Exact Minimization of Boolean Decision Diagrams Using Implicit Techniques

Abstract

This paper addresses the problem of Boolean decision diagram (BDD) minimization in the presence of don't care sets. Specifically, given an incompletely specified function g and a fixed ordering of the variables, we propose an exact algorithm for selecting f such that f is a cover for g and the Boolean decision diagram for f is of minimum size. The approach described is the only known exact algorithm for this problem not based on explicit enumeration of all possible assignments for the points in the don't care set.

We show that the BDD minimization problem can be formulated as a binate covering problem and solved using implicit enumeration techniques. In particular, we show that the minimum-sized Boolean decision diagram compatible with the specification can be found by solving a problem that is very similar to the problem of reducing incompletely specified finite state machines. We report experiments of an implicit implementation of our algorithm, by means of which a class of interesting examples was solved exactly.

1 Introduction

A completely specified Boolean function f is a cover for an incompletely specified function g if the value of f agrees with the value of g for all the points in the input space where g is specified. This paper describes an exact algorithm for selecting f such that f is a cover for g and the Boolean decision diagram (BDD) for f has a minimum number of nodes (complemented edges are not considered here). For a given ordering of the variables, the BDD for f is unique [4] and the problem addressed has a well defined solution.

We show that this minimization problem can be solved by selecting a minimum sized cover for a graph that satisfies some additional closure conditions. In particular, we show that the minimum sized Boolean decision diagram compatible with the specification can be found by solving a covering problem that is very similar to the covering problem obtained using exact algorithms for the reduction of incompletely specified finite state machines (ISFSM) [8]. This similarity makes it possible to use implicit enumeration techniques developed for the purpose of ISFSM reduction [9] to solve efficiently the BDD minimization problem. The manipulation of the characteristic functions of the sets of compatibles and prime compatibles, represented with ROBDDs [2], allows the generation of very large sets that cannot be enumerated explicitly, as it is demonstrated in the experiments.

The transformation presented in this paper and the algorithms developed for the solution are important for practical and theoretical reasons.

From a practical point of view, there are applications in learning and logic synthesis where an high-quality solution is of paramount importance. This requires an exact algorithm to find those solutions or at least to validate the quality of heuristic algorithms.

For instance, in inductive learning applications, the accuracy of the inferred hypotheses is strongly dependent on the complexity of the result [1]. One possible and very effective representation scheme for inferred hypotheses are BDDs. However, it was observed [12] that when BDDs are used as the representation scheme, existing heuristic algorithms for BDD minimization find solutions that are so far from the minimum that makes them of little value for this particular application.

The selection of the minimum BDD consistent with an incompletely specified function is important also in logic synthesis applications that use BDDs not only as a tool for representing discrete functions but also to derive implementations that minimize some cost function. For instance, *timed Shannon circuits* [10] use the structure of the BDD to derive low power implementations and stand to gain from algorithms for the reduction of BDDs. The same holds for DCVS trees and multiplexer-based FPGAs.

An exact algorithm, even though unable to solve large instances, helps to measure the quality of heuristic algorithms by gauging them on instances where an exact solution can be found.

From a theoretical point of view, the transformation presented in this work is an elegant characterization of the problem and also may help in strenghtening results in the field of BDDs and branching programs [11] complexity theory. We show in Section 3 that our problem is NP-complete. Reduction of incompletely specified FSMs is NP-complete [14] too. Moreover, a stronger result is that the smallest DFA consistent with a set of accepted and rejected strings cannot be approximated within any polynomial factor of the minimum solution [15]. Given that the smallest consistent DFA problem can be trivially transformed into the ISFSM reduction [13] problem, efficient approximation algorithms for ISFSM reduction will not exist unless P=NP. This result cannot be used directly to classify the complexity of approximation algorithms for the BDD minimization problem. However, the similarities with ISFSM reduction raise the possibility that a similar result may hold in this case too.

Several heuristic algorithms for the problem addressed here have been proposed. These algorithms are important in applications where the available degrees of freedom in the functions represented can be used to reduce the memory requirements of BDD based algorithms. The restrict operator [7] and the constrain operator [6] (also known as generalized cofactor [19]) are two heuristics used to assign the don't cares of a BDD. A comprehensive study of heuristic BDD minimization has been presented in [16]. Another heuristic algorithm has been reported in [5].

We are aware also of work for an exact algorithm [17] based on the enumeration of the different covers that can be obtained by all possible assignments of the don't care points. A pruning technique reduces the enumeration process thanks to a result by Shiple that changing the value of a function f of n variables on a minterm m cannot change the size of the BDD for f by more than n nodes.

The remainder is organized as follows. Section 2 introduces basic definitions on BDDs and Section 3 has a proof that BDD minimization is NP-complete. Sections 4, 5 and 6 describe respectively the compatibility graph, closed clique covers and the generation of a minimum BDD. Minimization of BDDs is formulated as a variant of FSM minimization in Section 7, while an implicit algorithm to compute a minimum closed cover is presented in Section 8. Results and conclusions are offered respectively in Sections 9 and 10.

2 Preliminaries

A BDD is a rooted, directed, acyclic graph where each node is labeled with the name of one variable. and every non-terminal node n_i has one *else* and one *then* edge that point to the children nodes, n_i^{else} and n_i^{then}, respectively. The terminal nodes are n_z and n_o. By convention we will draw the else (zero) edge as the edge pointing to left (west), and the then (one) edge as the edge pointing to right (east).

Any minterm m in the input space induces a unique path in a BDD defined in the following way: start at the root and take, at each node, the *else* or the *then* edge according to the value assigned by minterm m to the variable that is the label of the current node until a terminal node is reached. A BDD corresponds to the completely specified Boolean function f that has all the minterms in f_{on} (and only these) inducing paths in the BDD that terminate in n_o. For a fixed ordering of the variables, the BDD for a given Boolean function is unique. This implies that BDDs are canonical representations of Boolean functions and we will therefore use the notation n_i to denote both the node in the BDD and the Boolean function it corresponds to.

The level of a node n_i, $\mathcal{L}(n_i)$ is the index of the variable tested at that node under the specific ordering used. The level of the terminal nodes is defined as $N+1$, where N is the number of input variables. The maximum level of a set s of nodes, $\mathcal{L}_{\max}(s)$, is the maximum level of all the nodes in s. A BDD is called *complete* if all edges starting at level i terminate in a node at level $i+1$.[1] The level of a function h, $\mathcal{L}(h)$, is defined as the level of a BDD node that implements h. If n_i is a node in the BDD and m a minterm, $n_i(m)$ will be used to denote both the value of function n_i for minterm m and the terminal node that m reaches when starting at n_i. This notation is consistent because the two terminal nodes stand for the constant

[1] A complete BDD will not, in general, be reduced.

functions 0 and 1. The index 0 will be reserved for the root of the BDD. Therefore, if m is a minterm and F is the BDD for f, $n_0(m)$ represents the value of f for minterm m.

A 3 Terminal BDD (3TBDD) is defined in the same way as a BDD in all respects except that it has three terminal nodes : n_z, n_o and n_x. A 3TBDD F corresponds to the incompletely specified function f that has all minterms in $f_{\rm off}$,$f_{\rm dc}$ and $f_{\rm on}$ terminate in n_z, n_x and n_o, respectively.

3 Complexity of the Problem

Consider the problem of minimum BDD identification.

Problem: MINIMUM BDD IDENTIFICATION (MBI)

Instance: A set of minterms, labeled either positive or negative and an integer K.

Question: For a given fixed ordering, is there a BDD with less than K nodes that satisfies all the examples, i.e., a BDD for a function whose on-set covers the positive examples and whose off-set covers the negative ones ?

Takenaga [18] proved that this problem is NP-complete, by reduction from graph K-colorability. The problem we address in this paper is the following:

Problem: EXACT BDD MINIMIZATION (EBM)

Instance: BDDs for functions $f_{\rm on}$ and $f_{\rm dc}$ and an integer K.

Question: Is there a BDD with less than K nodes that implements a function that is a cover for f ?

Proof that is it in NP (due to Shiple [16]):

Guess a BDD with fewer than K nodes. Check whether the guessed BDD implements a function that is cover of f. This check can be done in time and space upper bounded by the product of the sizes of the BDDs for $f_{\rm on}$ and $f_{\rm dc}$ and of the guessed BDD. This product is polynomial in the input size.

Proof that it is NP-hard:

Suppose we could solve this problem in polynomial time with a deterministic algorithm. Then we can also solve the MBI problem. First, note that the BDD for a given function f of v variables cannot have more than $n \times v$ internal nodes, where n is the number of minterms in f [17]. To verify this, consider all the paths through the BDD defined by all the minterms in f. This set of paths has to go through each internal node in the BDD for f at least once. Otherwise, there are nodes other than the constant node 0 in the BDD that are only reached by minterms in the *off* set of f, thereby implying that the BDD is not reduced. Because a minterm can only traverse v internal nodes, we obtain immediately the above result.

It follows by lemma 3.1 that the BDDs for $f_{\rm on}$ and $f_{\rm off}$ can be constructed in time polynomial in the size of the input instance of the MBI problem we want to solve. This implies that the BDD for $f_{\rm dc}$ can also be constructed in polynomial time in the size of the input of the MBI problem because it can be obtained by polynomial time bounded BDD operations. To solve an MBI problem, simply transform it into an EBM problem and solve it. The resulting solution will represent directly the answer to the original problem. $\Box$

Lemma 3.1 *A BDD of a function $f : B^v \rightarrow B$ represented by n minterms can be constructed in $O(n^2 v^2 \log n)$ operations.*

Proof:

Build the BDD of f from the minterms by doing the following: first OR together each pair of minterms. Then OR the results together, and keep doing this until the final result is computed. The number of such iterations k is logarithmic in n.

At iteration i, one needs to perform $n/(2^i)$ operations on BDDs no larger than $v \times 2^{i-1}$. Therefore, per iteration one needs no more than $n/(2^i)v2^{i-1}v2^{i-1} = nv^2 2^{i-2}$ elementary operations, that is upper bounded by n^2v^2, because $i \leq \log n$. Since there are only $\log n$ iterations, the result can be built in time and space $n^2v^2 \log n$. $\Box$

4 The Compatibility Graph

Previous algorithms [17] for this problem used directly the BDD representation of $f_{\rm on}$ and $f_{\rm off}$. The exact approach described in this paper uses the 3TBDD F that corresponds to the incompletely specified function f. F is assumed to be complete. If necessary, F is made complete by adding extra nodes that have the *then* and *else* edges pointing to the same node. In general, the resulting 3TBDD is no longer reduced. Moreover, we suppose that 3TBDD does not use complemented edges.

Definition 4.1 *Two nodes n_i and n_j in F are compatible ($n_i \sim n_j$) iff no minterm m exists that satisfies $n_i(m) = n_z \wedge n_j(m) = n_o$ or $n_i(m) = n_o \wedge n_j(m) = n_z$.*

This definition implies that n_o and n_z are not compatible between them and that n_x is compatible with any node in a 3TBDD.

Definition 4.2 *Two nodes n_i and n_j in F are common support compatible ($n_i \approx n_j$) iff there exists a* **completely specified** *function h such that $h \sim n_i$ and $h \sim n_j$ and $\mathcal{L}(h) \geq \max(\mathcal{L}(n_i), \mathcal{L}(n_j))$.*

The definition implies that $n_z \not\approx n_o$ and $n_x \approx n_i$, for any node n_i.

It is important, at this point, to understand the relationship between these two concepts. First, note that the completely specified function h referred in definition 4.2 does not necessarily correspond to any node in F. In fact, in most cases, h will not correspond to any node in F, since most nodes in F correspond to incompletely specified functions.

The relationship between compatibility and common support compatibility (CSC) is given by the following lemma:

Lemma 4.1 *If $n_i \approx n_j$ then $n_i \sim n_j$.*

Proof : If $n_i \not\sim n_j$, then there is a minterm m such that $n_i(m) \not\sim n_j(m)$. Any completely specified function h will assign a unique value to m, and therefore cannot be compatible with both n_i and n_j. □

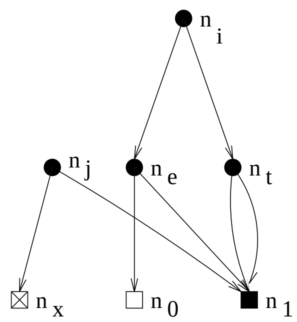

Figure 1: Nodes n_i and n_j are compatible but not common support compatible.

The reverse implication of lemma 4.1 is not true, in general. Figure 1 illustrates a situation where two nodes are compatible but are not CSC. Nodes n_i and n_j are compatible because no minterm leads to n_o for one of this nodes and to n_z to the other. However, n_i and n_j are not common support compatible because no completely specified function h that only depends on the second variable is compatible with both of them.

However, when two nodes belong to the same level, common support compatibility and compatibility are equivalent:

Lemma 4.2 *If $\mathcal{L}(n_i) = \mathcal{L}(n_j)$ then $n_i \sim n_j \Rightarrow n_i \approx n_j$.*

Proof : The completely specified function h required to satisfy definition 4.2 can be obtained by setting $h_{\rm on} = n_{j_{\rm on}} \cup n_{i{\rm on}}$ and $h_{\rm off} = \overline{h_{\rm on}}$. Since h depends only on the variables common to the supports of n_i and n_j, it can be used in definition 4.2 to show that $n_i \approx n_j$. □

The motivation for the definition of common support compatibility can now be made clear. Assume that two nodes belong to different levels and are compatible. In principle, they could be replaced by a new node that implements a function compatible with the functions of each node. In general, this function may depend on variables that are not on the support of the node at the higher level. Assume this node is n_j. Later, when we try to build the reduced BDD, edges that are incident into n_j will need to go upwards, against the variable ordering of the BDD. On the other hand, if both nodes are common support compatible, then they can be replaced by a node that implements the completely specified function h referred to in definition 4.2. Because this function only depends on the variables common to the supports of both nodes, this problem will not arise.

The concept of common-support compatibility can be extended to sets of nodes in the natural way:

Definition 4.3 *The nodes in the set $s_i = \{n_1, n_2, \ldots, n_s\}$ are common support compatible iff there exists a completely specified function h such that $(h \sim n_j)_{j=1,\ldots,s}$ and $\mathcal{L}(h) \geq \mathcal{L}_{\max}(s_i)$.*

Definition 4.4 *A set of nodes that are common support compatible is called a* **compatible set** *or, simply, a* **compatible**.

The definition of a compatible implies that any two nodes that belong to a compatible are pairwise common support compatible. The reverse implication is not true, but the next lemma holds.

Lemma 4.3 *Let s_i be a set of nodes belonging to the same level. Then, s_i is a compatible iff all nodes in s_i are pairwise common support compatible.*

Proof: To prove the *if* direction, note that, by the hypothesis, it can never happen that given a minterm m there are two nodes n_j and n_k in s_i that are CSC and satisfy $n_j(m) = n_z$ and $n_k(m) = n_o$. This would violate lemma 4.1. The function h that is needed to prove that s_i is a compatible set is the function h that has the value 0 for m if some n_j exists that satisfies $n_j(m) = n_z$ and has the value 1 for m if some n_k exists that satisfies $n_k(m) = n_o$ (the value for minterms not defined in this way can be chosen arbitrarily). Moreover $\mathcal{L}(h) = \mathcal{L}_{max}(s_i)$, since all nodes of s_i are at the same level as h. If one does not assume that all nodes of s_i belong to the same level, it is not guaranteed that h satisfies $\mathcal{L}(h) \geq \mathcal{L}_{max}(s_i)$, and one can build counterexamples to the lemma.

To prove the *only if* direction, note that if s_i is a compatible, then the function h referred to in definition 4.3 is compatible with any pair of nodes in s_i, thereby showing that they are pairwise CSC. □

Definition 4.5 *The* **compatibility graph**, *$G = (V, E)$, is an undirected graph that contains the information about which nodes in F can be merged. Except for the terminal node n_x, each node in F will correspond to one node in V with the same index. The level of a node in G is the same as the level of the corresponding node in F. Similarly, g_i^{else} and g_i^{then} are the nodes that correspond to n_i^{else} and n_i^{then}.*

Graph G is built in such a way that if nodes n_i and n_j are common support compatible then there exists an edge between g_i and g_j. An edge may have labels. A label is a set of nodes that expresses the following requirement: if nodes g_i and g_j are to be merged, then the nodes in the label also need to be merged. There are three types of labels: e, t and l labels. The following two lemmas justify the algorithm by which graph G is built:

Lemma 4.4 *If $\mathcal{L}(n_i) = \mathcal{L}(n_j)$ then $n_i \approx n_j \Rightarrow \left(n_i^{else} \approx n_j^{else} \wedge n_i^{then} \approx n_j^{then}\right)$.*

Proof : By contradiction. Since F is complete, the successors are at the same level. Therefore, by lemma 4.2, $n_i^{else} \not\approx n_j^{else} \Rightarrow n_i^{else} \not\sim n_j^{else}$ and a minterm m can be selected in such a way that $n_i^{else}(m) \not\sim n_j^{else}(m)$ and $m_{\mathcal{L}(n_i)} = 0$. The existence of this minterm shows that $n_i \not\sim n_j$ and therefore that $n_i \not\approx n_j$. A similar argument is true for the *then* branch. Therefore, $n_i^{else} \not\approx n_j^{else} \vee n_i^{then} \not\approx n_j^{then} \Rightarrow n_i \not\approx n_j$. □

Lemma 4.5 *If $\mathcal{L}(n_i) < \mathcal{L}(n_j)$ then $n_i \approx n_j \Rightarrow \left(n_i^{else} \approx n_j \wedge n_i^{then} \approx n_j \wedge n_i^{else} \approx n_i^{then}\right)$.*

Proof : By contradiction. If $n_i^{else} \not\approx n_j$ then, for any completely specified functions h at level $\mathcal{L}(n_j)$ or higher a minterm m can be selected in such a way that $n_i^{else}(m) \not\sim h(m)$ and $m_{\mathcal{L}(n_i)} = 0$. This minterm shows that $n_i \not\sim h$, thereby showing that $n_i \not\approx n_j$. Identically for the *then* branch. If $n_i^{else} \not\approx n_i^{then}$ then by lemma 4.2 there are minterms w and m such that $n_i^{else}(w) \not\sim n_i^{then}(w) \wedge w_{\mathcal{L}(n_i)} = 0$ and $n_i^{else}(m) \not\sim n_i^{then}(m) \wedge m_{\mathcal{L}(n_i)} = 1$. These minterms can be chosen to differ only in the value of the variable $x_{\mathcal{L}(n_i)}$ and lead to incompatible terminal nodes. Therefore, n_i cannot be compatible with any function h such that $\mathcal{L}(h) >= \mathcal{L}(n_j)$, thereby showing that $n_i \not\approx n_j$. □

The previous two lemmas justify the following algorithm to build the compatibility graph.

Algorithm 4.1

1. *Initialize G with a complete graph except for edge (g_z, g_o) that is removed.*

2. *If $\mathcal{L}(g_i) = \mathcal{L}(g_j)$ then the edge between g_i and g_j has two labels: an e label with $\{g_i^{else}, g_j^{else}\}$ and a t label with $\{g_i^{then}, g_j^{then}\}$. (By lemma 4.4.)*

3. *If $\mathcal{L}(g_i) < \mathcal{L}(g_j)$ edge (g_i, g_j) has an l label with $\{g_i^{else}, g_i^{then}, g_j\}$. (By lemma 4.5.)*

4. *For all pairs of nodes (g_i, g_j) check if the edge between nodes g_i and g_j has a label that contains $\{g_a, g_b\}$ and there is no edge between g_a and g_b. If so, remove the edge between g_i and g_j. Repeat this step until no more changes take place.*

Figure 2 shows an example of the 3TBDD F obtained from f defined by the following sets: $f_{\text{on}} = \{011, 111\}$, $f_{\text{off}} = \{010, 110, 101\}$ and the corresponding compatibility graph.

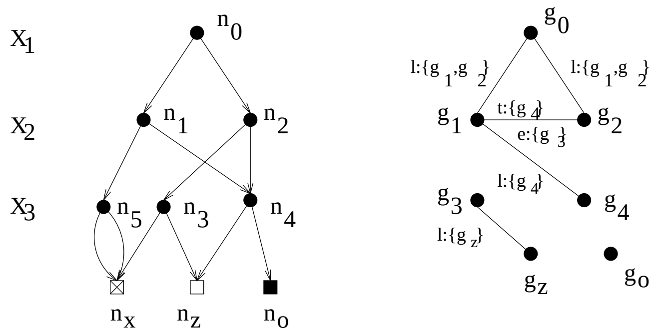

Figure 2: The 3TBDD F and the compatibility graph G. Nodes g_5 and g_x are not shown on the compatibility graph, since they are common support compatible with every node in the graph.

The existence of an edge in the incompatibility graph is related with common support compatibility and with compatibility between pairs of nodes in the following way:

Lemma 4.6 $n_i \approx n_j \Rightarrow \exists e \in E \text{ s.t. } e = (g_i, g_j) \Rightarrow n_i \sim n_j$.

Proof : The first implication follows from lemmas 4.4 and 4.5 and the algorithm definition. Given this lemmas, the algorithm only removes edges that are between nodes that are not CSC. Let us prove the second implication by contradiction. Now, let $I = \mathcal{L}(n_i)$ and $J = \mathcal{L}(n_j)$ and assume that $n_i \not\sim n_j$ and that $I \leq J$. Then, there exists a minterm m such that $n_i(m) \not\sim n_j(m)$. Let $n_{a_I}, n_{a_{I+1}} ... n_{a_{N+1}}$ be the sequence of nodes defined by minterm m in the BDD between the level I and $N+1$, starting at n_i. Let $n_{b_J}, n_{b_{J+1}} ... n_{b_{N+1}}$ be the sequence of nodes defined by minterm m between level J and $N+1$ starting at node n_j. Then, because $n_{a_{N+1}}$ and $n_{b_{N+1}}$ are the terminal nodes and there is no edge between them, after k steps, the edge between node $g_{a_{N+1-k}}$ and node $g_{b_{N+1-k}}$ will have been removed because of condition 2. This will happen for k such that $N+1-k >= J$. For k such that $N+1-k < J$, the edge between $g_{a_{N+1-k}}$ and g_{b_J} will have been removed because of condition 3. □

It is important to note that the reverse implications are not true. In particular, the existence of an edge between two nodes in G does not imply that they are common support compatible. Consider the 3TBDD shown in figure 3.

For this 3TBDD the algorithm described above does not remove the edge between nodes g_0 and g_5 because there are long range dependencies that can not by found by the simple minded algorithm used to prune away edges. The edge between g_0 and g_5 has the following l label: $\{g_5, g_1, g_2\}$.

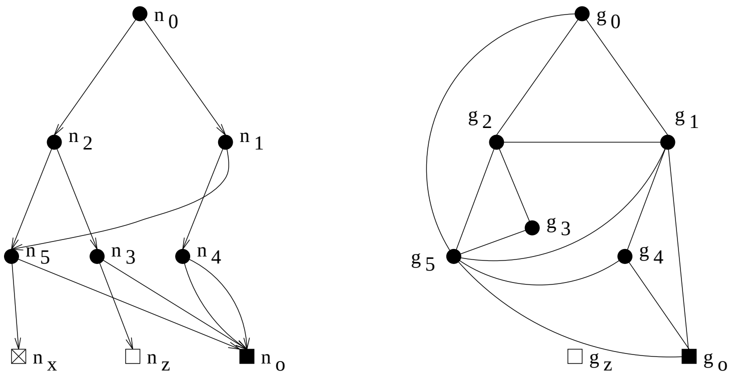

Figure 3: Nodes n_0 and n_5 are not common support compatible but the compatibility graph does have an edge between the corresponding nodes.

The edge between g_1 and g_5 has the l label $\{g_5, g_4\}$ and the edge between g_2 and g_5 has the l label $\{g_5, g_3\}$. Because n_5 is compatible with both n_3 and n_4, the edge between g_1 and g_5 and the edge between g_2 and g_5 are never removed. Moreover n_1 and n_2 are compatible. Therefore the edge between g_0 and g_5 is never removed. However, no function depending only on the last variable can be compatible with n_0, and therefore n_0 and n_5 are not common support compatible.

5 Closed Clique Covers

A clique of graph G is a completely connected subgraph of G. To any set s of nodes that is a clique of G there are associated class sets. If the nodes in s are to be merged into one, the nodes in its class sets are also required to be in the same set. Let $s_i = \{g_{i_1}, g_{i_2} ... g_{i_w}\}$ be a set of nodes that form a clique in G. The following are the definitions of the e, t and l classes of s_i. Notice that for concision we may blur the distinction between the nodes g's of G and the corresponding nodes n's of F. Strictly speaking, cliques are defined on sets of g's and compatibles on sets of n's.

Definition 5.1 *The e class of s_i, $C_e(s_i)$ is the set of nodes that are in some e label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(n_k) = \mathcal{L}(n_j) = \mathcal{L}_{\max}(s_i)$.*

Definition 5.2 *The t class of s_i, $C_t(s_i)$ is the set of nodes that are in some t label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(n_k) = \mathcal{L}(n_j) = \mathcal{L}_{\max}(s_i)$.*

Definition 5.3 *The l class of s_i, $C_l(s_i)$ is the set of nodes that are in some l label of an edge between a node g_j and g_k in s_i with $\mathcal{L}(g_j) \neq \mathcal{L}(g_k)$*

Lemma 5.1 *If a set s_i of nodes are a clique of G and $C_l(s_i) \subseteq s_i$, then s_i is a compatible set.*

Proof : Let k be the maximum level of any node in s_i. The definition of $C_l(s_i)$ implies that $C_l(s_i)$ includes all the nodes of s_i at level k that are descendents of some node in s_i. Call these nodes the *foundation* of s_i. Because these nodes are a clique of G (or else they wouldn't be in s_i), they are all pairwise compatible, by lemma 4.6. Because they are at the same level, lemma 4.2 implies that they are pairwise CSC. Therefore, by the result of lemma 4.3 these nodes are a compatible set. This implies that there exists a completely specified function h at the level $\mathcal{L}_{\max}(s_i)$ that is compatible with every node in the foundation of s_i. To finish the proof, we need to show that this function must be also compatible with every other node in s_i. To show this, assume that h is not compatible with some node n_j in s_i. Then, there must exist a minterm m such that $n_j(m) \not\sim h(m)$. This minterm defines a path in the BDD that goes through a node n_k in the foundation of s_i. Since $n_k(m) \not\sim h(m)$, n_k and h are not compatible, which violates the assumption that h is compatible with every node in the foundation of s_i. Therefore h must be compatible with every node in s_i, thereby satisfying definition 4.3. □

Note that a clique of G that does not satisfy the condition in lemma 5.1 is not necessarily a compatible set. For instance, in the example in figure 3 the nodes $\{g_0, g_1, g_2, g_5\}$ are a clique of G but are not a compatible set, because $g_3 \in C_l(\{g_0, g_1, g_2, g_5\})$ but $g_3 \notin \{g_0, g_1, g_2, g_5\}$.

The algorithm that selects the minimum BDD compatible with the original function works by selecting nodes of G that can be merged into one node in the final BDD. If a set s of nodes in G is to be merged into one, the set s has to be a compatible set. Therefore, it has to be a clique of G satisfying definition 5.3. The objective is to find a set of cliques such that every node in G is covered by at least one clique. However, to obtain a valid solution, some extra conditions need to be imposed.

Definition 5.4 *A set $S = \{s_1, s_2 ... s_n\}$ of sets of nodes in G is called a closed clique cover for G if the following conditions are satisfied:*

1. *S covers G : $\forall g_i \in G \, \exists s_j \in S \, : g_i \in s_j$*
2. *All s_k are cliques of G : $\forall g_i, g_j \in s_k \, : (g_i, g_j) \in \mathrm{edges}(G)$*
3. *S is closed with respect to the e and t labels :*
   $\forall s_i \in S \, \exists s_j \in S \, : C_e(s_i) \subseteq s_j \, \bigwedge \, \forall s_i \in S \, \exists s_j \in S \, : C_t(s_i) \subseteq s_j$
4. *All sets in S are closed with respect to the l labels : $\forall s_i \in S \, : C_l(s_i) \subseteq s_i$*

6 Generation of a Minimum BDD

From a closed clique cover for G, a reduced BDD R is obtained by the following algorithm:

Algorithm 6.1

1. *For each s_i in S, create a BDD node in R, r_i, at level $\mathcal{L}_{\max}(s_i)$.*
2. *Let the nodes in R that correspond to sets s_i containing nodes that correspond to terminal nodes in F be the new corresponding terminal nodes of R.*
3. *Let the* else *edge of the node r_i go to the node r_j that corresponds to a set s_j such that $C_e(s_i) \subseteq s_j$.*
4. *Let the* then *edge of the node r_i go to the node r_j that corresponds to a set s_j such that $C_t(s_i) \subseteq s_j$.*

Lemma 6.1 *R is an Ordered BDD compatible with F.*

Proof : Since the cover is closed, steps 3 and 4 four are always feasible. Any path in F that leads to a 1 or a 0 will lead to the corresponding terminal node in R. Finally, there will never be edges going upward in R because the node that results from a set s_i is at the lowest level of all the nodes in s_i. □

Now, the main result follows. Let $\mathcal{B}$ be the set of all BDDs that represent functions compatible with the incompletely specified function f. Then, the following result holds:

Theorem 6.1 *The BDD induced by a minimum closed cover for G is the BDD in $\mathcal{B}$ with minimum number of nodes.*

Proof : Given the result in lemma 6.1 it is sufficient to prove that there exists at least one closed cover of cardinality equal to the size of the minimum BDD in B.

Let U be a BDD in $\mathcal{B}$ with minimum number of nodes k. For each node in U, u_i, create a set s_i such that g_j is in s_i iff $n_j \sim u_i$ and $\mathcal{L}(n_j) \leq \mathcal{L}(u_i)$. Let $S = \{s_1, s_2 ... s_k\}$. We will show that S satisfies all the conditions in definition 5.4:

1. (S covers G) We show that the assumption that some g_i at level l is not in some set of S leads to a contradiction: let m be a minterm that defines a path in F that starts at the root and goes through n_i. Let M be the set of all minterms that have the same values as m for $x_1 ... x_{l-1}$. Each one of these minterms will define a path in U that goes through some node u_j in U at a level equal or higher than l. Since $n_i \not\sim u_j$ (by the hypothesis) there exists a minterm $m' \in M$ such that $u_j(m') \not\sim n_i(m')$. For this minterm m', $n_0(m') \not\sim u_0(m')$, thereby contradicting the assumption that U is compatible with F.

2. (All $s_i \in S$ are cliques of G) Since each node in s_i is compatible with a completely specified function (u_i) they satisfy definition 4.3 and therefore, by lemma 4.1, they are a clique of G.

3. (S is closed with respect to the e and t labels) Let u_i be a node in U, $u_a = u_i^{else}$ and $u_b = u_i^{then}$. Let $b_i = \{g_j \in s_i : \mathcal{L}(g_j) = \mathcal{L}_{\max}(s_i)\}$. For each node $g_j \in b_i$, $n_j \sim u_i$ implies $u_a \sim n_j^{else}$ and $u_b \sim n_j^{then}$. Therefore, $C_e(s_i) \subseteq s_a$ and $C_t(s_i) \subseteq s_b$.

4. (S is closed with respect to the l labels) Suppose $C_l(s_i) \not\subseteq s_i$. Then, there must be a node n_w such that $g_w \in s_i$ at level $l < \mathcal{L}(u_i)$ and $g_w^{else} \notin s_i$ or $g_w^{then} \notin s_i$. Assume the first is true and let $n_w^{else} = n_a$; n_a is not compatible with u_i (or else it would be in s_i) and depends only on the variables $\{x_{l+1} ... x_n\}$. Therefore, there exists a minterm m such that $u_i(m) \not\sim n_a(m)$ and $m_l = 0$. This minterm shows that $n_w \not\sim u_i$ which contradicts the hypothesis that g_w is in s_i.

Therefore, S is a closed clique cover for G and it has cardinality k. □

As an example, $S = \{\{g_0, g_1, g_2\}, \{g_4\}, \{g_3, g_5, g_z\}, \{g_o\}\}$ is a closed cover for the example depicted in figure 2 and induces the BDD R shown on the right side of figure 4.

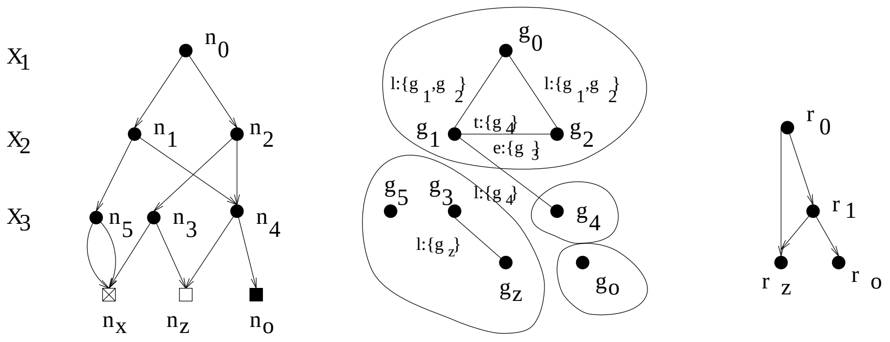

Figure 4: The 3TBDD F, the compatibility graph G and a solution R. Node g_5 was arbitrarily included in compatible $\{g_3, g_z, g_5\}$.

7 Formulation of BDD Minimization as FSM Reduction

The definition of a closed cover is very similar to the standard definition of a closed cover used in the minimization of finite state machines (FSMs). If the graph of a 3TBDD is viewed as the state transition graph of an FSM, the algorithms developed for the minimization of FSMs can be used with some modifications. The two important differences to consider are:

1. The definition of the e and t classes and the closure requirement in point 3 of definition 5.4 are different from the definitions used in standard FSM minimization. In BDD minimization, only nodes at the highest level in some compatible define the e and t classes, while in standard FSM minimization all nodes in a compatible set are involved in the definition of these classes.

2. The requirement in point 4 of definition 5.4 means that some sets of nodes that satisfy the definition of a compatible set in the FSM case do not satisfy the conditions for BDD minimization.

These two changes can be incorporated into existing algorithms for FSM minimization. In particular, the closure conditions with respect to the e and t labels are similar to the closure conditions imposed in standard FSM minimization. The restriction imposed by condition 4 in definition 5.4 simply eliminates some cliques of the compatibility graph from consideration and can be implemented by a filtering step.

Let F be the 3TBDD that should be minimized, and consider the FSM with a state transition graph (STG) obtained from F in the following way:

- Initialize the STG with a graph isomorphic to the 3TBDD, with nodes $S_0, S_1, \ldots, S_o, S_z, S_x$ each one corresponding to one node in F.
- Add a new node, S_f.
- Add transitions from S_z, S_o and S_x to S_f, labeled $-/0$, $-/1$ and $-/-$, respectively.
- Add a transition from S_f to S_f labeled $-/-$.

As an example, consider the FSM obtained from the 3TBDD in figure 2, shown in figure 5.

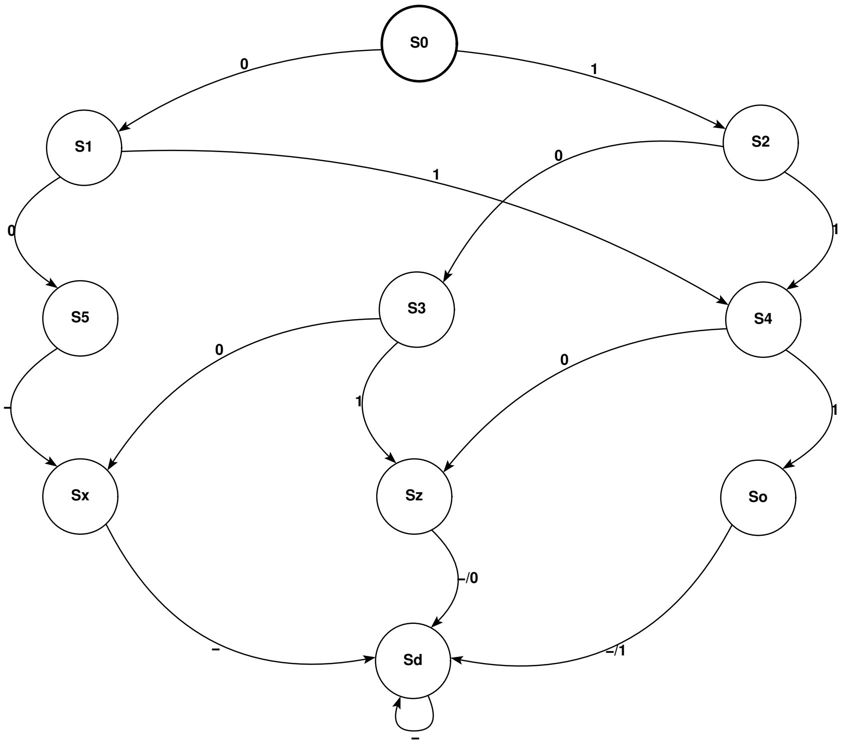

Figure 5: The STG that corresponds to the 3TBDD F defined in figure 2.

This transformation leads to our final important result. Let M be the incompletely specified FSM with the state transition graph obtained from F by the procedure outlined above and let G' be the compatibility graph for this finite state machine built in the following way:

- There is an edge between nodes g'_i and g'_j in G' if states n'_i and n'_j are compatible, in the FSM reduction sense.
- Edges of G' are labeled in accordance with algorithm 4.1

Note that the structure of graph G' reflects the compatibility between states of M defined as for FSM reduction. Therefore, G' can be computed using the standard procedures for FSM minimization. However, this means that G' is not isomorphic to G, defined by algorithm 4.1. In particular, any two nodes that are at different levels in G' are compatible for FSM reduction, and so G' always has an edge between g'_i and g'_j if $\mathcal{L}(g'_i) \neq \mathcal{L}(g'_j)$, and therefore the second implication in lemma 4.6 is not true, in general.

The following lemma establishes that the cliques of G and the cliques of G' that are closed with respect to the l class are exactly the same.

Lemma 7.1 *A set $s'_i = \{g'_{a_1} \dots g'_{a_k}\}$ is a clique of G' satisfying $C_l(s'_i) \subseteq s'_i$ iff $s_i = \{g_{a_1} \dots g_{a_k}\}$ is a clique of G satisfying $C_l(s_i) \subseteq s_i$.*

Proof : Since the edges of G' are a superset of the edges of G and the labels are the same, is is clear that to any clique of G containing $C_l(s_i)$ there corresponds a clique of G' satisfying that condition. To prove the other direction, note that for any two nodes in G' at the same level, the presence of an edge in G' implies that they are compatible, both in the sense of FSM reduction and according to definition 4.1. Therefore, the rightmost implication of lemma 4.6 is valid for nodes that are at the same level. The proof of lemma 5.1 only uses this implication for nodes at the same level, namely the nodes in the foundation of s_i defined in that lemma. Therefore, lemma 5.1 is still true if G is replaced by G' implying that any set s'_i in G' that satisfies $C_l(s'_i) \subseteq s'_i$ corresponds to a compatible of G', according to definition 4.3. Therefore, the corresponding set s_i in G is a clique. Because the labels are the same for any edges common to both G and G', $C_l(s_i) \subseteq s_i$.

Corollary 7.1 *A minimum closed cover for M satisfying definition 5.4 when G is replaced by G' induces a minimum BDD compatible with F, in accordance with theorem 6.1.*

Proof: Since the compatibles that can be part of the cover are the same in both G' and G, this result follows directly from theorem 6.1.

8 Implicit Computation of a Minimum Closed Cover

We will use the unified implicit framework proposed in [9] [2]. Implicit techniques are based on the idea of operating on discrete sets by their characteristic functions represented by binary decision diagrams (BDDs) [4].

To perform state minimization, one needs to represent and manipulate efficiently sets of sets of states. With n states, each subset of states is represented in **positional-set** form, using a set of n Boolean variables, $x = x_1 x_2 \dots x_n$. The presence of a state s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if state s_k is not a member of the set. For example, if $n = 6$, the set with a single state s_4 is represented by 000100 while the set of states $s_2 s_3 s_5$ is represented by 011010.

A set of sets of states S is represented in positional notation by a characteristic function $\chi_S : B^n \to B$ as: $\chi_S(x) = 1$ if and only if the set of states represented by the positional-set x is in the set S. A BDD representing $\chi_S(x)$ will contain minterms, each corresponding to a state set in S. As an example, $Tuple_{n,k}(x)$ denotes all positional-sets x with exactly k states in them (i.e. $|x| = k$). For instance, the set of singleton states is $Tuple_{n,1}(x)$, the set of state pairs is $Tuple_{n,2}(x)$, the set of full states is $Tuple_{n,n}(x)$, and the set of empty states is $Tuple_{n,0}(x)$. An alternative notation for $Tuple_{n,k}(x)$ is $Tuple_k(x)$.

Any **relation** R between pairs of sets S_1 and S_2 can be represented by its characteristic function $\mathcal{R} : B^n \times B^m \to B$ where $\mathcal{R}(x, y) = 1$ if and only if $\chi_{S_1}(x) = 1$, $\chi_{S_2}(y) = 1$ and the element of S_1 represented by x is in relation R with the element of S_2 represented by y. A similar definition holds for relations defined over more than two sets. For example, we represent the state transition graph (STG) of an FSM by the characteristic functions of two relations:

1. the output relation Λ, where input i, present state p and output o are in $\Lambda(i, p, o)$ if there is an edge from p with input/output label i/o, and
2. the next state relation $\mathcal{T}$, where where input i, present state p and next state n are in relation $\mathcal{T}(i, p, n)$ if there is an edge from p to n with input label i.

[2] $\exists x(\mathcal{F})$ $(\forall x(\mathcal{F}))$ denotes the existential (universal) quantification of function $\mathcal{F}$ over variables x; $\Rightarrow$ denotes Boolean implication; $\Leftrightarrow$ denotes XNOR; $\neg$ denotes NOT.

Proposition 8.1 *Set* **equality, containment** *and* **strict-containment** *between two positional-sets x and y are expressed by:* $(x = y) = \prod_{k=1}^{n}(x_k \Leftrightarrow y_k)$; $(x \supseteq y) = \prod_{k=1}^{n}(y_k \Rightarrow x_k)$; *and* $(x \supset y) = (x \supseteq y) \cdot (x \neq y)$.

Proposition 8.2 *Given two sets of positional-sets,* **complementation, union, intersection** *can be performed on them as logical operations* $(\neg, +, \cdot)$ *on their characteristic functions.*

Definition 8.1 *The* **substitution** *in the function $\mathcal{F}$ of variable x_i with variable y_i is denoted by:*

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

and the substitution in the function $\mathcal{F}$ of a set of variables $x = x_1 x_2 \ldots x_n$ with another set of variables $y = y_1 y_2 \ldots y_n$ is obtained simply by:

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

Proposition 8.3 *The* **maximal** *of a set χ of subsets is the set containing subsets in χ not strictly contained by any other subset in χ, and can be computed as:*

$$Maximal_x(\chi) = \chi(x) \cdot \nexists y \; [(y \supset x) \cdot \chi(y)].$$

8.1 Implicit Generation of Compatibles

It has been shown in Section 7 that given a BDD minimization problem it is possible to generate a companion FSM whose closed covers of compatibles correspond to closed clique covers of the BDD, if:

- FSM compatibles that do not satisfy the L-closure are discarded, and
- FSM compatible closure is replaced by E-closure and T-closure.

Our starting point is the fully implicit algorithm for exact state minimization reported in [9], to which we refer for a complete description of the implicit computations. In the sequel we discuss the modifications needed to generate closed clique covers of the BDD.

8.2 Implicit Computation of L-closure

We compute as in [9] the set of compatibles $\mathcal{C}(c)$, where $\mathcal{C}(c) = 1$ iff c is the positional set representing a compatible of the companion FSM. When minimizing an FSM obtained from an instance of BDD minimization one must delete from $\mathcal{C}(c)$ the compatibles c that are not closed with respect to their l-class. The l-class, $\mathcal{C}_l(c)$, of a compatible c is the set of nodes that are in some l-label of an edge between nodes g_j and g_k in c with $\mathcal{L}(g_j) < \mathcal{L}(g_k)$. If $\mathcal{L}(g_j) < \mathcal{L}(g_k)$ then edge (g_j, g_k) has the l-label $\{g_j^{else}, g_j^{then}, g_k\}$.

$\mathcal{L}_0(p) = r(p)$
$k = 0$
do {
  $\mathcal{L}_{k+1}(p) = [n \to p]\exists p, i\; [\mathcal{L}_k(p) \cdot \mathcal{T}(i, p, n)]$
  $k = k + 1$
} until $\mathcal{L}_{k+1}(p) = \mathcal{L}_k(p)$

Figure 6: Computation of array $\mathcal{L}$.

It is shown next how to capture the information on the level of the nodes. By construction, an FSM obtained by BDD minimization is represented by a direct acyclic STG rooted at the unique reset state r; each node has two successors, except the terminal node that has a self-loop. Fig. 6 illustrates a procedure to build an array $\mathcal{L}(p)$, that partitions the FSM states based on their distance from the root: $\mathcal{L}_k(p)$ is the set of states associated to the nodes having a distance k from r. Starting from r and visiting in breadth-first order the STG, one computes iteratively the array elements $\mathcal{L}_k(p)$, using the transition relation $\mathcal{T}(i,p,n)$. In fact, state n is a successor of state p iff $\exists i\ \mathcal{T}(i,p,n)$.

Using the informations stored in $\mathcal{L}(p)$, one defines the order relation $Level(p,u)$, for each couple of states (p,u) in the FSM. States p and u are in relation $Level(p,u)$ iff the distance of p from r is less than the distance of u from r, i.e. formally

$$Level(p,u) = 1 \Leftrightarrow \exists i\ \exists j\ \{(i<j) | \mathcal{L}_i(p) \cdot \mathcal{L}_j(u)\} \tag{1}$$

Fig. 7 illustrates the procedure to compute the global relation $Level(p,u)$.

```
Level(p,u) = ∅
for (i = 0; i < k; i++) {
    for each state Tuple_1(p) ∈ L_i(p) {
        for (j = i; j < k; j++) {
            for each state Tuple_1(u) ∈ ([u → p]L_j(p)) {
                Level(p,u) = Level(p,u) + (Tuple_1(p) · Tuple_1(u))
            }
        }
    }
}
```

Figure 7: Computation of the relation $Level(p,u)$.

A compatible c is pruned from the set of compatibles $\mathcal{C}(c)$ if:

1. c contains states p and u that are in the order relation $Level(p,u)$,
2. c does not contain all the successors of p.

Hence, the filtered set of compatibles is given by:

$$\begin{aligned}\mathcal{C}(c) \quad = \quad & \mathcal{C}(c) - \exists p\ \{\exists u\ [Tuple_1(p) \cdot Tuple_1(u) \cdot (c \supseteq p) \cdot (c \supseteq u) \cdot Level(p,u)] \\ & \exists n\ [\exists i\ \mathcal{T}(i,p,n) \cdot (c \not\supseteq n)]\}\end{aligned} \tag{2}$$

8.3 Implicit Computation of E-closure and T-closure

In standard FSM minimization one requires closure with respect to implied sets. Given a compatible c an implied set under input i is the set of next states from the states in c under i. Instead in the case of BDD minimization one must compute the implied sets only from the states in c of highest level. This requires a change in the computation of the relation of the implied classes $\mathcal{F}(c,i,n)$, which is used by the following procedures:

1. the computation of primes,
2. the set up of the binate clauses in the covering table,
3. the construction of a reduced FSM.

The new computation for $\mathcal{F}(c,i,n)$ is described by the following equation:

$$\mathcal{F}(c,i,n) \quad = \quad \exists p\ \{\exists c'\ [\mathcal{C}(c) \cdot Max\_Level(c,c') \cdot (c' \supseteq p)] \cdot \mathcal{T}(i,p,n)\} \tag{3}$$

Subsets of states c and c' are in relation $Max\_Level(c,c')$, iff c' is the subset of c that contains the states of c of maximum level, i.e. the states having the largest distance from r in the STG of the FSM.

$Max\_Level(c,c') = \emptyset$
$\mathcal{C}_j(c) = \mathcal{C}(c)$
for $(j = k-1; j = 0; j--)$ {
$$\mathcal{ML}_j(c,c') = \mathcal{C}_j(c) \cdot \exists l[\mathcal{L}_j(l) \cdot \sum_{n=1}^{N}(l_n \cdot c_n)] \cdot \prod_{n=1}^{N}\{c'_n \Leftrightarrow [c_n \cdot \exists l(l_n \cdot L_j(l))]\}$$
$Max\_Level(c,c') = Max\_Level(c,c') + \mathcal{ML}_j(c,c')$
$\mathcal{C}_j(c) = \mathcal{C}_j(c) - \exists c'[\mathcal{ML}_j(c,c')]$
}

Figure 8: Computation of the relation $\mathcal{C}(c).Max\_Level(c,c')$.

The computation of the relation $\mathcal{C}(c).Max\_Level(c,c')$ is based on the availability of $\mathcal{L}(p)$ and is summarized in Fig. 8. For each level j starting from the maximum to the minimum, a relation $\mathcal{ML}_j(c,c')$ is determined performing N bitwise conjunctions, where N is the number of states. The n-th element of c' is 1 iff the n-th element of c is 1 and it has level j. $\mathcal{ML}_j(c,c')$ represents the pairs (c,c') such that c is a compatible that contains at least one state at level j and no state at level greater than j, and c' has exactly the states of c of level j. Before examining level $j-1$, $Max\_Level(c,c')$ is updated adding the elements in $\mathcal{ML}_j(c,c')$, and the sets c already in $\mathcal{ML}_j(c,c')$ are removed from $\mathcal{C}(c)$. Notice that the time complexity of the computation depends only linearly from the explicit parameters N, number of states, and k, number of levels in the STG representation.

8.4 Example

Figure 5 illustrates the companion FSM obtained from the 3TBDD F shown in figure 2. The FSM has 2 pairs of incompatible states $(\{S_3, S_4\}, \{S_z, S_o\})$ while its set $\mathcal{C}(c)$ contains 575 compatibles. After filtering away by means of equation 2 the compatibles c that are not closed with respect to their l-class, 32 sets of compatibles are left:

$\{S_d\}, \{S_o\}, \{S_o, S_d\}, \{S_z\}, \{S_z, S_d\}, \{S_x\}, \{S_x, S_z\}, \{S_x, S_z, S_d\}, \{S_x, S_o\}, \{S_x, S_d\}, \{S_x, S_o, S_d\}, \{S_5\}, \{S_5, S_x\}, \{S_5, S_x, S_z\}, \{S_5, S_x, S_z, S_d\}, \{S_5, S_x, S_o\}, \{S_5, S_x, S_d\}, \{S_5, S_x, S_o, S_d\}, \{S_4\}, \{S_4, S_5\}, \{S_3\}, \{S_3, S_x, S_z\}, \{S_3, S_x, S_z, S_d\}, \{S_3, S_5\}, \{S_3, S_5, S_x, S_z\}, \{S_3, S_5, S_x, S_z, S_d\}, \{S_2\}, \{S_1\}, \{S_1, S_2\}, \{S_1, S_4, S_5\}, \{S_0\}, \{S_0, S_1, S_2\}$.

Using the computation of the implied classes $\mathcal{F}(c,i,n)$ of equation 3 the following 8 primes are identified from the previous 32 compatibles:
$\{S_0\}, \{S_0, S_1, S_2\}, \{S_1\}, \{S_1, S_4, S_5\}, \{S_2\}, \{S_3, S_5, S_x, S_z, S_d\}, \{S_4\}, \{S_5, S_x, S_o, S_d\}$.
Among the 8 primes, there are 2 essential primes:
$\{S_3, S_5, S_x, S_z, S_d\}, \{S_5, S_x, S_o, S_d\}$
and 6 nonessential primes:
$\{S_0\}, \{S_0, S_1, S_2\}, \{S_1\}, \{S_1, S_4, S_5\}, \{S_2\}, \{S_4\}$.
After solving the binate covering problem, 2 nonessential primes are chosen:
$\{S_0, S_1, S_2\}, \{S_1, S_4, S_5\}$.
Hence, the final reduced FSM has the following 4 states:

$R_0 \leftarrow \{S_0, S_1, S_2\}$
$R_1 \leftarrow \{S_1, S_4, S_5\}$
$R_z \leftarrow \{S_3, S_5, S_x, S_z, S_d\}$
$R_o \leftarrow \{S_5, S_x, S_o, S_d\}$

and is described by the following state transition table:

$$
\begin{array}{ccc}
0 & R_0R_z & - \\
1 & R_0R_1 & - \\
0 & R_1R_z & - \\
1 & R_0R_o & - \\
- & R_zR_z & 0 \\
- & R_oR_o & 1
\end{array}
$$

This state transition table induces the BDD R shown on the right side of figure 4. R is an exact solution of the BDD minimization problem.

9 Results

Starting from the program ISM for implicit state minimization [9] we developed IMAGEM, a new program based on the theory described in this paper for exact BDD minimization. In particular, we transformed the implicit algorithm for exact state minimization in a new algorithm for the implicit computation of a minimum closed cover as described in Section 8.

example	# orig. states	# compat.	# filtered compat.	# prime compat.	# red. states	*heuristic*	IMAGEM CPU time (sec)
dnfa	64	2.435821e+12	1332186	89	14	*16*	1388.9
dnfb	36	4.853883e+08	2987	94	6	*12*	28.98
dnfc	40	2.291224e+08	2613	102	10	*15*	33.43
dnfd	93	1.137739e+20	9.517899e+08	-	-	*23*	timeout
dnfe	63	2.102303e+13	141179	509	6	*12*	559.88
dnff	62	2.184367e+11	92027	357	15	*22*	389.3
xor3	9	179	14	7	6	*6*	0.28
xor4	17	14975	118	13	6	*6*	1.18
xor5	24	608255	267	36	9	*10*	3.73
xor6	40	3.355914e+08	1329	170	13	*20*	34.89
xor7	57	2.791115e+11	3076	640	15	*31*	208.38
xor8	94	1.539147e+17	164929	21830	17	*45*	25751.51
ex.paper	10	575	32	8	4	4	0.32

Table 1: **Results** on Machine Learning Problems.

To evaluate experimentally the algorithms presented in this paper, we assembled two sets of problems: the first set derives directly from a machine learning application and the second set was obtained from a logic synthesis benchmark. In all the problems, the original ordering specified for the variables was the ordering used.

For the first set of problems, 12 completely specified Boolean functions f_i were used as the starting point. For each of these functions, a randomly selected set of minterms was designated as the *care set*, resulting in a set of incompletely specified Boolean functions g_i. The original objective was to identify the set of problems for which it is possible to recover exactly the original functions f_i from the incompletely specified functions g_i, thereby characterizing the conditions under which it is possible to infer the original function from a training set [12]. For the purposes of this work, the functions g_i are used solely as a set of incompletely specified functions. An advantage that exists for this set is that upper bounds on the size of the solution are well defined, because the BDD sizes for the f_i are known. Under certain conditions, these upper bounds tend to become tight, with high probability, as the size of the problem increases, providing a welcome check for the results obtained.

The second set of problems was obtained by selecting a subset of the problems that are distributed with *Espresso* [3], a well known two-level minimizer. More specifically, we included in this set of problems the functions that are the first output from each of the PLAs that are included in the *industry* subset of the *Espresso* benchmark suite. From this set, we eliminated all the functions that have a null don't care set, since, for these functions, the problem is trivial.

Table 1 summarizes the results obtained from running the set of machine learning problems and Table 2 the ones from the problems derived by the *Espresso* benchmark suite. The last entry in table 1 is denoted *ex.paper* and simply refers to the case that has been presented in the paper to illustrate the theory.

example	# orig. states	# compat.	# filtered compat.	# prime compat.	# red. states	*heuristic*	IMAGEM CPU time (sec)
alu1	95	1.025649e+21	841993	1204	6	*6*	20887.33
br1	74	2.99581e+18	799173	329	6	*11*	3759.63
br2	51	5.937363e+14	53687	78	3	*8*	38.72
clpl	50	1.467671e+13	7559	39	3	*13*	32.07
dc2	46	8.277148e+10	8831	98	8	*12*	149.43
exp	54	2.695432e+11	10638	25	3	*3*	85.08
exps	71	1.8345e+10	3810	125	43	*44*	120.49
in0	151	2.622416e+25	1680740	1323	42	*44*	53036.33
in3	173	5.060229e+39	587880	12	9	*14*	4623.66
inc	35	1.119744e+07	364	26	12	*13*	9.88
mark1	71	7.487812e+18	8049	35	4	*5*	106.6
newapla	52	1.24299e+12	3252	33	10	*11*	107.1
newapla1	57	8.766887e+14	8733	63	6	*6*	360.27
newapla2	19	93311	137	6	5	*5*	1.29
newbyte	16	20735	127	9	5	*5*	1.05
newcpla2	39	3.396557e+08	477	68	10	*21*	14.57
newcwp	16	10367	106	10	6	*11*	1.01
newtpla	94	1.265561e+23	411525	148	7	*23*	731.99
newtpla1	39	6.912e+09	1441	31	4	*5*	11.34
newtpla2	26	3149279	158	9	9	*9*	2.4
newxcpla1	39	4.470682e+09	1473	35	5	*10*	13.1
p82	16	15551	102	10	7	*7*	1.01
prom1	65	5.189184e+09	382	77	50	*50*	76.53
prom2	33	2.17728e+08	446	38	12	*12*	8.55
sex	28	1.679616e+07	419	16	5	*5*	4.13
spla	155	1.647427e+39	1.401835e+12	-	-	*8*	timeout
sqn	41	1.05336e+07	173	43	19	*19*	23.39
t4	68	5.108787e+14	31775	157	9	*11*	234.43
vg2	150	3.655064e+36	4.038678e+07	-	-	*14*	timeout
wim	14	4319	82	8	6	*6*	0.68

Table 2: **Results** on problems from *Espresso* benchmark suite

For each example of a 3TBDD the number of states of the companion FSM is reported in the column denoted "*# orig. states*". This number is always equal to the number of nodes of the 3TBDD plus one because a new node is added to the STG as explained in Section 7. The following two columns report the number of compatibles of the FSM (i.e. the cardinality of the set $\mathcal{C}(c)$) and the number of compatibles after filtering as per Section 8.2 (i.e. the ones which are closed with respect to their l-class). This step reduces the number of compatibles of many order of magnitude.

Then, after the number of primes, in column "*# red. states*" we reported the number of states of the reduced FSM. This number coincides with the number of nodes of the final BDD and represents the exact solution of the BDD minimization problem. Instead, the column denoted with the label "*heuristic*" presented the solutions which can be obtained using the restrict operator [7], a well-known heuristic algorithm for BDD minimization; the same numbers [3], can be obtained also using the constrain operator [6] (also known as generalized cofactor [19]). Therefore, it is clear that IMAGEM represent effectively the first exact algorithm which can be used to measure the quality of the heuristics for BDD minimization.

Moreover, as we discussed in section 1, there are specific applications, as for instance inductive learning, where from one side BDD are used as very effective representation scheme, but on the other side heuristic algorithms return intrinsically unsatisfactory solutions. The results reported in table 1 show that IMAGEM is able to return the exact solution for this class of non trivial problems.

The last column contains the time spent by IMAGEM to find the solution: all run times are reported in CPU seconds on a DEC Alpha with 1Gb of memory. For all experiments, timeout has been set at 86400 seconds of CPU time and spaceout at 1Gb of memory.

10 Conclusions

This paper addresses the problem of Boolean decision diagram (BDD) minimization in the presence of don't care sets. Specifically, given an incompletely specified function g and a fixed ordering of the variables, we propose an exact algorithm for selecting f such that f is a cover for g and the Boolean decision diagram for f is of minimum size. We show that the minimum-sized Boolean decision diagram compatible with the specification can be found by solving a problem that is very similar to the problem of reducing an ISFSM. The approach described is the only known exact algorithm for this problem not based on an explicit enumeration of all possible assignments for the points in the don't care set.

We show that this minimization problem can be formulated as a binate covering problem and solved using implicit enumeration techniques. We have implemented this algorithm and performed experiments, by means of which exact solutions for an interesting benchmark set were computed. In particular we could solve exactly some non-trivial examples from the learning literature, where quality of the solution is of paramount importance.

The current bottleneck of our implicit computation is the step from filtered compatibles to prime compatibles. It would be interesting to study new techniques for the implicit computation of prime compatibles or of a superset of them, in order the enlarge the set of examples that can be solved exactly.

11 Acknowledgments

The authors thank Timothy Kam for very interesting discussions on the reduction of BDD minimization to FSM state minimization.

References

[1] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Inform. Proc. Lett.*, 24:377–380, April 1987.

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

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

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

[3] Notice that for the heuristic algorithms the sizes of the BDD have been measured without considering complement edges.

[5] S.-C. Chang, D.I. Cheng, and M. Marek-Sadowska. Minimizing ROBDD size of incompletely specified multiple output functions. In *The Proceedings of the European Design and Test Conference*, pages 620–624, March 1994.

[6] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using functional Boolean vectors. *Proceedings of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, November 1989.

[7] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, vol. 407 of Lecture Notes in Computer Science*, pages 365–373, June 1989.

[8] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[9] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *The Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

[10] L. Lavagno, P. McGeer, A. Saldanha, and A. L. Sangiovanni-Vincentelli. Timed Shannon Circuits: A Power-Efficient Design Style and Synthesis Tool. In *Proceedings of the* 32^{th} *Design Automation Conference*, pages 254–260, June 1995.

[11] Cristoph Meinel. *Modified Branching Programs and Their Computational Power*. Springer-Verlag, 1989.

[12] Arlindo L. Oliveira. *Inductive Learning by Selection of Minimal Complexity Representation*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, December 1994. Memorandum No. UCB/ERL M94/97.

[13] Arlindo L. Oliveira and Stephen A. Edwards. Inference of state machines from examples of behavior. Technical report, UCB/ERL Technical Report M95/12, Berkeley, CA, 1995.

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

[15] L. Pitt and M. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.

[16] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. Brayton. Heuristic minimization of BDDs using don't cares. In *The Proceedings of the Design Automation Conference*, pages 225–231, June 1994.

[17] R. Ranjan T. Shiple, R. Hojati. Exact minimization of BDDs using don't cares. EE290ls Project Report, May 1993.

[18] Yasuhiko Takenaga and Shuzo Yajima. NP-completeness of minimum binary decision diagram identification. Technical Report COMP 92-99, Institute of Electronics, Information and Communication Engineers (of Japan), March 1993.

[19] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *The Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.