HEURISTIC MINIMIZATION OF BDDs USING DON'T CARES

by

Thomas R. Shiple, Ramin Hojati,
Alberto L. Sangiovanni-Vincentelli,
and Robert K. Brayton

Memorandum No. UCB/ERL M93/58

20 July 1993

HEURISTIC MINIMIZATION OF BDDs USING DON'T CARES

by

Thomas R. Shiple, Ramin Hojati,
Alberto L. Sangiovanni-Vincentelli
and Robert K. Brayton

Memorandum No. UCB/ERL M93/58

20 July 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

HEURISTIC MINIMIZATION OF BDDs USING DON'T CARES

by

Thomas R. Shiple, Ramin Hojati,
Alberto L. Sangiovanni-Vincentelli
and Robert K. Brayton

Memorandum No. UCB/ERL M93/58

20 July 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Heuristic Minimization of BDDs Using Don't Cares

Thomas R. Shiple Ramin Hojati
Alberto L. Sangiovanni-Vincentelli Robert K. Brayton
Department of EECS, University of California, Berkeley, CA 94720

July 20, 1993

Abstract

We present heuristic algorithms for finding a minimum BDD size cover of an incompletely specified function, assuming the variable ordering is fixed. In some algorithms based on BDDs, incompletely specified functions arise for which any cover of the function will suffice. Choosing a cover which has a small BDD representation may yield significant performance gains. We present a systematic study of this problem, establishing a unified framework for heuristic algorithms, proving optimality in some cases, and presenting experimental results.

1 Introduction

The problem addressed is, given an incompletely specified Boolean function $\mathcal{F}$, find a cover for $\mathcal{F}$ whose reduced ordered binary decision diagram [2] (hereafter, BDD) representation is minimum. $\mathcal{F}$ is described by a pair of completely specified Boolean functions f and c, such that any cover of $\mathcal{F}$ must contain $f \cdot c$ and must be contained by $f + \overline{c}$. The usual interpretation is that we *care* about the value of f where c is true, and we *don't care* where c is false.

To make these notions concrete, consider Figure 1. Figures 1a and 1b show the BDDs for f and c, respectively. Figure 1c shows the binary decision *tree* for f. The left and right branches are the 0 and 1 branches, respectively. The leaves enclosed by a square indicate those points where $c = 0$: that is, where we don't care about the value of f. Finally, Figure 1d shows a suboptimal solution to this problem, and Figures 1e and 1f show two minimum solutions.

Coudert et al. posed this problem in the context of checking the equivalence of two finite state machines (FSMs) [4]. The check is done by a breadth-first traversal of the state space of the product machine. At each iteration, the states on the frontier of the search are explored. Since there is no harm in re-exploring states that have already been reached, the goal is to choose a set of states S that includes the frontier states U and is included in the reached states R. The characteristic function for S should have a small BDD representation. In this case, we take $f = U$ and $c = U + \overline{R}$.

Another application is the representation of an FSM using BDDs. We represent the transition relation $T \subseteq Q \times S \times Q$, where Q is the set of states and S the set of inputs, by the BDD for the characteristic function of T. Typically, only the behavior and structure of the FSM among

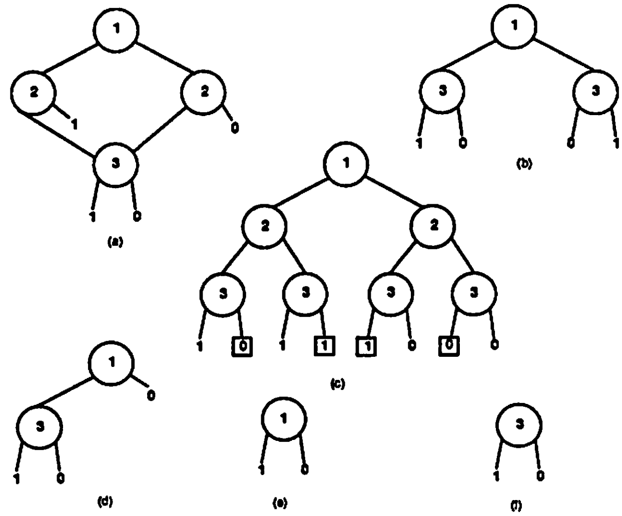

Figure 1: An instance of the problem: a) BDD for f, b) BDD for c, c) binary decision tree for f, annotated with don't care points (leaves enclosed by squares), d) suboptimum solution, e) an optimum solution, f) an optimum solution (left branch is 0, right branch is 1).

the states reachable from the initial states are of interest. Under this scenario, transitions *from* unreachable states are don't cares and can be used to simplify the BDD for T.

Other applications are found where circuit realizations are related to the structure of BDDs. In particular, some FPGA mapping algorithms work from a BDD representation to map circuits to multiplexer-based FPGAs [8]. For an incompletely specified circuit, heuristically minimizing the BDD can lead to a smaller implementation. Another application is in mapping Boolean functions to differential cascode voltage switch (DCVS) trees [6]. Because of the structural similarity of BDDs and DCVS circuits, minimizing the BDD leads to a smaller implementation.

Two heuristics have been reported for solving this problem: the *restrict* operator [4] and the *constrain* operator [3] (also known as the *generalized cofactor* [10]).[1] In this paper we present a general framework for heuristic solutions to finding minimum BDD size covers for incompletely specified functions. Our heuristics are based on the concept of making two BDD nodes equal by assigning values to some of their don't care (DC) points. We call this operation *matching.* We present a hierarchy of matching criteria, depending on how much don't care information is required to match two functions.

The compactness of BDDs derives from two rules: *merging*, which shares equal functions, and *deletion*, which deletes a parent wtih equal children. We present algorithms to exploit these rules. Specifically, one set of algorithms matches various functions in the same level of a BDD, hence sharing more subfunctions. A second set matches siblings in order to delete parents.

[1]It is worth noting that the *constrain* operator has a special property which permits an image computation of a *vector* of functions to be reduced to a range computation on the vector. This property arises because *constrain* uses the don't care points in a very restricted fashion. For this study, we are not interested in such properties: any cover of a given incompletely specified function is a candidate solution to this problem.

We view DC assignments as using degrees of freedom. At every point, several competing options may exist on how to use the DCs. We present scheduling algorithms which attempt to use the DC points in an optimal fashion.

The main contributions of this paper are:

1. We define a general framework to relate various heuristic solutions to the problem. The framework consists of matching criteria and the choice of functions to be matched.
2. We view each heuristic as a transformation which uses some of the DC freedom. Traditionally, only one heuristic is used during the optimization process. We have uncoupled the choice of transformations from the choice of where and when they should be applied. We present a schedule that uses different heuristics at different points in the optimization process. For example, we try to match siblings before matching arbitrary functions. Also, we first apply a restricted matching criterion before applying a more general criterion.
3. We prove several optimality results. Among the significant ones: *constrain* is optimum when the care set is a cube; and, for a certain matching criterion, matching functions at a given level is optimum with respect to the number of nodes below that level.

In Section 2 we define some terms and give a precise statement of the problem. Section 3 presents two classes of heuristic minimizers, and a method to combine them using scheduling. Experimental results are given in Section 4 and concluding remarks in Section 5.

2 Problem Statement

Let $B = \{0, 1\}$, and $x_1, \ldots, x_n$ be the variables of the space B^n. All functions considered are defined on $x_1, \ldots, x_n$.

Definition 1 A *literal* is a variable in its true or complement form (e.g. x_i or $\overline{x_i}$). A *cube* is a conjunction of a set of literals (e.g. $x_2\overline{x_4}x_5$). The *cofactor* of f by the literal a, denoted by f_a, is f evaluated at $x_i = 0$ if $a = \overline{x_i}$, or f evaluated at $x_i = 1$ if $a = x_i$.

We refer the reader to [1] for the definition of *reduced ordered binary decision diagrams* (BDD). A *binary decision tree* for a function is the full binary decision tree, before any reductions are applied. Two rules are applied to a binary decision tree to yield a BDD: *merging*, which shares two subfunctions that represent the same function, and *deletion*, which removes a node with equal children.

Our BDD package is based on [1] and employs output complement pointers to reduce storage requirements. A fixed variable ordering of $x_1 \prec x_2 \prec \ldots \prec x_n$, where x_1 is the topmost variable, is used for all BDDs. We use f to refer to both the function and its BDD representation. *Level i* refers to those nodes of a BDD rooted at x_i.

Definition 2 The *size* of the BDD f, denoted by $|f|$, is the number of nodes in the BDD, including the constant (terminal) node.

$[f, c]$ denotes an incompletely specified function, where $f \cdot c$ is the *onset*, $\overline{f} \cdot c$ the *offset*, and $\overline{c}$ the *don't care* set. When not ambiguous, $[f, c]$ is simply called a function.

Definition 3 g is a *cover* of $[f, c]$ if $f \cdot c \subseteq g \subseteq f + \overline{c}$. $[f_1, c_1]$ is an *i-cover* ("i" for incompletely specified) of $[f_2, c_2]$ if any cover of $[f_1, c_1]$ is a cover of $[f_2, c_2]$.

We will see later that when two incompletely specified functions are matched, they are replaced by their common i-cover. Now we formally state the problem that is addressed.

Definition 4 The exact BDD minimization (EBM) problem is to find a cover g of $[f, c]$ such that $|g|$ is minimum among all covers of $[f, c]$, under a fixed variable ordering. The corresponding decision problem for EBM is:
INSTANCE: BDDs for functions f, c, positive integer $N < |f|$.
QUESTION: Is there a cover g for $[f, c]$ such that $|g| < N$?

Proposition 5 The decision problem for EBM is in NP.

Proof Guess a BDD structure for g that has fewer than N nodes. If $f \cdot c \subseteq g \subseteq f + \overline{c}$, then g is a cover with less than N nodes. The containment checks can be done in time and space $O(|f| \cdot |c| \cdot |g|)$, and thus in time and space polynomial in the size of the input. ∎

The exact complexity of EBM is unknown. Finally, note that the problem of finding a cover with a minimum BDD size for the interval of functions (f_m, f_M), can be reduced to an instance $[f, c]$ of EBM by taking $c = f_m + \overline{f_M}$ and $f_m \subseteq f \subseteq f_M$.

3 Heuristic Minimization Algorithms

3.1 Framework

The general idea is to apply transformations to $[f, c]$ by selectively assigning values to don't cares until all have been used. A common aspect is the notion of "matching" a pair of functions $[f_j, c_j]$ and $[f_k, c_k]$ by finding a common i-cover. When one exists, we say the two functions *match*. The care function of the common i-cover contains c_j and c_k; thus, the size of the DC set monotonically decreases. Various constraints, or *matching criteria*, are defined according to which don't cares are used in finding a common i-cover.

All our heuristic algorithms iteratively apply three steps until the don't cares are exhausted:

Criterion	Reflexive	Symmetric	Transitive
osdm	no	no	yes
osm	yes	no	yes
tsm	yes	yes	no

Table 1: Properties of the matching criteria.

1. Choose a matching criterion.
2. Choose a set S of incompletely specified subfunctions of $[f, c]$.
3. Minimize the number of incompletely specified functions needed to i-cover the functions in S. Replace each function in S with its appropriate i-cover to yield a new function $[f', c']$.

The matching criteria are discussed in Section 3.1.1. The choice at Step 2 defines two classes of heuristics. If we restrict S to the two children of a given node, then Step 3 simply tries to replace these by a single function. This class of heuristics is described in Section 3.2. On the other hand, if we choose a subset of functions below level i, which are pointed to from level i or above, then we have an optimization problem in Step 3. This class of heuristics is described in Section 3.3.

3.1.1 Matching Criteria

We have experimented with three matching criteria, which are defined below.

Definition 6 Let $[f_1, c_1]$ and $[f_2, c_2]$ be incompletely specified functions.

1. One-sided DC match: $[f_1, c_1]$ *osdm* $[f_2, c_2]$ iff $c_1 = 0$. That is, one function is matched to another iff the first function has don't cares at all of its points.
2. One-sided match: $[f_1, c_1]$ *osm* $[f_2, c_2]$ iff $f_1 \oplus f_2 \subseteq \overline{c_1}$ and $\overline{c_1} \supseteq \overline{c_2}$. That is, one function is matched to another iff we can make the two equal by assigning DCs of only the first function, and the DC set of the first contains the DC set of the other.
3. Two-sided match: $[f_1, c_1]$ *tsm* $[f_2, c_2]$ iff $f_1 \oplus f_2 \subseteq \overline{c_1} + \overline{c_2}$. That is, two functions are matched iff we can make the two equal by assigning DCs from both functions.

Each matching criterion is a relation between incompletely specified functions. Table 1 lists some properties of these relations that are used in the sequel.

It is easy to prove for each criterion above, that if the matching definition is satisfied, then a common i-cover exists. If $[f_1, c_1]$ matches $[f_2, c_2]$, we want to find a common i-cover with maximal don't care part. In other words, if a DC point need not be assigned to make the match, we leave it unassigned. Thus, when a match is made, we produce the following:

1. $osdm : [f_2, c_2]$
2. $osm : [f_2, c_2]$
3. $tsm : [(f_1c_1 + f_2c_2), (c_1 + c_2)]$

There is a *strength hierarchy* implied by the above numbering, since

$$(c_1 = 0) \Rightarrow (f_1 \oplus f_2 \subseteq \overline{c_1}) \Rightarrow (f_1 \oplus f_2 \subseteq \overline{c_1} + \overline{c_2}), \quad \text{then}$$

$$([f_1, c_1]\ osdm\ [f_2, c_2]) \Rightarrow ([f_1, c_1]\ osm\ [f_2, c_2]) \Rightarrow ([f_1, c_1]\ tsm\ [f_2, c_2]).$$

Since f is itself a cover of $[f, c]$, it would be nice to have a single algorithm for solving EBM which never returns a result larger than $|f|$. However, we show that any non-optimal algorithm, based on the above matching criteria, cannot have this property.

Proposition 7 Let *alg* be any algorithm for solving EBM which is not sensitive to the value of f where $c = 0$, for a given instance $[f, c]$. Then there exists an instance $[f', c']$ where *alg* returns a result larger than $|f'|$ iff there exists an instance where *alg* is not optimum.

Proof ($\Leftarrow$) Let $[f, c]$ be an instance where *alg* is not optimum. Let $alg(f, c) = g$, and suppose a minimum cover for $[f, c]$ is $\hat{f}$. Now, create a new instance $[\hat{f}, c]$ where $f = \hat{f}$ on the care points. Since *alg* is insensitive to the value on the don't care points, then $alg(\hat{f}, c) = g$. Since $|g| > |\hat{f}|$, then $[\hat{f}, c]$ is an instance where *alg* increases the size.

($\Rightarrow$) If *alg* increases the size, then it is not optimum. ■

Of course, in practice we can compare the size of the result with the original f, and return the smaller of the two. Such an "algorithm" does not contradict the proposition since it is implicitly sensitive to the values of f on the don't care points.

In the special case $0 \neq c \subset f$, all the algorithms find the minimum solution, which is just $g = 1$. This follows since we always assign a don't care point the value of a care point, which in this case is always 1. Similarly, when $c \subset \overline{f}$, the 0 function is returned.

3.2 Matching Siblings

The heuristics based on matching "siblings" are motivated by the *constrain* and *restrict* operators. For a given subfunction $[f_j, c_j]$ of $[f, c]$, rooted at level i, we say that $[f_j\_E, c_j\_E]$ and $[f_j\_T, c_j\_T]$ are *siblings*, where f_j_E is f_j evaluated at $x_i = 0$ and f_j_T is f_j evaluated at $x_i = 1$ (likewise for c_j). The intuition behind these heuristics is that if two siblings can be matched, then both the parent node and one child node can be eliminated.

The heuristics simultaneously traverse f and c in a depth-first fashion, applying a given matching criterion to the children of each node visited. In the case that one sibling matches the other, we can eliminate the parent node by returning the result of recursing on the i-cover of $[f_j\_E, c_j\_E]$ and $[f_j\_T, c_j\_T]$. In the case where the siblings don't match, we recurse on each child, and return a node rooted at x_i and pointing to the results of the two recursions. Thus far, we have experimented with using only a single matching criterion throughout the traversal. However, one can imagine applying different criterion depending on the context.

Since BDDs with complemented output pointers are used, if two siblings cannot be matched in their uncomplemented forms, then it would seem beneficial to try matching one sibling to the *complement* of the other sibling. In this case, the parent node remains, but we need recurse on only one incompletely specified function.

The other condition for which we test is inspired by the *restrict* operator: if f_j is independent of x_i (i.e. $f_j\_E = f_j\_T$), then we keep it so by not attempting to match the children. This is accomplished by returning the result of recursing on the function $[f_j,\ c_j\_E + c_j\_T]$. The intuition behind this rule, called *no-new-vars*, is that it seems detrimental to introduce a new variable into the support of f_j. However, this is not always the case [7]: let f be a function independent of x with a "large" BDD, and let $c = xf + \overline{x}\,\overline{f}$. Then, by introducing x into the support, a cover for $[f, c]$ of size two results, namely, the function x.

It is never beneficial to introduce a variable which is in neither the support of f nor c. All of our algorithms guarantee that this never happens.

Thus, there are three parameters in our generic, top-down approach to matching siblings: 1) a matching criterion, 2) a match-complement flag, and 3) a no-new-vars flag. Different combinations of these parameters give rise to the heuristics listed in Table 2. Two of the heuristics are simply *constrain* and *restrict*. There are four heuristics listed which are not unique: since checking for a complement match has no effect on *osdm*, 3 and 4 are the same as 1 and 2, respectively; and since no-new-vars has no effect on *tsm*, 10 and 12 are the same as 9 and 11, respectively.

Pseudo-code for the generic top-down approach is presented in Figure 2. The first call to *bdd_get_branches* returns the *then* and *else* branches of f if $fId = topId$. Otherwise, (when f is independent of $topId$) it just returns f for both branches. The calls to *bdd_get_branches* keep the traversals through f and c in lock-step by splitting f and c only when their top variables are $topId$. The function *is_match* takes as input the matching criterion, the complement flag, and a pair of incompletely specified functions. If a match can be made (for *osdm* and *osm*, it tries in both directions), then it returns the i-cover.

It is easy to find small counter-examples to show that none of these heuristics are optimal. We give a few here. To specify a function, the values of the function on the leaves of the binary decision tree are listed from left to right, as suggested by Figure 1c. A don't care value for an incompletely specified function is indicated by d. For each example, we give the instance of the problem, the solution found by the heuristic, and a minimum solution, in that order.

1. *constrain*: (d1 01), (11 01), (01 01).
2. *osm_td*: (d1 01 1d 01), (01 01 11 01), (11 01 11 01).
3. *tsm_td*: (1d d1 d0 0d), (10 01 10 01), (11 11 00 00).

	Matching Criterion	match-compl	no-new-vars	Name/ Comment
1	*osdm*	no	no	*constrain*
2	*osdm*	no	yes	*restrict*
3	*osdm*	yes	no	same as 1
4	*osdm*	yes	yes	same as 2
5	*osm*	no	no	*osm_td*
6	*osm*	no	yes	*osm_nv*
7	*osm*	yes	no	*osm_cp*
8	*osm*	yes	yes	*osm_bt*
9	*tsm*	no	no	*tsm_td*
10	*tsm*	no	yes	same as 9
11	*tsm*	yes	no	*tsm_cp*
12	*tsm*	yes	yes	same as 11

Table 2: Heuristics based on matching siblings.

In addition, these examples demonstrate that for some heuristics, one heuristic is not always better than another. In particular, comparing *constrain*, *osm_td* and *tsm_td*, both *osm_td* and *tsm_td* find a minimum in example 1, *constrain* and *tsm_td* in example 2, and *constrain* and *osm_td* in example 3.

In the special case where c is a cube, all the algorithms do find a minimum solution. The intuition behind this is that for two subfunctions $[f_j, c_j]$ and $[f_k, c_k]$ rooted at a given level, when c is a cube, then either c_j or c_k is zero, or $c_j = c_k$. In the first case, if the care function of a function is zero, then that function will be "eliminated" entirely. In the latter case, if there exists a common cover for the two functions, implying they agree on the care points, then a common cover will be found, even though both subfunctions are minimized separately.

Theorem 8 Let $[f, c]$ be an incompletely specified function where c is a cube. Then *constrain* produces a minimum solution to EBM.

Proof By Proposition 7, we only need to show that the size of the BDD is never increased. The key is that for each node in the BDD for f, at most one node can be created in the result.

Consider the algorithm in Figure 2 when it is specialized to the case of *constrain* and c a cube. Since *no-new-vars* and *compl* are FALSE, the conditions at 2 and 4 will never be true. Since c is a cube, then if c depends on $topId$, a match will always be found at condition 3 because one child will be don't care. If c is independent of $topId$, line 5 will be executed, where $topId$ will be fId.

Hence, the only points where nodes can be created are lines 1 and 5. In both cases, nodes are created at locations in the binary decision tree where a node exists in the corresponding location in f. To complete the proof, we need to show that a shared node does not become "unshared".

```
    function generic_td(mcrtn, compl, no_new_vars, f, c) {
       assert (c ≠ 0);
1      if (c = 1 or is_constant(f)) return f;
       if (cache_lookup(f, c, &ret)) return ret;

       fId = get_var_id(f); cId = get_var_id(c); topId = MIN(fId, cId);
       bdd_get_branches(f, &f_T, &f_E, topId);
       bdd_get_branches(c, &c_T, &c_E, topId);

2      if ((f is independent of cId) and (no_new_vars = TRUE)) {
             /* keep f independent of cId */
          ret = generic_td(mcrtn, compl, no_new_vars, f, (c_T + c_E));
3      } else if (is_match(mcrtn, FALSE, f_T, c_T, f_E, c_E, &new_f, &new_c)) {
             /* matched siblings without complement */
          ret = generic_td(mcrtn, compl, no_new_vars, new_f, new_c);
4      } else if (compl and is_match(mcrtn, TRUE, f_T, c_T, f_E, c_E, &new_f, &new_c)) {
             /* matched siblings with complement */
          temp = generic_td(mcrtn, compl, no_new_vars, new_f, new_c);
          ret = topId · \overline{temp} + \overline{topId} · temp;
       } else {
             /* no match can be made */
          temp_T = generic_td(mcrtn, compl, no_new_vars, f_T, c_T);
          temp_E = generic_td(mcrtn, compl, no_new_vars, f_E, c_E);
5         ret = topId · temp_T + \overline{topId} · temp_E;
       }

       cache_insert(f, c, ret);
       return ret;
    }
```

Figure 2: The generic algorithm for matching siblings in a top-down fashion.

Consider a node in f with multiple incoming pointers. We must argue that node creation occurs at most once for such a node. The key observation is that all the non-zero care functions for this node, associated with the different incoming pointers, are the same since c is a cube. Thus, in each non-zero instance, the subfunction is optimized in exactly the same manner, whether node creation occurs at lines 1 or 5. In fact, the result is found in the cache on subsequent calls. Thus, whatever sharing occurred before, still occurs. function. ∎

The theorem for the other heuristics can be argued similarly. As a side note, Touati, et al. [10] showed that *constrain* just reduces to the Shannon cofactor when c is a cube.

3.3 Minimizing at a Level

The heuristics based on matching siblings take a local approach by just trying to match siblings. In this section a more global approach is taken, trying to match as many functions as possible at a given level in the BDD.

The basic procedure is to first choose a level i at which to apply minimization. The second step is to choose a set of incompletely specified functions below level i. For this set, a "matching graph" is constructed according to a selected matching criterion, indicating which functions can be matched. The graph is "solved" to yield a set of i-covers for the functions. Finally, the original f and c are updated with the new subfunctions. This procedure is called "minimizing at level i"; the individual steps are detailed in the following subsections.

3.3.1 Choosing Functions to be Examined

In minimizing at level i, we try to minimize the number of nodes pointed to from level i or above. This is done by matching subfunctions $[f_j, c_j]$, such that both f_j and c_j are pointed to from level i or above. Such subfunctions are gathered by traversing the BDDs for f and c in depth-first order, terminating the recursion whenever a pair of nodes both below level i are reached. Only unique pairs are added to the set.

Since this set may grow very large, we propose two methods to limit the size. The first simply limits the size of the set. When the limit is reached, the resulting set is processed. Then the traversal is continued, building a new set. An advantage to this method is that subfunctions which are nearby in the BDD will be grouped together, enhancing the possibility of reduction.

The second method is to add only subfunctions $[f_j, c_j]$ such that f_j is rooted at level $i+1$. This effectively minimizes the number of nodes at level $i+1$. These two methods are orthogonal and can be combined. In our current implementation, we do not limit the size of the set, preferring to trade runtime for quality. The largest set encountered so far had size 513, for a BDD with approximately 5000 nodes.

A major expense in this procedure is performing a complete traversal of the BDD down to level i, every time a different level is selected for optimization. However, if i is simply incremented at each step, it may be possible to make the traversal incremental.

3.3.2 Matching A Set of Functions

The previous step produces a set of incompletely specified functions. The next step is to match as many as possible to reduce the final BDD size.

Definition 9 Given a set of incompletely specified functions S and a matching criteria mat, the function matching minimization (FMM) problem using mat is to find a minimum set of incompletely specified functions R, such that for each function in S there exists an i-cover in R. Furthermore, it must be possible to obtain each function $[f,c]$ in R by performing matchings using *mat* among the functions in S that are i-covered by $[f,c]$.

For each matching criterion a matching graph is defined. We then show how to process the graph to solve FMM. First, we look at *osm*.

Definition 10 The *directed matching graph* (DMG) for the *distinct* functions $[f_1,c_1],\ldots,[f_r,c_r]$ is a directed graph with r vertices, and with a directed edge from vertex j to k iff $[f_j,c_j]$ *osm* $[f_k,c_k]$.

Proposition 11 Let H be the DMG for a set S of incompletely specified functions. Assume H has m vertices, k of which are sink vertices (i.e. vertices with no outgoing edges). Then, a minimum solution to the FMM problem using *osm* has k functions.

Proof First note that H is acyclic. This follows since *osm* is transitive, and if $[f_1,c_1]$ *osm* $[f_2,c_2]$ and $[f_2,c_2]$ *osm* $[f_1,c_1]$, then the two incompletely specified functions are equal, i.e. they have the same values on their care points, and have the same don't care functions. However, by definition of DMG, the incompletely specified functions must be distinct.

Choose the k functions corresponding to the sink vertices, to be in the set R. These are the functions that cannot be matched to any other functions in S. Now, any function in S can be matched to one of the functions in this set, since *osm* is transitive. Furthermore, if two different functions match a third function, then by definition of *osm*, the third function is a common i-cover for the two functions. There cannot be any smaller set than R, since the functions in R cannot be matched to any other function in S. ∎

We can solve FMM for *osm* by simply performing a depth-first search on the DMG and gathering the functions at the sink vertices as the i-covers. Note that Definition 10 and Proposition 11 carry over when the matching criterion is *osdm*; we do not discuss this case further.

Since an *osm* match uses don't cares from only one of the functions, we can prove that applying minimization at level i using *osm* does not lose the optimum solution below level i. By this, we mean that there exists an assignment to the remaining DC points such that the number of nodes below level i is equal to the number of nodes below level i in some minimum solution. The intuition behind this is that when $[f_j,c_j]$ is matched to $[f_k,c_k]$ using *osm*, $[f_j,c_j]$ need not be implemented, while the full freedom for $[f_k,c_k]$ is preserved. The caveat is that applying *osm* at level i may lose the optimum solution in the *superstructure* at and above level i.

Definition 12 Let $N_i(g)$ be the number of nodes below level i in the BDD for g, and $N_i[f,c]$ be the minimum of $N_i(g)$ over all covers g of $[f,c]$.

Theorem 13 Assume that a set of *osm* matchings is performed at level i for function $[f,c]$, resulting in $[f',c']$. Then, there is a cover g' of $[f',c']$ such that $N_i(g') = N_i[f,c]$.

Proof Let g be a cover of $[f, c]$ such that $N_i(g) = N_i[f, c]$. Consider the subfunctions of $[f, c]$ pointed to from at or above level i remaining after the *osm* matches are made. Since, we have not used any of the DCs of these functions (because we have done only *osm* matchings), we still have the same freedom for these functions that we originally had. We could just assign these DCs as they are assigned in g. The conclusion follows after noticing that there are no extra BDD nodes for the nodes that have already been matched. ∎

If minimization is applied near the top, then the number of nodes in the superstructure is small. Hence, this result implies that applying *osm* near the top will keep us near the optimum solution. As a corollary, we have the following.

Corollary 14 *osm* matching at level 1 does not lose the optimum solution.

Proof There are only 2 functions pointed to from level 1. If a match is made between these functions, then there are no nodes in the superstructure (i.e. level 1) after matching. Hence, by the theorem, the optimality is not lost. If a match is not made, then no don't cares are assigned, so the original freedom remains. ∎

For the *tsm* case we proceed in a fashion similar to *osm*, except that since the matching graph is undirected, solving FMM is not as straightforward.

Definition 15 The *undirected matching graph* (UMG) for the functions $[f_1, c_1], \ldots, [f_r, c_r]$ is an undirected graph with r vertices, and with an edge between vertex j and k iff $[f_j, c_j]$ *tsm* $[f_k, c_k]$.

Lemma 16 The functions $[f_1, c_1], \ldots, [f_r, c_r]$ have a common cover iff $[f_j, c_j]$ *tsm* $[f_k, c_k]$ for all $1 \leq j, k \leq r$.

Proof ($\Leftarrow$) Let $m \in B^n$. We need to show that a cover exists for each function such that each of these covers has the same value on m. Assume to the contrary: then there exist $[f_j, c_j]$ and $[f_k, c_k]$ such that $c_j(m) = c_k(m) = 1$ and $f_j(m) \neq f_k(m)$; but, this contradicts the assumption that $[f_j, c_j]$ *tsm* $[f_k, c_k]$.

($\Rightarrow$) If they have a common cover, then they match pairwise. ∎

FMM using *tsm* can be reduced to the graph-theoretic problem of covering the vertices of a graph with a minimum number of cliques.

Theorem 17 Let H be the UMG for a set S of incompletely specified functions. Then a minimum clique cover for H is a minimum solution to FMM using *tsm*.

Proof Assume that a minimum clique cover for H is given and is of size K. By Lemma 16, all the functions of a clique have a common cover. For each clique, produce an i-cover by matching all the functions in the clique. The set of i-covers produced in this manner yields a solution to FMM using *tsm* of size K.

To prove that a minimum solution to FMM using *tsm* has size of at least K, suppose there is a solution of size less than K. By Lemma 16, for each set of matched functions in such a solution, we can create a clique. The set of these cliques covers H, and hence is a clique cover of size less that K, a contradiction. ∎

Since the clique partitioning problem is NP-complete [5], heuristics are used. The following algorithm returns a clique cover of an undirected graph.

1. Start with some uncovered vertex v. Let $cur\_set = v$.
2. For each outgoing edge (u, w) of *cur_set*, where w is not in *cur_set*, check whether w has an edge to all the vertices in *cur_set*. If it does, add w to *cur_set*. If there are no such edges, go back to step 1, reporting *cur_set* as a clique.

We implemented this algorithm for our experiments. In addition, we propose two optimizations to find larger cliques containing matches of "nearby" functions.

1. Assume vertex v is in a 2-clique and a 10-clique. If the vertex corresponding to the 2-clique is visited first, then the 10-clique is missed. To avoid such situations, the vertices are processed in decreasing order of the number of outgoing edges, i.e. the vertices with more outgoing edges are processed first.
2. Functions that are siblings (or near-siblings) may match, but may be placed in different cliques, depending on the order in which vertices are visited when constructing the cliques. It generally seems beneficial to make such local matches where possible. To encourage such matches to be selected, we assign a weight to each match indicating the *distance* between two functions. For a subfunction g, let x_i^g denote the value on x_i used to reach g. Then the distance between two functions g and h rooted at level k is defined as[2]:

   $$dist(g, h) = \sum_{i=1}^{k-1} |x_i^g - x_i^h| 2^{k-i-1}$$

   This sum is over i such that neither x_i^g nor x_i^h is d (a don't care). For example, if g and h are siblings, then $dist(g, h) = 1$. Or, if the path to g is $1000d10$ and the path to h is $1d01111$, then $dist(g, h) = 9$.

   In building a clique, we would like to choose edges with smaller weights. To do this, the outgoing edges of *cur_set* are processed in ascending order of weights. Now, the edges with smaller weights have greater chance of being selected.

For our experiments, we have implemented one heuristic from the class of heuristics based on matching at levels. This heuristic, *opt_lv*, visits the levels in increasing order, and uses *tsm* to match functions.

[2]Based on the distance measure defined in [10].

3.4 Scheduling

Our heuristics fall into two distinct classes, sibling matching and matching at a level. However, better results might be achieved by *scheduling* the basic transformations outlined in Sections 3.2 and 3.3. The idea is to apply safer transformations first. These have less possibility of losing the optimal solution, and consume less don't care information. Then, potentially more powerful, but less safe, transformations are used. We propose the following schedule, whose theoretical justification derives from the fact that *osm* can only lose the optimal solution in the superstructure.

Apply the following top-down, with *initial_level* = 0:

1. Consider the window of *initial_level* through *initial_level* + *window_size*, where *window_size* is a given parameter.
2. Apply *osm* on siblings top-down in the window.
3. Apply *tsm* on siblings top-down in the window.
4. Apply *osm* on levels top-down in the window.
5. Apply *tsm* on levels top-down in the window.
6. If the number of remaining levels is less than *stop_to_down*, a given parameter, call a bottom-up minimizer, and stop. Otherwise, let *initial_level* = *initial_level* + *window_size.*

At each iteration, only the functions in a given window are considered. The idea is that if a match can be made using *tsm* in higher levels at the expense of losing *osm* matches in the lower levels, we may save BDD nodes. As we progress down the BDD, we cannot save many nodes by making matches at higher levels; so, it may be advantageous to apply a bottom-up minimizer to assign the rest of the DCs.

We can trade runtime for quality by choosing which optimizers to apply. Applying minimization at a level is generally expensive, so steps 4 and 5 should be skipped if runtime is a concern. Experimental verification of what values work well for *window_size* and *stop_to_down* remains.

4 Experiments

4.1 Purpose

The purpose is to measure the relative quality of the heuristics, and to compare the absolute size of the results to f to see how much reduction we can expect. The experiments are not intended to measure the impact of minimization on applications using the heuristics—this we leave for future research.

4.1.1 Overview

We tested the heuristics on the problem of checking equivalence between two FSMs. Specifically, the SIS [9] command `verify_fsm -m product` checks equivalence using the approach described in [10], and makes heavy use of BDD minimization. In this application, minimization on a function $[f, c]$ is currently performed using *constrain*. For the experiments, we intercept each call to *constrain*, apply all heuristics to $[f, c]$, measuring their runtimes and resulting sizes, and then return the result of *constrain* to `verify_fsm`. Actually, some of the calls to *constrain* assume the special property of *constrain* mentioned in Section 1, so it would be incorrect to return any cover of $[f, c]$. However, since the impact of minimization on the application is not being measured, each call can be treated as an instance of EBM.

Measuring runtimes is a delicate issue since the BDD package caches the results of earlier computations. Thus, when two heuristics make similar transformations on a particular example, the second heuristic can take advantage of the cached computations from the first, leading to reduced runtime. To avoid this, we invoke the BDD garbage collector, before each heuristic is called, to flush the caches of computations from earlier heuristics.

Theorem 8 can be exploited to calculate a lower bound on the size of a minimum solution to an instance $[f, c]$ of EBM. Let p be a cube of c. *Constrain* finds a minimum solution to the instance $[f, p]$, which we denote by $\hat{f}_p$. Since $f \cdot p \subseteq f \cdot c$ and $f + \overline{c} \subseteq f + \overline{p}$, then $\hat{f}_p$ is at least as small as any cover of $[f, c]$. By applying *constrain* on many such cubes p and noting the largest size seen, a lower bound can be obtained to measure the absolute quality of the heuristics.

Cubes of c can be generated by traversing its BDD in a depth-first order, returning a cube each time the constant 1 is reached. A large number of cubes may be found this way, so the lower bound computation is limited to the first 1000 cubes. Another approach would be to look for large cubes (ones with few literals) by finding short paths from the root of c to the constant 1.

It is worth noting that if $\hat{f}_p$ is a cover of $[f, c]$, then it *is* a minimum solution. This suggests another heuristic: apply *constrain* to each cube in a subset of cubes of c; if a cover is found, then stop; if the set is exhausted, then apply some other heuristic. This approach has an obvious limitation: the function ($d0$ $1d$) shows that there does not always exist a cube p of c such that $\hat{f}_p$ is a cover of $[f, c]$ (with either cube of c, $\hat{f}_p$ is a constant function).

To some extent, the degree of minimization possible is correlated inversely with the size of the onset of c. Indeed, when $c = 1$, no minimization is possible, and when $c = 0$, a solution of size one exists. However, between these extremes, this correlation may be weak. For example, if f is already a minimum cover of $[f, c]$, then regardless of the size of the onset of c, no minimization is possible. On the other hand, if $c \subseteq f$, then regardless of the size of the onset of c, a result with one node exists. Nonetheless, it is insightful to analyze our data based on the size of the onset of c. The number we compute for this, *c_onset_size*, is the percentage of the number of onset points in c to the size of the Boolean space over the union of the variable supports of f and c (i.e. $sup(f) \cup sup(c)$).

4.1.2 Detailed Description

In addition to the nine heuristics mentioned in Section 3 (eight sibling-match heuristics and one level-match heuristic), we tested four other "heuristics". Three of them are *f_and_c* and *f_or_nc* (which just compute the bounds $f \cdot c$ and $f + \overline{c}$), and *f_orig* (which is simply f itself). The fourth is *min*, which is the best result found over all the heuristics; all comparisons are made relative to *min*.

We ran `verify_fsm`, comparing a machine to itself, on the following benchmarks: s344, s386, s510, s641, s820, s953, s1238, s1488, scf, styr, tbk, mult16b, cbp.32.4, minmax5, and tlc. We aggregate the data over all the benchmarks to better understand the average performance of the heuristics (since there always exist an instance where one heuristic will perform better than another, it does not make sense to compare individual instances). We filtered out all calls where c is a cube or where c is contained in f or $\overline{f}$, since most heuristics find a minimum in these cases.[3]

We accumulate the results of the heuristics into different buckets, depending on the size of f. For each order of magnitude, we have a different bucket (i.e. 0-9, 10-99, etc.). Furthermore, within each of these buckets, we subdivide the data based on *c_onset_size* into three sub-buckets: $< 5\%$, 5%-95%, $> 95\%$. For our experiments using `verify_fsm`, we had no entries in the 5%-95% sub-buckets. We plan to investigate if this is inherent in checking for machine equivalence.

	All calls (2704)				< 5 % calls (2532)				> 95 % calls (172)			
Heur. Name	Total Size	% of *min*	Run-time	Rank	Total Size	% of *min*	Run-time	Rank	Total Size	% of *min*	Run-time	Rank
low_bd	17260	29	63K	-	13188	51	62K	-	4072	12	780	-
min	60415	100	0	-	25645	100	0	-	34770	100	0	-
osm_bt	65067	108	292	1	27261	106	254	1	37806	109	38	2
tsm_cp	66563	110	2059	2	28757	112	2017	5	37806	109	42	2
osm_nv	67198	111	308	3	27319	106	276	2	39879	115	32	5
restr	67707	112	239	4	27828	108	229	3	39879	115	10	5
tsm_td	68524	113	2134	5	28645	112	2099	4	39879	115	35	5
opt_lv	92101	152	5940	6	57331	224	3654	6	34770	100	2285	1
osm_cp	112430	186	949	7	74624	291	908	7	37806	109	40	2
const	114503	189	1077	8	74624	291	1067	7	39879	115	10	5
osm_td	114503	189	936	8	74624	291	902	7	39879	115	34	5
f_orig	479514	794	0	10	422735	1648	0	10	56779	163	0	12
f_and_c	2034640	3368	637	11	1994761	7778	631	11	39879	115	6	10
f_or_nc	2051088	3395	638	12	1994761	7778	633	12	56327	162	5	11

Table 3: Totals over all examples; over examples where *c_onset_size* $< 5\%$; and over examples where *c_onset_size* $> 95\%$.

[3]The heuristics *opt_lv*, *f_and_c* and *f_or_nc* are not guaranteed to find the minimum when c is a cube.

4.2 Discussion

Table 3 lists the primary set of results of our experiments. The first column lists the heuristic names[4] sorted in order of column two. For each heuristic, column 2 gives the cumulative sizes of the results over all calls (2704 calls). Column 3 gives the percentage of the corresponding entry in column 2 to the total size for *min* given in the second row of column 2. Column 4 gives the cumulative runtimes in seconds on a DECstation 5000/125, with 32 megabytes of physical memory. Finally, column 5 gives the rank order of the heuristics based on the cumulative sizes of column 2. The second set of columns gives the same data over all calls where *c_onset_size* < 5% (2532 calls), and the third set where *c_onset_size* > 95% (172 calls).

First, some general remarks. The *f_and_c* and *f_or_nc* heuristics perform badly and will not be discussed further. The lower bound computation shows that over all the calls, our *min* is only 3.4 times greater than the lower bound. It is not known how tight this bound is. However, there is reason to believe it can be increased by examining more cubes, and bigger cubes. In particular, when we increased the limit of cubes enumerated from 10 to 1000, this percentage increased from 24 to 29.

Over all the calls, we see a sizable reduction in the size of f: roughly a factor of 8, from 480K nodes to just 60K (for *min*). The reduction is understandably much greater when *c_onset_size* is small, and hence when there is more room for optimization (in this case a factor of 16, while only a factor of 2 for large onsets). The reduction observed suggests that BDD minimization can be expected to have a considerable impact on the performance of applications employing minimization.

The runtimes can be interpreted as follows. The heuristic *opt_lv* is easily the most costly. The current implementation requires the BDD to be traversed from the root to level i each time i is changed. It may be possible to avoid this cost. Among the heuristics which match siblings, the runtime is determined by the complexity of the matching test and of the computation to compute the common i-cover. The *tsm* heuristics are the most complex in both regards.

The data over all calls in the first set of columns is dominated by the instances where *c_onset_size* < 5%, and hence the first two set of columns are qualitatively the same. Hence, we focus on analyzing the last two sets of columns.

When *c_onset_size* < 5%, there is much freedom for optimization - in some sense, too much. Matches are easily found; the difficulty is in determining which matches to make. For *opt_lv*, the conjecture is that it cannot distinguish the good matches from the bad matches, and hence performs poorly. It is possible that the optimizations for constructing cliques suggested in Section 3.3.2 will alleviate this. Among the sibling-match heuristics, those that have no-new-vars turned off seem to make matches that unnecessarily introduce new variables, thus limiting the scope for reduction. Hence, we observe that the heuristics with no-new-vars turned on take the top five spots. Those with no-new-vars turned off take the bottom three spots (disregarding *f_and_c*, *f_or_nc*, and *f_orig*).

When *c_onset_size* > 95%, the scenario is much different. Now matchings are hard to find, and any extra effort made to find matchings is rewarded. Since *opt_lv* takes a global approach to finding

[4] *restr* is *restrict*, and *const* is *constrain*.

Heur.	f_orig	const	restr	osm_bt	tsm_td	opt_lv
f_orig	0.0	26.9	0.6	0.4	5.3	37.8
const	57.5	0.0	5.4	4.7	15.2	58.4
restr	62.5	32.7	0.0	0.3	17.2	64.5
osm_bt	62.5	33.9	4.3	0.0	18.2	65.0
tsm_td	72.9	39.1	20.6	18.9	0.0	64.7
opt_lv	46.3	26.1	11.0	10.3	2.6	0.0
min	74.3	41.9	24.2	21.9	22.4	66.4

Table 4: Head-to-head comparisons, over all examples.

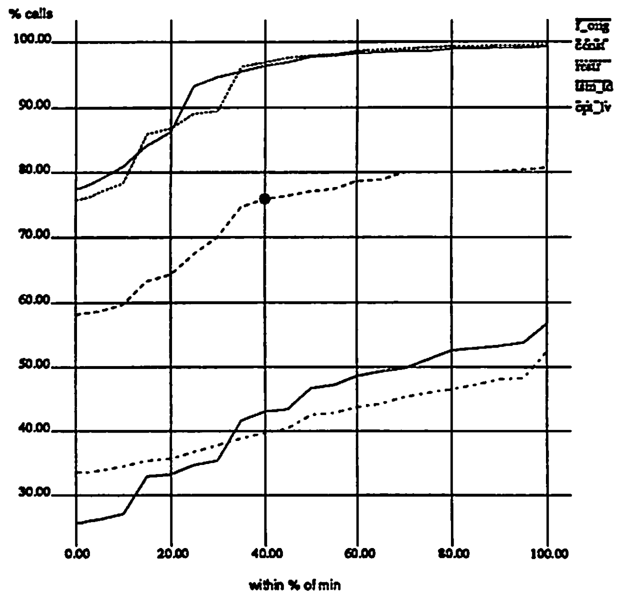

Figure 3: Plot showing what percentage of calls to a heuristics are within which percentage of the heuristic *min*.

matchings, it is not surprising that it is *never* out-performed in this category. The sibling-match heuristics fall into 2 categories: those that check for complement matches, and those that do not. Since checking for complement matches increases the likelihood of finding a match, the category that checks for complement matches performs about 6% better than the category which does not.

Considering all three sets of data, the matching criterion does not seem to have much effect on the results. It is possible that the f and c functions are generally such that when one type of match can be made, then usually all three types (*osdm*, *osm*, and *tsm*) of matches can be made. This would explain the similarity in results.

The above analysis tracks well with the size of f, and hence we do not show the data broken down by the size of f. In all the calls, $|f|$ is less than 10K.

Another way of analyzing the data is provided by Table 4. In this table, entry (i, j) gives the percentage over all calls in which heuristic i finds a strictly smaller result that heuristic j. We show only a representative subset of the heuristics. For example, (1,2) tells us that *constrain* increased the size of f 26.9% of the time. Column 6 tells us that *opt_lv* is routinely bettered by other heuristics. However, this data is dominated by the case when *c_onset_size* < 5%; in the corresponding table for *c_onset_size* > 95%, this column is all zeroes, which means that it is always the best. Entry (7,4) tells us that *min* bettered *osm_bt* only 21.9% of the time. Another way of saying this is that *osm_bt* was the smallest among all the heuristics 78.1% of the time.

This table reveals a few more pieces of information. The sum of entries (i,j) and (j,i) tell us how much "orthogonality" there is between heuristics i and j: the greater the sum, the more the orthogonality. For example, the sum for *constr* and *tsm_td* is 54.3%. Also, note that *tsm_td* betters *osm_bt* slightly more often than the converse case, even though *osm_bt* was the best overall. Finally, the row for *low_bd* (although not shown here) tells us that all of the heuristics in Table 4 (except for *f_orig*) achieve the lower bound 26.2% of the time.

A final method of analyzing the data is provided in Figure 3. Data is shown for five representative heuristics, which in ascending order of y-intercept are: *f_orig*, *opt_lv*, *constr*, *restr*, and *tsm_td*. The data point highlighted by the black dot is "read" as follows: on 76% of all the calls to *constrain*, *constrain* was within 40% of the smallest result found. This gives a measure of how robust a heuristic is: if a curve is high in the graph, then even when a heuristic does not find the smallest result, it is not too far off. By definition, all the curves increase monotonically toward 100%. The y-intercept of a curve indicates how often a heuristic finds the smallest result. We see that the classes represented by *restr* and *tsm_td* consistently perform about 20% better than *constrain* in this respect. Over all the data, *opt_lv* performs poorly; again, however, in the corresponding graph for *c_onset_size* > 95%, the curve for *opt_lv* is pegged at 100%.

Overall, *osm_bt* is preferred, since it combines good minimization with small runtimes. The *restr* heuristic is a close competitor.

It seems clear that a heuristic which combines the strong points of the level-match and sibling-match heuristics would be robust and would yield good results. In particular, we would like such a heuristic to consider many functions for possible matching, but favor matchings of nearby functions. The proposals that we have put forth in the body regarding scheduling and building cliques are a step in this direction.

5 Conclusion

In this paper, we presented a general framework for heuristic solutions to the BDD minimization problem, which is an important problem having many applications. In particular, we defined three matching criteria of differing levels of strength. We give two methods for choosing functions as matching candidates: siblings and functions at level i. We defined the general function matching problem and described exact solutions to the problem for the three matching criteria.

We proved that the sibling-match heuristics are optimal when c is a cube. Based on this, we formulated a technique to compute a lower bound on the size of the result. Also, we proved that applying minimization at level i using the *osm* match is optimal with respect to the number of nodes below level i.

Finally, a thorough set of experiments was done to characterize the relative power of the heuristics, and their absolute power in minimizing a function. For the FSM equivalence application on a standard set of benchmarks, on average, we were able to find a cover one-eighth the size of the original input. Also, we observed a distinct difference in the heuristics based on the size of the onset of the care function: when it is small, those heuristics that avoid introducing new variables work best; when it is large, those heuristics which examine many possible matches work best. We suggest combining the merits of both of these classes of heuristics to achieve a robust heuristic that finds small covers. The best heuristic overall is *osm_bt*, with *restrict* a close second.

Acknowledgements

We wish to thank Adnan Aziz, Sovarong Leang, Jean Christophe Madre, and Rajeev Murgai for their helpful comments. This work was supported by SRC grant 93-DC-008, and by an equipment grant from Digital Equipment Corporation. In addition, the first author was supported by an SRC Fellowship.

References

[1] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automat. Conf.*, pages 40–45, June 1990.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[3] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines using boolean functional vectors. In *Proceedings of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, November 1989.

[4] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[6] L. G. Heller, W. R. Griffin, J. W. Davis, and N. G. Thoma. Cascode voltage switch logic - A differential logic family. In *31st Int. Solid State Circuits Conference*, pages 16–17, 1984.

[7] Jean Christophe Madre. Private communication, 1992.

[8] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic Synthesis for Programmable Gate Arrays. In *Proc. Design Automat. Conf.*, pages 620–625, 1990.

[9] Ellen M. Sentovich, Kanwar Jit Singh, Cho Moon, Hamid Savoj, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proc. Int'l Conf. on Computer Design*, October 1992.

[10] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 130–133, November 1990.