An Exact Input Encoding Algorithm for BDDs Representing FSMs *

Wilsin Gosti[1] Tiziano Villa[2] Alexander Saldanha[3] Alberto L. Sangiovanni-Vincentelli[1]

[1] Dept. of EECS, University of California, Berkeley, CA 94720
[2] PARADES, Via di S. Pantaleo, 66, 00186 Roma, Italy
[3] Cadence Berkeley Labs, 2001 Addison St., 3rd Floor, Berkeley, CA 94704

Abstract

We address the problem of encoding the state variables of a finite state machine such that the BDD representing its characteristic function has the minimum number of nodes. We present an exact formulation of the problem. Our formulation characterizes the two BDD reduction rules by deriving conditions under which these reduction rules can be applied. We then provide an algorithm that finds these conditions and solves the problem by formulating it as a 2-CNF formula and extracting all its prime implicants. In addition to this, we implemented a simulated annealing algorithm for this problem and provide a thorough experiment of the impact of encoding on a BDD representing an FSM with different orderings.

1 Introduction

Reduced Ordered Binary Decision Diagrams (BDDs) are a data structure used to efficiently represent and manipulate logic functions. Since being introduced by Bryant [1] in 1986, 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 depends on the ordering of its variables. For some functions, the BDD sizes are linear in the number of variables for one ordering, and exponential for another [2]. Many heuristics have been proposed to find good orderings, e.g., the sifting dynamic reordering algorithm [8].

BDDs can also be used to represent the characteristic functions of the transition relations of finite state machines (FSMs). In this case, the size of the BDDs depends not only on variable ordering, but also on state encoding. Meinel and Theobald studied the effect of state encoding on autonomous counters in [7]. They experimented with 3 different encodings: the standard minimum-length encoding, which gives the lower bound of $5n-3$ internal nodes for an n-bit autonomous counter, the Gray encoding, which gives the 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 by state encoding the BDD size of an FSM representation is motivated by applications in logic synthesis and verification. Regarding synthesis, BDDs can be used as a starting point for logic optimization. An example is their use in Timed Shannon Circuits [5], where the circuits derived are reported to be competitive in area and often significantly better in power. 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 [3].

In this paper, we look into the problem of finding the optimum state encoding that minimizes the BDD that represents an FSM. We call this problem the *BDD encoding problem*. To the best of our knowledge, this problem has never been addressed before. The work 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 nodes in BDDs, Meinel and Theobald proposed in [6] 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 where sifting algorithm could not.

This paper addresses a restricted version of the BDD encoding problem, the *BDD input encoding problem*.

The remainder of this paper is structured as follows. In Section 2 the BDD input encoding problem is defined formally. In Section 3 the exact formulation and algorithm are presented in detail. Section 4 describes the simulated annealing algorithm experiment on the impact of encoding on BDD representing the characteristics of FSMs. Finally, we conclude in Section 5. Due to limited space, all proofs are not included in this paper but can be found in [4].

2 BDD Input Encoding Problem

The BDD input encoding problem is defined as:
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, $B = \{b_{s-1}, b_{s-2}, \ldots, b_0\}$.
Output: Bijection $e : D \mapsto \mathbf{B}^s$ such that the size of the BDD representing $e(F)$ is minimum, where $e(F) =$

*This research was supported in part by UC Micro 532419

0-8186-8409-7/98 $10.00 © 1998 IEEE

$\{e(f_0), e(f_1), \ldots, e(f_{|F|-1})\}$, 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., $e_{opt} = \min_e\{|e(F)|\}$, where $|e(F)|$ is the number of nodes of the multi-rooted BDD representing $e(F)$.

In other words, the problem is to find 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 minimum number of nodes. In this paper, 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.

This setting models the problem 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 the proper inputs 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 the encoded BDDs are actually multi-terminal BDDs (MTBDDs), we still refer to them as BDDs.

As an example, we consider the SLMVTs for functions f and g shown in Figure 1. If we encode v as: $e_1(0) = 010, e_1(1) = 100, e_1(2) = 000, e_1(3) = 001, e_1(4) = 011, e_1(5) = 110, e_1(6) = 111, e_1(7) = 101$, with ordering b_2, b_1, b_0 we get the 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 $e_2(0) = 010, e_2(1) = 100, e_2(2) = 000, e_2(3) = 011, e_2(4) = 001, e_2(5) = 111, e_2(6) = 101, e_2(7) = 110$, the BDD that we get has 10 nodes. Figure 3 shows the BDD representing f and g using this encoding.

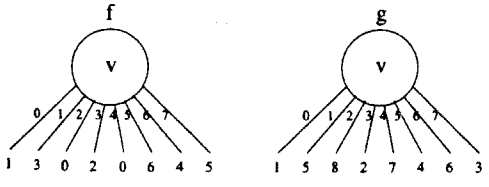

Figure 1: Multi-Valued Functions f and g

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

3 Exact Algorithm

3.1 Characterization of BDD Node Reductions

There are two reduction rules that are applied to a binary decision tree representing a logic function to get

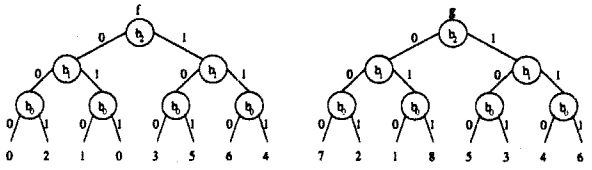

Figure 2: BDD for f and g Using Encoding e_1

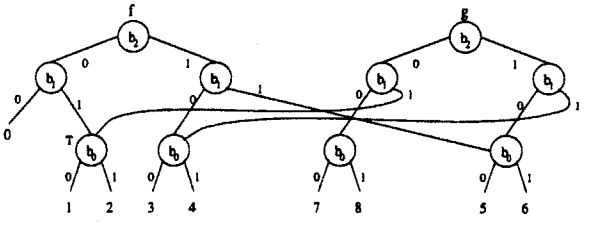

Figure 3: BDD for f and g Using Encoding e_2

a binary decision diagram. These rules are: 1) elimination of nodes with the same *then* and *else* children, and 2) elimination of duplicate isomorphic subgraphs. For a given SLMVT T, assume that we have an encoding of the domain and T is represented as a binary decision tree T_b. Applying Rule 1 to a node n of T_b implies that the leaf nodes of the *then* and *else* subtrees of n have the same labels. This also means that for each leaf node of the subtree rooted at n, there are at least 2 leaf nodes of T that have the same labels. From T, we can therefore find all pairs of ordered sets of leaf nodes whose labels are the same. We call such pairs *sibling sets*. So a sibling set is a pair of sets of leaf nodes of an SLMVT which has an implied node reduction associated with Rule 1. Since Rule 1 is applied within a single function, sibling sets are defined within a single function as well. Similarly, applying Rule 2 to s isomorphic subtrees of T_b implies that the leaf nodes of these isomorphic subtrees have the same labels. This means that for each leaf node of these isomorphic subtrees, there are at least s leaf nodes of T that have the same labels. From T, we can therefore find all groups of ordered sets of leaf nodes whose labels are the same. We call such groups *isomorphic sets*. So an isomorphic set is a set of sets of leaf nodes of an SLMVT which has an implied node reduction associated with Rule 2. Since Rule 2 can be applied across multiple functions, isomorphic sets are defined across multiple functions as well.

We now formally define sibling and isomorphic sets.

Definition 1 *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 notations are defined for d_f: $sym(d_f) = d$, $fn(d_f) = f$, and $val(d_f) = f(d)$.*

Definition 2 *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 as l_k. The set of all symbols of l is $Sym(l) = \{sym(l_k) \mid 0 \leq k \leq |l| - 1\}$. The function of l is $Fn(l) = fn(l_0)$.*

Definition 3 *An isomorphic set I is a set of at least two symbolic lists. The j-th element of I is denoted as 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. 2) The k-th elements of all symbolic lists of I have the same value. 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''.*

Definition 4 *A sibling set S is an isomorphic set with 2 symbolic lists, l^0 and l^1, and 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. 2) The functions of l^0 and l^1 are the same.*

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 $\mathcal{I}$.

The motivation of defining sibling and isomorphic sets is that an encoding exists which takes advantage of the reductions implied by these sets. This is discussed by the definition followed by the two propositions below. The term *tree* is used to mean the encoded binary tree representing a function.

Definition 5 *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 Figure 1 and the encoding e_2 as in page 2, the codes for the symbols 0 and 3 span the tree rooted at T in Figure 3.

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, and all T_l's are isomorphic.*

To illustrate these propositions, we look back to the example in Figure 1. The $\mathcal{S}$ and $\mathcal{I}$ of this example are: $S_0 = \{(2_f), (4_f)\}$, $I_0 = \{(0_f, 3_f), (0_g, 3_g)\}$, $I_1 = \{(3_f, 0_f), (3_g, 0_g)\}$, $I_2 = \{(1_f, 6_f), (7_g, 5_g)\}$, $I_3 = \{(6_f, 1_f), (5_g, 7_g)\}$, $I_4 = \{(7_f, 5_f), (1_g, 6_g)\}$, and $I_5 = \{(5_f, 7_f), (6_g, 1_g)\}$. For now, we focus only on S_0, I_0, I_2, and I_4. Each S_i or I_i justifies why encoding e_2 is better than 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. 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 encoding e_2, but not e_1.

Starting from symbolic lists with one symbol, larger lists can be built recursively. Then from these symbolic lists, the sets $\mathcal{S}$ and $\mathcal{I}$ can be constructed.

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

Theorem 3.1 *Using only $\mathcal{S}$ and $\mathcal{I}$, an optimum encoding e_{opt} can be obtained.*

Theorem 3.1 says that $\mathcal{S}$ and $\mathcal{I}$ contain all the information that is needed to find an optimum encoding.

3.2 Finding an Optimum Encoding

From here on, the number of nodes that can be reduced is with respect to the complete binary trees that represent the encoded F. When not specified, a set means either a sibling set or an isomorphic set.

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*. 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 isomorphic set $I = \{(2_f, 3_f), (2_g, 3_g)\}$ of Figure 4 can not be selected together because S says that symbols 1 and 2 should span exactly a subtree while I says that 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 can be selected together and which can not. For that we define the notion of *compatibility*.

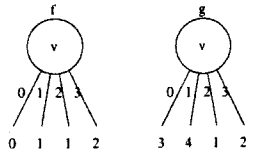

Figure 4: Example of Incompatible Sets

Definition 6 *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.*

The following definitions and theorems outline an algorithm for checking compatibility among sets.

Definition 7 *Symbolic lists l' and l'' are compatible, denoted as $l' \sim l''$, if at least one of the following conditions is true: 1) the set of symbols of l' does not intersect the set of symbols of l'', 2) the symbols of l' match exactly the symbols of l'' in the same order starting at position $a \times |l'|$, and 3) the symbols of l'' match exactly the symbols of l' in the same order starting at position $a \times |l''|$.*

Definition 7 defines pair-wise compatibility between symbolic lists. It says that two lists are compatible if their symbols do not intersect or the symbols of one list is a subset of the symbols of the other starting at a power-of-2 position. The next theorem states how the compatibility among a set of symbolic lists is related to the pair-wise compatibility.

Theorem 3.2 *If 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 3.2.

Corollary 3.1 *Sibling sets S' and S'' are compatible if $l^{S'}$ is compatible with $l^{S''}$.*

Corollary 3.2 *Sibling set S and isomorphic set I are compatible if l^S is compatible with every list of I.*

Corollary 3.3 *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 $\mathcal{I}$. We call a set of compatible sets a *compatible*.

Corollary 3.4 *Given a compatible C, there exists an encoding e such that the reductions implied by all its elements can be applied.*

We can compute the encoding of a compatible by the following: starting with a binary tree, we assign codes to the symbols of symbolic lists in the order of non-increasing length of the symbolic lists one by one. The symbols of each symbolic list are assigned to occupy the largest subtree of codes still available. For a compatible C, we denote the encoding found by this algorithm by $e_{alg}(C)$.

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. The gain of a compatible C is then 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 $e_{alg}(C)$.

Theorem 3.3 *A compatible of maximum gain yields an optimum encoding.*

Given a list of compatibles, we therefore need to calculate the gain of each compatible in order to find the maximum gain one. However, the compatible gain calculation is not straight forward. The gain of a compatible is not always equal to the sum of the gains of its elements because the reductions implied by a set may subsume the reduction implied by another set. Our gain calculation of a compatible is to find the sets with 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.

3.3 Maximal Compatibles

Having found all sibling and isomorphic sets, the next task is to find a maximum gain compatible. As 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.

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 8 *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 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.

Our procedure to find all maximal cliques of G is as follows:

- Formulate the problem into a 2-CNF formula ϕ as follows: for each unconnected pair of vertices, x_1 and x_2, we create a clause $(\overline{x_1} \vee \overline{x_2})$.
- 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.
- For each prime implicant, the variables that do not appear in it form a maximal clique.

The CNF expander used here is the one developed by [9] and some modifications. We explain briefly here how the algorithm works.

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. After all multiplications are done, the result is a list of all prime implicants of the formula.

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_i will be much smaller than n^2.

As a comparison, we generate an approximation algorithm where not all sibling and isomorphic sets are generated. For a more detail explanation of this approximation, we refer the readers to [4]. The approximation algorithm took 165 seconds and 350 seconds of CPU time to find the optimal 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.

Table 1: BDD sizes of CSFSMs using SA, exact and approximation (App.) algorithms.

Name	BDD Size			CPU Time		
	SA	Exact	App.	SA	Exact	App.
dk15x	19	19	19	11.54	0.17	0.14
dk17x	41	41	41	19.67	19.10	4.39
ellen	49	s.o.	46	15.52	s.o.	165.48
ellen.min	21	21	21	4.47	5.12	0.07
fsync	24	24	24	13.12	0.01	0.01
mc	20	20	20	2.70	0.23	0.09
ofsync	24	24	24	13.11	0.01	0.01
shiftreg4	47	s.o.	45	12.57	s.o.	350.14
shiftreg	21	21	21	3.43	4.98	0.07
tav	9	9	9	76.35	0.00	0.00

3.4 Experimental Results

The experiments were performed on a DEC AlphaServer 8400 5/300 with 2Gb of memory. Beside the exact algorithm, an experiment with an approximation algorithm was also done [4]. For comparison purposes, the results of these two algorithms and our simulated annealing runs (which is explained in Section 4) are shown in Table 1. CPU times are also included in this table. Circuits whose executions were timed out after one hour of CPU time are not listed. Except for *ellen* and *shiftreg4*, the simulated annealing algorithm finds the optimum solutions.

4 Encoding Using Simulated Annealing

The exact algorithm is useful to evaluate the quality of heuristic algorithms. We implemented an algorithm for FSM encodings based on simulated annealing. In Section 3.4, we saw that the exact algorithm confirm the effectiveness of this algorithm. In addition, we also perform experiments on the impact of encoding on variable orderings of BDDs representing FSMs.

4.1 Algorithm

We assume that logarithmic encoding is used for all states, which means that we use the smallest number of bits required to encode the states. A code where all the bits are either 0 or 1 is called a *code point*, e.g., if the number of bits used is 3, then `010` is a code point and `01-` is not. The initial move randomly assigns a code point to each state. If the number of states is not a power of two, then some code points are not used. The starting temperature is 100. The temperature is reduced by a constant factor of 0.8, i.e., $T = 0.8T$. The stopping criterion for each temperature is when the number of consecutively rejected moves is 3. The next move function is a swap of code points between two randomly chosen states if the number of states is a power of two. If it is not, then the next move function is a swap of the code points between two randomly chosen states or a swap of the code point of a randomly chosen state and a randomly chosen unused code point.

We perform our experiment for both the functional and the relational representations of FSMs. For either of these representations, we build the BDDs for each encoding in each move. The number of BDD nodes is our cost function.

For incompletely specified FSMs, all unspecified transitions are treated as no change in state. In other words, for a present state and primary inputs combination, the next state is the same as the present state if the transition is not specified.

It is well known that variable ordering affects the BDD size. In this experiment, we consider several variable orderings for the relational representation. In our variable orderings, when we say that the present state and next state variables are interleaved, we mean that the i-th present state variable is immediately followed by the i-th next state variable in the ordering.

The variable orderings from the lowest level to the highest level follow (note that the lower the level of a variable is, the higher its position is in the BDD): 1) Ordering **I**: inputs, present states, next states, outputs. 2) Ordering **II**: inputs, present states, next states, outputs. The present state and next state variables are interleaved. 3) Ordering **III**: inputs, outputs, present states, next states. 4) Ordering **IV**: inputs, outputs, present states, next states. The present state and next state variables are interleaved.

4.2 Experimental Results

Here we include a subset of the results, we refer the readers to [4] for more complete results. Our implementation uses the Long's BDD package. The test cases include the MCNC benchmark set. The simulated annealing algorithm is run once for each circuit.

The results of the simulated annealing runs for CSFSMs are tabulated in Table 4. Columns 2 through 5 list the minimum numbers of BDD nodes in each ordering. Columns 6 through 9 show the average BDD sizes. The standard deviations are listed from column 10 through 13.

Table 2: SA runs for functional representation of ISFSMs.

Name	Min BDD Size	Ave BDD Size	Std Dev
bbgun	3706	4069	388
cf	324	446	50
cpab	169	278	58
dec	9212	13090	2017
ex1inp	491	569	56
kirkman	403	415	3
master	3545	4244	387
p21stg	8631	9031	317
planet	1374	1452	65
rpss	879	990	159
s1a	940	1075	93
sand	2314	2884	328
saucier	412	485	48
scf	68798	84517	22819
slave	823	971	90
str	1042	1333	169
styr	710	751	59
viterbi	2431	3006	309

Our results show that for CSFSMs, interleaving present state and next state variables increases or decreases the BDD sizes by only a small amount. We see that Ordering I and II are generally better than Ordering III and IV. We also found that different encodings do not change the BDD size dramatically.

The simulated annealing results for ISFSMs are tabulated in Table 5. The entry "s.o." means that it ran out of memory, and "> 3600" means that it exceeded 3600 seconds of CPU time.

Our results show that Ordering I and II are better in most cases. In some cases like *bbgun*, *dec*, and *viterbi*, they are substantially better. The large discrepancy in BDD sizes for these cases is due to the large BDD needed to represent the primary outputs. Interestingly, BDD sizes are smaller when state variables are not interleaved. Different encodings do affect the BDD sizes of ISFSMs more than those of CSFSMs; however, the differences are not substantial.

For functional representations, the results are shown in Table 3. Our results show that even for CSFSMs, different encodings affect the BDD size considerably for some circuits. For example, the average size for *maincont* is 90 with a standard deviation of 12, while the minimum BDD size that the simulated annealing algorithm found is 43. This also means that there are not many encodings which would produce small BDDs for this circuit.

The results for ISFSMs are shown in Table 2. We see from this table that, similarly to CSFSMs, encoding plays an important role in determining the BDD size. For example, the minimum BDD size for *dec* is 9212, while the average size is 13090 with a standard deviation of 2017 nodes.

5 Conclusions

We presented an exact formulation and an implementation of the BDD input encoding problem. Although it is practical only for very small examples, it is useful to evaluate the quality of heuristic algorithms. In particular we implemented an algorithm for FSM encoding based on simulated annealing.

We ran the exact algorithm and the simulated annealing algorithm on the MCNC benchmark circuits. On eight out of ten circuits that we could run the exact algorithm on, the simulated annealing algorithm found the optimum solutions. We have also run the simulated annealing algorithm on BDDs representing the characteristic functions of FSMs and on BDDs representing the transition functions and output functions with different variable orderings. We have shown results comparing the impact of encoding on these different orderings.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C(35):677–691, 1986.

[2] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[3] G. Cabodi, S. Quer, and P. Camurati. Transforming boolean relations by symbolic encoding. In P. Camurati and P. Eveking, editors, *Proc. of CHARME, Correct Hardware Design and Verification Conference*, volume 987 of *LNCS*, pages 161–170. Springer Verlag, October 1995.

[4] W. Gosti, T. Villa, A. Saldanha, and A.L. Sangiovanni-Vincentelli. Input encoding for minimum BDD size: Theory and experiments. Technical report, UCB/ERL M97/22, 1997.

[5] L. Lavagno, P. McGeer, A. Saldanha, and A.L. Sangiovanni-Vincentelli. Timed Shannon Circuits: A Power-Efficient Design Style and Synthes is Tool. In *Proc. of the* 32^{th} *DAC*, pages 254–260, June 1995.

[6] Ch. Meinel and T. Theobald. Local encoding transformations for optimizing OBDD-representations of finite state machines. In *Proc. of FMCAD*, pages 404–418, 1996.

[7] Ch. Meinel and T. Theobald. State encodings and OBDD-sizes. Technical Report 96-04, Universität Trier, 1996.

[8] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of the ICCAD*, pages 42–47, 1993.

[9] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of FSMs: logic optimization.* Kluwer Academic Publishers, 1997.

Table 3: SA runs for functional representation of CSFSMs.

Name	Min BDD Size	Ave BDD Size	Standard Deviation
dk15x	38	39	1
dk17x	72	79	2
ellen.min	7	15	2
ellen	21	41	2
ex6inp	86	108	8
fstate	44	77	12
fsync	61	62	0
maincont	43	90	19
pkheader	64	76	4
scud	200	254	19
shiftreg	5	14	2
tbk	391	438	15
vmecont	365	398	13

Table 4: SA runs for relational representation of CSFSMs.

Name	Min BDD Size				Ave BDD Size				Standard Deviation			
	I	II	III	IV	I	II	III	IV	I	II	III	IV
dk15x	80	76	93	95	83	83	96	99	2	4	2	3
dk17x	75	79	81	87	84	91	91	96	3	4	3	3
ellen.min	93	75	77	73	93	87	89	88	0	4	3	3
ellen	113	105	110	90	124	140	144	138	2	8	9	11
ex6inp	167	189	287	297	180	211	297	309	5	7	3	4
fstate	176	177	220	219	189	188	225	226	5	9	2	2
fsync	67	66	92	96	70	71	96	98	2	3	2	2
maincont	108	104	116	115	115	117	123	128	2	5	2	4
pkheader	20305	20051	12475	12473	20670	21218	12484	12485	429	954	3	4
scud	303	407	582	600	342	455	598	629	15	18	5	7
shiftreg	45	23	27	27	45	39	40	40	0	3	3	3
tbk	493	552	584	668	530	645	633	717	13	24	15	18
vmecont	4238	3997	4999	5016	4392	4092	5016	5039	66	53	6	10

Table 5: SA runs for relational representation of ISFSMs.

Name	Min BDD Size				Ave BDD Size				Standard Deviation			
	I	II	III	IV	I	II	III	IV	I	II	III	IV
bbgun	6565	7040	10250	10136	6746	7355	10290	10198	172	387	16	13
cf	783	803	1164	1173	853	882	1178	1201	45	51	5	9
cpab	850	866	1220	1234	884	927	1247	1266	22	37	18	20
dec	8213	9636	32499	32482	8550	9716	32528	32521	228	81	12	16
ex1inp	1570	1652	1470	1478	1626	1697	1491	1522	33	50	8	13
kirkman	1164	1082	668	661	1205	1155	678	675	17	40	4	5
master	77247	78600	229775	229829	78515	80589	229815	229894	995	1719	38	44
p21stg	5268	7193	5385	6984	5585	7754	5607	7407	253	339	169	236
planet	2098	2138	3626	3640	2131	2258	3675	3700	32	70	16	19
rpss	3531	3626	9261	9279	3591	3715	9281	9311	64	139	11	15
s1a	721	966	1362	1607	780	1047	1421	1688	36	71	36	71
sand	3473	3806	5554	5561	3598	4057	5590	5620	195	290	14	21
saucier	938	1027	2051	2048	993	1090	2073	2090	43	58	10	16
scf	165369	165825	s.o.	s.o.	165754	166314	s.o.	s.o.	519	482	s.o.	s.o.
slave	1232	1316	5467	5499	1334	1453	5497	5542	41	69	14	24
str	1747	2311	2223	2204	1859	2371	2241	2233	86	59	5	8
styr	805	932	1342	1419	869	956	1417	1499	23	27	33	55
viterbi	10602	10757	122902	122906	10722	10807	122982	122995	195	37	43	37