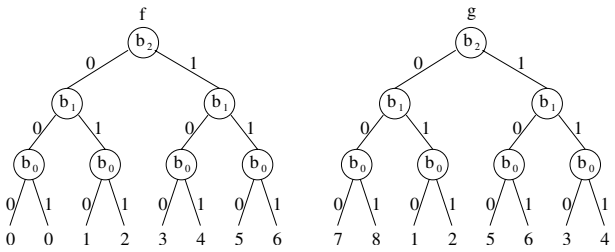


Fig. 3. Binary decision tree for f and g using the encoding e_2.

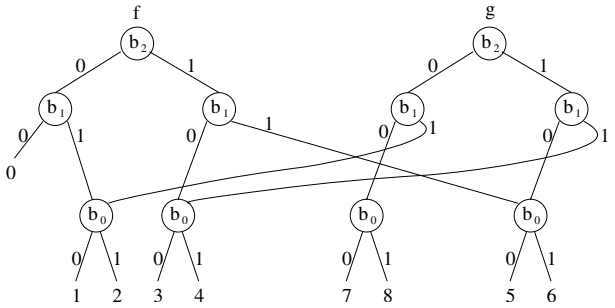


Fig. 4. BDD for f and g using the encoding e_2.

In this paper, we present an exact algorithm to solve the BDD input encoding problem. In Section 2 we give the definitions of FSMs and their BDD representation. We present our exact algorithm of the BDD input encoding problem in Section 3. The experimental results are shown in Section 4. We conclude in Section 5. For clarity purposes, all proofs and algorithms are included in the Appendix.

A restricted version of this material was presented at GLS98 (Gosti *et al.*, 1998).

2. Definitions and Terminology

We review briefly finite state machines and BDDs. We also describe an outline of the simulated annealing algorithm.

2.1. FSMs and Their BDD Representations.

Definition 1. A *finite state machine (FSM)* is a quintuple $(Q, I, O, \delta, \lambda)$ where Q is a finite set of states, I is a finite set of input values, O is a finite set of output values, δ is the next state function defined as $\delta : I \times Q \mapsto Q$, and λ is the output function defined as $\lambda : I \times Q \mapsto O$.

An FSM is said to be incompletely specified (ISFSM) if for some input value and present state combination (i, p), the next state or the output is not specified; otherwise, it is said to be completely specified (CSFSM). For an ISFSM, if the next state of (i, p) is not specified, then any state can become the next state. If the output of (i, p) is not specified, then any output can become the output. In the sequel, we will deal only with CSFSMs. ISFSMs will be converted to CSFSMs by selecting a next state and/or an output value for an unspecified (i, p). We assume that the inputs and outputs are given in binary forms, and the state variables are given in symbolic forms, i.e., multi-valued.

The next state and output functions of an FSM can be simultaneously represented by a characteristic function $T : I \times Q \times Q \times O \mapsto B$, where each combination of an input value and a present state is related to a combination of a next state and an output value.

Assume that we have an encoding of the states that uses s bits. Let $p_{s-1}, p_{s-2}, \ldots, p_0$ and $n_{s-1}, n_{s-2}, \ldots, n_0$ be the present and next state binary variables. Then for the next state n_i, there is a next state function $n_i = \delta_i(p_{s-1}, p_{s-2}, \ldots, p_0)$. Next state and output functions can be represented using BDDs. We call this representation the *functional representation*. We can also represent an FSM using BDDs by representing its characteristic function T. We call it the *relational representation*.

3. Exact Algorithm

In this work, we present an exact algorithm for a restricted version of the encoding problem, namely, the *BDD input encoding problem*. We do not address the *BDD output encoding problem* where an FSM is represented in a functional representation and the state variable, and hence the next state function, ought to be encoded. In the section, we state a formal definition of the input encoding problem and provide an exact algorithm to solve it.

3.1. BDD Input Encoding Problem.

We define the BDD input encoding problem as follows:

Input:

1. A set of symbolic values, $D = \{0, 1, 2, \ldots, |D| - 1\}$, where $|D| = 2^s$, for some $s \in \mathcal{N}$; a symbolic variable, v, taking values in D.
2. A set of symbolic values, $R = \{0, 1, 2, \ldots, |R| - 1\}$.
3. A set of functions, $F = \{f_0, f_1, f_2, \ldots, f_{|F|-1}\}$, where $f_i : D \mapsto R$.
4. A set of s binary variables represented as $B = \{b_{s-1}, b_{s-2}, \ldots, b_0\}$.

Output:

The bijection $e : D \mapsto \mathbf{B}^s$ such that the size of the BDD representing $e(F)$ is minimum, where $e(F) = \{e(f_0), e(f_1), \ldots, e(f_{|F|-1})\}$, and $e(f_i) : \mathbf{B}^s \mapsto R$. We call e an encoding of v and of F interchangeably. We call e_{opt} an encoding e that minimizes the size of the BDDs of $e(F)$, i.e., $\forall_e |e_{\text{opt}}(F)| \leq |e(F)|$, where $|e(F)|$ is the number of nodes of the multi-rooted BDD representing $e(F)$.

In other words, the problem is about finding an encoding of a multi-valued variable v such that the multi-rooted multi-terminal BDD representing a set of multi-valued functions of v has a minimum number of nodes. Diagrammatically, a multi-valued function f of v is represented as a single level multi-way tree. The root is labeled with v. A mapping $f(d) = r$ is represented by an edge labeled with d going from the root to a leaf node labeled with r. We call this diagram a *Single Level Multi Valued Tree (SLMVT)*. For clarity purposes, the leaf nodes are replaced by their labels in all figures. Examples of SLMVTs are the functions f and g shown in Fig. 1.

With this formulation, we model the process of encoding the present state variables of a completely specified finite state machine (CSFSM) when the characteristic function of the CSFSM is represented by a BDD. We assume that the state variables are not interleaved in the variable ordering. In this respect, $f_i(d_i) = r_i$ represents the state transition from the present state d_i to the next state r_i under a proper input combination that causes this transition. Essentially, we cut across the BDDs representing the characteristic functions of CSFSMs and only look at the present state variables. Therefore, although encoded BDDs are actually multi-terminal BDDs (MTBDDs), we still refer to them as BDDs. It is worth mentioning here that our formulation can also be applied to other BDD encoding problems, like MDD encoding and BDD re-encoding.

We assume that BDDs are represented by their *true* edges. We do not model yet the complemented edges.

3.2. Characterization of BDD Node Reductions.

We first outline our strategy to find an optimum encoding. Assume that we have binary decision trees representing an instance of the BDD input encoding problem. The BDD representing this instance of the problem is obtained by applying the two BDD reduction rules, i.e.,

Rule 1: eliminating nodes with the same *then* and *else* children, and

Rule 2: eliminating duplicate isomorphic subgraphs.

We would like to characterize the conditions in the original problem in which these rules can be applied. To apply these reduction rules, there must exist some isomorphic subgraphs. This means that there is a set of values in R where each value is incident to more than one edge in the SLMVT representing F. So, from F, D, and R, we can group those edges into sets. Each group represents a possible reduction. However, there are many isomorphic subgraphs, and applying a reduction to one of them may interfere with applying a reduction to another. We therefore find the sets whose reductions do not interfere with each other and yield the largest possible total reduction. Once these are found, we encode each set in such a way that it occupies a subtree in the BDD representing $e(F)$.

With this characterization, we explain why encoding e_2 is better than encoding e_1 for the example in Fig. 1.

1. $f(2) = f(4) = 0$. One node is reduced when 2 is encoded as `001` and 4 as `000`.
2. $f(0) = g(0) = 1$ and $f(3) = g(3) = 2$. Encoding 0 as `010` and 3 as `011` allows to share one node between f and g, i.e., the subtree identified by the cube `01-`.
3. $f(1) = g(7) = 3$ and $f(6) = g(5) = 4$. Encoding 1 as `100`, 6 as `101`, 7 as `110`, and 5 as `111` permits to share one node between f and g, i.e., the subtree of encoded f identified by `10-` or the subtree of encoded g identified by `11-`.
4. $f(7) = g(1) = 5$ and $f(5) = g(6) = 6$. Using the same encoding as in 3 permits to share one node between f and g, i.e., the subtree of encoded f identified by `11-` or the subtree of encoded g identified by `10-`.

Equivalently, encoding e_2 allows us to apply two BDD reduction rules, namely, eliminating a node with the same children and eliminating isomorphic subgraphs; while encoding e_1 does not.

3.2.1. Sibling and Isomorphic Sets. The objective of this section is to identify all cases where Rule 1 and Rule 2 can be applied. For that purpose, we define two sets, the *sibling set* and the *isomorphic set*. Intuitively, we are trying to capture in the sibling sets the conditions in which Rule 1 can be applied, and in the isomorphic sets the conditions in which Rule 2 can be applied. Informally, each element of a sibling set S is a pair (l^0, l^1) where l^0 and l^1 are ordered sets of symbolic values that can be encoded so that they share an isomorphic subgraph and the isomorphic subgraph is both the *then* child and the *else* child of a node (i.e., the only child). An isomorphic set I is a collection of ordered sets l of symbolic values that can be encoded so that all ordered sets share an isomorphic subgraph.

As examples, consider the four SLMVTs shown in Fig. 5. There are 8 edges in each of the four cases shown. All edges not shown are assumed to point to values other than 0 and 1. The variables needed to encode these cases are b_2, b_1, and b_0 in that order. The binary decision trees representing an optimum solution for each case are shown in Fig. 6. The corresponding BDDs are shown in Fig. 7. We show for these examples and for these optimum encodings the relation of isomorphic subgraphs versus sibling and isomorphic sets.

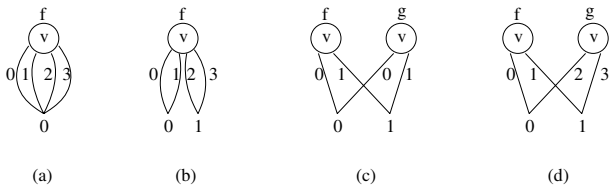


Fig. 5. Examples of sibling and isomorphic sets.

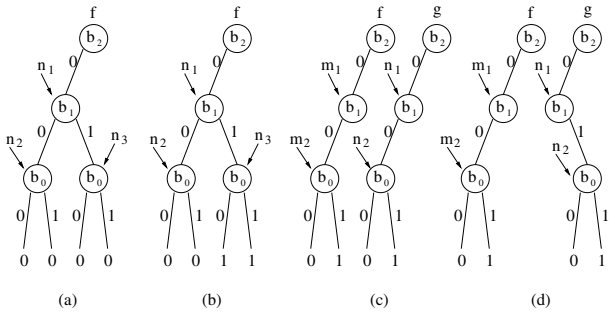


Fig. 6. Binary decision trees for an optimum encoding.

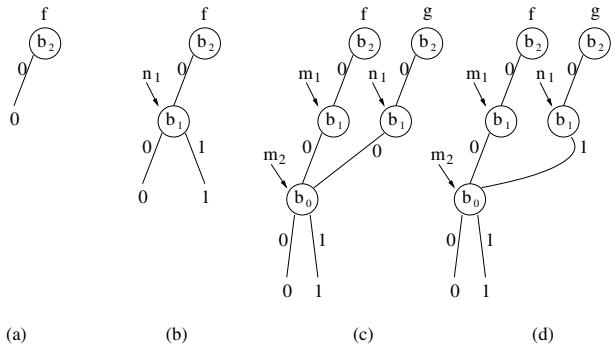


Fig. 7. BDDs for an optimum encoding.

Case (a): Since $f(0) = f(1)$, there is an encoding (e.g., $e(0) =$ `000`, $e(1) =$ `001`) such that in the encoded BDD there is a node (i.e., n_2) whose edges point to the same node (and so they can be reduced). This fact is captured by $S_0 = \{(0_f), (1_f)\}$. For S_1, the technique is similar and proceeds by replacing "$f(0) = f(1)$" with "$f(2) = f(3)$".

Since $f(0) = f(2)$ and $f(1) = f(3)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{001}$, $e(2) = \texttt{010}$, $e(3) = \texttt{011}$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (2_f, 3_f)\}$.

Since $f(0) = f(2)$ and $f(1) = f(3)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{001}$, $e(2) = \texttt{010}$, $e(3) = \texttt{011}$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3) with isomorphic subgraphs (and so they can be reduced) and a node (i.e., n_1) whose edges point to the same node. This fact is captured by $S_2 = \{(0_f, 1_f), (2_f, 3_f)\}$. We would like to point out that although this constraint provides also the information of the previous constraint, it is used differently. I_0 is defined to capture Rule 2 and S_2 is defined to capture Rule 1. Both are needed to calculate correctly the number of nodes that can be later reduced by an encoding.

Case (b): Since $f(0) = f(1)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{001}$) such that in the encoded BDD there is a node (i.e., n_2) whose edges point to the same node (and so can be reduced). This fact is captured by $S_0 = \{(0_f), (1_f)\}$. For S_1, the technique is similar and proceeds by replacing "$f(0) = f(1)$" with "$f(2) = f(3)$".

Since $f(0) = f(1)$ and $f(2) = f(3)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{010}$, $e(2) = \texttt{001}$, $e(3) = \texttt{011}$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3 of Fig. 8) with isomorphic subgraphs (and so they can be reduced). This fact is captured by $I_0 = \{(0_f, 2_f), (1_f, 3_f)\}$.

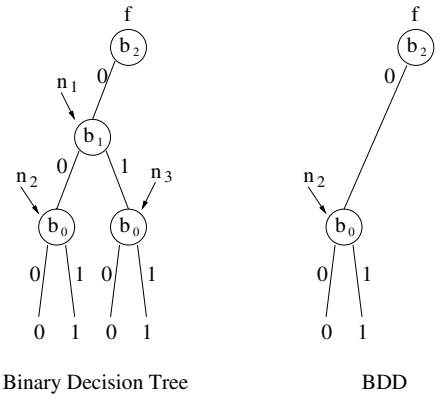


Fig. 8. Alternative optimum encoding for Case (b).

Since $f(0) = f(1)$ and $f(2) = f(3)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{010}$, $e(2) = \texttt{001}$, $e(3) = \texttt{011}$) such that in the encoded BDD there are nodes (i.e., n_2 and n_3 of Fig. 8) with isomorphic subgraphs (and so they can be reduced) and a node (i.e., n_1 of Fig. 8) whose edges point to the same node. This fact is captured by $S_2 = \{(0_f, 2_f), (1_f, 3_f)\}$.

For this case, the encoding induced by S_0 and S_1 cannot satisfy the encoding induced by I_0 and S_2, and vice versa. This leads to our notion of compatibility presented below. An encoding that satisfies S_0 and S_1 is $e(0) = \texttt{000}$, $e(1) = \texttt{001}$, $e(2) = \texttt{010}$, $e(3) = \texttt{011}$, and the BDD is shown in Fig. 7. An encoding that satisfies I_0 and S_2 is $e(0) = \texttt{000}$, $e(1) = \texttt{010}$, $e(2) = \texttt{001}$, $e(3) = \texttt{011}$, and the BDD is shown in Fig. 8.

Case (c): Since $f(0) = g(0)$ and $f(1) = g(1)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{001}$) such that in the encoded BDD there are nodes (i.e., m_2 and n_2) with isomorphic subgraphs (and so they can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (0_g, 1_g)\}$.

Case (d): Since $f(0) = g(2)$ and $f(1) = g(3)$, there is an encoding (e.g., $e(0) = \texttt{000}$, $e(1) = \texttt{001}$, $e(2) = \texttt{010}$, $e(3) = \texttt{011}$) such that in the encoded BDD there are nodes (i.e., m_2 and n_2) with isomorphic subgraphs (and so can be reduced). This fact is captured by $I_0 = \{(0_f, 1_f), (2_g, 3_g)\}$.

For each case above, there are other sibling and isomorphic sets. Later we will show how to construct a complete collection of all sibling and isomorphic sets.

Formally, sibling and isomorphic sets are defined as follows.

Definition 2. A *labeled symbol* d_f has a symbol $d \in D$ and a label $f \in F$. It is the d-edge of the SLMVT representing f. The following notation is defined for d_f: $sym(d_f) = d$, $fn(d_f) = f$, and $val(d_f) = f(d)$.

Definition 3. A *symbolic list* l is an ordered set (or list) of labeled symbols with no duplicate and all labeled symbols have the same function. The k-th element of l is denoted by l_k. The set of all symbols of l is $Sym(l) = \{sym(l_k) \mid 0 \le k \le |l| - 1\}$. The function of l is $Fn(l) = fn(l_0)$.

Definition 4. An *isomorphic set* I is a set of at least two symbolic lists. The j-th element of I is denoted by l^j. I satisfies the following three conditions:

1. The sizes of all symbolic lists of I are the same and they are a power of two, i.e., $\exists a \in \mathcal{N}\ \forall l \in I\ (|l| = 2^a)$.
2. The k-th elements of all symbolic lists of I have the same value, i.e., $\exists r_k \in R\ \forall l \in I\ (val(l_k) = r_k)$, $0 \le k \le |l| - 1$.
3. For any two lists $l', l'' \in I$, either for every index k the symbols of the k-th elements of l' and l'' are the same or the symbol of no element of l' is the same as the symbol of an element of l'', i.e., $\forall l' \in I\ \forall l'' \in I\ ((\forall k\ sym(l'_k) = sym(l''_k)) \vee (\forall i\ \forall j\ sym(l'_i) \neq sym(l''_j)))$, $0 \le i, j, k \le |l'| - 1$.

Definition 5. A *sibling set* S is an isomorphic set with 2 symbolic lists, l^0 and l^1, which satisfies the following conditions:

1. The symbol of no element of l^0 is the same as the symbol of an element of l^1, i.e., $\forall i\ \forall j\ (sym(l_i^0) \neq sym(l_j^1)), 0 \leq i, j \leq |l^0| - 1$.
2. The functions of l^0 and l^1 are the same, i.e., $Fn(l^0) = Fn(l^1)$.

We will see that for every sibling set there is an equivalent isomorphic set. For an instance of the BDD input encoding problem, the set of all sibling sets is denoted as $\mathcal{S}$, and the set of all isomorphic sets is denoted as $\mathcal{I}$.

In the following discussion, the term *tree* is used to denote the encoded binary tree representing a function.

Definition 6. Given an encoding e and a set of symbols $D' \subseteq D$, the tree *spanned* by the codes of the symbols in D' is the tree T whose root is the least common ancestor of the terminal nodes of the codes of the symbols in D'. Furthermore, every leaf of T is the code of a symbol in D'. We say also that D' spans T (denoted by $T_{D'}$).

For example, given the problem in Fig. 1 and the encoding e_2 as in p. 114, the codes for the symbols 0 and 3 span the tree rooted at the parent of the leaves 1 and 2 in Fig. 4.

Proposition 1. *Given a sibling set $S = \{l^0, l^1\}$, there is an encoding e such that the codes of the symbols in $l^0 \cup l^1$ span exactly a tree whose root has a left subtree spanned exactly by the symbols in l^0 and a right subtree spanned exactly by the symbols in l^1, and both subtrees are isomorphic.*

Proposition 2. *Given an isomorphic set $I = \{l^i\}, 0 \leq i \leq |I| - 1$, there is an encoding e such that $\forall l^i \in I$, the symbols in l^i span exactly a subtree T_{l^i} and all T_{l^i}s are isomorphic.*

To illustrate these propositions, we look back to the example in Fig. 1. The $\mathcal{S}$ and $\mathcal{I}$ of this example are

1. $S_0 = \Big\{(2_f), (4_f)\Big\}$.
2. $I_0 = \Big\{(0_f, 3_f), (0_g, 3_g)\Big\}$, $I_1 = \Big\{(3_f, 0_f), (3_g, 0_g)\Big\}$.
3. $I_2 = \Big\{(1_f, 6_f), (7_g, 5_g)\Big\}$, $I_3 = \Big\{(6_f, 1_f), (5_g, 7_g)\Big\}$.
4. $I_4 = \Big\{(7_f, 5_f), (1_g, 6_g)\Big\}$, $I_5 = \Big\{(5_f, 7_f), (6_g, 1_g)\Big\}$.

For now, we focus only on S_0, I_0, I_2, and I_4. Each S_i or I_i justifies why the encoding e_2 is better than the encoding e_1 in this example. In other words, S_0, I_0, I_2, and I_4 contain requirements to find an optimum encoding. Following the above propositions, S_0 states that 2 and 4 should be encoded such that they differ only in b_0 to span a subtree and save a node. I_0 states that 0 and 3 should be encoded such that they differ only in b_0 for symbols in I_0 to span a subtree and share a node. I_2 states not only that 1 and 6 should be encoded such that they differ only in b_0, and similarly for 7 and 5, but also that the value of b_0 of 1 should be the same as the value of b_0 of 7 and the value of b_0 of 6 should be the same as the value of b_0 of 5 for symbols in I_2 to span isomorphic subtrees and share a node. I_4 essentially states the same requirements as I_2. All of these requirements are satisfied by the encoding e_2, but not by e_1.

3.2.2. Finding $\mathcal{S}$ and $\mathcal{I}$. Given an instance of the BDD input encoding problem, we propose an algorithm that finds the sets $\mathcal{S}$ and $\mathcal{I}$. The idea is to find all symbolic lists that are mapped to the same values. We first find a table of symbolic lists corresponding to single values, e.g., for the SLMVT of Fig. 1, the algorithm finds the following: $f(2) = f(4) = 0, f(0) = g(0) = 1$, etc.

Values	Symbolic lists
0	$\{2_f\}, \{4_f\}, \{2_f, 4_f\}$
1	$\{0_f\}, \{0_g\}$
2	$\{3_f\}, \{3_g\}$
⋮	⋮

From this table, we generate larger symbolic lists by combining rows of this table in all possible ways to generate new rows. In these new rows, symbolic lists are generated only if they satisfy Definition 3. For example, from the above table, we get the following:

Values	Symbolic lists
$0, 1$	$\{0_f, 2_f\}, \{0_f, 4_f\}, \{0_f, 2_f, 4_f\}$
$0, 2$	$\{2_f, 3_f\}, \{3_f, 4_f\}, \{2_f, 3_f, 4_f\}$
⋮	⋮
$0, 1, 2$	$\{0_f, 2_f, 3_f\}, \{0_f, 3_f, 4_f\}, \{0_f, 2_f, 3_f, 4_f\}$
⋮	⋮

Then, for each row of the table, we check all possible combinations of the symbolic lists and generate sibling and isomorphic sets according to Definitions 4 and 5.

Because the elements of symbolic lists are ordered, the last step of the algorithm is to permute all the sibling and isomorphic sets generated so far to generate the complete set of all sibling and isomorphic sets. We call this step the *permutation step*.

Having computed $\mathcal{S}$ and $\mathcal{I}$, we can state the following result:

Theorem 1. *Using only $\mathcal{S}$ and $\mathcal{I}$, an optimum encoding e_{opt} can be obtained.*

Theorem 1 says that $\mathcal{S}$ and $\mathcal{I}$ contain all the information that is needed to find an optimum encoding. The question now is to find a subset of $\mathcal{S}$ and $\mathcal{I}$ that corresponds to an optimum encoding. This constitutes the subject of the next section.

3.3. Finding an Optimal Encoding. From now on, the number of nodes that can be reduced is with respect to the complete binary trees that represent the encoded F. Unless otherwise specified, a set means either a sibling set or an isomorphic set.

3.3.1. Compatibility of Sibling and Isomorphic Sets. Sibling sets and isomorphic sets specify that if their symbols are encoded to satisfy the reductions implied, then Rule 1 and Rule 2 can be applied to merge isomorphic subgraphs and reduce nodes. Hence, they implicitly specify the number of nodes that can be reduced, which we refer to as *gains*.

Definition 7. The *gain of a sibling set* S, denoted as *gain*(S), is equal to 1. The *gain of an isomorphic set* I, denoted as *gain*(I), is equal to $(|I|-1)\times(|l^0|-1)$, where $l^0 \in I$.

$\mathcal{S}$ and $\mathcal{I}$ contain the information for all possible reductions. However, not all sets may be selected together. For example, the sibling set $S = \{(1_f), (2_f)\}$ and the isomorphic set $I = \{(2_f, 3_f), (2_g, 3_g)\}$ of Fig. 9 cannot be selected together because S says that the symbols 1 and 2 should span exactly a subtree while I says that the symbols 2 and 3 should span exactly a subtree. Hence, an encoding can only benefit from either S or I. We therefore need to identify which sets may be selected together and which may not. For that purpose we define the notion of *compatibility*.

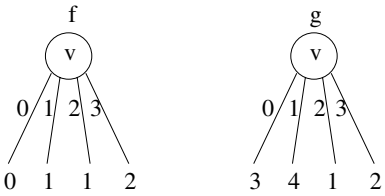


Fig. 9. Example of incompatible sets.

Definition 8. A collection of sets $\mathcal{S}$ and $\mathcal{I}$ are *compatible* if there is an encoding e such that all reductions implied by the sets $S \in \mathcal{S}$ and $I \in \mathcal{I}$ can be applied to the complete binary decision tree yielded by e.

Since both sibling and isomorphic sets are built out of symbolic lists, it is natural to define the compatibility of symbolic lists.

Definition 9. The symbolic lists l' and l'' are *compatible*, which is denoted by $l' \sim l''$, if at least one of the following conditions is true:

1. $Sym(l') \cap Sym(l'') = \emptyset$, i.e., the set of symbols of l' does not intersect the set of symbols of l''.
2. $\exists a \in \mathcal{N}\ \forall k\ sym(l'_k) = sym(l''_{a|l'|+k})$, $0 \le k \le |l'|-1$, $|l''| \ge (a+1)\times|l'|$, i.e., the symbols of l' match exactly the symbols of l'' in the same order starting at the position $a \times |l'|$.
3. $\exists a \in \mathcal{N}\ \forall k\ sym(l''_k) = sym(l'_{a|l''|+k})$, $0 \le k \le |l''|-1$, $|l'| \ge (a+1)\times|l''|$, i.e., the symbols of l'' match exactly the symbols of l' in the same order starting at the position $a \times |l''|$.

Definition 9 says that two lists are compatible if their symbols do not intersect or the symbols of one list are a subset of the symbols of the other starting at a power-of-2 position.

Theorem 2. *If l' and l'' are compatible, then there exists an encoding e such that the symbols of l' span exactly a subtree and so do the symbols of l'', and both subtrees are isomorphic.*

Definition 9 defines pair-wise compatibility between symbolic lists, and the next theorem states how the compatibility among a set of symbolic lists is related to the pair-wise compatibility.

Theorem 3. *If the elements of a set L of symbolic lists are pair-wise compatible, then there exists an encoding e such that the symbols of every symbolic list in L span exactly a subtree.*

Let the symbolic list created by concatenating l^0 and l^1 of a sibling set S be called the *sibling list* of S, denoted by l^S. Then the following are corollaries of Theorem 3:

Corollary 1. *The sibling sets S' and S'' are compatible if $l^{S'}$ is compatible with $l^{S''}$.*

Corollary 2. *A sibling set S and an isomorphic set I are compatible if l^S is compatible with every list of I.*

Corollary 3. *The isomorphic sets I' and I'' are compatible if every list $l' \in I'$ is compatible with every list $l'' \in I''$.*

These theorems and corollaries give us an algorithm to find compatible sets among a collection of sets $\mathcal{S}$ and $\mathcal{I}$. A set of compatible sets is called a *compatible*.

3.3.2. Encoding Sibling and Isomorphic Sets. We begin this section by stating the following corollary, which follows immediately from the theorems and corollaries in the previous section:

Corollary 4. *Given a compatible C, there exists an encoding e such that the reductions implied by all its elements can be applied.*

Definition 10. Let X' be either a sibling or an isomorphic set and X'' another sibling or isomorphic set. Then X' is *contained* in X'' if $\forall l' \in X' \; \exists l'' \in X'' \; (l' \subset l'')$ and X' is *completely contained* in X'' if $\exists l'' \in X'' \; \forall l' \in X' \; (l' \subset l'')$.

For example, the set I_0=$\{(0_f, 1_f), (0_g, 1_g)\}$ is contained, but not completely contained in I_1=$\{(0_f, 1_f, 2_f, 3_f), (0_g, 1_g, 2_g, 3_g)\}$, while S_0=$\{(0_f), (1_f)\}$ is completely contained in I_1. This definition is used for gain calculation and encoding of a compatible. The motivation behind this definition is that the reduction implied by I_0 is covered by I_1, but the reduction implied by S_0 is not. The gain of a compatible that contains only S_0, I_0, and I_1 is equal to the sum of the gains of S_0 and I_1 only.

The algorithm to compute the codes given a compatible can be found in (Gosti *et al.*, 1997). The idea is that starting with a binary tree, we assign codes to the symbols of symbolic lists by a non-increasing length of the symbolic lists. The symbols of a symbolic list are assigned to occupy the largest subtree of codes still available.

3.3.3. Gain of a Compatible. Using the encoding algorithm outlined above, an encoding that permits the reductions implied by all sibling and isomorphic sets of a compatible C can be found. We denote the encoding found using the above algorithm by $e_{\mathrm{alg}}(C)$. Since there may exist many compatibles for an instance of the BDD input encoding problem, we would like to find a compatible implying the largest reduction. Hence, we need to calculate the number of nodes that are reduced by a compatible. We call this quantity the *gain* of a compatible.

Definition 11. The *gain* of a compatible C is equal to the difference in the number of nodes of the binary decision trees representing F and the number of nodes of the BDDs representing F encoded by $e_{\mathrm{alg}}(C)$.

With this definition, the following theorem can be stated:

Theorem 4. *A compatible of maximum gain yields an optimal encoding.*

The task is then to find a compatible with the *largest* gain. Unlike in the example in Fig. 1, where the gain of the compatible formed by S_0, I_0, I_2, and I_4 is simply the sum of the individual gains of its elements, the gain of an arbitrary compatible is more complicated to calculate without actually building the BDDs. If we apply a reduction rule induced by a set, then this reduction causes a merging of two isomorphic subgraphs. For these two subgraphs, there may exist two identical reductions within them. The gain of these two reductions should only be counted once. An example of this kind is shown in Fig. 10. The following sibling and isomorphic sets form a compatible:

$$S_0 = \Big\{(0_f), (1_f)\Big\},$$
$$S_1 = \Big\{(0_g), (1_g)\Big\},$$
$$I_0 = \Big\{(0_f, 1_f), (0_g, 1_g)\Big\},$$
$$I_1 = \Big\{(2_f, 3_f), (2_g, 3_g)\Big\},$$
$$I_2 = \Big\{(0_f, 1_f, 2_f, 3_f), (0_g, 1_g, 2_g, 3_g)\Big\}.$$

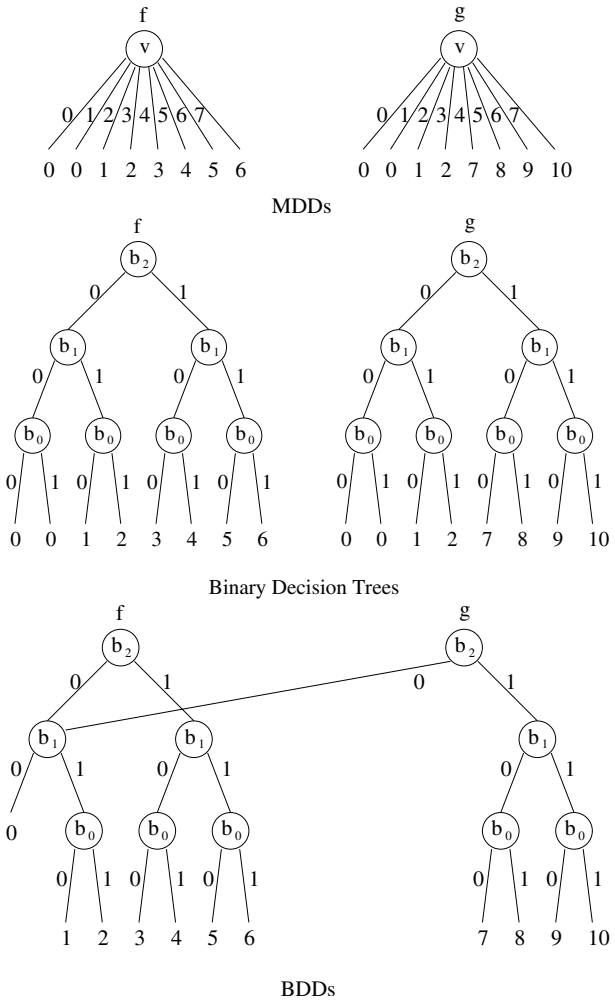


Fig. 10. Gain calculation example.

The gain of this compatible is not the sum of the gains of its elements because the reductions implied by I_0 and I_1 and one of the reductions implied by S_0 and S_1 are subsumed by the reduction implied by I_2. Then the gain of this compatible is equal to $gain(I_2)+gain(S_0)=3+1=4$.

The basic idea is to find sets with the largest lists, calculate their gains, remove all gains of lists that are counted more than once and remove all sets that are subsumed by other sets. The complete algorithm can be found in (Gosti *et al.*, 1997).

Theorem 5. *Given a compatible C,* Gain*(C) computes the gain of C.*

3.4. Maximal Compatibles. Having found all sibling and isomorphic sets, the next task is to find a maximum gain compatible. As it was shown in the previous section, the gain of a compatible is not proportional to the size of the compatible. In other words, the gain of a compatible may be smaller than the gain of another compatible which contains fewer sets. Luckily, we do not have to enumerate all compatibles to find a maximum gain compatible. A maximal compatible, i.e., a compatible where no set can be added while still maintaining compatibility always has a larger or equal gain as any proper subset of the compatible. This means that we only need to find all maximal compatibles. A maximum gain compatible is a maximal compatible that has the largest gain among all maximal compatibles.

3.4.1. 2-CNF SAT Formulation. We find all maximal compatibles by first building a *compatibility graph*. In the following definition, X denotes either a sibling set or an isomorphic set.

Definition 12. A *compatibility graph* $G = (V, E)$ is a labeled undirected graph defined on an instance P of the BDD input encoding problem. There is a vertex x for each set X of P. No other vertices exist. There is an edge $e = (x_1, x_2)$, if and only if X_1 and X_2 are compatible.

As a consequence of this definition, a compatible of P is a clique in G.

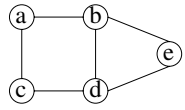
As it has been mentioned above, we need to enumerate all maximal compatibles of P and calculate their gains. Enumerating all maximal compatibles corresponds to finding all maximal cliques of G. The technique we use to find all maximal cliques in G is consists in formulating the problem as a 2-CNF SAT formula ϕ and then finding satisfactory truth assignments of ϕ. The formula ϕ is created as follows: for each unconnected pair of vertices, x_1 and x_2, we create a clause $(\overline{x_1} \vee \overline{x_2})$. A satisfying truth assignment to ϕ is a set of vertices that do not form a clique. Hence a cube of ϕ is also a set of vertices that do not form a clique. It follows that a prime implicant of ϕ contains the minimum number of vertices that do not form a clique. Then the set of vertices that are missing from a prime implicant corresponds to a maximal clique.

In summary, our procedure to find all maximal cliques of G is as follows:

- Formulate the problem into a 2-CNF formula ϕ.
- Pass ϕ to a program (which we call a *CNF expander*) that takes a unate 2-CNF formula and outputs the list of all its prime implicants. We refer the readers to (Brayton *et al.*, 1984) for further reading about unate functions.
- For each prime implicant, the variables that do not appear in it form a maximal clique.

For example, consider the graph G_1 in Fig. 11. The 2-CNF formula ϕ_1 is

$$\phi_1 = (\overline{a} \vee \overline{d})(\overline{a} \vee \overline{e})(\overline{b} \vee \overline{c})(\overline{c} \vee \overline{e}),$$



G_1

Fig. 11. Example of the maximal clique problem.

where each clause corresponds to a pair of unconnected vertices. The prime implicants of ϕ_1 are $\overline{ac}$, $\overline{ab}\overline{e}$, $\overline{bd}\overline{e}$, and $\overline{c}\overline{d}\overline{e}$. The maximal cliques of G_1 are bde (corresponding to $\overline{ac}$), cd (corresponding to $\overline{ab}\overline{e}$), ac (corresponding to $\overline{bd}\overline{e}$), and ab (corresponding to $\overline{c}\overline{d}\overline{e}$).

3.4.2. CNF Expander. The CNF expander used here is the one described in (Villa *et al.*, 1997), Sec. 6.5, originally proposed in (Saldanha *et al.*, 1994). We explain briefly here how the algorithm works and report its pseudo-code in Appendix B.3.

The algorithm first simplifies clauses with a common literal, say a, into a single clause with two terms, a and the concatenation of other literals in the original clauses. After all such clauses have been processed, the reduced formula is expanded by multiplying out two clauses at a time. After each multiplication, a single cube containment operation is performed to eliminate non-prime cubes[1]. After all multiplications are done, the result is a list of all prime implicants of the formula. The following example shows how the algorithm expands the formula of Fig. 11:

$$\phi_1 = (\overline{a} \vee \overline{d})(\overline{a} \vee \overline{e})(\overline{b} \vee \overline{c})(\overline{c} \vee \overline{e}),$$

$$\phi_1 = (\overline{a} \vee \overline{d}\overline{e})(\overline{c} \vee \overline{b}\overline{e}),$$

$$\phi_1 = \overline{ac} \vee \overline{ab}\overline{e} \vee \overline{bd}\overline{e} \vee \overline{c}\overline{d}\overline{e}.$$

Although this algorithm is linear in the number of prime implicants, the number of clauses that need to be created for a graph with n vertices is proportional to n^2. If n is large and the graph is sparse, this number can be very big. We can reduce the amount of memory that the algorithm needs by partitioning the graph into multiple subgraphs. The idea is to invoke the CNF expander k times. A subgraph of size n_i is passed to the i-th invocation, where each n_i is much smaller than n if the graph is sparse. Then the sum of the squares of all these n_is will be much smaller than n^2.

[1] The effect of single cube containment is to remove any cube contained by another cube of the set (Brayton *et al.*, 1984).

Given a graph G, the CNF expander enhanced by partitioning is as follows:

1. Initialize the set of all candidate prime implicants **P** to be an empty set. A candidate prime implicant is an implicant that is either a prime implicant or is covered by a prime implicant of G.
2. Choose a subgraph G_i of G which consists of a smallest degree vertex v and all vertices that are connected to v. By G_i, we essentially look for maximal cliques in G that contain v. Each prime implicant of G_i corresponds to a clique of G.
3. Call the CNF expander with G_i as the input.
4. Perform the logical AND operation of every prime implicant of G_i with the complements of the vertices of the original graph G that are not in G_i, and include them in **P**. This step is to map the Boolean space of G_i into that of the original problem. The mapped terms are the candidate prime implicants. The mapping means that a prime implicant of G_i is a clique in G which does not contain nodes not in G_i.
5. Remove v and all edges that are incident at v from G.
6. If there is more than one vertex left, go to Step 2.
7. Perform a single cube containment operation on **P**.
8. Return **P**.

It is easy to see that **P** contains all the prime implicants of G at the end of the algorithm.

We illustrate this algorithm on the graph G_1 in Fig. 11. We refer to the subgraphs in Fig. 12 as we illustrate this example. By choosing a as the smallest degree vertex, the first subgraph we pass to the CNF expander is G_{11}, which consists of a, b and c. The prime implicants of G_{11} are $\overline{b}$ and $\overline{c}$. The candidate prime implicants are $\overline{bd}\overline{e}$ and $\overline{c}\overline{d}\overline{e}$. We then remove the vertex a and all its edges from G_1. The smallest degree vertex of the new G_1 is c. Since a has been removed, the only neighbor of c is d. Then the next subgraph G_{12} consists of only c and d. Since G_{12} is a complete graph, the only prime implicant of G_{12} is 1. The only candidate prime implicant of G_{12} is therefore $\overline{a}\overline{b}\overline{e}$. The only subgraph left after removing c and its edge is G_{13}. Since G_{13} is also a complete graph, the only prime implicant is also 1 and the candidate prime implicant is $\overline{ac}$. Altogether, the set of all candidate prime implicants is $\{\overline{bd}\overline{e}, \overline{c}\overline{d}\overline{e}, \overline{a}\overline{b}\overline{e}, \overline{ac}\}$, which is also the set of prime implicants of G_1.

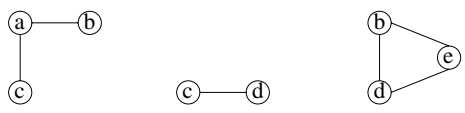


Fig. 12. Example of the partitioned graph for the maximal clique problem.

As a comparison, the CNF expander without partitioning invokes the CNF expander only once, but with four clauses for G_1, whereas the CNF expander with partitioning invokes the CNF expander three times, but with a total of one clause. Also, an approximation algorithm, which is simply the exact algorithm without the *permutation step*, took 165 seconds and 350 seconds of CPU time to find the optimum solutions for the circuits *ellen* and *shiftreg4*, respectively, using the CNF expander with partitioning. Without partitioning, the executions were timed out after some hours of elapsed time.

4. Experimental Results

The experiments were performed on DEC AlphaServer 8400 5/300 with 2Gb of memory on circuits shown in Table 1.

Table 1. Completely specified FSMs.

Name	Number of functions	Domain size
dk15x	7	4
dk17x	4	8
ellen	2	16
ellen.min	2	8
fsync	5	4
mc	6	4
ofsync	5	4
shiftreg4	2	16
shiftreg3	2	8
tav	1	4

The test cases are taken from the MCNC collection (Lisanke, 1989). Column 2 of this table lists the number of distinct state transitions regardless of the primary input combinations. Note that *shiftreg3* is a 3-bit shift register and *shiftreg4* is a 4-bit shift register. Column 3 lists the size of the domain or $|D|$.

Beside the exact algorithm, an experiment with an approximation algorithm, which is the exact algorithm without the *permutation step*, was also done. For comparison purposes, the results of the exact algorithm, the approximation algorithm, and the simulated annealing runs from (Gosti *et al.*, 1997) are shown in Table 2. CPU times are also included in this table. Circuits whose executions were timed out after one hour of CPU time are not listed.

We are aware that these experiments do not witness for the practicality of the exact method; however, they provide exact results for small examples, and may serve to certify the quality of heuristics. We refer to the report (Gosti *et al.*, 1997) for an extensive set of experiments

Table 2. BDD size for completely specified FSMs for simulated annealing and the exact algorithm. (The exact algorithm was run with no permutations for *ellen* and *shiftreg4*).

Name	Number of BDD nodes			CPU time		
	SA	Exact w/ permute	Exact w/o permute	SA SA	Exact w/ permute	Exact w/o permute
dk15x	19	19	19	11.54	0.18	0.15
dk17x	41	41	41	19.67	19.10	4.40
ellen	49	spaceout	46	15.52	spaceout	165.49
ellen.min	21	21	21	4.48	5.13	0.08
fsync	24	24	24	13.12	0.02	0.01
mc	20	20	20	2.70	0.24	0.10
ofsync	24	24	24	13.12	0.02	0.01
shiftreg4	47	spaceout	45	12.57	spaceout	350.15
shiftreg	21	21	21	3.44	4.99	0.07
tav	9	9	9	76.35	0.00	0.00

performed with simulated annealing using both relational and functional representations of finite state machines.

A brief discussion is in order on the restriction that the number of states be a power of two. It was adopted in the theoretical frame as a convenient hypothesis in the build-up of the theory. The question is how we can deal with it in practice since most FSMs do not necessarily comply with it.

The solution is to introduce enough redundant states (i.e., states equivalent to existing states) so that the FSM has a number of states that is a power of two. There is a lot of freedom on what states to duplicate, and to be rigorous one could define the optimization problem about what states to duplicate with the objective of an optimal BDD encoding. To dispel the objection that this might seem artificial, let us point out that in general FSM encoding (say for two-level or multi-level optimal representations) it is an open problem to find the state transition graph representation leading to the best state-encoded logic representation. This was already noticed in the classical paper by Hartmanis and Stearns (1962), where they showed by a counter-example that a state-minimized FSM is not always the best starting point to achieve minimum encoded logic. So there is no reason to assume that the given CFSMs are or must be state-minimized, and we can make them redundant when needed to satisfy our power-of-two restriction. The optimal way to introduce redundancies in order to minimize the final number of BDD nodes is, of course, a more complex optimization problem, but the same is true of any formulation of 'exact' state encoding for any cost function, if one introduces into the formulation also finding the state transition graph that maps to the best encoded logic. There have been sporadic but inconclusive attempts to solve it for two-level and multi-level logic implementations; see (Lee and Perkowski, 1984) for concurrent state minimization and state assignment of finite state machines. An in-depth discussion of these general formulations can be found in (Villa *et al.*, 1997), Sec. 8.3.1, under the title 'Symbolic Don't Cares and Beyond', with references to relevant literature. So by proposing to add heuristically some redundant states to a CSFSM to make it a power-of-two, we are not in a position too different from the traditional case where we encode a CSFSM without knowing what state transition graph (maybe with redundancies) would be the best starting point, and therefore we start arbitrarily, say, with the state-minimized one. A recent work (Yuan, 2005) proposes 'FSM reengineering' as a technique to re-construct an FSM that is functionally equivalent to a given one to enable optimization tools to find better solutions to synthesis problems, and applies it to state encoding for low power.

In summary, the issue of our restriction is a special case of the more general problem: Given an FSM and a cost function, find an equivalent state transition graph and an encoding of its states in order to minimize the cost of the encoded logic.

5. Conclusions

We have presented an exact solution to the BDD input encoding problem. Our exact algorithm characterizes the two BDD reduction rules as combinatorial sets and finds encodable compatible sets with a maximum gain to produce the optimum encoding. We are aware that the exact algorithm is not practical for realistic problems. We presented it for two reasons:

1. as the first characterization of the exact solutions of the problem;

2. as a measure of the quality of a heuristic in the cases where we can compute the exact solutions.

We presented the results of a simple heuristic where no permutation was done when generating all sibling and isomorphic sets. This simple heuristic is, however, not practical enough, either. Future work on this topic includes practical heuristics based on the characterization that we provided through the exact algorithm, e.g., pruning smaller sibling and isomorphic sets at each step of their generation.

References

Bern J., Meinel C. and Slobodova A. (1996): *Global rebuilding of OBDD's avoiding memory requirement maxima.* — IEEE Trans. CAD Int. Circuits Syst., Vol. 15, No. 1, pp. 131–134.

Brayton R.K., Hachtel G.D., McMullen C.T. and Sangiovanni-Vincentelli A.L. (1984): *Logic Minimization Algorithms for VLSI Synthesis.* — Kluwer Academic Publishers.

Bryant R.E. (1986): *Graph-based algorithms for Boolean function manipulation*. — IEEE Trans. Comput., Vol. C(35), No. 8, pp. 677–691.

Bryant R.E. (1992): *Symbolic Boolean manipulation with ordered binary-decision diagrams*. — ACM Comput. Surveys, Vol. 24, No. 3.

Buch P., Narayan A., Richard Newton A. and Sangiovanni-Vincentelli A. (1997): *Logic synthesis for large pass transistor circuits*. — Proc. 1997 IEEE/ACM Int. Conf. Computer-Aided Design, ICCAD '97, Washington, DC, USA, pp. 663–670.

Cabodi G., Quer S. and Camurati P. (1995): *Transforming Boolean relations by symbolic encoding*, In: Proc. Correct Hardware Design and Verification CHARME'95, (P. Camurati and P. Eveking, Edi.), LNCS, Vol. 987, pp. 161–170.

Drechsler R., Drechsler N. and Gunther W. (2000): *Fast exact minimization of BDD's*. — IEEE Trans. CAD Int. Circ. Syst., Vol. 19, No. 3, pp. 384–389.

Drechsler R., Junhao Shi and Görschwin Fey (2004): *Synthesis of fully testable circuits from BDDs.* — IEEE Trans. CAD Int. Circ. Syst., Vol. 23, No. 3, pp. 440–443.

Gosti W., Villa T., Saldanha A. and Sangiovanni-Vincentelli A. (1998): *An exact input encoding algorithm for BDDs representing FSMs*. — Proc. 8-th Great Lakes Symp. VLSI, Lafayette, LA, USA, pp. 294–300.

Gosti W., Villa T., Saldanha A. and Sangiovanni-Vincentelli A.L. (1997): *Input encoding for minimum BDD size: Theory and experiments*. — Techn. rep., University of California at Berkeley, No. UCB/ERL M97/22.

Gunther W. and Drechsler R. (1998): *Linear transformations and exact minimization of BDDs.* — Proc. 8-th Great Lakes Symp. VLSI, Lafayette, LA, USA, pp. 325–330.

Gunther W. and Drechsler R. (2000): *ACTion: Combining logic synthesis and technology mapping for MUX-based FPGAs.* — J. Syst. Archit., Vol. 46, No. 14, pp. 1321–1334.

Hartmanis J. and Stearns R.E. (1962): *Some dangers in the state reduction of sequential machines*. — Inf. Contr., Vol. 5, No. 3, pp. 252–260.

Lavagno L., McGeer P., Saldanha A. and Sangiovanni-Vincentelli A.L. (1995): *Timed Shannon Circuits: A Power-Efficient Design Style and Synthesis Tool*. — Proc. 32-th Design Automation Conf., San Francisco, CA, USA, pp. 254–260.

Lee E.B. and Perkowski M. (1984): *Concurrent minimization and state assignment of finite state machines*. — Proc. IEEE Int. Conf. Syst., Man Cybernetics, Halifax, Nova Scotia, Canada, pp. 248–260.

Lisanke R. (1989): *Logic synthesis benchmark circuits for the International Workshop on Logic Synthesis*. — Research Triangle Park, North Carolina, Microelectronics Center of North Carolina.

Macchiarulo L., Benini L. and Macii E. (2001): *On-the-fly layout generation for PTL macrocells*. — Proc. Conf. Design, Automation and Test in Europe, DATE '01, pp. 546–551, Piscataway, NJ, USA, IEEE Press.

Meinel C., Somenzi F. and Theobald T. (2000): *Linear sifting of decision diagrams and its application in synthesis*. — IEEE Trans. CAD Int. Circ. Syst., Vol. 19, No. 5, pp. 521–533.

Meinel C. and Theobald T. (1999): *On the influence of state encoding on OBDD-representations of finite state machines*. — Theoret. Inf. Applic., Vol. 33, No. 1, pp. 21–31.

Meinel Ch. and Theobald T. (1996 a): *Local encoding transformations for optimizing OBDD-representations of finite state machines*. — Proc. Int. Conf. Formal Methods in Computer-Aided Design, London: Springer, No. 1166, pp. 404–418.

Meinel Ch. and Theobald T. (1996 b): *State encodings and OBDD-sizes*. — Techn. rep. 96-04, Universität Trier, Germany.

Rudell R. (1993): *Dynamic variable ordering for ordered binary decision diagrams*. — Proc. Int. Conf. Computer-Aided Design, San Francisco, CA, USA, pp. 42–47.

Saldanha A., Villa T., Brayton R. and Sangiovanni-Vincentelli A. (1994): *Satisfaction of input and output encoding constraints*. — IEEE Trans. CAD Int. Circ. Syst., Vol. 13, No. 5, pp 589–602.

Villa T., Kam T., Brayton R. and Sangiovanni-Vincentelli A. (1997): *Synthesis of FSMs: Logic Optimization*. — Boston: Kluwer.

Yuan L., Qu G., Villa T. and Sangiovanni-Vincentelli A.L. (2005): *FSM re-engineering and its application in low power state encoding*. — Proc. 2005 Asia South Pacific Design Automation Conf. (ASP-DAC 2005), Shanghai, China, Vol. 1, pp. 254–259.

Appendix

A. Proofs

A.1. Proof of Theorem 1. Let T be the forest of binary decision trees representing $e_{\rm opt}(F)$. To get from T the BDD representing $e_{\rm opt}(F)$, the BDD reduction rules, Rule 1 and Rule 2, are applied. It suffices to prove that any reduction can be found by using only $\mathcal{S}$ and $\mathcal{I}$. We divide the proof into two parts, according to whether Rule 1 or Rule 2 is applied:

1. Applying Rule 1: Consider the part of T in Fig. 13.

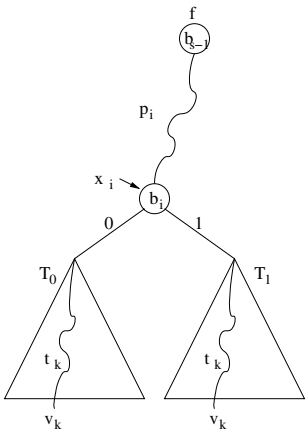


Fig. 13. Binary tree for Case 1 of the proof of Theorem 1.

Let x_i be a node with the label b_i. Assume that we can apply Rule 1 at x_i, then *then*(x_i) is isomorphic with *else*(x_i). Let an arbitrary path from a function f in $e_{\rm opt}(F)$ to x_i be p_i. Also, let t_k be the path from *then*(x_i)/*else*(x_i) to leaf v_k, $0 \le k \le m-1$, where m is the number of leaves in the subtree rooted at *then*(x_i)/*else*(x_i). Define the symbolic lists

$$M_i = (d_0, d_1, \ldots, d_{m-1}),$$

where $sym(d_k) = e_{\rm opt}^{-1}(p_i\overline{b_i}t_k)$, $fn(d_k) = f$, $val(d_k) = v_k$, and

$$M_i' = (d_0', d_1', \ldots, d_{m-1}'),$$

where $sym(d_k') = e_{\rm opt}^{-1}(p_i b_i t_k)$, $fn(d_k') = f$, $val(d_k') = v_k$.

Then we have

(a) $|M_i|$ and $|M_i'|$ are equal and are powers of two,

(b) For any t_k, $f(p_i\overline{b_i}t_k) = f(p_i b_i t_k) = v_k$.

$S = \{M_i, M_i'\}$ is a sibling set because:

- Property (a) is exactly Condition 1 of Definition 4.
- Property (b) satisfies Condition 2 of Definition 4 because all k-th elements of $|M_i|$ and $|M_i'|$ have the same value.
- Property (b) satisfies Condition 3 of Definition 4 and Condition 1 of Definition 5 because all elements of $|M_i|$ and $|M_i'|$ are different.
- By definition, both $|M_i|$ and $|M_i'|$ contain symbols from the same function; therefore, Condition 2 of Definition 5 is satisfied.

We will show later that an encoding that takes advantage of the reduction implied by S can be found.

2. Applying Rule 2: Consider the part of T in Fig. 14.

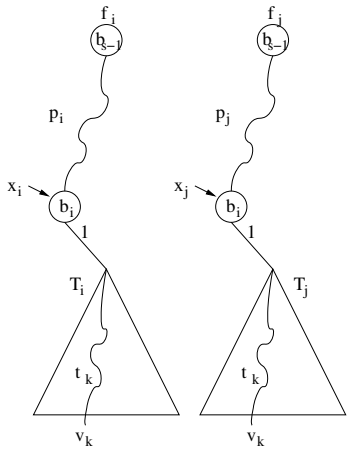


Fig. 14. Binary tree for Case 2 of the proof of Theorem 1.

Let x_i and x_j be two nodes in T with the label b_i. Without loss of generality, assume that we can apply Rule 2 at *then*(x_i) and *then*(x_j), then *then*(x_i) is isomorphic with *then*(x_j) in the BDD representing $e_{\rm opt}(F)$. Let p_i and p_j be two arbitrary paths in T from the function f_i to x_i and the function f_j to x_j, respectively. Also, let t_k be the path from *then*(x_i)/*then*(x_j) to leaf v_k, $0 \le k \le m-1$, where m is the number of leaves in the subtree rooted at *then*(x_i)/*then*(x_j). Define the symbolic lists

$$M_i = (d_0, d_1, \ldots, d_{m-1}),$$

where $sym(d_k) = e_{\rm opt}^{-1}(p_i b_i t_k)$, $fn(d_k) = f_i$, $val(d_k) = v_k$, and

$$M_j = (d_0', d_1', \ldots, d_{m-1}'),$$

where $sym(d_k') = e_{\rm opt}^{-1}(p_j b_i t_k)$, $fn(d_k') = f_j$, $val(d_k') = v_k$.

Then, we have the following:

(a) $|M_i|$ and $|M_j|$ are equal and are powers of two,

(b) For any t_k, $f_i(p_i b_i t_k) = f_j(p_j b_i t_k) = v_k$.

$I = \{M_i, M_j\}$ is an isomorphic set because:

- Property (a) is exactly Condition 1 of Definition 4.

- Property (b) satisfies Condition 2 of Definition 4 because all k-th elements of $|M_i|$ and $|M_i'|$ have the same value.
- If $p_i = p_j$, then all k-th symbols of M_i and $Sym(M_j)$ are the same, and if $p_i \neq p_j$, then no element of $Sym(M_i)$ is the same as any element of $Sym(M_j)$, and this satisfies Condition 3 of Definition 4.

We will show later that an encoding that takes advantage of the reduction implied by I can be found. If there are more than two nodes where we can apply Rule 2, the set I would simply contain more elements.

Note that Cases 1 and 2 are sufficient for this proof. All other reductions are just a combination of Cases 1 and 2. For example, consider Fig. 15. By case 2, there is an

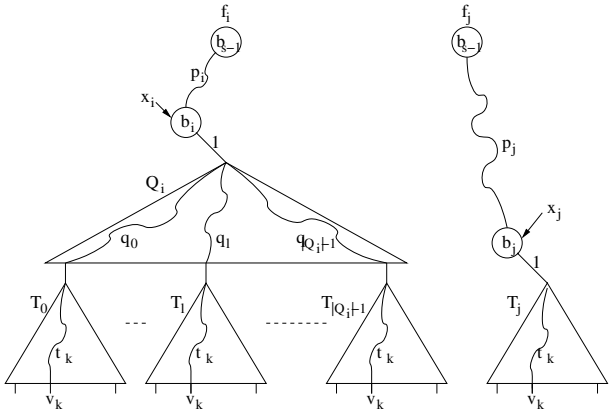


Fig. 15. Binary tree for the combination of Cases 1 and 2 of the proof of Theorem 1.

$I = \{l^0, l^1, \ldots, l^{|Q_i|-1}, l^{|Q_i|}\}$, where each list l^r, $r = 0, 1, \ldots, |Q_i| - 1, |Q_i|$ contains the symbols encoded by the minterms of paths passing through $then(x_r)$ and ending respectively in the leaves of subtrees $T_0, T_1, \ldots, T_{|Q_i|-1}, T_{|Q_i|} = T_j$. By Case 1, there exists S for each node in the subtree Q_i, where Q_i is the subtree rooted at $then(x_i)$ and all leaves have labels b_j.

A.2. Proof of Theorem 2. Note that $l' \sim l''$ implies one of the following:

1. Every symbol of l' is different from any symbol of l''. In this case, there certainly exists an encoding such that the theorem is true.
2. The symbols of l' match exactly the symbols of l'' starting at a position $a \times |l'|$ in the same order and $|l''| \geq (a+1) \times |l'|$. In this case, we encode the symbols of l' and l'' such that the symbols of l'' span a tree and the symbols of l' span a subtree of the tree formed by the codes of the symbols of l''.
3. This case is the dual of Case 2.

A.3. Proof of Theorem 3. Since two symbolic lists are compatible if and only if their symbols do not overlap or the symbols of one are a sublist of those of the other starting at a power-of-2 position, there is a notion of maximality in L. Sorting L in non-increasing order and applying the encoding procedure in the proof of Theorem 2 produces the results satisfying the claim of this theorem.

A.4. Proof of Theorem 4. Suppose that the theorem is not true. Then either of the following must be true:

1. There exists a better encoding, but no compatible captures it. A better encoding in this case means that more reductions than those implied by any compatible can be applied. But by Theorem 1, we know that every reduction is modeled by either a sibling or an isomorphic set and, by definition of compatibility, reductions implied by two incompatible sets cannot be applied together. Hence, all optimal encodings must be yielded by compatibles.
2. There exists another compatible with a lower gain that yields BDDs with fewer number of nodes. This is not possible because, by Definition 11, compatibles with larger gains yield smaller BDDs.

A.5. Proof of Theorem 5. This is an inductive proof. At the step i we have a set $C_i \in C$ of the sibling sets $\mathcal{S}_i$ and the isomorphic sets $\mathcal{I}_i$. $\mathcal{S}_i$ and $\mathcal{I}_i$ are sets that are contained completely in $\mathcal{S}_{i-1}, \mathcal{S}_{i-2}, \ldots, \mathcal{S}_0$ and $\mathcal{I}_{i-1}, \mathcal{I}_{i-2}, \ldots, \mathcal{I}_0$ and not contained in any other sets in C. At the step i, we compute the total gain g_i of C_i (i.e., $\mathcal{S}_i, \mathcal{S}_{i-1}, \ldots, \mathcal{S}_0$ and $\mathcal{I}_i, \mathcal{I}_{i-1}, \ldots, \mathcal{I}_0$). Let $\mathcal{J}_i$ be the set of isomorphic sets of $\mathcal{I}_i$ that do not have any symbolic lists that are subsets of any symbolic lists of any isomorphic sets of $\mathcal{I}_i$. Let $\mathcal{J}_i'$ be the set difference of $\mathcal{I}_i$ and $\mathcal{J}_i$, with the symbolic lists of isomorphic sets that are subsets of those of isomorphic sets in $\mathcal{J}_i$ removed. To illustrate what $\mathcal{J}_i$ and $\mathcal{J}_i'$ represent, we look at the binary decision trees for the functions f_0, f_1, and f_2 in Fig. 16. In this figure, T_3

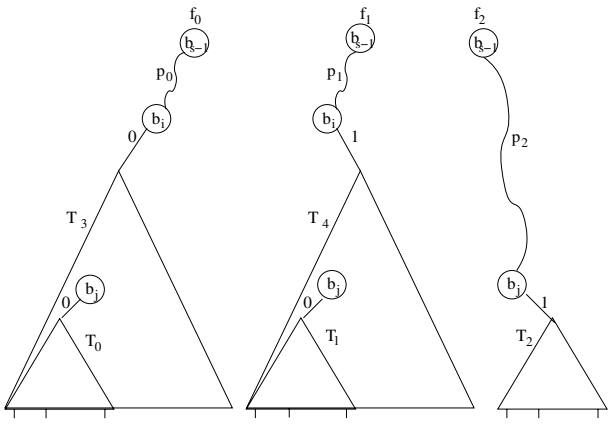


Fig. 16. Example for proving Theorem 5.

and T_4 are isomorphic and T_0, T_1, and T_2 are isomorphic. Let T_i' denote the symbolic list corresponding to the symbols whose codes are represented by the subtree T_i. Using the algorithm for finding all sibling and isomorphic sets as described in Section 3.2.2, we generate the isomorphic sets $I_0 = \{T_3', T_4'\}$ and $I_1 = \{T_0', T_1', T_2'\}$ among other sets. For this example, $\mathcal{J}_i$ will contain $I_0 = \{T_3', T_4'\}$ and $\mathcal{J}_i'$ will contain $I_1' = \{T_2'\}$. When we apply the BDD reduction rules to this example, the isomorphic subgraphs associated with each isomorphic set in $\mathcal{J}_i'$ will be removed by Rule 2. Hence, if l^i is the i-th symbolic list of an $I \in \mathcal{J}_i'$, its gain needs to be updated to $|I| \times (|l^0| - 1)$.

Case $i = 0$. g_0 is simply equal to the total gain of $\mathcal{S}_0, \mathcal{J}_0$, and $\mathcal{J}_0'$, which is what **Gain**(C) computes if we do not allow its recursion.

Case $i = k$. Assume that **Gain**(C) computes g_k if we allow the recursion k times.

Case $i = k + 1$. Since the gain g_k implies the merging of isomorphic subgraphs into one subgraph at the recursion k, the additional gain going from the step k to the step $k + 1$ is the reduction applied to any single isomorphic subgraph. It suffices to consider only the first symbolic list of every isomorphic set in $\mathcal{J}_k$. The reason is that the isomorphic subgraphs corresponding to the isomorphic sets of $\mathcal{J}_k'$ form a subgraph.

B. Algorithms

B.1. Algorithm to Encode a Compatible.

Algorithm 1 (Encoding a Compatible)

encode(C, D)
Input: A compatible C, a set of symbols D.
Output: Codes for D stored in a 2-dimensional array code.
Comment: *reverseBit*() takes an integer argument and reverses all its bits.
$c_{i,j}$ denotes the j-th element of the i-th list of c.
order is the array of ordered codes, e.g. 0000, 1000, 0100, 1100, 0010, 1010, ...
This array recursively partitions all the codes into two equal partitions and orders them in non-increasing size.

```
/* Initialize codes */
for d = 0 to |D| − 1 do
  for i = 0 to s − 1 do
    code[d][i] = '-'

/* Initialize orders */
for i = 0 to |D| − 1 do
  order[i] = reverseBit(i)

/* Get top level sets and sort them
in non-increasing cube size */
Top = {c | c ∈ C and no d ∈ C contains c}
for each c ∈ Top do
  if c is a sibling set
    cubeSize(c) = 2 × |l^0| of c
  else
    cubeSize(c) = |l^0| of c
T_sorted = sort T in non-increasing cubeSize

/* Encode sorted top level sets */
for each c ∈ T_sorted do
  if (c is a sibling set and code[c_{0,0}][0] = '-') then
    while (code[order[j]][0] = '-') do
      j = j + 1
    for i = 0 to |c_0| − 1 do
      code[sym(c_0, i)] = order[j] + i
    for i = 0 to |c_1| − 1 do
      code[sym(c_1, i)] = order[j] + |c_0| + i
  else
    for i = 0 to |c| − 1 do
      if (code[c_{i,0}][0] = '-') then
        while (code[order[j]][0] = '-') do
          j = j + 1
        for k = 0 to |c_i| − 1 do
          code[sym(c_{i,k})] = order[j] + k

/* Encode remaining codes */
for d = 0 to |D| − 1 do
  if code[d] = '-' then
    while (code[order[j]][0] = '-') do
      j = j + 1
    code[d] = order[j]
return code
```

B.2. Gain Calculation Algorithm.

Algorithm 2 (Gain Calculation)

Gain(C)
Input: A compatible C
Output: The gain of C

```
if C = ∅ then
  return 0
end if

gain = 0

/* Get top level sets */
Top = {c | c ∈ C and no d ∈ C contains c}
I = {I | I ∈ Top and I is an isomorphic set}

/* J contains isomorphic sets whose
symbolic lists are not subsets of any
symbolic list of any other isomorphic
sets.  J' contains isomorphic sets
whose symbolic lists are subsets
of some symbolic lists of some other
isomorphic sets.  Symbolic lists that
are subsets of other symbolic lists
are removed from J' */
J = ∅
```

$\mathcal{J}' = \emptyset$
for each $I \in \mathcal{I}$ do
  found = FALSE
  for each $I^{aux} \in \mathcal{I}$ do
    $I' = I$
    if $\exists l \in I, l^{aux} \in I^{aux}$ $Sym(l) \subset Sym(l^{aux})$ then
      $I' = I' \backslash \{l\}$
      found = TRUE
  if found = TRUE then
    $\mathcal{J}' = \mathcal{J}' \cup \{I'\}$
  else
    $\mathcal{J} = \mathcal{J} \cup \{I\}$

/* Add gains contributed by $\mathcal{S}$ */
$\mathcal{S} = \{S \mid S \in Top$ and S is a sibling set$\}$
for each $S \in \mathcal{S}$ do
  $gain = gain + gain(S)$
end for

/* Add gains contributed by $\mathcal{J}'$ */
for each $I \in \mathcal{J}'$ do
  $gain = gain + gain(I)$
end for

/* Recursively add gains contributed by $\mathcal{J}$ */
for each $I \in \mathcal{J}$ do
  $C_s = \{c \mid c \in C, \forall l \in c$ $Sym(l) \subset Sym(l^0)$,
    l^0 is the 0-th list of $I\}$
  $gain = gain + \mathbf{Gain}(\mathcal{I}_s)$
end for
return *gain*

B.3. CNF Expander.

Algorithm 3 (Convert 2-CNF to Sum-of-Products)

cnf_to_sop(*expr*) {
  x = splitting variable
  C = all sum terms with the variable x
  reduced_expr = *expr* without the sum-terms in C
  x_expr = sum-of-product expression of C
  return (product_sop (*x_expr*, cnf_to_sop(*reduced_expr*)))
}

/* Obtain the product of two expressions;
expr1 has 2 terms, where the first term is a single variable */
product_sop (*expr1*, *expr2*) {
  product_expr = product of *expr1* and *expr2*
  result_expr = single_cube_containment (*product_expr*)
  return (*result_expr*)
}

Received: 12 May 2006